Compilers for Parallel Computers

edited by

M. Gerndt
Proceedings of the 6. Workshop
Compilers for Parallel Computers

Manuscripts of the papers presented at the 6. Workshop held in Aachen, Germany, from 11 – 13 December 1966

edited by
Michael Gerndt
Preface

Parallel computers are powerful tools for scientific and industrial research. To allow easier and portable program development for parallel systems, High Performance Fortran and extensions of C have been developed in the past few years. But, although work on implementing these languages has been going on for some time, compiler developers still face a lot of challenges. Moreover, the languages have to be extended to support a broader spectrum of applications. Since research in parallel computing is focused to a large extend on scientific applications, architectures, programming models, languages, and tools for non-scientific applications will require much more attention in the future.

This workshop is the sixth of a series of workshops which were held at Oxford, Paris, Vienna, Delft, and Malaga. The goal of these workshops is to bring together experts in the field of compiling for parallel computers to discuss the state-of-the-art and to exchange ideas for future developments.

This workshop has participants from nine countries. The thirty-three presentations are covering compilation techniques for High Performance Fortran, new language constructs for unstructured and multidisciplinary applications, language and tool support for shared virtual memory, program analysis and optimization techniques, parallel models, and loop transformation techniques based on a strong theoretical framework.

We would like to thank the board of directors of the Research Centre Jülich for sponsoring the workshop and Prof. Hoßfeld, the director of the Central Institute of Applied Mathematics, for accepting the invitation to arrange the workshop in Aachen and for giving us any support we needed in the local organization. Without the experience and the extensive support of the staff of the Central Institute of Applied Mathematics and of the Conference Service of the Research Centre Jülich during the preparation of the workshop, the organization would have been a full-time job for us.

The Program and Organizing Committee:

*Michael Gerndt (Research Centre Jülich)*
*Thomas Brandes (German Research Centre for Information Technology)*
*Andreas Krumme (Research Centre Jülich)*
Contents

Compilation Systems

W. Aßmann
An Universal Usable Frontend Processor for Fortran 90 and HPF ......................................................... 3

J. Borowiec, Th. Ernst
Transforming Data-parallel Fortran90/HPF Constructs Into a Uniform Internal
Representation.......................................................................................................................... 11

C. Ancourt, F. Coelho, B. Creusillet, R. Keryell
How to Add a New Phase in PIPS: the Case of Dead Code Elimination ........................................... 19

A. Persson, J. Ringström, P. Fritzson
A Case Study of Code Generator Generation for Embedded SIMD Computers ...................................... 31

Parallel Models

A.J.C. van Gemund
Compile-time Optimization and the SPC Parallel Programming Model ................................................. 45

D.K. Arvind, T.A. Lewis
Distributed Recursive Datatypes ................................................................................................. 57

P.F.G. Decherin, L.C. Breebaart, F. Kuijiman, C. van Rieuwijk, H.J. Sips
A Sound and Simple Semantics of the forall Statement within the
V-nus Compiler Framework...................................................................................................... 59

HPF Code Optimizations

T. Brandes, F. Desprez
Pipelining Data Parallel Computations in an HPF Compiler .................................................................. 75

S. Dutta, J. Ramanujam
Data Structures for Efficient Execution of Programs with Block-Cyclic Distributions ........................ 91

Multi-Level Parallelism

T. Rauber, G. Rünger
Compiler Support for the Use of Two-Level Parallelism.................................................................. 95

P. Launay, J.L. Pazat
Integration of Control and Data Parallelism in an Object Oriented Language ................................. 107

Shared Virtual Memory

M.F.P. O’Boyle, A.P. Nisbet, R.W. Ford
A Compiler Algorithm to Reduce Invalidation Latency in
Virtual Shared Memory Systems............................................................................................. 115

M. Gerndt, A. Krumme
Automatic Performance Analysis for Shared Virtual Memory Systems .............................................. 125
F. André, Y. Mahéo
Cidre: Programming with Distributed Shared Arrays ................................................................. 139

Analysis of Access Patterns

J. Touriño, R. Doallo, R. Asenjo, O. Plata, E.L. Zapata
Analyzing Data Structures for Parallel Sparse Direct Solvers: Pivoting and Fill-In .................. 151

E. Laure, B. Chapman
A Refined Method for Alignment Analysis Combining Inter- and Intradimensional Alignment Preferences ................................................................. 169

A. Cohen, J.F. Collard, M. Griebl
Data Flow Analysis of Recursive Structures .................................................................................. 181

New Language Constructs

B. Chapman, P. Mehrotra, H. Zima
HPF+: New Language and Implementation Mechanisms for the Support of Advanced Irregular Applications ........................................................................ 195

M.J. Voss, I. Park, R. Eigenmann
On the Machine-Independent Target Language for Parallelizing Compilers .......................... 207

C.W. Keßler, H. Seidl
Language Support for Synchronous Parallel Critical Sections ..................................................... 219

Program Optimization

A.W. Lim, M.S. Lam
Finding Affine Partitions that Maximize Parallelism and Minimize Synchronization .................. 233

C. Eisenbeis, A. Sawaya
Optimal Loop Parallelization under Register Constraints ................................................................. 245

E. Rohou, F. Bodin, A. Seznec
SALTO: System for Assembly-Language Transformation and Optimization ............................. 261

Data and Loop Transformations

A. Darte, F. Vivien
Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs ......................................................................................... 275

M.F.P. O’Boyle, P.M.W. Knijnenburg
Non-Singular Data Transformations: Definition, Validity and Applications ............................. 287

A.J.C. Bik, P.M.W. Knijnenburg
Reshaping Access Patterns for Improving Data Locality ................................................................. 299

M. Griebl, C. Lengauer
The Loop Parallelizer LooPo ........................................................................................................... 311
HPF Implementation Techniques

M. Kandemir, R. Bordawekar, A. Choudhary, J. Ramanujam
A Unified Tiling Approach for Out-Of-Core Computations ......................................................... 323

S.P. Midkiff
Computing the Local Iteration Set of a Block-cyclically Distributed Reference
with Affine Subscripts .................................................................................................................. 335

L.R. Mullin, D. Eggleston, L.J. Woodrum, W. Rennie
The FGI-PSI Project: Preprocessing Optimizations for Existing and New F90 Intrinsics
in HPF using Compositional Symmetric Indexing of the ψ Calculus ........................................ 345

Compile-Time and Run-Time Support

W. Ogata, A. Yoshida, M. Okamoto, K. Kimura, H. Kasahara
Near Fine Grain Parallel Processing without Explicit Synchronization
on a Multiprocessor System ........................................................................................................ 359

W. Amme, E. Zehender
Data Dependence Analysis in Programs with Pointers ............................................................... 371

G. Edjlali, A. Sussman, J. Saltz
Interoperability of Data Parallel Runtime Libraries with Meta-Chaos .................................... 383
Compilation Systems
An Universal Usable Frontend Processor for Fortran 90 and HPF

WERNER ABSTANN

German National Center for Information Technology (GMD)
Research Institute for Computer Architecture and Software Technology (FIRST)
Rudower Chaussee 5, D - 12489 Berlin, Germany
(email: assmann@first.gmd.de)

ABSTRACT

Numerical applications are mostly written in FORTRAN until now. Especially caused by the evolution of FORTRAN to FORTRAN 90, FORTRAN 95 and later on to FORTRAN 2000 on the one side and to HPF-1 and HPF-2 for the description of parallel processing features on the other side the development of compilers and tools for the processing of FORTRAN programs is of increasing importance. The effort for such tools is rather high particularly for complete frontends which are able to check the programs for standard conformance. The necessity of compatibility to older FORTRAN versions complicates the situation.

Under these assumptions multiple usable (and more or less free available) frontends have high advantages. Tool developers can start their work on a very high level - usually some high level internal representation - and can use already existing tool sets for the processing of the internal representation.

This paper describes such a frontend for Fortran 90 and HPF-1 and the used implementation techniques as well as some application results.

INTRODUCTION

FORTRAN is until now THE language for numerical applications mainly caused by the fact that this programming language exists for tens of years and huge lots of software are still active inside of big applications. On the other side the language was relatively stable in the past time - FORTRAN 77 was the last standardized version, and the development of compilers and other tools was restricted to the adaptation to new machines and operating systems, internal optimizations, and vendor-specific extensions.

In the last years this situation has been changed dramatically. The new language standard for Fortran 90 appeared containing some important features of modern programming languages destined to improve program development in FORTRAN. The development of new compilers and other tools was necessary. But the newly included modern language facilities such as module handling and operator overloading render the implementation of FORTRAN processors more difficult - together with the old problems such as line structure, keyword use and some syntactically hard to decide language constructs. To prove the standard conformity of FORTRAN programs lots of constraints (to some degree under revision) must be inspected additionally.

On the other side the needs of High Performance Computing had required extensions for the description of parallel processing features - HPF-1 was defined as a first step requiring

1. Parts of this frontend were implemented together with P. Enskonatus (GMD/FIRST)
sophisticated algorithms for the transformation of HPF constructs on the usual language level. All together, high effort is necessary to build a complete Fortran 90/HPF frontend.

Also under the evolution aspect of Fortran 90 to Fortran 95 and later on to Fortran 2000 on the one side, and of HPF-1 to HPF-2 on the other side, it’s not more possible to build a complete frontend for these language versions on the hurry. A multiple usable frontend has therefore high importance, and could heavily accelerate research projects on the field of HPC - the traditional application area of Fortran.

**THE INTERNAL REPRESENTATION**

The design of the Internal Representation (IR) was made under the supposition that the reconstruction of the source program should be possible, of course in some pretty-printed format under neglect of stylistic elements such as comments and so on. Another design criterion was the selection of compiler construction tools. These tools work usually on attributed trees (in the true sense: on attributed graphs). As result an IR was defined relatively near to the original syntax of Fortran 90 for executable statements but only with generalized declaration statements giving the symbol table administration an important role in the internal representation of all available information.

As a first example the IR for the simple binary expression $a + 1$ will be shown to demonstrate its structure:

```
binary_expr
    type    shape    value    opcode    left    right
```

```
simple-variable
    type    shape    value    pos    symbol
```

```
integer-const
    type    shape    [value]    pos    val
```

Expression nodes have generally ‘type’, ‘shape’, and ‘value’ information (‘value’ is a reference to a constant node if the expression is computable during compile-time, for example by constant folding), leaves of the tree have a position information (file, line and column of first and last character in the source), and the other attributes are more or less different for the diverse nodes. The ‘val’ attribute in the ‘integer-const’ node contains the binary representation of the constant ‘1’. The ‘symbol’ attribute references to the symbol table entry for ‘a’ so it’s possible to find the text of the identifier and its declaration:

```
symbol-table-entry
    id-text    visible    scope    def
```

```
variable-definition
    symbol    pos    prop-s    type    shape    init    hpf-attr
```

The attribute ‘scope’ of symbol table entries addresses the scope of local variables resp. other attached visibility areas such as the scope of components in case of derived types. With the help of some control information in the attribute ‘visible’ the visibility of entries may be switched off or on especially according to special needs in the handling of modules,
and the attribute 'prop-s' (in all definition nodes) is a set of Boolean attributes, for example PARAMETER, POINTER, SAVE, and OPTIONAL.

As an last example the internal representation of a statement node is given:

```
if-stmt
    next-st | label | pos | name | if-blocks
```

All together about two hundred different nodes are used (together with so-called NO-nodes to represent absent attribute values resp. end of lists). By using such a relatively big set of nodes it's possible to allow the reconstruction of the source text without a substantial loss of information.

Of course it's necessary to use compiler construction tools to handle such data structures effectively. The definition of the IR was done with the help of 'ast' - one of the tools in the Cocktail Toolbox [2]:

```
Statement = next-st : Statement
            | label : Label
            | pos : Pos
<
assign = lhs : Variable
         | rhs : Expression
if-stmt = name : IDENT
         | if-blocks : IF-Blocks
```

After the processing of these definitions by 'ast' a set of operations on nodes is available, for example for the creation of nodes and the output of trees.

**BUILDING THE INTERNAL REPRESENTATION**

The processing of FORTRAN texts raises some problems because of the lexical features of FORTRAN but also because of special declaration issues: some declarations must not be given explicitly and will be replaced by default assumptions, other declarations may occur in the text behind their first use. It's therefore nearly necessary to use several (logical) passes in the frontend:

- Source text processing
- Lexical analysis
- Syntactical analysis
- Declaration analysis
- Object identification
- Constraint checking

These logical passes are not always strongly separated in the implementation because of effectiveness. General principle in the design of these passes is their autonomous work as far as possible. Especially backward connections from the syntactic analysis to the lexical passes should be avoided.
Source Text Processing

This module transforms the original FORTRAN source text consisting of a sequence of physical lines into a sequence of logical lines. During this processing insignificant characters such as blanks, comments, and continuation line information are removed. The newly built logical line is therefore processable by usual scanners. Additionally, an second data structure is created holding for every character in the logical line its original source text position (file, line, column).

But this module takes over another task. The well known example

```fortran
  DO 10 I = 1,2
  DO 10 I = 1,2
```

demonstrates the problem: because of the insignificance of blanks (in fixed source format) the lexical analysis doesn’t know at the beginning of the line whether to reduce to the terminal symbol ‘DO’ or to the identifier ‘DO10I’. With other words it’s necessary to make a decision if the current (logical) line (always a complete statement) is an assignment statement or some other statement (only assignment statements don’t start with a keyword!).

When creating the logical line every character of the line must be touched. It’s obvious to use this opportunity to make the decision by using some trivial heuristics on the fly and (nearly) without additional efforts:

- An assignment statement must contain a ‘:=’ character (on parentheses level 0 and outside of strings).
- A statement containing ‘::’ can’t be an assignment statement.
- A statement containing a ‘,’ character can’t be an assignment statement.

Unfortunately a few problems remain but only for rather seldom used statement types, which must be resolved by a special lookup procedure at the start of the lexical analysis for such logical lines for which this decision was not yet made. Of course, this method improves only the processing speed but with considerable effect.

The source text processing module is implemented as a hand-written instance of a regular automaton with states for every significant physical line column (column 1, 2, 3, 4, 5, 6, 7-72, >= 73 in fixed source form). It consists mainly of one big switch statement inside a loop to decide for every character what to do and how to update the state. A few extra state variables are necessary to extend the states, for example to count the parentheses level:

```fortran
  for (;;) {
    next: ch = getc(file);
    switch (State, CharacterClass(ch)) {
      case 7, '\n': Print('\n');
        ParLevel++;
        goto next; /* stay in current state */
      ...
    }
  }
```

Lexical Analysis

The lexical analysis produces a sequence of lexical items consisting of the terminal code, position information, and sometimes (for pseudo-terminals) its textual representation. Special care is necessary to handle correct keywords which can be used under some contexts also as identifiers. In principle the scanner is composed of several subscanners: a scanner
for the leading keywords of statements, another scanner for the handling of tokens inside a statement, a scanner for FORMAT statements, and a few other small scanners for special tasks.

The switching between these subscanners and (partly) the work of the scanners itself is controlled by the main scanner together with state variables. The following example demonstrates their use (see on the right hand side some of the scanner actions associated with the source text shown on the left hand side):

```plaintext
INTERFACE
MODULE PROCEDURE p
    (ProcedureAllowed = true;)
    if (ProcedureAllowed)
        return MODULE, PROCEDURE, ID(P);
    else
        return MODULE, ID(PROCEDURE);
    }

END INTERFACE
```

In this way (after all some restricted syntax knowledge) it's possible to avoid retrospective actions from the syntactic analysis to the lexical analysis to make the life easier for the correct placement of semantic actions in the parser according to natural bindings.

For the implementation of the scanner also a tool of the Cocktail Toolbox was used. The scanner generator 'rex' has some useful features at hand to support the concept of subscanners (by 'start states' and the handling of 'right context'). Additionally, in cooperation with the error recovery mechanism of the syntactic analysis also inserted pseudo-terminals can be handled correctly.

**Syntactic Analysis**

The parser is generated by the 'lalr' tool of Cocktail. The syntactic rules are decorated by semantic actions - mainly tree building and symbol table operations. Because of some FORTRAN features, it's not possible in all cases to decide on the syntactic level and at parsing time which of a few alternative meanings is correct. For example, to distinguish between a function call and an array element some semantic analysis is needed. Therefore, the grammar must be simplified to avoid conflicts. Sometimes another parser with such features as semantic predicates to control the reduction operations in dependence of semantic properties would have advantages against the LALR(1) based parser used here.

The following example gives an impression of the set of rules necessary for the description of the FORTRAN syntax and the decoration with semantic operations:

```plaintext
main_program : 'PROGRAM' program_name ';;' main_program_body
    {SetPos($2.Tree->program.pos,$1,$4);
     $2.Tree->program.body = $4.Tree;
     CheckEndName($2.Tree,$4.Scan);
     $5.Tree = $2.Tree;
    }.

program_name : IDENTIFIER
    {
     $$.Tree = mprogram{...};
     $$.Tree->program.symbol = EnterIdentifier($1);
    }.
```
Declaration Analysis

The first step to build the IR is the analysis of declaration statements resulting in the creation of all declarative nodes. This can be done completely during parsing together with the necessary symbol table operations (including scoping) using the algorithms described in [1]. Some additional data structures must be built on the fly for example the list of types to be used for implicit type declarations, the list of overloaded intrinsic operations, and the list of labels for example - some of them scope dependent.

After the syntactic analysis is finished the declarative nodes are built but not yet completely checked because of the fact that some information appears later in the source text as needed for correct processing (for example: an IMPLICIT statement follows the SUBROUTINE or FUNCTION statement but must be considered in the typing of dummy arguments and the function results). A first traversal of the IR to compute not yet known attribute values is necessary. Attribute evaluation and tree traversal are done with the help of the pattern-matcher 'puma' [3].

Object Identification

Executable statements can be handled only preliminary at parsing time because the objects used in expressions are not yet known exactly (the declaration analysis must be finished before!). Therefore statement and expression nodes are built during parsing but representing to this moment only the syntactical structure without consideration of its true meaning.

After finishing syntactic and declaration analysis another tree traversal (depth first strategy) is necessary to associate every used object exactly with its declaration, for example calls of internal procedures. Furthermore, the resolution of overloaded objects (intrinsic operations and procedure calls) is performed by sophisticated argument analysis considering also optional and keyword arguments. During this pass tree transformations are necessary to build the final IR. HPF directives are represented only as some abstract syntax and are processed later.

Constraint Checking

As a last but nevertheless very important task the semantic correctness in the sense of fulfilling all the constraints defined in the standard must be checked. This can be done partly during the two preceding passes but a few problems must be solved in a third tree traversal. After this pass the IR is complete and ready for further processing by following processes.

MODULE HANDLING

The introduction of the module concept is one of the most important features in Fortran 90 to improve the software quality. But the pleasure is not complete because the implementation raises some problems. Especially the missing separation of the public visible part and the implementation specific details complicates the application and also the implementation in the compiler frontend (MODULA-2 uses separate definition and implementation modules, C uses normally extra header files).

One way to process modules is the generation of special files during the processing of modules containing only the information needed in importing modules and to read these files when they are needed. Besides the necessity of the additional administration of these files validity problems can occur if the modules must be updated.
The other solution is to import directly the complete module text and to extract only the needed information. This way is known from advanced MODULA-2 compilers and proved as very effective. The additional effort of processing unneeded parts in the frontend is not so very important because of the very high speed of the frontend.

This frontend goes the second way. If the frontend has to process an USE statement the first action is to look if the addressed module is already available. If this test fails the source text of the module is searched using some strategies, scanner and parser are interrupted and set to a new state allowing the analysis of the module text, the module is processed, and after all the original state of scanner and parser are restored for normal continuation. This process may be recursive if the imported module contains references to other modules.

The administration of the symbol table is a little bit more troublesome to secure the correct object identification. This problem is solved by additional scoping sometimes resulting in various symbol table entries for the same object (especially for renamed objects) all of them pointing to only one declaration node, and by controlling the visibility of entries using its ‘visible’ attribute.

APPLICATIONS

Besides the use of the frontend in a complete HPF compiler a lot of other applications is possible. The generated IR can be used to apply sophisticated parallelization algorithms resulting in an modified tree. This tree can be processed to get executable programs as input for an usual compiler but also the transformation to FORTRAN (either FORTRAN 77 or Fortran 90) or C is an usual method (special care is needed for the handling of crucial constructs such as entries and internal procedures for the transformation to a language other than Fortran 90).

Another class of applications is the development of simple tools to produce symbol table listings, cross-reference listings, call graphs, and last but not least pretty-printers. All of these tools can be produced with small effort by applying pattern-matching algorithms on the IR generated by this frontend. In this way it’s possible to improve the software production process considerably.

Currently the frontend is used in three main applications: PREPARE HPF compiler [5], Vienna Fortran Compilation System+ (a new version of [6]), and SNAP [4].

REFERENCES

Transforming Data-parallel Fortran90/HPF Constructs Into a Uniform Internal Representation*

Jan Borowiec, Thilo Ernst

Abstract

As a part of the PREPARE High Performance Fortran compiler, a component has been designed and implemented which transforms an abstract, high-level internal representation of Fortran90/HPF programs to a medium-level internal form. Normalizations and pre-optimizations are applied to simplify the task of the subsequent compiler phases. This paper describes a selection of translation techniques for data-parallel statements which were developed in this context.

1 Background

This paper focuses on one specific component of the PREPARE HPF compiler. The context (other compiler components, data structures, the CoSy compiler generation environment) cannot be described here see [1], [2], [3], [4], [5], [6], [7] for more information. To understand the methods presented here it is most important to know that the PREPARE compiler, unlike many other HPF systems, is not a source-to-source translator but a real compiler: it translates the source program to an internal representation which is then subject to several complex transformations; at the end, native SPMD code is generated for the target architecture.

The role of the HPT (HIR-to-PIR-Transformer) compiler component is to transform a high-level internal representation (HIR) of the Fortran90/HPF program being compiled into a normalized medium level internal representation (PIR, Prepare Internal Representation) the remaining compiler components will be operating on.

Input to the HPT is the high-level internal representation (HIR) of a Fortran90/HPF program as generated by the GMD Fortran90/HPF front-end (see [8], where the HIR is simply called "IR"). The HPT component performs its job while recursively descending the input HIR tree. Thus saying that, e.g. a FORALL construct is being transformed means that a list of statements is analysed and the current one is a FORALL.

HPT’s output is a PIR program unit which contains, amongst others, a list of types, a list of procedure units, a list ofglobals declarations, etc. A procedure unit, amongst others, contains: a list of locals declarations and a body which is a list of basic blocks. In turn, a basic block entails a list of statements. Thus, stating below that HPT generates a statement, we mean that the corresponding PIR subtree is being created and appended to the statement list of the currently created basic block of the currently created procedure, etc. The PIR is an extension of CCMIR, the central IR of CoSy, to accomodate the requirements of HPF and SPMD execution.

As even an overview of the PIR structure and components would burst the frames of this report, we introduce here just those of its constructs which will be needed to formulate the transformation techniques selected for this paper. All PIR constructs introduced here are very much simplified compared to their actual specification. Some additional context will be provided where necessary.

2 Data-parallel array assignments

Let two Fortran arrays A, B be declared as follows:

\[
\text{INTEGER, DIMENSION (3, 6) :: A, B}
\]

*This work was supported by the the Commission of the European Communities under Esprit Project 6516 "PREPARE".
Authors’ address: GMD-FIRST, Rudower Chaussee 5, D-12489 Berlin, Germany; e-mail: (borowiec,te)n@first.gmd.de

For a distributed memory architecture, the arrays also would be distributed, but we shall abstract from that here.
Consider the following classes of Fortran90/HPF constructs, illustrated by examples:

(1) Array assignment.

\[ A(1:2, 2:6:2) = B(1:2, 2:6:2) - 1 \]

(2) DO INDEPENDENT loop nest containing "leaf" assignments only.

\begin{verbatim}
!HPF$INDEPENDENT
DO I = 1, 2
  !HPF$INDEPENDENT
  DO J = 2, 6, 2
    A(I, J) = B(I, J) - 1
  END DO
END DO
\end{verbatim}

(3) WHERE statement/construct.

\begin{verbatim}
WHERE (A > 0) A = B - 1
\end{verbatim}

(4) FORALL statement/construct.

\begin{verbatim}
FORALL (I = 1:2, J = 2:6:2) A(I, J) = B(I, J) - 1
\end{verbatim}

We will refer to these constructs as \textit{data-parallel constructs} and process them in a uniform manner since all of them:

- serve to evaluate an expression for all elements of array \( B \) and assign the resulting values to array \( A \), or to do this for a subset of elements
- have an (explicit or implicit) \textit{iteration space} that can be described by a cartesian product of integer regular \textit{sections}, optionally restricted by \textit{masks}
- allow parallel execution, i.e., the language semantics does (intentionally) not assume a specific order of iterations/elemental assignments

The \texttt{FORALL} is the most general of them all – it is always possible to equivalently replace any of the other forms by a \texttt{FORALL}. Indeed the \texttt{FORALL} (4) is equivalent to the forms (1) and (2) \footnote{However the \texttt{FORALL} (4) can introduce overhead compared to (2) since the \texttt{INDEPENDENT DO} loop does not imply \texttt{fetch-before-store} (or implicit \texttt{right-hand-side buffer}) semantics (see next section), which the \texttt{FORALL} does}. These observations form the basis for normalizing all the data-parallel constructs listed above into a uniform internal representation, the \textit{parallel array assignment}, which will be described in the next section.

3 The PIR parallel array assignment

In its expressive capabilities, the internal representation for data-parallel Fortran90/HPF constructs proposed here is very close to the HPF \texttt{FORALL} statement. However there is an important deviation – the \texttt{fetch-before-store}, or implicit \texttt{right-hand-side buffer} semantics:

Both for standard Fortran90 array assignments and for HPF \texttt{FORALL}s, the language definition (see \cite{9}, \cite{10}) prescribes that the whole right-hand side is evaluated ("fetched") and buffered before the assignment ("store") to the left-hand side is executed. This is useful for a high-level programming language since it offers the concept of computing "new values" of an array from the previous "generation", which is found in many numerical algorithms, as a built-in language feature, so the programmer can rely on "old values" when using the left-hand side array also (in read mode) on the right-hand side.

However, an IR concept, implicit buffering is not desirable:

- It would cause considerable computational complexity to be hidden and thus out of the control of the compiler
• There are many situations where the buffering is not actually needed and thus can be optimized out.

• At some point on the way down to the code generator, the buffering would have to be made explicit anyway.

Therefore, we decided to make the buffering explicit in the PIR from the beginning, and thus to take it out of the semantics of the PIR parallel array assignment construct. As a consequence, the PIR parallel array assignment comes, compared to the HPF FORALL, with an additional semantical constraint: No element assigned to on the left-hand side may be accessed on the right-hand side. The translation techniques described in subsequent sections will make sure this constraint is never violated.

Now the parallel array assignment and its auxiliary IR constructs are described in more detail:

A regular section triplet \( T = (l : u : s) \) where \( l, u, s \) are integer expressions representing a lower bound, an upper bound, and a stride, respectively, describes the set of integer values:

\[
T = \{ i | i = l + k * s, i \leq u, k \in \mathbb{N} \}
\]

Regular section triplets are useful to describe iteration spaces of data-parallel assignments, but also index domains of arrays (where the stride equals 1). Therefore, a corresponding PIR construct regular section triplet was introduced:

\[
t(l, u, s)
\]

Here \( l, u, s \) are IR nodes representing integer expressions.

Both the FORALL (4) and the DO INDEPENDENT loop (2) use explicit iteration variables \( (I, J) \). To model the other forms with a FORALL or the (similar) PIR parallel array assignment, explicit iteration variables scanning an iteration space which is obtained from the array’s index domain need to be introduced synthetically by the compiler.

So in the PIR representation, a regular section is associated with an iteration variable, by means of the iterator construct, defined as a pair:

\[
i(i, t(l, u, s))
\]

where \( i \) is the PIR representation of an iteration variable and \( t(l, u, s) \) represents the regular section which \( i \) is intended to iterate over.

The PIR representation of source expressions is usually an attributed tree whose nodes represent operators and whose leaves are PIR constructs corresponding to single variables and constants. We won’t go into details here. Instead, to refer below to an unspecified PIR expression tree involving (among others) variables \( v_1, v_2, \ldots, v_n \), we shall simply write:

\[
t(\ldots, v_1, v_2, \ldots, v_n, \ldots)
\]

When presenting PIR statements, we shall apply a similar, general notation:

\[
s(\text{componentinfo})
\]

(Below the \( e \) and \( s \) will sometimes be subscripted to represent more specialized expressions and statements.)

Declarations will be marked only very roughly as follows:

\[
d(\ldots)
\]

In FORALL and WHERE statements, mask expressions involving arrays (and indices) can occur. The role of a mask expression is to narrow a previously defined iteration space down to the subset of those tuples for which the mask expression evaluates to true. A PIR construct named \( \text{mask} \) is introduced whose only component is a mask expression. A mask containing a mask expression involving indices \( i_{x_1}, i_{x_2}, \ldots, i_{x_n} \), will be written

\[
m(e_{\text{mask}}(\ldots, i_{x_1}, i_{x_2}, \ldots, i_{x_n}, \ldots))
\]

A PIR list

\[
I = \langle i_{m_1}, i_{m_2}, \ldots, i_{m_k} \rangle
\]

where each \( i_{m_j} \) is either a PIR iterator or a PIR mask, and the list must contain at least one iterator, is called an iteration space. An iteration space carries information about iteration variables, their regular sections and (optionally) masks, i.e., information completely equivalent to a FORALL header. Interpretation
of an iteration space is from left to right: an iterator expands the iteration space seen so far by multiplication (cartesian product); a mask narrows it down by selecting a subset.

It is sometimes useful to consider an "unfiltered" version of an iteration space $I$ by completely discarding the masks, such that it consists exclusively of PIR iterators. Subsequently this will be called the corresponding total iteration space and denoted $TOTAL(I)$.

Now the parallel array assignment statement PIR construct can be defined with the following structure:

$$ s_{pass}(lhs, rhs, ispace) $$

where $lhs$ represents an array access involving iteration variables from $ispace$, $rhs$ is a PIR expression possibly containing array accesses involving iteration variables from $ispace$, and $ispace$ is the PIR representation of an iteration space as described above.

PIR parallel array assignment constructs are further processed by other compiler components to generate efficient SPMD code. This, amongst others, entails the following steps:

- determine processor-local iteration spaces
- organize memory management for local storage of mapped arrays and compiler-generated buffers
- transform parallel array assignments that require communication
- generate target-optimized code to iterate over processor-local iteration spaces. Note that the parallel array assignment IR construct described avoids early sequentialization (lowering to loop nests), but retains parallelism right until the code generation phase. As a consequence, generation of very efficient code for superscalar and other vector-handling architectures becomes possible.

More information about the methods applied in these steps can be found in [1], [2], [3], [4], [5].

Subsequent sections demonstrate how the PIR parallel array assignment is employed in the implementation of data-parallel Fortran90/HPF constructs in the PREPARE compiler.

4 Compiler-introduced array buffers

During translation of data-parallel Fortran90/HPF statements, we sometimes employ synthetic (compiler-introduced) temporary array buffers which have the same shape and, in the distributed case, mapping$^3$ as some original array (i.e. some array occurring in the source code and accessed in the statement being translated).

A detailed discussion of how Fortran arrays, and in particular distributed ones, are processed by the PREPARE compiler is outside the scope of this paper; however we want to provide, on a general level, some information on how the compiler can organize the memory management for such buffers.

Temporary array buffers are allocated just like their original counterparts, except that allocation is always done on the heap.

Allocation code is inserted just before the location where the buffer is used for the first time. Matching deallocation code is appended at the end of all code resulting from the transformation of the current data-parallel Fortran90/HPF statement (where it is sure that the buffer won't be used anymore). That way, the life range of array buffers (which can be quite memory-consuming, depending on the original array) is kept short.

Buffer allocation just before the first use also means all shape (and, if the array is distributed, mapping) information needed will be available. For static arrays and compile-time known mappings this information is known and can be exploited even at compile-time. Otherwise, allocation code is generated that fetches the needed information from a run-time descriptor object associated to the original array under consideration.

5 General transformation strategy

The compiler has to transform the data-parallel statements permitted in Fortran90 and HPF in all their possible combinations. Consider the following nested FORALL containing a WHERE statement:

$^3$This makes sure no additional communication is needed when the buffer is written back.
FORALL [level1]
...
FORALL [level n]
  WHERE condition
  assignment
END WHERE
END FORALL
...
END FORALL

From the point of view of the innermost assignment statement, each of the surrounding statements provides a share to the iteration space: FORALLs provide index domains and (optionally) masks, whereas a WHERE construct provides an additional mask expression.

Thus, at each nesting level, information from a FORALL's header is to be processed, appended to the information obtained from the surrounding level (if any), and passed down.

Generally, a HPF FORALL construct can contain a list of statements, each of which can be an assignment statement, a WHERE construct, or a FORALL construct again.

The transformation generally works by recursively descending the tree formed by the nesting of FORALL constructs and WHERE statements while maintaining a "current iteration space" subsuming the information of the surrounding levels. Each time a leaf (assignment) is encountered, the leaf statement together with the current iteration space is transformed into one (or two, if right-hand side buffering is necessary) PIR parallel array assignments. That way, the entire FORALL construct is successively transformed into a sequence of PIR parallel array assignments.

6 Transformation of array assignment

The transformation is the same for the Fortran90 array assignment and for the exemplary assignment occurring in a FORALL – just the iteration space is constructed differently.

Input.

- \( I_{\text{inher}},t \): PIR representation of iteration space established at the surrounding FORALL and/or WHERE levels. (Empty for Fortran90 array assignment)
- \( s_{\text{assign}}(e_{\text{hs}}, e_{\text{hs}}) \): HIR representation of an array assignment statement, where \( e_{\text{hs}} \) and \( e_{\text{hs}} \) are HIR representations of, correspondingly, an array reference (say \( A(\ldots) \)) and of an expression.

Note: We do not treat the case where \( e_{\text{hs}} \) has a scalar shape and \( I_{\text{inher}} \) is empty – i.e., a single (scalar) assignment to one array element.

Output.

If the assignment is buffered, i.e. right-hand side buffering is necessary\(^4\):

- a declaration of a temporary array buffer \( T \)
- a statement allocating \( T \) with the same shape and mapping as \( A \)
- two parallel array assignments:
  \[ s_{\text{pass}}(e_{\text{subscr}}(T,\ldots), e_{\text{hs}}(\ldots), ispace) \]
  \[ s_{\text{pass}}(e_{\text{subscr}}(A,\ldots), e_{\text{subscr}}(T,\ldots), ispace) \]
- a statement freeing \( T \).

If the assignment is non-buffered:

\(^4\) see below
• one parallel array assignment:
  \( s_{paas}(e_{subscr}(A, \ldots), e_{rhs}(\ldots), ispace) \)

Algorithm.

(1) Create iteration space I:
   (a) If \( e_{\text{Hir}}^{\text{Hir}} \) is of array shape\(^5\), create \( I_{\text{priv}} \), the iteration space induced by the implicitly iterated dimensions of the left-hand side.
   (b) Create the concatenation \( I = I_{\text{inherit}} \circ I_{\text{priv}} \)

(2) Create \( e_{\text{subscr}}(A, \ldots) \) using \( e_{\text{Hir}}^{\text{Hir}} \) and indices from I

(3) Create \( e_{\text{rhs}}(\ldots) \) using \( e_{\text{Hir}}^{\text{Hir}} \) and indices from I

(4) If the assignment is non-buffered:
   create the PIR parallel array assignment \( s_{paas}(e_{subscr}(A, \ldots), e_{rhs}(\ldots), I) \)

(5) Else (assignment is buffered):
   (a) Create a PIR declaration introducing a temporary array buffer T
   (b) Create a statement allocating T with same shape and mapping as A
   (c) Create the PIR expression \( e_{\text{subscr}}(T, \ldots) \) using indices from I
   (d) Create two PIR parallel array assignments:
       \( s_{paas}(e_{subscr}(T, \ldots), e_{rhs}(\ldots), I) \)
       \( s_{paas}(e_{subscr}(A, \ldots), e_{subscr}(T, \ldots), I) \)
   (e) Create a statement deallocating the buffer T

The decision whether an assignment is buffered or not is taken in a conservative manner: By default, an assignment is buffered. A list of criteria is applied each of which is a sufficient condition for the buffer being unnecessary, that is, for the left-hand side and right-hand side access sets being disjoint. Here are some of these criteria:

- The assignment statement lies in the scope of an \textsc{independent} directive (i.e. in a \textsc{forall} \textsc{independent} or \textsc{do} \textsc{independent}).
- \( e_{\text{Hir}}^{\text{Hir}} \) is a scalar expression over scalar constants.
- \( e_{\text{Hir}}^{\text{Hir}} \) is a (whole or subscripted) reference to array A, and \( e_{\text{Hir}}^{\text{Hir}} \) contains neither references to A nor references to objects which A shares in an equivalence ring.

This simple method could be improved by employing sophisticated dependency analysis techniques.

7 Transformation of the \textsc{forall} construct

The goal of this transformation is to prepare and control the creation of a PIR array assignment construct for each of the assignment statements occurring in the \textsc{forall}’s body. Before going into more detail, a few words about the treatment of mask expressions. The problem is that a mask expression should be evaluated once, before the iterations of the involved \textsc{forall} loop are actually executed, such that the execution of assignment statements inside the loop might not change the mask’s value(s). To meet this requirement, we create - at each \textsc{forall} nesting level - a boolean mask buffer which is initialized at the beginning. This buffer - not the mask expression itself - is subsequently used as a mask to control the loop’s execution.

Input.

- \( I_{\text{inherit}} \): Iteration space established at the surrounding levels (PIR representation)
- \( e_{\text{forall}}^{\text{Hir}}(\text{section}^{\text{Hir}}, \text{mask}^{\text{Hir}}, \text{body}^{\text{Hir}}) \): HIR representation of a \textsc{forall} statement:

\(^5\)i.e. there are implicitly iterated array dimensions
where

- \( \text{sections}^{\text{HIR}} = (t_1^{\text{HIR}}, t_2^{\text{HIR}}, ..., t_n^{\text{HIR}}) \) is the HIR representation of the \text{FORALL}’s triplets list
- each \( t_j^{\text{HIR}} \) is a quadruple carrying the control index, its lower bound, upper bound, and stride, i.e. \( t_j^{\text{HIR}}(i_j, l_j^{\text{HIR}}, u_j^{\text{HIR}}, s_j^{\text{HIR}}) \)
- \( \text{body}^{\text{HIR}} = (s_1^{\text{HIR}}, s_2^{\text{HIR}}, ..., s_k^{\text{HIR}}) \) represents the statements list of the \text{FORALL}’s body.

According to the \text{FORALL}-\text{WHERE}-assignment nesting structure shown above, a component \( s_{mj}^{\text{HIR}} \) may represent a \text{FORALL} construct, a \text{WHERE} construct, or an assignment statement.

Output (PIR).

- Declaration introducing a boolean mask buffer
- Statements setting the mask buffer;
- Statements generated by looping over the \text{FORALL}’s \( \text{body}^{\text{HIR}} \) list during a recursive descent;

Algorithm.

1. Get \( \text{TOTAL}(I_{\text{inherit}}) \), extracting from \( I_{\text{inherit}} \) its total part;
2. Using input \( \text{sections}^{\text{HIR}} \) list, create \( \text{TOTAL}(I_{\text{priv}}) \), the PIR representation of this \text{FORALL}’s ”private” list of iterators;
3. Create the iteration space passed ”down” the tree, \( I_{\text{down}} = I_{\text{inherit}} \circ \text{TOTAL}(I_{\text{priv}}) \)
4. If the input \( \text{mask}^{\text{HIR}} \) is not null
   a. Transform \( \text{mask}^{\text{HIR}} \) to PIR thus creating this \text{FORALL}’s private mask \( m_{\text{priv}}(\ldots) \)
   b. Create and set the mask buffer, \( e_{\text{mask.buff}}(\ldots) \):
      - Create a declaration to introduce a mask buffer, a boolean array over the index domain determined by \( \text{TOTAL}(I_{\text{inherit}}) \circ \text{TOTAL}(I_{\text{priv}}), d(\text{buff}, \ldots) \)
      - Create a PIR parallel array assignment initializing the mask buffer,
        \( s_{\text{pass}}(e_{\text{subjct}}(\text{buff}, \ldots), false, \text{TOTAL}(I_{\text{inherit}}) \circ \text{TOTAL}(I_{\text{priv}})) \)
      - Create a PIR parallel array assignment selectively setting the mask buffer,
        \( s_{\text{pass}}(e_{\text{subjct}}(\text{buff}, \ldots), true, I_{\text{inherit}} \circ \text{TOTAL}(I_{\text{priv}}) \circ m_{\text{priv}}(\ldots) \)
   c. Modify the iteration space passed ”down” the tree,
      \( I_{\text{down}} = I_{\text{down}} \circ m(e(\text{buff})) \)
5. Loop over the \text{FORALL}’s \( \text{body}^{\text{HIR}} \), apply transformation algorithms for assignments, \WHERE \) constructs and \text{FORALL} constructs recursively as appropriate using \( I_{\text{down}} \) as the inherited iteration-space input.

8 Transformation of \WHERE \) construct and DO INDEPENDENT loop

The \WHERE \) construct is translated similarly to the \text{FORALL}, its main effect being the introduction of an additional mask which is applied to an iteration space either inherited from a surrounding \text{FORALL} or inferred from the array’s index domain. The \ELSEWHERE part is implemented by using the mask buffer with a negation.

For DO INDEPENDENT loop nests only containing array (element) assignments as their ”leaves” (which can be identified by IR pattern matching techniques), a technique similar to the \text{FORALL} transformation described above can be applied: an iteration space is successively built and transferred to a recursive invocation of the array assignment transformation algorithm (which in this case does not need to introduce a buffer).
9 Buffer storage optimization

In some cases, the compiler can analyze that only part of the temporary buffer will actually be used, because the data-parallel source statement being translated only accesses a subset of the original array's elements. Then the compiler can restrict the actual allocation to a ”packed” array, that way minimizing memory waste. This requires techniques to rewrite access functions and determine a ”projected” mapping for the buffer that again ensures perfect alignment with the original array which, however, cannot be described any further here.

10 Conclusions and acknowledgements

Techniques were presented to uniformly translate several data-parallel Fortran90/HPF constructs. The unification of these source constructs into one expressive yet simple internal structure simplified design and implementation of many subsequent components of the PREPARE HPF compiler. This structure also had many more interesting applications inside the HPT component itself that cannot be shown here due to space limitations – e.g., for the implementation of HPF procedure interfaces. The implemented translation techniques were tested on a wide range of applications and have proven to be efficient.

We want to thank our collaborators in the PREPARE project. Concepts and techniques presented here were influenced in a very advantageous manner by discussions with the following people: Arthur Veen, Erik van Konijnenburg, Will Denissen, Charalampos "Babis" Koutsoumalis and Franck Gousset (who also was heavily involved in the implementation of the HPT).

References

How to Add a New Phase in PIPS: the Case of Dead Code Elimination

Corinne ANCOURT  Fabien COELHO  Béatrice CREUSILLET  Ronan KERYELL


Abstract

PIPS is an experimental tool to implement and evaluate various interprocedural compilation, parallelization, analysis and optimization techniques. This paper focuses on the workbench used to build these compilers and how to add a new phase in PIPS, a dead code elimination, using the management of data structures, of dependences between the various analysis and transformation phases, and using the mathematical libraries in PIPS. PIPS is available to the community and it is what motivates this article.

Keywords: programming environment, workbench, make, interprocedural parallelizer, interactive parallelizer, optimizer, dead code elimination, reverse engineering, CASE.

1 Introduction

Together with the computer architectural evolution, three main directions in compiler developments have been pursued: compilers for sequential machines (with superscalar processors), parallelizers of sequential programs and compilers for explicit parallel programs. All these approaches benefit from deep global program analyses, such as interprocedural and semantical analyses, to perform optimizations, vectorization, parallelization, transformations, restructuring and reverse-engineering of programs. Since these approaches often require the same or share a common ground of analyses, it is interesting to factorize them out in a common development tool to get the benefit of code re-use and modular programming.

PIPS is such a highly modular workbench for implementing and assessing various interprocedural compilers, optimizers, parallelizers, vectorizers, restructurers, etc. without having to build a new compiler from scratch. It has been used in various compilation areas since its inception in 1988 as the PIPS project (Interprocedural Parallelization of Scientific Programs).

This project aims at combining advanced interprocedural and semantical analyses [10] with a requirement for compilation speed. The mathematical foundations of the semantical analysis implemented in PIPS are linear programming and polyhedral theory. Despite such advanced techniques, PIPS is able to deal with real-life programs such as benchmarks provided by ONERA (French research institute in aeronautics), or the Perfect Club in quite reasonable time.

PIPS is a source to source and multi-target parallelizer. It incorporates many code transformations and options to tune its features and phases. PIPS offers interesting solutions to solve programming design problems such as generating high-level data structures from specifications,
co-generating their documentation, defining user interface description and configuration files, dealing with object persistence and resource dependences, being able to write parts of PIPS in various languages, etc.

This paper details how to deal with some of these features to add a new phase in PIPS and a dead code elimination phase will serve as a running example throughout the paper. Such a description could seem somewhat cumbersome but it gives an idea of how the PIPS features are organized for people wanting to use PIPS as a compiler development tool since it is publicly available.

We first present the general design of PIPS in Section 2. In Section 3 we sketch a dead code elimination phase and we explain how it is added to PIPS in Section 4.

## 2 PIPS overview

PIPS is a source-to-source Fortran translator (Figure 1), Fortran still being the language of choice for most scientific applications. Beside standard Fortran 77, various dialects can be used as input (HPF) or output (CMF, Fortran 77 with Cray directives, Fortran 77 with HPF parallelism directives, etc.) to express parallelism or code/data distributions.

Fortran can be seen here as a kind of portable language to run on various computers without having to deal with an assembly code back-end, although the PIPS infrastructure could be used as a basis to generate optimized assembly code. Other parsers or prettyprinters could be added and the internal representation could be extended if need be to cope with other imperative languages such as C.

Following the field research history, PIPS has first been used to improve the vectorization of Fortran code for parallel vector computers with shared memory (Fortran 77 with DUAL, Fortran 77 with Cray micro-tasking directives, Fortran 90). It is now mainly targeted at generating code for distributed memory machines, using different methods (processor and memory bank code for control distribution [1], CMF, CRAFT polyhedral method [9] or message-passing code from HPF [2]).

In PIPS, the translation process is broken into smaller modular independent operations called phases that communicate only by sharing resources (such as the module code, semantic information, etc.) stored in a resource manager PIPSDRM. Since PIPS is an interprocedural environment, procedures and functions are very important in PIPS and are indifferently referred to as modules in the sequel. For the user and the programmer, the phases can be classified into various categories according to their usage.

Figure 1: User view of the PIPS system.
A first class of phases are called analyses and compute some internal information that can be later used to parallelize (or generate) some code, some user information (such as a program complexity measure) to be later displayed by a prettyprinter, etc. The most interesting analyses available in PIPS are the semantical analyses such as some predicates on some integer variables of the program [5] or the regions that describes some array elements [4].

Another important class is the set of the code generation phases that produce a new code from the information given by the analyses. Important code generators are the parallelizer, the code distributor [1] and the HPPF compiler [3].

A third class contains the prettyprinters that transform and merge some internal PIPS resources (code, preconditions, etc.) in a user viewable file.

The last phases are called transformations because they modify the program code. The available transformations are loop distribution, scalar and array privatization based on regions [4], etc. In this article, we present a dead code elimination transformation.

All the resources about a program are kept in a workspace database with the help of PIPSDBM and the global coherence is insured with the interprocedural PIPSMAKE manager. More details can be found in [8].

3 Defining a dead code elimination phase

Semantical information known at compile time can be used to simplify the code in many situations, such as for code specialization when a library function is used in a specific context, when all the code is not used and thus can be removed, for simplifying old code with useless parts or as back-end optimizer after some generic code transformations or compilation phases.

In PIPS for example, the preconditions give some affine predicates on integer variables of the program. These predicates can be used to simplify the code locally by partially evaluating it.

The simplest rule is to remove a statement when its predicate has no integer solution such as in an IF statement with a condition never satisfied such as \( \{ 0 \leq -1 \} \) on Figure 3.

In fact this can be done further: since the IF is then empty, it can be removed itself if the condition computation has no side effect such as doing an I/O or touching a global variable. This last information is abstracted in PIPS by the proper effects resource that give information about the variables read or written for each statement.

Figure 2: Programmer view of the PIPS system.

Figure 3: Simple code with dead code highlighted by preconditions
For the loop optimizations, if the preconditions can prove that a loop cannot be executed, it can be removed. But if the computation of the loop bounds has some side effects, the loop must be replaced with these equivalent side effects, such as setting the index variable to its lower bound value if it is used later.

In the same way, if a loop can be proved to be executed only once, the loop can be replaced by its body and if the loop statement itself has a side effect, the same side effect should be kept. The information needed to implement this kind of transformation is the preconditions, the proper effects and of course the code itself.

Many rules such as the previous one can be easily added and implemented in a dead code elimination transformation in PIPS. In the following part we do not focus on the kind of optimizations that can be made but on how to use the PIPS framework to add such a transformation.

4. Adding a new phase: dead code elimination

In this section, we are explaining the different tasks needed to add a new phase in PIPS. First we describe the global directory hierarchy and phases basic files before describing the use of the PIPSMAKE, PIPSDBM and NEWGEN tools.

4.1 Directory hierarchy

The PIPS main directory hierarchy is mainly organized with the following concepts [8]:

Production: contains the last coherent version of the PIPS software (Bin, Runtime), documentation (Doc, Html), examples of validation (Validation) and sources (Src). The production directory is pointed by $PIPS_ROOT;

Development (PIPS_DIR/Development) is used to develop all the pieces of PIPS. By default, a part of PIPS in development is tested with all the other parts from the Production directory to avoid as much as possible the interactions between concurrent developments of different parts at the same time ($PIPS_DEVEDIR). Important directories of Development are Libs (the internal parts of PIPS, such as each phase), Passes (the various PIPS executable with the different user interfaces pips, tpips and wpips), Documentation (the documentation itself of course but also many configuration files), Runtimes (support for HPF, message passing and graphics) and Scripts. Installing a part of PIPS (with a classical GNU make install) is done in fact by copying the needed files and directory in the ad hoc directories of Production;

For portability purpose, an architectural hierarchy à la PVM has been reused in PIPS with the $PIPS_ARCH variable to compile and store different binary architectures.

4.2 Basic file definitions

Since all the phases are kept in the PIPS library, a new phase p needs a new directory d in $PIPS_DIR/Development/Libs or at least should be put in an old one d. The generic structure of a PIPS source directory and a library in particular contains a config.makefile, a d-local.h file and various source files.

A phase p itself is a function compatible with the C declaration of Figure 4. Thus if one wants to write a new phase in another language such as Fortran or Common Lisp, a C wrapping function should be used to call the phase kernel itself in this particular language and return the success flag. A phase returns false when it fails and PIPSMAKE stops executing the various phases.

The config.makefile is aimed to derive a read-only GNU Makefile through the PIPS make file generator pips-makesmake that will be used to compile the sources in the directory d to
build a library libd.a that will be linked later with other parts of PIPS to have a working executable. Many PIPS-specific generic rules are generated by pips-makemake (for C, Fortran, lex and yacc files, HTML and \LaTeX\ documentation, various binary architectures, etc) and often the config.makefile is quite short, typically declaring the source files to use.

In the same way, a local include file d-local.h is used by the Makefile to generate a file d.h that declares all the functions and variables to be known from outside to use the d library. d.h is the concatenation of d-local.h and all the non-static objects of the local C files extracted by cp. 

To sum up the directory creation and declaration for a new phase:

1. create the directory d if necessary;
2. declare the list of the useful files in a config.makefile;
3. run pips-makemake;
4. write a p function that does the phase job;
5. run GNU make
6. debug it and iterate on previous points, eventually by doing a local pass as described later;
7. once the previous code has been compiled, the library is ready to be added to PIPS with editing the file $PIPS_DIR/Development/Scripts/dev/define_libraries.sh and typing make install.

In fact, the last points are a little bit interleaved since it rather hard to debug a new or modified phase without installing it, but it is dangerous to install some unstable code. For that purpose, there exist the rules test, ttest and vtest to locally build the passes pips, tpips and wpips respectively by linking with the local library instead of the one installed in the production version $PIPS_ROOT/Lib. Since often several libraries are co-developed and need to be linked together to be debugged, one can add all the other library d’ by doing a ln -s .. /d’/libd’ .a.

In our case, we want to add a dead code elimination phase that is a transformation. Since there are many transformations with a rather small code, these are put all together in the transformations library. Since this library already exists, it is useless to create a new transformations directory in $PIPS_DEVEDIR/Libs. Just create the file dead_code_ elimination.c and add its name in the LIB_CFILES of the config.makefile for the file to be considered when building the library.

Our dead code elimination has no reason to fail since we only apply legal optimization if necessary, thus the suppress_dead_code() function always returns TRUE, as we will see on Figure 6.

4.3 Using the interprocedural PIPSMake consistency manager

From a theoretical point of view, the object types and functions available in PIPS define an heterogeneous algebra with constructors (e.g. parsers), extractors (e.g. pretty-printers) and operators (e.g. loop unrolling and other transformations). Very few combinations of functions make sense, but many functions and object types are available. This abundance is confusing for casual and even experienced users and it was deemed necessary to assist them by providing default computation rules and automatic consistency management.
The PIPSMake library - not so far from the Unix make utility - is the corner stone of PIPS interprocedurality. The objects it manages are resources stored in memory or/and on disk. The phases are described in a pipsmake-rc file by generic rules that use and produce resources. The ordering of the computation is demand-driven, dynamically and automatically deduced from the pipsmake-rc specification file.

In our dead code elimination example, we need the code of the current module of course, but also its preconditions and its proper effects for the semantical information. The entities of all the program are also needed to have access to variables, COMMONs and so on. The global effect of this transformation is to rewrite the code of the current module. These dependences are abstracted by the pipsmake-rc syntax on Figure 5.

For each rule, that is in fact the textual name of a phase C function, it is possible to add a list of constraints composed with a qualifier, an owner name and a resource name. Each constraint has the following syntax:

\[
\langle \text{qualifier} \rangle \langle \text{owner-name} \rangle . \langle \text{resource-name} \rangle
\]

The qualifier describes how the resource can be used: \langle for needed, \rangle generated, \# destroyed by the rule, = asserts that the resource remains valid even if other dependence rules would suggest to invalidate the resource, \! asks for applying another rule before executing the rule.

An owner name specifies the owner of the needed resource. For example for the code resource, only the code of this module, or the code of all the modules, etc. Two interesting owners are CALLEES (the set of modules called by the current one, used to describe bottom-up analyses on the call-graph) and CALLERS (the set of modules calling the current one that can expresses top-down analyses). They are the mechanism that deals with interprocedurality in PIPS. In our small example, the interprocedurality is not highlighted because if the dead code elimination uses interprocedural information (the preconditions and the effects), it is not interprocedural per se.

An API is available for advanced usage to call PIPSMake directly to build a resource inside a user interface for example.

4.4 Using the PIPS data structures with PIPSDBM database manager

Every useful phase needs at least to read and use some data structures describing the user program or some other data and also needs to write some results or the modified user program for example. In PIPS, data management is done through PIPSDBM, a system able to read and write any NEWGEN [7] data structure or even non-NEWGEN ones (with user-provided read, write and copy functions) in a portable way (architecture and memory location independent) and persistent way. Such a resource can be on disk or memory but when PIPS exits, all the resources that are flagged as useful are stored on disk and when the programmer asks for a resource, it is loaded from disk if it is not in memory.

In such a way, the PIPS design is more modular, with many small independent modules. It is preemptive at each phase boundary since then all the data are in a coherent state with PIPSDBM. It is useful to have more reactive user interfaces. Further more, a PIPS session can be shutdown and brought up later by retrieving the former state from disk.
bool suppress_dead_code(string module_name)
{
    /* get the resources */
    statement module_stmt = (statement)
    db.get_memory_resource(DBR_CODE, module_name, TRUE);
    set.proper.effects.map((statement_mapping)
    db.get_memory_resource(DBR_PROPER_EFFECTS, module_name, TRUE));
    set.precondition.map((statement_mapping)
    db.get_memory_resource(DBR_PRECONDITIONS, module_name, TRUE));

    set.current.module.statement(module_stmt);
    set.current.module.entity(local_name.to.top.level.entity(module_name));

    debug.on("DEAD_CODE_DEBUG_LEVEL");
    /* really do the job here: */
    suppress_dead_code.statement(module_stmt);
    debug.off();

    /* returns the updated code to Pips DBM */
    DBR_PUT_MEMORY_RESOURCE(DBR_CODE, module_name, module_stmt);
    reset.current.module.statement();
    reset.current.module.entity();
    reset.proper.effects.map();
    reset.precondition.map();

    return TRUE;
}

Figure 6: Excerpt of the main function of the dead code elimination phase.

4.4.1 Loading and storing resources

Since the only useful operations are to get a resource from or to put a resource into PIPSDBM, the programmer interface is reduced to

string db.get_resource(string resource_name, string module_name, bool genuine)
void db.put_resource(string resource_name, string module_name, void* value)

The first function returns a pointer to the data structure resource_name for module module_name. If genuine is TRUE, the programmer gets the genuine resource. That means that modifying it will modify the resource used by subsequent users. The genuine flag with a value TRUE is useful to modify a resource or to use it read-only for example. When using it as a scratch-pad or modifying it to transform it in another resource, one must ask for a copy of the genuine resource with the value FALSE for flag genuine.

For our dead code elimination, we need to modify the module code, thus we need to get the genuine resource and put it back into PIPSDBM. The information about the code used to simplify it is used on a read only basis, that means that the genuine resources can be accessed to avoid useless copies since we do not modify them. DBR_CODE (the abstract syntax of the module), DBR_PROPER_EFFECTS (the proper effects of the statements of the module) and DBR_PRECONDITIONS (the preconditions for the statements of the modules). We can now write the main function for our dead code elimination as shown on Figure 6 with the use of PIPSDBM.
4.4.2 Creating new resources

Our dead code elimination phase is a simple code transformation example and in fact all the needed resources already exist in PIPS and we do not produce some new resources. However, if we try to add some new semantical analyses, some new languages features, etc. some new resources to store these information is necessary. This new resource creation process is explained in the PIPS development environment guide [8].

To avoid conflicts with already defined data structures in the PIPS sources, a resource resource is named internally DBRRESOURCE with big caps.

Since we do not use dynamic linking and UNIX nlist() for portability reasons, adding a new resource with the related methods require a recompilation of PIPS. But since adding a new resource does not occur very often and implies that a new phase is also added or at least modified to use it and that it requires also a recompilation, it is not a big deal.

The new resources must be added in the $PIPS_DIR/Validation/Libs/pipsdbm library that is the core of PIPSDBM. The methods to be used for loading, unloading, freeing and verifying are added in the method_map[] array of the file method_io.c. The file module.c must be modified to reflect the unloading order of the resource hierarchy. For file resources (used for example for prettyprinters), the file disk.c need also to be updated. After this modification, the PIPSDBM library must be installed and PIPS recompiled (with make-pips -r). Further explanation are found in the development environment guide [8].

4.5 Avoiding global variables

Global variables in big projects such as PIPS should be avoided as much as possible since they are often the source of big trouble when many different modules are interacting and when the global state has not been proved to be safe. Two encapsulations are used in PIPS to provide global state keeping and enabling global options.

4.5.1 Hidden global variables

Nevertheless, some information such as “what is the statement of the current module” is often used in many places and it is cumbersome having to pass this information through some recursions. For that purpose, some interesting global variables exist but they have been encapsulated in set_ and reset_ methods to insure that a variable will not be left with an old value when leaving a phase. The get_method is used to use the value.

In the main function of dead code elimination on Figure 6, the current module statement and entity are stored by using the set_current_module_statement() and set_current_module_entity().

local_name_to_top_level_entity(n) gets the entity corresponding to the module name. The module statement and entity are useful to directly access data structures that are global to the current module (such as variables, etc.) when doing some code analysis or transformation.

Since the dead code elimination needs to know some semantical information about the current module, such as the proper effects of a statement to deal with I/O and the preconditions to have informations on variable values, it is stored with set_proper_effects_map() and set_precondition_map() for later use. At the end of the procedure, the global variables are reset accordingly.

4.5.2 Properties

The other object in PIPS that allows parameterization is called a property. Properties are used for various purposes: adding some information or selecting a style in a prettyprinter, asking to compute some statistics, selecting a special feature, etc.
The user interfaces can be used to modify the current properties and if a local properties.rc exists, the last one modifies the properties defined in the main PIPS properties.rc file.

Note that since the PIPS behavior can be highly altered by the properties, the properties used at the time of a workspace creation is stored in the workspace itself to be sure to always use the same properties along the life of a workspace. In such a way, even if the main or local properties.rc is modified, the PIPS behavior is coherent with the information persistence through different runs.

4.6 NewGen data structures in PIPS

An other key point of PIPS is the NewGen [7] description language that is used in PIPS to define all the data structures. Domains are defined NewGen to express complex data structures such as sums (+ like a C-union) or products (× like a C-struct) of other domains, lists (•), sets (Ω), array (Ω) or maps (→) of domains. These domains are translated by NewGen in type definitions for different languages (C and Common-Lisp for now) with different methods to deal with this data: creators, destructors, accessors, readers and writers, etc. The last ones allow persistence and having PIPS written in different languages but sharing the same data. In this way, NewGen adds an object oriented layer that abstracts the target language data structures.

In our dead code elimination example, we do not define any new type but only use the internal representation of the abstract syntax tree. An excerpt of its definition is on Figure 7. All the NewGen data structures are defined in the LATEX files from $PIPS_DIR/Development/Documentation/newgen that are used to generate the documentation about the types and of course the target language specifications.

NewGen defines a generic mono- or multi-domain iterator (gen_recurse and gen_multi_recurse in C) that recursively applies functions on a list of domain top-down and then a bottom-up on a data structure. In our example on Figure 8 in suppress_dead_code_stateinent() we iterate on all the statements of the code and apply them dead_statement_filter() top-down to eliminate some dead code, recurse if this function returns TRUE, and then dead_statement_rewrite() bottom up to remove some code that has become useless because of the previous elimination of nested statements. All the accessors such as statement_instruction() to get or set an instruction of a statement or the is_instruction_loop tag to test if an instruction is a loop are generated by NewGen.

4.7 Using the C³ linear algebra library

To test if an instruction is useful, we need to evaluate the preconditions to know if there is an integer solution to the predicate. For this purpose, we use the C³ linear algebra library that provides support for vector and matrix operations, feasibility tests with simplex of Fourier-Motzkin, etc. In this example, we use in dead_statement_filter()
bool
statement_weakly_feasible_p(statement s)
{
    transformer pre =
        load_statement.precondition(s);
    /* Get the precondition linear constraint
     system associated with s */
    systems sc =
        predicate.system(transformer.relation(pre));
    /* Test if there is at least a solution */
    bool feasible_p = lsc_empty_p(sc);
    return feasible_p;
}

static bool
dead_statement_filter(statement s)
{
    instruction i = statement.instruction(s);
    bool ret = TRUE;
    ifdef(9) {
        debug(9, "dead_statement_filter",
            "The current statement:\n");
        print.text(stderr, text_statement(get_current_module.entity(), 0,
            s));
    }
    if (!statement_weakly_feasible_p(s))
        /* empty precondition */
        ret = remove_dead_statement(s, i);
    else
    {
        switch (instruction.tag(i)) {
            case is_instruction_loop:
            {
                loop l = instruction_loop(i);
                if (dead_loop.p(l))
                    /* DO I=1,0 for example */
                    ret = remove_dead_loop(s, i, l);
                else if (loop.executed_once.p(l))
                    /* DO I=N,N for example */
                    remove_loop_statement(s, i, l);
                suppress.dead.code_statement(body);
            }
        }
    }
    return ret;
}

void
supress.dead.code_statement(module stmt)
{
    gen_recurs(module.stmt, /* recursion from
        here */
        statement.domain, /* on
        statements */
    /* entry function (top-down): */
    dead_statement_filter,
    /* exit function (bottom-up): */
    dead_statement_rewrite;
}

Figure 8: Excerpt of the dead code elimination phase.

statement_weakly_feasible_p() that calls the $C^3$ function sc_empty_p() to know if there is a solution to the system of constraints.

4.8 Debug support

In addition to fine debug support at the debugger level of NEWGEN and PIPS objects, higher level debug can be triggered using environment variables such as $DEAD\_CODE\_DEBUG\_LEVEL$ on Figure 6 with the debug_on() called up to a matching debug_off(). For example, if $DEAD\_CODE\_DEBUG\_LEVEL$ has the value 9 or more, dead_statement_filter() displays its current statement on Figure 8.
4.9 Non-regression validation tests

Managing a big project is very hard without appropriate test procedures. For this goal, it is quite useful to add some test examples to run PIPS every night on it and to compare the output with files containing the expected results. In this way, running a validation after modifying a part of PIPS give more confidence about the global status.

The typical test example is put in a directory in $PIPS_ROOT/Validation, for example in the Transformation directory. For each example Fortran code contained in example.f, a script example.test and a example.result/test should be written so that every night the script is executed and use example.f to generate an output compared to the test expected output.

4.10 User interface

The various user interfaces of PIPS use the pipsmake-rc.tex file to deduce some automatic name completion and graphical menus for WPIPS and EPIPS. For the graphical interface, the transformation menu layout is specified by the \begin{PipsMenu}{Transformations} ... \end{PipsMenu} \[ \text{T}_{\text{TX}} \text{X} \] environment of the pipsmake-rc file.

Since the phase names are not very user-friendly, some aliases are defined also in the pipsmake-rc file to be used instead of the phase names in the Options menu for example.

4.11 Documentation

This section comes late but of course it is nevertheless one of the most important to deal with long-term usage and software maintenance.

The documentation system heavily rely on GNU make, \texttt{\LaTeX} and \texttt{\LaTeX2HTML} since they are portable, widely spread and easily integrated in an automatic documentation process \textit{à la} UNIX. \texttt{\LaTeX} is very nice to have a high quality printed document and \texttt{\LaTeX2HTML} to add navigation capabilities for browsing.

When the pipsmake-rc file is updated to add a new phase, some documentation should also be added to present what the phase is about. In this way, the user manual can automatically be derived with a description of every phase accessible from a menu.

Once the documentation exists, it can automatically be published on the Internet with the World Wide Web protocol and retrieved using the standard meta-indexers as AltaVista by an interesting world-wide side effect...

5 Conclusion

The originality of PIPS is the conjunction of:

- modularity and easy adding of a new phase as stated in this article;
- interprocedurality and automatic coherency management in a demand-driven system combining database (PIPSDBM) and dependence (PIPSMAKE) functions;
- advanced interprocedural semantic and data-flow analyses already implemented, and their mathematical foundation upon linear algebra (Linear C0 library).
- software foundation upon an high level description system (NEWGEN) which automatically generates functions (access, constructor, destructor) and can iterate on arbitrary structures. An additional by-product is that the code look and feel is homogeneous over the 150,000 lines of the compiler.
- multi-target:
- automatic parallelization for shared-memory
- compilation for distributed-memory models (HPFC [2], WP65 [1])
- automatic data mapping ...

- the availability of different interfaces (shell, terminal TPIPS, graphical WPIPS).

Eight years after its inception, PIPS as a workbench is still alive and well. PIPS provides a robust infrastructure for new experiments in compilation, program analysis, optimization, transformation and parallelization. The PIPS developer environment is described in [6], but it is also possible to develop new phases on top of but outside of PIPS since all (in fact, most...) PIPS data structures can be reloaded using NEWGEN primitives.

PIPS can also be used as a reverse engineering tool. region analyses [4] provide useful summaries of procedure effects, while precondition-based partial evaluation and dead code elimination reduce code size.

PIPS may be less robust than other publicly available source-to-source compilers but the stress is put on the ability to quickly add new phases such as program transformations, semantic analyses, parsers, parallelizers, etc. or new user interfaces without spending time developing a cumbersome infrastructure.

The PIPS software and documentation is available on http://www.cri.ensmp.fr/pips. The HTML version of this paper should be used to obtain many links to the PIPS project and related work.

5.1 Acknowledgment

We would like to thank many people that have contributed to the PIPS project: Bruno BARON, Denis BARTHOU, Pierre BERTHOMIER, François IRIGOIN, Pierre JOUVELOT, Arnauld LESERVOT, Guillaume OGET, Alexis PLATONOFF, Rémi TROLET, Yi-Qing YANG, Lei ZHOU.

References

A Case Study of Code Generator Generation for Embedded SIMD Computers

Andreas Persson  
Johan Ringström  
Peter Fritzson

Department of Computer and Information Science  
Linköping University, S-581 83 Linköping, Sweden  
Email: petfr@ida.liu.se, johri@ida.liu.se

Abstract

Can today’s most advanced compiler generation systems handle specialized parallel processor architectures? To answer this question, a compiler targeting the embedded RVIP SIMD architecture was generated, using a combination of the DML-P front-end generator and the BEG back-end generator from the CoSy compiler generation toolset.

A number of difficulties were encountered when specifying the code generator, for example inability to denote arbitrary register sequences in BEG specifications. However, the end result was positive and a number of lessons were learned on how to improve and generalize the code generation framework. An industrial-strength radar image filtering application was compiled with the generated compiler, giving a benchmarked performance of 2.8 times slower compared to the same application implemented in micro-code like assembly. Despite the slow-down, industry considered this to be much better than initially expected. By improving BEG, including optimizing transformers from CoSy, and utilizing instruction scheduling, we expect that future versions of the generated code generator can come close to the quality of hand-written assembly code.

To partly answer this question in the context of data-parallel languages, we have made a case study on generating a compiler for a very special purpose SIMD processor architecture, the RVIP.

The RVIP (Radar Video Image Processor) is a SIMD [16] special-purpose LIW architecture that uses bit-serial processing elements and is used in embedded systems for the execution of radar image processing applications. It may soon appear in other image analysis applications. This architecture is currently programmed in a mixture of C for the sequential part of the applications, running on standard processors, and microcode assembly for the data-parallel processing elements. There is a need for a high-level language and implementation which can compile programs to this architecture.

The special properties of RVIP strains the current compiler generation system to its limits, thus providing an independent experimental check on the generality of the tool approach also in the case of compilers for parallel architectures. The tools used in this case study are the DML-P system for front-end generation, and the BEG system for code generator generation.

The DML-P [15] compiler generation system currently is able to produce code generators for two data-parallel architectures: the MasPar MP-1 SIMD system and the RVIP system. Data-parallel calls are generated to the CVL[1] library which is implemented on the MasPar[9], but for which special code was needed for each used operator on the RVIP. The DML-P system has so far been used to produce compilers for the Predula Nouveau language, which is a Pascal-like programming language with a data-parallel extension consisting of a set of data-parallel functions which are similar to the CVL operators.

This paper presents a case study of applying the BEG back-end generator targeting RVIP, and extending and applying the DML-P front-end generator in order to generate a practical compiler for a data-parallel high-level language on the special purpose RVIP architecture. A radar image filtering application was compiled and benchmarked on the RVIP. In addition, a number of lessons were learned on how to generalize and extend the code generation framework.

1 Introduction

Compiler generation systems partially automate the process of constructing compilers. For example, recent progress in this area allows automatic generation of code generators that include global register allocation and instruction scheduling, as for the BEG tool[8] in the CoSy toolset used in this work. BEG is probably one of the most advanced code generator of the dynamic programming type. As compared to Burg[10], another well-known tool, it allows general rule conditions to be evaluated at code generation time, as well as including global register allocation and instruction scheduling. However, the question is whether such tools can be made flexible and general enough to cope with a wide range of processors and languages, e.g. including special purpose parallel architectures.
2 Data Parallel Operations

Data parallel operations operate on collections of similar data elements, usually represented as vectors. If these operations are created with portability and efficiency in mind, the programmer can more easily write portable programs using these operators without having to worry about the details of the underlying architecture.

There seems to be no commonly accepted definition for data parallel operations. We chose the Scan Vector Model [5], and the CVL (C Vector Library) implementation. CVL is a collection of low-level vector routines, where the basic idea is to present an abstract model of a vector machine, so that the user doesn’t need to know the details of the actual hardware architecture. The CVL has been implemented both on single processor computers and on various parallel machines.

We compile a simplified subset of CVL where we don’t consider nested data-parallel operations: vector-valued operations on elements in other vectors. We also don’t consider segmented vector operations and data-parallel operations such as reduce which produce a single scalar value. These simplifications were made both because the availability of these operations were not crucial for the intended application area, and because the rather specialized hardware architecture could not or would not be able to support the operations without a substantial amount of work.

The data-parallel operations can be divided into a number of classes: elementwise, reduce, scan, permute and vector-scalar operations.

An elementwise operation, elementwise-\(\oplus\), operates on vectors of equal size and produces a result vector of the same length. The element \(i\) of the result depends only on element \(i\) of the input vectors and is the result of operator \(\oplus\).

A reduce operation, reduce-\(\oplus\) produces a scalar result which is the result of placing the \(\oplus\) operator between the elements in argument vector. A reduce-\(\oplus\) operation on \([a_1, a_2, a_3, \ldots, a_n]\) returns \(a_1 \oplus a_2 \oplus \ldots \oplus a_n\) as result.

A scan-\(\oplus\) operation uses one vector as input, and returns an equal sized vector result which contains the results of using reduce-\(\oplus\) on all prefixes of the argument vector.

The permute operations rearrange the order of the elements in a vector using another vector, the index vector. The simplest versions of permute correspond to parallel array indexing operations \(dest[index[i]] := src[i]\), where \(src\) and \(index\) are the input vectors, and \(dest\) is output.

Vector-Scalar operations take both scalar and vector arguments. Two common ones are distribute and replace. Distribute fills a vector with a scalar value, and replace changes one element in a vector.

3 The DML-P System

The DML-P (Denotational Meta Language Parallel) compiler generation system uses the denotational semantics [14] specification formalism to specify the sequential parts of programming languages. Generated compilers produce sequential code with quality comparable to hand-written compilers. Data parallel languages are currently supported through the CVL library which makes it possible to generate efficient compilers for data parallel languages.

The DML-P system uses the Predula Nouvelle Pascal-like imperative programming language as its principal test case. This language contains a set of data-parallel primitives which uses arrays that can be stored in a possible vector memory, and provides special functions that operates on arrays, such as each (elementwise operation), scan and reduce. Flexible array handling, including dynamic array sizes, is available. Array variable placement directives is currently not included in the language: the compiler automatically chooses array variable placement for architectures where this is required.

A compiler generated by DML-P compiles to low-level ANSI C code in quadruple form: the C compiler is viewed as an advanced optimizing assembler. The separation between the sequential and parallel parts is made by the CVL library, that is, parallel computation is made through calls to CVL.

4 The RVIP Processor Array

The RVIP [13, 11] is the newest in a series of bit serial architectures with a one-dimensional connection
topology which has been developed by the LSI Design Group at the Department of Physics and the Image Processing Group at the Department of Electrical Engineering at Linköping University, together with Ericsson Microwave Systems Inc. The RVIP architecture is intended for embedded use.

The processing elements (PEs) are controlled by a dedicated control unit, the micro controller, which broadcasts instructions to the PEs. A front-end computer is also needed that tells the micro controllers what to do, and that does the sequential computing.

The fact that each PE in the RVIP is bit serial has both pros and cons:

- Programming is cumbersome. For almost anything that needs to be done, an indexed loop must be used.
- The usual bit length limitations don’t exist. On a normal processor, data often has to be padded. On the bit-addressed RVIP all the available memory and arithmetic and processing units can be used in an economical way.

The RVIP isn’t capable of switching off PEs, instead a multiplex instruction handles the cases where only some of the PEs should produce results.

4.1 RVIP Implementation

Since one important consideration of the RVIP is chip space economy, the architecture has been made very simple, which makes it possible to fit 64 PEs on an RVIP chip. A PE is shown in Figure 2. Eight RVIP chips are packaged in a Multi Chip Module, MCM. The MCM also contains a micro controller and a program memory. Several MCMs can be assembled to create a larger system.

A PE contains:

- a 32 bit accumulator register, or seen in another way: 32 one bit accumulators.
- 2048 bits of RAM memory.
- a 16 bit shift register which is used to shift values from one PE to a neighbour PE.
- a 10 bit serial/parallel multiplier.
- four 32 bit IO registers.
- a bit serial ALU.

The accumulator isn’t directly connected to the bus, all accesses are done via the ALU. This has the advantage that the RVIP can for example do a multiply-then-add, a very useful instruction in the video image area, in just one clock cycle.

4.2 Programming the RVIP

The RVIP is programmed at the micro code level. An instruction has the following format:

```
instruction := busaction? nonbusaction
```

A busaction specifies which parts of the PE that should be connected to the data bus. More than one destination is allowed. The nonbusactions are independent of the data bus.

An example RVIP instruction looks as follows:

```
sreg(1) = ram(17), loadcooff(7);
```

This will copy the contents of the ram address 17 to bit 1 of the shift register. At the same time the parallel multiplier register is loaded with the constant 7.

As the RVIP is a bit serial processor, one of the most frequently used constructs is the indexed loop. The following is an example of how to use indexing:

```
loop(3,i);
    ram(i) = iorreg(1,i),
endloop;
```
These instructions (note that there are only two!) will copy the eight bit number stored in the IO register to ram memory address 1. i is the name of the index register; the RVIP microcontroller has three loop registers, but only one of them can be used for indexing.

The complete RVIP instruction set is given in [13].

5 The CoSy System

CoSy [4, 5] (Compilation System) is a tool set for compiler construction.

- A standardized intermediate representation form, CCMIR, Common COMPARE Medium-level Intermediate Representation [3], which all CoSy compilers work on. The CCMIR definition is described in [SDL].
- The concept of engines. A CoSy compiler is built of several engines, these could for instance be front-ends, optimizers and code generators.
- A special language, SDL, Engine Description Language, is used for describing how the engines will interact, if they are going to run in parallel or in sequence, etc.
- Another special language, SSDL, full-Structure Definition Language [7], used to define the abstract data types on which the engines will work.
- Tools that assist the construction of the engines. Most important is perhaps the previously mentioned code generator generator BEG.
- Many already implemented engines [6], that can be used when developing new compilers.

5.1 CCMIR

The aim of the CCMIR is to be a general intermediate representation which can cover the needs of the source languages and the target architectures that CoSy is used for. The languages yet supported include C, Fortran, ML, Modula, Fortran90 and Predula Nouveau. The CCMIR is represented as an abstract syntax tree.

5.2 BEG

BEG, the back-end generator [8], is a tool for code generator construction. The idea is that the developer writes a relatively short rule-based description of the code generator and the target architecture. BEG uses these descriptions to automatically generate a code generator.

The code generator input is an intermediate language in tree form, where a program is represented as a sequence of expression trees. When BEG is used together with CoSy, the trees are on the CCMIR form.

5.2.1 BEG Tree Pattern Matching and Code Generation

A BEG rule maps a tree pattern to a machine instruction. Figure 3 shows a rule for the ADD instruction. If BEG finds a node Plus with two children of type reg somewhere in the input tree, then this subtree is replaced with a single leaf node, reg, and the instruction ADD x, y, z is emitted. Every rule also has an associated cost which is a measure of how much the emitted instruction will cost in execution time or possibly in memory usage.

The code generator first tries to find an optimal cover of the input tree. This means that adjacent nodes are grouped together, so that each group of nodes corresponds to the matching pattern. Then the tree can be traversed in postfix order and instructions specified in corresponding rules can be emitted.

A nonterminal is a leaf node that appears in the rules, but not in the input tree. In Figure 3, the reg nodes are nonterminals. A nonterminal specifies how an intermediate result is stored on the target machine.

A nonterminal graph has nonterminals as nodes, and chain rules as edges. A chain rule transforms a nonterminal to another.

BEG handles register spilling by a special nonterminal, BegHomeLoc. The code generator writer only has to specify spilling and reloading chain rules between register nonterminals and BegHomeLoc.

An instruction scheduler has also been introduced in the CoSy BEG. Scheduling is done after the covering rules have been chosen, and tries to reorder the rules to improve the code.

The original BEG did not perform any inter-expression optimizations. Register allocation was only made locally to each expression. Therefore a global register allocator was introduced.

6 Implementation

The vipcg system, which is shown in Figure 4, is split in two parts, where the first part translates the input to the CoSy intermediate representation CCMIR, and the second part is the code generator proper.

6.1 The Quadruple Language

The parallel parts of the DML-P output, which are the input to the code generator, are from now on called the quadruple language. Its formal syntax is shown
in Figure 5. A program consists of instructions and
variable declarations.

6.1.1 Variables

All instructions in the quadruple language operate on
vector variables. The vectors are assumed to be of the
same length as the RVIP processor array, and they are
also assumed to be distributed so that one element
is placed on each PE. The DML-P front-end is responsi-
bles for the splitting and distribution of larger vectors.

Currently only integer variables are handled by the
code generator. Integer bit length and sign are consid-
ered part of the integer variable. Floating point num-
bers are not supported since they are very seldom used
in the RVIP application area. (A future VIP design
called MVIP [12], will however support floating-point
arithmetic.)

All variables in the quadruple program are global
and must be declared before use. Unique numbers,
which are not addresses, are used as variable names.

Variable declarations use three arguments, first the
variable name, then the bit length and last a flag field
specifying whether the variable is signed or not and if
the variable is temporary, which means that the value,
One important difference between the two calls is that in the quadruple language, the arguments don't need to have equal precision. It is quite possible to add an unsigned seven bit integer with a signed five bit integer, and store the result in an unsigned 13 bit variable.

The elementwise operations work as their CVL counterparts. The RVIP rix_vuz and wio_vuz operators handle parallel input and output, and are not present in the original CVL.

All of the (unsegmented integer) scan operations are implemented:

\[
\begin{align*}
\text{add}_\text{suz} & \quad \text{and}_\text{suz} \\
\text{mul}_\text{suz} & \quad \text{ior}_\text{suz} \\
\text{xor}_\text{suz} & \quad \text{max}_\text{suz} \\
\text{min}_\text{suz} & \quad \text{xor}_\text{suz}
\end{align*}
\]

None of the reduce operations are implemented, which is motivated by the fact that there is no data flow from the PEs back to the front-end computer in an RVIP system, which implies that there is no place to store the scalar result.

For the same reason the vector-scalar operations are problematic. Therefore only

\[
\text{dis}_\text{vuz} \quad \text{distribute}
\]

is implemented with the restriction that the scalar has to be a constant.

The general CVL permute operations are not implemented, instead there are the following restricted versions:

\[
\begin{align*}
\text{shl}_\text{puz} & \quad \text{shift left} \\
\text{shr}_\text{puz} & \quad \text{shift right}
\end{align*}
\]

\text{shf}_\text{puz} is used to move values left or right in the RVIP processor array where x is the constant number of steps the vector is shifted. The shift operations are not circular, instead the value 0 is shifted in.

\[
\text{shf}_\text{puz} \quad \text{shift left or right}
\]

The shift operation shifts the bits within a single integer. It shifts right if x is negative, and left otherwise. The shift operation is not circular.

As the RVIP has microcode for shifting along the PE array, the interprocessor shift operations are very fast. A general permute operation is on the agenda, but will, given the restrictions of the hardware, be much slower. The shift operations will however still be meaningful as more efficient solutions for these very common special cases.

Here is an example of a complete code generator input (the comments are not part of the program):

\[
\begin{align*}
\text{var} & \quad \text{8 u} \\
\text{var} & \quad \text{2 u} \\
\text{var} & \quad \text{3 u} \\
\text{var} & \quad \text{10 7 s} \\
\text{var} & \quad \text{11 7 st} \\
\text{var} & \quad \text{12 10 s} \\
\text{add}_\text{vuz} & \quad \text{12 3} \\
\text{min}_\text{vuz} & \quad \text{1 1}
\end{align*}
\]

\[
\begin{align*}
\text{block} & \quad \text{start of new basic block} \\
\text{dis}_\text{vuz} & \quad \text{11 17} \\
\text{shl}_\text{puz} & \quad \text{10 10 1} \\
\text{mul}_\text{vuz} & \quad \text{12 10 11}
\end{align*}
\]

\[
\begin{align*}
v1 = v10 \\
v2 = v1 \\
v3 = v2 + v3 \\
v1 = \text{scan-min} \quad \text{v1}
\end{align*}
\]

6.2 Translation to CCMIR

As previously stated, the code generator must first convert the Preluca front-end output to proper CCMIR form. This is performed by a separate CoSy engine, vipfe.

As the quadruple language is very low level, translation to CCMIR is straightforward. A skeleton for a complete program in CCMIR is first built. Then one or more small CCMIR expression trees are built for each operation in the quadruple program and inserted in the skeleton. For every variable declaration, a new local variable is created, and if the variable has a type that hasn't been seen before, the new type is inserted in the tree.

6.2.1 Expressions

Every expression tree built from the operators has a \text{mirAssign} node as root due to the fact that every operation has a destination argument, a variable where to put the result.

Every operation uses variable names as arguments. The value of a variable is fetched in CCMIR with the \text{mirContent} node. If the arguments of an operation have different types, the types must be converted, as CCMIR demands equal types on every operator argument. \text{mirConvert} nodes must then be inserted according to the following algorithm: in operators with boolean results, for division, remainder and right-shift operators, the shortest argument is converted to the same type as the longest argument. For other operators with integer results, the arguments are converted to the result type.

Division, remainder and right-shift is handled differently because all bits in the operands need to be taken care of, even if just a few bits in the result are wanted. On the other hand in, for example, an addition, if the wanted sum is to be five bits, there is no need to look at more than five bits of the terms.

As an example, the CCMIR representation for the \text{add_vuz} operation is shown in Figure 6. In CCMIR, variables are referred to as \text{mirObjectAddr}.

The scan operation translation is more complicated than the other operations. It is implemented using the following algorithm: (\text{scan-plus} is used as example operator in the algorithm).

Every PE gets the value of \text{source} in their left neighbor PE. This value is then added to the own value of \text{source}, and the result is stored in a temporary variable, say \text{tempo}. That is:

\[
\text{tempo} := \text{source} + \text{ShiftRight} \text{source} \times \text{step}
\]

Then \text{tempo} is added to the value of \text{tempo} in the PE two step to the left:
example be viewed as one register, or as 32 independent one-bit registers. The latter is perhaps the most correct choice, as the RVIP is a one-bit architecture. It would also lead to a better use of the available registers, as several values could be held in the accumulators at the same time.

One-bit registers are of course not particularly useful when used separately. Whenever an integer longer than one bit is required, a sequence of registers has to be used. The register allocator that BEG generates lacks the possibility to allocate sequences of registers.

Some processor architectures however uses pairs, or groups of registers together as larger registers. BEG has a feature for this, the part relation: a virtual register is defined by telling that this register consists of two normal registers. Such virtual registers create new register classes. But this strategy isn't practical on the RVIP bit serial registers. Every bit length from 1 to 32 is possible, so there would be far too many register groups.

The model with one 32-bit accumulator was therefore chosen. The same model was also chosen for the other registers. Care has however been taken regarding the length of the integers in the registers. For example, operations which involve the accumulator don't have to loop as much as 32 laps, if the integers are shorter.

6.3.2 Nonterminals

The choice of nonterminals is crucial when BEG is used to construct a code generator. These are selected nonterminals, representing three of the possible storage types for RVIP:

- Acc: The value is stored in the accumulator. Two attributes are used to specify how to interpret the value; length is the number of bits that are relevant, and sign tells whether the value is signed or unsigned.

- Sreg: The value is stored in the shift register. It is defined and used in a similar way to the Acc nonterminal.

- Multreg: The value is in the parallel multiplicator register.

The reason why the registers are split into different nonterminals, is that they are used in different ways. The accumulator is used for ALU calculations, where many operations are only possible with the accumulator as argument, which is why it was put in a separate register class. The shift and multiregister also don't fit very well together, as the multiregister only can be used as destination in a busaction. The input/output registers also need special treatment, as they can't be used as normal data holders. A value in an input/output register can only be used during a short time interval, as it soon will be overwritten by new input. In the current code generator input/output is handled with special

\[
\begin{array}{cccccccc}
3 & 5 & 2 & 1 & 4 & 6 & 2 & 7 \\
+ & + & + & + & + & + & + & + \\
3 & 8 & 7 & 3 & 5 & 10 & 8 & 9 \\
+ & + & + & + & + & + & + & + \\
3 & 8 & 10 & 11 & 12 & 13 & 13 & 19 \\
+ & + & + & + & + & + & + & + \\
3 & 8 & 10 & 11 & 15 & 21 & 23 & 30 \\
\end{array}
\]

Figure 7: Scan algorithm.

tempo \leftarrow \text{tempo + ShiftRight tempo 2 steps}

And again, but this time four PEs away:

tempo \leftarrow \text{tempo + ShiftRight tempo 4 steps}

At last the result is stored in dest:

dest \leftarrow \text{tempo + ShiftRight tempo NPROC/2 steps}

An example of this algorithm is shown in Figure 7.

6.3 The BEG Model of RVIP

The most important part of our program is the code generator description (CGD). From this file, BEG builds source files for five engines, which work together as a code generator, translating CCMIR to RVIP assembly code.

Compared to other processors BEG has been used to generate back-ends for, RVIP is a very odd architecture. Therefore modeling RVIP in BEG has probably been trickier than for most more ordinary processors.

6.3.1 Bit Lengths

The fact that RVIP is bit serial leads to problems when RVIP registers are modeled. The accumulator could for
IR constructs, but this isn’t the ideal way of doing it, as the different input/output banks are not dealt with. The problem lies not only within the code generator, there has to be some sort of support in the compiler front-end too.

The RVIP architecture has absolute addressing only. This is represented by the following nonterminal:

**ram** The value is stored in ram memory. In addition to length and sign attributes, there is also an address attribute that tells where in memory the value is stored.

The normal way to perform an arithmetic operation, is to use the accumulator as the first operand, and another register or a memory access as the second operand, which makes another nonterminal necessary:

**bus** The value is in ram, or in the shift register.

There is no explicit compare instruction in RVIP. Instead subtraction is used, where the carry register tells the result. This is represented by

**Carry** The value is a boolean, stored in the carry register.

**Notcarry** The value is a boolean, and the inverted value is stored in the carry register.

The last nonterminal is used for spilling and reloading, see Section 5.2.1:

**BegHomeLoc** The value is spilled to ram memory.

### 6.3.3 Chain Rules

The chain rules in the nonterminal graph in Figure 8 were implemented to let the code generator easily choose between different nonterminals. Dashed arrows represent chain rules that don’t emit any instructions. A value in ram memory for example, can be represented equally well by the nonterminals ram and bus. Therefore a transition from ram to bus doesn’t need any instructions. The other chain rules need a move of the value from one storage location to another.

### 6.3.4 Output

The code generator outputs assembly language which is processed by the VIP assembler, pvlc. Here’s an example:

```c
#define ODR 2047
#define SIGN 2046

/* acc = ram */
loop(7 i);
  lacc = ram(0 i), acc(1 i)
endloop;
/* acc += ram */
ram(SIGN) = busconnect, acc(7)
loop(2 i);
  lacc = ram(SIGN), acc(8 i)
endloop;
```

Comments are inserted by the code generator for every matched rule that emits code.

### 6.3.5 Rule Construction

The ideal procedure when constructing a code generator with BEG, is to go through the target architecture instructions one by one, and to make a rule for each one. Each rule describes what the instruction does, and the emit part of the rule simply outputs an instruction. RVIP complicates this ideal scheme, though. For example, the `llacc` instruction loads the accumulator with the inverted value of the given operand. The problem is now to find a CCMIR expression that describes this instruction. Here this is easy, the `mirNot` operator has the same meaning as the `llacc` instruction. The `mirNot` operator has one operand, and the result of the operation also has to be specified, two nonterminals have to be chosen. The `llacc` operates on something that can be connected to the bus, apart from the accumulator, and the result of `llacc` is in the accumulator. Thus, the header of the rule describing `llacc` will look something like

```
RULE mirNot b:bus -> r:acc;
```

(b and r are only abbreviations used to refer to the arguments later in the emit part of the rule.) In the emit part, there are statements that output the actual `llacc` instruction. Using the dot tool, a program which simplifies C program output, this would have been very simple if the RVIP hadn’t been a bit serial architecture:

```c
EXIT

..llacc {* r}, {* b}
```

Instead, the attributes, length and sign, of the result nonterminal must first be calculated. To get as good code as possible, the length attribute of all non terminals is always tried to be minimized. The bit length of the result value of a `mirNot` operation is therefore calculated as the shortest of the following lengths:

```c
1. the length of the argument to `mirNot`.
2. the wanted length of the `mirNot` operator.

Whether an expression is signed or not, isn't changed by a `mirNot` operator, so the attribute sign of \( r \) can just be copied from \( b \).

If the calculated bit length is greater than two, a single instruction isn't enough, a loop instruction is necessary. The complete `mirNot`/`xacc` rule is then:

```c
RULE mirNot b:bus \rightarrow r:Acc; 
COST 2; 
EXIT 
{ 
  (\&) \{ \( b, a \) \}
  \( r \).length = MIN(MIN(b.length,\text{getLength}(mirNot.Type)),r.sign \. b.sign); 
  xloop(r.length);
  \( i \).xacc = \{ a\ b.at \} \{ (d\ b.addr)\{i\}, \text{acc}(i) \} 
  xendloop;
}
```

The first line in the emit part generates a comment, for example:

```c
/* xacc = !ram */
```

`xloop` and `xendloop` are macros that generate loop instructions if necessary.

Another example, the `mirPlus` operator, shows a more complex handling of the length and sign attributes. The header of the rule is simple:

```c
RULE mirPlus a:Acc b:bus \rightarrow r:Acc; 
TARGET a;
```

The `TARGET` command ensure that \( r \) and \( a \) refers to the same register. The bit length of \( r \) depends on the lengths of \( a \) and \( b \), the maximal possible length when adding two integers of length \( a.length \) and \( b.length \), is \( \text{max}(a.length, b.length) + 1 \). But if this value is greater than the length that actually was wanted by the `mirPlus` operator, it is unnecessary to compute that many bits. Therefore the `r.length` is calculated like this:

```c
r.length = MIN(MIN(a.length, b.length)+1, \text{getLength}(mirPlus.Type));
```

The result of an addition can only be an unsigned integer if both arguments are unsigned:

```c
r.sign = (a.sign \| b.sign);
```

Now, when both attributes of the result have values, the instructions can be emitted. But the instructions needed to add two integers depend on the lengths and types of both the arguments and the result, which gives many alternatives. To simplify a bit, first assume that all integer are unsigned. As different argument bit lengths are allowed, the addition has to be done in three stages. First, if the value in the accumulator isn't as long as the wanted value, the accumulator has to be padded with zeros:

```c
xloop(r.length - a.length);
  lacc = zero, acc({d a.length + 1}\{i\})
xendloop;
```

(Note that the `xloop` macro won't generate any instructions at all if `r.length` \( \leq a.length \).) Then the actual addition takes place:

```c
  .lbc = zero
  xloop(MIN(b.length, r.length));
  .add = \{ a\ b.at \} \{ (d b.addr)\{i\}, \text{acc}(i) \} 
xendloop;
```

If the wanted value is longer than the value in \( b \), the remaining bits can be calculated by adding the remaining accumulator bits with zero:

```c
  xloop(r.length - b.length);
  .add = zero, acc({d b.length + 1}\{i\})
xendloop;
```

When argument type is also considered, additional lines are needed to ensure that arguments that need to be extended are extended with zeros if they are unsigned, and with the correct sign if they are signed. The the complete `mirPlus` code emission section therefore looks like this:

```c
if (r.length > a.length) {
  if (a.sign) {
    .ram(SIG) = busconnect, acc({d a.length})
  }
  xloop(r.length - a.length);
  if (a.sign) {
    .lacc = zero, acc({d a.length + 1}\{i\})
  } else {
    .lacc = ram(SIG), acc({d a.length + 1}\{i\})
  }
xendloop;
}
```

```c
  .lbc = zero
  xloop(MIN(b.length, r.length));
  .add = \{ a\ b.at \} \{ (d b.addr)\{i\}, \text{acc}(i) \} 
xendloop;
```

```c
  xloop(r.length - b.length);
  if (b.sign) {
    .add = zero, acc({d b.length + 1}\{i\})
  } else {
    .add = \{ a\ b.at \} \{ (d b.addr + b.length - 1)\{i\}, \text{acc}(d b.length + 1)\{i\}
  }
xendloop;
```

The other DCMIR operations need similar solutions, with special care taken to the calculation of bit lengths and types. The following heuristic is used:

The front-end engine inserts `mirConvert` nodes, when arguments have different bit lengths or types.

BEG rules that handle type conversion first simply remove the `mirConvert` node. The bit length of the child of the `mirConvert` is set to the bit length of the `mirConvert`, if this length is less than its original
length, otherwise its length is unchanged. This means that if a nonterminal is made shorter, the length attribute is changed. The length attribute will therefore always indicate how many bits in the nonterminal that are significant.

The other CCMIR operator rules are constructed with respect to the length and sign attributes of the operands. The rules will set length and sign of the result nonterminal to a minimal value (where a unsigned type is considered less than a signed).

7 Measurements

Some measurements were done on performance on the data-parallel code generated by the produced compilers. The RADAR program which was used in the measurements consists of a fixed sequence of digital filter and signal processing steps which process an incoming radar signal. The program run time does not depend on the data. The processing time for the signal generated by a radar sweep was measured on an RVIP simulator with 64 PEs where the number of clock cycles spent by the programs could be measured.

The hand-written program consumed 969482 clock cycles on the simulator. With a 50 MHz clock frequency, this corresponds to an execution time of 19.4 ms.

Due to current limitations of the DML-P frontend, a single 16-bit size was used for all integer variables. Also, a number of optimizations, including copy propagation, are not yet implemented in the DML-P system. Using a compiler generated by this system, the compiled RADAR program consumed 5518555 cycles, which translates to 106.3 ms.

The ratio between the execution times in this initial measurement is therefore a factor of about 5.5. Measurements have also been done on a program where copy propagation and elimination of resulting unused variables was applied semi-manually to the data-parallel code which was produced by the DML-P generated front end. This reduced the execution time of the data-parallel code another 50%, which resulted in a ratio of 2.8.

8 Conclusions

The code generator vipcg was successfully implemented as described in Section 6. The radar application program compiles to code which is 2.8 times slower than an equivalent handwritten program in RVIP microcode-like assembly language. As mentioned before, even though this is slower it was considered much better than expected by industry people. The code generator handles all of the operations in the input language specified in Figure 5, not only the ones necessary for the radar program.

The results and suggestions for improvements are related to BEG version 1.11 and CoSy framework version 106.1, both released spring 1995. The situation one year later is that the CoSy framework has improved substantially by including a number of optimizing transformation engines not previously available to us. By making use of these engines coupled to an enhancement of BEG to allow the specification of register sequences that would simplify rules and enable using its instruction scheduling capability, we believe that a substantial improvement in code quality would occur, getting close to the quality of hand coded assembly. Finally we present a few concrete suggestions for future improvements on the RVIP code generator:

Some problems with the global register allocator have to be solved, by enhancing the BEG tool.

The use of COST in the BEG rules that describe the current code generator isn’t very accurate, care should be taken to make the length of the operands available in the rule.

A different modeling approach for the RVIP registers is to use some or all of the memory as registers. This would move the memory allocation from the frontend engines to the register allocator. In the current code generator the global register allocator is poorly utilised, as it often has very few registers to choose from. However, there could be problems because the current BEG can’t allocate a sequence of registers, one register nonterminal represents several one-bit registers. The memory would therefore have to be segmented for example, a number of 16-bit registers. This would lead to a probably unacceptable waste of memory for smaller values.

Even if the memory-as-register idea is hard to implement, some alternative modeling of the memory could result in better code. When a mirassign is done, values to be stored in RAM always have to be in registers, which isn’t always the case. If RAM had been more like a register nonterminal, rules like mirIntConst -> ram could have been used to store a constant directly in ram. Another similar idea is to make a constant nonterminal, which could be used as a register, so that operations with constants can be used without having to first store the constant in a register.

Despite the quite sophisticated bit handling in the current solution, there are possible improvements. The registers in the RVIP model have different lengths, for example a 20-bit integer will fit in the accumulator, but not in the shift register. There are no checks in the BEG rules for this, so there is a risk of erroneous code, when larger programs with variables longer than 16 bits are compiled. A BEG rule may contain a conditional statement, but this may not include any attributes of nonterminals. There are however a special type of attributes, condition attributes, that may be used. The bit lengths in the current model are stored in normal nonterminal attributes, so a solution may be to transfer them to conditional attributes. Care must then be taken to the certain restrictions on the use of conditional attributes.
BEG code generators provide scheduling, but this is currently not used in vipcg, as RVIP has no pipelining. An idea is however to use it for making better use of fine-grained parallelism by combining parts of instructions into correct RVIP microcode.

As our work proceeded, it became apparent that the RVIP register model doesn't fit well with the current version of BEG, too much work is spent squeezing RVIP into a model suitable for BEG. The most serious problem is that the registers can't be modeled accurately, that is, one single machine register can't be seen as a BEG register. This also means that the fundamental rule that one machine instruction should be modeled as one BEG rule breaks. All the tricks that are currently needed makes it hard to utilize the full functionality of BEG. However, if BEG is extended to handle allocation of register sequences according to Section 6.3.1, most of these problems would go away.

References


Parallel Models
Compile-time Optimization and the SPC Parallel Programming Model

Arjan J.C. van Gemund
http://dutepp0.et.tudelft.nl/~gemund

Department of Electrical Engineering
Delft University of Technology
P.O.Box 5031, NL-2600 GA Delft, The Netherlands

Abstract
We present a framework for the automatic compilation of efficient code that evaluates logic decisions concerning optimizations such as pipelining and data (re)mapping. Depending on program parametrisation this optimization logic can either be evaluated at compile-time or executed at runtime as an integral part of the object code at negligible overhead. This framework comes with the use of a new parallel programming model, called SPC, that provides a restricted coordination mechanism based on the use of high-level task synchronization operators. By imposing these synchronization restrictions, efficient, symbolic cost estimation is possible that forms the basis for the optimization logic. We present the SPC programming model and demonstrate its use by deriving a number of optimisations for a line relaxation algorithm on a distributed-memory machine.

1 Introduction
The design of parallel algorithms as well as their compilation essentially constitutes a performance optimization problem. Depending on the target machine architecture, the degrees of freedom involved entail optimization decisions such as computation and/or communication pipelining, loop transformation, as well as the choice of data layout, especially in distributed-memory machines. Thus far, optimization decisions are either left to programmer annotation (e.g., data distribution directives, possibly aided by specific tools [18]), or are hard-wired in terms of heuristic, compile-time transformation schemes that are partially based on the use of cost estimation [25]. In order to achieve the ultimate goal of truly portable, yet efficient, parallel program design, an optimization framework is needed that is based on a machine model that associates time cost models for the entire spectrum of functions that are supported. The generic "model of compilation" is such that, depending on the available cost models of the target machine, each optimization is decided upon based on a direct cost comparison between the alternative instructions that are applicable. If possible, this decision is evaluated at compile-time. On default, however, the optimization logic is compiled as part of the object code. A trivial example is vectorization, where, assuming the availability of the appropriate vector instruction (e.g., cost < ∞), the resulting optimization logic constitutes a simple test whether the vector length exceeds a threshold in order to sufficiently amortize the extra startup overhead. This optimization logic can be derived from a comparison between the scalar implementation cost (scalar loop) and the vector implementation.

An important feature of this approach is that optimization is achieved merely based on cost functions, rather than using optimization algorithms in which machine characteristics are hard-wired. One of the essential requirements for this generic compilation scheme to be efficient, however, is that the cost of the optimization logic itself be minimal in view of the call frequencies involved in optimizing a whole program. Furthermore, the optimization gains must outweigh the overhead induced by optimization logic that has to be executed at run-time. For example, the generic cost models involved in the vectorization decision would be the time estimation expressions for the scalar loop and the vector code. Direct comparison of both estimates could be too costly. Decision code based on comparison between the vector length and the machine-dependent threshold, however, is not readily available in a generic scheme, since it involves an optimization-specific derivative of the general time cost comparison scheme. We therefore propose a model of compilation based on the concept of meta optimization. The optimization logic, initially generated in terms of the machine cost models, is optimized by symbolic reduction at "meta-optimization-time", i.e., at compile-time but before the actual optimization process is invoked. For instance, let \( T_L = N \tau_L \) denote the time cost for scalar implementation of an \( N \) element vector loop (\( \tau_L \) time cost of scalar floating point operation). Let \( T_V = \tau_V + N \tau_L \) denote the familiar, linear time cost estimate of the alternative
vector code ($r$, startup overhead, $1/r$, bandwidth). Instead of evaluating the expression $T_v < T_s$ in the course of program optimization; at meta-optimization-time the expression is symbolically reduced to $N' \geq \lceil T_t / (r_t - r_c) \rceil$ of which the RHS is further reduced to a constant $N^*$. The resulting comparison $N' \geq N^*$ is used as a basis for the program optimization scheme mentioned earlier. Even if $N$ is not known at compile-time, in cases where there is a distinct probability that $N$ is often significantly different from $N^*$, it is clearly worth while to compile the above test (a mere integer comparison) as part of the vector code. Although the gains at run-time are clear, the concept of meta-optimization may still induce considerable costs at compile-time. We assume, however, that large parts of the optimization framework can be meta-optimized at "meta-compile-time", i.e., the time at which the compiler itself is compiled into a machine-specific instance.

Although many questions need to be answered before the above compilation model can offer a useful framework for automatic optimization, it is clear that a crucial requirement is that the cost models must have an analytic, closed form in order to fit the algebraic reduction concept that underlies the optimization approach. At the same time, however, the analytic models must have a low symbolic complexity, and yet be reliable in order to generate the correct optimizations.

On the one hand, abstract models of parallel computation have been developed that provide analytical cost (complexity) estimates that provide an adequate insight into the performance trade-offs of various design choices [7, 24]. However, their high level of abstraction from actual algorithms and machines introduces a modeling problem in itself that cannot be handled in the context of an automatic compile-time optimization scheme. On the other hand, more detailed static cost prediction methods exist [3, 26]. However, either the underlying analysis technique (and the associated parameter space) is targeted to a particular machine organization or programming model, or the technique is not designed to produce reliable estimates in extreme points in the entire parameter space. (Erratic parameter solutions may be temporarily generated in the course of the optimization process). Consequently, a framework for automatic optimization based on the extensive use of analytic cost modeling has not yet emerged.

The heart of the problem with respect to analytic cost modeling in parallel system optimization (or any form of parallel program analysis for that matter) is the choice of parallel programming model. Traditionally, parallel programming models are focused on programming power, i.e., the ability to express the intended parallel computation most faithfully (often entailing "synchronization spaghetti"), rather than analyzability, which in this context means the ability to predict the parallel computation's (temporal) behavior in terms of closed-form, analytic models with sufficient cost/performance. We feel that the current trend in the development of parallel programming models, such as message-passing, data parallelism, but also more recent developments aimed at providing task parallelism (or coordination in general), may not sufficiently reflect the importance of analyzability, especially with regard to the possibilities for fully automatic, re-targetable compilation.

In this paper we present the following contributions:

- We present a new, high-level, shared data programming model, called SPC, that is specifically aimed at providing analyzability, while preserving programming power. By imposing a specific restriction in the synchronizations that are provided, symbolic optimization logic can be compiled, that is highly efficient, yet sufficiently accurate.

- We show how the SPC model captures the essential parallelism in parallel applications, we conjecture on its generalizability, and demonstrate its use to derive program optimizations. After an introductory example involving vectorization, we show how the optimum implementation of a line relaxation kernel on a distributed-memory machine is derived.

The "SP" prefix in SPC applies to the fact that the algorithm (and machine) is to be described in terms of a series-parallel structured computation, which implies structure with respect to the condition synchronization (CS [2]) patterns that are possible (only SP task graphs). The "C" term refers to the structured use of mutual exclusion (ME [2]) in contention for (processing or data) resources, which, combined with a material-oriented [19] programming paradigm, we coin contention programming. Although the concept of material-oriented simulation as well as the use of structured operators for CS (e.g., fork/join programming) and ME (e.g., monitors) are not new, to the best of our knowledge the combination of both concepts to create a highly structured model of parallel computation has not been introduced before.

\[1\] The original terminology material-oriented modeling, and its dual, machine-oriented modeling, stem from the domain of simulation of, e.g., plant production lines. We feel that the application of material-oriented modeling in the distinct domain of parallel programming justifies using the specific name "contention programming". Unlike its dual, in the contention modeling approach the data computations flow (i.e., the algorithm) is modeled by active processes while the machine is modeled as a passive collection of resources, being shared by contending processes.
While the question whether SPC delivers sufficient programming power is clearly at the heart of our approach, in this paper we will primarily concentrate on the analyzability aspect of the SPC approach, by demonstrating its use in the intended model of compilation as discussed earlier. In Section 2 we present the SPC programming model along with some examples as well as a rationale. In Section 3 we show how SPC provides a framework for the automatic compilation of optimization logic. In particular, we describe a case study in which we consider a number of optimizations for a line relaxation algorithm on a distributed-memory machine. In Section 4, we review related work in the field of parallel programming models in order to place our approach into perspective. Finally, in Section 5 we summarize our work.

2 The SPC programming model

2.1 Language

SPC is not a full fledged programming language which would very much require the specification of the data computation. It specifies a parallel programming paradigm with respect to the coordination of the program’s (synchronization) control flow. As such, SPC stands for a coordination paradigm.

To facilitate the presentation, however, we shall loosely introduce a simple language implementation as a description vehicle. For this language we borrow a number of programming concepts used in the performance modeling language PAMELA [12], that is well-suited to support the SPC approach, especially as we will only be concentrating on the control-flow aspects of the parallel computation, not the data processing itself. Apart from some syntactic sugar, the syntax used for SPC conforms to the PAMELA language implementation [20]. Although PAMELA has indeed been developed with the SPC paradigm in mind, note that the SPC approach represents a distinct model of parallel computation that is independent of the underlying description language. Furthermore, being an full-fledged performance simulation language, PAMELA features much more operators than the few we will use for the SPC programming model.

2.1.1 Processes

Like PAMELA, SPC is based on a process-algebraic specification model. This implies that SPC programs are a set of functional process equations. Programming, i.e., applying compositions is by specifying process equations using a straightforward substitution mechanism. In order to ensure the correct binding, (,)’s can be used to delimit compound process expressions. Apart from data types needed to express numeric computations, the main data type in SPC is the process.

Basic data computations are expressed in terms of processes. For example, the equation

\[ \text{update}(i) = \ldots \text{C} \ldots \{ A[i] = A[i-1] + A[i+1]; \} \]

describes a process update(i) where the \ldots \text{C} \ldots \{ \ldots \} is an inlining facility (as implemented in PAMELA) to provide an interface with actual data computation host languages (C in this case).

An SPC program consists of a set of process equations that through substitution constitutes one parallel process expression. By convention, the expression tree is rooted by a special process called main, which represents the overall program.

The process abstract data type comes with the following operators:

- sequential composition
  A sequential composition of two tasks is described by the infix operator \( ; \); like in \text{task\_1} \; \text{task\_2}. Sequential replications ("loops") are described by the sequential reduction operator seq defined as \( \text{seq} (1 = 1, n) \text{task}(i) = \text{task}(1) \; \text{task}(2) \; \ldots \; \text{task}(N) \). Note that this definition provides a basic form of CS.

- parallel composition
  A parallel composition of two tasks is described by the infix operator || like in \text{task\_1} \; | | \; \text{task\_2}. The parallel operator has a fork/join semantics which implies a mutual barrier synchronization at the finish of the parallel section for each task involved. Note that this implies a structured form of CS. Parallel replications ("loops") are described by the parallel reduction operator par which

---

\(^2\) Although we have already gained compelling evidence suggesting that the a priori loss of inherent parallelism due to the SPC synchronization structure is indeed very limited [9] this is still subject of ongoing research. However, the important conjecture motivating this research is presented in Section 2.4.
is defined as \( \text{par} \ (i = 1, N) \ \text{task}(i) = \text{task}(1) \ || \ \text{task}(2) \ || \ldots \ || \ \text{task}(N) \). Note that this definition again provides a structured form of CS.

- **conditional composition**
  A conditional process is specified by the if operator that has an optional else part, like in
  
  \[ \text{if} \ (c) \ \text{task\_1 else task\_2} \]

Finally, for Turing completeness SPC also includes a while construct, which, however, we do not consider in our cost calculus.

### 2.1.2 Resources

A central philosophy within SPC is that parallel computations are expressed in terms of processes (the computations) and resources (the computation providers), the latter introducing (dynamic) limitations with respect to the actual parallelism in the system. While the process abstract data type provides a means to specify both parallelism and CS, the resource abstract data type provides the means to impose ME. A process that is to be executed under ME is assigned (mapped) to a resource as in

\[ f, g \rightarrow \text{some\_resource} \]

where the \( \rightarrow \) operator denotes the mapping of processes \( f \) and \( g \) to processor \( \text{some\_resource} \). Here \( f \) and \( g \) are executed under ME as a result of the fact that they are mapped onto the same resource (e.g., a critical section or the same processor). Hence, in the program

\[
\begin{align*}
\text{main} = f \ || \ g \\
, f, g \rightarrow \text{some\_resource}
\end{align*}
\]

the execution of \( f \) and \( g \) are serialized. Note, that again synchronization is structured, which rules out the possibility of deadlock. A resource can either be logical, e.g., model a critical section (software server), or physical, e.g., model a CPU, memory bank, communication bus, etc. The scheduling policy (we will consider in this paper) associated with resources is simply FCFS with non-deterministic, fair conflict arbitration (but other scheduling disciplines are possible [13]).

The notion of resources is universal in SPC, i.e., in principle, each process in SPC is always mapped onto a resource. The underlying concept is that a processor must always execute in the context of some resource (it must cost cycles to at least one of more resources). However, the declaration is not required when a process is mapped to an exclusive resource (i.e., when there is no contention anyway). Hence, many SPC models will not require the explicit use of the resource (limitation) mechanism.

In summary, the SPC model of coordination is based on only a few constructs, i.e., the ;; ||, and if process composition operators and the \( \rightarrow \) resource assignment. Note that within SPC the resource concept is only used to express ME in order to express dynamic synchronization between program level components. Although possible, it is not intended to direct the actual mapping of processes onto physical processors or other machine resources as in our aim to study automatic program optimization the program description is meant to be completely machine-independent. The actual implementations involving the mapping onto physical machine resources are only introduced in the course of deriving program optimizations.

### 2.2 Examples

In the following we demonstrate the programming power of SPC by discussing four examples, each of them being representative for a large class of computations. In general, each example may be viewed as a “benchmark” for programming power.

#### 2.2.1 Data parallelism

Being an SP programming model, clearly all data parallel programs (as well as task parallel programs) can be simply expressed in SPC. For example, the following process equations describe a parallel section where each process \( \text{proc} \) executes a sequence of operations \( \text{op}(i,j) \) on some data structure \( \text{imag}[i] \) (e.g., some image computation).

\[
\begin{align*}
\text{main} = & \ \text{par} \ (i = 1, N) \ \text{proc}(i) \\
\text{proc}(i) = & \ \text{seq} \ (j = 1, N) \ \text{op}(i,j) \\
\text{op}(i,j) = & \ \ldots \ \{ \ f(j, \text{imag}[i]); \ \} 
\end{align*}
\]
Being a process-algebraic language, the program could also have been described by the single expression

\[
\text{main} = \text{par (i = 1, N) seq (j = 1, M) op(i,j) ! op(i,j) = .c_. ( . . . )}
\]

which can optionally be spread across multiple lines to increase (nesting) readability. SPC also allows the expression of nested data parallelism, such as used in the following Divide-and-Conquer example.

\[
\text{main} = \text{par (i = 1, N) par (j = 1, M) f(i,j) ! f(i,j) = .c_. ( . . . )}
\]

Note that SPC allows dynamic parallelism as the value of \(N\) and \(M\) may be controlled at run-time. Thus irregular problems can be expressed naturally.

2.2.2 Pipelining

As the programming power of the popular data parallel programming models is only limited within the SP computation class, there is currently a great deal of interest into extending this programming model to allow the expression of other forms of parallelism (see Section 4 on related work). In the discussions software pipelining is often used as motivating example.

Software pipelining represents an important class of computations that are inherently non-SP (NSP) structured. The pipelining example shows an essential difference in the SPC contention programming approach compared to current approaches to pipelining. In the SPC model, pipelining is still expressed in terms of data parallelism (which in fact is), yet being constrained in terms of the available computing resources. The crucial difference in the SPC approach is that we consider current programming solutions as overspecifications with regard to the rigid synchronization structure that they specify\(^3\).

Recall the original data parallel operation

\[
\text{main} = \text{par (i = 1, N) seq (j = 1, M) op(i,j)}
\]

Now consider a pipelined scheme restricting the parallelism to only \(M\) processing units (note that the data parallel example has a parallelism of \(N\)). Typically, the solution is programmed through the use of a message-oriented model in which the subsequent occupation of each data frame in each processing stage is explicitly specified. In contrast, our SPC solution is given by

\[
\text{main} = \text{par (i = 1, N) seq (j = 1, M) op(i,j) \rightarrow unit(j)}
\]

In this approach we simply specify the inherent data parallelism in the algorithm in terms of the construct \(\text{par (i = 1, N)}\) just like before. However, we introduce the constraint that each unit can only process one frame at the time. This is specified by mapping each individual function \(\text{op(i,j)}\) to an exclusive resource called \(\text{unit(j)}\). The pipeline behavior simply follows from the ME induced by this resource assignment. Note that this is actually a very natural way of programming which is also very portable. In essence, the main equation still specifies the intended algorithm, which is essentially data parallel\(^4\). The pipeline implementation simply follows from the fact that the operations are mapped to only \(M\) processing resources, which, by nature, impose ME with respect to their service. Thus, the synchronization mechanism we use is based on the fact that each data frame actively “contends” for the available computing resources, hence, the name “contention programming”, a concept that lies at the heart of the SPC programming model.

Note, that we left the order of frames undetermined, i.e., the program does not specify that \(\text{imag(1)}\) be processed first, etc. It only specifies that the first frame that is available in the stream be processed first, that there should only be one frame processed at each unit simultaneously, and that there be no race conditions (which is guaranteed by the FIFO resource scheduling mechanism). Indeed, the pipelining example illustrates that usually there is no need to specify such an additional ordering constraint. In fact, the traditional way of expressing pipeline behavior is overspecification. The omission of this additional ordering constraint allows for the data parallel way of specifying. Note that the SPC alternative produces an SP computation, in contrast to the original specification, which produces an NSP graph. As

\(^3\)Note, that we refer to software (asynchronous) pipelines, not systolic (hardware) pipelines which operate under a synchronous scheme. The synchronous pipeline version can be adequately modeled in terms of a data parallel scheme.

\(^4\)Note therefore that it is not an essential requirement for a programming model to be able to express pipelining in an explicit way in order to guarantee sufficient programming power. The intended pipelining could also automatically follow from the implementation of the data parallel operation when \(M\) computing resources happen to be available rather than \(N\). However, as mentioned earlier, many feel that including the ability of explicit specification of pipelining in a coordination model is preferable than to rely on data parallel operators alone.
mentioned earlier, it is the SP structure that allows for the symbolic, compile-time analyzability, and all the consequent advantages with respect to optimizability.

Also note, that the SPC solution involves the use of \( N \) processes instead of \( M \) where \( N \) can be quite large (or infinite). As mentioned earlier, we assume the existence of an (automatic) mapping layer that supports this "process virtualization" in order to provide a truly portable programming model. The allocation of only \( M \) physical processes/processors in the above example would be an optimization matter.

2.2.3 Non-determinism

Although in the above example we employ the mechanism of ME in order to express pipelining, many computations such as pipelining have an inherently static synchronization structure. However, as many problems involve dynamic synchronization, in order to provide sufficient programming power a parallel programming model must be able to express non-determinism. In the SPC approach, non-determinism is provided through the ME mechanism, where under exactly simultaneous resource access the conflict arbitration order is not determined.

A classical problem requiring the use of non-determinism is the Machine Repair Model (MRM) where \( N \) clients simultaneously execute a loop (of \( L \) iterations) in which each client performs some local computation \( do\_compute \), followed by a \( get\_service \) access call to a server\(^5\). Conventional solutions are based on paradigms that are either procedure-oriented (shared variables, using semaphores) or message-oriented (distributed variables, using message-passing) [2]. The SPC approach is close to the procedure-oriented paradigm be it that the \( P \) and \( V \) semaphore operations are combined in terms of one construct to achieve ME (operation-oriented, in terms of [2]). The SPC program version is simply given by

\[
\text{main} = \text{par} \ (i = 1, N) \\
\qquad \text{seq} \ (j = 1, L) \{ \\
\qquad \quad \text{do\_compute} \ ; \ \text{get\_service} \\
\qquad \} \\
\text{get\_service} \rightarrow \text{server}
\]

In general, systems that include non-deterministic synchronizations are not amenable to analytic cost modeling. The structured, higher-order form in which non-determinism is provided in the SPC programming model, however, allows for an analytic estimation technique that is both efficient, as well as sufficiently accurate. This aspect will be elaborated upon when we present the rationale behind our approach.

2.2.4 Arithmetic Reduction

Another computational form which requires the use of a dynamic control mechanism is reduction. Like parallelism, the concept of reduction is central in many computations. For example, consider the summation \( y = \sum_{i=1}^{N} f(i) \) where \( f \) represents some computation. A specification in a language merely based on static CS would seriously decrease the inherent parallelism as in the following solution\(^6\),

\[
\text{main} = \text{par} \ (i = 1, N) \ \text{term}(i) \ \\
\quad \text{seq} \ (i = 1, N) \ \text{sum}(i) \ \\
\text{term}(i) \ ; \ \text{sum}(i) \\
\text{sum}(i) \rightarrow \text{critical}
\]

The problem with this solution is that the actual summation does not start unless \textit{all} terms are computed. Thus much inherent parallelism may be lost (i.e., potential overlap between the operations \( += \) and \( f() \)). We introduce dynamic control flow by exploiting ME around the reduction operation itself according to

\[
\text{main} = \text{par} \ (i = 1, N) \{ \\
\quad \text{term}(i) \ ; \ \text{sum}(i) \\
\text{sum}(i) \rightarrow \text{critical} \}
\]

which produces the dynamic data flow scheduling as intended. Again, we avoid the (index order) over-specification which would result from a solution based on static CS.

\(^5\)In the original machine repair model this was a repair unit, where machines were submitted for repair. In the context of this paper one may also think of compute of data services.

\(^6\)Chosen for simplicity, a recursive doubling solution falls in the Divide-and-Conquer class. However, even recursive doubling, being a static solution, does not solve this fundamental problem, it only minimizes it.
2.3 Cost Estimation

In this section we briefly describe the analytic cost estimation process associated with SPC programs which is based on the use of PAMELA. As mentioned earlier, PAMELA is almost equivalent to the above SPC language, be it that it is a performance modeling formalism. As such it incorporates the notion of (simulated) time. While the process composition constructs are exactly similar to SPC, basic computations are modeled by the primitive time delay process \( \text{use}(r, \tau) \), where \( \tau \) denotes a resource that is acquired during a period of time determined by \( \tau \) which denotes the work load. Apart from modeling the time delay associated with computations, the use operation also provides the basic mechanism to express ME as only a limited number of processes can execute the statement simultaneously (typically, only one process\(^7\), specifyable through the multiplicity of \( r \)). Hence, the total delay due to the use statement can be longer due to additional queuing delay. In cases where there is guaranteed absence of ME the use statement is simply written as \( \text{delay}(\tau) \) which delays the process by the exact amount of \( \tau \). A detailed description of the PAMELA language appears elsewhere [12, 13].

In terms of PAMELA, the semantics of an SPC process - resource mapping such as

\[
\text{main} = \ldots \text{ process } \ldots \\
\text{ process } \rightarrow \text{ resource}
\]

is given by the PAMELA model (by convention denoted by \( L \), similar to main in SPC)

\[
L = \ldots \text{ use} (\text{resource}, \tau_{\text{process}}) \ldots 
\]

where \( \tau_{\text{process}} \) denotes the time cost estimate of process. Note, that SPC programs are captured in the PAMELA domain using only three constructs, i.e., \( \text{par} \), \( \text{seq} \), and \( \text{use} \).

The time cost of an SPC expression is determined by mapping the corresponding PAMELA expression to a closed-form, symbolic expression in the time domain. This linear-time, automatic transformation process is described elsewhere [13] where it is also shown that on average the cost estimation error is typically limited within tens of percents, regardless the size and shape of the model. This property forms a sufficient basis for automatic optimization through comparative cost modeling.

As a simple example, we describe the cost estimation procedure for the MRM. The SPC program

\[
\text{main} = \text{par} \ (i = 1, N) \ \text{seq} \ (j = 1, L) \ \{ \ \text{do\_compute} ; \ \text{get\_service} \} \\
\text{get\_service} \rightarrow \text{server}
\]

is automatically translated to the PAMELA model

\[
L = \text{par} \ (i = 1, N) \ \text{seq} \ (j = 1, L) \ \{ \ \text{delay}(\tau_i) ; \ \text{use}(s, \tau_j) \}
\]

where \( \tau_i \) and \( \tau_j \) denote the work loads associated with \( \text{do\_compute} \) and \( \text{get\_service} \), respectively, and \( s \) represents the resource \( \text{server} \). \( L \) is automatically mapped to the symbolic time cost estimate

\[
T = \max (L(\tau_i + \tau_j), N\tau_i)
\]

which is quite close to the actual time cost [13]. The cost estimation procedure is based on a critical path analysis combined with an approximation of the serialization that may occur due to ME [13].

2.4 Rationale

The choice of programming model involves a trade-off between programming power and analyzability. One the one hand, sufficient programming power is needed in order to minimize the loss of parallelism when capturing the problem in terms of the programming model. On the other hand, programming power should be limited in order to retain the analyzability needed for adequate optimization.

With respect to analyzability, the SP synchronization structure imposed by the SPC model is essential in order to derive symbolic estimates that are at least closed-form expressions [13]. With respect to non-determinism, in [13] it is also shown that programming models based on the use of a low-level non-deterministic "choice" operator (selective communications operator, CSP: \( \text{C} \)) are not amenable to a symbolic cost estimation scheme. This motivates the choice for an operation-oriented rather than a message-oriented paradigm, which in our view is detrimental to analyzability (see [13] for details).

With respect to programming power, due to the choice of high-level synchronization constructs, programming power will be less than, e.g., a message-passing formalism. Hence, the crucial question becomes how much inherent parallelism we sacrifice at maximum by imposing the SPC model. The essential rationale for SPC with regard to programming power is stated in terms of the following conjecture.

\(^7\)As in SPC, in PAMELA each process is assumed to consume cycles from one (or more) resources.
Conjecture 1 Let $G$ be a general parallel computation without any restrictions with respect to synchronization structure (e.g., have NSP structure, possibly including the use of non-determinism). Let $T_G$ denote the critical path of $G$. Let $G'$ be an SPC version that does not violate the intended computation semantics of $G$ and which is closest to $G$ in terms of its critical path $T_{G'}$. Then we conjecture that it holds

$$\frac{T_{G'}}{T_G} \leq 2$$

**Argument:** The argument breaks down in the following (interrelated) parts:

- **Equivalence.** As illustrated by the data parallel examples many practical computations can be stated in terms of parallel algorithms that are merely composed of (possibly nested) parallel sections and reductions. Clearly, this class of problems simply falls within SPC. In addition, as illustrated by the MRM and the summation example, some form of non-determinism is needed in order to adequately describe dynamic problems in order to avoid a priori loss of parallelism. Again, the class represented by these examples falls within SPC.

- **Overspecification.** Clearly, there exist computations that are expressed in terms of an NSP synchronization structure. However, as illustrated by the software pipelining example, in a number of important cases the NSP form is due to needless, static overspecification in which the SPC model provides a dynamic alternative that adequately expresses the intended computation (i.e., as also discussed in the MRM and the summation example, the order of parallel processing is better determined dynamically, rather than being statically scheduled in advance).

- **Approximation.** Although the above categories show that many computations can be expressed in SPC without any loss of parallelism, many computations remain that are justifiably NSP structured. For this remaining class of NSP computations we argue that $T_{G'} \leq 2T_G$. Note that this bound does not depend on the number of nodes. For example, large parallelism/reduction graphs such as occurring in LU factorization can be converted to SPC form (i.e., the familiar SP form), again well within a factor 2 loss of potential parallelism, while the average loss is typically much less [9].

Clearly, the notion of "intended computation" is crucial with regard to the truth of the conjecture. What we mean is that, compared to the actual computation, the intended computation can always be expressed in SPC under our factor 2 conjecture. For instance, the NSP pipeline violates the factor 2 conjecture when compared to a static SP solution. However, the actually intended computation can be perfectly expressed in terms of an alternative, dynamic SPC model, without any loss of parallelism at all (for purposely systolic pipelines the factor 2 conjecture also holds, by the way). □

The above conjecture has been a major inspiration for the SPC approach. In order to substantiate this conjecture, research aimed at deriving proofs as well as supporting empirical evidence has recently started. Preliminary empirical data, based on the results of a (polynomial-time) algorithm converting NSP graphs to SP versions [9] indeed provide compelling evidence in support of the above conjecture. However, this area is still subject of active research.

3 Compiling Optimizations

In this section we demonstrate how SPC is used for the compilation of optimization logic. First, we introduce the technique by discussing vectorization. Next, we look at a more realistic example involving an ADI line relaxation algorithm on a distributed-memory machine. In the examples we will use SPC as an intermediate formalism, in terms of which the optimizations will be described. The associated performance models are derived using the PAMELA cost modeling calculus.

3.1 Vectorization

Consider the following data parallel computation, i.e.,

```
par (i = 1, N) mult(i)
```

where the `mult` task represents the actual data computation (ignoring data traffic). Depending on the target machine architecture, there may be various implementations possible. As the above SPC description
is independent of the underlying architecture, we can consider these mappings just in terms of defining the implementation of \texttt{mult} depending on the instructions available on the (virtual) machine.

Scalar processing simply implies the following mapping

\[
\text{\texttt{mult}(i) -> \texttt{scalar\_unit}}
\]

Thus all \(N\) \texttt{mult} operations are implemented by a single computational processor \texttt{scalar\_unit}. The \texttt{PAMELA} model of the parallel computation is given by

\[
L_s = \text{par} \ (i = 1, N) \ \text{use}(s, \tau_s)
\]

where \(s\) represents the scalar processing unit and \(\tau_s\) denotes its (effective) performance. This directly generates the cost estimate \(T_s = N\tau_s\). The \(s\) subscript in \(T\) denotes the scalar implementation.

Vector processing implies the existence of a vector instruction which corresponds to the following implementation

\[
\text{\texttt{mult}(i) = seq} \ (j = 1, M) \ \text{op}(i, j)
\]

\[
\text{\texttt{op}(i, j) -> \texttt{unit}(j)}
\]

as discussed in the pipelining example. Then the \texttt{PAMELA} model is given by

\[
L_v = \text{par} \ (i = 1, N) \ \text{seq} \ (j = 1, M) \ \text{use}(u_j, \tau_v)
\]

where \(u_j\) represents unit \(j\) of the vector processor, \(\tau_v\) denotes the cycle time\(^8\), and where the \(v\) subscript denotes the vectorization alternative. This directly generates the cost estimate \(T_v = (M + N - 1)\tau_v\).

Given the above alternatives, the optimization decision, becomes \(T_v < T_s\), which can be symbolically reduced at (meta-)optimization-time according to

\[
T_v < T_s = \tau_v + N\tau_v < N\tau_f = N \geq \left[ \frac{\tau_f}{\tau_f - \tau_v} \right] = N^* \geq N^*
\]

where \(N^*\) (i.e., \([\tau_f/\tau_f - \tau_v]\)) is comparable in spirit to Hockney’s \(n_{1/2}\) parameter [17].

The example shows two aspects. First, it demonstrates the abstract approach towards the two choices of mapping the (inherently) parallel algorithm onto the scalar or vector machine, merely by discussing alternative \texttt{mult} implementation models while using the same algorithmic description. (Note that an explicitly sequential implementation of the algorithm in the scalar case has not been specified. It simply followed from the contention mechanism. Second, it shows how the optimization problem can be expressed in terms of low-complexity cost models. Even though the generic optimization scheme is based on the use of unreduced cost estimation expressions, due to their algebraic nature, mechanical reduction is possible, yielding low-cost optimization logic. Even when \(N\) is only known symbolically, the optimization logic is evaluated at run-time at negligible expense. The decision to make at compile-time is merely the question if it is worth-while to generate the additional integer test \(N \geq N^*\).

For brevity, the above discussion did not include the performance effects of the memory system (e.g., memory pipelines based on parallel memory bank access). However, we stress the fact that this in no way complicates the optimization scheme other than adding syntactic complexity.

### 3.2 Line Relaxation Example

In this section we will demonstrate the SPC approach applied to the well-known, data parallel line relaxation algorithm kernel for a distributed-memory machine [1]. First, we show how the optimal data partitioning is derived. Next, we show how the optimal implementation is derived, based on an interprocessor pipelining scheme, instead of remapping between the two phases of the algorithm. This solution is similar to the compiler optimizations discussed in [1]. However, we are able to automatically derive this design solution within the SPC framework.

\(^8\)Note that in reality the startup time will be determined by more factors than just the pipeline hardware stages, e.g., call overhead, memory latency. However, the above linear model can always account for the startup and bandwidth parameters as measured in practice simply by defining the pipeline as a combined software/hardware pipeline such that \(S\) and \(\tau_v\) satisfy (i.e., fit) the performance measurements.
3.2.1 Data Partitioning

Consider a line relaxation phase applied to an $N \times N$ matrix $A$, expressed in SPC according to

$$
\text{main = seq (i = 1, N-2)}
\text{par (j = 0, N-1)}
\text{update(i,j)}
\text{!!C!! ... \{ A[i,j] = f(A[i-1,j], A[i+1,j]); \}}
$$

In the above phase the relaxation sweep direction is in the $i$ direction (typically followed by a sweep in the $j$ direction in the next phase which is not considered for now). In the parallelization for a $P$ processor distributed-memory machine we consider the choice between two regular block partitioning strategies, i.e., either along the $i$ axis or along the $j$ axis (a choice, of course, that is trivial).

Let the mapping function $\mu(i,j)$ denote the index of the processor resource responsible for the update of $A_{ij}$. In terms of the above SPC model this implies the mapping

$$
\text{update(i,j) \rightarrow cpu(\mu(i,j))}
$$

The corresponding PAMELA model is

$$
L = \text{seq (i = 1, N-2)} \text{par (j = 0, N-1)} \text{use(\mu(i,j), \tau_u)}
$$

where $\tau_u$ represents the computation work load associated with the update of element $A_{ij}$ on node $\mu(i,j)$. For the $j$ axis partitioning it holds $\mu(i,j) = j/B$ where $B = N/P$ denotes the block size (for simplicity we assume $P|N$). It directly follows (the dots represent some algebraic reductions)

$$
L = \text{seq (i = 1, N-2)} \text{par (j = 0, N-1)} \text{use(\mu(j/B), \tau_f)} \rightarrow T = \ldots = (N-2) \frac{N}{P} \tau_f
$$

which implies speedup. For the $i$ axis partitioning it holds $\mu(i,j) = i/B$. Hence,

$$
L = \text{seq (i = 1, N-2)} \text{par (j = 0, N-1)} \text{use(\mu(j/B), \tau_f)} \rightarrow T = \ldots = (N-2)N \tau_f
$$

which directly reveals the algorithm’s sequential nature\(^9\). Indeed, in the present algorithm setting, an $i$ axis partitioning will not yield any speedup\(^10\).

3.2.2 Pipelining

A possible solution is to remap the matrix between each phase. The corresponding optimization logic is described at length in [13]. However, for the horizontal phase an alternative optimization can be performed which puts the whole issue of possible remapping in a different perspective. The optimization we consider is based on pipelining the computation across the processors which is explained at length in [1]. Unlike in the remapping case, we ignore data communication for simplicity. Recall the original code for the horizontal phase, i.e.,

$$
\text{main = seq (j = 1, N-2) par (i = 0, N-1) update_h(i,j)}
$$

For the purpose of a future algorithm transformation we consider the following equivalent version

$$
\text{main = seq (p = 0, P-1) par (i = 0, N-1) update_b(i,p)}
$$

where the $j$ loop is stripmined in $P$ blocks of size $B$, which, under the original data mapping, are precisely local to processor $p$ according to

$$
\text{update_b(i,p) = seq (j = 0, B-1) update_h(i,p*B+j)}
\text{update_b(i,p) \rightarrow cpu(p)}
$$

The PAMELA model is given by

$$
L = \text{seq (p = 0, P-1) par (i = 0, N-1) use(cpu_p, B \tau_f)}
$$

which, as before, predicts the absence of speedup.

\(^9\)Note, that static cost estimation based on the message-passing SPMD code will not reveal this.

\(^{10}\)This property only holds for the current algorithm. Next, we will consider a modified version of the algorithm in which $i$ axis partitioning does yield speedup.
\[ \text{main} = \text{par} \ (i = 0, N-1) \ \text{seq} \ (p = 0, P-1) \ \text{update}_b(i, j) \]

which is allowed as the i and j references are independent. Note that this produces a pipeline (see examples earlier). Indeed, instead of running sequential, each i loop is now pipelined such that the next processor executes a different i loop instance concurrently, yet obeying the j sequence [1]. Thus we assume a schedule such that the next i loop is executed when the previous j loop has traversed exactly one processor (a block of B indices)\(^{11}\).

The loop reversal has a great impact on the cost estimate as it directly follows

\[ L' = \text{par} \ (i = 0, N-1) \ \text{seq} \ (p = 0, P-1) \ \text{use}(cpu_p, B_T) \rightarrow T' = \ldots = O(N^3/P)T_f \]

which implies speedup. Thus, simply by loop reversal in the SPC model, the same (order of performance can be achieved as in the first phase, without remapping the data. The associated test is given by \(O(N^3/P)T_f < O(N^3)T_f\), which is reduced at compile-time in favor of processor pipelining. Of course, \(L'\) ignores the additional processor pipeline startup delay as well as the additional communication overhead as the need for communication is still present. Especially when multiple sweeps are performed in both phases remapping can still be appropriate. This point is further discussed in [13] which includes the communication effects into the analysis.

4 Related Work

In the following we briefly review some of the parallel programming models that have been proposed in terms of the trade-off they represent with regard to programming power and performance analyzability. For an extensive review on the subject of performance analysis techniques one is referred to [13].

As argued in the introduction, many programming models (or coordination models) do not impose any restrictions with regard to programming power, such as the message-oriented programming models (e.g., Strand [11], CSP [16], MPI [22], PVM [23]). As mentioned in the rationale in Section 2, the problems due to NSP synchronization structure as well as the low-level at which non-determinism is provided, rule out symbolic cost analysis. As shown by the MRM model, however, restrictions, as e.g., in Fortran-M in order to guarantee determinism [10] suffer from the inability to adequately express dynamic control flow. This problem also arises in functional data parallel models (e.g., SCL [8]) which only provide deterministic coordination skeletons.

In more procedure-oriented programming models synchronization is achieved through the use of shared objects (via queues in Orca [16], sync objects in CC++ [8], monitors in Opus [6], tuples in Linda [4]). While queues, sync objects, and tuples provide unrestricted programming power (like semaphores, based on full-empty semantics), the Opus construct is more or less comparable to our approach to provide high-level non-determinism through ME. However, the flexibility offered by their constructs for CS does allow for the specification of NSP graphs which complicates symbolic analysis. Other approaches that also have the ability to produce NSP structure are models where task parallelism (i.e., CS) is directly deduced from the data dependencies (Fx [14], Jade [21]). Again, the primary aim is at capturing maximum (data flow) parallelism, rather than imposing restrictions to facilitate performance analyzability.

5 Conclusion

We have presented the SPC parallel programming model that is inspired by the need to balance programming power and performance analyzability in order to provide a framework for automatic optimization. This balance is achieved by imposing restrictions regarding synchronization structure in conjunction with a programming concept, called contention programming. Although this concept as well as the use of structured operators for CS (SP programming) and ME (monitors) is not new, to the best of our knowledge the combination of these concepts to create a highly structured parallel programming model has not been introduced before. We have illustrated the advantages of SPC by deriving a number of optimizations for a line relaxation algorithm on a distributed-memory machine. In the near future we will focus on the possible advantages SPC offers with regard to automatic partitioning and scheduling, as well as on our research to substantiate our conjecture that the use of SPC entails a limited loss of parallelism.

\(^{11}\)The reason for the strip mining is that simply reversing the original loops would not produce the intended schedule. Because of the fair contention mechanism the \(i, j\) accesses would still be scheduled according to a column-major scheme rather than the intended row-major scheme. Thus the model would behave similar to the original (without loop reversal).
References


Distributed Recursive Datatypes

D. K. Arvind and T. A. Lewis

Department of Computer Science, The University of Edinburgh,
Mayfield Road, Edinburgh EH9 3JZ, Scotland.

Abstract

The principal feature of the Single Program, Multiple Data (SPMD) model of parallel programming is the execution of the same basic program, but which operates on data local to each processor. This locality is easily determined if the structure is regular and static, as in an array, but less so if it is irregular and recursive, as in a tree, and which evolves during the course of the program. One solution is to introduce a scheme to name the nodes, i.e. for a node \( x \) in the structure with a sub-node \( s \), the scheme generates a name for \( s \), when given a name \( n \) for \( x \). This name will, however, not be unique if the node is shared, and will result in multiple names for the same node. But if we know in advance which nodes will be shared, then the aliases can be normalised to a unique name. This paper introduces an algorithm, with a correctness proof, for generating these unique names. The method is encapsulated in a library for manipulating distributed recursive data structures in C++ and MPI for the Cray T3D. Performance results for two case studies are also included.
A Sound and Simple Semantics of the forall Statement within the V-nus Compiler Framework.

P.F.G. Decherin L.C. Breebaart F. Kuijiman
C. van Reeuwijk H.J. Sips
BoosterTeam@cp.tn.tudelft.nl
Delft University of Technology
The Netherlands

Abstract

In this paper we present the V-nus compiler framework that we use to introduce a generalized forall statement for parallel languages. The forall statement is an important language construct in many (data) parallel languages. It specifies which computations can be performed independently. Many different definitions of such a construct can be found in literature, with different conditions and execution models. We will show how forall constructs as found in a wide class of parallel languages can be mapped to the generalized forall statement in V-nus. In addition, the forall statement we propose has the ability to spawn more complex independent activities than can be found in these languages.

Denotational semantics are used to define the meaning of the forall and define only one possible program state change; i.e. only one outcome is possible after execution of the forall. It is shown that it is feasible to implement this forall efficiently. Furthermore, it has a concise operational semantics that can easily be used in programming.

An introduction to the V-nus compiler framework presents the context in which the generalized forall statement has been constructed. An overview is given of the role of the intermediate language V-nus in our compiler framework.

1 Introduction

The forall statement is an important language construct in many (data) parallel languages [3], [4], [5], [9], [13], [19]. It specifies which computations can be performed independently. Although its necessity is widely accepted, the forall definition differs per language. The forall statement in each of the languages was designed with specific implementation criteria in mind.

We think it is important to have a clear and generalized semantics for forall statements in all languages in which they occur. When translating different high-level languages to an intermediate representation it is important to have only one meaning for a forall statement. This paper defines a generalized forall statement and discusses its semantics and implementation. We will show how forall constructs as found in the languages Booster [3], Connection Machine Fortran (CM Fortran) [5], and High Performance Fortran (HPF) [9] are mapped to this generalized forall statement without forfeiting semantics and efficiency. Furthermore, the forall statement we propose has the ability to spawn more complex independent activities than can be found in these languages. Having a single language construct that spawns a parallel loop increases the orthogonality of a language. It is our opinion that this forall statement is not only suited to an intermediate representation, but can also be adopted at the syntactic level in high-level parallel languages.

The context of our forall statement is supplied by V-nus, a concise intermediate language we have defined for data parallel programs [6]. The purpose of V-nus is providing a language platform to which other data parallel languages can be translated, and subsequently optimized. Therefore, V-nus is a suitable language for the introduction of the generalized forall.
We will use the word *iteration* to mean any generic index-driven iteration language construct. Such a construct consists of two parts: an *index-space specification* and a *body*. The body is parameterized with respect to, and will be executed for, every index in the index-space specification. Each separate instance of the body is called a *body-instance*. We use the word *forall* to represent a specific kind of *iteration* that allows a concurrent execution of the body-instances. We use denotational semantics to define the meaning of the *V-nus* language constructs, which will allow us to verify and optimize *forall* statements.

It is our goal to find a *forall* statement that complies with the following requirements:

- The denotational semantics of a *forall* statement must represent only one possible program state change; that is, only one outcome should be possible after execution of a *forall*.

- It must be possible to implement the *forall* statement efficiently. This means that the administration that is needed to execute the *forall* should not use excessive amounts of computational resources.

- The *forall* statement must be capable of representing a wide class of *forall* definitions as can be found in (data) parallel languages.

- It must be possible to give a concise operational semantics of the *forall* statement that can easily be understood.

Section 2 presents an overview of different types of *iteration* statements, one of which is chosen for the *V-nus* language. In Section 3 we investigate the *forall* statement in different (data) parallel languages. The meaning of the *forall* statement of *V-nus* is defined in detail in Section 4. In Section 5 we explain how the *V-nus* *forall* can be implemented efficiently. Subsequently, it is shown in Section 6 how different interpretations of a *forall* statement can be represented in *V-nus*. In Section 7 we again consider the goals of this paper. A short introduction to our compiler framework is given in Section 8. Finally, we draw some conclusions in Section 9.

## 2 Different types of *iteration*

In the set of *iteration* statements, we can identify two extremes: the sequential loop and the completely parallel loop.

**The sequential *iteration*** This *iteration* is equivalent to the conventional for-loop. The body-instances are executed one after another, in a predefined order. Data dependencies are of no consequence.

**The chaos *iteration*** The body-instances are executed completely concurrently. All body-instances work on the same memory locations, and no assumptions are made about the order in which writes to and reads from these variables take place. A non-deterministic behaviour can be a result of this model of execution.

Besides these extremes we present a number of other *iteration* statements below.

**The merge *iteration*** The body-instances are executed completely concurrently. All body-instances work on their own copy of the program state, so determinism is guaranteed. At the end of the *iteration* statement all the now-changed individual program states of the body-instances must be merged back into a single parent program state by a merge function.

**The statement-atomic *iteration*** The body-instances are executed concurrently, but the statements within the body are considered to be atomic. This means that during the execution of a statement *S* it is guaranteed that no other body-instances will be updating the value of any of the variables used in *S*. 
The body-atomic iteration The body-instances are executed concurrently, but the entire body is considered to be atomic; i.e. during the execution of a body-instance \( i \) it is guaranteed that no other body-instances will be updating the value of any of the variables used in body-instance \( i \).

These intermediate forms of iteration statements are called forall statements. Both the statement-atomic and the body-atomic forall statement imply a certain amount of synchronization and variable-shielding. We have chosen the merge forall in V-nus, because it has the most potential parallelism, and is well-suited for use in programming.

3 Existing approaches

Both data parallel languages as well as control parallel languages use the concept of a forall statement to denote the spawning of concurrent actions. There is a common trade off in the definitions of forall statements in these languages: constraints on the body decrease the potential parallelism, but lack of these constraints may cause non-determinism. An assignment in a specific body-instance may affect the computation of another body-instance, when these body-instances share the same variable. The outcome of a forall statement is then dependent on the order of computation. In general, it is impossible to know at compile time which data elements are assigned to. The solution for this problem is putting restrictions to forall statements to reduce undesirable behaviour. Function calls and procedure calls complicate the task of finding well-defined restrictions even more, since it is hard to analyse their effect on the program context in general.

One of the first versions of the forall statement was introduced by Thinking Machines Corporation in CM Fortran [5]. It is used to distribute computations over the processing elements of the Connection Machine (CM). The keyword forall indicates that the body-instances can be executed independently. The body-instances consist of one assignment with a left-hand side that is not assigned to by another body-instance. The use of certain kinds of expressions, such as user defined functions and assignments to array sections that depend on the index variable, always causes the forall statement to be executed serially.

Vienna Fortran [19] defines a broader forall statement by permitting private variables. These variables are known only in the forall statement in which they are declared, and each body-instance has a separate copy. A body-instance can consist of any legal FORTRAN 77 executable statement. Tightly nested forall statements can be used to specify multiple levels of parallelism. Vienna Fortran also restricts the forall body by requiring that a value written in one body-instance is neither read (define-use dependence) nor written (define-define dependence) in any other body-instance (see [20] for a description of define-use and define-define dependencies). The result is always deterministic.

Experiences with the forall statement in the Fortran dialects CM Fortran, Vienna Fortran, and Fortran D [10] led to the construction of the HPF forall. CM Fortran uses the forall statement to create parallelism explicitly by distributing body-instances over the CM. Vienna Fortran uses the forall statement to indicate that the different body-instances are independent and can be logically executed in parallel. In HPF [9] it is the distribution of data that introduces parallelism.

The HPF forall statement consists of a single assignment statement. The left-hand side of each body-instance of this assignment can only be assigned once. This excludes define-define dependencies. Execution of the forall statement requires the right-hand sides of the body-instances to be evaluated before these are assigned to the left-hand sides. This implies that a synchronization is needed. Only function calls to pure functions (functions that have no side effect) may be used in the right-hand side. It is then assured that define-use dependencies leave the outcome of the forall statement deterministic.

It is allowed to have multiple statements in the HPF forall body\(^1\), but this means that each

---

\(^1\)HPF distinguishes between forall statements and forall constructs; the latter may have multiple statements in their bodies.
assignment of the body is executed completely; i.e. as if the assignments were written as forall statements in the same order (see Section 6). In addition a directive independent has been introduced for both do loops and forall statements. The directive assures the compiler that the body-instances can be executed in an arbitrary order, without any computational differences in the result. In case of the multiple statement forall this means no synchronization is needed between the statements. Both the single assignment and the multiple assignment forall statement of HPF are used in the same form with the same semantics in Fortran 95, according to the proposed revision [8].

The data parallel language Booster [3] has no forall keyword. It is possible to assign array sections in parallel by using an aggregate assignment. Unambiguous semantics are enforced by the requirement that no element is used as a target before it is used as a source. Function calls do not make such analysis harder, since Booster requires the functions to be referentially transparent; i.e. no side effects occur and no global variables are accessed.

In the control parallel language SuperPascal [13] the forall statement is used to denote an array of parallel processes. A severe restriction is imposed on the forall body to prevent ambiguous computations: the body may not assign to a variable. This implies that a body-instance must output its results through a communication channel or a file. Otherwise, the results will be lost when the body-instances terminate. Procedure calls can be used in the body, which causes no problems under the given circumstances.

The forall statement in Compositional C++ [4], denoted by the keyword parfor, also initiates the parallel execution of the body-instances. Multiple statements are allowed in the forall body, where the statements of a specific body-instance are executed sequentially. Note that this is in contrast with the multiple statement forall of HPF. No copies are made of data that is used in the body-instances, so loop carried dependencies can lead to non-deterministic results.

The Myriads parallel do uses a copy-in/copy-out semantics [2]. When a program executes a parallel do construct, parallel tasks are created, one for each iteration of the parallel do. Each task gets a separate copy of the parent program state. At the end of the parallel do all child program states are merged to form the new program state. It is, however, not explained how this merging can be done efficient.

Li and Wolfe [15] mentioned the difficulties in defining well-behaved parallel constructs without making arbitrary decisions. They developed a framework for analyzing the behaviour and relations of various sequential and parallel control constructs. Their DoPar iteration has a similar meaning as the merge forall, described in Section 2 and is based on the parallel do of the Myriads system. Also here, it is not mentioned how to implement this general iteration construct efficiently. Using their framework they present how and when different loop constructs can be substituted by another loop construct.

In the remainder of this paper we will use the forall statements of Booster, CM Fortran, and HPF as representatives of the many forall definitions that can be found in literature on data parallel languages.

4 The semantics of the V-nus forall

Similar to the other languages, the V-nus forall statement is represented by the syntax: forall IndexSpace Body. The term IndexSpace specifies the range of the index variable; the term Body represents the block of statements that will be executed for each value of the index variable (see Example 4.1).

---

Example 4.1 The V-nus forall statement

Consider: forall [i:3] (a := i). The index variable is i and ranges over 0, 1 and 2. The body is a := i; an example of a body-instance is a := 1.
Body-instances of the \textit{V-nus forall} statement are to be executed completely independently. By this we mean that data that can be changed by a body-instance \(i\) will not affect the computation of another body-instance \(j\). However, a global interference is still possible when there is a define-define dependence between the possible body-instances; i.e. two body-instances that write to the same variable. We say that

a \textit{forall} statement is deterministic if no define-define dependence is present between any two different body-instances of the \textit{forall} statement.

We want to record the concept of the \textit{forall} statement in a semantic model, such that we can use this model to reason about a program. We use denotational semantics [14], in which the meaning of a program can be expressed by the composition of the meanings of its parts. The denotational semantics are useful when we want to rewrite only parts of a program, and leave the meaning of the whole program as it is.

In denotational semantics a program state captures all necessary information about the context in which a program fragment is executed. A program state is valid only if each variable of the program state is given exactly one value (see Example 4.2).

\begin{center}
\textbf{Example 4.2 Program states.}
Consider the \textit{forall} statement of Example 4.1. A valid program state after execution of the body-instance \(a := 1\) is: \((a = 1)\). The program state \((a = 0, a = 1)\) is invalid, because the variable \(a\) is given two values.
\end{center}

The semantics of a program fragment are given by a program state change, represented by a pair \((ps, ps')\) of program states. In case of the \textit{forall} statement, program state changes are computed for all body-instances. Say, for body-instance \(i\) the state change \((ps, ps_i)\) is computed. Then the different program states \(ps_i\) (for all \(i\)) are merged into the final program state \(ps'\), which will be the program state after the \textit{forall} statement has been executed. This merge operation consists of two actions. First \(ps_i\) is compared with \(ps\), providing only the difference \(diff_i\) between these program states. Secondly, all elements of \(diff_i\) will be put into \(ps\). This is done for all \(ps_i\) in arbitrary order.

The mathematical framework for the denotational semantics of \textit{V-nus} (including the \textit{forall} statement) is described in [6].

5 The implementation

Implementing the \textit{forall} statement as presented in Section 4 may cause problems when efficiency is considered. Merging the different program states of the body-instances is inefficient, since computing the difference between program states is time consuming.

In order to arrive at an efficient implementation of the \textit{forall} statement, we take the following approach. At the start of a \textit{forall} statement the program state \(ps\) is preserved. For the execution of a body-instance a subset \(qs_i\) of \(ps\) is used for the context in which this body-instance will be executed. Only the data that is needed in the body-instance is extracted from \(ps\) and will be used for \(qs_i\). Each time something needs to be read from memory, it is read from \(qs_i\). When something needs to be written to memory, it is not only stored in \(qs_i\), but the same store action is also performed on \(ps\). In this way, each change that is made by a single body-instance is also visible in the global program state, but will not affect the other body-instances. This is how the final program state \(ps'\) arises from the original program state \(ps\), without the need for a merge or a difference operation (see Figure 1).

The construction of \(qs_i\) is dependent on the information the compiler has about the data that is used in the body-instance. This information can be generated automatically by well-known
dependence analysis techniques and by hand via pragmas. A pragma is an optional annotation for the compiler that gives additional information about a certain program construct. Pragmas that can be used for a `forall` statement specify which data should be copied in `qs_i`.

If a `forall` statement is not annotated by a pragma, then the local program states `qs_i` are created as explained above. If a pragma is present the compiler relies on this information and only copies the given data structures for the accompanying program states `qs_i`. In our opinion, it is more useful to specify for which data structures a dependency exists, than it is to specify those structures for which no dependency exists. The syntax of a pragma for a `forall` statement is:

```
<< dependsOn Expression >>
```

which expresses a dependency for the data structure(s) `Expression`. An empty list of specifications (i.e. `<< >>>`) means that no data needs to be copied. Of course, it is the responsibility of the programmer to avoid the introduction of non-determinism due to a pragma.

Especially when the compiler can not determine at compile-time what dependencies exist between the body-instances, it is useful to be able to give additional information to the compiler. In Example 5.1 is shown how the efficiency of a `forall` statement can be optimized by introducing a pragma.

---

**Example 5.1** Using pragmas for a `forall` statement.

Consider the program fragment:

```c
forall [i:n] { 
   A[i] := B[C[i]]
   B[C[i]] := A[i+1]
 }
```

At compile-time it is unknown which elements of `B` are referenced. The conservative approach is taken so that this `forall` is characterized as non-deterministic. Furthermore, for each body-instance `qs_i` a complete copy of `B` is created. If all elements of `C` are different then each body-instance will write to a different element of `B`. In that case, there is no need to create a copy of `B` in each `qs_i`. Note that for `A` it is necessary to create a local copy. So we can safely annotate the `forall` statement as follows:

```c
<< dependsOn A[i+1] >> forall [i:n] { 
   A[i] := B[C[i]]
   B[C[i]] := A[i+1]
 }
```

which means that each body-instance `qs_i` must have a copy of `A[i+1]`, and no other copies are needed. When a pragma is used, the compiler assumes that the `forall` is deterministic.

---

When using pragmas the execution model is slightly changed. Each time something needs to be read from memory, it is read from `qs_i` if it exists in `qs_i`; otherwise it is read from `ps`. Proper use of pragmas still guarantees determinism when the original program was deterministic.

In the implementation of a deterministic `forall` statement, all differences between the program states `qs_i` are collected in the global program state `ps'`. This is exactly as it is described by the denotational semantics.

The denotational semantics use the same computation for both deterministic and non-deterministic `forall` statements. That makes the result of a non-deterministic `forall` statement dependent on the computation order. In this case the efficient implementation of a `forall` statement may compute other results than the theory prescribes. In Example 5.2 a possible difference is presented between the computation used in the implementation, and the computation used in the semantics.
Example 5.2 Difference between theory and implementation.

Consider the program fragment: forall [i:2] {a := i; b := i}. The denotational semantics predict that the body-instance for i = 0 will result in the program state \( p_{S0} = (a = 0, b = 0) \). The body-instance for i = 1 will result in the program state \( p_{S1} = (a = 1, b = 1) \). \( p_{S}' \) will then be either \( p_{S0} \) or \( p_{S1} \).

The implementation, on the other hand, may cause the following execution orders:

- \( a := 0, b := 0, a := 1, b := 1 \) or \( a := 1, b := 1, a := 0, b := 0 \) or
- \( a := 0, a := 1, b := 0, b := 1 \) or \( a := 1, a := 0, b := 1, b := 0 \) or
- \( a := 0, a := 1, b := 1, b := 0 \) or \( a := 1, a := 0, b := 0, b := 1 \)

which will lead to the same possible program states as predicted by the theory, plus the program states \( (a = 0, b = 1) \) and \( (a = 1, b = 0) \).

![Program state changes caused by a forall statement](image_url)

Figure 1: Program state changes caused by a forall statement

In Example 5.2 both the body-instances write to the variables \( a \) and \( b \), which makes the forall statement non-deterministic. Theory and implementation only differ for non-deterministic forall statements. We want to use a semantic model in which the outcome of a program (fragment) is unambiguous. When non-determinism is forced by a non-deterministic forall statement it is sufficient to mention that the outcome is unpredictable. For now, there is no need for a semantic function that defines the set of all possible outcomes.

6 The forall compared

As shown in Section 3, many languages have a notation that describes some independent iteration over an index space. However, the semantics of these constructs differ for each language. In this section, we compare the forall statements of the data parallel languages Booster, CM Fortran, and HPF, and we show how these differently defined forall statements can be mapped to the V-nus forall statement.
CM Fortran as well as HPF use the same method for the evaluation of forall IndexSpace Body: first, evaluate the expressions in IndexSpace, then, evaluate all expressions present in Body, and finally, perform the assignments of Body. More detailed descriptions are given in the appropriate language specifications.

Consider the following examples in pseudo code:

\begin{align*}
\text{forall } i=0,n, j=0,m & \quad \text{forall } i=0,n, j=0,m \\
\quad \quad a[i,j] = expr & \quad \quad a[i,j] = F(X) \\
\text{end } & \quad \text{end } \\
\quad \quad \quad \quad (6.1) & \quad \quad \quad \quad (6.2)
\end{align*}

\begin{align*}
\text{forall } i=0,n, j=0,m & \quad \text{forall } i=0,n, j=0,m \\
\quad \quad a[i,j] = expr, & \quad a[i+1,j] = F(X) \\
\text{end } & \quad \text{end } \\
\quad \quad \quad \quad (6.3)
\end{align*}

where the expressions \(n\) and \(m\) are not dependent on each other, \(expr\) is some arbitrary expression that does not contain a function call, \(F\) represents a function, and \(X\) is an actual argument list that is not dependent on the array \(a\). In each of the languages Booster, CM Fortran, and HPF the index space over which is iterated is the Cartesian product \([0 \ldots n] \times [0 \ldots m]\).

**CM Fortran** In CM Fortran, Example 6.1 will cause the assignments to be executed on the CM in parallel. The assignments of Example 6.2 will be executed sequentially because of the function call on the right hand side. Example 6.3 is not valid since CM Fortran allows only one statement in a forall body.

**Booster** In Booster, both Example 6.1 and Example 6.2 will perform the assignments in arbitrary order. Because in Booster functions are referentially transparent, the function call causes no side effects, and therefore it is guaranteed that each element is used as a source before it is used as a target. In Booster too, only one assignment is allowed in the forall body, which makes Example 6.3 invalid.

**HPF** In HPF, Example 6.1 and 6.2 have the same meaning as in Booster. Although pure functions in HPF need not be referentially transparent, it is forbidden for those functions to have side effects. This allows the different body instances of a forall statement to be evaluated in arbitrary order. Example 6.3 is semantically equivalent to the following consecutive forall statements:

\begin{align*}
\text{forall } i=0,n, j=0,m \quad & a[i,j] = expr, \\
\text{forall } i=0,n, j=0,m \quad & a[i+1,j] = F(X)
\end{align*}

Note that the second forall statement only starts when the first forall statement has finished. It can not be rewritten to one do independent loop, because a define-define dependence exists for \(a[i]\), \(1 \leq i \leq n - 1\).

**V-nus** Example 6.1 interpreted in Booster, CM Fortran, or HPF can be represented in V-nus by:

\begin{align*}
\text{forall } [i:n+1, j:m+1] \{ a[i,j] := expr \}
\end{align*}

Example 6.2 interpreted in CM Fortran needs a sequential loop in V-nus, such as:

\begin{align*}
\text{for } [i:n+1, j:m+1] \{ a[i,j] := F(X) \}
\end{align*}

In Booster and HPF this example can be represented in the same way as Example 6.1 is represented. Example 6.3 interpreted in HPF can be rewritten to two single assignment forall statements as presented above. These can easily be translated to V-nus. Note that if Example 6.3 was interpreted in V-nus directly, it would denote a non-deterministic forall statement because of the define-define dependencies. Define-define dependencies are allowed if they occur in the same body-instance. For example, if the subscript \(i+1\) of Example 6.3 is replaced by \(i\) then the forall statement has become deterministic.
HPF directives  Every INDEPENDENT DO loop in HPF can be represented by the _V-nus forall_ statement, since no loop carried dependencies occur at all. Due to _V-nus_ pragmas the effectuality of the INDEPENDENT directive can also be utilized.

The NEW directive in HPF is used to create variables that are local to a single body-instance. In _V-nus_ it is possible to use loop-bodies as scope-boundaries. So, the named variables in the NEW directive of HPF can be represented in _V-nus_ by locally declared variables in a loop.

Functions  Since _V-nus_ requires functions to be referential transparent, functions of other languages that are less restrictive need to be rewritten in _V-nus_. If a non-_V-nus_ function uses (or writes to) a global variable, it can be represented by a corresponding _V-nus_ function where this global variable is passed via another function parameter (and consequently becomes local to the function). As a result, an HPF _forall_ statement with a call in its body to a pure function that uses a global variable can be represented in _V-nus_ while fully preserving the semantics and effectiveness.

_V-nus_ specific expressiveness  Now, we show an example of an optimization that can only be expressed by using the _V-nus forall_. Consider the following matrix operation:

\[
\text{for } [i,m] \{ \\
  \text{forall } [i,n] \{ a[i,j] := a[i,j-1] + a[i,j+1] + a[i-1,j] + a[i+1,j] \} \\
\}
\]

The optimization we have in mind is based on synchronization elimination [12]. By reversing the i and j loop the operation can be expressed as

\[
\text{forall } [i,n] \{ \\
  \text{for } [j,m] \{ a[i,j] := a[i,j-1] + a[i,j+1] + a[i-1,j] + a[i+1,j] \} \\
\}
\]

which has no computational differences in the result. Instead of executing _forall_ statements in sequence, the _forall_ body-instances can now be executed concurrently, yet obeying the j sequence. It is easy to see that no define-define dependence occurs, which makes it a deterministic _forall_ statement. This _forall_ statement is not 'valid' in the other parallel languages mentioned in this paper.

7 The goals revisited

We can now show that we have met the four requirements for the generalized _forall_ statement, as listed in Section 1.

denotational semantics  For non-deterministic _forall_ statements an unambiguous program state change is forced by the specification of a computation order; i.e. the order in which the program states _ps_i_ of the body-instances are merged. The program state change of a deterministic _forall_ statement is not dependent on the computation order.

Suppose that _diff_i_ and _diff_j_ contain the same variable _x_. That means that both in _ps_i_ and in _ps_j_ the variable _x_ has been changed with respect to the original _ps_. That can only be so, if both body-instances _i_ and _j_ have a 'define' for _x_. But for deterministic _forall_ statements this can never occur. So, there are no two different _diff_i_ and _diff_j_ containing the same variable. When all the _diff_i_ are put into _ps_ for the construction of _ps_i_ , _ps_i_ will be a valid program state. Note that it does not matter in which order the _diff_i_ are inserted into _ps_ – the same _ps_i_ is constructed.
efficiency Computing the difference between two program states can be inefficient when done naively. Therefore, the \textit{V-nus} implementation does not use the same computation as given in the semantics. The approach taken here requires some computation overhead compared to a sequential loop. This overhead is due to the following computations:

- Before the body-instances can be executed, each body-instance must get its own program state (which is usually a small subset of the global program state).

- During execution of a body-instance, each write action is performed twice. One to update the local program state and one to update the global program state. In many cases, one of these two write actions can be omitted.

The computation time for the construction of the program state $ps'$ is in the order of the number of variables that are used in the \textit{forall} body. A direct implementation of the theoretical scheme would need linear time in the number of variables of the entire program and the number of body-instances of the \textit{forall} statement.

Pragmas can be used to help the compiler such that no unnecessary copies of variables are made. This reduces computation and space overhead.

representation \textit{V-nus} can be used to capture the meaning of different definitions of \textit{forall} statements, as is shown in Section 6. Therefore, we think that our \textit{forall} definition is suitable for an intermediate representation.

expressiveness When a programmer specifies a \textit{forall} statement, only one extra condition has to be taken care of compared to a sequential loop: the same variable may not be written to by two (or more) body-instances. We think that this concept can easily be applied when programming \textit{forall} statements. However, the programmer must be able to verify whether the condition is met, such that non-determinism can be detected. Partially, this can be done at compile-time. A runtime solution for the other cases requires too much overhead in general. But while using execution trace techniques it is possible to recognize a define-define dependence, when different values are written to the same variable. When the same value is written twice to that variable a define-define dependence is not recognized, but nevertheless the result is deterministic. So, a weaker condition can be checked: the execution trace of a program has no define-define dependence that causes non-determinism.

8 The \textit{V-nus} compiler

In the compiler we have constructed for the \textit{V-nus} language, all traditional compilation phases (parsing, translation, optimization, code generation) are handled by separate tools [7] [17] (see Figure 2).

Since \textit{V-nus} is an intermediate language we rely on separate front-ends to perform the translation from high-level, implicitly parallel languages, such as \textit{Booster} or HPF, to \textit{V-nus}. A programmable transformation-rule based compiler translates the data parallel code to explicitly parallel code, and performs various optimizations. The \textit{V-nus} back-end, finally, translates the resulting code to input for an existing compiler for a sequential language combined with a communications package.

At this moment, we have a partial front-end for \textit{Booster}, and a complete front-end for an experimental data parallel language. The rule-based compiler translates to SPMD code with explicit communication. The current back-end translates the explicitly parallel \textit{V-nus} code to C++, using calls to the PVM library [11], or alternatively, calls to the p-threads package.

The set of \textit{V-nus} constructs covers a wide range of concepts found in parallel languages. Implicit and explicit parallelism as well as high-level and low-level semantics can all be expressed in \textit{V-nus}. 
Program transformations  One of the main features of $V$-nus is to allow for program transformations. Once a program is translated to the $V$-nus language, program transformations can be used for optimizations.

High-level semantics  $V$-nus must be able to express high-level semantics. That means that $V$-nus has (almost) the same descriptive power as the original language that is processed by the front-end. Hence, information about the computation described in the original program is preserved when it is translated to $V$-nus. Then, program transformations can be more powerful since high-level semantics can be used within the intermediate representation. Another important consideration that requires high-level semantics is its use for existing libraries of templates. A template is a representation for a rewrite rule that matches on code patterns. As early as possible in the compilation process program constructs can be recognized that are also present in libraries of templates [14]. When the original program has been unraveled too much, certain code patterns can not be recognized anymore as an instance of an existing code pattern.

As a result, having high-level semantics in $V$-nus, a rather straightforward front-end can be used in the compiler framework.

Low-level semantics  Once the original program is expressed in $V$-nus without loss of information, it must be possible to rewrite the program into lower-level program constructs. Important transformations are those that convert a data parallel program to an explicitly parallel program; i.e. a program with explicit communication primitives. These and other transformations are carried out by optimizing engines. An example of complex transformations can be found in [16]. Mirroring the advantage of high-level semantics, low-level semantics do not require a complex back-end.

Generality  Only a single language should be used for compiling and optimizing programs of different high-level languages. All effort can be put into the engines that transform $V$-nus programs. For an arbitrary data parallel language it should suffice to build a front-end that translates to $V$-nus.

Semantic framework  Strict semantics are required to serve as a base for transformation rules. Since the meaning of $V$-nus programs can be expressed formally one can verify whether a transformation preserves the semantics of the original program.
9 Conclusion and future work

We have constructed a general forall statement in the intermediate language V-nus, capable of representing forall statements of other data parallel languages. The forall statement has a semantics that is easy to understand and is unambiguous. The compiler or execution tracing tools can check whether the requirements are met that make the forall statement deterministic. The body of this forall statement has the same syntax as an ordinary loop body. This allows the spawning of more complex concurrent computations than can be found in other data parallel languages. Furthermore, we have shown it is still possible to have an implementation without a major increase of computation overhead. Computation and space overhead can be adjusted by pragmas. In this way, our forall statement is suited to be used as an intermediate representation and as a language construct in parallel languages.

We have shown that V-nus can be used as an intermediate representation for different forall statements by using three examples. In the near future we will investigate a generalized translation scheme for forall statements in the languages Booster, CM Fortran, and HPF to V-nus. In an unpublished short note we have compared more forall examples than presented in this paper. This note is available at:


References


Pipelining Data Parallel Computations in an HPF Compiler

Thomas Brandes*
Institute for Algorithms and Scientific Computing (SCAI)
German National Research Center for Computer Science (GMD)
Schloss Birlinghoven, 53754 St. Augustin, Germany
Phone: +49 2241 14 2492 Fax +49 2241 14 2181
eMail: Thomas.Brandes@gmd.de

Frédéric Desprez†
Laboratoire de l'Informatique du Parallélisme (LIP)‡
Ecole Normale Supérieure de Lyon (ENS)
46, Allée d'Italie, 69364 Lyon Cedex 07, France
Phone: +33 7272 8037 Fax +33 7272 8080
eMail: Frederic.Desprez@inria.fr

Abstract

Many scientific applications can benefit from pipelining computation and communication. Our aim is to provide compiler and runtime support for High Performance Fortran applications that could benefit from these techniques.

Though pipelined computations and their utilization within a compiler has already been studied and investigated for a long time, our approach is based on an efficient runtime support. This does not only decrease the complexity of the compiler but also allows optimizations based on runtime data.

There is already a library available that provides efficient basic subroutines that minimize the overhead by using asynchronous pipelining. This library has been adapted to an existing HPF compilation system for supporting pipelined computations and communications.

This paper describes the necessary compilation techniques for utilizing pipelined execution and the integration of this library in the runtime system. Results on some application kernels as well as on a real application program are given.

---

*This work has been supported by the Esprit-6948 project PPPE (Portable Parallel Programming Environment).
†This work has been supported by INRIA Rhône-Alpes, the Eureka Eurotops project and the PRG-GDR PRS.
‡URA 1398 CNRS and INRIA Rhône-Alpes
1 Introduction

With the introduction of High Performance Fortran (HPF) [11, 16], it is possible to use the data parallel programming paradigm in a very convenient way for scientific applications.

With current compilation technology provided in commercial HPF compilers, these programs will execute phases of computations and communications on different sets of data and no overlap exists between communications and computations. Moreover, communication phases are synchronous, i.e. each processor executes these phases at the same time and waits until the last processor completes his communication phase. An important task of the HPF compiler is to detect the potential of overlapping computation and communication and to take efficient use of it.

Overlapping is not always possible because of the dependences within the code. In this case the computation might be broken into smaller pieces that can be executed in a pipelined fashion. This is also called macro-pipelining [15, 21]. Usually, the resulting code of macro-pipelining is very complicated. But we will show that it is possible to use runtime system functions that do this splitting at runtime. This does not only decrease the complexity of the HPF compiler, but also allows the optimization of overlapping computation and communication at runtime. This is done by making some runtime measurements that determines the best size of granularity.

Though most of the techniques are already known, this paper focus on the efficient use and the integration in an existing HPF compilation system. The runtime functions are a new version of the LOCCS library (Low Overhead Communication and Computation Subroutines) that has been first presented in [7].

In section 2 we present the basic idea of pipelined computations where section 3 gives some more formal terms. The realization of utilizing pipelining in the HPF compilation system ADAPTOR is described in section 4. The necessary runtime support based on the LOCCS library is outlined in section 5. Results are presented in section 6 where also the advantages of pipelining in a real application are shown. Based on the results we will draw conclusions and discuss future work in section 7.

2 Pipelined Computations

In pipelined computations, a processor cannot begin execution until it receives results computed by its predecessor (see (a) on Figure 1). Though this kind of pipelined execution for its own is still a sequential execution, there are two possibilities to extract partial parallelism by overlapping computations.

- If one pipelined computation follows another one (e.g. within a loop), the processors can overlap the computations.
- For a given pipelined execution it might be possible to break the computation and to send partial results. By this way the processors may overlap their computations (see (b) and (c) on Figure 1).

The following code (Gauss-Seidel relaxation) shows a typical example that benefits from pipelining computation.

```
REAL, DIMENSION (P1,N2) :: F
!HPF!
DISTRIBUTE F(*,BLOCK)

DO J=2,N2-1
  DO J=2,N1-1
    F(I,J) = (F(I,J-1)+F(I-1,J)+F(I,J+1)+F(I+1,J))*0.26
  END DO
END DO
```

Both of the two nested loops contain loop carried true data dependences that will imply communication between columns and rows of the array F. Assume that the columns are distributed in a block
fashion among the available processors. Every processor computes the results in row order, sending the last value to the next processor as soon as it is ready. This strategy produces a pipelined effect.

Usually, this method is not very efficient due to the large communication startup time on MIMD message-passing machines. Therefore, a variant of the method is chosen where each processor computes a few rows before communicating the results (the number of these rows will be referred as block size). Figure 2 shows the data transfers between the different processors.

![Figure 1: Breaking up a pipelined computation.](image1)

![Figure 2: Wavefront in pipelined computations on distributed arrays.](image2)

The current definition of HPF does not allow to specify this kind of parallelism explicitly, neither by array operations nor by any kind of parallel loops. HPF compilers can only use pipelining by providing advanced compilation technology that detects automatically this kind of implicit parallelism.

## 3 Notations and Definitions

For the pipelining we are considering a multi-dimensional loop nest of depth \( k \).

\[
\begin{align*}
\text{DO } & I_1 = 1 b_1, \; ub_1 \\
\text{DO } & I_2 = 1 b_2, \; ub_2 \\
& \ldots \\
\text{DO } & I_k = 1 b_k, \; ub_k \\
& \text{Body } (I_1, I_2, \ldots, I_k) \\
\text{END } & \text{DO} \\
& \ldots \\
\text{END } & \text{DO} \\
\text{END} \\
\end{align*}
\]

We make the following assumptions about this loop nest:

- The iteration space of the loop nest is rectangular, in other words, the values \( lb_j \) and \( ub_j \) do not depend on outer loop variables.

- Data dependences between the different iterations of the body are allowed only if the dependences can be described by a distance vector \( d = (d_1, \ldots, d_k) \).

- The iterations of the loop can be mapped to a rectangular processor grid so that every write access to a variable appearing on the left hand side of an assignment is local.

Loop carried data dependences will not cause any problems. If non-local values are accessed within the loop, the necessary communication can take place before the execution of the local part of the iteration space. In the following example, this results in a communication from the right processors to the left ones for the one-dimensional processor array.
REAL A(N,N)
!HPFS DISTRIBUTE A(*,BLOCK)
...
! anti dependences
DO J = 1, N-1
  DO I = 1, N
    A(I,J) = f(A(I,J-1))
  END DO
  END DO
A(I,J) = f(A(I,J-1))
END DO
END DO
END DO

A loop is called a cross-processor loop if the iterations are distributed among the processors and if it has a loop-carried true dependence \( d = (..., d_i, ...) \) with \( d_i \neq 0 \). In this case, one processor needs results from the previous processor to start its own iterations [13].

A compiler can find cross-processor loops by considering all pairs of array references that cause loop-carried true dependences. If non-identical subscript expressions occur in a distributed dimension of the array, all loop index variables appearing in the subscript expressions belong to cross-processor loops. In the following, cross processor loops are marked as DO ACROSS.

REAL A(N,N)
!HPFS DISTRIBUTE A(*,BLOCK)
...
DO J = 2, N
  DO I = 1, N
    A(I,J) = f(A(I,J-1))
  END DO
END DO
END DO

In this example, there is a loop carried true dependence caused by \( A(I,J-1) \) and \( A(I,J) \). Since the last dimension of \( A \) is distributed, the compiler marks the \( J \)-loop as a cross-processor loop. For loop carried true dependences, the new computed data has to be sent. Before starting its iterations, the processor has to receive corresponding values from its left neighbor and has to send values after its iterations to the right processor.

if (not first processor) recv A(:, my_low-1)
DO J = my_low, my_up
  DO I = 1, N
    A(I,J) = f(A(I,J-1))
  END DO
END DO
if (not last processor) send A(:,my_up)

A direct execution of this loop will result in totally serial code. Processor 1 starts with its iterations, then passes data to the next processor 2. This process can then start with its iterations and so on. Figure 1 (a) shows this behavior.

The presence of any cross-processor loop in a loop nest indicates that it is a pipelined computation. The granularity of pipeline parallelism is determined by the amount of computation enclosed by cross-processor loops. If the cross-processor loop is the outermost loop there is usually no advantage of parallel execution.

Fine-grain pipelining interchanges all cross-processor loops as deeply as possible, so that they enclose the least amount of computation. On loop interchange, there is a more fine-grain pipelining possible.

DO I = 1, N
  if (not first processor) recv A(I, my_low-1)
  DO J = my_low, my_up
    A(I,J) = f(A(I,J-1))
  END DO
END DO
if (not last processor) send A(I, my_up)
END DO

The resulting program execution order generates values needed by other processor in shortest time, achieving the finest granularity of pipelining. Unfortunately, it also results in high message overhead since a message is sent for each iteration accessing non-local data (see (b) on Figure 1).
A more coarse grain pipelining will be possible after strip-mining of the outer loop with strip size BLOCK and then interchange the strip-element loop and the loop with cross-processor dependences [22]. By this way, communication costs are reduced as more elements are communicated in a single message. However, parallelism is also reduced since processors must wait longer before beginning to compute.

\[
\begin{align*}
\text{DO } & I = 1, N, \text{ BLOCK} \\
\text{DO ACROSS } & J = 2, N \\
\text{DO } & I = 11, I1+\text{BLOCK}-1 \\
A(I,J) &= f(A(I,J-1)) \\
\text{END DO} & \\
\text{END DO} & \\
\text{END DO} & \\
\text{if (not first processor) recv } A(I1:11+\text{BLOCK}-1, \text{my_row-1}) \\
\text{DO } & J = \text{my_row, my_up} \\
\text{DO } & I = I1, I1+\text{BLOCK}-1 \\
A(I,J) &= f(A(I,J-1)) \\
\text{END DO} & \\
\text{END DO} & \\
\text{END DO} & \\
\text{if (not last processor) send } A(I1:11+\text{BLOCK}-1, \text{my_up})
\end{align*}
\]

The legality of this transformation is due to the fact that the two loops can be interchanged (fully permutable loop nest).

Taking large block sizes (tiles) reduces the number of communications but also causes a delay in starting the second and the following processors. This tradeoff has been examined theoretically by many other authors [1, 14, 18, 19, 20, 23].

One way to calculate the optimal block size is by applying some model. We calculate the time to compute an \( m \times n \) data array that is block distributed in the second dimension (so every processor works on a \( m \times \frac{\beta}{p} \) block). The computation time for a single element is always the same so we can assume the cost 1. Let \( B \) the chosen block size, \( C \) the communication overhead. As one processor computes at first \( B \) rows before sending results to the next processor, the delay between two processes is (note that we ignore the data transfer time which is minimal compared to the startup time):

\[
\frac{n B}{p} + C
\]

The total parallel execution time is:

\[
\frac{n m}{p} + \frac{m C}{B} + (p-1) \left( \frac{n B}{p} + C \right)
\]

By differentiation of that expression with respect to \( B \) and holding the result to zero

\[
-\frac{m C}{B^2} + \frac{n (p-1)}{p} = 0
\]

there will be a solution for \( B \):

\[
B = \sqrt{\frac{m p C}{n (p-1)}} \approx \sqrt{\frac{m}{n}} \sqrt{C} = \sqrt{\frac{m}{n}} \frac{\text{block communication cost}}{\text{element computation cost}}
\]

This model is quite simple and does not hold for the general case (if the complexity of the computation and/or communication is not a linear function of the number of data sent). Formulas for the more-dimensional cases of DO ACROSS loops are given in [18]. A complete analysis of the general case is given in [6].

4 Implementation within ADAPTOR

4.1 Description of the ADAPTOR Tool

ADAPTOR (Automatic Data Parallelism Translator) is a public domain compilation system developed at GMD for compiling data parallel HPF programs to equivalent message passing programs [3].

By means of a source-to-source transformation, ADAPTOR translates the data parallel program to an equivalent SFMD program (single program, multiple data) that runs on all available nodes.
The essential idea of the translation is to distribute the arrays of the source program onto the node processors where the parallel loops and array operations are restricted to the local part owned by the processor. Communication statements for exchanging non-local data will be generated automatically. The control flow and statements with scalar code are replicated on all nodes.

Beside the translation system, a runtime system called DALIB (distributed array library) [2] has been developed that will be linked with the generated message passing program (see Figure 3). It realizes functions for global reductions, transposition, gather and scatter operations, circular shifting, replication and redistribution of distributed and local arrays. Timing and tracing facilities as well as a random number generator are also part of this library. As the runtime system is available on most parallel systems, the generated message passing programs will run on all these machines.

**Figure 3: Overview of ADAPTOR.**

At the beginning of the ADAPTOR project at GMD, the system has been used to study compilation techniques for the HPP language and to evaluate HPP for applications. Currently, it is used to evaluate optimization strategies for HPP compilers and to experiment with new language features.

### 4.2 Shadow Edges

Usually, every processor has only memory allocated for its own local part of the distributed array. A shadow edge is an extension of the local part to keep some non-local values of the array. By this way, less temporary data and less copying of local data is necessary. Shadow edges were first introduced by Gerndt [9] as overlap areas, they are also supported in ADAPTOR and in many other compilation systems.

In most cases the compiler itself will generate the most convenient size of shadow edges. In some cases, e.g. for separate compilation, the user itself can specify the necessary size of the shadow edges. With the introduction of HPP 2.0 [12] the definition of shadow edges is standardized.

```
REAL, DIMENSION (N1,N2) :: F
!
HPP$ DISTRIBUTED (+, BLOCK), SHADOW (0:0,1:1) :: F

! ...
DO I=2,N2-1
  DO J=2,N1-1
    ! shadow edges keep non-local copies of F(I,J-1) and F(I,J+1)
    F(I,J) = (F(I,J-1)+F(I,J)+F(I,J+1)+F(I+1,J))/4.05
  END DO
END DO
```

An update of shadow edges is a corresponding operation that makes the values of the corresponding memory area to valid ones. This operation implies communication for the distributed dimensions. The update of a shadow edge can be restricted to a certain part of the whole shadow area, e.g. only the left shadow edge is updated for the first ten elements. The compiler itself has to find out where this update is necessary. For the pipelined execution of nested loops the compiler exploits this mechanism to drive the communication.
4.3 Extrinsic Subroutines

HPF provides the EXTRINSIC mechanism to interface HPF with message passing programs. By the keyword HPF::LOCAL the compiler knows that in this procedure only the local part of the distributed arrays is visible. This mechanism allows also to specify a subroutine that works only on a sub-block of the local part of the distributed array.

```fortran
REAL, DIMENSION(N,N) :: A
HPFF$ DISTRIBUTE (*.BLOCK) :: A

... CALL BLOCK (A(M1:M2,N1:N2)) ! executes BLOCK on a subgrid ...
...

EXTRINSIC (HPF::LOCAL) SUBROUTINE BLOCK (A)
REAL A(1:1)
HPFF$ DISTRIBUTES (*.BLOCK) :: A ! assigns block distribution in last dimension
DO J=1,bound(A,2),ubound(A,2)
  DO I=1,bound(A,1),ubound(A,1)
    A(I,J) = f(A(I,J-1))
  END DO
END DO
END SUBROUTINE BLOCK
```

Within a local subroutine, the compiler will not look for implicit communications and therefore assume that every computation is a local one. The directives for the data mapping within a local subroutine will only assert a certain distribution. The compiler itself handles the correct addressing of the local array parts. The access on non-local values within shadow edges requires that the values are updated before.

4.4 Identification of Pipelined Computations

The current version of the ADAPTOR compiler can identify pipelined computation under the following conditions:

- the iterations space is rectangular and can be mapped according the HPF conventions to a rectangular processor grid so that all write accesses are local,
- true data dependencies between the different iterations of the body can be described by a distance vector \( d = (d_1, \ldots, d_k) \) that has only one non-zero entry (in this case the entry will be positive).
- there is no possibility for parallel execution without pipelining.

The restriction on the data dependences imply that the loop nest is fully permutable, in other words, the loops can be interchanged arbitrarily. Then a pipelined execution is possible where the driver can split up the computation into arbitrary subblocks. The following steps are executed during the compilation:

- All arrays on the left hand side involved in data dependences will get the corresponding shadow edges to hold the data that will be communicated (see section 4.2).
- Transform the loops into code restricted to corresponding blocks and call a function that drives the execution of the blocks (see section 4.3).

```fortran
! gives code
REAL A(N,N)
HPFF$ DISTRIBUTES A(*.BLOCK)

... DO_ACROSS J = 2, N
  DO I = 1, N
    A(I,J) = f(A(I,J-1))
  END DO
END DO
END DO
!

! intermediate code during compilation
REAL A(N,N)
HPFF$ DISTRIBUTES A(*.BLOCK(LOW_SHADOW=1))

... call DALIB::GCO::DRIVER (block, nop, 1, 0, A(1:N,2:N), [0,1]) ...
...
EXTRINSIC (HPF::LOCAL) SUBROUTINE block (A)
...
END SUBROUTINE block
```
The code on the right hand side will be generated automatically if the compiler has identified the possibility of pipelined computation for the loop nest. It is quite easy for the compiler to extract the required information (sections, dependence vectors). But the interface is written in such a way that it can also be used directly by the user. This was advantageous for testing the pipelined computation during the implementation within the ADAPTOR compilation system.

4.5 Runtime Support

The drivers for pipelined computations of loop nests are realized as corresponding routines in the DALIB runtime system. A driver routine is called with the following arguments:

- the local subroutine that realizes the code executed for one block,
- the element computation cost in terms of number of operations that will be used to calculate the block size,
- the description of the iteration space which can be given in most cases by a corresponding array section,
- the arrays belonging to the iteration space with the corresponding data dependences.

```
subroutine S (A,N)
  real A(N,N)
  !HPFS INHERIT A
  K = ..., ! value of K only known at runtime
  DO J = K+1, N
    DO I = 1, N
      A(I,J) = f(A(I,J-K),B(I,J))
    END DO
  END DO
! will result in the following intermediate code during the compilation:
! call DALIB LOOCCS DRIVER (BLOCK, nop, 2, 0, A(1:N,K+1:N), [0,K],
!   B(1:N,K+1:N), [0,0])
end subroutine S
```

Setting up the correct parameters to the driver routine is possible without knowing the distribution of the arguments. As with any section also the distribution and the sizes of the full arrays are passed, the driver can decide at runtime about the actual splitting into subblocks. The driver will not split up the computation at all if no cross processor dependences exist.

Obviously the runtime approach introduces some overhead at execution time that would not be necessary if a full compiler approach has been chosen, e.g. like in the Fortran D compiler [15]. Our experiments have shown that this overhead can be neglected. But our approach offers some important advantages:

- The runtime approach is more general if certain information is only available at runtime. This can be the value of a dependence distance vector but also the actual distribution of a dummy argument.

- The code for pipelined execution on a message passing system is rather complex. Especially the incorporation of different parameters increases dramatically the implementation effort within the code generation of the compiler. The realization of pipelining in the runtime system is simpler, especially as many other existing functionality can be used.

- Moving machine dependent optimizations into the runtime system results in a better design and maintainability of the compilation system.

The realization of the pipelined execution within the runtime system is described in section 5. A detailed description of the interface for the most important routine is given in section 5.1.
5 Integrating LOCCS within the HPF Runtime System

The LOCCS library [7, 8] has been first designed for a use in parallel programs with explicit communications. The idea behind the LOCCS is to provide a set of routines to avoid coding complicated codes using pipelined computations. Many codes present such communication and computation patterns [4, 5]. A subset of these routines has been implemented on various architectures like Intel Paragon and a network of workstations.

Its first interface was not well-suited for a use within a compiler. A new version of the library is currently under development for ADAPTOR. The DALIB_LOCCS_DRIVER routine is the interface for the general pipeline scheme of nested loops. Other routines for other communication and computation patterns are in development.

5.1 Interface of the LOCCS Driver Routine for Nested Loops

The driver routine that realizes pipelining for rectangular loops with simple data dependence distance vectors is called in the following way:

\[
\text{call DALIB_LOCCS_DRIVER (block, nOp, n, m, a1, d1, a2, d2, ..., an, dn, b1, ..., bm)}
\]

- The first parameter block is the name of the local subroutine that contains the code executed for a single block.
- The parameter nOp specifies the number of operations needed to compute one iteration of the iteration space. A negative value can be used to specify directly the block size.
- The parameter n specifies the number of array sections that will be split up in corresponding blocks.
- The parameter m specifies the number of arguments that will be passed like usual parameters of local subroutines. The \( b_i \) are these arguments.
- Every parameter \( a_i \) specifies an array section. All sections must be implicitly or explicitly aligned with each other and can be identified with the iteration space.
- Every parameter \( d_i \) specifies the dependences corresponding to the section \( a_i \). This information is needed to set up the corresponding communication.

The driver will call the local subroutine with the corresponding arguments. The local subroutine is called with the corresponding blocks of the array sections \( a_1, ..., a_n \), and with the other arguments \( b_1, ..., b_m \).

\[
\text{INTRINSIC (HPF_LOCAL) SUBROUTINE block (a1, ..., an, b1, ..., bm)}
\]

This driver will set a pipeline for each distribution and each communication scheme. Every detail of the pipeline is hidden inside the routine. The pipeline is set at run-time according to the distribution, the shape of the grid of processors and the amount of communications and computations.

5.2 Realization of the LOCCS Driver

In the following we explain in more detail how the driver routine has been realized.

- The iteration space for the pipelined computation is extracted from the section arguments and the local iteration space is fixed.
- Some additional runtime checks will make sure that all array sections involved in the computation are aligned with each other. For efficiency, this additional checks can be disabled.
• The dependence vectors will be analyzed and in case of cross-processor dependences communication schedules will be set up. The communication updates the corresponding values within the shadow edges.

• A dimension of the iteration space will be chosen along which the computation and communication will be split. Usually this is a serial dimension.

• The initial block size will be chosen based on certain formulas and heuristics. This block size decides about the grain for the splitting.

5.3 Choosing the Block Size

In section 3 we mentioned that in certain situations there is a theoretical approach for choosing the optimal block size to obtain good performances. This size is heavily system dependent and should be computed at run time. Usually the formulas give only an approximated size which does not take into account system overheads such memory hierarchy management for the computations, buffering and links contention for the communications.

A first estimation of the optimal size can be computed using the formulas. Therefore the driver takes into account the computation effort for a single iteration as well the parameters of the machine. The computation effort will be given by the number of operations for one iteration and is passed as an additional argument to the driver routine.

The size can be dynamically adjusted at run-time depending on the real execution time of the previous loops. The new block size will be send to the processors with the data. This dynamic behavior has also been suggested in [20].

6 Results

In this section, we give the results of first experiments using optimized pipelined computations within ADAPTOR.

6.1 Gauss-Seidel Relaxation

The code we used for the Gauss-Seidel relaxation is the following one:

```plaintext
ITER = 0
PHAIX = 1
DO WHILE ((PHAIX .ge. 0.001) .AND. (ITER .LT. MAXITER))
  DF = F
  DO J = 2, NI-1
    DO I = 2, N2-1
      P(I,J) = (P(I,J-1)+P(I-1,J)+P(I,J+1)+P(I+1,J))*0.25
    END DO
  END DO
  ITER = ITER + ITSTEPS
  DF = ABS(F-DF)
  PHAIX = MAXVAL (DF)
END DO
```

Figure 4 shows the speedups achieved by pipelining on the IBM SP2 (AIX 3.2.5) for different matrix sizes. In figure 4, we compare the execution time of the parallel program against the execution time of the serial program to show the real speed-ups. The pipelined execution uses in the serial dimension the block size 16.

Figure 5 shows the execution times for different block sizes. The results are given for a grid of the size 1024 x 1024.

Considering only one single node, the execution times should be the same as there is no pipelined execution and no communication at all. But if a small block size is chosen (e.g. 1 or 2) the strides for accessing the array elements is much higher and will result in worse performance.

Though the efficiency of the pipelined execution is not very high and only around 50% for bigger problem sizes, there cannot be better results expected due to the limited parallelism.
6.2 Alternating-Direction-Implicit Integration (ADI)

One step in the ADI algorithm consists of sweep along the columns followed by a sweep along the rows (a more detailed discussion can be found in [17]).

```
PARAMETER (N=...)
REAL, DIMENSION (N,N) :: A, B
!RFP$ DISTRIBUTE (+,BLOCK) :: A, B
!
! sweep along the columns
DO I = 2, N
   DO J = 1, K
      A(I,J) = A(I,J) - A(I-1,J)*B(I,J)
   END DO
END DO
!
! sweep along the rows
DO J = 2, N
   DO I = 1, N
      A(I,J) = A(I,J) - A(I,J-1)*B(I,J)
   END DO
END DO
```

In the first loop nest, the J loop is perfectly parallel and can be mapped to the distributed dimension of the arrays A and B. In the second loop nest, the I loop is perfectly parallel, but is mapped to a serial dimension of the arrays. It will be executed serially. The J loop is a cross processor loop due to the data dependences between \(A(I,J)\) and \(A(I,J-1)\).

One parallelization strategy is to redistribute the arrays A and B before the second loop nest. In this case, the parallel I loop will belong to a distributed dimension (Figure 6) and requires no communication. The ADAPTOR tool supports dynamic redistributions and will generate efficient communication for this kind of transposition.

The other parallelization strategy for the algorithm is to have a pipelined execution of the loop nest as described in this paper. In this case, no redistribution is necessary. The compiler will generate in an intermediate step the corresponding call to the corresponding driver routine (Figure 7).

Figure 8 shows the speedups of the two parallelization strategies on the Intel Paragon for a \(512 \times 512\) grid. The pipelined execution achieves nearly the optimal speed-up, the dynamic data remapping is not profitable in this case.

6.3 Helmholtz-Solver

In this section we discuss the pipelined execution within a real application program [10]. The German Weather Service daily distributes a medium weather forecast for every part of Germany. Therefore
PARAMETER (N,...)
REAL, DIMENSION (N,N) :: A, B
!
HPF$ DISTRIBUTE (*.BLOCK) :: A, B
!
DO J = 1, N ! parallel execution
  DO I = 2, N
    A(I,J) = A(I,J) - A(I-1,J) * B(I,J)
  END DO
END DO
!
HPF$ REDISTRIBUTE (BLOCK,*) :: A, B
!
DO J = 1, N ! parallel execution
  DO I = 2, N
    A(I,J) = A(I,J) - A(I,J-1) * B(I,J)
  END DO
END DO
!
HPF$ REDISTRIBUTE (*.BLOCK) :: A, B
!
Figure 6: Parallelization of the ADI algorithm with redistribution.

it employs the European global weather model to extract the necessary initial data and boundary conditions, respectively, for the final German model. The DWD uses for its computation a 3D-mesh that consists of 109 x 109 x 20 grid points. The majority of computations on this mesh deals with the solution of a two-dimensional Helmholtz equation that has to be solved for each of the vertical levels and for each prognostic variable. In detail, a weather forecast of 78 hours with time-steps of 300 seconds based on 20 vertical surfaces requires the solution of 18720 boundary value problems.

The used Helmholtz equation is set up by an implicit time discretization. A five star discretization of the Helmholtz equation gives a new equation system. The given program solves this equation by using a Fourier transformation and a Gauss elimination. Since there are dependences for column and lines in the original case, the idea is to resolve the dependences in one dimension by applying the Fourier transformation. This leads to a tridiagonized equation system that could be easily solved with Gauss elimination.

The following subroutine is the central routine.

SUBROUTINE SOLVE (X,Y,F,COEF,BPT,AP,TRIGS,MT,N,N1,N1,MT1,N2)
DIMENSION X(N,N),Y(N,N),F(N,N)
DIMENSION COEF(N,N),BPT(N,N),AP(N)
!
HPF$ DISTRIBUTE F(*,BLOCK)
!
HPF$ ALIGN (i,j) WITH F(*,j) :: X,Y
!
HPF$ ALIGN (i,j) WITH F(i,*j) :: COEF,BPT
HPF$ ALIGN (j) WITH F(*,j) :: AP
!
CALL FFT (F,X,Y,F,COEF,MT,*,+1)
!
Gauss elimination
!
X(2:1:2) = X(2:1:2) + BPT(2:1:2)
DO J = 3, N1 ! forward sweep
END DO
!
DO J = N2, 2, -1 ! backward sweep
  X(2:1:J,N1) = X(2:1:J,N1) - COEF(2:1:J) * X(2:1:J,J+1)
END DO
!
CALL FFT (X,Y,F,COEF,MT,*,+1)
Within FFT it is absolutely necessary that X, Y and F are serial in the first dimension. The
Figure 8: Speedup ($N = 512$): transposition vs. pipelining (Intel Paragon XP/S).

Parallelism of the do loops can only be used with redistribution or with pipelined execution.

Similar to the ADI algorithm discussed in section 6.2 we can apply two parallelization strategies: one with transposition of the arrays and one with pipelined execution. The two versions have been measured on a IBM SP2 (AIX 3.2.5).

Table 1 compares the results of the two parallelization strategies (for pipelining the block size 16 has been chosen). Here, only the execution times of the loops between the two calls of the FFT routines are compared with each other.

<table>
<thead>
<tr>
<th></th>
<th>1 node</th>
<th>2 nodes</th>
<th>4 nodes</th>
<th>8 nodes</th>
<th>16 nodes</th>
<th>32 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>transposition</td>
<td>0.380 s</td>
<td>0.588 s</td>
<td>0.361 s</td>
<td>0.227 s</td>
<td>0.107 s</td>
<td></td>
</tr>
<tr>
<td>pipelining</td>
<td>0.380 s</td>
<td>0.361 s</td>
<td>0.123 s</td>
<td>0.107 s</td>
<td>0.100 s</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Execution times (Gauss elimination): transposition vs. pipelining (IBM SP2).

Table 2 shows the execution times for the full solver. The FFT dominates the whole computation where due to cache effects there is also a superlinear speedup for the execution on 8 nodes if compared with the execution on 4 nodes. The benefits of pipelining for the full server are not very high. But it should be noted that the pipelined version needs less memory as no temporary data for the redistribution is required.

<table>
<thead>
<tr>
<th></th>
<th>1 node</th>
<th>2 nodes</th>
<th>4 nodes</th>
<th>8 nodes</th>
<th>16 nodes</th>
<th>32 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT 1</td>
<td>4.400</td>
<td>2.131</td>
<td>1.091</td>
<td>0.965</td>
<td>0.179</td>
<td>0.096</td>
</tr>
<tr>
<td>LOOPS</td>
<td>0.380</td>
<td>0.199</td>
<td>0.181</td>
<td>0.123</td>
<td>0.107</td>
<td>0.105</td>
</tr>
<tr>
<td>FFT 2</td>
<td>4.265</td>
<td>2.142</td>
<td>0.969</td>
<td>0.346</td>
<td>0.179</td>
<td>0.087</td>
</tr>
<tr>
<td>total</td>
<td>9.042</td>
<td>4.516</td>
<td>2.157</td>
<td>0.839</td>
<td>0.476</td>
<td>0.285</td>
</tr>
</tbody>
</table>

Table 2: Executions times for the full Helmholtz solver (IBM SP2).

7 Conclusion and Future Work

The results of the previous section show that there is no doubt about the usefulness of pipelining and about the efficient realization within ADAPTOR. Our experiments have shown that the pipelined
computation can be integrated successfully within an HPF compiler.

Using these kind of optimizations in a message passing program is difficult and usually machine-dependent. By their integration in a compiler, the user can benefit from it a very convenient way. The optimization of pipelining at run-time ensures that good performances can be obtained, even if the system presents a chaotic behavior.

The interface of the LOCCS library is now well suited for the optimization of a compiler like ADAPTOR. The driver routine has to be extended, for example to allow broadcast communications and non-rectangular iteration spaces. An MPI version of the LOCCS is currently under development.

Acknowledgments

We thank ZAM, Jülich for providing access to the Intel Paragon XP/S and the BETLAB group at SCAI, GMD for supporting the access to the IBM SP2.

References


Data Structures for Efficient Execution of Programs with Block-Cyclic Distributions

Swaroop Dutta and J. Ramanujam
Dept. of Electrical
Computer Engineering
Louisiana State University
Baton Rouge, LA 70803-5901, USA
swaroop.jxr@ee.lsu.edu

Abstract

Efficient run-time techniques for generating node code on distributed-memory machines is important. Node code generation exploits the repetitive access pattern displayed by the distributed array section. Several techniques for the efficient enumeration of the access pattern already exist. But only one paper so far addresses the effect of the data structures used in representing the access sequence on the execution time. In this paper, we present various run-time methods involving new data structures for lexicographic enumeration of all the array elements assigned to a particular processor. The methods, namely strip-mining and table-compression facilitate the generation of time-efficient code for execution on each processor. While strip-mining codifies the problem as a double nested loop, table compression proves to be a worthwhile data-structure for faster execution. The underlying theory behind the data-structures introduced is explained and their effects on all possible set of problem parameters is observed. Extensive experimental results show the efficacy of our approach. The results have been compared with the results of the earlier methods proposed by Kennedy et al.
Multi-Level Parallelism
Compiler Support for the Use of Two-Level Parallelism

Thomas Rauber, Gudula Rünger
Computer Science Department, Universität des Saarlandes,
PF 151150, 66041 Saarbrücken, Germany,
+49-681-302-4130, FAX +49-681-302-4290, {rauber,ruenger}@cs.uni-sb.de

Abstract

A large number of numerical algorithms exhibit a two-level structure of potential method and system parallelism which can be exploited for different alternative parallel implementations on distributed memory machines. The compiler system TwoL (Two Level) provides interactive or semi-automatic support for the design and the realization of efficient parallel algorithms in this two-level parallel programming model. The design is structured into well-defined decision steps which are formalized in a TwoL specification language and transformations on this language. We present how the design steps lead to a parallel algorithm, how the design is formalized in the TwoL system, how this compiler system is realized, and which algorithms can lead to derivation steps that automate decision steps. Design or derivation steps are based on parametrized cost functions derived from runtime predictions for the specific parallel target machine. The design process is illustrated by the parallelization of several methods from the area of differential equations.

1 Introduction

Many algorithms in the area of scientific computing exhibit different kinds of potential parallelism, often referred to as task (or function) parallelism and data parallelism. An efficient implementation on parallel machines may require a mixed exploitation of these different kinds of potential parallelism. The consideration of mixed parallelism leads to a large variety of alternatives for parallel implementations differing in the costs on distributed memory machines (DMMs). The choice of the most efficient implementation is desired but often difficult. Because of the benefits one can gain from exploiting mixed parallelism there is a variety of approaches that combine task and data parallelism, including the Fx [14], Fortran M [4], and PARADIGM [1, 7] projects. Mainly, these are projects in the context of parallelizing compilers, which transform sequential programs into parallel output programs.

In this article, we present the integrated framework TwoL (Two Level parallelism) which supports the design of parallel programs with a hierarchical two-level structure of method and system parallelism, a specific structuring of function and data parallelism inherent in a large number of algorithms. Examples are methods from numerical analysis, e.g. methods to solve systems of differential equations, or physical simulations where different simulation techniques have to be combined. In contrast to parallelizing compilers, our approach starts with a specification of the
computational problem describing the maximum degree of method parallelism available in the algorithm. The design process comprises this specification, a fixed set of design decisions and the resulting non-executable parallel frame program, along with a runtime prediction of the final implementation realizing the design.

TwoL comprises compiler tools for the interactive or semiautomatic design process starting from the specification and ending with the frame program, and the final translation into an actual parallel program. For the design step, TwoL provides language support for specifying potential method parallelism, for expressing a fixed set of design decisions, and for transforming the specification according to the design decisions into the frame program. The TwoL system is able to execute performance predictions of the frame program as well as of incomplete specifications in order to evaluate design decisions before the actual implementation. Appropriate interfaces allow interactive design decisions but also automatic algorithms by approximation or optimization decisions based on performance predictions.

The specifications are capable to express hierarchical module structures, where each module represents regular system parallelism or further potential method parallelism. All components of the compiler framework TwoL exhibit the strict separation of method and system parallelism into a two-level structure. The design decisions concern a static assignment of modules to groups of processors, which can be interpreted as scheduling and load balancing of multiprocessor tasks, and a mapping of data or uni-processor tasks to processors within a group of processors. All decisions are based on performance predictions with runtime formulas that are parameterized with the decision to be taken. The evidence of the performance prediction has been demonstrated on the Intel iPSC/860 and Paragon for several examples from numerical analysis, including solution methods for linear and nonlinear equation systems [8] and for stiff [10] and non-stiff [11] ordinary differential systems, and extrapolation methods. The design process has been used for parallel simulations of collisionless electron plasma and a chemical reactor-diffusion system [11].

The main advantage of the TwoL framework lies in the integrated support for a systematic derivation of efficient implementations for which the programmer only provides a description of the computational method. This relies on the assumption that most programmers from a specific application area have a detailed knowledge of the algorithm but are not always familiar with the hardware details of a specific parallel machine that have to be exploited for an efficient implementation. By using TwoL, the programmer can concentrate on the algorithmic properties of the program and does not have to deal with low-level details of the parallel machine. The main emphasis of the paper is a description of the design of the compiler components and appropriate interfaces for the interactive design.

The remainder of the article is organized as follows. Section 2 describes the programming model of TwoL and the language support. Section 3 proposes a realization of the support of design steps. Section 4 outlines the application of the TwoL approach to some solution methods for differential equations that show the usefulness of the approach, Section 5 concludes.

2 TwoL programming model

In contrast to parallelizing compilers, the TwoL approach starts with a specification of the maximum degree of parallelism available in the algorithm to be implemented
and derives a parallel program for a specific DMM in several stages. All stages exhibit a strict separation of method and system parallelism in a two-level structure. The separation of method and system parallelism corresponds to a distinction of potential parallelism depending on the granularity and the regularity of computations. Potential \textit{system} parallelism denotes fine-grain regular executions of independent computations leading to one-processor tasks but comprises also potential \textit{data} parallelism. This kind of parallelism can be realized in an SPMD execution model. Method parallelism occurs when a numerical method consists of independent submethods which cooperate in a non-uniform way and can be realized by multiprocessor tasks according to a group-SPMD execution model.

An important class of algorithms exhibit a two-level structure of potential method and system parallelism. Examples in the area of numerical analysis are solution methods for ordinary differential equations (ODEs) like extrapolation methods, iterated Runge-Kutta (RK) methods, or implicitly iterated RK methods \cite{18, 19}. These methods compute several independent approximation vectors in each time step, which can be computed in parallel to each other and which are combined to determine the final solution vector. In contrast to this medium-grain method parallelism, examples from physics often exhibit coarse-grain parallelism. Examples from physical simulation include multidisciplinary simulations like environmental models and aircraft simulations where different simulation methods are combined. In environmental models atmospheric, surface water, and ground water models are combined, each requiring different numerical simulations which can be computed in parallel to each other. In aircraft simulations, models for fluid dynamics, structural mechanics, and surface heating can be simulated in parallel.

2.1 Design of parallel algorithms

In the design of a parallel implementation, the TwoL framework distinguishes between the \textit{numerical algorithm}, a \textit{module specification} and a \textit{parallel frame program}. The numerical algorithm is a mathematical description of a solution method for a specific problem as it typically used in textbooks on numerical analysis. The TwoL programming model requires the programmer to formulate the numerical algorithm as a module specification which uses an execution-oriented notation with control statements and data structures. The module specification subdivides the computations to be executed into submethods which are called \textit{modules}. The hierarchical structure resembles a call-graph of procedures in sequential programming. Independent modules are a source of parallel execution without any communication between them.

A specification language provides appropriate constructs in order to express explicitly in the hierarchical module specification which module executions must be performed in a fixed order because of data dependencies and which module executions may be performed in parallel because they are independent from each other. Internally, the modules specify the operations to be executed in a notation which is similar to the \textit{Unity} notation \cite{2, 17}. In contrast to specifications in a sequential programming language, this notation has the advantage to clearly specify which program parts can be executed in parallel and, thus, no data dependence analysis is needed to extract the available parallelism within the modules. A different approach would be the use of a parallel programming language like HPF or Fortran 90. A module specification is more general than the specification of a parallel section in Fx \cite{13} because module calls are allowed to have an arbitrary call structure. Currently, we
do not allow the use of recursive calls.

The module specification does not determine a data distribution for composed variables of the modules nor does it contain an assignment of processors to modules. It even does not fix an execution order for the modules because independent modules are allowed to be executed consecutively one after another by all available processors if this results in a better execution time. Fixing all these relevant implementation decisions results in a parallel frame program which is the basis for a parallel implementation on a specific parallel machine. The parallel frame program is not executable, but contains all design decisions for a parallel implementation and can be translated into an executable program by techniques used in parallelizing compilers. The design process of a frame program from a module specification is split into several decision steps to decouple the design decision from each other. In the first step, the decision is made which independent modules are executed concurrently by independent groups of processors and which are executed consecutively one after another by all available processors. This step is referred to as scheduling step in the following. In the second step, processors are assigned to the modules that are executed concurrently. The goal is to assign the processors in such a way that the execution of the different modules is terminated at about the same time, thus yielding a good load balance for the execution of modules. In the third step, the internal data distribution of the modules is fixed. The goal is to determine the data distribution such that the communication (composed of the communication inside the modules and the redistribution communication between cooperating modules) is minimized.

### 2.2 Language support

For each of the design steps, the module specification or an incomplete frame program is augmented with additional constructs to capture the decision of the corresponding step. Thus, the module specification language has several levels depicted in Figure 1. The pure specification language is denoted ModLang; the extensions describing the scheduling, the group sizes, or the data distributions with appropriate annotations, are denoted ModSched, ModGroup, and ModData, respectively.

Examples for a simplified module specification and a corresponding frame program are given in Figures 2 and 3 for an iterated Runge-Kutta (RK) method for solving initial value problems (IVPs) of ordinary differential equations (ODEs). The figures show the upper level method parallelism which is important for the design steps and abstracts from the internal structure of the modules. In Figure 2, the module initialize(f) initializes the iteration vectors \( \eta \). The argument \( f \) denotes the right hand side of the differential equation to be solved. The keyword for denotes a sequential loop of iterations each of which contains a parfor loop of the module
\[ \text{IRK}(f) = \]
\[
\text{while } (x < x_{\text{end}}) \{ \\
\quad \text{Initialize}(f) \\
\quad \text{o for } (j = 1, \ldots, m) \\
\quad \quad \{ \text{parfor } (l = 1, \ldots, s) \{ \text{Corrector}_\text{irk}(f, \sigma^{(l)}_{i}) \} \} \\
\quad \text{o Update}(f, \eta_{\alpha}) \text{ o Stepsize}\_\text{control}(x) \}
\]

**Figure 2: Module specification for the iterated RK method.**

\text{Corrector}_\text{irk}. The construct \texttt{parfor} indicates independent computations; it is a generalization of \( M_1 \parallel M_2 \) denoting the execution of two independent modules. The module \text{Corrector}_\text{irk} realizes a corrector step of the specific Runge-Kutta method. The result of the nested corrector steps is a set of \( s \) iteration vectors which are input to the subsequent module. The module \text{Update} combines the iteration vectors to the final approximation vector \( \eta \) of the time step according to the underlying Runge-Kutta method. The module \text{Stepsize}\_\text{control}(x) realizes a stepsize control mechanism by using the embedded solutions provided by the iterated RK method. The modules are connected by the construct \texttt{o} which denotes a data dependence between two successive modules, i.e., two modules \( M_1 \) and \( M_2 \) are combined with \texttt{o} if output data of \( M_1 \) are required by \( M_2 \) as input data. The loop over the time steps is formulated in a \texttt{while} loop assuming data dependencies between successive time steps.

In the corresponding frame program in Figure 3, the scheduling decisions are expressed explicitly by language constructs. The construct \texttt{for} represents a sequential loop, the construct \texttt{parfor} represents a parallel loop, and the construct \texttt{seqfor} represents a sequential loop of independent modules. Thus, in contrast to the module specification, these constructs specify the exact execution order and are not only a description of the available parallelism.

The decision on the load balancing is represented by assigning numbers of processors to module calls. The notation is \( M \parallel [p] \) where \( p \) denotes the number of processors executing module \( M \). Parallel executions of modules are annotated by a vector of group sizes, e.g. \( [p, \ldots, p] \) describes the group sizes of \( s \) independent methods to be executed consecutively. The input and output parameters of the call of a module \( M \) and their distribution among the executing processors are identified by \texttt{input} and \texttt{output} directives attached to the call of \( M \). The data distribution of the local variables of a module \( M \) are notated in a similar way (not shown in the figure). The annotation \texttt{input: } \eta \texttt{replic}(p) of module \texttt{Initialize}(f) denotes that the input variable \( \eta \) is replicated among the set of \( p \) processors and the annotation \texttt{output: } \sigma^{(i)}_{(0)}, \ldots, \sigma^{(i)}_{(0)} \text{ block}(p) \texttt{denotes that all } s \text{ initialized vectors } \sigma^{(i)}_{(0)} \text{ are distributed blockwise. In a complete specification, the description of a distribution also defines the block sizes. Combining modules with incompatible data distributions makes it necessary to apply an corresponding redistribution operation between modules transforming the given data distribution into the data distribution required by the next module. Such a redistribution operation for the initial vectors is \( \sigma^{(i)}_{(0)}, \ldots, \sigma^{(i)}_{(0)} \text{ is given by } \text{Re}(\sigma^{(i)}_{(0)}, \ldots, \sigma^{(i)}_{(0)}, n/p, \text{block}(p) \rightarrow \text{replic}(p)). \text{ These redistributions are explicitly shown in the frame program as calls to a redistribution library. They are combined with the } \texttt{o} \text{ constructor according to the required sequential execution.}
IRK(f) =
while (x < xend) {
    Initialize(f) [on p] input: η replic(p), output: σ^{0,...,t}_{(p)} block(p)
    o Re(σ^{0,...,t}_{(p)}, n/p, block(p)→replic(p)) o
    for (j = 1,...,m) {
        seqfor (l = 1,...,s) {
            Corrector.irk(f,σ^{l}_{(j)}) [on p]
            input: η replic(p), σ^{l,...,t}_{(j)} replic(p), output: σ^{l,...,t}_{(j)} block(p)
            o Re(σ^{l,...,t}_{(j)}, n/p, block(p)→replic(p)) o
        }
    }
    o Update(f,η) [on p] input σ^{l,...,t}_{(j)} replic(p), output: η block(p)
    o Re(η,n/p,block(p)→replic(p)) o Stepsize.control(x)
}

Figure 3: Parallel frame program for the iterated RK method.

3 Compiler Support

In this section, we describe the realization of the TwoL compiler. In particular, we describe how the scheduling, the load balancing, and the data distribution decisions are formalized in decision structures and outline an automatic support for the creation of decision structures.

3.1 Scheduling of modules

For the scheduling decision, the module specification is transformed into a global module dependence graph (MDG) which is a directed acyclic graph. The global MDG is composed of local MDGs. There is a separate local MDG $G_M$ for each composed module $M$. The nodes of $G_M$ are the module calls in the body of $M$. There is an edge from a node $n_1$ to a node $n_2$, if the call corresponding to $n_1$ has to be executed before the call corresponding to $n_2$ because of data dependencies. The root of $G_M$ represents the entry point of module $M$. In the absence of recursive calls, the local MDG of the different composed modules can be inserted into each other, leading to the global MDG for the complete module specification.

The TwoL compiler is designed to provide two alternatives for the scheduling decision: an interactive version in which the programmer specifies an execution order for the modules and an automatic version in which the compiler determines an execution order with approximation or optimization algorithms such that a minimal execution time results. In both cases, the scheduling is determined by a top-down traversal of the global MDG. At each branching point, the scheduling procedure decides on an execution order of the subsequent module calls. In the general case, there are $s$ modules that can be executed in parallel. The scheduling procedure first determines the number $k \leq s$ of subgroups in which the group of executing processors is subdivided and then partitions the $s$ module calls among the $k$ processor groups. Choosing $k = 1$ subgroups corresponds to a sequential execution of the $s$ module calls whereas $k = s$ corresponds to a parallel execution that exploits the available degree of parallelism completely.

For each branching point, the scheduling decision is recorded in a scheduling de-
cision structure (SDS) which is annotated to the branching point. The SDS contains $k$ lists of modules, each denoting the modules assigned to one of the $k$ processor groups. The SDS can be determined interactively by the programmer or automatically by a compiler tool using results from scheduling theory [16, 15]. Such a compiler tool must be based on estimates of the runtime of specific module calls. The computation and communication times have to be estimated for different numbers of processors. For a fixed number $k$ of processor groups, the problem of assigning independent module calls to processor groups is equivalent to assigning independent uni-processor tasks to single processors which is an NP-complete problem. Therefore, a greedy algorithm can be used which assigns the modules one after another to the processor group with the minimal global execution time. This algorithm has been shown to have a worst-case suboptimality bound $4/3$ [5].

The result of the scheduling step is a global MDG annotated with SDS decision structures at each branching point. To reflect the resulting execution order, the MDG is transformed into a module execution graph (MEG). The MEG is obtained from the MDG by replacing the $n$ successor modules at a certain branching point according to the SDS by $k$ chains of modules, see Figure 4 for an example. Chain $i$ contains the modules that are assigned to processor group $i$, $i = 1, \ldots, k$.

3.2 Assigning processors to modules

The MEG specifies the exact execution order of the modules as it has been fixed in the scheduling substep. In particular, it specifies the number of processor groups that are used for the execution of independent modules and the assignment of modules to the processor groups. But it does not specify the size of the corresponding processor groups. This size is determined in the load balancing substep. Similar to the scheduling substep, the TwoL compiler provides two alternatives for the load balancing decision: an interactive version in which the programmer can specify the size of the processor groups and an automatic version in which the compiler tries to determine the size of the individual groups such that the executions of the modules assigned to the groups terminate at about the same time, thus leading to a good load balance.
of the executing processors. The sizes of the processor groups are determined by a top-down traversal of the MEG. At each branching point, the sizes of the following groups are determined by partitioning the group of available processors into \( k \) groups where \( k \) is the number of successor modules in the MEG.

The result of the load balancing substep is recorded in a group decision structure which is an array with \( k \) entries annotated at each branching point of the MEG with \( k \) successors where entry \( i \) specifies the size of processor group \( i, i = 1, \ldots, k \). The sum of the entries must be smaller than or equal to the number of processors before the group partitioning. The \( i \)th entry of the array is propagated into the \( i \)th branch. At union points, the processor groups of the incoming branches are merged. Similar to the scheduling substep, the automatic version of the module load balancing step must be based on estimates of the runtimes of the modules in the different branches of the MEG for different numbers of executing processors.

The result of the load balancing substep is an execution plan that specifies for each processor \( p \) the order in which \( p \) executes (parts of) the modules assigned to processor groups to which \( p \) belongs during the execution of the entire program. The execution plan is obtained from the MEG with annotated load balancing information by traversing the MEG for each processor top-down, following at each branching point the branch to which \( p \) is assigned and collecting the modules found on the path traversed.

### 3.3 Data distribution

The data distributions for the input and output variables of the different modules have a large influence on the resulting runtime. The TwoL compiler offers two alternatives for the determination of data distributions. The first alternative assumes that the programmer provides data distributions for the input and output variables of the basic modules as well as for their internal variables. This is a common situation when modules from a scientific library like ScaLAPACK [3] or LAPACK-DM are used which are optimized towards a specific data distribution. In this situation, the TwoL compiler uses interference rules of a distribution type system and derives redistribution operations that are required between cooperating modules for a correct behavior.

The second alternative assumes that the data distributions of the basic modules have not yet been determined and that the programmer requests the assistance of the system. The TwoL compiler provides an algorithm that tries to determine the data distributions for the basic modules such that a minimal global execution time for the entire frame program results. To achieve this goal it is not sufficient to consider the basic modules in isolation because this minimizes only the intra-module communication of the basic modules. Because of inter-module communication to arrange redistributions between cooperating modules, optimal distributions of basic modules need not lead to globally optimal solutions.

An analytical approach to compute optimal data distributions for numerical algorithms has been presented in [8]. The approach is based on runtime formulas as described in the next section and parametrized data distributions, i.e., in addition to the machine parameters and the application parameters, the runtime functions contain parameters that describe the shape of the data distributions. The runtime formulas are considered as function of those parameters of the data distributions. The determination of the minimum of those functions leads to parameter values that
correspond to the optimal data distribution. The approach is described in the context of a basic module, but it could be extended to composed modules.

A more general approach using dynamic programming has been presented in [12]. The approach is based on the use of data types for the description of data distributions for composed variables. The approach assumes that composed modules are described by their corresponding module syntax tree. The dynamic programming approach determines (bottom-up for each node of the module tree) a distribution time function that maps each data distribution type of the corresponding module $M$ onto the minimal runtime that can be achieved for this data distribution type. The distribution time function for a module $M$ has the form

$$DT_M : I_M \times O_M \rightarrow N$$

where $I_M$ is the set of possible input distributions of $M$ and $O_M$ is the set of possible output distributions of $M$. In the recursive computation of $DT_M$ all possible input and output distributions of $M$ are considered, but for a fixed combination of input and output distributions, the distributions for the submodules of $M$ are chosen such that a minimal execution time results. No other distribution for these modules needs to be considered to find an optimal solution for $M$. The runtime estimations of basic modules (represented by the leaves of the module tree) are provided by the performance prediction method. For the sequential composition $M = M_1 \circ M_2$ of two modules $M_1$ and $M_2$, the distribution time functions $DT_{M_1}$ and $DT_{M_2}$ of the participating modules are combined, i.e., $DT_M$ of $M$ is computed according to

$$DT_M(\tau_1, \tau_2) = \min\{A, B\}$$

$$A = \min_{\rho \in O_{M_1} \cap I_{M_2}} [DT_{M_1}(\tau_1, \rho) + DT_{M_2}(\rho, \tau_2)],$$

$$B = \min_{\rho \in O_{M_1}} \{DT_{M_1}(\tau_1, \rho) + T_{Re}(\rho \rightarrow \sigma) + DT_{M_2}(\sigma, \tau_2)\} \}$$

where $T_{Re}(\rho \rightarrow \sigma)$ denotes the time for a data redistribution from distribution $\rho$ to distribution $\sigma$. The term $A$ of the minimum operation describes the situation that the output distribution of $M_1$ is used as input distribution of $M_2$. The term $B$ takes into account that the optimal runtime might also come from the composition of data distributions that require a data redistribution in between. Every output distribution $\rho$ of $M_1$ can be converted into a distribution $\sigma$ by applying the redistribution $Re(\rho \rightarrow \sigma)$. Thus, all times $DT_{M_i}(\tau_1, \rho)$ are combined with all times $DT_{M_i}(\sigma, \tau_2)$ and the time for an appropriate data redistribution $T_{Re}(\rho \rightarrow \sigma)$.

4 Examples

In the following, we consider several examples from the area of scientific computing and show how the TwoL compiler can be used for the generation of an efficient parallel implementation. As examples, we consider solution methods for initial value problems of ODEs. These have been used for the numerical solution of physical simulations that are described by time-dependent partial differential equations (PDEs). The PDEs are transformed in a system of ODEs by a spatial discretization.

The IRK method has been used for the solution of a reactor-diffusion system from chemical kinetics that models the reaction of two chemical substances, see [11] for a
detailed description. The reaction of the two chemical substances is described by the following reaction-diffusion equation, see [6]:

\[
\frac{\partial u}{\partial t} = 1 + u^2v - 4.4u + \alpha \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \\
\frac{\partial v}{\partial t} = 3.4u - u^2v + \alpha \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right)
\]

(2)

(3)

for \(0 \leq x \leq 1, 0 \leq y \leq 1, t \geq 0\), and \(\alpha = 2 \times 10^{-3}\). The unknown functions \(u\) and \(v\) describe the concentrations of the two substances. A standard discretization of the spatial derivatives with a uniform grid with mesh size \(1/(N-1)\) leads to the following ODE system of dimension \(2N^2\):

\[
\frac{du_{ij}}{dt} = 1 + u_{ij}^2v_{ij} - 4.4u_{ij} + \alpha(N-1)^2(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{ij}) \\
\frac{dv_{ij}}{dt} = 3.4u_{ij} - u_{ij}^2v_{ij} + \alpha(N-1)^2(v_{i+1,j} + v_{i-1,j} + v_{i,j+1} + v_{i,j-1} - 4v_{ij}).
\]

which can be solved with the IRK method described below.

The DIIRK method (diagonally-implicit iterated Runge-Kutta method) has been used for the simulation of a collisionless electron plasma described by a Vlasov–Poisson system [9]:

\[
\frac{i}{\hbar} \frac{\partial \Psi}{\partial t} = -\frac{\hbar}{2m} \frac{\partial^2 \Psi}{\partial x^2} + e \Phi \Psi \\
\frac{\partial^2 \Phi}{\partial x^2} = \frac{e n_0}{\varepsilon_0} (1 - \Psi \Psi^*)
\]

(4)

with electron charge \(e\) and electron mass \(m\). \(\Psi\) is the unknown Schrödinger wave function, \(\Phi\) is a known potential. A Galerkin method results in the following ODE system:

\[
\frac{d\alpha_l}{dt} = -\lambda_l \alpha_l + \sum_{|j| \leq n} \alpha_l \cdot (\phi_n, h_j, h_l), \quad l = 0, \pm 1, \ldots, \pm n \\
\beta_l = \frac{1}{\lambda_l} (|\psi_n|^2, h_l), \quad l = \pm 1, \ldots, \pm n
\]

where \(\alpha_l(t)\) and \(\beta_l(t)\) are the coefficients of \(\Psi_n\) and \(\Phi_n\) in the Fourier sum

\[
\Psi_n = \sum_{|l| \leq n} \alpha_l(t) h_l, \quad \Phi_n = \sum_{|l| \leq n} \beta_l(t) h_l, \quad n \in \mathbb{N},
\]

and where \(\{h_l\}_{l \in \mathbb{Z}}\) represents an orthonormal basis.

### 4.1 Iterated Runge-Kutta methods

The iterated RK method (IRK) has been introduced in Section 2.2, the module specification and the parallel frame program are shown in Figures 2 and 3. Figure 3 shows that the scheduling has been chosen such that the \(s\) iteration vectors \(\sigma^1, \ldots, \sigma^s\) of the \(m\) corrector steps are computed one after another by all processors available. (In Figure 3, \(\sigma^{1 \ldots s}\) is used as an abbreviation for \(\sigma^1, \ldots, \sigma^s\).) The vector \(\eta\) denotes the approximation vector computed in the current time step. The modules Initialize
and Corrector·irk output the iteration vectors $\sigma^1, \ldots, \sigma^s$ in a block distribution. Because the modules executed afterwards expect the vectors in a replicated way, a redistribution

$$\text{redistr}(\sigma_0, n/p, \text{block}(p) \to \text{replic}(p))$$

has to be performed where each processor contributes $n/p$ elements. An implementation on the iPSC/860 shows that this execution order and this data distribution lead to the best performance. [11] shows how the IRK method can be used for the solution of a reactor-diffusion system from chemical kinetics that describes the reaction of two chemical substances.

4.2 DIIRK methods

The diagonally-implicit iterated Runge-Kutta method (DIIRK) is a method for the solution of stiff ordinary differential equations. The DIIRK method shows an upper-level module structure similar to the IRK method. A module initialize($f$) initializes the iteration vectors $\eta$. Nested for and parfor loops compose the module Newton($F$) that realizes a Newton method for the solution of non-linear equation systems. The Newton module contains calls of other modules, including a module for the solution of a linear equation system with a Gaussian elimination. A module Update combines the $s$ iteration vectors to the final approximation vector $\eta$ of the time step according to the underlying Runge-Kutta method.

The frame program specifies that the $s$ iteration vectors of one corrector step are computed by independent groups of processors where each group contains $p/s$ processors. It is convenient to implement the Newton method in such a way that the result is delivered replicated among the executing processors. Internally, a row cyclic distribution of the iteration vectors is used because this leads to a good performance of the Gaussian elimination method which is used within each iteration step of the Newton method. At the end of each corrector step, a redistribution has to be executed to make the iteration vectors available to all processors. An implementation on the iPSC/860 shows that the described group execution leads to a better performance than a consecutive execution order that solves the independent systems of one corrector step by all processors available one after another [10]. The reason for this lies in the fact that the broadcast operations executed in the Gaussian elimination process are less expensive when executed on independent groups of processors (group broadcast).

5 Conclusions

The implementation of efficient parallel programs for DMMs is still a time-consuming work that discourages many potential users. Parallelizing compilers have been proposed to release these programming difficulties, but their development is still at the beginning, to some extent because of the ambitious goal to present the same programming model to the programmer than for sequential machines.

In contrast to parallelizing compilers, the TwoL programming model requires the programmer to give a specification of the application program that expresses the maximum degree of parallelism. This usually requires a detailed knowledge of the underlying algorithms but not of a specific target architecture. The TwoL system is designed to support the programmer in the derivation of a parallel implementation for
a specific DMM exploiting the available degree of parallelism whenever this results in an efficient implementation. This represents a clean interface between the programmer and the system and assigns each component a manageable piece of work. The programmer is released from knowing the details of the underlying DMM, the system is released from extracting the available parallelism.

References


Integration of control and data parallelism in an object oriented language

Pascale Launay, Jean-Louis Pazat
IRISA, Campus de Beaulieu, F35042 RENNES cedex
e-mail: Pascale.Launay@irisa.fr, Jean-Louis.Pazat@irisa.fr

October 7, 1996

1 Introduction

High Performance Fortran is an attempt to define a “standard” for languages based on data distribution features. HPF [9] has merged ideas from many research projects avoiding the development of too many dialects of Fortran. It is a success because this language have helped researchers and companies to focus their efforts in developing compiling techniques and tools and have brought Data Distribution in the main stream of scientific computing.

However, High Performance Fortran is an extension to Fortran and this language will always be limited to scientific computing (who else uses Fortran nowadays?). Parallelism and distribution features of HPF are only applicable to array structures, so HPF is well suited for regular computations but do not cover the whole domain of scientific computing. Moreover, pure data parallelism is not sufficient for some problems and data distribution is not always the best way to guide distributed code generation.

Attempts to extend this language have been made in the HPF forum, but it is necessary to go beyond the definition of HPF language extensions in order to broaden the field of automatic synthesis of distributed programs. New trends in parallel processing include the use of networks of workstations (NOW) and extends the concept of parallel computers to world wide networks of computers (Metacomputing).

In order to develop new tools for these computing environments we need better portable languages than Fortran or C such as object oriented languages. These languages are now widely used in many areas of computing including high performance computing. Whereas compiling Fortran for distributed memory computers was the challenge of the early 90’s, the new challenge is to use efficiently object oriented languages for computing on networks of workstations.
2 Related Work

Like other sequential languages, object oriented languages can benefit from communications libraries like PVM [3] or MPI for providing an easy way to write parallel programs. In this case, each process is written in an object oriented fashion but the whole program is not object oriented because one can not use inheritance "across" processes. A better way to provide communication in object oriented languages is to use common object brokers to make and receive requests between distant objects. The Object Management Group (OMG) Common Object Request Broker Architecture (CORBA) specification provides a standard for this kind of communication.

Some attempts to add explicit parallelism in object oriented languages are currently being investigated.

Parallelism in C++ [7] can be expressed through parallel statements, parallel loops and a spawn statement. Synchronization is provided through single assignment variables and atomic functions. In Cool [5], parallelism is expressed through parallel functions that execute asynchronously when invoked. These functions can share data and can be synchronized by condition variables (like in monitors) or by mutual exclusion. Parallelism has also been introduced in the Java language [2] in the Thread package. It provides a basic form of asynchronous call (only the run method of a thread is called asynchronously by the start method).

A more natural way to introduce parallelism is to promote objects to the rank of processes. Geib et al [6] introduce new classes in the Eiffel language [11]: Thread, Call and Host. Thread contains the representation and control of parallel activities, Call handles asynchronous calls and Host is used for self running objects. Synchronization between processes can be encapsulated in classes. Caramel [4] introduces processes in Eiffel as active objects. Objects parameters are passed by deep copy (not by reference as usual). There is no shared data between processes, so there is no need for synchronization mechanisms. Interprocess communication takes the form of requests between active objects. The main drawback of these approaches is that an active object modifies its state during its life (like any process does); so, according to its state, it may or may not accept the invocation of one of its method. Another drawback of this approach is that inheritance between active objects is not very useful just because one cannot overload parts of the code without rewriting the whole code.

Meyer states that the active object approach is not the right one: in [12], he suggest to define processors as autonomous threads of control that can handle objects. There are no active objects in the language, but if two objects are handled by different processors they can perform operations in parallel. This is achieved by asynchronous method invocation: an invocation of a method of a separate object is asynchronous, whereas it is synchronous if the object is handled by the same processor.

Another way to introduce parallelism is to encapsulate parallelism in regular
classes of an object oriented language without any extension to the language nor modification of its semantics. EPEE [10] (Eiffel Parallel Execution Environment) contains a set of classes providing the communication and location facilities used to build parallel classes. Common data distribution patterns and computation schemes are factored out and encapsulated in abstract parallel classes, which are to be reused by means of multiple inheritance.

Apart from EPEE, we found no approach using data or task distribution in object oriented languages for generating distributed programs from a sequential expression of an algorithm. Whereas EPEE relies on multiple inheritance, we want to use compiling techniques for code generation.

3 Programming models

At the user level, we provide a parallel language through the use of parallel methods (tasks) and data parallel methods. Like in HPF, a global name space is available at this level. Distribution features are annotations (comments) used to describe data and task distribution. There is no need for the user to describe a complete distribution of the program: the compiler computes the missing ones using rules as the owner compute rule to generate an intermediate level representation of the program. The generated code is a distributed program suitable for execution on a network of workstations.

Our model is implemented in an object language. Tasks are object's methods and data are object's attributes.

3.1 User level programming model

Parallel methods

A parallel method is composed of methods that run in parallel. Atomic tasks are the sequential methods of objects. A task may only modify its own local variables, its parameters, and attributes of its objects.

Calling a parallel method causes the call of each of its components. A parallel method terminates when each of its components is terminated. Synchronization between each component takes place at the beginning and at the end of the parallel method invocation. Methods communicate through shared read-only objects attributes or read only parameters\(^1\). Due to the fact that different attributes can dynamically refer to the same object, checking illegal data sharing is often deferred to run-time.

\(^1\)The way a method accesses its parameters (read-only or read-write) is described in its interface
A parallel method can be written as below:

```c
void tPM(double x, y) { /* parallel method x in, y inout */
    m1(x)
    m2(x, y)
}
```

Though this model of synchronization and communication is restrictive, it ensures that parallel methods are closed for communications. Therefore, it is possible to express nested parallelism: a parallel method can equally be constituted of sequential methods (atomic tasks) or parallel methods.

**Data parallel methods**

In data parallel languages, parallel data types are used to describe data structures that can be accessed globally. In our model, the parallel data structures are represented as *structure* objects (similar to *containers*). They contain a set of data items, methods to access elements and methods to manipulate the structure (add or remove elements). Different kinds of structures can be defined, especially regular structures (arrays, lists...).

In our model, a data parallel method contains one single method and allows to globally access a data structure. When calling the data parallel method `dPM`, a copy of `m` is called to access each element of the data structure.

```c
void dPM(struct X) { /* data parallel method */
    m(x)
}
```

**Integration of task and data parallelism**

Task parallelism (parallel methods) and data parallelism (data parallel methods) can be nested: parallel and data parallel methods are composed of sequential, parallel or data parallel methods.

To unify the expression of task and data parallelism, we define structures of tasks in the same way as structures of data. A structure of tasks is an object composed of a set of methods (tasks), methods for naming and controlling tasks and methods to add or remove tasks. A parallel method is expressed by using a structure of tasks, whereas a data parallel method uses a data structure.

**Distribution**

Having unified the expression for sets of tasks and sets of data, we use *structure* objects to describe distribution of tasks and data: annotations for distribution are expressed on structures, that are divided and distributed on processors or
on memories. The grain of task distribution is the sequential method (i.e. sequential methods are not distributed). One can define alignments between data structures, between tasks structures or between tasks and data structures.

For regular structures, distributions and alignments are expressed by annotations similar to HPF array distributions and alignments.

3.2 Execution model

At the intermediate level, parallel programs contain annotations for the distribution of each data and tasks structure. Ignoring annotations for distribution, this parallel program can be executed in a non distributed environment.

To run in a distributed environment, it must be derived in a distributed program, consisting of communicating tasks distributed on processors, and data distributed on memories.

Structures are fragmented according to the annotations for their distribution. Each fragment [8] is an object (substructure), composed of local elements: each fragment of a data structure is on a memory; methods in the same fragment of a task structure run on the same processor.

4 Conclusion

Our model is intended to provide an easy way to develop distributed applications, with good performances. At the user level, parallelism and distribution relevant to the application are expressed. The distributed program is generated using compiling techniques such as those defined in the Pandore project [1], but in order to obtain efficient distributed programs, the grain of parallelism and distribution (the sequential method) is larger than the statement.

We are implementing a prototype within the Java language. Java is a simple object oriented language. Despite it does not have all the features of the best object oriented languages (multiple inheritance, genericity), it is widely used.

References


Shared Virtual Memory
A Compiler Algorithm to Reduce Invalidation Latency in Virtual Shared Memory Systems

M.F.P. O’Boyle
Department of Computation
UMIST, UK

A.P. Nisbet, R.W. Ford
Department of Computer Science
University of Manchester, UK

Abstract

This paper presents a new compiler algorithm to eliminate invalidation traffic in virtual shared memory using a hybrid distributed invalidation scheme. It aggressively exploits static scheduling and data layout to accurately determine only those instances when invalidation is necessary, thus avoiding the additional read misses of previous schemes. Equations determining precisely what data should be invalidated are presented and followed by the derivation of approximations amenable to compiler manipulation. Compiler-directed invalidation in the presence of arbitrary control-flow is described and the definition of a compiler algorithm is presented. Preliminary experimental results on three programs show that this analysis can drastically reduce the amount of invalidation traffic and write misses.

1. Introduction

Scalable Virtual Shared Memory (VSM) systems often use a directory structure (which typically records the location and the read/write access permissions of copies of data) and invalidation/update based protocols to maintain consistency. In the sequential consistency model with an invalidation protocol an attempt to write a new value is delayed until all remote copies are invalidated. Performance can be degraded both by the delay on the writing node, and by the resulting network latency. In an update model, all remote copies are sent the new data values. Although this helps reduce read misses, it increases the amount of write traffic, may increase read-miss latency and further delays the writing node [20].

One approach to improving performance is to reduce the overhead of invalidation traffic within a write-validate based protocol by using distributed invalidation (DI). DI (also called self invalidation [13]) transfers the responsibility of invalidation from the writing processor to the processors with the remote copies. The writing processor does not incur a write miss and can proceed without stalling as the invalidation of copies is done locally. The DI scheme also has the advantage of reducing network invalidation traffic by removing invalidation and acknowledgement messages. This paper presents a new compiler algorithm to exploit a hybrid DI scheme in the context of invalidation based sequential consistency. This algorithm eliminates invalidation traffic without incurring the read misses of previous schemes. The compiler is aggressive in its analysis as it is does not have to make the conservative assumptions of compiler directed schemes, where the compiler is entirely responsible for coherence. In the case of compile-time unknowns, sequential consistency guarantees correctness.

Existing analyses [4, 2, 6] for compiler directed coherence assume a fork/join model. Parallel regions are separated into epochs by a synchronisation even though there may be no cross-processor dependence between regions. Scheduling within an epoch is considered to be dynamic so it is impossible to determine the location of accesses between epochs. This has the effect of making worst-case assumptions about the location of previous read and writes, generating unnecessary invalidation or misses.

Our compiler [4] exploits data parallelism with owner-computes scheduling. Using compile-time knowledge of data layout and alignment, we have implemented a scheme that determines whether two accesses can cause a cross-processor dependence [17]. We have used this technique to optimise barrier placement and the same analysis can be used to detect dependences requiring invalidation.

By using array section analysis [8, 18] we can accurately determine the data to invalidate, without the need of an additional bit mask per write instruction [14] or invalidating the entire array as required in previous schemes. Although using local information, our scheme approaches the accuracy of global schemes [6] without the run-time overhead of invalidation traffic.

This paper makes the following contributions:

- It introduces a new compiler directed distributed invalidation algorithm to eliminate invalidation traffic in
validation based sequentially consistent VSM.

- It precisely defines what data should be locally invalidated when using an SPMD execution model and static-scheduling to eliminate unnecessary invalidations.

- It presents results illustrating the performance benefit in terms of invalidation traffic reduction and actual execution times on an existing system.

In Section 2 we describe our scheme, discuss related work and present an illustrative example. Section 3 defines what data is to be invalidated when using this scheme. Section 4 describes the impact of control-flow on our scheme and in Section 5 the compiler algorithm is developed. Section 6 details the experiments and results and Section 7 concludes the paper.

2. Distributed Invalidation

This section describes the hybrid scheme and compares our approach with the widely used write-invalidate sequential consistency model and previous approaches to distributed invalidation.

2.1. Hybrid Scheme

We assume sequential consistency and an invalidation based protocol with three states; exclusive (Ez), read-only (Ro) and invalid (I). Our approach is a hybrid one in that we use compiler analysis to implement DI where possible; however, when compiler analysis is difficult or impossible, sequential consistency will ensure correctness and maintain a reasonable level of performance.

Invalidation is required when a processor writes to a location which has valid copies on other processors. If these remote copies are not invalidated then a remote processor may read an ‘old’ or stale data value. In traditional sequential consistency, the directory will point (either directly or indirectly) to all remote copies. Before writing to a location a processor will send invalidation messages to all processors pointed to by the directory. Only when all acknowledgments have been received will exclusive access be granted and the new value written. If, however, each processor invalidates its stale read copies and marks as exclusive the data it will write, then memory consistency is maintained without incurring any invalidation traffic.

We have implemented [9] a call to our VSM system LocalExclusive(addr) which sets local access for the location addr of a variable to exclusive and, by default, updates the directory pointers [16]. This both allows a write to proceed without stalling (as in weak coherency) and removes the associated invalidation traffic. The corresponding call LocalInvalidate(addr) sets a read-only copy of the addr to invalid.

Notice that a write can now occur before the remote copies have been invalidated, however, compiler placement of barriers ensures that no read to invalid data can take place. An important feature of these calls is that after they have completed, the system is left in a coherent state. Thus, in the absence of further compiler optimization the directory will continue to maintain sequential consistency.

2.2. Example

To illustrate the compiler approach developed in this paper, consider the Tred2 fragment (from the EISPACK suite) shown in column 1 of Figure 1. The second column shows the local node program generated by the MARS compiler [1] after column partitioning; lo1, hi1 are the introduced lower and upper bounds of the local data space. The barriers are introduced by the compiler due to the detected cross-processor data dependences.

The compiler uses a SPMD execution model, with static data layout and owner-computes scheduling and detects that the only cross-processor data dependences (e.g. ) are the flow-dependence on s from (s2, s5, s6) to s7 and the anti-dependence (e.g. ) on d from (s1, s2, s4, s7) to s8 and s9 and on z from s2 to s7. Placing a barrier between s6, s7 and between s7, s8 covers these cross-processor dependences. The owner-computes rule guarantees that all output dependences are within a processor; for further details on barrier placement optimization see [17]. With static data layout and owner-computes scheduling, invalidation is required only where there exists a cross-processor anti-dependence. Arrays d and z must both be invalidated; the code for the invalidation of array d, for example, is shown in column 3 of Figure 1.

2.3. Related Work

The majority of research related to DI has been targeted at scalable cache coherent non-uniform memory access architectures (CC-NUMA). However, the coherence issues for CC-NUMAs and VSM architectures are very similar.

Compiler directed coherence places the entire burden of maintaining cache coherence on the compiler. Some schemes use a compiler controlled directory to help in runtime dependence analysis, whilst others remove the need for a directory altogether [2, 6]. Early work invalidated all cached data at each epoch. More recent schemes have used tags or timestamps to maintain cached data across epochs. This method relies on the compiler analysis for correctness and is limited in the amount of inter-epoch locality that can be exploited by the size of the tag and epoch counters, as well as by the conservative nature of the compiler analysis.

1 after induction variable substitution etc.
<table>
<thead>
<tr>
<th>Original Program</th>
<th>Local Program</th>
<th>Local invalidation of d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do j = 1, n+1-ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1: z(j,n+2-ii) = d(j)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = 1, n+1-ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do k = j+1, n+1-ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2: e(k) = e(k) + z(k,j)*d(j)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = 1, n+1-ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3: gg(j) = e(j) + z(j,j)*d(j)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = 1, n+1-ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do k = j+1, n+1-ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4: gg(j) = gg(j) + z(k,j)*d(k)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = 1, n+1-ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s5: e(j) = gg(j)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = 1, n+1-ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s6: e(j) = e(j)/h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = 1, n+1-ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s7: z(k,j) = z(k,j) - d(j)*e(k)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- e(j)*d(k)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = 1, n+1-ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s8: d(j) = z(n+1-ii,j)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do k = 1, n+1-ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s9: d(k) = d(k)/scale</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (n+2-ii .LE. hil)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (lol .LE. n+2-ii)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = 1, n+1-ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s: z(j,n+2-ii) = d(j)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndIf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = 1, n+1-ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do k = max(j+1,lol),min(hil,n+1-ii)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e(k) = e(k) + z(k,j)*d(j)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = max(1,lol),min(hil,n+1-ii)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gg(j) = e(j) + z(j,j)*d(j)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = max(1,lol),min(hil,n+1-ii)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do k = j+1, n+1-ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gg(j) = gg(j) + z(k,j)*d(k)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = max(1,lol),min(hil,n+1-ii)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e(j) = gg(j)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = max(1,lol),min(hil,n+1-ii)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e(j) = e(j)/h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call barrier1()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = max(1,lol),min(hil,n+1-ii)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e(j) = e(j)/h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call barrier2()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do j = max(1,lol),min(hil,n+1-ii)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d(j) = z(n+1-ii,j)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do k = max(1,lol),min(hil,n+1-ii)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d(k) = d(k)/scale</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndDo</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1. Compiler directed invalidation example
Fast selective invalidation (FSI) [2] will invalidate all the data, read and write, accessed within an epoch while time stamping (TS) strategies [2] will invalidate all write data after the parallel loop, if in the next epoch the data is read. In the example in Figure 1, FSI will invalidate the entire contents of different arrays 21 times while TS reduces this to 8. The life span [3] strategy will only invalidate read data at the next write in a different epoch. In our example, this restricts the invalidation to the entire arrays after the epochs surrounding s5, s7 and s8. A more sophisticated form of analysis [4], based on reads after RDS (relaxed deterministic sequence)\(^2\) examines each epoch for the first occurrence of an upwardly exposed read which could be preceded by an RDS. Such reads will require a special cache read to determine their validity. In [7], vectorisation is used to minimise the overhead of redundant invalidation when using RDS analysis. In our example, the data will be considered invalid on the read access to gg in s5, e in s6, e in s7, z in s8 and d in s9. Thus previous compiler based schemes invalidate or consider state between 3 and 21 of the array accesses, while our scheme accurately determines that there are just 2. In fact, this is a generous analysis. When using KAP [12], an autoparalleling compiler, it parallelises the k loop of s2, so the inner rather than the outer loop defines the epoch boundary. As e is both read and written in this smaller epoch, the above schemes would all increase the amount of invalidation performed by an order of magnitude.

A directory based method, Dynamic Self Invalidation [13], developed for CC-NUMAs, adds extra state to tag copies which are to be invalidated. The invalidation is part of the directory protocol, so no compiler or human intervention is needed. A hybrid technique, CICO [10], lets the programmer, or a profile based tool, annotate the program with checkin and checkout directives to inform the directory that it should invalidate data. In contrast to compiler directed coherence, these directives are only performance hints; the directory hardware is still responsible for correctness. The use of hybrid techniques, where the compiler is used to eliminate unnecessary coherence actions rather than relying on it to maintain coherence, is also investigated by Mounes-Toussi et al. [15] for CC-NUMAs. They note the critical role of (but do not develop) compiler analysis and show that if static scheduling and accurate dependence analysis were available, then their coherence scheme would show marked improvement across all the programs they examined from the Perfect Benchmark suite. The analysis described in this paper will be of use in their scheme. Our hybrid scheme [9] allows exploitation of the SPMD execution model; however, if a fork/join model is preferred, the above techniques may also be used.

3. What to Invalidate

A processor must invalidate all its copies of pages that will be written by others and ensure exclusive access to the pages it will write. This section is concerned in precisely determining what pages should be modified when using owner-computes scheduling. It develops a set of equations which are used to determine the local action each processor must take with respect to local data to maintain coherence, without incurring invalidation traffic. This analysis is related to that used in communication optimisation for distributed memory machines [11]. In the remainder of this paper, we assume that cross-processor dependence analysis and barrier placement have taken place and our discussion is restricted to data involved in cross-processor anti-dependences.

3.1. Equations

Before deriving the data to be marked invalid and exclusive, it is necessary to define a number of primitives. Let an action \(i\) be a read or write operation occurring after action \(i - 1\) and before \(i + 1\). This ordering must be ensured by program synchronisation or local sequential ordering.

Definition 1. Let \(W_i\) and \(R_i\) be the set of all the pages (global data) written and read respectively in action \(i\).

Definition 2. Let \(W_i\) and \(R_i\) be the set of all local pages written and read respectively in action \(i\).

Definition 3. Let \(E_i\) and \(R_i\) be the set of local pages in exclusive and read state respectively after action \(i\).

A superscript is used, if necessary, to distinguish between actions occurring on different processors e.g. \(W_i^1\) and \(W_i^2\) refer to the local data written on processors 1 and 2 respectively.

We use an owner-computes rule and static scheduling. To eliminate multiple writer false sharing, arrays are padded and partitioned along page boundaries where necessary [15]. Therefore, each processor writes to the same data throughout the life time of the program and no two processors write to the same pages. We also assume no read/write false sharing when invalidating copies and making local data exclusive.

The state of a page after a particular action is dependent on its previous state and on the action. After a write, the local pages in exclusive state will be increased by any pages written locally that were not previously marked as exclusive.

\[E_i = E_{i-1} \cup W_i\]

Remote processors will not write to local data previously in exclusive state. Thus, remote writes do not invalidate or effect local data in exclusive state.
The number of local pages in read state will be decreased by writes occurring across the processors, if they refer to local pages previously in read state i.e.

$$R_{O_i} = R_{O_{i-1}} - (R_{O_{i-1}} \cap W_i)$$

Let $p$ be the number of processors and $z$ be the processor id of the local processor. After a read action, the number of local pages in read state will be increased by other processors reading $z$'s local pages previously in exclusive state and by the remote pages read by the local processor $z$.

$$R_{O_i} = R_{O_{i-1}} \cup \left( \bigcup_{k \in \{1, \ldots, p\}} (R_k \cap E_{x_{i-1}}) \cup (R_k \cap (E_{x_{i-1}} \cap R_k^f)) \right)$$

Note that from this equation, reads by a processor to its own local data do not change its state from exclusive to read state, only reads by other processors will do this. The directory structure guarantees that remote reads are not made to invalid core and therefore reads to previous invalid data need not be considered. The local pages in exclusive state are decreased by reads from other processors to local pages which were previously in exclusive state.

$$E_{x_{i-1}} = E_{x_{i-1}} - \left( \bigcup_{k \in \{1, \ldots, p\}} (R_k^f \cap R_k) \right)$$

Once again the pages in exclusive state are unaffected by local reads to local data. We now introduce the equations which are used to define coherence state transition and will be used by the compiler algorithm to perform distributed invalidation.

Definition 4: Let $LE_{x_i}$ and $LL_i$ be the local pages to be set to exclusive and invalid state respectively due to action $i$.

The local pages to be made exclusive are those which will be written locally but are not already exclusive.

$$LE_{x_i} = W_i - (E_{x_{i-1}} \cap W_i^f)$$

The local pages to be invalidated are those that will no longer be in either a read or exclusive state.

$$LL_i = R_{O_{i-1}} - (R_{O_i} \cap R_{O_{i-1}}) - (E_{x_i} \cap R_{O_{i-1}})$$

By set manipulation (2) can be expressed more usefully as equation (3) where those local pages which are in a read state should be invalidated if written to by remote processors.

$$LL_i = (W_i - W_i^f) \cap R_{O_{i-1}}$$

The above equations (1) and (3), if honoured by the compiler, will eliminate invalidation traffic and unnecessary misses.

3.2. Properties

Sequentially consistent write-invalidate protocols have the following property:

Property 1 Any page, $z$, that is in exclusive state on processor $z$, has been written by processor $z$. All read copies on other processors will have been invalidated.

It can be shown that the equations (1) and (3) also satisfy this property. However, as we use a hybrid scheme it is not necessary for those invalidations, inserted by the compiler, to satisfy this property to maintain memory consistency. The following weaker property is all that need be satisfied:

Property 2 Any page $z$ that is marked exclusive on one processor is marked as invalid on all other processors containing that page.

If the compiler overestimates the region to mark exclusive, there will be excess invalidation and later unnecessary read misses. If it underestimates, then there will be invalidation traffic. As long as this property holds, however, the memory will be sequentially consistent at the barrier immediately following the write. This property is used in section 5 when presenting an approximate scheme.

3.3. Example

The previous subsection defines the data to be invalidated in terms of set algebra which is not suitable for compiler manipulation. However, the elements belonging to these sets are derived from the array sections generated from the iteration spaces and access functions. In [18] we described how the set based expressions may be translated into affine constraints (polyhedra) and access functions. We will use this representation in deriving the data regions, of array $d$ to be invalidated for the program shown in Figure 1. To denote the elements or array section accessed, we use the (iteration space, access) pair, e.g. the elements of $d$ accessed in $s_2$ are

$$[(1 \leq j \leq n + 1 - ii), [d(j)]]$$

which are also the local copies of $d$ in read state before $s_8$ ($R_{O_7}$). We assume for the purposes of this example that the remaining elements of $d$ are in exclusive state, i.e.

$$E_{x_7} : [\max(l_0, 1 + n + 2 - ii) \leq j \leq \min(h_{i1}, n)], [d(j)]$$

The data to be written in $s_8$ is determined by its local iteration space i.e.

$$W_8 : [\max(l_0, 1) \leq j \leq \min(h_{i1}, n + 1 - ii)], [d(j)]$$

Substituting these values into equations (1) and (3) gives

$$LE_{x_8} : [\max(l_0, 1) \leq j \leq \min(h_{i1}, n + 1 - ii)], [d(j)]$$
\[
\begin{array}{|c|c|c|}
\hline
\text{Original Program} & \text{SSA for a} & \text{SSA for b} \\
\hline
\text{Do } j = 1, n \& \text{ Do } i = \max(1, l_1), \min(n, h_1) \& \text{ Enddo} \& \text{ Enddo} \\
1: x(i) = x(i) + a(i) \& \text{ Enddo} \\
\text{Enddo} \& \text{ Do } t = 1, n \& \text{ Call barrier}() \& \text{ Do } i = \max(2, l_1), \min(n, h_1) \& \text{ Enddo} \& \text{ Enddo} \& \text{ If } (P) \text{ then} \& \text{ Do } i = \max(1, l_1), \min(n, h_1) \& \text{ Enddo} \& \text{ Else} \& \text{ Endif} \& \text{ Call barrier}() \& \text{ Do } i = \max(1, l_1), \min(n, h_1) \& \text{ Enddo} \\
3: c(i) = i + t \\
4: x(i) = b(n-i+1) \\
\text{ Enddo} \& \text{ If } (P) \text{ then} \\
3: c(i) = i + t \\
4: x(i) = b(n-i+1) \\
\text{ Enddo} \& \text{ Else} \& \text{ Endif} \\
6: \text{ Enddo} \\
7: y(i) = c(i+1) \\
\text{ Enddo} \\
\text{ Enddo} \\
\hline
\end{array}
\]

**Figure 2. Control-flow**

\[LL_b : [1 \leq j \leq \min(\max(lo_1, 1), n + 1 - i)] \cup \{d(j)\} + [\max(\min(h_1, n + 1 - i) + 1, 1) \leq j \leq n + 1 - i] \cup \{d(j)\}\]

Thus, there is one region to mark exclusive and two to mark invalid. The code for this is shown in column 3 of Figure 1.

Note there is a step s, which denotes page size; invalidation need only be done on a page rather element-wise basis. The impact of page size on code generation is discussed in [18]. Enumerating all previous reads and writes even for simple programs would be very expensive. Section 5 develops approximations to equations (1) and (3) amenable to compiler manipulation.

4. Control-flow

The coherence state is defined recursively in terms of previous states. For general programs, it is therefore necessary to have a systematic method to determine previous actions in programs with possibly complex control-flow. To determine previous read and write actions, we use a modified version of the gated Static Single Assignment (SSA) form described in [21]. Due to space restrictions, only a very brief overview of this approach is given. At certain points in the program pseudo-functions \(\mu, \gamma, \eta\) are inserted allowing the tracing back of reaching dependences.

- \(\mu\): merges values at the head of a loop. On the first iteration, values before the loop body are selected; afterwards, values from the previous iteration are selected
- \(\gamma\): merges values after a conditional selection
- \(\eta\): selects the last value at the end of a loop if non-empty

Construction of SSA-forms of programs is well documented [5]; the benefit of gated SSA is that the conditionals are preserved. For our application, we are not interested in the values of reaching dependences, but the access regions of data involved in cross-processor anti-dependences, which is determined by their enclosing local iteration space and access function after partitioning. In [21], an elegant method to determine previous actions, by demand-driven back propagation, is described.

Consider the local node program in Figure 2 and the modified gated SSA form of the program for the access to \(a\) in the second column (where \(a = \mu, \text{gamma} = \gamma\) and \(\text{eta} = \eta\)). For each access, the previous write and read accesses are available in the pair \{prev.write, prev.read\}. For instance, the previous access to array before the write to \(a_3\) is given by \{a2.write, a2.read\}. This can be expanded by substitution (given \(n > \max(1, lo_1)\)).

\[
\begin{align*}
\text{a2.write} & = \mu((t = 1, na), a1.write, a4.write) \\
& = \mu((t = 1, nt), a3.write) \\
& = \mu((t = 1, nt), 0, a3) \\
\end{align*}
\]
Thus there was either no previous write (\(t = 1\)) or it was as3 on a previous iteration (\(t > 1\)). Translating into a system of inequalities [18] gives:

\[
\mu((t = 1, nt), 0, a3) = \begin{cases} 
\{(max(1, lo1) \leq i \leq min(n, hi1)) \\
\land (t \geq 2)\}, [a3(i)] 
\end{cases}
\]

The previous read action can be found similarly to be \(\mu((t = 1, nt), a1, a4)\) or

\[
\{(max(1, lo1) \leq j \leq min(n, hi1)) \land (t = 1)\}, [a1(j)] + \\
\{(max(1, lo1) \leq i \leq min(n, hi1)) \land (t \geq 2)\}, [a4(i-1)]
\]

Note that the value of \(t\) is decremented by the number of previous iterations considered. If we wish to traverse further back, previous actions may form a recurrence [21]. Reaching read actions in the presence of \(\alpha\) construct actions are handled in a similar manner. The previous read action before the write to \(b3\) in statement \(s9\) (as shown in column 3 of Figure 2) depends on the conditional \(P\), i.e.

\[
\{(max(1, lo1) \leq i \leq min(n, hi1)) \land P\}, [b(n - i + 1)] + \\
\{(max(1, lo1) \leq i \leq min(n, hi1)) \land \neg P\}, [b(i + 1)]
\]

Determining previous accesses within loop nests and conditionals is more complex. If condition \(P\) is invariant and statement \(s3\) is reached and a write action to \(a\) takes place then there can be no previous reads, as they may only occur in statement \(s3\) when \(P\) must be false. Difficulties arise when \(P\) is not loop invariant. Determining the last iteration of \(t\) when \(P\) was false would require an array of values of \(P\), \(P[1 . . i]\), to be maintained. The previous read would be defined as:

\[
\{(max(1, lo1) \leq i \leq min(n, hi1)) \\
\land \neg P[k] \land P[k + 1 . . i]\}, [c(i + 1)]
\]

More complex control-flow structures make this impractical and as our compiler is, at present, restricted to systems of affine constraints involving iterators and parameters, it does not consider general predicates. However, as we have a hybrid scheme, approximations can be made while guaranteeing that correctness is preserved.

5. Algorithm

Determining all previous read and write actions using gated SSA will lead to an explosion of terms and therefore a number of approximations are necessary for a practical algorithm. This section describes a tractable technique to determine which pages to invalidate and a compiler algorithm to perform these tasks.

1. If a write access \(w_i \in W\) of each shared variable \(V\) with a non-linear access remove \(V\)

2. \(V\) write access \(w_i \in W\) of each remaining shared variable \(V\)

(a) determine all dominating cross-processor reads \(R\) after all possible previous writes, \(w_{i-1}\)

(b) \(V\) read access \(r_j \in R\)

   i. \(pre\text{ }read_i := pre\text{ }read_i \cup r_j\)

   ii. If \(w_i\) does not cover read \(r_j\) then

      A. \(not\text{ }cover_i := not\text{ }cover_i \cup r_j\)

3. \(V\) write access \(w_i \in W\) of each shared variable \(V\), if \(not\text{ }cover_i \neq \emptyset\) then

   (a) If \(\exists w_i\) that strictly postdominates \(w_i\) then

   i. \(V\) read access \(r_j \in not\text{ }cover_i\)

   A. If \(w_i\) cover \(r_j\) then

   B. * \(pre\text{ }read_i := pre\text{ }read_i \cup r_j\)

   C. * \(not\text{ }cover_i := not\text{ }cover_i \cup r_j\)

4. \(V\) write access \(w_i \in W\) of each shared variable \(V\) determine

   (a) \(V\) read access \(r_j \in pre\text{ }read_i\), \(R = \cup r_j\)

   (b) \(L_{\text{\small{1}}} = (w_i - w_i) \cap R\)

   (c) \(L_{\text{\small{2}}} = w_i \cap R\)

5. \(V\) write access \(w_i \in W\) of each shared variable \(V\)

   (a) Insert conditional invalidation before the write access \(w_i\)

Figure 3. Algorithm

5.1. Approximate Read and Exclusive State

The major problem of generating a correct but efficient implementation is in determining precisely the state of memory before a write. The equations in section 3 define precisely what data is to be invalidated in terms of all previous read and write actions. Unfortunately, in the general case, enumerating all previous actions is impractical, leading to an explosion in the number of terms and hence approximations must be made. In this section we start at the point where a write will take place and restrict our attention to those reads that have occurred since the last output dependence or write.

Let the previous write action be considered as taking place at \(i = 1\). Thus there are \(i - 2\) read actions between write action 1 and the write action \(i\), under consideration.

Let \(R_i\) denote the read actions since the last write, i.e.

\[
R_i = \bigcup_{k \in \{2, . . . , i - 1\}} \left( \left( \bigcup_{k \in \{2, . . . , i - 1\}} R_k^2 \right) \cup (R_k^1 \cap (R_k^1 \cap R_k^2)) \right)
\]
As there are only read accesses, it can be shown that:

\[ R_{\Omega} = R_{\Omega_0} \cup \overline{R_i} \]

The data to be invalidated, defined in equation (2) can therefore be rewritten as:

\[ L\overline{I}_i = (W_i - W_i) \cap R_{\Omega_0} \cup (W_i - W_i) \cap \overline{R_i} \]

It can be shown that the data to be marked exclusive can also be expressed as:

\[ LE\overline{E}_i = W_i - (E\overline{E}_i \cap W_i) + (W_i \cap \overline{R_i}) \]

If we assume that all data accessed in actions \(2..i-1\) were in an exclusive state when \(i = 1\), i.e.

\[ (W_i - (E\overline{E}_i \cap W_i)) = 0, R_{\Omega_0} = 0 \]

then equations (1) and (3) can be rewritten as:

\[ LE\overline{E}_i = W_i \cap \overline{R_i} \quad (4) \]
\[ L\overline{I}_i = (W_i - W_i) \cap \overline{R_i} \quad (5) \]

where \(LE\overline{E}_i, L\overline{I}_i\) denote approximations to the actual data to be marked exclusive and invalidated under the assumption that data was previously in exclusive state. These definitions satisfy the property 2 described in section 3 and maintain memory coherence.

If we assume that all reads are covered by the first write post-dominating them, then at each write action, only reads involved in a cross-processor anti-dependence after the preceding write need be handled. From this assumption, after each write, all the data is in exclusive mode and it is relatively straightforward to determine the data to be marked exclusive/invalid.

Clearly, this assumption is not true in general and when a write does not cover a read, we make the decision that it is explicitly considered by the immediately post-dominating write. If there does not exist a unique post-dominator, or if it too does not cover the read, we can either ignore it, possibly incurring later invalidation traffic, or kill the dependence by invalidating all read copies even though the write access does not cover it. This is at the possible expense of additional read misses. In our algorithm, we choose not to incur unnecessary reads misses at the potential cost of invalidation traffic. Thus, we guarantee that we incur no more read misses than a sequential consistent scheme and should significantly reduce (but not entirely eliminate) invalidation traffic. Later work will consider a more sophisticated approach.

5.2. Description

The algorithm in Figure 3 is based on the approximations described above and makes simplifying assumptions in the presence of non-linear array access regions and compile-time unknown control-flow. In the first step, if a variable has non-linear array access regions, we resort to sequential consistency for that variable; this is the benefit of a hybrid model. There is always a reasonably efficient fall back position that maintains memory consistency. The problem of considering all previous array accesses is solved by starting at the point where the write will take place and considering only previous read actions after the most recent write on all possible reaching control paths. This is achieved by associating a set preread with each write access. Each set contains only those reads occurring after the previous write. As a write access may not cover all of the reads in preread a second set notcover is generated and is used in the next step. Step 3 considers those accesses that are not covered by the write access and determines if subsequent write accesses will cover the dependence. To prevent an expensive search of future write actions, only write accesses that strictly post-dominant are considered. Those read accesses remaining in notcover after step 3 may, however, cause invalidation traffic if later write accesses refer to them. Step 4 generates the invalidation sets based on equations (4) and (5) which are then inserted into the program in step 5. Once the data accesses to be considered for invalidation have been determined, we place coherence calls immediately before the appropriate write. As this follows the barrier nearest the sink of the dependence, there cannot be any remaining read requests.
<table>
<thead>
<tr>
<th>Seq. Consistent</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Misses</td>
<td>71147</td>
<td>192401</td>
<td>465473</td>
</tr>
<tr>
<td>Write Misses</td>
<td>72177</td>
<td>109671</td>
<td>135315</td>
</tr>
<tr>
<td>Invalidations</td>
<td>71141</td>
<td>192365</td>
<td>465269</td>
</tr>
<tr>
<td>Compiler Invalid</td>
<td>71147</td>
<td>192401</td>
<td>465473</td>
</tr>
<tr>
<td>Misses</td>
<td>1546</td>
<td>1558</td>
<td>1582</td>
</tr>
<tr>
<td>Invalidations</td>
<td>510</td>
<td>1530</td>
<td>3570</td>
</tr>
</tbody>
</table>

**Figure 6. Tred2**

This algorithm is conservative in the presence of non-affine terms and does not consider procedures or subroutines. They can be ignored at present as the hybrid scheme guarantees correctness, but later work will need to consider inter-procedural analysis.

6. Experiments

The compiler algorithm has been partly implemented in MARS [1] - an autoparallellising compiler based on an extended linear algebraic representation of Fortran programs. Cross-processor, cover, and kill dependences have been implemented, as have dominator and post-dominator determination. Gated SSA has not been implemented within MARS, so this information was added by hand. The target architecture for the experiments was a 30 node EDS prototype [19] running the EMEX operating system which provides a fixed-distributed scheme for the management of VSM on a 4KByte page basis.

6.1. Results

We present performance results for three applications, an iterative solver Mvm (N=1024), Jacobi (N=512) and Tred2 (N=512) a benchmark from the EISPACK suite. These are executed on 1 to 16 processors, except for the case of Tred2, which was not executed on 16 processors due to hardware problems with our EDS prototype.

Invalidation Figures 4, 5 and 6 present counts of read misses, write misses and invalidation traffic with and without distributed invalidation for the three programs. The invalidation traffic for Mvm and Jacobi is entirely eliminated without incurring any additional read misses. The number of write misses is drastically reduced, those remaining being unavoidable cold start misses. For Tred2, the compiler reduces the amount of invalidation and write misses by over 97%. On closer inspection of the misses associated with each array, the number of invalidations is unaltered for one of the arrays. This is due to the algorithm assuming that previous writes have covered all previous read accesses. If the algorithm is altered for this dependence, such that all dependences are explicitly killed by over invalidating at each write, the write misses (apart from coldstarts) and invalidations are entirely eliminated. In this case the number of read misses is not increased, but if such a decision were used throughout the program for all arrays, there would be an increase in read misses.

Execution Time Although the primary purpose of these experiments is to demonstrate how the algorithm can reduce invalidation in a hybrid sequentially consistent scheme, it is interesting to see its impact on actual execution time when this scheme is implemented on a real system. Figure 7 shows the execution time for standard sequential consistency (Seq. Cons.) and compiler invalidation (Comp. Inv.) for Mvm, Jacobi and Tred2. Compiler-inserted invalidation enables a finer-grained synchronisation optimisation [16] to be used in Mvm and Jacobi and is labelled optimised in Figure 7. The performance improvement for Mvm and Jacobi (not including initialisation) varies from 0.26% to 46.78% across 1 to 16 processors. The communication within Jacobi is restricted to border elements between processors, while in Mvm an entire vector is read (and later invalidated) by all processors. For 16 processors, there is over 10 times as much invalidation traffic for Mvm as for Jacobi and thus Mvm benefits to a larger extent with its removal.

Although the compiler reduces the execution time for Tred2 by between 24% and 41% this is not sufficient to prevent the slowdown of this application. This slowdown is due to the large number of read misses and the current inefficient implementation of EMEX. Nevertheless, near elimination of invalidations has improved execution time and more efficient implementations would show a corresponding improvement.

7. Conclusions and Further Work

This paper has described compiler analysis for a hybrid distributed invalidation scheme, making explicit use of compile-time knowledge of scheduling and data layout. We applied this algorithm to some sample programs and showed a dramatic reduction in invalidation messages, without the increase in unnecessary misses of previous compiler-controlled schemes. The method is applicable to any cross-processor anti-dependence, in the absence of false sharing.

Further experimentation on a larger set of problems is needed to validate our algorithm. Future work will focus on implementing gated SSA in our compiler, investigating the trade-off between invalidation traffic and unnecessary miss optimisation and extending our work to deal fully with false sharing. We also intend to consider write-update protocols
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mvm</td>
<td>1364.3</td>
<td>704.5</td>
<td>384.2</td>
<td>251.7</td>
<td>233.4</td>
</tr>
<tr>
<td>Comp. Inv.</td>
<td>-</td>
<td>704.5</td>
<td>383.2</td>
<td>234.1</td>
<td>198.5</td>
</tr>
<tr>
<td>% reduction</td>
<td>0</td>
<td>0.58</td>
<td>0.26</td>
<td>6.99</td>
<td>14.95</td>
</tr>
<tr>
<td>Optimised</td>
<td>-</td>
<td>688.2</td>
<td>362.3</td>
<td>191.6</td>
<td>124.2</td>
</tr>
<tr>
<td>% reduction</td>
<td>0</td>
<td>2.31</td>
<td>5.70</td>
<td>23.87</td>
<td>46.78</td>
</tr>
<tr>
<td>Jacobi</td>
<td>1670.8</td>
<td>856.1</td>
<td>450.5</td>
<td>242.1</td>
<td>145.4</td>
</tr>
<tr>
<td>Comp. Inv.</td>
<td>-</td>
<td>844.4</td>
<td>434.9</td>
<td>228.2</td>
<td>128.4</td>
</tr>
<tr>
<td>% reduction</td>
<td>0</td>
<td>1.37</td>
<td>3.46</td>
<td>5.74</td>
<td>10.46</td>
</tr>
<tr>
<td>Optimised</td>
<td>-</td>
<td>839.7</td>
<td>421.0</td>
<td>219.2</td>
<td>116.6</td>
</tr>
<tr>
<td>% reduction</td>
<td>0</td>
<td>1.92</td>
<td>6.55</td>
<td>9.46</td>
<td>18.68</td>
</tr>
<tr>
<td>Tred2</td>
<td>431.9</td>
<td>1516.0</td>
<td>1546.3</td>
<td>1948.5</td>
<td></td>
</tr>
<tr>
<td>Comp. Inv.</td>
<td>-</td>
<td>890.7</td>
<td>1119.6</td>
<td>1469.0</td>
<td></td>
</tr>
<tr>
<td>% reduction</td>
<td>0</td>
<td>41.0</td>
<td>56.4</td>
<td>24.6</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7. Execution Times

and investigate compiler analysis for other VSM optimisations such as latency-hiding post-storing and develop a unifying approach to compiling for VSM.

References


Automatic Performance Analysis for Shared Virtual Memory Systems

M. Gerndt, A. Krumme
Research Centre Jülich (KFA)
Central Institute for Applied Mathematics
52425 Jülich, Germany
{m.gerndt, a.krumme}@kfa-juelich.de

Abstract

Programming distributed memory multiprocessors requires program parallelization as well as program optimization with respect to data locality. SVM-Fortran is a programming language for shared virtual memory architectures with special language features for specifying the distribution of parallel tasks onto the processors. It is realized on top of a shared virtual memory implementation on Intel Paragon. A programming environment provides performance analysis tools helping the user in the optimization of data locality. This paper outlines the environment, describes the basic concepts of the performance analysis support, and presents a design for the automation of performance analysis.

1 Introduction

Parallelization of sequential programs for massively parallel computers with distributed memory is simplified by languages providing a global name space. SVM-Fortran (SVMF) is a shared memory parallel programming language based on Fortran 77. It provides additional language features for scheduling loop iterations and parallel sections onto the processors. The target architectures of SVM-Fortran are distributed memory systems with a global address space and with dynamic replication and migration of data among the processors via local caches or Shared Virtual Memory (SVM) [1]. This data migration allows to optimize data locality by optimizing data reuse which results from a well-done work distribution onto the parallel processors.

The primary target system of SVM-Fortran is the Intel Paragon. The Advance Shared Virtual Memory system (ASVM) is a replacement of the XMM system in the MACH 3.0 micro-kernel and implements the global address space. The SVM-Fortran compiler transforms SVM-Fortran into Fortran 77 code which accesses shared memory objects via the UNIX System V shared memory interface. This code is then compiled by the native Fortran compiler.

Performance analysis tools are required for understanding the run-time behavior of an application. These tools have to be source-code-based and have to provide automatic user guidance to be applied by the users. In addition, detailed information must be
obtainable without forcing the user to suffer from enormous trace files. The lack of solutions for these requirements are at least one important reason for the reluctance of users to apply current tools. The SVM-Portran programming environment provides performance analysis tools taking into account these requirements via selective tracing in combination with an incremental performance analysis approach. Based on the experiences gained with the Optimizer and Locality Analyzer (OPAL) we designed an automatic analysis module for this tool.

This article outlines SVM-Portran and its programming environment. We focus on the performance analysis support and present related work in that area in Section 2. Section 3 gives a short overview of the language extension for global work distribution in SVM-Portran. In Section 4 we describe the incremental analysis concept and its implementation. The design for the automation of the performance analysis procedure is presented in Section 5. The last section summarizes the status of the described work.

2 Related Work

Performance analysis tools for message passing programs, such as Paragraph [2], Pablo [3], and VAMPIR [4], support the user in analyzing enormous amounts of trace data for low-level send/receive operations. The tools allow to filter and visualize trace data after the program run but do not include any knowledge about potential bottlenecks in such programs.

Data parallel languages, e.g. HPF, allow the user to focus on the data distribution. For HPF, profiling tools are available which give coarse information about program performance. Detailed information is only available for the generated message passing code and thus, the relation to the original source code is lost.

The performance analysis tool APPRENTICE [5] for the Craft programming model of the Cray T3D supports the investigation of performance data based on the high-level source code. It provides information on the number of private, local shared, and remote shared load and store operations on the level of basic blocks of the shared memory parallel program. Since the information is only summary information for a whole program run and for all processors it is not sufficient to perform a detailed analysis. In addition, the overhead for performance monitoring is extremely high and thus, the gathered information is only of limited relevance.

The only tool known to the authors providing automatic performance analysis is the Paradyn environment developed at the University of Wisconsin [6]. It applies on-the-fly run-time instrumentation and performance analysis. This tool tries to proof hypotheses on performance bottlenecks by applying appropriate rules, but supports only a very limited number of rules and suffers from performing the analysis while the program is still executing. The investigation has to be done for significant program phases which are unknown to the analysis tool.

3 SVM-Portran Language Summary

SVM-Portran [7, 8] is a shared memory parallel Fortran77 extension targeted mainly towards data parallel applications on shared virtual memory systems and distributed
shared memory systems which provide hardware support for a global address space on top of physically distributed memory. It is based on HPF [9], Fortran-S [10], and KSR Fortran [11]. SVM-Fortran supports coarse-grained functional parallelism where the parallel task itself can be data parallel.

The execution model of SVM-Fortran is an extension of the Single-Program-Multiple-Data (SPMD) model. SVM-Fortran provides the standard features of shared memory parallel Fortran languages as well as specific features to determine the distribution of tasks onto processors. Similar to Fortran-S and KSR Fortran, loop annotations can be used to determine a static or dynamic work distribution scheme. Examples are direct scheduling, such as BLOCK and CYCLIC, as well as dynamic scheduling, e.g. self-service scheduling. Aligned scheduling can be used to schedule iterations following the principle of data locality, i.e. execute an iteration on that processor where the data resides.

Data locality is not a problem to be solved on the level of individual do-loops but is a global problem. Therefore, SVM-Fortran borrowed the concepts of processor arrangements and templates from HPF as tools to specify scheduling decisions globally via template distributions. With predefined scheduling, loop iterations are assigned to processors according to the distribution of the appropriate template element.

Templates can be handled very flexible. They can be created, distributed and redistributed dynamically at any point in the program, and passed via the subroutine interface. SVM-Fortran supports standard distributions (like BLOCK, CYCLIC, and GENERAL_BLOCK), indirect distribution and linked distribution. The programmer can specify for each template element the target processor by an arbitrary integer expression within an indirect distribution. Linked distribution is a form of alignment where a distribution is described via the distribution of another template.

```c
PARAMETER (n=64000)
INTEGER work(3), map(n)
CSVMS PROCESSORS:: p(4)
CSVMS TEMPLATE:: t1(n), t2(n)
C    -- general block distribution --
    work(1) = 18000
    work(2) = 16000
    work(3) = 22000
CSVMS REDISTRIBUTE(GENERAL_BLOCK(work))
CSVMS+    ONTO p::t1
C    -- indirect distribution --
    READ(*,*) map
CSVMS REDISTRIBUTE (i) ONTO p(map(i)):: t2
CSVMS PDO(STRATEGY(ON/Home(t2(i))))
    DD i=1,n
    a(ind(i)) = ...
ENDDD
```

Figure 1: Examples of template distributions and their usage in loop scheduling

Figure 1 gives some examples of distributions. In the first example, the user specifies in a general block distribution how the work (i.e. the loop iterations) should be distributed, i.e. 18000, 16000, 22000, and 8000 template elements are distributed to the 4 processors. In the second example, the user specifies an indirect distribution, the indirection is given with a mapping array map. The template can then be used for loop scheduling as can be seen in the parallel loop.
4 Performance Analysis for SVM-Fortran

SVM-Fortran facilitates program development for SVM systems. It provides easy-to-use language features to implement a global parallelization strategy to support the optimization of programs with respect to data locality. The optimization, which is an important task in the incremental parallelization of programs, is supported by a performance analysis environment. This environment consists of the performance monitoring support integrated into the ASVM, the SVM-Fortran Application Monitor (SAM) [12], and the performance analysis tools OPAL and PARvis.

4.1 Performance Bottlenecks

Performance analysis aims at detecting performance bottlenecks in a program. Either the user or the performance analysis tools have to understand the potential bottlenecks typical for SVM-Fortran programs and the information required to identify these bottlenecks in a program run.

Potential bottlenecks in SVM-Fortran programs mainly consist of:

- missing data locality
- load imbalances
- synchronization
- SVM-Fortran administration

Although each overhead type can be crucial, the data locality bottleneck is used in the remainder of the article to simplify the description of the analysis concepts.

Data locality bottlenecks can be identified in SVM systems in form of the page transfers among the processors. The reasons for pagefaults are manifold: A pagefault occurs when the page is first accessed in a processor, pagefaults also occur due to capacity problems and coherence operations. While the first access to pages is not critical for program performance, capacity problems, and coherence problems can lead to enormous paging overhead.

Capacity problems in the processors mainly have three reasons. First, regions of the code may not be parallelized and thus a single processor accesses the whole data structure. Second, the processor requires more data for the computation of its part of the application domain than fits in its local memory. Third, the work distribution of some parallel loops is not based on the global parallelization strategy and thus, unnecessary access to other parts of the data structures occur.

Pagefaults due to coherence operations result from two main reasons: true sharing and false sharing. While true sharing results from accesses to the same data in different processors, false sharing occurs when different data are accessed which are laid out on the same page. Especially false sharing is frequently the reason for page thrashing, i.e. multiple exchanges of the same page among the same pair of processors.
4.2 Incremental Analysis

Besides the detection of the existence of a bottleneck, the user has to determine the exact area of code and the reason for the bottleneck. This requires very detailed information, e.g. the existence of a data locality bottleneck can be proven based on the amount of pagefault time in the whole program, but the identification of the location requires pagefault information for individual program regions and program variables.

Since the approach of generating detailed information for the whole program run and selecting useful information afterwards in the analysis tool leads to enormous trace files, the SVM-Fortran programming environment is based on selective tracing and on the incremental performance analysis concept outlined in Figure 2.

![Diagram](image)

Figure 2: Incremental performance analysis

Performance monitoring is guided by a trace request file. The requests are handled either directly by SAM or SAM instruments the ASVM to trace information only available for the kernel. Trace data are stored in a trace buffer which is flushed at barriers to reduce the effect of program perturbation.

Based on this selective tracing, the user can first request coarse-grained information and then request more specific information in interesting program regions based on the previously gathered information. This incremental approach reduces the amount of trace data by taking into account already known information.

4.3 Trace Request File

The user specifies in the trace request file which information should be gathered at run-time. The requests are formulated by using routine and variable names from the source code. Examples for such requests are:

- `request (*) local *reg_no(*) : RPFsum, WPFsum`
- `request (*) local foo.reg_(5) : RPFsum(A,B)`
- `request (*) local foo.reg_no(5) : RPF(A)`

The first request specifies that in all processors and all regions, i.e. parallel loops, parallel sections, and subroutines, the sums of the read and write pagefaults have to be
traced. Thus, for each instance of a region the information including an identification of the region is written to a trace file. This demonstrates two features of the SVM-Fortran trace format: source code support and the hierarchy of trace information. The requested sums allow to determine regions with high pagefault rates and, more precisely, the incarnations with high pagefault rates.

The second request determines for a specific region in subroutine foo the individual pagefault sums for specific arrays. This typically leads to the array responsible for a large number of page faults.

The last request generates individual events for each pagefault. This precise information is used in the analysis of the program behavior to determine exactly the array element accessed when the pagefault occurred.

### 4.4 Program instrumentation

The concept of incremental analysis requires a flexible monitoring support. The monitoring system has to provide appropriate information at different levels of granularity. Only if coarse- and fine-grained information is supported, the user can make appropriate decisions.

The performance analysis monitor SAM provides such levels in three different areas:

1. **program regions**
   
   The instrumentation can be requested for individual program regions, such as subroutines, parallel loops and parallel sections. On the coarsest level, information is generated only for the whole program, on the finest level for each region.

2. **execution time**

   SAM provides different resolutions with respect to the execution time. Information can be gathered as statistics written at the end of the program run summarizing the information for all instances of a region in a process. SAM also provides sampling, i.e. the information is generated for every n-th execution of a region thus providing an overview of the behavior of multiple instances. The most precise information results from generating data for each instance.

3. **information detail**

   The trace format supports a hierarchy of trace events [13]. This hierarchy is outlined for the area of data locality:

   (a) read-page-fault-sum, write-page-fault-sum, write-upgrade-sum, page-inder-surn, page-out-pager-sum, page-out-node-sum, page-discard-sum:

   The sums for read and write pagefaults as well as for paging activity due to capacity problems are the coarsest information.

   (b) page-travel:

   This event determines the number of page exchanges among pairs of processors.

   (c) read-page-distribution, write-page-distribution:
The page distribution when a parallel loop starts and terminates can give a good overview of the individual page faults without generating individual records for each pagefault.

(d) read-page-fault-serviced, write-page-fault-serviced, page-in, page-out, page-discard, reduce-access-permission, invalidate-copy:
These are the most detailed trace records generated for individual pagefaults.

In addition to tracing run-time information, SAM provides features to gather information on the overhead induced by the tracing itself. This information is necessary to decide, whether the measured information reflects the actual program behavior.

4.5 The Performance Analysis Tool OPAL

OPAL (Optimizer and Locality Analyzer) [14] supports a menu-driven specification of trace requests that frees the user from the subtle specification syntax. In addition, OPAL analyzes trace data and extracts the most useful statistical data, but does not store all the trace information in memory. These statistical data can be visualized as annotations to the source code in a separate performance column in the main window. This way of presenting information does provide a good overview of the performance results.

If the user needs more information on individual trace regions, he can mark a trace region and request a list of all trace events of a specific type. For example, OPAL presents all shared variables accessed in the selected trace region, and the user can select specific variables for which he would like to see individual pagefaults. The tool then extracts from the trace file of an individual processor only the pagefault information for these variables and the selected program region. Due to the source code information in the trace files, all the information can be requested and presented in form of trace regions and program variables, e.g. the faulting address is translated into the array name and the indices of the array element.

Thus, the tool allows to instrument selected regions of the source code to obtain certain information. Starting from statistical data for these program regions, incarnation-specific information can be analyzed for regions of the source code.

The major drawback of such an environment is that still the user has to navigate through the large number of analysis choices and has to detect bottlenecks by himself. Future tools have to include user guidance as well as the ability to automatically analyze program performance. For frequently occurring performance bottlenecks, the tool should be able to automatically proof the existence of these bottlenecks.

5 Automation of Performance Analysis

Experiences with the tool OPAL in optimizing the performance of SVM-Fortran applications build the basis for an automatic rule-based optimizing approach. One experience is that the optimizing approach in sequential programs, namely concentrating on the regions with the highest execution time, can be applied also in a slightly different way to parallel programs. In order to judge the successfulness of the parallelization, the
user computes the speedup between the sequential and the parallel program. In the case of insufficient speedup he concentrates on the regions with the highest absolute total overhead, calculated as follows:

\[
\text{absolute total overhead} = \text{parallel execution time} - \frac{\text{sequential execution time}}{\text{processor count}}
\]

The analysis of different applications has shown that programs fall between the following two extremes:

1. few regions with a high overhead time
2. many regions with a comparable overhead time

Clearly the first item is easier to analyze than the second. But this difference does not influence the choice of an automatic analysis method for which two approaches are considered.

The first approach assumes that an execution of the sequential version on one processor and the parallel version on more processors can be carried out. The great advantage of this approach is the detection of all overheads, even if they cannot be traced. The absolute overhead time consists of a portion with already traceable overhead types like page faults and synchronization and a portion of unknown or untraceable overhead types like cache misses. With this approach, no time is wasted with the analysis of regions with a large traceable but relatively small total overhead and the user can detect new overhead reasons that should be integrated into the automatic performance analysis.

If the sequential run cannot be performed, the second approach will be chosen. The user concentrates on the region with the highest traceable overhead. Unknown and untraceable overhead cannot be detected and thus no knowledge about how much overhead could be spared in total is achieved.

In the reminder of this article, we assume that the sequential run is possible. After the description of the general optimizing approach, the next section will discuss the implementation concepts of a rule-based automation of the performance analysis.

5.1 Design of an Automatic Analysis Module

This section outlines the components of an automatic analysis module (Figure 3) for the SVMF environment. This module analyzes the available performance information, i.e. run-time information and analysis information determined from the source code, and tries to proof hypotheses about performance bottlenecks. If a hypothesis can be proven it is output as a performance bottleneck. A hypothesis is output as a bottleneck hint if it has a probability below 100% and more refined rules are tested negative or no rule exists which might have a higher probability. The probability is also output as part of the hint.

To proof other hypotheses, more run-time information may be necessary. In this case the analysis module creates new information requests. The information requests identify all useful information and the instrumentation synthesis has to decide which instrumentation to perform during the next program run. This decision takes into account the execution time of the program and the possible perturbation due to the instrumentation. For example, it can predict the amount of perturbation caused by instrumenting
the pagefaults in a program region based on the pagefault counts available for that region.

Figure 4: Hypothesis evaluation and refinement

Figure 4 gives a detailed overview of the evaluation phase. The whole process is based on two sets of rules: refinement rules and proof rules. In a first step, the current set of hypotheses is evaluated based on the proof rules and the available performance information. The proof rules contain knowledge about necessary performance information and are used to test a hypothesis. The evaluation may lead to performance bottlenecks and bottleneck hints as described earlier. If refinement rules are available for a proven hypothesis a number of new hypotheses is generated which are more detailed than the current hypothesis. After the refinement step, the applicable proof rules are determined and the required information is requested.

The analysis support module is based on knowledge about the potential performance bottlenecks and the information required to proof the bottlenecks. The knowledge is implemented as a rule base. This implementation leads to a good documentation of the available knowledge and can be easily extended.
<table>
<thead>
<tr>
<th>Rule</th>
<th>hypothesis</th>
<th>refined hypotheses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TRUE</td>
<td>bottleneck</td>
</tr>
<tr>
<td>2</td>
<td>bottleneck</td>
<td>bottleneck caused by unmeasured overhead</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bottleneck caused by measured overhead</td>
</tr>
<tr>
<td>3</td>
<td>bottleneck caused by measured overhead</td>
<td>locality bottleneck</td>
</tr>
<tr>
<td></td>
<td></td>
<td>load balancing bottleneck</td>
</tr>
<tr>
<td></td>
<td></td>
<td>synchronization bottleneck</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SVMF overhead bottleneck</td>
</tr>
<tr>
<td>4</td>
<td>locality bottleneck</td>
<td>locality bottleneck(R)</td>
</tr>
<tr>
<td>5</td>
<td>locality bottleneck(R)</td>
<td>locality bottleneck(R,P)</td>
</tr>
<tr>
<td>6</td>
<td>locality bottleneck(R,P)</td>
<td>locality bottleneck(R,P,I)</td>
</tr>
<tr>
<td>7</td>
<td>locality bottleneck(R,P,I)</td>
<td>locality bottleneck(R,P,I,V)</td>
</tr>
<tr>
<td>8</td>
<td>locality bottleneck(R,P,I,V)</td>
<td>thrashing(R,P,I,V)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>false.sharing(R,P,I,V)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>true.sharing(R,I,P,V,MODE)</td>
</tr>
<tr>
<td>9</td>
<td>true.sharing(R,P,I,V,MODE)</td>
<td>true.sharing(R,P,I,V,MODE) &amp; sequential(R1)</td>
</tr>
</tbody>
</table>

Table 1: Refinement rules

5.2 Refinement Rules

Refinement rules consist of a proven hypothesis and more precise hypotheses. These hypotheses can include variables that are bound to the appropriate value of the same variable in the performance bottleneck and variables that are bound to a value when the hypothesis is proven. Therefore, multiple bottlenecks can be proven from a single hypothesis with variables. The following rules are examples for refinement rules as shown in Table 1:

Rule 1:
This is the starting rule for the analysis process. It creates the basic hypothesis.

Rule 2:
The hypothesis of the existing bottleneck is refined to an unmeasured and measured overhead.

Rule 3:
It refines the hypothesis according to the different traceable overhead types.

Rule 4:
If a locality bottleneck can be proven for the whole program, this hypothesis will be refined according to the program regions. The goal is to identify program regions that have a locality bottleneck. The variable R is bound to a value when a proof rule identifies a specific region with a locality bottleneck.
Rule 5:
It determines the refinement according to the processes in the parallel program.

Rule 6:
It refines the hypothesis with respect to the instances of the region in the process.

Rule 7:
It refines the hypothesis for a specific instance. It inserts a new hypothesis for a bottleneck resulting from accesses to a single variable.

Rule 8:
It determines the possible reasons for the locality bottleneck. It shows three reasons: thrashing and coherence misses due to true and false sharing. The variable mode can be true, anti, or output, according to the access sequence write-read, read-write, and write-write responsible for the page miss.

Rule 9:
One reason for true sharing is that region R1 has not been parallelized and thus the master process accesses the variable.

5.3 Proof Rules

Proof rules contain the declaration of required performance information necessary to prove a hypothesis and appropriate predicates that represent the kernel of the rule. In addition, a confidence rate is given for each rule which is determined by the expert formulating the rule.

The first rule in Table 2 proofs if a bottleneck exists in the analyzed program.

If a bottleneck exists, the tool proofs if the unmeasured or the measured overhead dominates the total overhead. If the unmeasured part dominates, the automatic bottleneck search will stop. In the other case, described in rule three, the tool concentrates on the region with the highest total overhead.

The fourth rule determines that a bottleneck occurs in r if the region r belongs to the group with the highest total overhead time.

The next rule defines that in region r is a locality bottleneck if the locality overhead dominates in this region.

The sixth rule determines that a locality bottleneck occurs in one specific process if the process belongs to the group with the highest pagefault time for this region. The classification is calculated from the trace data by the analysis tool.

The next rule requires much more detailed information. The page fault sums have to be determined for each instance of the region in this process. The last rule proofs a locality bottleneck for a specific variable.

Table 3 outlines the proof rules for a specific reason of a locality bottleneck, namely thrashing. A vague hint for page thrashing results from the inspection of the pagefault sums of this region in all processes. If there are significant differences and some process p has the same high value as process p thrashing might have occurred.

A stronger hint for page thrashing is the page travel information. If two processes exchanged between each other nearly the same number of pages of that variable and there are processes with much less pagefaults, thrashing could be the reason.
<table>
<thead>
<tr>
<th>Information</th>
<th>predicates</th>
<th>hypothesis</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>execution time for the sequential and parallel program run for the whole execution in each process</td>
<td>Total overhead time &gt; 10% of the parallel execution time</td>
<td>bottleneck</td>
<td>100%</td>
</tr>
<tr>
<td>total overhead and all unmeasured overhead for the parallel program run, whole execution, each process.</td>
<td>mean of the sum of each process of the unmeasured overhead time &gt; 50% of the total overhead time.</td>
<td>bottleneck caused by unmeasured overhead</td>
<td>100%</td>
</tr>
<tr>
<td>total overhead and all measured overhead for the parallel program run, whole execution, each process.</td>
<td>mean of the sum of each process of the measured overhead time &gt; 50% of the total overhead time.</td>
<td>bottleneck caused by measured overhead</td>
<td>100%</td>
</tr>
<tr>
<td>statistical data for all overhead types for the parallel program run, whole execution, all regions, each process.</td>
<td>region r has the greatest portion of the total overhead.</td>
<td>bottleneck (r)</td>
<td>100%</td>
</tr>
<tr>
<td>instrumentation of all traceable overhead types in region r for the parallel program run, whole execution, each process.</td>
<td>locality overhead has the greatest portion of the total overhead.</td>
<td>locality bottleneck (r)</td>
<td>100%</td>
</tr>
<tr>
<td>read/write-page-fault-sum of r for the whole execution, each process + classification of processes according to pagefault time</td>
<td>Process p belong the class with the highest pagefault time</td>
<td>locality (r,p)</td>
<td>100%</td>
</tr>
<tr>
<td>read/write-page-fault-sum of all instances of r in p + classification of instances according to pagefault time</td>
<td>Instance i belongs to the class with the highest pagefault time.</td>
<td>locality (r,p,i)</td>
<td>100%</td>
</tr>
<tr>
<td>read/write-page-fault-sum of r, p, i for all variables accessed in r.</td>
<td>v is the variable with the highest pagefault time.</td>
<td>locality (r,p,i,v)</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 2: Proof rules for refined hypotheses

There are two ways to proof thrashing. The first is based on the page fault sums and thus does not require much trace information. If the number of pagefaults in a process is higher than the number of pages for that variable thrashing must be the reason.

The other proof rule is based on the information of individual page faults. Tracing individual page faults generates more trace data than just tracing the pagefault sums but this information determines precisely which process received the pages and serviced the page faults. Thus, it is easy to proof page thrashing.

The last rule is an example of a rule that combines run-time information and analysis information. A very common situation is that an assignment statement is executed in a parallel loop and the work distribution results in assignments to the same page in different processes. If the number of pages computed in each process is additionally very small, thrashing might occur. If a large number of pages is computed in each
<table>
<thead>
<tr>
<th>Information</th>
<th>predicates</th>
<th>hypothesis</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>read/write-page-fault-sum of r and v for each process</td>
<td>$\exists p$ in this region with a similar number of pagefaults and other processes with much less pagefaults</td>
<td>thrashing(r,p,v)</td>
<td>20%</td>
</tr>
<tr>
<td>page-travel of r and v for each process</td>
<td>$\exists p$/ which received the same number of pages from $p$ as $p$ from $p$/ and there are other processes with much less pagefaults</td>
<td>thrashing(r,p,v)</td>
<td>50%</td>
</tr>
<tr>
<td>read/write-page-fault-sum of r and v for each process</td>
<td>the number of pagefaults is greater than the number of pages</td>
<td>thrashing(r,p,v)</td>
<td>100%</td>
</tr>
<tr>
<td>pagefault event records of r and v for p and p'</td>
<td>a page was exchanged multiple times among the same processes</td>
<td>thrashing(r,p,i,v)</td>
<td>100%</td>
</tr>
<tr>
<td>template distribution and variable mapping information</td>
<td>the region includes an assignment $v(...)=...$ + the work distribution and the page layout are not aligned on page boundaries + the number of pages written in a process is small</td>
<td>thrashing(r,p,i,v)</td>
<td>90%</td>
</tr>
</tbody>
</table>

Table 3: Proof rules for page thrashing

process it is very likely that accesses to the same page in the first iteration of a process and the last iteration of another process will not lead to page thrashing.

The rules in this section demonstrate the integration of the automatic analysis module into the current system supporting manual performance analysis. The existing environment provides all the information required to apply the rules and to detect performance bottlenecks.

6 Summary

This article outlined the performance analysis environment for SVM-Fortran. OPAL supports the whole incremental analysis procedure via an interactive interface, but still the user has to decide which information to trace in the next program run to be able to identify performance bottlenecks. For the most common performance bottlenecks this analysis can be automated. The article presented a design to implement this automation.

Currently, several applications have been parallelized with SVM-Fortran. These experiments lead to a deep understanding of the performance bottlenecks of SVM systems and enabled us to start formulating the rules for the automatic performance analysis module.
References


Cidre: Programming with Distributed Shared Arrays

F. André and Y. Mahéo
IRISA, Campus de Beaulieu, F-35042 Rennes, France
Email: fandre@irisa.fr

Abstract

A programming model that is widely approved today for large applications is parallel programming with shared variables. We propose an implementation of shared arrays on distributed memory architectures: it provides the user with an uniform addressing scheme while being efficient thanks to a logical paging technique and optimised communication mechanisms.

1 Introduction

Modularity and extensibility are two strong points of distributed memory architectures. In these machines, that are composed of interconnected processor-memory nodes, the number of nodes can vary easily so that the power of the machine is adapted to the size of the problem and to the expected performances.

Among these architectures one can find on the one hand machines built with parallel computing in mind like the Intel Paragon or the IBM SP2 and on the other hand, high performance networks of workstations like for instance ATM networks of PCs. This second family makes parallel machines available to a large number of users.

Despite important research efforts and a remarkable improvement in the last few years, programming this kind of machines remains complex because of the distribution of the memories that imposes the use of communication operations.

The programming model that is widely approved today for large applications is parallel programming with shared variables. With this model, the programmer can bring out the parallelism that will yield performance while keeping a global view on the manipulated data structures.

Improving the use of distributed memory architectures relies on the implementation of the model of communicating-through-shared-variables processes. Distributed shared memory systems are a solution to this problem [8, 7, 5]. Shared variables are placed in the virtual memory, hence they are addressed in a uniform way. However, this apparent ease of use is tempered with several drawbacks. The page size is independent from that of accessed variables. This may bring about a larger than necessary amount of communication. Above all, when a page contains several variables accessed in parallel by different processors, some “ping-pong” communication patterns may occur.
Our objective is to propose an alternative to this distributed shared memory system. We have designed an implementation of shared variables that is original and efficient on the following two key-points:

- On the addressing side: it provides the user with an uniform addressing scheme for his variables but the interpretation in terms of local addresses in distributed memories is optimized thanks to a logical paging technique.

- On the communication side: we use message passing for required elements but we largely take benefit from vectorization techniques, connected to our logical paging mechanism, in order to reduce the number of messages and the amount of transferred elements.

The remaining of this paper expounds the abovementioned mechanisms. Section 2 gives a global presentation of the programming environment and details the different levels of abstraction. Through an example, section 3 describes the programming model offered to the user. Section 4 is devoted to the implementation of the distributed array library and explains the addressing techniques and the communication optimizations. Section 5 concludes.

2 Structure of the programming environment

In this paper, we focus on data structure management. Other programming aspects like parallel process management are not addressed here.

| Level 6 | High level programming : provides access to any shared data structure, including protection and coherence control |
| Level 5 | Coherence and protection protocols library |
| Level 4 | Distributed array library and coherence function | Dynamic structures library (lists, trees) |
| Level 3 | Logical paging mechanism |
| Level 2 | Communication and threads system |
| Level 1 | Physical architecture |

Figure 1: Structure of the programming environment

The most external level provides a programming environment where the user can declare any data structure that may be accessed by several processes. References to this shared data structure are made through the usual notation of any programming language. As data structures may be read or written by many processes, the programmer is given different protection mechanisms and coherence control protocols.
These tools are typically included in a library. References to several works on such protocols may be found in [5].

The high level programming model is not fixed in our environment: different versions may be proposed depending on the chosen source language.

High level programming layers use the distributed data structure management system described in this paper. The distributed shared array library Cidre at the 4th level allows the creation of data structures like multi-dimensional arrays and the specification of their logical decomposition upon a distributed memory architecture. This decomposition is expressed in HPF style.

As we said, the usual index notation is used at levels 5 and 6 to access data structures. Indexes are computed globally according to array bounds regardless of the distribution. That is the key point of the programming model we propose: references to distributed arrays are the same as shared memory references. The example 2 illustrates this point.

The Cidre library provides mechanisms to transform global addresses into physical addresses and performs data communication if necessary. Moreover, a synchronization function called coherence allows groups of parallel processes sharing the same data structure to synchronize so that they have the same coherent view of the structure. This function, detailed in section 3, is the key element for the construction of enhanced protocols at the 5th level.

Other libraries may be built at level 4 to handle other data structures than arrays.

The Cidre library implementation is built upon a mechanism we call logical paging. Level 3 manages the distribution of arrays in different local memories. A distributed array is described as a set of logical pages; the size of these pages is related to the size of the array. The processes that own a part of the array in their local memories are given an array descriptor which consists mainly of a table of logical pages. It will be explained in section 4 how we manage to translate references in a very efficient way, which is a major advantage of our model.

Finally, according to the location of the requested page, access may be local or distant. For the distant accesses, we use the communication system of the target machine (level 2). Though it is not compulsory, the use of threads makes the implementation of the distributed arrays library easier, allowing overlap between different activities. At last, level 1 represents the physical layer.

Many shared object libraries have been conceived lately [1, 3, 11, 4]. Situated at different levels of user-interface, they provide various coherence protocols. Compared to these libraries, the Cidre library offers optimizations for data structure implementation and communication management.

3 The programming model

In this section we describe the library interface and the way it may be used for programming parallel applications.

The appropriate programming model is based on user-defined parallel processes. As it is often the case in the context of highly parallel architectures, processes may execute the same basic code according to the SPMD (Single Program Multiple Data) model. Depending on its identity or on the data it owns, each process will specialize in executing specific parts of the program.
The Cidre library allows user processes to share variables which will be implicitly distributed according to a user-defined partitioning scheme.

The example in figure 2 illustrates the use of the library. \( P \) processes are involved in the execution of the given code which corresponds to a Jacobi algorithm. Shared arrays are declared by calling the Cidre function `create`. This function takes as parameters the dimensions of the array, followed by the definition of how it is partitioned. Cyclic or block partitioning schemes on the \( P \) processors of the architecture are possible. In the example, \( N \times N \) matrices \( A \) and \( B \) are defined and partitioned into blocks of size \( N/P \times N \).

Our programming model is specifically defined to allow global references to shared variables, avoiding the need for explicit data transfers and global to local addresses computations. The assignment

\[
\text{write}(B,i,j,f(\text{read}(A,i+1,j), \text{read}(A,i-1,j), \\
\text{read}(A,i,j+1), \text{read}(A,i,j-1)))
\]

where \( f \) is a predefined function, illustrates these facilities.

In the example, process \( P_{i} \) owns the blocks of lines \( (N/P \times i) \) to \( (N/P \times (i+1) - 1) \). To execute its computation at step \( k \), it needs the line \( (N/P \times i - 1) \), after it has been computed by process \( P_{i-1} \) at step \( k - 1 \), and the line \( (N/P \times (i+1)) \) computed by \( P_{i+1} \) at step \( k \). Reciprocally, \( P_{i-1} \) needs the line \( (N/P \times i) \) and \( P_{i+1} \) needs the line \( (N/P \times (i+1) - 1) \).

To ensure that each process uses up-to-date values at step \( k \) (i.e. values computed at step \( k - 1 \)), we introduce two synchronization-and-coherence operations named coherence. The first one performs a cooperation between \( P_{i} \) and \( P_{i-1} \) for updating their lines \( (N/P \times i - 1) \) \( (N/P \times i) \) and for providing to both processors a coherent view of these lines; the second one makes \( P_{i} \) et \( P_{i+1} \) cooperate for obtaining an up-to-date and coherent view of lines \( (N/P \times (i+1) - 1) \) and \( (N/P \times (i+1)) \).

Generally speaking, the parameters of the coherence function describe the set of shared array elements (an array section defined by a lower bound, an upper bound and a step in each dimension) that has to be made equally visible to a group of processes. This group is referenced by the last parameter. The semantics of the coherence function applied to an array section \( AS \) and a group of processes \( G \) is the following:

- Synchronization of all the processes in \( G \) in order to take into account every write operation performed on \( AS \) since the last coherence call (or the beginning of the execution);
- Broadcasting of an up-to-date version of \( AS \) to all processes in \( G \).

Of course, if several writes to the same array element have been performed by different processes before the coherence call, the content of the up-to-date version of this element is non deterministic.

4 Logical paging mechanism

The Cidre library provides access to shared distributed arrays. The involved mechanisms have been used in the HPF Pandore compiler [2]. They exploit logical paging of arrays according to the user-specified rectangular block distribution. The goal is
process myself

\[ A = \text{create('A', } N, N, N/P, N) \]
\[ B = \text{create('B', } N, N, N/P, N) \]
\[ \text{prev.set} = \{ \text{myself, myself-1} \} \]
\[ \text{next.set} = \{ \text{myself, myself+1} \} \]
\[ \text{my.first.line} = N/P*\text{myself} \]
\[ \text{my.last.line} = N/P*(\text{myself+1}) - 1 \]

for k=1 to nloop
  if (myself ≠ 0)
    coherence(A, my.first.line-1, my.first.line, 1, 0, N-1, 1, prev.set)
  if (myself ≠ P - 1)
    coherence(A, my.last.line, my.last.line+1, 0, N-1, 1, next.set)
  for i = my.first.line to my.last.line
    for j = 1 to N - 2
      write(B, i, j, f(read(A, i + 1, j), read(A, i - 1, j), read(A, i, j + 1), read(A, i, j - 1)))
  for i = my.first.line to my.last.line
    for j = 1 to N - 2
      write(A, i, j, read(B, i, j))

Figure 2: Programming example: Jacobi algorithm

to have a quick elementary access while keeping the memory cost at a reasonable level [10].

The multi-dimensional address space defined for each array is linearized and split into pages. These pages are used to store temporary copies of distant data as well as local data.

Elements are uniformly accessed: global indices are translated into a page number and an offset in the page. The couple \((PG, OF)\) and a table of pages available on each processor are then used to access the corresponding memory element.

The size of the pages and the direction of the pages — i.e. a linearization function — are defined for each array according to the array distribution parameters. The direction of the pages corresponds to the dimension of the largest block extent. The page size is chosen to be a power of two to speed up accesses: computation of the couple \((PG, OF)\) only needs simple logical operations (shifts and masks).

Figure 3 illustrates the two different cases that may occur:

- If there is a non-distributed dimension, the page size is equal to the first power of two greater than the size of the array in that dimension. Computation of \((PG, OF)\) is then very efficient (identity in the 2D-case).

- If all the dimensions are distributed, the page size is the first power of two lower than the largest block extent. A page may then overlap a block border. In this case, each of the involved processors is responsible for a part of the page.
Figure 3: Logical paging of arrays

In addition to efficient elementary accesses, logical paging permits the optimization of the communications involved in the synchronization-and-coherence operation.

Communications are organized in segments (adjacent elements of a page). Direct communications are used to transfer big segments without any communication buffer, while small segments are aggregated in a larger buffer to minimize the effect of message latency. The limit between small and big segments can be expressed as a function of platform-specific parameters. Furthermore, multiple occurrences of elements are eliminated when preparing the transfers. A complete description of these mechanisms is available in [9].

5 Performances of logical paging

We have already compared the joint use of the logical paging system discussed above and message passing with shared virtual memory in the framework of the HPF Pandoré compiler [9].

Indeed, two versions of the compiler have been written. With the first one, the generated code makes use of the logical paging system and of a portable message passing library, the POM library [6], that allows executions on several parallel architectures and systems (Intel iPSC/2, Intel Paragon, BSD Sockets, PVM...). The second version of the compiler generates code for the SVM Koan [7] build on top of the NX/2 system on the Intel iPSC/2.
Several experiments have been made on the iPSC/2 in order to compare these two approaches. Figure 4 shows the speedups obtained for three numerical kernels: the Jacobi iterative relaxation, the LU factorization and the matrix-matrix product. These results give a good idea of how the CIDRE library would compare to a SVM because, for these regular examples, the code produced by the Pandore compiler is very similar to a hand-coded version of the parallel SPMD code.

Speed of local accesses turns out to be a critical parameter for the overall efficiency. In this respect, logical paging is very close to SVM—that can be considered optimal—as illustrated in the Jacobi example.

As logical paging is associated with message passing, complex communication patterns can be handled more efficiently than with SVM. This is the case with the LU factorization and the matrix-matrix product where broadcasting of lines and above all communications of parts of lines are necessary. To solve this, broadcasting is employed in both systems. But in the logical paging version, segment broadcasting is carried out so that the number of messages and the amount of transferred elements is kept at a minimum, whereas in the SVM version, a producer-consumers communication pattern for which entire pages are broadcasted is used. As a consequence, a much more important falling off can be observed for the SVM version when the number of processors increases, especially for small array sizes. Moreover, it is clear that the difference would be greater without this broadcasting protocol that is to say when only point to point page transfers would be authorized through page faults solving.

Besides, we believe that the superiority of the logical paging (combined with message passing) that is used in the CIDRE library over SVM is likely to be greater in the context of networks of workstations where the message latency is very high.

6 Conclusion

We are currently implementing the CIDRE library for shared distributed arrays. The interface language may be subject to some slight modifications, for example concerning the syntax of primitives such as coherence.

Moreover, we must work out some implementation mechanisms. For instance, we are experimenting different solutions to efficiently perform the test that must determine if references correspond to data already present or not in the local memory. A prefetch operation appears to provide a good way to avoid numerous executions of this test.

At last, the writing of high level coherence protocols is envisaged in order to have enlightenments on the adequacy of CIDRE to parallel application programming and on the performances that may be obtained at the user level.

References


Figure 4: Comparison between logical paging + message passing and SVM
Analysis of Access Patterns
Analyzing Data Structures for Parallel Sparse Direct Solvers: Pivoting and Fill-In *

J. Touriño  R. Doallo
Dept. Electronics and Systems
University of La Coruña, Spain
e-mail: {juan,doallo}@udc.es

R. Asenjo  O. Plata  E.L. Zapata
Dept. Computer Architecture
University of Málaga, Spain
e-mail: {asenjo,oscar,ezapata}@atc.ctima.uma.es

Abstract

This paper addresses the problem of the parallelization of sparse direct methods for the solution of linear systems in distributed memory multiprocessors. Sparse direct solvers include pivoting operations and suffer from fill-in, problems that turn the efficient parallelization into a challenging task. We present some data structures to store the sparse matrices that permit to deal in a efficient way with both problems. These data structures have been evaluated on a Cray T3D, implementing, in particular, LU and QR factorizations as examples of direct solvers. Any of the data representations considered enforces the handling of indirections for data accesses, pointer referencing and dynamic data creation. All of these elements go beyond current data-parallel compilation technology. Our solution is to propose new extensions to HPF that permit to deal with these codes, and to support part of the new capabilities on a runtime library at the compiler level.

1 Introduction

The solution of systems of linear equations, \( Ax = b \), where \( A \) is a large sparse matrix, plays a basic role in many fields of science and engineering. There are two different approaches to solve such systems, direct and iterative methods. In direct methods \([10][15][18]\), the system is converted into an equivalent one whose solution is easier to determine by applying a number of elementary row and/or column operations to the matrix \( A \). A different approach is taken in iterative methods \([4][28][18]\), where the number of operations required is not known in advance.

In this paper we will focus on two of the most important direct methods, LU and QR factorizations \([15]\). LU factorization is used for the conversion of a general system of linear equations to triangular form via Gauss transformations. The QR decomposition has various other applications in linear algebra, such as solving least squares problems, eigenvalue

*The work described in this paper was supported by the Ministry of Education and Science (CICYT) of Spain under project TIC96-1125-C03-01 and by the Human Capital and Mobility programme of the European Union under project ERB40501921660 and by the Training and Research on Advanced Computing Systems (TRACS) at the Edinburgh Parallel Computing Centre (EPCC)
problems, coordinate transformations and projections problems. All this kind of computations appears in many scientific areas, such as fluid dynamics, structural analysis, circuit simulation, device simulation and quantum chemistry, among many others.

Over the last decades, there have been major research efforts in developing efficient parallel numerical codes, emerging the data-parallel paradigm as one of the most successful programming models to reach the above objective. As a result, during the last few years, a number of high-level data-parallel languages have been designed, such as Vienna-Fortran [29], Fortran D [14], High-Performance Fortran (HPF) [16] and Craft [20].

All these languages had initially focused in regular computations, that is, well-structured codes that can be efficiently parallelized at compile-time using simple data distributions. However, the current constructs included in these languages lead to a low efficiency when they are applied to irregular codes, such as sparse computations, appearing in the majority of real scientific and engineering applications. In order to help solving this problem we have developed and extensively tested a number of pseudo-regular data distributions, designed as natural extensions of the regular data distributions [1] [2] [3] [7] [22] [25] [26] [27]. The aim of these distributions is their simplicity to be incorporated to a data-parallel language and be used by a programmer, together with their effectiveness to obtain high efficiencies from the parallelization of irregular codes. Other important aspect that may influence the parallel efficiency is how the partitioned data is stored in the local memories of the processors. In this paper we will discuss all these related issues in the scene of sparse direct methods, specifically LU and QR decompositions. Special attention will be given to the efficient solution of the fill-in problem and the pivoting operation.

The rest of the paper is organized as follows. In Section 2 we discuss the sparse direct methods, in particular the LU and QR factorizations. In Section 3 we describe and discuss the data structures we have designed to implement efficiently the factorization codes. The pseudo-regular data distributions we propose, specifically for the efficient parallelization of sparse direct methods are introduced in Section 4. The parallelization strategy of such direct methods, LU and QR factorizations, are presented in Section 5, together with some experimental results comparing different data structures and distributions. Based on the experimental results obtained, we propose in Section 6 new extensions to the HPF data-parallel language for solving efficiently the main issues which appear during the computation of sparse direct methods. Finally, Section 7 presents concluding remarks.

2 Sparse Direct Methods

2.1 QR Factorization

The QR factorization of a m-by-n matrix A is given by $A = QR$, where $Q$ is m-by-n orthogonal matrix (that is $Q^T = Q^{-1}$) and $R$ is a n-by-n upper triangular matrix. The QR computation can be arranged in several ways, such as methods based on Householder reflections and Givens rotations. The Gram-Schmidt orthogonalization process and, particularly, the more numerical stable variant called Modified Gram-Schmidt (MGS) is the method considered here.

The MGS algorithm is a rearrangement of the Classical Gram-Schmidt algorithm with better numerical properties. Figure 1 shows an in-place algorithm that includes column pivoting in order to deal with rank deficiency problems ($\text{rank}(A) < n$) and to provide numerical stability. Basically, the MGS algorithm is an iterative procedure of up to $n$ iterations. In each iteration $k$ a pivot column is identified and swapped for the current column $k$ in both
\begin{verbatim}
rank = n;
do j = 1, n
    norm(j) = \sum_{i=1}^{m} A(i, j) + A(i, j)
endo
do k = 1, n
    Find p with k \leq p \leq n so norm(p) = \max_{k \leq j \leq n} norm(j)
    if (norm(p) < \phi)
        rank = k - 1; break
    else
        swap A(1: m, k) and A(1: m, p)
        swap R(1: n, k) and R(1: n, p)
        swap norm(k) and norm(p)
    endif
    R(k, k) = \sqrt{norm(k)}
    A(1: m, k) = A(1: m, k)/R(k, k)
    do j = k + 1, n
        R(k, j) = \sum_{i=1}^{m} A(i, k)A(i, j)
        norm(j) = norm(j) - R(k, j)R(k, j)
        A(1: m, j) = A(1: m, j) - A(1: m, k)R(k, j)
    enddo
endo
\end{verbatim}

Figure 1: Modified Gram-Schmidt (MGS) algorithm

matrices $Q$ (that is $A$) and $R$. Afterwards all the $j$ columns, $k \leq j \leq n$, of $Q$ are updated and the $k$ row of $R$ is computed. Once the algorithm has finished, what we really obtain is a $A II = QR$ factorization due to the pivoting operation carried out.

Considering that $A$ is a sparse matrix, we have taken special actions during the pivoting operation in order to reduce the fill-in problem in the MGS algorithm and to ensure numerical stability. A simple strategy is based on the selection of a column with the maximum norm (the one considered in Figure 1), but we have also experimented with a more elaborated pivoting criterion, where columns with few nonzero elements are the only eligible columns to be the pivot [7]. This way the fill-in in $R$ and $Q$ is reduced.

Figure 2 presents, in a graphical way, the data accesses (dependencies) and flows for the three main operations in the MGS code: pivoting (column swapping), columns ($Q$) and row ($R$) updating and fill-in. An efficient parallel implementation of the first two operations requires fast accesses to data by columns. This fact strongly determines the data structures we should use for storing the sparse matrices, as well as the method chosen to distribute these matrices among the local memories of the multiprocessor. Moreover we should consider some dynamic mechanism at the data structure level in order to deal with the fill-in efficiently. All these issues will be discussed in the next Section.

2.2 LU Factorization

The LU factorization of a $n$-by-$n$ matrix $A$ produces a couple of $n$-by-$n$ matrices, $L$ (lower triangular) and $U$ (upper triangular), and the $n$-by-$n$ permutation matrices $\Omega$ and $\Pi$, such that $\Omega\Pi A = LU$. There are different strategies to deal with the LU factorization of generic
sparse matrices [8]. Some of the options are based on supernodal or multifrontal approaches, in which arithmetic is performed on dense submatrices [9] (level 2 or 3 BLAS can be used). A third approach, the one considered in this paper, may consist in selecting some generic methods, such as left-looking or right-looking LU.

The left-looking strategy, also known as column-based (row-based) method and implemented for instance in the MA48 routine [11], updates in the $k$-th iteration the $k$-th column (of $L$ and $U$) starting from the $k - 1$ previously updated columns. This is accomplished by two different steps for each iteration, a symbolic factorization and a numerical factorization. In the first step, the nonzero structure of the current column is predicted. The second step performs the arithmetic operations strictly needed, using the information gathered by the symbolic factorization. The right-looking LU, also called submatrix-based method, performs a total of $n$ iterations. In the $k$-th iteration a pivot is chosen, a column and a row permutations may be performed so that the pivot occupies the $(k, k)$ position, and, finally, the submatrix defined by the pivot is updated (that is, elements $(k + 1 : n, k : n)$).

Any of the above methods can complete the resolution of the linear system of equations following four stages: reordering, analyze, factorize and solve. The first step realigns the matrix with the aim of reducing the complexity of the subsequent processing stages. The analyze stage chooses the appropriate row and column permutations ($\Omega$ and $\Pi$), required for the selection of the pivot elements. These pivots must be chosen such that the sparsity rate is preserved (applying, for example, the Markowitz criterion [19]), and the numerical stability is guaranteed (choosing those pivots greater than a certain threshold value). Afterwards, in the factorize stage, the factorization $\Omega \Pi = LU$ is performed. This may be the most consuming-time stage, since it performs the update operations for floating-point numbers. Finally, in the solve stage, the above factorization is used to solve the equation $Ax = b$ or $A^T x = b$.

From a parallel implementation point of view, the left-looking organization does not let us to exploit as much parallelism as in the right-looking strategy [2], as in the first one a column is
Initialize $\Omega$ and $\Pi$ to the $n$-by-$n$ identity matrix
Initialize $R$ and $C$

\[
\begin{align*}
do & \quad k = 1, n - 1 \\
& \quad \text{Find pivot } p = (\mu, \lambda) \text{ (Markowitz criterion)} \\
& \quad \text{Swap } A(k, 1 : n) \text{ and } A(\mu, 1 : n) \\
& \quad \text{Swap } A(1 : n, k) \text{ and } A(1 : n, \lambda) \\
& \quad \text{Swap } R(k) \text{ and } R(\mu) \\
& \quad \text{Swap } C(k) \text{ and } C(\lambda) \\
& \quad \text{Swap } \Omega(k, 1 : n) \text{ and } \Omega(\mu, 1 : n) \\
& \quad \text{Swap } \Pi(1 : n, k) \text{ and } \Pi(1 : n, \lambda) \\
& \quad A(k + 1 : n, k) = A(k + 1 : n, k)/A(k, k) \\
\text{do } & \quad j = k + 1, n \\
& \quad A(k + 1 : n, j) = A(k + 1 : n, j) - A(k + 1 : n, k)/A(k, j) \\
\text{enddo}
\end{align*}
\]

\[\text{Update } R \text{ and } C \text{ nonzero counts}\]

Figure 3: Outline for the right-looking LU factorization algorithm, where the analyze and factorize stages appear joined together.

updated in each iteration, while in the second one a complete submatrix is updated. Moreover, an additional overhead arises from the parallelization of the depth-first search in the symbolic factorization.

The submatrix-based (right-looking) approach presents two sources of parallelism we can exploit, as can be derived from the the pseudo-code of Figure 3. The algorithm performs a number of iterations, each of them involving a pivot search in the reduced matrix (defined as the $(n - k)$-by-$(n - k)$ submatrix of $A$, such that $k \leq i, j \leq n$, in the $k$-th iteration), followed by a row and a column swapping, and an update of range one in the same reduced matrix. In this code the Markowitz's heuristic was chosen for finding the pivot element, which is based on the minimization of the $M_{ij} = (R_i - 1)(C_j - 1)$ parameter, where $R_i$ ($C_j$) counts for the number of nonzero elements in the $i$-th row ($j$-th column).

The first source of parallelism in the above code corresponds to the loops in charge of updating of the reduced matrix. The second source of parallelism comes from the sparse nature of $A$. In many cases it is possible to merge several updates of range one in only one update process of multiple range (say, $s$), by modifying the Markowitz strategy in such a way that we search for a pivot set containing $s$ compatible pivots, instead of only one pivot element. A couple of pivots, $A_{ij}$ and $A_{kj}$, are said compatibles (and independent) if $A_{kl} = A_{kj} = 0$.

The data flows and accesses for the right-looking LU code is shown in Figure 4, for the pivoting (row/column swapping) and sub-matrix updating operations, as well as fill-in. Observe that for an efficient parallel LU algorithm fast accesses to data by both rows and columns are required. Therefore we need to use more complex data structures than for the MGS algorithm.

3 Data Structures for Sparse Direct Methods

Direct methods, specifically LU and QR factorizations, decompose the original sparse matrix $A$ by using simple row and/or column operations. When implementing these methods on a distributed-memory multiprocessor we should distribute the data across the local memories in such a way that workload balance and limited communication overhead is assured. Such
data distributions are discussed in the next Section.

Once the local sparse matrices are obtained, we can select a number of data structures to store them. Typically, in order to save memory (and computations), zero elements of the sparse matrices are not stored. There are many methods for storing the nonzero elements of the matrices [4]. Here we will only discuss the Compressed Row and Column Storages (CRS and CCS). The CRS format represents a sparse matrix $A$ as a set of three vectors (DATA, COL and ROW). DATA stores the nonzero values of $A$, as they are traversed in a row-wise fashion, COL stores the column indices of the elements in DATA, and ROW stores the locations in DATA that start a row. By convention, we store in the position $n+1$ of ROW ($n$ is the number of rows of $A$) the number of nonzero elements of $A$ plus one. The CCS format is identical to the CRS format except that the columns of $A$ are traversed instead of the rows.

We can simply take some sort of packed vector format [10] (such as CRS or CCS), or use some other more complex and flexible data structure for storing the local sparse matrices. We have experimented with linked lists, pure CRS and CCS compressed formats and some mixed structure, depending on the type of data accesses we have to deal with.

In a MGS factorization of a sparse matrix only efficient accesses by matrix columns are needed (see Figure 2). This fact implies large memory and computation savings because we can use simple packed vectors (CCS, for instance) or one-dimensional doubly linked lists to store the local sparse matrices. As we can see in Figure 5 (b) each linked list represents one column of the local sparse matrix where its nonzero elements are arranged in growing order of the row index. Each item of the list stores the row index, the matrix element and two pointers. A simplification of the linked list is showed in Figure 5 (a), where the columns are stored as packed vectors and they are referenced by means of an array of pointers. The packed vectors do not have pointers inside and, therefore, this mixed structure requires only almost half as much memory space as the doubly linked list (considering the fact that in the C compiler of the Cray T3D, for example, the int as well as the double types take up 8 bytes both). Note that the data structure shown in Figure 5 (a) is a variant of the CCS compressed format, where the DATA and ROW vectors are joined together and the COL vector is represented as an array of pointers, with the size of the corresponding column associated with each one.

In a LU decomposition, on the other hand, we require a data structure such as a two-dimensional doubly linked list (see Figure 5 (c)) in order to make efficient data accesses both by rows and by columns (see Figure 4). Each item in such a dynamic structure stores not only the value and the local row and column indices, but also pointers to the previous and next nonzero element in its row and column (four pointers in total).
From the point of view of an efficient computation, packed vectors are very compact and allow fast accesses by rows and columns to the matrix elements (but not both at the same time). Linked lists are specially useful when more flexible accesses to the matrix elements (by rows and columns simultaneously) are needed. Additionally, there are two critical issues to be taken into account when factorizing a sparse matrix using a direct method, pivoting operations and fill-in.

Doubly linked lists make easy the insertion and deletion operations and, hence, we can deal with the fill-in and pivoting problems in an efficient way. In the case of using packed vectors, the fill-in problem is more difficult to solve. For a MGS code, for instance, we have followed this procedure: an auxiliary buffer (which is also a packed vector) long enough to store one column is allocated. During the updating process of a column, each nonzero entry, a previously existing one or a new one (fill-in), is stored in the auxiliary buffer, instead of in the original packed vector. This way, the new elements are just added to the buffer, but the zeroing of existing entries are discarded. After finishing the column updating process,
Table 1: Properties for the different data structures

The buffer contains the new packed column. Hence, this auxiliary buffer is just reallocated to a memory block of its exact size, becoming the new column of the sparse matrix, while the memory space of the old column is freed. Packed vectors, such as CRS and CCS, have also the inconvenience of not allowing the pivoting operation (column/row swapping) in an efficient way. This is the reason of using some mixed data structure, such as the one presented in Figure 5 (a). Column pivoting is implemented just interchanging pointer values.

But linked lists have also severe drawbacks. The dynamic memory allocation for each new element is time-consuming, and the list traversing even more, as well as they consume more space memory than packed vectors. But one major problem is the memory fragmentation due to allocation/deallocation of items, and spatial data locality loss (cache inefficiency) during traversing rows and columns due pivoting operations (pivoting does not move data, only changes pointer references). For these reasons we have made an effort to develop an efficient parallel MGS and right-looking LU factorization algorithms avoiding linked lists.

Table 1 summarizes the discussed properties of the described data structures from the point of view of their behaviour when using in parallel sparse direct methods codes. Linked lists entry in this table corresponds to the structures shown in Figure 5 (b) and (c), whereas packed vectors (w/ pointers) corresponds to that of Figure 5 (a). The last entry is the standard CCS and CRS compressed formats. All these structures were implemented for the parallel right-looking LU and QR factorization, this last one using the MGS algorithm, Householder reflections and Givens rotations (the parallel implementation of the last two methods are described in [13] and [24]).

In any case, the use of data structures such as packed vectors implies the appearing of code segments like

\[
\begin{align*}
\text{DO } i = a, b \\
\text{vu(row(i))} = \text{vp(i)} \quad \text{and} \quad \text{init = c} \\
\text{DO } j = a, b \\
\text{vu(row(j))} = \text{vp(j)} \quad \text{IF (vu(i) .NE. 0.0) THEN} \\
\text{vp(init)} = \text{vu(i)} \quad \text{row(init)} = 1 \\
\text{init = init + 1} \\
\text{ENDIF} \\
\text{ENDDO}
\end{align*}
\]

The first piece of code is used to convert a packed vector to an unpacked one, and has an assignment statement with indirections at the left-hand side. The second construct is the reverse operation, used to pack the elements of a sparse vector. In this case we have a loop containing an induction variable. In general, the current data-parallel compilers (such as the T3D-Craft compiler) fail when compiling these kind of constructs. In the first case because the contents of row() is unknown until runtime and the compiler simply assume dependencies across iterations. In the second piece of code, the increment of the induction variable is included into a conditional statement.
Figure 6: (a) A sparse matrix. (b) The BRS partitioning of the sparse matrix for a 2x2 processor mesh, where the data elements for processor 0 are underlined. (c) The compressed local submatrix for processor 0

4 Pseudo-Regular Sparse Data Distributions

Current data-parallel languages, such as HPF, Vienna Fortran or Craft, include the most useful and simple schemes for distributing data across the processors, specifically block, cyclic and a combination of both. All these distributions allow us to parallelize in a efficient way most Fortran codes with regular accesses to data. However, this is not true for applications with irregular patterns for accessing data. These programs contain array indirections that produce not well-balanced parallel codes and/or with complex communication patterns when using regular data distributions.

Consider, for instance, applications that process data organized as sparse matrices. The use of compressed formats for storing sparse matrices implies the appearing of array indirections in the code. Our approach to deal with this kind of data accesses is to define pseudo-regular data distributions as extensions of the classical block and cyclic regular distributions, such as MRD (Multiple Recursive Decomposition) and BRS (Block Row Scatter) or BCS (Block Column Scatter) [22][1]. Let us concentrate on the last two data distributions, as they are extensions of the regular cyclic distribution, one of the most successful data distribution for matrix computations.

In our current situation, a sparse matrix is represented by a set of vectors (arrays), depending on the compressed format considered (CRS or CCS, for example). Instead of decomposing these arrays separately, as in commonly done, we follow the approach of considering the sparse matrix as a dense one, mapping this dense matrix on the processors using some standard data distribution and, finally, representing the local sparse matrices using the adopted compressed format. In this way, BRS (BCS) uses a cyclic mapping of the matrix represented by its CRS (CCS) format, as shown in figure 6.

An HPF-like description of the BRS data distribution may be as follows,

```
!HPF$ SPARSE(CRS(DATA,COL,ROW)) :: A(N,N)
!HPF$ DISTRIBUTE(CYCLIC,CYCLIC) ONTO MESH :: A
```

The SPARSE directive means that the sparse matrix A is actually represented in a CRS format, using the arrays DATA, COL and ROW. The BRS distribution is a cyclic distribution of the compressed representation of a sparse matrix. Stating CYCLIC in a DISTRIBUTE directive
<table>
<thead>
<tr>
<th>Distribution</th>
<th>SHL400</th>
<th>JPWH991</th>
<th>SHERMAN1</th>
<th>MAHINDAS</th>
<th>ORANI678</th>
<th>SHERMAN5</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCS</td>
<td>6</td>
<td>59</td>
<td>27</td>
<td>21</td>
<td>134</td>
<td>191</td>
</tr>
<tr>
<td>CHAOS</td>
<td>52</td>
<td>615</td>
<td>243</td>
<td>179</td>
<td>1228</td>
<td>1685</td>
</tr>
</tbody>
</table>

Table 2: Execution times (in sec.) for MGS using BCS and CHAOS

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Origin</th>
<th>$m \times n$</th>
<th>#elem($A$)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHL400</td>
<td>Linear programming problems</td>
<td>653x663</td>
<td>1712</td>
<td>0.39%</td>
</tr>
<tr>
<td>JPWH991</td>
<td>Circuit physics modeling</td>
<td>591x991</td>
<td>6027</td>
<td>0.61%</td>
</tr>
<tr>
<td>MAHINDAS</td>
<td>Economic modeling</td>
<td>1258x1258</td>
<td>7582</td>
<td>0.49%</td>
</tr>
<tr>
<td>ORANI678</td>
<td>Economic modeling</td>
<td>2529x2529</td>
<td>90158</td>
<td>1.41%</td>
</tr>
<tr>
<td>WELL1850</td>
<td>Least squares problems in surveying</td>
<td>1850x712</td>
<td>8758</td>
<td>0.66%</td>
</tr>
<tr>
<td>LNS3937</td>
<td>Compressible fluid flow</td>
<td>3937x3937</td>
<td>25407</td>
<td>0.16%</td>
</tr>
<tr>
<td>ORSREG1</td>
<td>Oil reservoir simulation</td>
<td>2205x2205</td>
<td>14133</td>
<td>0.29%</td>
</tr>
<tr>
<td>STEAM2</td>
<td>Oil reservoir simulation</td>
<td>600x600</td>
<td>13760</td>
<td>3.82%</td>
</tr>
<tr>
<td>SHERMAN1</td>
<td>Oil reservoir modeling</td>
<td>1000x1000</td>
<td>3750</td>
<td>0.37%</td>
</tr>
<tr>
<td>SHERMAN2</td>
<td>Oil reservoir modeling</td>
<td>1060x1060</td>
<td>23094</td>
<td>1.98%</td>
</tr>
<tr>
<td>SHERMAN5</td>
<td>Oil reservoir modeling</td>
<td>3312x3312</td>
<td>20993</td>
<td>0.19%</td>
</tr>
</tbody>
</table>

Table 3: Harwell-Boeing test matrices

is understood by the compiler as applying a BRS distribution to the DATA, COL and ROW arrays declared in the SPARSE statement. This way we have the benefits of a cyclic data distribution (load balancing and simple and limited communication patterns) applied to a sparse matrix independently of the compressed format used to represent it.

As a test to compare our parallel solutions with others using the standard dense data distributions, we made an experiment implementing the sparse algorithms using such distributions and the CHAOS runtime library [21], in order to deal with the irregular data accesses. We have inserted routines from the CHAOS library [23] to rebalance the load (and data) of the parallel MGS algorithm. Basically, through these routines, we have generated a translation table which assigns the global indices of matrix $A$ to the different processors by following an irregular model. This table is distributed across the processors and is used by the routine localize to translate the global indices into local indices within each processor. It also generates a communication schedule which is used to gather the off-processor data which are needed during computation, and to scatter back local copies after computation. For the parallel algorithm MGS, table 2 shows a comparison of the execution times using BCS pseudoregular data distribution as opposed to the irregular distribution used in the CHAOS routines. The execution times have been taken in a cluster of 16 workstations Sun SPARCstation 4 with 85-MHz microSPARC-II processors in a PVM message-passing environment. The test sparse matrices were taken from the Harwell-Boeing suite and are described in table 3.

The CHAOS approach has a large number of communications and high memory overhead, as a consequence of accessing a large distributed data addressing table. This results in high execution times. BCS (and BRS) distribution, on the other hand, is adequate for sparse matrix problems (in particular, the MGS algorithm) because it exploits the data and computations locality and minimizes the communications. It does not require neither additional storage nor communications for addressing nonlocal data, as all the processors know where data are allocated.
5 Evaluating Data Structures and Distributions

5.1 Parallel Sparse MGS

We have designed three different parallel implementations of the MGS algorithm of the Figure 1. The parallelization procedure we have used is extensively explained in [7]. Here we will focus in the performance evaluation of the parallel MGS algorithm in terms of the discussed data structures and distributions. All the experimental results were obtained using a Cray T3D multiprocessor [5].

A first parallel implementation is a simplification of the considered MGS algorithm. The pivoting process is discarded and the upper triangular $R$ matrix is the only one calculated (and the only needed in many codes that incorporate this kind of factorization). In this parallel version all the local sparse matrices are stored using packed vectors. Specifically, the initial $A$ matrix is stored by packed columns (CCS compressed format) in a buffer. As $R$ is calculated row by row, each one is stored as a packed vector (CRS format) in other buffer. As we do not intend to preserve $Q$, the buffer associated to $A$ (and $Q$ in the in-place algorithm) operates as a ring buffer, saving this way local memory in the processors.

Figure 7 shows an example for a $A$ matrix with four columns. Each filled block represents a column for $A$ (and $Q$) and a row for $R$. As we can see, in each $k$ iteration, the last $n-(k+1)$ columns of $Q$ are updated, and the $k$ row of $R$ is calculated. Whenever the end of the ring buffer storing $Q$ is reached, the new columns are stored from the beginning of such buffer. Note that we have to dimension the ring buffer to a size long enough to preserve, during the $k$ iteration, the $k$ column of $Q$ until the updating of the last column of $Q$ (see Figure 1).

This parallel algorithm was coded in Fortran 77 and the Cray T3D SHMEM [5] native shared-memory library was used for communication. With these routines we can minimize the communication overhead at the expense of a very careful programming due to possible synchronization and cache coherence problems. It only has three communication operations per iteration (say, $k$), a reduction for obtaining the norm of the $k$ column, a broadcast of the normalized $k$ column to all processor columns of the mesh, and a reduction for calculating the dot products (see Figure 1). The rest of computations are completely local. This reduced number of communications comes from the absence of the pivoting operation and the resolution of the least squares problem. Figure 8 presents the parallel execution times for several processor meshes and Harwell-Boeing sparse matrices. We can see that, in general, the parallel efficiency is near optimal (in some cases we have superlinearity). A very low
communication overhead (small number of communications and very efficient due to the use of SHMEM routines) and very high spatial locality exploitation (packed vectors allow us to avoid the use of pointers and dynamic memory allocations) justify this behaviour.

Two more parallel implementations of the complete MGS algorithm (including pivoting) were designed using the one-dimensional doubly linked list and the packed vectors shown in Figure 5 (b) and (a), respectively, for storing the local sparse matrices. Both versions were coded in C and using PVM routines for message-passing. In order to reduce the communication overhead the low-latency communication functions pvm_fastsend and pvm_fastrcv (non-standard PVM routines) were used for messages of length less than 256 bytes. Efficient custom reduction operations suitable for the algorithm were also developed. Figure 9 shows the parallel execution times obtained for different processor mesh sizes and Harwell-Boeing sparse matrices (see Table 3) for both implementations. The execution times include the QR
factorization as well as the solving of the least squares problem. The time required for data distribution and collection is not included because we assume that these algorithms are sub-problems within wider programs. As we can see, the execution times are in correspondence with the size of the input matrices. In the other hand, the dependence of the times with the size of the machine is almost linear, that is, the speedup (or efficiency) is very high. Comparing both figures it can be seen that the packed vectors based implementation is faster than the one using linked lists.

5.2 Parallel Sparse LU

Two parallel implementations of the right-looking LU algorithm, based on doubly linked lists and packed vectors, are described. Both parallel algorithms were designed using the same environment as in the parallel MGS.

The first parallel algorithm is mapped on a mesh of \( P_1 \times P_2 \) processors using a BCS (sparse cyclic) data distribution and a local representation with two-dimensional doubly linked lists. Note that the use of linked lists is almost unavoidable in order to handle in an efficient way the pivoting operation (that is, row and column swapping). This algorithm is extensively described and evaluated in [3]. Figure 10 (a) reproduces the parallel executions times obtained on the Cray T3D, for different mesh sizes and Harwell-Boeing sparse matrices (see Table 3 for a description). The parallel algorithm was coded in C and PVM routines were used for message-passing (the Cray T3D specific low latency PVM functions were used as in the MGS). There are some improvements to the linked list implementation developed by Jacko Koster [17], such as the reduction of the communications cost by an implicit pivoting together with sporadic workload re-balancing phases.

On the other hand, in the parallel version based on packed vectors, the \( A \) matrix is distributed following the BCS scheme and the sparse local matrices are stored in the implicit CCS format. While in the left-looking LU the fill-in only appears in the \( k \) column at each iteration, in the right-looking LU the fill-in affects to the whole reduced submatrix. This
fill-in implies much more data movement than in the left-looking case, and also it could be associated with a garbage collection operation. We are also forced to minimize the number of row and column permutations during the factorization stage. To accomplish that the analyze and factorize steps are carried out separately. For the analyze stage the MA50AD routine [12] (included in the MA48 software package) is used, but it has not been parallelized, it is just executed in only one processor before the factorize stage.

As the sparsity rate of the matrix decreases during the factorization, a switch to a dense LU factorization is advantageous at some point. The iteration beyond which a switch to dense code takes place is decided in the analyze stage. Hence, a sparse factorization code is executed initially, but when reaching the switch point a dense code continues the factorization. This dense code is based on Level 2 (or 3 if a block cyclic distribution is used) BLAS, and includes numerical partial pivoting in order to assure stability. Once the sparse computations are carried out the reduced submatrix is scattered to a dense array. Therefore, the overhead of the switch operations is negligible and the reduced dense submatrix appears distributed in a regular cyclic manner.

Figure 10 (b) presents the parallel execution times for the BCS-based right-looking LU algorithm. This time the parallel algorithm was coded in Fortran 77 and the Cray T3D SHMEM [5] native shared-memory library was used for communication. The factorization numerical errors of our parallel algorithm are similar to those of the MA48 routine, and they can be reduced by applying a previous scaling to the matrix. Besides, the fill-in of our algorithm is also similar to that of the MA48 (using the same threshold). Nevertheless, the sequential time of our new algorithm is significantly higher than in the MA48 (very optimized).

6 A Proposal for Extending HPF Capabilities

Current data-parallel languages (HPF, Fortran D, Vienna Fortran, Craft ...) do not provide support to specify efficient data distributions for sparse matrices, nor flexible data structures for storing the local sparse matrices. However the distribution strategy of a sparse matrix across the processors and the data structures used to store the corresponding local sparse matrices are crucial decisions in order to obtain high parallel efficiencies. As an example, the experimental evaluation presented in the above section shows that we can obtain high efficiencies from parallel sparse direct methods codes just selecting carefully a suitable data structure and distribution. Indeed in all our parallel sparse algorithms we have used our pseudo-regular data distributions (specifically BRS and BCS and variants), obtaining very good speedups. None of them are included in the existing data-parallel languages.

It would be of major interest if we could incorporate to a current data-parallel language, such as HPF, some sort of data structure declaration, as linked lists and packed vectors, but without loosing the convenient matrix-like computation specifications. That is, if we could use some efficient data structure for storing the sparse matrices but with no need to deal with the complexities of use it directly. Figure 11 presents an example of such a specification for the MGS algorithm. The SPARSE directive is used to specify A as a sparse matrix that is stored using a LLCS (Linked List Column Storage) data structure (the structure shown in Figure 5 (b)). That is, with this directive we establish an identification between the matrix A and its machine storage representation. From this point on, we can specify the computations using matrix notations but with the confidence that the compiler will translate these specifications

\[\footnote{Obviously we must incorporate in some way a numerical pivoting to the factorize stage, only when the chosen pivot may introduce numerical instability.}\]
Figure 11: Outline for a HPF-like specification of the parallel MGS algorithm

to computations using linked lists.

We have to take a special action to deal with the fill-in problem, because the sparse matrices are stored compressed. For instance, consider the inner loop I at the end of the code in Figure 11. By default, this loop runs only over the non-zero elements of a J column of A. But for the column updating to be correct, the I loop must run over all the elements, zero and non-zero, of A, because A(I, J) could be zero but not VCOL(I) * VSUM(J) (fill-in). We have incorporated the HPFS FILLIN IN A just before the I loop in order to declare this fact, and change the normal behaviour of that loop.

In the proposed code, the meaning of some Fortran 90 and HPF standard procedures and functions should be extended. For instance, at the beginning of the code, the intrinsic F90 procedure DOT_PRODUCT is used for computing the square of the norm of the j column of A. In
our case, that column is stored as a linked list and, therefore, this procedure should consider this storage representation. DOT_PRODUCT is also called later to compute the elements of the \( k \) row of \( R \) (first inner loop). In this case, its first argument is an array (VCDL) and the second one is a linked list (\( j \) column of \( A \)). There are also new procedures, SWAP(\( ) \) and UNPACK(\( ) \). The first one is used to swap two arrays, in our case, two columns (pivoting operation). The second one converts a packed vector to an unpacked one (with zeroes). This second procedure is a local operation (no communications) because the sparse matrices are distributed using BCS, that is, a cyclic distribution as dense (unpacked) matrices, and then the local matrices are compressed locally, at each processor.

As the final indirections for access pattern depend on the actual input data, part of the analysis must be done during program execution. In order to support all this new functionality, we are in the process of extending our run-time library DDLY (Data Distribution Layer) [25] with a set of routines to handle linked lists, to be called from the output machine code generated by a HPF compiler.

7 Conclusions

One of the major reasons why data-parallel computation has not achieved outstanding results, in terms of functionality and efficiency, has been the development of very general programming and compilation techniques without a deep orientation to real codes.

It is clear that sparse direct methods are complex computations, in such a way that the current data-parallel technology does not have the elements to solve them efficiently and in an elegant manner. Two are the main difficult issues, pivoting and fill-in. Both are difficult to handle, time-consuming and with high memory overhead, because the matrices are stored in a compressed format (that is, only nonzero elements are stored). The more flexible way to manage these problems implies the use of some sort of linked list data structure to store the sparse matrices. Nowadays, there are no data-parallel tools with an efficient handling of this kind of data structures, and the cost of development for those strategies are still under evaluation.

This paper has presented a possible solution to deal with these computations in a HPF environment. The idea is to extend the HPF capabilities in such a way that the programmer may not only specify a particular sparse data distribution but also a particular sparse data storage representation. This way we establish an identification of the array representation of a sparse matrix at the programmer level and the storage representation (packed vectors, linked lists, \( \ldots \)) at the compiler (machine) level. The programmer deals with convenient matrix notations and the compiler translates them to machine code by using packed vectors and/or linked lists handling routines (from the DDLY runtime library). The results from our manual parallel implementations of direct solvers (emulating the output of a HPF compiler) show that we can obtain high efficiencies using the above strategy.

Acknowledgements

We gratefully thank Iain Duff and all members in the parallel algorithm team at CERFACS, Toulouse (France), for their kindly help and collaboration. We also thank the Ecole Polytechnique Federale de Lausanne, Switzerland, and the Edinburgh Parallel Computing Centre, UK, for giving us access to the Cray T3D multiprocessor.
References


A Refined Method for Alignment Analysis Combining Inter- and Intradimensional Alignment Preferences

Erwin Laure¹  Barbara Chapman²

¹Institute for Software Technology and Parallel Systems
   University of Vienna - Austria
   erwin@par.univie.ac.at

²VCPC, European Centre for Parallel Computing at Vienna
   University of Vienna - Austria
   Barbara.Chapman@vcpc.univie.ac.at

Abstract

The specification of efficient data distribution schemes is one of the major tasks in programming distributed memory multiprocessors with state of the art parallel languages. Although there are no optimal strategies for generating such data distributions several heuristics have arisen to provide some support to the user. Alignment analysis, for instance, is able to provide help for finding a good distribution scheme and is furthermore a useful prerequisite for automatic data distribution tools. We present an overview of an automatic alignment analysis tool within the framework of VFCS elsewhere, which is able to generate alignment proposals for the arrays accessed in a procedure automatically and thus simplifies the data distribution problem.

Traditionally the alignment problem can be split up into two components, the inter- and intradimensional alignment problem. In this paper we focus on the models for both problems. We show how they can be modeled together in an efficient way and how are their interactions during the solution process. We present an effective heuristic for weighting interdimensional alignment preferences (i.e. a measure reflecting its overall importance for the performance) and introduce a concept of locality to model intradimensional preferences. Another issue is the formulation of efficient heuristics to solve both, interdimensional and intradimensional conflicts.

1 Introduction

The development of high level data parallel languages has significantly alleviated the task of programming state of the art distributed memory multi processor machines (DMMP) under the single program multiple data (SPMD) paradigm.¹ However, these languages require the user to specify the data and sometimes the work distribution; yet both of these have a profound influence on the execution time, since they greatly affect the load balance as well as the amount and pattern of data exchange between processors.

Alignment analysis is a helpful prerequisite for specifying data distribution functions, either automatically or by the user. It deals with the problem of reducing communication costs due to array cross-references by detecting which dimensions of arrays are generally used together. When aligning two array dimensions the aligned array inherits its distribution function from the source array via an alignment function, thus alleviating the task of specifying data distribution functions. Alignment primitives are included in state of the art data parallel languages like HPF [8] or Vienna Fortran 90 [2].

In [15] we presented a tool performing intraprocedural alignment analysis automatically within the framework of the VFCS compiler [3]. This tool exploits the sophisticated capabilities of VFCS such as

¹ Although the hardware now seems to evolve from the DMMP architecture towards more complex architectures providing different levels of parallelism (like e.g. clusters of SMPs), DMMPs are still widely in use. Furthermore we do not expect the new architectures to be easier to program and believe that the concepts of data locality will remain important.
subscript normalization [19] in order to enlarge the applicability of alignment analysis. For estimating the goodness of an alignment preference sequential performance data provided by the Weight Finder [7], an advanced profiler for sequential Fortran programs, is used.

Traditionally the alignment problem can be split up into two subproblems: The *interdimensional problem* deals with the problem which array dimensions should be aligned and the *intradimensional problem* tries to find the relative position of those arrays to each other in order to find appropriate alignment functions.

In this paper we focus on the kernel parts of the tool, namely the problem of weighting interdimensional alignment preferences and the model for intradimensional alignment preferences. We present our approach for both problems in detail and show, how intradimensional preferences (expressed with the *locality* concept) drive the solution of the interdimensional problem. Furthermore we show how the locality can be used to model multiple occurrences of the same intradimensional preference. This allows the detection of conflicts and contributes moreover to the accuracy of the results. We make extensive use of examples throughout our work to motivate and explain the concepts used.

The field of automatic alignment and, subsequently, automatic data distribution became very popular during recent years. The pioneering work was done by Li and Chen at Yale University [16] and Manish Gupta [12] whose results strongly influenced our analysis. Instead of using heuristics to solve the alignment problem Kremser [13, 14] proposed to use 0/1 integer programming techniques. The approach of Garcia et al. [9] introduces a new framework combining both alignment and data distribution analysis. Recently their work was extended to dynamic intraprocedural analysis taking control flow information into account [10]. However, their approaches treat the inter- and intradimensional alignment as orthogonal problems and do not take into consideration the impact of intradimensional preferences on the interdimensional solution.

Other researchers such as [6, 17, 5] deal with the alignment problem by solving matrix equations based upon the array access patterns. Although with these approaches obviously both problems are solved contemporaneously, they seem to be more restricted than ours and can moreover be very time consuming, thus preventing the analysis of bigger applications. Chatterjee et al. [4] managed the problem using algorithms in the field of dynamic programming.

The remainder is organized as follows: Section 2 introduces the concepts needed for interdimensional analyses. Strong emphasis is laid on weighting the alignment preferences. In Section 3 the intradimensional problem, expressed through the locality, is handled in full detail. The solution of both problems including their interaction is presented in Section 4. Some concluding remarks will end this paper.

## 2 The Interdimensional Alignment Problem

In this section we deal with the interdimensional alignment problem using the fundamental concepts introduced by Li and Chen [16] and Manish Gupta [12]. Our main extensions to their works are presented elsewhere [15] and we focus on our heuristic for weighting alignment edges after a short summary of the overall problem.

The interdimensional alignment problem is loosely defined as the problem of detecting array dimensions which are most frequently used together. These dimensions are coupled for the distribution step, thus avoiding communication effort due to cross references. Preferences for aligning two array dimensions are gathered by analyzing their subscript expressions and modeled in a graph theoretic model, the Component Alignment Graph (CAG).

### 2.1 The Component Alignment Graph

To model the alignment problem we use the CAG framework introduced by Li and Chen [16] which we extend with respect to the weight with which an edge is annotated (cf. Section 2.2) and we also include intradimensional information (cf. Section 3). Alignment preferences are detected using the framework of Manish Gupta [12] which we extended substantially in [15]. We introduced the concept of *identity* of two subscript expressions (i.e. where they are may be exactly the same, or may differ by a constant or invariant offset) and extended the *single index* concept allowing invariant components within the

---

2 Gupta defined that a subscript expression may contain only one loop index and all other components must be integer variables
subscript expressions. Using further the variation level concept of Gupta (i.e. the nesting level of the deepest loop changing the value of a subscript expression) we defined an improved alignment preference, also given here:

Definition 1. Let $S$ be an assignment statement in an arbitrary loop nest, $a$, $b$ and $c$ be references to arbitrary dimensioned arrays $A$, $B$ and $C$ respectively, $A \neq B \neq C$, with $\text{expr}_a(a)$ and $\text{expr}_b(b)$ as the respective subscript expressions in dimension $d$ and $t$.

If

- $\text{expr}_a(a)$ and $\text{expr}_b(b)$ are of type single index with the same variation level or of type ident

and $S$ is either

- of the form $a = F(b)$

- of the form $c = F(a, b)$ and neither $a$ nor $b$ are alignment preferring with $c$,

those two dimensions of $A$ and $B$ are called alignment preferring.

The CAG is constructed by inserting a node for each dimension of each array and connecting each two alignment preferring nodes with an undirected, weighted edge, the alignment edge ($e$). In the remainder we use the following notation: A node of the CAG is denoted by the capitalized name of the array, subscripted with the dimension, ($A_1$ represents therefore the first dimension of array $A$), and alignment edges are denoted by $e($node,node$);$ ($A_1, B_2$) therefore represents the edge between the first dimension of array $A$ and the second one of array $B$.

2.2 Weighting the Edges

As mentioned above, the edges of a CAG are annotated with information about the underlying alignment preference. Due to the fact that the alignment analysis should provide some information helpful for avoiding communication, a weight indicating the presumed communication effort arising from a cross reference might be useful. Li and Chen [16] modeled a very coarse estimate by attributing an edge with the value of 1 and $e$ respectively, depending on whether communication costs could be avoided or not. [12] extended their approach by estimating the possible magnitude of communication cost, but uses strong assumptions and time consuming heuristics to do so.

In our approach we also try to obtain a comparable heuristic measure for assumed communication costs, but we will achieve this goal with simpler heuristics, thus reducing the time spent in computation while nevertheless gaining a sufficient level of reliability. Since neither distribution schemes nor the number of processors are known at analysis time, we cannot provide a realistic measure of communication costs. Hence we use strong worst case assumptions regarding the possible communication effort throughout our work. A test run of the program is profiled using the Weight Finder and we use the execution frequency of specific statements, as well as values of loop bounds and true ratios.

A simple approach for estimating the presumed worst case communication costs would be to assume that at each single statement instance one data item has to be communicated. However, this approach is quite straightforward and even without any communication optimization such a situation will hardly occur at each statement. Another possibility would be to assume that we have to communicate one data item each time the subscript expression changes. This assumption makes sense since it allows us to examine each dimension independently of the others and is thus closely related to the interdimensional alignment problem. But again, this approach does not take into account any dependence information nor communication optimization and can again lead to inexact results which might differ from the real communication costs in both directions. Nevertheless, we will compute this value, denoted as alignment frequency, by taking the Weight Finder frequency of a (virtual) statement nested at the same level as the variation level of the alignment edge. This measure will be used only to show the effects of our communication optimization model which we describe below.

A more exact estimation of the real worst case communication costs can be achieved by modeling the compiler optimizations, especially the vectorization of communication [11]. Since we know, that we can move the communication outside the loops if no true dependence is violated [20], we can denote the nesting level to which the communication caused by an alignment edge may be extracted as the communication
dependence (cd). Hence the cd is equal to the deepest nesting level of loop-carried dependences or the deepest nesting level of a statement involved in a loop-independent dependence relation.

Now we can easily compute both the frequency with which a communication routine might be invoked, and the amount of data each message will transmit in the worst case:

**Definition 2** Let \( F(lev) \) be the frequency of a statement at nesting level \( lev \), \( varlev(e) \) the variation level of alignment edge \( e \), \( cd(e) \) the communication dependence of \( e \) and \( NI(DO) \) the number of iterations of the loop at the nesting level \( varlev(e) \). With this information we can calculate the data amount sent by each communication instance (communication amount \( ca \)) and the frequency a communication routine might be invoked (communication frequency \( cf \)) as follows:

\[
ca = \begin{cases} 
1 & \text{if } varlev(e) \leq cd(e) \\
NI(DO) & \text{if } varlev(e) > cd(e)
\end{cases} 
\]

\[
 cf = F(cd(e)) 
\]

Note that so far we do not include the true ratio in case of control dependent statements, since also the compiler's parallelization process uses worst case assumptions for each conditional branch. Anyway, the true ratios for each statement are provided by the Weight Finder and can easily be included in the below formula.

We also included a simple machine model in our communication model using the average startup time a machine needs to start a transmission and the average transfer time, required to transmit a single data item.

Now we are able to compute the respective weight of an alignment edge using a simple, yet effective method that leads to Definition 3.

**Definition 3** Let \( e \) be an arbitrary alignment edge. This edge is annotated with a weight representing an heuristic estimation of the possible communication amount due to this edge, calculated by:

\[
\omega(e) = (tt * ca + st) * cf 
\]

with \( \omega(e) \) symbolizing the weight of edge \( e \), \( tt \) the transfer time, \( ca \) the amount of communication, \( st \) the startup time and finally \( cf \) the communication frequency.

This weight associated with an alignment edge will be used in subsequent steps to evaluate the relative importance of an alignment preference and thus for solving the alignment problem and matching the nodes to the respective sets (cf. Section 4.1).

We want to motivate our models for interdimensional alignment with a short example:

**Example 1** In the example code given in Figure 1 both statements are nested and all references within them contain alignment preference information, however, with some intra- and interdimensional conflicts. Focusing on the interdimensional conflicts we find that the references to array A and C reflect an unsolvable conflict. Hence we must accept a certain amount of communication, but the alignment analysis will help us to minimize this.

The resulting CAG of the interdimensional alignment analysis is plotted below and the table holds all necessary information about the alignment edges: for each edge we print the alignment frequency, the dependence information (0 means that no dependence is detected and the eventual communication can be extracted from loops), and finally the weight (the startup time was fixed with 100 and the transfer time with 0.1 - this reflects the real order of magnitude for the Meiko CS-2). Note that the dependence information is not an edge attribute. It is printed in order to demonstrate the computation of the weight. (In fact the dependence information cannot be annotated to an edge because an edge may symbolize multiple statements.)

The importance of modeling communication optimization becomes evident when looking at the two edges \( A_1 - C_2 \) and \( A_2 - C_2 \), which both have an alignment frequency of 10000 but the difference of their weights is on the order of \( 10^8 \).

---

3 FR gives the associated frequency and LP the value of loop bounds provided by the Weight Finder.
3 The Intradimensional Alignment Problem

The second subproblem of alignment analysis is the problem of finding the relative position of array references which are alignment preferring. This problem is generally referred to as the *intradimensional alignment problem*. Solving this problem can give useful information about the alignment functions which should be applied when aligning and distributing these two dimensions. The main alignment functions under consideration are shift, stride and the reflection. Furthermore intradimensional preferences have a strong impact on the solution of the interdimensional problem since conflicts in intradimensional preferences might lead to a certain amount of communication effort unavoidable when distributing the arrays. In [15] we introduced the concept of the locality an alignment edge is annotated with. The next subsection will give a brief summary on the definition of locality, before we go into deeper details and show the detection of competing intradimensional preferences and how intradimensional preferences drive the interdimensional model.

### 3.1 The Locality Relation

To model intradimensional preferences we use the locality as defined in Definition 4. Note that we have defined intradimensional alignment preferences only on array references with regular subscript expressions which are either identical or consisting of only one DO-loop index while all other components are loop invariant. We can therefore either exactly compute intradimensional shifts and different strides or can at least say that the alignment preference is based on invariant components, thus keeping the same (unfortunately unknown) intradimensional preferences for all loop iterations. A runtime system may be used to get more information about the behavior of those invariant components.

**Definition 4** Let $e$ be an alignment edge lying upon two subscript expressions $expr_a(a)$ and $expr_b(b)$. The locality between these subscript expressions is defined as a tuple (locality type, locality relation) and $e$ is attributed with this tuple.

The locality relation consists of the tuple (multiplicator, constant offset) or $(m,o)$ where the multiplicator is defined as the quotient of the two index coefficients $(a/b)$ and the constant offset is defined as the difference of the two constant components $(\gamma - \delta)$ respectively. The locality relation is related to one of the two subscript expressions.

The locality type is defined as follows:
locality type = \[
\begin{cases}
\text{perfect} & \text{if the locality relation } = (1,0) \\
\text{constant} & \forall \text{ locality relations:} \\
\{m, o\}|m \text{ and } o \text{ are integer constants } \wedge (m \neq 1 \lor o \neq 0) \\
\text{invariant} & \text{otherwise}
\end{cases}
\]

Note that the locality relation is not symmetric, i.e. \(A \otimes B \neq B \otimes A\)!
Due to the lack of symmetry in the locality relation, a kind of "direction" is put on the alignment edges which must be observed very carefully when handling the CAG.

For alignment preferences emerging from ident subscript expressions we can deduce the locality by

**Lemma 1** If two subscript expressions are said to be ident then they have a multiplicative of 1 and a constant or invariant offset

The locality relation can give useful hints for choosing the appropriate alignment and distribution functions and also helps us handle multiple occurrences of the same alignment preferences (this means that the same two array references are alignment preferring in distinct parts of the code). Furthermore, competing intradimensional alignment preferences can be detected.

A continuation of example 1 shows the application of the locality concept:

**Example 2** After the intradimensional analysis the CAG shown in Figure 1 is fully annotated as depicted in the below table. Note that in the case of constant locality the locality relation is connected to one of the two participant nodes:

<table>
<thead>
<tr>
<th>EDGE</th>
<th>FREQ.</th>
<th>LOCALITY TYPE</th>
<th>LOCALITY RELATION</th>
<th>DEP.</th>
<th>WEIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_1 \cdot B_1)</td>
<td>100</td>
<td>perfect</td>
<td>((1,0))</td>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td>(A_1 \cdot C_1)</td>
<td>100</td>
<td>constant</td>
<td>((1,1)) for (C_1)</td>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td>(A_1 \cdot C_2)</td>
<td>10000</td>
<td>perfect</td>
<td>((1,0))</td>
<td>2</td>
<td>100100</td>
</tr>
<tr>
<td>(A_2 \cdot B_1)</td>
<td>10000</td>
<td>perfect</td>
<td>((1,0))</td>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td>(A_2 \cdot C_1)</td>
<td>100</td>
<td>perfect</td>
<td>((1,0))</td>
<td>2</td>
<td>100100</td>
</tr>
<tr>
<td>(A_2 \cdot C_2)</td>
<td>10000</td>
<td>constant</td>
<td>((1,2)) for (C_2)</td>
<td>0</td>
<td>110</td>
</tr>
</tbody>
</table>

3.2 Multiple Alignment Occurrences and Intrdimensional Conflicts
The locality relation is used to detect and solve conflicts in the intradimensional alignment scheme. Such conflicts can occur when more than one pair of array references express an interdimensional alignment preference.

Other researchers deal with this problem by simply accumulating the weights of all preferences thus enlarging the probability for its consideration in the final solution. This procedure is however inaccurate and we show, that better results can be obtained using the concept of locality.

The following example illustrates the importance of locality when handling multiple occurrences of the same interdimensional preference:

**Example 3** Suppose the following two statements are both nested in arbitrary loops and loop index \(i\) goes from 1 to \(k\). Both arrays express an interdimensional alignment preference in the first and second dimension, respectively. However in the second statement the \(i\) subscript of array \(A\) is used in a reflection function.

\[A(i, j) = B(i, j)\]
\[B(i, j) = A(N-i, j)\]

If we simply accumulate the weight of these two preferences we would model the situation incorrectly, since in this case a certain amount of communication remains no matter what distribution function will be applied. Fortunately we are able to detect such cases via the locality: the locality of the two preferences would be \((\text{perfect},(1,0))\) and \((\text{constant},(1,0))\) respectively, thus making an intradimensional alignment conflict evident.
We can define the case when the localities of two alignment preferences have now conflicts and refer to them as *ident localities*:

**Definition 5** The localities of two alignment preferences are said to be ident if they are both of type perfect or constant, have the same multiplicator and their constant offset differs only by a small constant.

In the above definition we allow a difference in the constant offset in order to include stencil operations in it. Nevertheless the difference should not be more than ±5.

Another aspect for the intradimensional model is the question whether the values referenced multiple times might have changed or not, thus if (potentially) a renewed communication is necessary or not. Since an edge of the CAG is not connected to a specific statement, we must assume - without the use of more sophisticated analysis - the worst case when detecting a second occurrence of this preference. Only when an array is identically referenced multiple times in the same statement, we can immediately infer that no renewed communication will be necessary.

Suppose $A$ and $B$ are two occurrences of an alignment preference between the same pair of array dimensions. The one of the four cases will hold and they are handled as described:

**Case 1** $A$ and $B$ belong to the same statement and have an ident locality:

$B$ is redundant information and can be ignored because any potential communication is modeled by $A$.

**Case 2** $A$ and $B$ belong to the same statement and have a different locality:

In this case an intradimensional conflict arises. We model this by generating a new edge in the CAG for preference $B$ and annotate it as usual.

**Case 3** $A$ and $B$ belong to different statements and have an ident locality:

This case implies a strengthening of the alignment preference, because the possible communication effort may arise twice if the two dimensions are not aligned. We model this by accumulating the weights of both preferences.

Since we allowed ident localities to have a small difference in the constant offset, we store the maximum and minimum offset of all accumulated preferences. The locality relation therefore has the form $(m, \sigma_{\min}/\sigma_{\max})$.

**Case 4** $A$ and $B$ belong to different statements and have a different locality:

This case is equivalent to the second one.

The following example based upon the code fragment given in Figure 2 motivates our differentiation.

**Example 4** The analysis of the code in Figure 2 results in alignment preferences between $A_1 - B_2$ and $A_2 - B_1$ in both statements. The intradimensional analysis of statement $S_1$ results in an perfect locality for both preferences. The preferences in statement $S_2$ belong to one of the above cases:

The preference between $A_1 - B_2$ due to the first occurrence of $A$ on the rhs is annotated with a perfect locality, thus it belongs to case 1 and the weights of both preferences are accumulated. The second occurrence of $A$ on the rhs also expresses an alignment preference with $B_2$ annotated with a perfect locality. This preference belongs to case 1, thus nothing needs to be done.

We can further detect an alignment preference between $A_2 - B_1$, which is however annotated with a constant locality due to the shift in the subscript of $B$. Hence this preference belongs to case 4 and a second edge is constructed between those nodes, thus representing an intradimensional conflict. When looking at the CAG depicted in Figure 2 we can see two edges between those nodes: edge "a" is generated from statement $S_1$ whereas edge "b" results from statement $S_2$.

### 4 Solving the Alignment Problem

This section presents our heuristic solution algorithms which combine both the solutions for the inter- and the intradimensional problems. Major focus is laid on the impact of intradimensional preferences on the interdimensional solution.
4.1 Solving the Interdimensional Problem

The Heuristic Algorithm

To solve the interdimensional problem we must match all nodes of the CAG to disjoint sets (the Alignment Sets) in such a way that no two nodes of the same array belong to the same set and the accumulated weight of all edges between sets is minimized[16]. All nodes within such a set should be aligned. The remaining edges symbolize unavoidable communication. Unfortunately this problem was proved to be NP-complete by Li and Chen [16], however they presented an heuristic algorithm which reduced the problem to a bipartite-graph matching problem. In our approach we rely on their heuristic algorithm which we changed in some details in order to improve its results.

The heuristic algorithm begins by creating initial Alignment Sets: Each Alignment Set contains a single element which is a dimension of an array in the code. The array selected must have the highest accumulated weight of alignment preferences in that program unit. At each step the algorithm constructs a bipartite graph from the current Alignment Sets and an array which is to be matched to them. There is a node for each dimension of this array. Once matched, the nodes are merged with the corresponding Alignment Set. A weighted edge connects a set with an array node if there is an edge between this node and any node belonging to the set in the CAG. When building the bipartite graph, the intradimensional problem is dealt with as shown below.

In the original algorithm, indirect alignment preferences are given a weight (in most cases too high) independent of other matchings; this has been improved by a new algorithm for building a bipartite graph which only considers direct alignment preferences (the transitivity of the alignment preference is observed by ordering the graph). Ayguade et al. proposed to use the min-cut of all paths connecting the nodes, thus achieving more accurate results [1]. We are planning to investigate their proposal in the near future.

Furthermore the results computed by the original algorithm depended to some extent on the (random) order in which columns were chosen. This has been overcome by ordering the CAG with respect to the weight of the edges.4

4 An outline of the heuristic algorithm can be found in [15].

4.2 Impact of Intradimensional Preferences

In section 3.2 we emphasized the importance of observing intradimensional conflicts and showed how to model them. The same is valid for the interdimensional solution phase. However, conflicting intradimensional preferences do not only occur between the same two nodes, but also during the matching step in the interdimensional solution.

In order to detect these additional intradimensional conflicts, we have to modify the locality in such a way that it expresses a relationship between a node and all nodes belonging to a set rather than a relation between two single nodes. We therefore relate all the localities to one node, called the basis node. The first node associated with a set becomes its basis node5 and is attributed with a perfect locality. When an array dimension is included in an Alignment Set, the locality relationship with the basis node is stored in order to retain details of alignment information. We call this the node locality. When other nodes are successively matched with a set, the edge of the bipartite graph is annotated with alignment

5 Note that the basis nodes of all sets may not necessarily belong to the same array
Let \( k \) be a node matched to an Alignment Set and \( \text{loc}(A) \) its associated locality. If a node \( B \) with an alignment edge \( e(A, B) \) should be matched to the same set, the locality information of \( A \) is propagated to \( e \) as follows:

\[
\begin{align*}
\text{if (loc}(e) &= \text{perfect}) \\
\quad \text{loc}(e) &= \text{loc}(A); \\
\text{elseif ((loc}(e) &= \text{invariant}) || (\text{loc}(A) &= \text{invariant})) \\
\quad \text{loc}(e) &= \text{invariant}; \\
\text{elseif ((loc}(e) &= \text{constant}) || (\text{loc}(A) &= \text{constant})) \\
\quad \text{loc}(e) &= \text{constant}; \\
\quad \text{multiplicator}(e) &= \text{multiplicator}(e) \times \text{multiplicator}(A); \\
\quad \text{constant offset}(e) &= \text{constant offset}(e) + \text{constant offset}(A); \\
\end{align*}
\]

Figure 3: Propagation of Locality Information

Information summarizing the information of all CAG edges between the candidate node and any node associated with the set.

The node locality is propagated to the edge in order to establish a relationship between the candidate node and the basis node. The algorithm for propagating locality information in order to relate all of them to the basis node is given in Figure 3.

If we propagate all locality information while constructing the bipartite graph, we are able to compare pairs of alignment preferences and deal with them as described in the distinguishing cases of Section 3.2. Thus all edges are merged if there are no conflicts, or several distinct edges connect the bipartite graph in the case of intradimensional conflicts.

The following example gives an intuitive understanding of the propagation algorithm:

Example 5  Consider the following code excerpt with the given alignment preferences:

\[
\begin{align*}
A(I) &= B(100-I) \quad e(A_1, B_1): \text{locality } = \text{constant}, \text{ locality tuple for } B_1 &= (-1, 0) \\
A(I) &= C(100-I) \quad e(A_1, C_1): \text{locality } = \text{constant}, \text{ locality tuple for } C_1 &= (-1, 0) \\
B(I) &= C(I) \quad e(B_1, C_1): \text{locality } = \text{perfect}
\end{align*}
\]

Furthermore we assume that \( A_1 \) is matched as the first node to Set 1, thus becoming the basis node. The next node we want to match to Set 1 is \( B_1 \), having a constant locality with \( A_1 \). It is therefore attributed with its locality tuple \((-1, 0)\). Now \( C_1 \) is to be matched and we detect two alignment preferences \( e(A_1, C_1), e(B_1, C_1) \), but with different localities. There is a reflection of \( C_1 \) in relation to \( A_1 \) and a perfect locality to \( B_1 \). Since \( B_1 \) is also a reflection in relation to \( A_1 \), we can propagate the locality information of \( B_1 \) to the edge \( e(B_3, C_1) \) as shown in Figure 3. This results in a constant locality expressing a reflection for this edge, thus solving the intradimensional conflicts, as is evidently appropriate for the code.

4.3 Solving the Intradimensional Alignment Problem

The solution of the intradimensional alignment problem consists of two steps: first potential conflicts (expressed by the fact that there is more than one edge between a set and a node in the bipartite graph) have to be solved; then the nodes matched to an Alignment Set have to be annotated with their locality information relative to the basis node (the node locality).

The detection of conflicts in the intradimensional alignment preferences indicates that a certain amount of communication effort will remain, no matter what distribution function is chosen. To model this situation correctly we would have to accumulate the weights of all competing edges, reduced by the unavoidable communication effort. Unfortunately without any a priori knowledge of distribution functions and the available processors, we are not able to estimate this.

To overcome this problem, we decided to rely still on worst case assumptions. We assume that only one of the intradimensional preferences can be observed, the others cannot. Hence we consider only the edge with the highest weight for the bipartite graph matching step and ignore all the others.
Let us denote by $e$ an edge of the bipartite graph, $A_k$ any node belonging to set $S_l$, and $B_j$ a node of the CAG. Then we can formulate the weight of the edge of the bipartite graph as follows:

$$\omega(e) = \max \left( \sum_{i \in \text{dest set}} \omega(e(A_k, B_j)) \middle| A_k \in S_l \right)$$

The second step of this phase is solved in a straightforward manner. We can simply deduce the propagated locality an edge of the bipartite graph is annotated with to the node. Of course, it is important to record whether conflicts regarding a node have arisen during analysis. Such an information can indicate that the user should take care when distributing this array. In order to do so, we introduce a new locality type competing; a node is attributed with this locality type if intradimensional alignment conflicts have occurred.

The output of the heuristic algorithm are the Alignment Sets, containing all nodes of the CAG so that the weights of the edges between nodes belonging to different sets is minimized. Those edges represent unavoidable communication and reflect conflicts in the interdimensional alignment. The Alignment Sets are arbitrarily numbered in order to distinguish them and can be represented as an ordered list. All nodes belonging to the same set should be aligned with each other, where the precise alignments will be based upon the intradimensional information with which each node is annotated.

The alignment information of an alignment set can be directly used to generate alignment directives in HPF-like languages. When generating the alignment directives, we consider an array with highest dimensionality to be the alignment source array and align all other arrays with it. Subsequently, the user only has to specify the distribution functions for this single alignment source array.

**Example 6** The solution for our example 1 is plotted in Figure 4. It shows two sets, since the maximum dimensionality of all arrays examined is two. Note that the nodes are annotated with locality information, relative to the first node. (In fact all nodes are annotated with a perfect locality because the two edges with constant locality are not considered to be observed for the final solution.)

We measured our example loop with several different alignment and distribution settings on the MEIKO CS-8 using the pglhp Compiler from The Portland Group [18]. The results can be found in Figure 5. The Figure shows that our suggested solution outperformed all other alignment settings. These results confirm our hypothesis that our alignment suggestions will be among the best ones independently of the distribution function (despite the fact that not all aspects of parallelization are modeled). The faster execution times with the $\ast, \text{BLOCK}$ distribution can be explained by the fact that parallelization of the (distributed) J-loop is possible here, but not in the other case.

![Figure 4: Alignment Sets](image)

## 5 Conclusion

In this paper we presented a pragmatic approach for modeling and solving the alignment problem. Our main contribution was a unified model for both the inter- and intradimensional alignment problem, which allows their simultaneous solution. We showed that the observance of intradimensional preferences is essential for the goodness of the interdimensional solution.

We developed a model to weight the interdimensional alignment preferences reflecting the presumed communication effort due to an alignment preference. This model uses a compiler model reflecting the
communication optimization and a machine model for communication parameters (transfer and startup time). Furthermore we used a sequential profiler to estimate the worst case communication amount.

The intradimensional alignment problem was modeled together with the interdimensional problem introducing the concept of locality. Locality is a relation between two array dimensions reflecting their relative position to each other (shifts, strides, reflection). We showed how locality can be used to detect conflicts in the intradimensional alignment preferences and how these conflicts drive the solution of the interdimensional problem. Therefore we developed a model to differentiate multiple occurrences of the same interdimensional preference. Furthermore the locality can be related to the array dimensions, thus allowing the deduction of alignment functions directly. For solving the interdimensional alignment problem we adopted the heuristic algorithm introduced by [16] to get more accurate results and included the intradimensional solution step in it.

Future work will focus on the interprocedural analysis problem including the topic of realignment between phases of the program, thus making the tool a real general purpose tool. The extension of our models towards automatic data distribution is one of the major research directions. We believe that our models can easily be extended towards automatic data distribution with minor changes. In this context we will study the proposals of Garcia et al. [9, 10] for including parallelism information in the alignment model and will consider replacing the current heuristic algorithm for solving the alignment problem with integer programming techniques, as proposed by Kremer and Garcia et al. However, we have to carefully study the time complexity of these techniques within our framework.

References


Data Flow Analysis of Recursive Structures

Albert Cohen\textsuperscript{1}, Jean-François Collard\textsuperscript{1,2}, Martin Griebl\textsuperscript{3}

\textsuperscript{1} PRISM, Univ. de Versailles, 45 Av des Etats-Unis, 78035 Versailles, FRANCE
\textsuperscript{2} Kungliga Tekniska Högskolan, Electrum 204, 164 40 Kista, SWEDEN
\textsuperscript{3} FMI, Universität Passau, Innstraße 33, 94032 Passau, GERMANY

Abstract. Most imperative languages only offer arrays as “first-class” data structures. Other data structures, especially recursive data structures such as trees, have to be manipulated through explicitly management of memory. On the other hand, recursion in the flow of control also is an open problem in automatic parallelization. To help and solve this problem, this paper proposes a data flow analysis for both data and control recursive structures.

\textit{Keywords:} Static analysis, automatic parallelization, compiler

1 Introduction

Data structures in imperative languages are in general of little variety. In most cases, such structures are arrays, records and unions of basic types. The advantage of such restrictions is that the the compiler can construct a finite representation of data structures. When possibly infinite data structures are to be used, most imperative languages rely on pointers and dynamic memory allocation. For instance, a binary tree can be expressed and manipulated using a record of one datum and two pointers. This record has finite size, easing the work of the compiler (and of the compiler writer). Of course, there are languages which offer recursive data structures without explicit manipulation of memory. Most logic or functional languages do so. As an example of imperative language, see for instance CLU [1]. However, we believe most languages do not provide this feature because of the lack of suitable compile-time analyses. The aim of this paper is to report preliminary results on the data flow analysis of imperative languages offering both recursive control and recursive data structures. Section 2 defines our program model and introduce a small toy language. Section 3 describes our data flow analysis. Section 4 gives two extended examples.

2 The Source Language

We present below a subset of LEGS [2], a toy language that allows to cleanly define data and control recursive structures. It is enough here to say that LEGS embodies restrictions and assumptions we have had to make on the way recursive structures are defined and handled. (See Section3.1.)

2.1 Defining Recursive Data Structures

Let $E$ by a finite alphabet and $\epsilon$ the empty word. To each word in the language corresponds an element of the data structure. In this paper, we only consider two classes
of languages on $E$: those where concatenation is the usual one, and those where concatenation is commutative. (We will say that the data structure itself is commutative or not.) For instance, if $E = \{l, r\}$, then a non-commutative concatenation labels the nodes of a binary tree. Otherwise, if $l.r = r.l$, then an “array” is described.

2.2 Recursive Control Structure

Recursive control is expressed by (possibly nested) explore constructs, each consisting of three parts: labels to recursive calls, a predicate $P$ controlling the recursion, and a body. Statements in the body are of two kinds: recursive calls and “regular” statements (assignments, I/O, etc.). Statements of both kinds may be arbitrarily mixed and are separated by a semicolon. All statements are labeled. Recursive calls are done by key word recurse.

The main benefit of this construction, when compared to guarded recursive calls, is that the execution of the body occurs if and only if the predicate of the explore holds. (Consider a simple recursive function, say $f(x) = \text{if } x = 0 \text{ then } 0 \text{ else } f(x - 2)$. Deriving that $f$ is defined for even integers already needs some semantical analysis: an issue we wanted to avoid here.)

Definitions The set of assignment labels $\{S_1, \ldots, S_n\}$ is denoted by $\text{STMTS}$. The set of labels of recursive calls in all explore constructs is denoted by $\text{LABELS} := \{C_1, \ldots, C_n\}$. We also define $\text{UP} = \{C_1^{-1}, \ldots, C_n^{-1}\}$, i.e., the set of inverses of recursive call labels. Any execution of an assignment can thus be uniquely labeled by a word in $\text{LABELS}^* \cdot \text{STMTS}$. Such a word is called a control word.

Since a statement is a static object (a “line” in the program), we call operations the (run-time) instances of statements. Statements are labeled in the program text. Labeling operations can be done using a pair $(S, w)$, where $w$ is the control word.

The textual order on labels is denoted by $\prec$. For instance $b \prec S \prec z$ in Figure 2. The order in which statements are executed is thus simply given by the usual lexicographic order, denoted $\ll$, on $\text{LABELS}^* \cdot \text{STMTS}$. Let $w, w'$ be two words labelling instances of Statements $S$ and $S'$, respectively. (So, $w \in \text{LABELS}^* \cdot S$ and $w' \in \text{LABELS}^* \cdot S'$.) Thus: $(S', w') \prec (S, w) \equiv w' \ll w$. A formal definition of the order is:

$$w' \ll w \equiv \exists u \in \text{LABELS}^*,$$

$$(\exists x \in \text{LABELS} \land \exists v \in \text{LABELS}^* \cdot S) \text{ or } (x = S \land v = e),$$

$$(\exists x' \in \text{LABELS} \land \exists v' \in \text{LABELS}^* \cdot S') \text{ or } (x' = S' \land v' = e),$$

$$w = u \cdot x \cdot v, w' = u' \cdot x' \cdot v', \text{ and } x' \prec x$$

2.3 Accessing Elements of Data Structures

To access an element of a data structure, the name of the structure is followed, between brackets, by a word in the language of the data structure: structure [ access word ]. access word has to be the image of the current control word, say $w$, by a substitution $\sigma$. This image is denoted by $w\sigma$. A substitution has the intuitive usual syntax, i.e. $\{x/a, y/ab\}$ is written as $\{x/a, y/ab\}$. Notice that letters which do not appear in the

\footnote{Note that there is some redundancy in this notation since the control word of $S$ includes exactly one $S$, at the very last position. The definition of order is still valid when $S = S'$.}
substitution are implicitly replaced by ε, since the letters in the “subscripting” word have to belong to the language describing the data structure. Figures 1 and 2 give two examples whose data flow analyses will be derived in following sections. The former is an exploration of a binary tree. The latter is a recursive program over an array-like data structure (and actually an excerpt from a program to recursive sort an array).

Fig. 1. Example LEGS program. tree is a binary tree-shaped data structure described by the language on \{1, r\}. P is some affine predicate.

Fig. 2. Example LEGS program.

3 Data Flow Analysis

The purpose of data flow analysis is to find, for any read in a given operation, which previous write produced the read value.

Let \( S' (S') \) be a statement reading (writing into) data structure \( A \) using substitution \( \sigma (\sigma') \), and whose control word is \( w (w') \). The fact that memory cells accessed by \( (S, w) \) and \( (S', w') \) are the same is denoted by \( A [ \, w \sigma ] := : A [ \, w' \sigma' ] \).

So, for any instance \( (S, w) \) of Statement \( S \), we are looking for the last instance \( w' \) of \( S' \) that wrote into \( A [ \, w \sigma ] \), i.e., such that:

\[
A [ \, w \sigma ] := : A [ \, w' \sigma' ] \ \Leftrightarrow \ w \sigma = w' \sigma' \tag{2}
\]

\( (S', w') \) is then called the source of the read in \( (S, w) \).
Our framework for data flow analysis, first developed by Feautrier[3], consists in two steps. For a given statement $S$:

- Construct the set of instances of $S'$ that are possible sources of $\langle S, w \rangle$. This set $Q_{S'\preceq}(w)^5$ is built from all operations $\langle S', w' \rangle$ such that: (1) $\langle S', w' \rangle$ is actually executed (2) $\langle S', w' \rangle$ writes into the memory cell that $\langle S, w \rangle$ reads, and (3) $\langle S', w' \rangle$ executes before $\langle S, w \rangle$, written $\langle S', w' \rangle \prec \langle S, w \rangle$. I.e., $Q_{S'\preceq}(w) = \{w' | P(w'),(2), w' \ll w\}$.
- The source is $K_{S'\preceq}(w) = \max_{\ll}(Q_{S'\preceq}(w))$.

It may be the case that the source does not exist. $K_{S'\preceq}(w)$ is then by definition equal to the undefined operation $\bot$. By convention, $\bot$ is the earliest operation in the program, i.e.,

$$\forall S, \forall w, \bot \prec \langle S, w \rangle \lor \langle S, w \rangle = \bot. \quad (3)$$

For technical reasons, we cannot handle directly the disjunction in the definition (1) of $w' \ll w$. So, we "split" $Q_{S'\preceq}(w)$ into $n$ subsets of possible sources $Q_{S'\preceq}^1(w), \ldots, Q_{S'\preceq}^n(w)$ according to each disjunct. We solve them in turn, the results being $K_{S'\preceq}^1(w), \ldots, K_{S'\preceq}^n(w)$. Each $K_{S'\preceq}^i(w)$, $1 \leq i \leq n$, is a nested conditional whose predicates depend on $w$ and whose leaves include the maximum element according to order $\prec$. Notice that this order is strict.

We then combine the intermediate results in any order, using rules similar to those given in [3]: two intermediate results are merged by plugging one of them at the other's leaves. Leaves are equal to the lexicographical maximum of the two corresponding leaves. Leaves governed by contradictory predicates are dismissed. Obviously, this phase has to be repeated for all possible statements $S'$, and the intermediate results have to be combined.

### 3.1 Program Model: Restrictions on Input Programs

Let $\#_l(w)$ denote the number of occurrences of letter $l$ in word $w$. In this preliminary study of data flow analyses for regular structures, we assume the following restriction on the input program:

**Recursion Predicate** Let $w$ be a control word on $\{C_1, \ldots, C_n\}$. Then, we restrict ourselves to predicates $P$ of explore that are conjunctions of affine (in)equalities in the number of occurrences of labels of LABELS in control words. Notice that predicate $P$ does not have to be a function of recursion depth.

**Substitutions** Substitutions are restricted to mappings from one letter (of the language of control words) to a word (in the language of the data structure).

**The Word Problem** Finding the elements in $Q_{S'\preceq}(w)$ requires to check that $\exists \sigma : w\sigma = w\sigma'$, i.e. that $w\sigma = w\sigma'$. Checking that two words are equal, known as "the word problem", is undecidable. We thus restricted ourselves to words of which a normal form can be defined.

---

5 When clear from the context, both $S$ and $S'$ will be dropped.
3.2 Numbered Occurrence Languages

The pumping \([e_1^{i_1}, \ldots, e_d^{i_d}]\) of an alphabet \(\Sigma = \{e_1, \ldots, e_d\}\) by a mapping of \(\Sigma\) into \(Z\) which maps \(e_j\) into \(i_j\) is the language built of all words containing exactly \(i_j\) occurrences of \(e_j\), for all \(j\). An affine parametrized pumpings, denoted by

\[
\frac{[e_1^{i_1}, \ldots, e_d^{i_d}]}{A(i_1, \ldots, i_d)}
\]

is the union of all pumpings such that parameters \(i_1, \ldots, i_d\) satisfy the system \(A\) of affine (in)equalities. The inverse of a pumping \(P\) has no sense by itself but we define the notation \(w.P^{-1}\), where \(w\) is a word as the set of prefixes of \(w\) which corresponding suffixes are elements of \(P\). \(P^{-1}\) can be seen either as a “selective” back space over \(w\), or as a back space “recording” what has been erased. is the language \(\{w' \mid 3w \in P, w' = w^{-1}\}\).

A pumping is finite if the corresponding language is finite. This can be checked statically by checking that upper bounds on all parameters exist.

More generally, a numbered occurrence language, NOL, is given by \(p\) words \(M_1 \ldots M_p\) (possibly empty), \(p - 1\) pumpings (or pumping inverses) \(P_1 \ldots P_{p-1}\), and a conjunction \(A\) of affine (in)equalities. An NOL is denoted by

\[
L = \frac{M_1.P_1 \cdots M_{p-1}.P_{p-1}.M_p}{A}
\]

(Notice that several pumpings in an NOL may share integer parameters that are constrained by \(A\).) A simple NOL has no pumping inverses.

**Lemma 1.** If predicate \(P\) controlling the explore is affine (cf Section 3.1) and if the data structure is commutative, then any subset of possible sources \(Q_{S'/S}^j\) (for any \(j\)) is a NOL.

**Proof.** Let \(w\sigma\) and \(w'\sigma'\) be the two words defined in (2). Since the structure is commutative, both words have a normal form, which are \(w\sigma = e_1^{#_{e_1}(w\sigma)} \ldots e_d^{#_{e_d}(w\sigma)}\) and \(w'\sigma' = e_1^{#_{e_1}(w'\sigma')} \ldots e_d^{#_{e_d}(w'\sigma')}\), respectively. These normal forms are equal iff:

\[
\bigwedge_{i=1}^d #_{e_i}(w\sigma) = #_{e_i}(w'\sigma')
\]

Thus, due to (1), we have \(Q_{S'/S}^j = \frac{w.w^{-1}.\sigma.\sigma'.w'}{\overline{P(w')} \wedge (4)}\), hence the lemma.

In the case of non commutative structures, sets of possible sources will be approximated using NOLs. This clearly implies that our data flow analysis may yield imprecise (but always correct) results.
Computing Lexicographical Maxima on Simple NOLs

Given a word $M$ and a pumping $P$ on alphabet $\{C_1, \ldots C_n\}$, computing the lexicographical maximum of the words $M \cdot P$ is done as follows:

- Change of variables: let $x_1, \ldots x_n$ be the parameters of letter of the alphabet in $P$. Each $x_i$ may be an affine function of the original parameters. (These new variables are ordered in the same order as letters of the alphabet.)
- Compute the lexicographical maximum $(u_n, u_{n-1}, \ldots u_1)$ of $(x_n, x_{n-1}, \ldots x_1)$, under the affine constraints $A$. This is exactly what a software such as PIP[4] was crafted for.
- The lexicographical maximum in $M \cdot P$ is $C_n^{u_n} \cdot C_{n-1}^{u_{n-1}} \ldots C_1^{u_1}$

Example: What is the lexicographical maximum of $\frac{(x^n+i-j+n)}{0<n_i
\leq \ldots \leq n_j<n}$? $x_1 = n + i - j, x_2 = j \Rightarrow \text{lexmax}(x_2, x_1) = (n, n) \Rightarrow \text{lexmax}(M \cdot P) = b^n \cdot a^n$.

Lexicographical Order in the Presence of Pumping Inverses

We believe that deciding the order of two words $w \cdot P^{-1}, w' \cdot P'^{-1}$ is impossible in general, because the answer depends on the actual values of $w$ and $w'$, even when $w = w'$. However, some restricted results happened to be sufficient in practice for most (simple) programs. In the case of binary recursion, we have the following lemma:

**Lemma 2.** If $C_1 \prec S \prec C_2$ (i.e., the recursion is inorder) then, for all $w, \phi > \phi'$ and $\beta \geq \beta'$:

$$w \cdot S^{-1} \cdot [C_1^\phi, C_2^\beta]^{-1} \cdot C_1^{-1} \cdot S \ll w \cdot S^{-1} \cdot [C_1^{\phi'}, C_2^{\beta'}]^{-1} \cdot C_1^{-1} \cdot S$$

**Proof.** For both words to be defined, $w$ must be such that

$$w = z \cdot C_1 \cdot [C_1^{\phi - \phi'} \cdot C_2^{\beta - \beta'}] \cdot [C_1^{\phi'}, C_2^{\beta'}] \cdot S,$$

which is equal to $z \cdot C_1 \cdot [C_1^{\phi - \phi' - 1} \cdot C_2^{\beta - \beta'}] \cdot C_1 \cdot [C_1^{\phi'}, C_2^{\beta'}] \cdot S$. Then $z \cdot S$ is the left-hand side expression in (5), and $z \cdot C_1 \cdot [C_1^{\phi - \phi' - 1} \cdot C_2^{\beta - \beta'}] \cdot S$ the right-hand side. The first letters after the common prefix are $S$ and $C_1$, respectively. Moreover, $S \prec C_1$, hence the lemma.

Similar lemmas for binary prefix and postorder recursion can easily be found.

Hard and Soft Bottoms

As we said, $Q_{S+S}(w)$ is split into subsets $Q_{S+S}^1, \ldots, Q_{S+S}^d$ in the course of the computation. Each $Q_{S+S}^i$ may be empty for two reasons:

- $P(w')$ and (2) cannot (perhaps simultaneously) be satisfied. Then, all others $Q_{S+S}^i$, $i \neq j$, are empty too. We then know that the source of $(S, w)$ cannot be an instance of $S'$, and all the $K_{S+S}^i$ can be set to a hard bottom value, denoted by $\bot$.
- The current disjunct of (1) is not compatible with $P(w')$ and/or (2). Even though $Q_{S+S}^j$ is then empty, other subsets $Q_{S+S}^i$ may not. So, the maximal element $K_{S+S}^j$ of $Q_{S+S}^j$ should be set to a soft bottom value, denoted by $\bot$.

Notice that distinguishing hard bottoms from soft bottoms is not semantical, but speeds up computations since this allows to prune impossible cases in other conditionals.
4 Examples

4.1 First Example

The set $Q(w)$ of all previous instances $w'$ of $S$ that wrote in the memory cell read by $w$ is:

$$Q(w) = \{w' \mid P(\#L(w')) \land \#R(w') < q\},$$  
\hspace{1cm} (6)

$$w' \{L/l\}.\{R/r\} = w\{L/l\}.l^{-1}.\{R/r\},$$  
\hspace{1cm} (7)

$$w' \preceq w$$  
\hspace{1cm} (8)

(8) boils down to:

$$w' \preceq w \Leftrightarrow \#_L(w') < \#_L(w) \lor (\#_L(w') = \#_L(w) \land \#_R(w') < \#_R(w))$$

We split $Q(w)$ into $Q^1(w)$ and $Q^2(w)$ according to the two disjuncts: $Q^1(w) = \{w' \mid (6) \land (7) \land \#_L(w') < \#_L(w)\}$. From (7), we learn that $\#_R(w')$ has to be equal to $\#_R(w)$. (And since $w$ exists, this proves that $\#_R(w') < q$ holds.) Moreover, $\#_L(w')$ has to be equal to $\#_L(w) - 1$, implying $\#_L(w) \geq 1$. (Otherwise, we get a hard bottom since this property does not depend on the execution order.) Thus, the first intermediate result is:

$$K^1(w) = \text{if } w = L'^{\geq 1}.R'^{\geq 0} \text{ then } L'^{\leq -1}.R'^{\leq j} \text{ else } \bot$$  
\hspace{1cm} (9)

In a second step, $Q^2(w) = \{w' \mid (6) \land (7) \land \#_L(w') = \#_L(w) \land \#_R(w') < \#_R(w)\}$. Equations (7) and $\#_L(w') = \#_L(w)$ cannot simultaneously be satisfied, so $K^2(w) = \bot$. Merging it with (9) yields the final result:

$$\max(K^1(w), K^2(w)) = \begin{cases} 
\text{if } w = L'^{\geq 1}.R'^{\geq 0} \\
\text{then } \max(L'^{\leq -1}.R'^{\leq j}, \bot) \\
\text{else } \max(\bot, \bot)
\end{cases}$$

4.2 Second Example

We now study the program in Figure 2. Its simulated execution is given in Figure 4.2 so as to give the reader an intuitive feeling of how the flow of data behaves. Let $P(x) = \#_f(x) + \#_b(x)$.

Set of Possible Sources The set $Q(w)$ of all previous instances $w'$ of $S$ that wrote in the memory cell read by $w$ is:

$$Q(w) = \{w' \mid \#_f(w') + \#_b(w') < N, \quad \#_f(w') - \#_b(w') = \#_f(w) - \#_b(w) - 1, \quad w' \preceq w\}$$  
\hspace{1cm} (10)

Thanks to (1), (12) is equivalent to:

$$(w' = x.b.y'.S, w = x.S) \lor (w' = x.b.y'.S, w = x.f.y'.S) \lor (w' = x.S, w = x.f.y.S)$$  
\hspace{1cm} (13)

where $x, y, y'$ are in $\text{LABELS}^*$. (10) is the existence predicate, (11) is the conflict predicate, i.e. the read element is the same as the one written into. We split $Q(w)$ according to the disjuncts in (13).
First Disjunct in (13)

\[ Q^1(w) = \{ w' \mid (10), (11), w' = w.S^{-1}.b.y'.S \} \quad (14) \]

This is a simple NOL. (10) and (14) imply \( P(w) < N - 1 \), otherwise, \( w' = \bot \). From (11) and (14), we have \( \#f(y') - 1 - \#b(y') = -1 \Leftrightarrow \#f(y') = \#b(y') \). Let \( a \) be the number of \( f \)'s and \( b \)'s in \( y' \) (\( a = \#f(y') = \#b(y') \)). From (10) and (14):

\[
\#f(w) + 0 + a + \#b(w) + 1 + a < N \Leftrightarrow P(w) + 1 + 2a < N. \]

Thus:

\[
Q^1(w) = \begin{cases} 
  \text{if } P(w) < N - 1 & \text{then } w.S^{-1}.b.\frac{f^a.b^a}{P(w)+1+2a}S \\
  \text{else } \emptyset 
\end{cases} \quad (15)
\]

Let \( a = \max \left\{ x \mid x \in Z \land x < \frac{N - P(w) - 1}{2} \right\} \). Then:

\[
K^1(w) = \text{if } P(w) < N - 1 \text{ then } w.S^{-1}.b.f^a.b^a.S \text{ else } \bot \quad (16)
\]

Second Disjunct in (13) \( Q^2(w) = \{ w' \mid (10), (11), w' = x.b.y'.S, w = x.f.y'.S \} \). \( w = x.f.y'.S \) implies that \( w \) must contain at least one \( f \). Moreover, \( w' = w.S^{-1}.y'.f^{-1}.b.y'.S \).

Thus, we have to consider the set of all \( w.S^{-1}.uP*.f^{-1}. \):

\[
Q^2(w) = \begin{cases} 
  \text{if } w = b^*.S & \text{then } \emptyset \\
  \text{else } \left\{ \begin{array}{l}
  \text{if } \#f(w') + \#u(w') < N, \\
  \#f(w') - \#b(w') = \#f(w) - \#b(w) - 1, \\
  w' = w.S^{-1}.uP*.f^{-1}.b.SLABELS*S 
\end{array} \right. 
\end{cases}
\]
Lemma 2 implies that the last executed element in this set is the one with rightmost application of \( f^{-1} \). I.e., the last one has the following shape: \( w' = w.S^{-1}.b^{-k}.f^{-1}.b.\gamma'.S \), where \( k \geq 0 \) is the greater value such that \( w = \text{LABELS}^*.f.b^k.S \), and some word \( \gamma' \).
(To see this, notice that Lemma 2 implies that \( w.S^{-1}.f^{-1}.b^{-k}.f^{-1}.b.\text{LABELS}^*.S \ll w.S^{-1}.f^{-1}.b.\text{LABELS}^*.S \).) Let \( F, B \) be the number of \( f, b \) in \( \gamma' \). From (c): \( B - F = k - 1 \).

\[
K^2(w) =
\begin{cases}
  \text{if } w = b^*.S & \\
  \text{then } \bot & \\
  \text{if } w = \text{LABELS}^*.f.b^k.S, \ k \geq 0 & \\
  \text{then } w.S^{-1}.b^{-k}.f^{-1}.b.\frac{|f^r.b^p|}{B-F=k-1,B+F<N+k-P(w)}.S & \\
  \text{else } \bot & \\
  \text{else } w.S^{-1}.b^{-k}.f^{-1}.b.\frac{|f^r.b^p|}{B-F=k-1,B+F<N+k-P(w)}.S & \\
\end{cases}
\]

Actually, we can’t have \( w \neq b^*.S \land w \neq \text{LABELS}^*.f.b^k.S \), so the last leaf is impossible:

\[
K^2(w) =
\begin{cases}
  \text{if } w = b^*.S & \\
  \text{then } \bot & \\
  \text{else } w.S^{-1}.b^{-k}.f^{-1}.b.\frac{|f^r.b^p|}{B-F=k-1,B+F<N+k-P(w)}.S & \\
\end{cases}
\quad (17)
\]

Note that, in the second leaf, \( w \) has to have shape \( w = \text{LABELS}^*.f.b^k.S, \ k \geq 0 \). Moreover, this leaf actually is a set. Even though this may mean some lack of precision in the analysis, this set is guaranteed to include the maximum element of \( Q^2(w) \) according to the lexicographical order.

**Third Disjunct in (13)** \( Q^2(w) : \{ w' \mid (10), (11), w = w'.S^{-1}.f.y'.S \} \). Let \( F, B \) be the number of \( f, b \) in \( y \), respectively. From (11), we know that \( w \) cannot be equal to \( b^*.S \). (However, since this is a property local to the current disjunct of the execution order, \( w = b^*.S \) yields a soft bottom, not a hard one.) From \( w = w'.S^{-1}.f.y'.S \) and (11), \( \#f(w') - \#b(w') = \#f(w') + 1 + F - \#b(w') - B \Leftrightarrow B - F = 0 \). Thus \( w \) has to be of the form: \( \text{LABELS}^*.f.[f_i, b_i].S \). Hence:

\[
K^3(w) =
\begin{cases}
  \text{if } w = b^*.S & \\
  \text{then } \bot & \\
  \text{if } w = \text{LABELS}^*.f.[f_i, b_i].S & \\
  \text{then } w.S^{-1}.[f_i, b_i]^{-1}.f^{-1}.S & \\
  \text{else } \bot & \\
\end{cases}
\quad (18)
\]

where \( i \) is a non-negative integer (\( = B = F \)). Lemma 2 implies that \( i > j \Rightarrow w.S^{-1}.[f^{-i}, b^{-i}].f^{-1}.S \ll w.S^{-1}.[f^{-j}, b^{-j}].f^{-1}.S \), which in turns implies that the maximum of \( w.S^{-1}.[f_i, b_i].f^{-1}.S \) is given by the smallest possible \( i \), depending on the actual value of \( w \).

**Combining Intermediate Results** Combining \( K^3(w) \) and \( K^2(w) \) (\( K^3(w) \) is plugged in at \( K^2(w) \)'s leaves):

\[
S_1(w) =
\begin{cases}
  \text{if } w = b^*.S & \\
  \text{then } \bot & \\
  \text{if } w = \text{LABELS}^*.f.[f_i, b_i].S & \\
  \text{then } (a) & \\
  \text{else } \bot & \\
  \text{else } w.S^{-1}.b^{-k}.f^{-1}.b.\frac{|f^r.b^p|}{B-F=k-1,B+F<N+k-P(w)}.S & \\
\end{cases}
\]
where:

\[
(a) = \max \left( \frac{w.S^{-1}.f^{-1}.b^{-1}.S,}{w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^p b^p]}{B-F=k-1,B+F<N+k-P(w)} .S} \right)
\]

- if \( i = k = 0 \)
  - then \( w.S^{-1}.[f^i, b^i]^{-1}.f^{-1}.S \)
  - else \( w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^p b^p]}{B-F=k-1,B+F<N+k-P(w)} .S \)

\[
S_1(w) = \begin{cases} 
\text{if } w = b^*.S \\
\quad \text{then } \bot \\
\text{else } w.S^{-1}.f^{-1}.S \\
\quad \text{if } w = \text{LABELS}^*.f.S \\
\quad \quad \text{then } w.S^{-1}.f^{-1}.S \\
\quad \quad \text{else } w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^p b^p]}{B-F=k-1,B+F<N+k-P(w)} .S 
\end{cases}
\]

Plugging \( K^3(w) \) at \( S_1 \)'s leaves:

\[
S_2(w) = \begin{cases} 
\text{if } w = b^*.S \\
\quad \text{if } P(w) < N - 1 \\
\quad \quad \text{then } w.S^{-1}.b.f^a.b^a.S \\
\quad \quad \text{else } \bot \\
\quad \quad \text{if } P(w) < N - 1 \\
\quad \quad \quad \text{then } \max \left( w.S^{-1}.f^{-1}.S, w.S^{-1}.b.f^a.b^a.S \right) \\
\quad \quad \quad \text{else } w.S^{-1}.f^{-1}.S \\
\quad \quad \text{else } w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^p b^p]}{B-F=k-1,B+F<N+k-P(w)} .S \\
\quad \quad \quad \text{if } P(w) < N - 1 \\
\quad \quad \quad \quad \text{then } \max \left( w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^p b^p]}{B-F=k-1,B+F<N+k-P(w)} .S, w.S^{-1}.b.f^a.b^a.S \right) \\
\quad \quad \quad \text{else } w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^p b^p]}{B-F=k-1,B+F<N+k-P(w)} .S 
\end{cases}
\]

Moreover, \( w = y.f.s \Rightarrow w' = w.S^{-1}.f^{-1}.S \Rightarrow w' = w.S^{-1}.b.\text{LABELS}^*.S \), and \( w = y.f.b^k.S \Rightarrow w.S^{-1}.b^{-k}.f^{-1}.b.\text{LABELS}^*.S \Rightarrow w.S^{-1}.b.\text{LABELS}^*.S \), thus the two maxima should be replaced by their second arguments. Since we can then notice that all the leaves are equal when \( P(w) < N - 1 \), we may factor the latter conditional out (just to make the result more readable to a human eye).

\[
S_2(w) = \begin{cases} 
\text{if } P(w) < N - 1 \\
\quad \text{then } w.S^{-1}.b.f^a.b^a.S \\
\quad \quad \text{if } w = b^*.S \\
\quad \quad \quad \text{then } \bot \\
\quad \quad \quad \text{if } w = \text{LABELS}^*.f.S \\
\quad \quad \quad \quad \text{then } w.S^{-1}.f^{-1}.S \\
\quad \quad \quad \quad \text{else } w.S^{-1}.b^{-k}.f^{-1}.b.\frac{[f^p b^p]}{B-F=k-1,B+F<N+k-P(w)} .S 
\end{cases}
\]

Notice also that, since all the intermediate results have been combined, remaining soft bottoms became hard ones, meaning that the analysis is now able to guarantee that there is no source under the corresponding conditions.
4.3 Comments on the Examples. Applications

The reader may check that the closed form (19) expresses the flows of data tabulated in Fig 4.2. For instance, \( w = b.b.b.S \) corresponds to the second leaf (\( \perp \)). \( w = f.b.b.S \Rightarrow k = 2 \) in the last leaf, so \( w' = \max(b, \frac{f^c}{B-1, F^c < 3, S}) = b.b.S \). All “inner” nodes correspond to the first leaf: for instance, \( w = S \Rightarrow a = 1 \Rightarrow w' = b.f.b.S \).

Application of our data flow analysis to program checking is clear: in the first example, undefined values are read during the first instance of the outer explore construct.

Applications to compiling and automatic parallelization are best illustrated by the first example. Due to the regularity of the data flow, we can apply the method in [5] to automatically find a scheduling function \( \theta(i, j) \) such that \( \theta(i, j) > \theta(i - 1, j) \). A possible solution is \( \theta(i, j) = i \), i.e., the program can be “wavefronted” along the right branches of the tree. More formally, all instances \( (S, i, j) \), for a given \( i \) and all \( j \), can be executed simultaneously.

In turn, we can apply the methods in [6] to detect that only a finite subset of the data structure tree (declared to be infinite) has to be allocated at any time: exactly 2 branches of \( q \) elements of tree need to be allocated, and can be reused in turn. In a sense, we thus have new means to compute the extent of the data structure [7].

5 Conclusion

We presented a data flow analysis for a restricted class of recursive programs over recursive data structures. This restricted class include recursive exploration of commutative (e.g. arrays) and non commutative (e.g. trees) recursive data structures. These restrictions are in part due to the intrinsic difficulty of the subject, and in part to the way our framework is restricted to data flow problems that can be expressed as integer linear programming ones, enforcing for instance that recursion predicates should be conjunctions of affine (in)equalities.

Rinard and Diniz [8] proposed an analysis that detects commuting (so, parallel) computations in C++ programs that manipulate recursive data structures. Even though they handle more general structures than we do, their analysis is local (i.e., does not take global properties of the structure into account) and is based on semantical properties of a limited number of operators.

There are obviously numerous deficiencies in our analysis: our inability to directly handle programs written in usual Pascal-like imperative languages (the main problems being the extraction of the existence predicate and the detection of induction variables that access data structures); the approximation due to NOLs in the case of non commutative structures; the lack of general algorithms to compute lexicographical maxima of NOLs with pumping inverses; and, of course, our need for more of practical experience. All these issues will be addressed in future work. In addition, we intend to study cases when the existence of operations cannot be predicted at compile-time, i.e., to extend our framework along the lines of [9].

Acknowledgments The first two authors are partly supported by the CNRS. The second, in addition, by a Training and Mobility of Researchers Grant from the European Union. The third author is partly supported by the DFG project RecuR.
last two authors are, in addition, supported by a German-French Procope exchange programme.

We would also like to thank Karl-Filip Faxén, Paul Feautrier, Chris Lengauer and Björn Lisper for helpful comments on this material.

References

New Language Constructs
HPF+: New Language and Implementation Mechanisms for the Support of Advanced Irregular Applications

Barbara Chapman, Piyush Mehrotra, and Hans Zima

Institute for Software Technology and Parallel Systems,
University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria
E-Mail: {barbara,zima}@par.univie.ac.at

ICASE, MS 132C, NASA Langley Research Center,
Hampton VA. 23681 USA
E-Mail: pm@icase.edu

Abstract

In the past, most high performance applications dealt with numerical simulation, and parallel machines were essentially used only by a relatively small group of dedicated professionals which had to cope with the idiosyncrasies of the machine at a low level of abstraction. This situation is changing quickly, driven by enhanced hardware and software support offered by a new generation of parallel computers, and the rapid expansion of global networks that lead to the feasibility of applications distributed over geographically widely distant areas.

In this paper, we discuss the language support required for the efficient handling of advanced applications. We will outline the major features of Vienna Fortran and compare the language to High Performance Fortran (HPF), a de-facto standard in this area. A significant weakness of HPF is its lack of support for many advanced applications, which require irregular data distributions and dynamic load balancing. We introduce HPF+, an extension of HPF based on Vienna Fortran, that provides the required functionality.

1 Introduction

The continued demand for increased computing power has led to the development of highly parallel scalable multiprocessing systems (HMPs), which are now offered by all major vendors and have rapidly gained user acceptance. These machines are relatively inexpensive to build, and are potentially scalable to large numbers of processors. However, they are difficult to program: most of the architectures exhibit non-uniformity of memory access which implies that the locality of algorithms must be exploited in order to achieve high performance, and the management of data becomes of paramount importance.

Traditionally, HMPs have been programmed using a standard sequential programming language (Fortran or C), augmented with message passing constructs. In this paradigm, the user is forced to deal with all aspects of the distribution of data and work to the processors, and to control the program's execution by explicitly inserting message passing operations. The resulting programming style can be compared to assembly language programming for a sequential machine; it has led to slow software development cycles and high costs for software production. Moreover, although MPI is evolving as a standard for message passing, the portability of MPI-based programs is limited.

*The work described in this paper was partially supported by the Austrian Research Foundation (FWF) and by the Austrian Ministry for Science and Research (BMWF). This research was also supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605, while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681.
since the characteristics of the target architectures may require extensive restructuring of the code.

As a consequence, much research and development activity has been concentrated in recent years on providing higher-level programming paradigms for HMPs. Vienna Fortran, building upon the KALI programming language [8] and experiences from the SUPERB parallelization system [12], was the first fully specified data-parallel language for HMPs. It provides language features for the high-level specification of data distribution and alignment, as well as explicitly parallel loops. High Performance Fortran (HPF) [7], a de-facto standard developed by a consortium including participants from industry, academia, and research laboratories, is based on concepts of CM Fortran [11], Vienna Fortran, and Fortran D [6]. It provides support for regular applications, alleviating the task of the programmer for a certain segment of applications. However, it is generally agreed that the current version of the language, HPF-1, is not adequate to handle many advanced applications, such as multiblock codes, unstructured meshes, adaptive grid codes, or sparse matrix computations, without incurring significant overheads with respect to memory or execution time. This fact has been acknowledged by the HPF Forum in its decision to start the development of HPF-2 at the beginning of 1995.

In this paper, we outline the major features of Vienna Fortran and compare the language to HPF-1. We then identify some of the weaknesses of HPF-1 by considering language requirements posed by irregular algorithms and dynamic load balancing. This study leads to the description of an HPF extension, "HPF+", which, based upon Vienna Fortran, solves many of these problems by introducing proper extensions, and thus contributes to the present effort of the HPF Forum for defining a suitable successor to HPF-1.

2 Vienna Fortran: A Short Overview

Vienna Fortran is based on the SPMD (Single Program Multiple Data) or data parallel model of computation. With this method, the data arrays in the original program are each partitioned and mapped to the processors. This is known as distributing the arrays. The specification of the mapping of the elements of the arrays to the set of processors is called the data distribution of that program. A processor is then thought of as owning the data assigned to it; these data elements are stored in its local memory. Now the work is distributed, in general according to the data distribution: computations which define the data elements owned by a processor are performed by it - this is known as the owner computes paradigm. The processors then execute essentially the same code in parallel, each on the data stored locally.

The compiler analyzes the source code, translating global data references into local and non-local data references based on the distributions specified by the user. The non-local references are satisfied by inserting appropriate message-passing statements in the generated code. Finally, the communication is optimized where possible, in particular by combining messages and by sending data at the earliest possible point in time.

A major characteristic of this style of programming is that the performance of the resulting code depends to a very large extent on the data distribution selected by the programmer. The data distribution determines not only where computation will take place, it is also the main factor in deciding what communication is necessary. If, in a given scope, the association between an array and a distribution is invariant, we speak of a statically, otherwise of a dynamically distributed array.

The Vienna Fortran language extensions provide the user with the following features:

- The processors which execute the program may be explicitly specified and referred to. It is possible to impose one or more structures, or views, upon them.
• The distributions of arrays can be specified using annotations. These annotations may use processor references related to processor structures introduced by the user.
  
  – Intrinsic functions are provided to specify the most common distributions.
  – Distributions may be defined indirectly via a map array.
  – Data may be replicated to all or a subset of processors.
  – The user may define new distribution functions.

• An array may be aligned with another array, providing an implicit distribution. Alignment functions may also be defined by the user.

• The distribution of arrays may be changed dynamically. However, a clear distinction is made between arrays which are statically distributed and those whose distribution may be changed at runtime.

• In procedures, formal array parameters may
  
  – inherit the distribution of the actual argument, or
  – be explicitly distributed, possibly causing some data motion.

• A forall loop permits explicitly parallel loops to be written. Intrinsic reduction operations are provided, and others may be defined by the user. Loop iterations may be executed
  
  – on a specified processor,
  – where a particular data object is stored, or
  – as determined by the compiler.

• Arrays in common blocks may be distributed.

• Allocatable arrays may be used in much the same way as in Fortran 90. Array sections are permitted as actual arguments to procedures.

Vienna Fortran does not introduce a large number of new constructs, but those it does have are supplemented by a number of options and intrinsic functions, each of which serves a specific purpose. They enable the user to exert additional control over the manner in which data is mapped or moved, or the code is executed. A full specification of the language is given in [13]; its use for solving a range of typical application problems is described in [1].

3 A Comparison of Vienna Fortran and HPF

The main concepts in HPF have been derived from a number of predecessor languages, including mainly CM Fortran [11], Kali [8], Fortran D [6], and Vienna Fortran, with the last two languages having the largest impact.

The basic elements of HPF's language model are – similarly to Vienna Fortran – abstract processors, distributions, and alignments. In addition, HPF has introduced the concept of a template, which is essentially a named index domain that can be used as an alignment base. The implications of this construct which significantly complicates the underlying semantic model are discussed in [2].

HPF follows Vienna Fortran closely in a number of features. This includes in particular

• abstract processor arrays

• direct distribution and alignment of arrays
• distinction between static and dynamic distributions
• definition of the procedure interface, in particular inherited and enforced distributions
• FORALL loops (called INDEPENDENT loops in HPF)

On the other hand, a number of advanced concepts of Vienna Fortran have not been included in HPF. Among them are

• different processor views
• distribution of arrays to processor sections
• GENERAL.BLOCK distributions
• INDIRECT distributions
• user-defined distribution functions

These omissions, in particular the absence of language features for the formulation of more general distribution functions, significantly impairs the applicability of HPF to advanced algorithms using, for example, irregular or adaptive grids.

4 HPF+

The above discussion has indicated that the use of HPF-1 for advanced applications leads to problems related to expressivity and performance. In this section we discuss extensions to HPF-1 that are needed to address these problems. The discussion informally introduces an HPF extension, called "HPF+", using an ad-hoc HPF-like syntax. The extensions to HPF-1 incorporated into HPF+ are of two sorts: first, a generalization of the data distribution mechanisms, and, secondly, control facilities providing coarse-grain task parallelism integrated with the data-parallel HPF computation model. They will be discussed in individual subsections below.

4.1 Distribution to Processor Subsets and Subobject Distribution

The HPF-1 DISTRIBUTE directive specifies the distribution of data to a processor array which has been declared by the user. It does not permit distribution to a part of the processor array. Also, HPF-1 allows only the distribution of top-level objects – components of a derived type cannot be distributed. Here, we show that multiblock problems need both features, and describe simple extensions to HPF-1 which provide these functionalities. This will be illustrated by a program skeleton creating a corresponding grid structure.

Scientific and engineering codes from diverse application areas may use multiple grids to model the underlying problem domain. These grids may be structured, unstructured or a mixture of both types, individually chosen to match the underlying physical structure and allocate the computational resources efficiently with high node densities in selected areas. A typical application may use anywhere from 10 to 100 grids of widely varying sizes and shapes. Each sweep over the domain involves computation on the individual grids before data is exchanged between them. Thus, these types of applications exhibit at least two levels of parallelism. At the outer level, there is coarse grain parallelism, since the computation can be performed on each grid simultaneously. The internal computation on each grid, on the other hand, exhibits the typical loosely synchronous data parallelism of structured grid codes.

Distributing the array of grids to the processors so that each grid is mapped to exactly one processor offers limited parallelism since it only exploits the outer level, the number of grids may be too small to allow the use of all processors, and the grids may vary significantly in size resulting
in an uneven workload. Another strategy is to distribute each grid independently to all processors, enabling the parallelism within a grid to be exploited. This will lead to a more even workload; however, the grids may not all be large enough for this to be a reasonable solution.

Both of the above distribution strategies are likely to be inefficient, particularly on machines with a large number of processors. A flexible alternative is to permit grids to be separately distributed to a suitably sized subset of the available processors. This approach allows both levels of parallelism to be exploited while providing the opportunity to balance the workload.

HPF-1 does not, however, permit data arrays to be distributed directly to subsets of processors. In HPF-1 this can be expressed indirectly by using templates and alignment, but these solutions are difficult to achieve and will generally require a priori precise knowledge of the size of both the grid and the processor array, and must be reimplemented for each modification of the problem. A simpler solution is to adopt a direct approach, which permits a processor subsection to be the target of a distribution. An example of this is shown in Figure 1.

Consider the case of a multiblock problem where the number of grids and their sizes are not known until runtime. Each grid can be declared as a pointer within a derived type and the set of all grids can be an allocatable array, where each element is a grid. HPF-1 allows us to distribute the array of grids to the processors. However, we may not distribute the individual grids across processors, since these are subobjects and their distribution is explicitly prohibited. This is necessary for exploiting the parallelism present within the individual grids. The algorithm in Figure 1 illustrates a solution for this problem, by providing a notation for the distribution of subobjects. We assume that at most one level in a nested structure can be distributed in this way.

4.2 General Block Distributions

Dimensions of data arrays or templates can be mapped in HPF-1 by specifying either block or cyclic distributions. There are a number of problems for which these regular mappings do not result in
an adequate balance of the workload across the processors of the target machine, but which can be handled by general block distributions, a relatively simple extension of HPF-1's regular block distributions.

General block distributions were initially implemented in SUPERB and Vienna Fortran. They are similar to the regular block distributions of HPF-1 in that the index domain of an array dimension is partitioned into contiguous blocks which are mapped to the processors; however, the blocks are not required to be of the same size. Thus, general block distributions provide more generality than regular blocks while retaining the contiguity property, which plays an important role in achieving target code efficiency.

Consider a one-dimensional array $A$, declared as REAL $A[l : u]$, and assume that there are $N$ processors $p_i, 1 \leq i \leq N$. If we distribute $A$ using a general block distribution $\text{GENERAL.BLOCK}(B)$, where $B$ is a one-dimensional integer array with $N$ elements, and $B(i) = s_i$ (with $s_i > 0$ for all $i$) denotes the size of the $i$-th block, then processor $p_i$ owns the local segment $A[l + s_1 - 1]$, $p_2$ owns $A[l + s_1 + s_2 - 1]$ and so on. $B$, together with the index domain of $A$, completely determines the distribution of $A$ and provides all the information required to handle accesses to $A$, including the organization of the required communication. The above scheme can be readily generalized to multi-dimensional arrays, each dimension of which is distributed by regular or general block.

The following code fragment illustrates an array $A$, whose rows are distributed in blocks of sizes 400, 400, 200, 100, 100, 100, 500, and 800.

```
HPF PROCESSORS R(8)
INTEGER :: B(8) = (/400, 400, 200, 100, 100, 100, 500, 800/)
REAL A(2600, 100)
HPF+R$ DISTRIBUTES (GENERAL.BLOCK(B),*)::A
```

Although the representation of general block distribution requires on the order of the number of processors to describe the entire distribution, optimization often permits a local description of the distribution to be limited to just a few processors, with which there will be communication. Also, the space overhead due to this representation is not large in general, since most problems do not require a large number of distinct general block distributions.

Arrays distributed in this way can also be efficiently managed at runtime, allowing the use of the overlap [15] concept to optimize communication related to regular accesses. Finally, codes can be easily parameterized with such distributions: for example, a procedure with a transcriptive formal argument\(^1\) that is supplied with differently distributed actual arguments can be efficiently compiled if the representation of the argument's distribution is passed along as a set of additional implicit arguments created by the compiler.

4.3 Irregular Distributions

General block distributions provide enough flexibility to meet the demands of some irregular computations: if, for instance, the nodes of a simple unstructured mesh are partitioned prior to execution and then appropriately renumbered, the resulting distribution can be described in this manner. This renumbering process is similar to domain decomposition and can be a complex and computationally demanding task. However, this approach is not appropriate for all irregular problems. A general block distribution, even with two or three dimensions, may not be able to provide an equal workload per processor. Also, block distributions are always constrained by the adjacency of data. The single workspaces typically used in Fortran programs cannot necessarily be renumbered in such a way as to produce a sequential group of regions. Irregular distributions offer the ability

\(^1\)If such an argument is passed by reference, the distribution is left intact, and thus no movement of data will be necessary.
to express totally unstructured or irregular data structures but at some cost in terms of the code
the compiler must generate.

We will here introduce two different mechanisms to handle general data distributions. We begin
with indirect distribution functions, which allow the specification of a distribution via a mapping
array and continue with user-defined distribution functions.

4.3.1 Indirect Distributions

Indirect distribution functions can express any distribution of an array dimension that does
not involve replication. Consider the following program fragment in HPF+:

```
!HPF$ PROCESSORS R(M)
REAL A(N)
INTEGER MAP(N)
...
!HPF$ DYNAMIC, DISTRIBUTED BLOCK::A
!HPF$ DISTRIBUTED (BLOCK)::MAP
...
| Compute a new distribution for A and save it in the mapping array MAP: the j-th element
| of A is mapped to the processor whose number is stored in MAP(j)
CALL PARTITIONER (MAP, A,...)
| Redistribute A as specified by MAP.
!HPF+& REDISTRIBUTE A (INDIRECT (MAP))
```

Array A is dynamic and initially distributed by block. MAP is a statically distributed integer
array that is of the same size as A and used as a mapping array for A; we specify a reference to
an indirect distribution function in the form INDIRECT(MAP). When the reference is evaluated,
all elements of MAP must be defined and represent valid indices for the one-dimensional processor
array R, i.e., they must be numbers in the range between 1 and M. A is then distributed such
that for each j, 1 ≤ j ≤ N, A(j) is mapped to R(MAP(j)). In this example, MAP is defined by a
partitioner, which will compute a new distribution for A and assign values to the elements of MAP
accordingly. (This distribution will often be used for a number of arrays in the program).

The example in Figure 2 illustrates the use of indirect distributions in the context of a sweep
over an unstructured mesh.

Indirectly distributed arrays must be supported by a runtime system which manages the internal
representation of the mapping array and handles accesses to the indirectly distributed array. The
mapping array is used to construct a translation table, recording the owner of each datum and its
local index. Note that this representation has O(N) elements, on the same order as the size of
the array; however, most codes require only a very small number of distinct indirect mappings.
The PARTI routines developed by J. Saltz and collaborators [9] represent a runtime library which
directly supports indirect distribution functions, in connection with irregular array accesses.

4.3.2 User-Defined Distribution Functions

Indirect distribution functions incur a considerable overhead both at compile time and at runtime.
A difficulty with this approach is that when a distribution is described by means of a mapping array,
any regularity or structure that may have existed in the distribution is lost. Thus the compiler
cannot optimize the code based on this complex but possibly regular distribution. User-defined
distribution functions (UDDFs) provide a facility for extending the set of intrinsic mappings
defined in the language in a structured way. The specification of a UDDF establishes a mapping
from (data) arrays to processor arrays, using a syntax similar to Fortran functions. UDDFs have
two implicit formal arguments, representing the data array to be distributed and the processor
array to which the distribution is targeted. Specification statements for these arguments can be
given using the keywords TARGET-ARRAY and PROCESSOR-ARRAY, respectively. UDDFs may contain local data structures and executable statements along with at least one distribution mapping statement which maps the elements of the target array to the processors.

UDDFs constitute the most general mechanism for specifying distributions: any arbitrary mapping between array indices and processors can be expressed, including partial or total replication. We illustrate their use by an example, representing indirect distributions. For simplicity we assume here that A and MAP have the same shape.

```hpfp`

```

4.4 Extensions of the INDEPENDENT Loop Concept

Whenever a do loop contains an assignment to an array involving an indirect access, the compiler will not be able to determine whether the iterations of the loop may be executed in parallel. Since such loops are common in irregular problems, and may contain the bulk of the computation, the user must assert the independence of its iterations.

For this purpose, HPF-1 provides the INDEPENDENT directive, which asserts that a subsequent do loop does not contain any loop-carried dependences, allowing the loop iterations to be executed in parallel. A NEW clause introduces private variables that are conceptually local in each iteration, and therefore cannot cause loop-carried dependences.

There are two problems with this feature:

- There is no language support to specify the work distribution for the loop, i.e., the mapping of iterations to processors. This decision is left to the compiler/runtime system.

- Reductions, which perform global operations across a set of iterations, and assign the result to a scalar variable, violate the restriction on dependences and cannot be used in the loop. ²

The first problem can be solved by extending the INDEPENDENT directive with an ON clause that specifies the mapping, either by naming a processor explicitly or referring to the owner of an element. For example, in Figure 2 INDEPENDENT is used for the I loop over edges so that the loop is executed on the processor that owns the array element EDGE(I,1).

The second problem can be solved by extending the language with a REDUCTION directive – which is to be permitted within independent loops – and imposing suitable constraints on the statement which immediately follows it. It could be augmented by a directive specifying the order in which values are to be accumulated. Note that simple reductions could be detected by most compilers.

In Figure 2, many proposed features of HPF+ are illustrated for a simple unstructured mesh code. The mesh for this code consists of triangles; values for the flow variables are stored at their vertices. The computation is implemented as a loop over the edges: the contribution of each edge is subtracted from the value at one node and added to the value at the other node. The mesh is represented by the array EDGE, where EDGE(I,1) and EDGE(I,2) are the node numbers at

²Note however that HPF-1 and Fortran 90 provide intrinsics for some important reductions.
PARAMETER (NNODE = ..., NEDGE = ...)  
!HPF$ PROCESSORS R(M)  
...  
REAL X(NNODE),Y(NNODE),EDGE(NEDGE,2)  
INTEGER MAP(NNODE)  
...  
!HPF$ DYNAMIC :: X,Y,EDGE  
!HPF$ DISTRIBUTE (BLOCK) :: X, MAP  
!HPF$ ALIGN WITH X :: Y  
!HPF$ DISTRIBUTE (BLOCK,*) :: EDGE  
...  
CALL PARTITIONER(MAP,EDGE)  
...  
!HPF+$ REDISTRIBUTE X(INDIRECT(MAP))  
!HPF+$ REDISTRIBUTE EDGE(1,:) ONTO R(MAP(EDGE(1,1)))  
...  
!HPF+$ INDEPENDENT, ON OWNER (EDGE(1,1)), NEW (N1, N2, DELTAX)  

DO I = 1, NEDGE  
...  
   N1 = EDGE(I,1)  
   N2 = EDGE(I,2)  
...  
   DELTAX = F(X(N1), X(N2))  

!HPF+$ REDUCTION  
   Y(N1) = Y(N1) - DELTAX  
!HPF+$ REDUCTION  
   Y(N2) = Y(N2) + DELTAX  
...  
END DO  
...  
END

Figure 2: Code for Unstructured Mesh in HPF+
the two ends of the $I$-th edge. The arrays $X$ and $Y$ represent the flow variables, which associate a value with each of the $NNODE$ nodes.

Consider the distribution of the data across the one-dimensional array of processors, $R(M)$. The array $X$ is declared to be dynamically distributed with an initial block distribution. At runtime this array is distributed indirectly, as defined in the mapping array $MAP$ obtained from the user-specified routine $PARTITIONER$. $Y$ is also declared with the keyword $DYNAMIC$ and is aligned to $X$. Whenever $X$ is redistributed, $Y$ is automatically redistributed with exactly the same distribution function.

Since the elements of $EDGE$ are pointers to flow variables – in iteration $I$, $X(EDGE(I,1))$, $X(EDGE(I,2))$ and the corresponding components of $X$ and $Y$ are accessed – we relate the distribution of $EDGE$ to the distribution of $X$ and $Y$ in such a way that $EDGE(I,1)$ is mapped to the same processor as $X(EDGE(I,1))$.

This kind of relationship between data structures occurs in many codes, since a mesh is frequently described in terms of elements, and values are likely to be accumulated at the vertices. It can be simply expressed if we extend the $REDISTRIBUTE$ directive as shown in the example.

The computation is specified using an extended $INDEPENDENT$ loop. The work distribution is specified by the $ON$ clause: the $I$-th iteration is to be performed on the processor that owns $EDGE(I,1)$. The variables $N1$, $N2$ and $DELTAX$ are private, so, conceptually, each iteration is allocated a private copy of each of them. Hence assignments to these variables do not cause loop-carried dependences [14].

For each edge, the $X$ values at the two incident nodes are read and used to compute the contribution $DELTAX$ for the edge. This contribution is then accumulated into the values of $Y$ for the two nodes. But since multiple iterations will accumulate $Y$ values at each node, different iterations may write to the same array elements. As a consequence, we have indicated that these are reductions.

The dominating characteristic of this code, from the point of view of compilation, is that the values of $X$ and $Y$ are accessed via the edges, hence a level of indirection is involved. In such situations, either the mesh partition must be available to and exploitable by the compiler, or runtime techniques such as those developed in the framework of the $inspector-executor$ paradigm [9] are needed to generate and exploit the communication pattern.

4.5 Data Distribution and Alignment – Other Issues

There are a number of other issues with the specification of data distribution and alignment in HPF-1 which we have not discussed here. These include processor views, control of dynamic data distributions, library interfaces, and the procedure boundary; they are discussed in [4].

4.6 Integration of Task With Data Parallelism

With the rapidly growing computing power of parallel architectures, the complexity of simulations developed by scientists and engineers is increasing fast. Many advanced applications are of a multidisciplinary and heterogeneous nature and thus do not fit into the data-parallel paradigm.

Multidisciplinary programs are formed by pasting together modules from a variety of related scientific disciplines. For example, the design of a modern aircraft involves a variety of interacting disciplines such as aerodynamics, structural analysis and design, propulsion, and control. These disciplines, each of which is initially represented by a separate program, must be interconnected to form a single multidisciplinary model subsuming the original models and their interactions. For task parallelism to be useful, the parallelism both within and between the discipline models needs to be exposed and effectively exploited.

In this section, we propose language features that address this issue. These extensions provide a software layer on top of data-parallel languages, designed to address the "programming in the large"
issues as well as the parallel performance issues arising in complex multidisciplinary applications. A program executes as a system of tasks which interact by sharing access to a set of Shared Data Abstractions (SDAs). SDAs generalize Fortran 90 modules by including features from object-oriented data bases and monitors in shared-memory languages. They can be used to create persistent shared "objects" for communication and synchronization between coarse-grained parallel tasks, at a much higher level than simple communication channels transferring bytes between tasks.

A task is spawned by activating a subroutine with a list of arguments all of which must be of intent IN. Tasks are asynchronously executing autonomous activities to which resources of the system may be allocated. For example, the physical machine on which a task is to be executed, along with additional requirements pertaining to this machine, may be specified at the time a task is created.

Tasks may embody nested parallelism, for example by executing a data-parallel HPF program, or by coordinating a set of threads performing different functions on a shared data set.

An SDA consists of a set of data structures along with the methods (procedures) which manipulate this data. A set of tasks may share data by creating an SDA instance of appropriate type, and making it accessible to all tasks in the set. Tasks may then asynchronously call the methods of the SDA, with each call providing exclusive access. Condition clauses associated with methods and synchronization facilities embodied in the methods allow the formulation of a range of coordination strategies for tasks. The state of an SDA can be saved on external storage for later reuse. This facility can be seen as providing an I/O capability for SDAs, where in contrast to conventional byte-oriented I/O the structure of the object is preserved.

Other Fortran-based approaches to the problem of combining task with data parallelism include the programming languages Fortran M [5], which provides a message-passing facility in the context of a discipline enforcing determinism, and Fx [10], which allows the creation of parallel tasks that can communicate at the time of task creation and task termination by sharing arguments. These approaches address a small grain of parallelism.

5 Conclusion

In this paper, we have outlined the features of Vienna Fortran, and compared the language to HPF-1. After an analysis of weaknesses of HPF-1 in the context of advanced algorithms, we proposed an extension, HPF+, which is based on Vienna Fortran and addresses these problems. HPF+ can be seen as a contribution to the work of the HPF Forum towards the standardization of parallel languages for scalable High Performance architectures.

References


On the Machine-independent Target Language for Parallelizing Compilers *

Michael J. Voss        Insung Park        Rudolf Eigenmann

Purdue University
School of Electrical and Computer Engineering

Abstract

Although shared memory machines provide one of the easier models for parallel programming, the lack of standardization for expressing parallelism on these machines makes it difficult to write efficient portable code. The Guide™ Programming System is one solution to this problem. In this paper, we discuss a back-end to the Polaris parallelizing compiler that generates Guide™ directives. We then compare the performance of parallel programs expressed in this way to programs automatically parallelized by a machine’s native compiler, and by code expressing parallelism with native directives. The resulting performance is presented and the feasibility of this directive set as a portable parallel language is discussed.

1 Introduction

Shared memory machines provide one of the easier conceptual models for parallel programming. Although this makes programming these machines relatively straightforward, there has been a lack of standardization of expressing parallelism. In moving from one machine of this type to another, it is often necessary to learn a new set of vendor-specific parallel language constructs in order to program on the new architecture.

One way to address this problem is to use a parallelizing compiler, which would free the user from the need to invest time in learning architecture-specific parallel languages. The burden, then, is on compilers which face the challenge of translating into a vendor-specific target language (this can be either an explicit parallel language or an internal representation). However, this creates the need for a unique set of language constructs to be generated for each machine that a code may be run on. Creating such architecture-specific back-ends to a compiler adds complexity and costs time resources. Resources, that may otherwise be available for developing techniques that apply to a range of architectures.

In this paper we begin to explore one possible solution to these problems. Kuck and Associates (KAI) of Champaign, Illinois has developed a parallel programming model for shared memory machines, based on the results of two previous working groups, the Parallel Computing Forum, and the ANSI subcommittee, X3H5. They have implemented their model with the Guide™ parallel language and its translator. The Fortran version of Guide, which is used in this study, is a Fortran 77 (F77) plus parallel directive language. Its associated translator converts code expressed in this dialect into F77 plus vendor-specific library calls, which are available for all of the popular SMP architectures in use today. This frees programmers from machine-specific syntax, allowing portable code to be written by users, and generated by parallelizing compilers.

The Guide language can be used as both an explicit parallel user language and a target language of a parallelizing source-to-source translator. Since our main interest lies in automatic parallelization,

---

*This work was supported in part by U. S. Army contract #DABT63-93-C-0033. This work is not necessarily representative of the positions or policies of the U. S. Army or the Government.
we chose this as our means of obtaining experimental codes. In order to automatically generate Guide
constructs, we added a back-end (referred to as Guide back-end henceforth) to the Polaris parallelizing
compiler. Polaris is a restructuring compiler originally developed at the University of Illinois at
Urbana-Champaign, that uses many advanced techniques to recognize, enhance, and exploit loop-level
parallelism. An overview of the Polaris compiler can be found in [BEH+94].

To determine the efficiency of portable code expressed in the Guide parallel language, two shared-
memory machines were then chosen for experimentation. These machines are a four-processor Sun
SPARC workstation, and a four-processor SGI Challenge machine. Most of our experimentation was
done on the SPARC machine, while the SGI Challenge is used as a demonstration of portability. The
performance of code expressed in the portable Guide language is compared with that automatically
parallelized by the Sun SPARC20 native compiler, and with that of the same code expressed in the
native directive language.

Section 2 will give some details of the history and syntax of the Guide programming language.
Section 3 gives an overview of the Polaris compiler and describes the modifications that we have made
so that it generates optimized code for Guide. Next, in Section 4, we briefly outline the architectures
of the machines on which our tests were performed. Following that, Section 5 presents measurements of
parallel programs run on our machines and a discussion of their performance. Finally, some of the issues
in generating portable code and the conclusions of the paper are presented in Section 6.

2 The Search for a Portable Language for Shared-Address-
Space Machines

2.1 A Brief History of Portable Parallel Languages

The quest for portable parallel programming languages is an old one. In this paper we concentrate
on a reduced question: is there a portable parallel language for shared-address-space machines? The
primary machine class we are considering in our project are even shared-memory machines, or SMPs.
SMPs, although one of the oldest classes of parallel machines, have received only little attention over
the past few years - primarily due to broad interest in massively parallel architectures. Only recently,
researchers have re-focused their attention on SMPs, perhaps because the programming model promises
to be simpler, or perhaps because the performance scalability of alternatives is not as good as was hoped.
Because of this renewed interest, we revisit a topic that had also been in the background for some time:
the parallel language effort started by the Parallel Computing Forum.

In 1987 the Parallel Computing Forum (PCF), an informal industry group, was formed with the goal
of standardizing DO loop parallelism in Fortran 77. This group was active from April 1987 through
February 1989, with their final report being published in 1989. Although PCF was dissolved, an ANSI
authorized subcommittee, X3H5, was chartered for building a language independent model for parallelism
in high level programming languages, as well as the bindings of this model into Fortran and C. This
subcommittee used the PCF final report as a foundation for their model. The group began their work
in February of 1990; however by May of 1994 interest was lost, and the proposed standards abandoned.
Nevertheless, their results yielded a completed language-independent model, an almost complete binding
for Fortran 90, and a preliminary draft for ANSI C. [KA]

Kuck and Associates (KAI) continued work on the partially completed documents of the ANSI
X3H5 committee, and expanded them when required. This resulted in the Guide programming system,
illustrated in Figure 1, which takes Fortran 77 with the proposed ANSI X3H5 directives as input, and
generates F77 code with vendor specific thread calls as output. These threads are managed by calls to
the Guide support libraries.

Before describing Guide in more detail, two related efforts shall be noted, that also attempt to
 facilitate portable parallel programs: the High-Performance Fortran (HPF) [For93] and the PVM/MPI
[BDG+93, For94] efforts. Both PVM and MPI provide message passing functions for programs that
run on parallel processors with separate memories. Several factors have contributed to the fact that
programming in these message passing packages has become widely applied. These factors include the standardization achieved by PVM and MPI, the fact that, for some time, message passing has been the only feasible way to program distributed-memory machines, and the fact that such programs can be easily ported to shared-address-space machines as well. On the negative side, message-passing programming can be tedious because of the task of partitioning a program and data space across processors and bookkeeping loop indices, array subscripts and conditional operations on partition boundaries.

The HPF effort was initiated with the goal of simplifying this very task. It provides a shared-address model to the user and generates the message passing program through intelligent translators. The user specifies data distributions of arrays, but does not need to get involved in array subscript bookkeeping operations. Although HPF supports a shared-address-space model, its emphasis is on distributed-memory/message passing machines. For example, it lacks concepts of loop-private arrays, which are among the most important language elements for programming SMP machines. On the other hand, HPFs pioneer work in data distribution concepts has generated knowledge that may, in the future, benefit the languages focused on in this paper. Currently the Guide language does not provide data distribution or other latency-hiding facilities, which may become important when we deal with true non-uniform memory access machines.

2.2 An Overview of the Guide Programming System

![Diagram of the Guide Programming System]

The input to the Guide Directive Translator, pictured in Figure 1, is standard sequential Fortran 77, with Guide directives used to express parallelism. The Guide language is a relatively large directive language, so only those directives which were required for our back-end will be discussed here. For a more detailed description of the Guide programming model and syntax see [Kuc96].

To initiate a parallel section of code, the **C$PAR PARALLEL** directive is used. With this directive, subdirectives categorizing each variable within the section as shared or private, must be included. These subdirectives are **SHARED()** and **LOCAL()** respectively, with each taking the variables it describes as arguments. At the conclusion of a parallel region, a **C$PAR END PARALLEL** directive must be placed.
The version of Guide available at the time of this study, had not yet implemented a REDUCTION() subdirective. We used critical sections for implementing reduction operations. Critical sections begin with a CSVAR CRITICAL SECTION directive, and end with a CSVAR END CRITICAL SECTION directive. The implementation of reduction operations are discussed in more detail in Section 3.

3 The Polaris Parallelizing Compiler

3.1 Polaris overview

Polaris is a parallelizing compiler, originally developed at the University of Illinois. As illustrated in Figure 3.1, the compiler takes a Fortran77 program as input, transforms it so that it can run efficiently on a parallel computer, and outputs this program version in one of several possible parallel Fortran dialects.

```
DO i=1,n
   a(i) = b(i)
ENDDO
```

Figure 2: Overview of the Polaris parallelizing compiler

The input language includes several directives, which allow the user of Polaris to specify parallelism explicitly in the source program. The output language of Polaris is typically in the form of Fortran77 plus parallel directives as well. For example, the generic parallel language includes the directives "CSRDS PARALLEL" and "CSRDS PRIVATE a,b", specifying that the iterations of the subsequent loop shall be executed concurrently and that the variables a and b shall be declared "private to the current loop", respectively. Figure 3.1 shows several other output languages that Polaris can generate, such as the directive language available on the SGI Challenge machine series, the Sun SC4.0 Fortran directive language, and the Guide directives introduced above.

Polaris performs its transformations in several compilation passes. In addition to many commonly known passes, Polaris includes advanced capabilities for array privatization, symbolic and nonlinear data dependence testing, idiom recognition, interprocedural analysis, and symbolic program analysis. An extensive set of optional switches allow the user and the developer of Polaris to experiment with the tool in a flexible way. An overview of the Polaris transformations is given in [BEH+94].

The implementation of Polaris consists of some 200,000 lines of C++ code. A basic infrastructure provides a hierarchy of C++ classes that the developers of the individual compilation passes can use for manipulating and analyzing the input program. This infrastructure is described in [FHP+94]. The Polaris Developer's Document [Hoe96] gives a more thorough introduction for compiler writers.

Polaris is a general infrastructure for analyzing and manipulating Fortran programs. The use of this infrastructure as a parallelizing source-to-source restructurer in the main application. Another
application is that of a program instrumentation tool. Currently, Polaris can instrument programs for gathering loop-by-loop profiles, iteration count informations, and for counting data references. We made use of these facilities in order to obtain the results described in this paper.

3.2 Polaris modifications and enhancements

The Guide Programming System is targeted at shared-memory architectures, which is one of the classes of machines for which Polaris has been developed. Because of this, the initial movement of the compiler to this language was fairly simple. Most of the additions to the code were localized to the final output pass, since it is here that the generic directives used in the internal representation are mapped to vendor-specific directives. The output pass is the last one in a series of compilation passes. It generates the Fortran-plus-directive output language from the internal representation.

Polaris internally provides to its output pass generic directives that distinguish between serial and parallel loops, and shared and private variables. These generic directives could be directly mapped to their corresponding Guide directives. There was no directive for scalar reductions in the version of Guide used in this study, although in the most recent release, this has been included. We therefore had to create local copies of any scalar reduction variables for each processor, and combine the sums in critical sections after the loop had finished.

One issue that we had to resolve in the output pass for Guide, was to provide an efficient means of identifying the current processor on which a particular loop iteration is executed. The next section describes why this is necessary. The Guide language provides a function that can return the current processor number at runtime. However, Polaris-generated programs read this number once or even several times per loop iteration, and so a subroutine call would cause too much overhead. The Guide syntax does allow for preambles to parallel loops, and so a single call could be placed there, since it would be amortized over the total loop execution time. This has not yet been implemented in our Guide back-end, however.

Array Reduction and Privatization

Both reduction parallelization and array privatization are among the most important transformations of a parallelizing compiler [BEH+94, TP93, PE95]. As we have mentioned above, Polaris is capable of transforming array reductions into fully parallel loops. Figure 3 gives an example array reduction in its serial and parallelized forms. Often, the number and indices of the array elements involved in an array reduction operation (array A in our example) cannot be determined at compile time. In the case of such reductions, Polaris provides a local copy of the array to each processor, performs the accumulation operations of this array in the now fully parallel loop, and then recombines the local arrays after the loop is complete.

Array privatization likewise requires creation of local copies of an array for each processor. In cases that warrant privatization, an array is used as temporary storage for a given iteration. Data dependencies that may exist are a matter of storage reuse, and are not true flow dependencies [TP93]. Private arrays can be expressed in the Guide directive language by listing the arrays on the LOCAL list. Arrays whose size is not known at compile time cannot be declared private in this way. Instead, Polaris expands these arrays by a dimension equal to the number of processors and uses the processor identification as an index, as shown in Figure 4. Dynamic allocation of shared arrays is supported in most parallel Fortran dialects.

As mentioned above, there is no efficient way yet of getting the value my.proc.id() used in these examples. Polaris resolves this problem by stripmining the loop with the number of processors for the outer iterations and then using the outer loop index instead of the processor number. This is shown in Figure 5. We have used this transformation in both cases where the processor_id would be needed: parallel loops with reduction operations and private arrays. Again, in future versions of our back-end this value may be obtained within loop preambles, and the cost amortized over the total loop execution time.
DIMENSION A(N), Alloc(N, number_of_processors)
PARALLEL DO I=1,N
  Alloc(I, my_proc_id()) = 0
ENDDO

DIMENSION A(N)
DO I=1,N
  A(B(I)) = A(B(I)) + X
ENDDO

PARALLEL DO I=1,N
  Alloc(B(I), my_proc_id()) =
  Alloc(B(I), my_proc_id()) + X
ENDDO

PARALLEL DO I=1,N
  DO J=1, number_of_processors
    A(I) = A(I) + Alloc(I, J)
  ENDDO
ENDDO

(a) Serial Array Reduction  
(b) Parallel Array Reduction

Figure 3: Array Reduction operation

DO J=1,N
  DO I=1,N
    A(I) = ...
  ENDDO
...
DO I=1,N
  DO I=1,N
    A(I, my_proc_id()) = ...
  ENDDO
...
...
  DO I=1,N
    A(I, my_proc_id()) = ...
  ENDDO
ENDDO

(a) Serial 
(b) Expanded array

Figure 4: Array privatization through expansion

4 The Machines Used for Experimentation

The Sun SPARC 20 Multiprocessor architecture

Figure 6 shows the Sun Sparc20 workstation used in our experiments. It consists of four 100 MHz hyperSPARC processors that share a single global memory space [Sun96]. Each processor has its own 256-KB external cache and an 8-KB on-chip instruction cache. Both write-through with no write allocate and copy-back with write allocate caching schemes are provided. In the case of multiprocessing, the external cache uses the copy-back scheme, maintaining cache coherency through a high performance snoop mechanism. The copy-back scheme is preferred as it writes to main memory less often than a write through scheme, and therefore saves memory bandwidth for interprocessor communications through the shared global memory.

The SGI Challenge Multiprocessor architecture

A four-processor Silicon Graphics Challenge machine was our second target. It too is a shared memory machine, with a physically shared address space. As with the SPARC20, it is a bus-based machine which uses a snoop cache-coherence scheme; however, unlike the Sun workstation, it always uses an Illinois Protocol write-invalidate scheme.
5 The Performance of Guide

5.1 The Experiment

The main focus of our experiment was on the performance of code expressed with the Guide directives on the Sun SPARC20 workstation. In our experiments, we have studied several programs in four forms: the serial version, the version produced by Polaris expressing parallelism with the Sun specific directives, the version produced by Polaris expressing parallelism with the Guide directives, and the version parallelized by the Sun SC 4.0 compiler.

The serial version is the original version with some instrumentation functions. In the first step of our experiments, we measured the execution time of the serial loops. We inserted the instrumentation functions at the beginning and end of each loop and ran the resulting code to generate the timing profile. In order to minimize perturbation, we then eliminated the instrumentation functions inserted for either insignificant loops or the loops that execute a large number of times. We did this based on the profile obtained previously. We ran the resulting program to obtain the final profile, which then served as the basis for performance comparisons. To ensure correctness of the data, we usually ran the programs several times under the same environment and checked the range of execution time before we obtained the timing measurement.

In some cases we had to hand-modify the Polaris-generated code which expressed parallelism with the Sun directives, in order to resolve compatibility problems between Polaris and the back-end compiler. The back-end compiler is a preliminary release of Sun's Fortran SC4.0 compiler; we expect most of these
problems to be resolved in the next release. In the case of code expressed with the Guide directives, no such modifications were necessary.

The Fortran SC4.0 compiler also includes a capability to do automatic parallelization. We will compare the performance of programs parallelized by Polaris with the performance of this compiler. The comparison will be very interesting because the SC4.0 compiler is a commercially available optimizing compiler of the newest generation. Hence, it allows us to compare Polaris with the latest state of technology in industry. The two version of code generated by Polaris will then show us the differences in expressing parallelism with vendor specific directives, and the portable Guide directives.

Finally, overall timings of the codes were made on an SGI Challenge machine, to verify that the code was actually portable. Again, no modifications to the code were necessary to compile with the SGI F77 compiler.

5.2 Results on the SPARC20

Overall Performance Results

This section presents the experimental results obtained through various runs of the Perfect Benchmark programs AR2D, MDG, and TRFD on the four processor SPARC20 workstation.

![Graphs showing overall execution times](image)

Figure 7: Overall Execution times on the SPARC20

Figure 7 shows the overall timing of the serial and 4-processor parallel execution of these programs. The curves correspond to programs parallelized with Polaris generating Sun directives, Polaris generating Guide directives, and with the Sun parallelizing compiler, respectively. For comparison, the “ideal” curve shows \textit{serial.execution.time}. The speedup of these programs are given in Figure 8.

It is interesting to note that the two Polaris generated versions are very close to each other in performance. This demonstrates the fact that there is not a significant difference in expressing the parallelism with the vendor specific directives or with the Guide directives. However, it can also be seen in TRFD that the initial parallelization overhead, the difference between the serial execution time and the execution time of the 1 processor parallel version, is larger for the code expressed with Guide. One reason for this difference is that Guide code must go through two levels of control, since the code calls routines from the Guide libraries, which in turn call routines from the Sun system libraries, which would not be the case for codes directly expressed with the Sun directives.
Performance results of individual loops

The performance figures of individual loops give us more insight into the behavior of the programs and further support the similar efficiency of these means of expressing parallelism. We measured the total execution time of each loop accumulated over repetitive runs in the program as well as the average execution time.

Figure 9: Execution times and speedups of individual loops in ARC2D

The first code that we studied is ARC2D. The overall performance of ARC2D is less than that of the other codes. Moreover, the major loops parallelized by Polaris in MDG and TRFD show good speedups while the loops in ARC2D, as presented in Figure 9, do not. Individual loop speedups are within the range of 1 to 2 times the serial version. Figure 9 shows the parallel execution times of a few of the most time-consuming loops. Unlike the other programs, where there are only a few loops dominating the execution time, ARC2D has a large amount of smaller loops that contribute to the overall execution time. Almost all of these loops are easy for compilers to parallelize. In fact, in studies with previous generations of parallelizing compilers, we have found that all loops of this code were almost fully parallelized [BE92], leading to good parallel efficiency. Our performance curves show that fully parallel loops do not necessarily guarantee good speedup. A more detailed explanation of this performance on this machine is presented in [EPV96]. Most interesting, for our study, is that for all versions the performance is similar, with the Guide version incurring no significant loss of performance for its portability.

This effect is also visible in the experiment with MDG, as seen in Figure 10. The execution time of MDG is dominated by one loop, INTERF.do1000. The speedup of this loop was excellent in both Polaris versions, exceeding 3. The loop POTENG.do2000 was not run in parallel in the version expressed with the Sun directives, because of a problem in the back-end compiler. The Guide version, did not have this problem, and POTENG.do2000 was able to execute in parallel. This lead to a better performance with
the Guide directives than the SC4.0 directives. Polaris detected both loops as parallel. In contrast, the Sun parallelizer did not detect parallelism in either loop, resulting in no overall speedup.

In TRFD, there are only two loops that have noticeable effects on the execution time: OLDA_do100 and OLDA_do300. Parallelization of these loops requires the substitution of generalized induction variables [PE95] and a dependence test on non-linear subscripts [BE94]. They were successfully parallelized by Polaris, but not by the Sun compiler. Instead, the Sun compiler parallelized several inner loops within these two loops. However, the execution times of these loops is so small that their parallelization is not profitable. The execution times and speedups are plotted in Figure 11. In this case, as was mentioned above, the initial parallelization overhead in the Guide version is higher. This leads to an overall worse performance for the Guide version, although this difference is much less by the time 4 processors are used.

5.3 Results on the SGI Challenge

Preliminary Performance Results

In order to prove the concept of portability of the Guide language, we have run the codes on a 4-processor SGI Challenge machine. As mentioned earlier, no modifications were necessary to the Guide code in order to compile it on this machine.

Figure 12 shows the overall timing of the serial and 4-processor parallel execution of these programs executed on the SGI Challenge. Since the SGI Challenge serves as a verification of portability only, the performance of these programs will not be analyzed in more detail. It should be noted, however, that the slow-down at four processors on ARC2D and MDG, was due to the loading on the machine and not due to other performance issues. A single user environment was not enforced on the SGI machine, and the performance shown in Figure 12 should be seen as only an indication that the Guide language is capable of speedup on other machines without any modification of the source-level code.
6 So, is Guide the answer to portable code?

The Guide Programming System is one possible solution to the problem of generating portable code for shared-address space machines. In this paper, we have shown that the performance of code expressing parallelism with Guide directives can be comparable to that of code with vendor-specific directives. We have also shown that programs expressed in Guide for a Sun multiprocessor could be ported and run with comparable efficiency on an SGI Challenge machine.

However, to what degree can portable code be efficient for a wide range of architectures? We are currently involved in answering this question, by analyzing the performance of Guide on architectures other than SPARC and SGI machines.

Currently, the Polaris parallelizer performs mostly architecture-independent transformations: it recognizes parallelism and expresses it in one of several language dialects. Beyond these techniques, there are a range of transformations that will need to be applied in an architecture-dependent way. For example, if the architecture has a long latency for starting a parallel loop then the “profitability threshold” will need to be set high. If an architecture has a shared cache then the locality-enhancement transformations (loop interchanging, strip mining, tiling, fusion) will need to be applied differently from distributed-cache machines.

To what extent such transformations can be done in the back-end compiler (the Guide compiler in our case) needs to be quantified. Where difficult decisions are necessary, the user or the parallelizing compiler should get involved, and they should find constructs that let them express important facts in the language. We have found indication that this will be necessary. For example, we have expected that the Sun-specific compiler would generate better code where it can recognize parallelism equally well as Polaris. However, we have found that SC4.0’s architecture-specific transformations only occasionally outperform the more straight-forward transformations of Polaris. In other cases they “over optimize” the code and cause slow downs. Evidently more informed decisions on applicable techniques will need to be made. This may necessitate language extensions for passing this information from the preprocessor or user to the back-end compiler.

Finally, we will need to address the needs of true non-uniform memory access machines. We will need to quantify the extent to which latency-hiding language constructs will need to be introduced. Will it be prefetch operations or data-distribution a la HPF, or a mix of both? This is currently a wide-open
References


Language Support for
Synchronous Parallel Critical Sections

Christoph W. Kessler
Helmut Seidl
Fachbereich IV - Informatik, Universität Trier
D-54286 Trier, Germany
e-mail: kessler@psi.uni-trier.de

Abstract
We introduce a new parallel programming paradigm, namely synchronous parallel critical sections. Such parallel critical sections must be seen in the context of switching between synchronous and asynchronous modes of computation. Thread farming allows to generate bunches of threads to solve independent subproblems asynchronously and in parallel. Opposed to that, synchronous parallel critical sections allow to organize bunches of asynchronous parallel threads to execute certain tasks jointly and synchronously. We show how the PRAM language Fork95 can be extended by a construct join supporting parallel critical sections. We explain its semantics and implementation, and discuss possible applications.

1 Introduction

In a parallel environment, critical sections (for a survey, see e.g. [4]) are segments of code accessing data which are visible to more than one parallel thread. Their implementation is one of the key problems, e.g., of global resource management or consistency in parallel databases. Classically, semaphores are used to avoid more than one thread at a time to execute the critical section. The remaining threads also aiming to enter the section are kept in a (priority) queue where they wait until the presently executing thread has left the critical section. In the sequel, we will call such a mechanism sequential critical section.

The performance of sequential critical sections is acceptable as long as the critical section is short and the frequency by which threads demand to enter is low. In a massively parallel surroundings, however, with several thousand processors a sequential critical section can very soon become the bottleneck for the overall performance of the system.

One solution to this problem is given by synchronous parallel critical sections. In a synchronous parallel critical section (PCS for short) several threads are allowed to enter simultaneously. Inside the critical section and as soon as the entering phase has been finished they jointly execute a synchronous parallel algorithm. Having terminated, the threads return to their original mode of computation, the critical section gets unlocked, and a new bunch of threads is allowed to enter.

To make this idea work, several questions have to be answered: What are possible conditions under which threads are allowed to enter? When should the enter procedure be terminated? What happens with threads not allowed to enter? Should they be blocked? Should they be allowed to continue?

In order to investigate possible answers to these questions and to study its implications on the semantics and efficiency of programs we extend the parallel language Fork95 by a new language construct join.

Fork95 is an experimental parallel programming language which has been designed to write elegant and efficient programs for synchronous shared memory MIMD ma-
chines (also known as PRAM’s). PRAM’s are particularly well suited for the implementation of irregular numerical computations, non-numerical algorithms, and database applications. One such machine currently under construction at Saarbrücken University is the SB-PRAM [1, 2]. The SB-PRAM is a lock-step-synchronous, massively parallel multiprocessor with up to 4096 RISC-style processing elements and with a (from the programmer’s view) physically shared memory of up to 2GByte with uniform memory access time.

In Fork95, processors are organized in groups. Groups may be temporarily subdivided into subgroups; this may also be applied recursively. Thus, at any point of program execution, the groups form a tree-like hierarchy, with the group consisting of all started processors as root, and the leaf groups being currently active. A synchronous mode of computation guarantees exact synchronicity for the leaf groups. Furthermore, Fork95 provides the possibility for thread farming, i.e., for locally switching from synchronous mode into an asynchronous mode of computation where desired by the programmer. This facility is crucial when tuning programs for efficiency. In some respect, the new construct join turns out to be complementary of farming.

The rest of the paper is organized as follows. The next section gives a short overview over the language Fork95 as it is. Section 3 presents the join construct together with its semantics. Section 4 explains how this construct can be implemented efficiently. Section 5 contains some examples together with explications of areas of possible applications. Section 6 contains measurements. Section 7 discusses further generalizations and concludes.

2 A short introduction to Fork95

For reasons of self-containment let us briefly recall the basic concepts of the programming language Fork95. A more detailed description can be found e.g. in [8] or [9]. Fork95 is a redesign of the PRAM language FORK [6]. In order to enable reuse of existing C code, Fork95 is based on ANSI C [3]. Additionally, it offers constructs that manage shared and private address subspaces, control synchronicity, and hierarchically divide groups of processors into subgroups. Fork95 makes the instruction-level synchronicity of the underlying hardware available to the programmer. It further enables direct access to hardware-supplied multiprefix operations.

2.1 Shared and private variables

The entire shared memory of the PRAM is partitioned into private address subspaces (one for each processor) and a shared address subspace. Accordingly, variables are classified as either private (pr, this is the default) or shared (sh), where “shared” always relates to the processor group that defined that variable.

There is a special private variable $ meant to hold the current group-relative processor ID which is initially set to the physical processor ID __PROC_NR__. The special shared variable @ is meant to hold the current leaf group ID. @ and $ are automatically saved and restored when subgroups are entered resp. left. However, the user is responsible to assign reasonable values to them (e.g., at the fork() instruction).

We consider an expression to be private if it is not guaranteed to evaluate to the same value on each processor, e.g. if a private variable occurs in it.

Fork95 inherits the concurrent write conflict resolution scheme from the target hardware. On the SB-PRAM, if several processors write the same (shared) memory location in the same cycle, the processor with maximal __PROC_NR__ will win and
write its value (PRIORITY-CRCW-PRAM). However, as several other write conflict resolution schemes (like ARBITRARY) are also used in theory, meaningful Fork95 programs should not be dependent on such specific conflict resolution schemes; there are better language elements (multiprefix instructions, see below) that cover practically relevant applications for concurrent write.

2.2 Synchronous and Asynchronous Mode of Execution

Fork95 offers two modes of program execution that are statically associated with program regions. In synchronous mode, Fork95 maintains at each point of program execution the synchronicity invariant (SI) which says that all processors belonging to the same leaf group are operating strictly synchronously, i.e., they follow the same path of control flow and execute the same instruction at the same time. Also, all processors within the same group have access to a common shared address subspace. Thus, newly allocated "shared" objects exist once for each group allocating them.

— In asynchronous mode the SI is not preserved, and no shared variables can be allocated. The processors start the main() program in asynchronous mode.

Switching to synchronous mode for any <statement> could be simply expressed by

\[
\text{start <statement>}
\]

where all processors form, after exact barrier synchronization [8], one single processor group and execute <statement> in synchronous mode. Up to now, start (its name is due to historical reasons) is only supported at the top level of program control, and nesting (also dynamic) of start statements is forbidden; a weakness of the language that we would like to overcome. Providing a more general construct to switch from asynchronous to synchronous mode is the main issue of this paper.

Switching to asynchronous mode for a <statement> is done by

\[
\text{farm <statement>}
\]

Within <statement> any synchronization is suspended; at the end of <statement> the processors synchronize explicitly within their current leaf group.

Functions are declared to be either synchronous (sync) or asynchronous (async). Synchronous regions, namely sync functions (except from farm bodies) and start bodies, are executed in synchronous mode, while asynchronous regions, i.e. async functions (except from start bodies) and farm bodies, are executed in asynchronous mode. In order to obtain this static classification of code, from asynchronous regions only async functions can be called. A call to an async function from a synchronous region is automatically casted to a farm body. Using farm within asynchronous mode is superfluous and will be ignored.

The importance of being synchronous. The synchronous mode overcomes the need for protecting shared variables by locks because they are accessed in a determinisitic way: The programmer can rely on a fixed execution time for each operation which is the same for all processors at any time during program execution. Thus no special care has to be taken to avoid race conditions.

This is only possible because there is no virtual processing in Fork95. Consider e.g. the following program fragment

\[
\text{sync void foo( sh int x, a )}
\]

\[
\{ \text{... if ($<10$) a = x;}
\]

\[
\text{else y = a; ... }
\]

By the semantics of synchronous mode all processors of the else part must read the same value of a. In order to guarantee this in the presence of virtual processing, it
would be required to keep a group lock\(^1\) for each shared variable. Due to the presence of pointers and weak typing in C, a lock would be required for each shared memory cell!

_Pipelining_ through an arbitrary graph can be implemented in synchronous mode in a rather straightforward manner, as we have shown in [9]. This both covers pipelining through multidimensional arrays as used by systolic algorithms and all sorts of trees as needed in certain combinatorial algorithms. We summarize that Fork95's synchronous mode saves the time and space overhead for locking and unlocking but incurs some overhead to maintain the SI during synchronous program execution.

**The importance of being asynchronous.** In asynchronous program regions there are no implicit synchronization points. Maintaining the SI requires a significant overhead also for the cases where each group consists of only one processor, or when the SI is not required for consistency because of the absence of data dependencies. Hence marking such regions as asynchronous can lead to substantial savings. In our experimental work, we found considerable usage of the for statement and asynchronous functions to pay off in significant performance improvements (execution time was reduced by up to 50 percent).

### 2.3 Hierarchical group concept

In synchronous mode, the synchronicity invariant permits to relax synchronicity in two ways: either by using for as described above (and thus leaving synchronous mode), or by splitting a group into subgroups and maintaining the invariant only within each of the subgroups. This has to be taken into consideration if control flow may diverge due to private branching conditions. Shared if or loop conditions do not affect the synchronicity, as the branch taken is the same for all processors executing it. At an if statement, a private condition causes the current processor group to be split into two subgroups: the processors for which the condition evaluates to nonzero form the first subgroup and execute the then part while the remaining processors execute the else part. The available shared address space of the parent group is subdivided among the new subgroups. When both subgroups have finished the execution of their branch, they are released, and the parent group is reactivated by exact synchronization of all its processors. — A similar subgroup construction is required also at loops with private exit condition. All processors that will execute the first iteration of the loop enter the subgroup and stay therein as long as they iterate. Processors that leave the loop body are just waiting at the end of the loop for the last processors of their (parent) group.

Subgroup construction can also be done explicitly. Executing

\[
\text{fork (e1; $=e2; $=e3) <statement>}
\]

means the following: First, the shared expression $e1$ is evaluated to the number of subgroups to be created. Then the current leaf group is split into that many subgroups. Evaluating $e2$, every processor determines the number of the newly created leaf group it will be a member of. Finally, by evaluating $e3$, each processor may renumber its group–local processor ID within the new leaf group. Note that empty subgroups (with no processors) are possible; an empty subgroup's work is immediately finished, though. Continuing, we partition the parent group's shared memory subspace into that many equally-sized slices and assign each of them to one subgroup, such that each subgroup has its own shared memory space. Now, each subgroup executes <statement>; the processors within each subgroup work synchronously, but

\(^1\)This would partially sequentialize the then with the else group w.r.t. accessing variable a.
different subgroups can choose different control flow paths. After <statement> has been completed, the processors of all subgroups are synchronized, the subgroups and their shared memory subspaces are released, and the parent group is reactivated as the current leaf group.

Clearly, if a leaf group consists of only one processor, the effect is the same as working in asynchronous mode. However, the latter avoids the expensive time penalty of continued subgroup formation and throttling of computation by continued shared memory space fragmentation.

2.4 Pointers and heaps

The usage of pointers in Fork95 is as flexible as in C, since all private address subspaces have been embedded into the global shared memory of the SB-PRAM. Thus, shared pointer variables may point to private objects, and vice versa. The programmer is responsible for such assignments making sense.

Up to now, Fork95 supplies two kinds of heaps: one automatic shared heap for each group, and one private heap for each processor. While space on the private heaps can be allocated by the asynchronous malloc function known from C, space on the automatic shared heap is allocated using the synchronous shalloc function. The live range of objects allocated by shalloc is limited to the live range of the group by which that shalloc was executed. Thus, such objects are automatically removed if the group allocating them is released. Supplying a third variant for a global, "permanent" shared heap is addressed later in this paper.

Pointers to functions are also supported. For efficiency reasons, calls to functions via private pointers automatically switch to the asynchronous mode if they are located in synchronous regions. Private pointers may thus only point to async functions.

2.5 Multiprefix instructions

The SB-PRAM offers built-in multiprefix instructions which allow the computation of multiprefix integer addition, maximization, and and or for up to 4096 processors within 2 CPU cycles. Fork95 makes these powerful instructions available as Fork95 operators (atomic expression operators, not functions). For instance, consider

\[ k = \text{mpadd}( \&k\text{var}, \text{expression}); \]

All \( m \) processors executing this statement at the same clock cycle and with the same shared address \&k\text{var} as first parameter, let them be indexed \( j = 0, \ldots, m - 1 \) in the order of increasing \_PRC_NR_, evaluate first their private expression locally into a private integer value \( e_j \). Then, processor \( j \) assigns its private integer variable \( k \) the value \( s + e_0 + e_1 + \ldots + e_{j-1} \) where \( s \) denotes the previous value of the shared integer variable \&k\text{var}. Immediately after the execution of the mpadd instruction, \&k\text{var} contains, as a side effect, the global sum \( s + \sum_j e_j \) of all participating expressions.

3 Semantics of join: The bus analogy

A useful analogy to understand the behaviour of the new join operator is a bus stop. Imagine a city with several excursion bus lines. One excursion bus circulates on each bus line. At the door of each bus there is a ticket automaton that sells tickets when the bus is waiting. Tickets are numbered consecutively from 0 upwards. All passengers inside a bus form a group and behave synchronously. They can be distinguished by
an ID number $ which is initialized to their ticket number. Each bus has a bus driver, namely the passenger that obtained ticket number zero.

What happens at the bus stop? Passengers come by asynchronously and look for the bus to join the excursion. If the bus is done, they have the choice to either retry and wait for the next bus of this line (if there is one), perhaps by doing some other useful work meanwhile, or to resign and continue with the next statement. If the bus is not gone, it is waiting and its door is not locked, thus the passenger can get a ticket at the ticket automaton and enter. If a passenger in spe happens to be the first at the bus stop (which means that he receives ticket number 0), he becomes the bus driver and does some initialization work at the bus. He waits according to a certain delay strategy. Then he signals that he will start the bus and switches off the ticket automaton — thus no one can enter this bus any more. At this point, some passengers inside are still allowed to immediately spring off the starting bus if they desire. After that, the door is definitely locked. The passengers inside form a group and behave synchronously for the time of the bus tour. During the tour, they can allocate shared objects that are accessible to all bus passengers during the tour. After the tour, all passengers leave the bus at the bus stop and continue, again asynchronously, with their next work.

What does this mean in the context of parallel programming? The behaviour described in the bus analogy is supplied in Fork96 by a language construct

\[
\text{join ( delaystmt; springoffcond; Smalloc ) statement else useful work}
\]

The passengers are the processors. Each join instruction installs a unique bus line with a bus stop. delaystmt specifies a statement that is executed by the bus driver and models a time interval or a condition that must be met to start the bus. The spring-off condition springoffcond is a boolean expression supplied by the programmer; it may be different for different processors.\(^2\) Smalloc is a statement executed by the bus driver to install a new shared stack and heap for the bus. Its specification is optional. The bus tour corresponds to the proper body of the join instruction and must be a synchronous statement.

The else part is optional and specifies an asynchronous statement useful work that is executed by processors that miss the bus and by those that spring off. A retry statement occurring inside useful work causes the processor go back to the bus stop and try again to get the bus, similar to continue in loops.

Note that a bus cannot have more than one entry point (join instructions) within the program. If this is desired by the programmer, he can encapsulate the join into a function and call that function from several sites.

Internally, there is for each bus a lock gone, i.e., a shared variable that guards access to the ticket automaton, which, in turn, is a shared variable that is accessed by a multiprefix increment operation.

Buses of different bus lines can be nested. Recursion (directly or indirectly) to the same bus line will generally introduce a deadlock because that processor waits for a bus whose return he is blocking by his waiting.

Note that the passengers inside a bus will generally have their origin in different former leaf groups. The old group structure, as well as all locally defined shared objects of that old group structure, are not visible during the bus tour. Global shared objects are always visible.

\(^2\)If the consecutive numbering of the $ ID's is destroyed by some processors springing off, the programmer can re-install this by simply recomputing $ with a multiprefix incrementation at the beginning of the bus tour.
4 Implementation

Shared memory allocation One possibility to allocate a new shared stack and heap for the group of processors in the bus is, of course, a call to the permanent shared malloc routine, a sequential asynchronous Fork95 function. This, however, is too simple because we just want to use the join construct to implement such parallel storage allocation frameworks. Another possibility would be to let the compiler allocate a statically fixed quantum of shared memory for every bus. This has the drawback that it excludes reuse of this memory for other purposes while no bus is running. Therefore, we offer a different solution: The bus driver sacrifices a memory block from his private heap for the new shared stack of the new bus. The size of this memory block could be chosen dynamically such that, for instance, half of the bus driver's currently free portion of its private heap might be nationalized.  

Data structures For each join instruction encountered in the code, the following global variables are allocated to implement the bus concept (given as C pseudocode to facilitate the description. The implementation has been coded in SB-PRAM assembler):

```c
    sh char *SM;
    sh int gone;
    sh int ticket;
```

gone and ticket are statically initialized by zero at the beginning of each Fork95 program.

Translation of the join instruction The join instruction is only admissible in asynchronous mode. `delaystmt` is optional; it should be chosen appropriately to delay the departure of the bus as desired. Variable `ticket` can be used as a parameter in `delaystmt`. `springoffcond` should evaluate to a nonzero value if a processor that entered the bus should leave immediately when starting. `SMalloc` must return a pointer to a block of memory which will install the bus's shared memory.

The instruction

```c
    join ( delaystmt; springoffcond; SMalloc ) statement else useful_work
```

is translated as follows (again described as Fork95 pseudocode)

```c
    save the old value of $;
    if ( !gone ) { /* I am allowed to enter the bus. */
        $ = mpadd( &ticket, 1 ); /* get a ticket */
        if ( $ == 0 ) {
            /* I am the bus driver and set up its shared memory: */
            SM = SMalloc;
            save old shared group pointer and build private group frame;
            compute new shared group, stack, and heap pointer from SM;
            allocate a new shared group frame in this block;
            delaystmt;
            done = 1;
            wait two cycles. Now the value of the ticket variable remains unchanged; it is the exact number of passengers.
            Write it into the new synchronization cell.
        }
    } else {
```

\[\text{To compute this quantity, we provide an asynchronous library function } \text{int pravail()} \text{ which returns the number of free private storage cells.}\]
5 Examples

Parallel memory allocation As an example let us consider a simple storage allocation scheme where all memory is divided into $N$ equally sized blocks. Pointers to free blocks are maintained by a queue avail. The queue is implemented by a shared array together with two integers low and high. Integer low $\% N$ points at the first occupied cell in avail whereas integer high $\% N$ points to the first free cell.

sh char *avail[N]; /* array containing pointers to blocks */
sh int high, low;

To implement operations void free(char *p); and char *balloc(); we introduce an auxiliary function char *pcs(char *ptr, int mode); with an extra argument mode to distinguish between the two usages.

/* modes of pcs: */
#define ALLOC 1
#define FREE 0

Using pcs() we define:

void free(char *p) {
    pcs(p, FREE);
}

void balloc() {
    return pcs(NULL, ALLOC);
}

Now, function pcs() is implemented using the join construct. Essentially, it consists in applying an madd-operation to variable high (in case mode == FREE) resp. to variable low. It only has to be taken care of that no block can be delivered from
an empty queue.

```c
char *pcs( pr char *ptr, pr int mode ) {
  pr t, my_index, h, *result;
  result = NULL;

  join(for(t=0; t<2; t++); 0; malloc(100)) {
    if (mode == FREE) {
      my_index = mpadd( &high, 1);
      avail[my_index % N] = ptr;
      /* insert block ptr into list of available blocks! */
    }
    if (mode == ALLOC) {
      my_index = mpadd( &low, 1)
      if (my_index >= high) {
        /* sorry, cannot get block from avail queue! */
        result = NULL;
        mpadd( &low, -1);
        /* value of low must be corrected. */
      }
      else result = avail[my_index % N];
    }
  }
  else retry;
  return result;
}
```

This implementation should be contrasted to a conventional one protecting access to variables low and high by means of locks. The run time figures for a varying number of processors are given in Fig. 1. The break-even point is approximately at 128 processors. For $p > 128$ the run time for the asynchronous run time begins to grow linearly in the number of processors since the sequentialization due to the sequential critical section dominates the execution time.

Parallel critical sections A (sequential) critical section, as known from many concurrent programming platforms and operating systems, is a block of statements that can be executed by only one processor at a time. (In our analogy, this corresponds to a bus with only one passenger.)

Our construct join allows the implementation of a generalization of this concept, namely parallel critical sections. Using join, blocks of statements can be executed in parallel and synchronously by a set of processors that has been specified by the programmer. The programmer may specify an exact number of passengers or only a
maximum or minimum number for these — or some other shared criterion esteemed relevant: any desirable constellation can easily be programmed using delaystat and springoffcond appropriately. The implementation of the bus guarantees that no other processor can join a bus tour as long as the door is locked. They have to wait (at least) until the processors inside the bus have finished their excursion, i.e. have left the parallel critical section.

We conjecture that this concept of a parallel critical section is an important step towards the usability of Fork95 for programming not only for nice “stand-alone” parallel algorithms but also of realistic and critical applications such as parallel operating system kernels.

Encapsulating existing synchronous Fork95 code A second advantage is that now existing synchronous Fork95 code can be re-used much easier than before — it just has to be encapsulated into an appropriate join instruction. Since now several joins can be nested, it is possible to construct more complex Fork95 software packages [10].

6 Sequential versus Parallel Critical Sections

To compare the runtime behavior of sequential critical sections versus parallel critical sections, we have implemented a test suite with the following parameters: The asynchronous variant using usual sequential critical sections of length \(D_{crit}\) clock cycles (excluding overhead for locking/unlocking and the time to wait for the lock to become free), and further executing \(D_{noncrit}\) cycles in a noncritical section. The synchronous variant needs no locks, executes \(D_{noncrit}\) cycles in a join statement, excluding the overhead for the current join implementation. Each processor generates \(N\) queries to access the shared resource; the time \(T\) before issuing the next query is modeled using an exponentially distributed probability function.

The overall simulation is done according to the following pseudocode:

```c
sh simple_lock lock;
initlock( lock );
desynchronize the processors randomly;
for (i=0; i<N; i++) {
    generate(y);
    #if JOIN
        join ( do 20 cycles, 0, malloc(100) ) {
            do \(D_{noncrit}\) cycles
        }
    #else
        lockup( &lock );
        do \(D_{crit}\) cycles;
        unlock( &lock );
        do \(D_{noncrit}\) cycles;
    #endif
}
```

where \(generate(y)\) is a delay routine that takes exactly \(T = 6t\) steps\(^4\) with probability

\[^4\text{The numbers that occur here are multiples of 6 because 6 is the length of an empty for loop iteration.}\]
<table>
<thead>
<tr>
<th>$D_{crit}$</th>
<th>$y = 0$</th>
<th>$y = 4$</th>
<th>$y = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>proc</td>
<td>JOIN</td>
<td>asynchronous</td>
<td>JOIN</td>
</tr>
<tr>
<td>1</td>
<td>9507</td>
<td>7855</td>
<td>7613</td>
</tr>
<tr>
<td>8</td>
<td>9533</td>
<td>10555</td>
<td>8597</td>
</tr>
<tr>
<td>16</td>
<td>11057</td>
<td>13635</td>
<td>9593</td>
</tr>
<tr>
<td>32</td>
<td>11057</td>
<td>19769</td>
<td>11705</td>
</tr>
<tr>
<td>64</td>
<td>11057</td>
<td>32067</td>
<td>15951</td>
</tr>
<tr>
<td>128</td>
<td>11057</td>
<td>56443</td>
<td>24403</td>
</tr>
<tr>
<td>256</td>
<td>11057</td>
<td>103795</td>
<td>41301</td>
</tr>
<tr>
<td>512</td>
<td>11057</td>
<td>204089</td>
<td>75065</td>
</tr>
<tr>
<td>1024</td>
<td>11057</td>
<td>406697</td>
<td>142681</td>
</tr>
<tr>
<td>2048</td>
<td>11057</td>
<td>793913</td>
<td>277817</td>
</tr>
<tr>
<td>4096</td>
<td>12605</td>
<td>1580531</td>
<td>548191</td>
</tr>
</tbody>
</table>

Figure 2: Measurements for the test scenario on the SBPRAM (the simulator provides exact timings), given in clock cycles.

\[(1 - 2^{-y})^{(t-1)}2^{-y}\]. Thus $y$ controls the density of queries.

With $D_{noncrit} = 384$ and the overhead of the current implementation of join being around 150 cycles (which is due to programming in high level language and will be drastically reduced once coded in assembler), we obtain the measurements on the SB-PRAM given in Fig. 2.

Three variations of issuing queries to the parallel resp. sequential critical section have been examined: The first one ($y = 0$) assumes that a processor raises a new query immediately after the previous one has been answered. For long critical sections ($D_{crit} = 50$ cycles) the break-even point is between 8 and 16 processors; for short critical sections ($D_{crit} = 15$ cycles) it is at 32 processors. $y = 4$ models exponentially distributed query generation with an expected delay of around 200 cycles from the completion of the previous query. The break-even point for long critical sections is near 32 processors and for short critical sections at 128 processors. $y = 8$ models exponentially distributed query generation with an expected delay of around 3000 cycles from the completion of the previous query. The break-even points are here at 512 and 2048 processors, respectively.

Generally the join construct is profitable for a high density of queries to the shared resource which is influenced by the number of processors as well as the average density of queries on each processor. It is interesting to observe that at high numbers of processors, with the density of queries being very high, the query generation time becomes meaningless because the sequentialization by the critical section (and thus, $D_{crit}$) dominates the execution time.

[7] analyzes the average time behaviour of sequential and parallel critical sections using stochastic models based on the theory of queuing systems in discrete time. This work confirmed our empirical results and provided an exact prediction of the break-even point for given model parameters.

In our experiments we assumed that query generation follows some regular pattern; the modeling produced expected values for query generation time that were equally resp. exponentially distributed. In real parallel programs, this may not generally be the case. Instead, it is more likely that, considering a longer trace of program execution, there will occur bursts of more intensive query generation, separated by time intervals with rather sparse query generation. We believe that this scenario will even better profit from the join statement because it allows to collect also the large amount of queries in a burst interval and process them immediately as a whole, while
the asynchronous variant would result in a long queue of processors waiting to enter the critical section.

7 Conclusion

We presented a new language construct for synchronous parallel programming languages, the join. Complementary to farming, join supports the implementation of synchronous parallel critical sections, thus avoiding the bottleneck of sequential critical sections in a massively parallel programming environment.

An interesting direction of further research is the extension of Fork95 with object-oriented features. Object-oriented programming allows for a different and surprisingly elegant view on busses: An individual bus line, along with its properties common for all processors, could be interpreted as a special object equipped with certain data and methods. Especially, arrays of bus lines could be constructed each of which may run independently of the others. The idea of encapsulating access to shared data structures into some kind of object has already been proposed for sequential critical sections in OPUS where they are called shared data abstractions [5].

The Fork95 compiler is available from ftp.informatik.uni-trier.de in directory /pub/users/Kessler by anonymous ftp. This distribution also contains documentation, example programs and a preliminary distribution of the SB-PRAM system software tools including assembler, linker, loader and simulator. The Fork95 documentation is also available by www, http://www.informatik.uni-trier.de/~kessler/fork95.html.

References


Program Optimization
Finding Affine Partitions that Maximize Parallelism and Minimize Synchronization

Amy W. Lim and Monica S. Lam

Computer Systems Laboratory
Stanford University, Stanford, CA 94305
{aimee, lam}@cs.stanford.edu

Abstract

This paper presents the first algorithm to use the technique of affine partitioning to maximize the degree of coarse-grain parallelism in programs with arbitrary loop nestings and whose array accesses are affine functions of outer loop indices and loop-invariant variables. By finding a combination of time and space affine mappings, our algorithm can achieve the same effects as combinations of unimodular transformations with loop fusion, fission, scaling, reindexing and statement reordering.

1 Introduction

As multiprocessors become popular, it is of great importance to develop compilers that can automatically translate sequential programs into efficient parallel code. Since synchronization is expensive on a multiprocessor, getting high performance on a multiprocessor requires not only finding parallelism in the program but also minimizing the synchronization overhead. Furthermore, high synchronization frequency is generally accompanied with high levels of data communication between processors, which again reduces the efficiency of the machine. In fact, experience with parallel applications on multiprocessors indicates that it is not uncommon for programs with fine-grain synchronization to run even slower than their serial counterpart. It is therefore important that a compiler find parallelism that requires minimal synchronization. In other words, we wish to identify the coarsest granularity of parallelism in a program by finding the largest units of independent computations that can be carried out by different processors without synchronization.

There has been a lot of research on program transformations to enhance a program’s parallelism and data locality. The domain of such techniques is generally limited to loops, whose loop bounds and array accesses are affine functions of loop indices. Many forms of program transformations have been proposed, a catalog of which can be found in a survey by Bacon, Graham, and Sharp[2]. The key challenge that remains is how to combine these transformations optimally to achieve particular goals.

This paper presents an algorithm that finds the maximum degree of coarse-grain parallelism. We say that a program has $k$ degrees of parallelism if $O(n^k)$ units of computations can be executed in parallel, where $n$ is the number of iterations in a loop. In terms of loop transformations, a loop nest with $k$ degrees of parallelism is one that can be transformed to contain $k$ nested parallel loops, and the granularity is measured by the length of the computation in the parallelized loop. The domain of our algorithm is general programs with arbitrarily nested loops and whose array accesses are affine functions of outer loop indices or loop-invariant variables.

Research supported in part by DARPA contract DABT63-91-K-0003 and an NSF Young Investigator award.
Our algorithm is based on the concept of affine partitions. Instances of each instruction are identified by their surrounding loop index values, and affine transforms are used to map these loop index values to a partition number. Partition numbers are used for two different purposes:

- **Space Partition.** Operations belonging to the same partition number are mapped to the same processor. One possible mapping is to assign operations in partition \( i \) to processor \( i \).

- **Time Partition.** Operations belonging to partition \( i \) execute before those in partition \( i + 1 \). This execution order can be expressed as a loop, whose \( i \)th iteration executes all the operations belonging to partition \( i \).

Our algorithm finds a combination of time and space affine mappings that maximize the degree of parallelism for successively greater degrees of synchronization. The problem is formulated without using any imprecise abstractions, such as distance and direction vectors, commonly used in existing parallelizing compilers. The algorithm can be used to find all the parallelism at the coarsest granularity, or just up to the degree needed to exploit a particular parallel hardware configuration. From these affine mappings, a straightforward algorithm based on Fourier-Motzkin elimination\[^4, 13\] can be used to generate the desired SPMD (Single Program Multiple Data) code.

The technique of affine partitioning is strong enough to find all the loop level parallelism that can be exposed through a combination of many previously defined transformations, including

- loop fission (or loop distribution)
- loop fusion
- unimodular transforms (interchange, reversal, skewing)
- loop scaling
- loop reindexing (or index set shifting)
- statement reordering

This paper presents our parallelization algorithm, concentrating on how the problem is decomposed into subproblems and how their solutions fit together. We assume that readers are familiar with the literature on parallelizing compilers. Space limitation prohibits us from including details of our algorithm and proofs for our theorems, they can be found in another paper\[^12\].

The rest of this paper is organized as follows. We first describe our approach and present an overview of our algorithm in Section 2. We then formally define our model of a program and affine partition in Section 3. Next, we present algorithms for solving several subproblems that can be combined to find the maximum degree of coarse-grain parallelism. Finally, we discuss related work in Section 7 and conclude in Section 8.

## 2 Overview of the Algorithm

We decompose the overall problem of finding the maximum degree of parallelism into several subproblems; namely, how to maximize the degree of parallelism that requires \( O(1), O(n) \) amounts of synchronization, respectively. By solving each of these problems in turn, the algorithm finds successively more degrees of parallelism at higher synchronization costs. These steps are then repeated to find parallelism requiring \( O(n^2), O(n^3), \ldots \) synchronization until all the loop level parallelism is found. In practice, it is not necessary to find all the degrees of parallelism. It is sufficient to find just enough coarsest parallelism to keep all the processors available busy.

The subproblem of maximizing synchronization-free parallelism is formulated as dividing the dynamic operations into the largest number of independent partitions. More specifically, our algorithm finds an affine partition for each instruction that maximizes the degree of parallelism in each instruction. The partitions are subject to a set of space partition constraints, which ensures that processors executing operations in different partitions need not synchronize with each other.
In our first step to find parallelism with \( O(1) \) synchronizations, we consider all instances of the same instruction as a unit, and divide the instructions into a maximal set of partitions that can be executed sequentially. We then find the maximum degree of synchronization-free parallelism within each of the partitions as above, and enforce the sequential ordering between the partitions by barrier synchronization. The number of synchronizations can be no greater than the number of instructions in a program, which is constant for a given program and is independent of the number of iterations in a program.

Finally, to find parallelism with \( O(n) \) synchronizations, we wish to find an affine time partition for each instruction. The affine partitions are subject to *time partition constraints*, which ensure that data dependences can be satisfied by executing the partitions sequentially. The objective is to find a partition that yields maximum parallelism among operations within each of the sequential partitions.

The space partition and time partition constraints are similar in many ways and are amenable to the same kind of techniques. We use the affine form of the Farkas lemma\[5\] to transform the constraints into systems of linear inequalities. The problem of finding a partition that gives the maximum degree of parallelism while satisfying the space or time partition constraints reduces to finding the nullspace of a system of equations. The desired affine partitions can be found easily with a set of simple algorithms.

3 Definitions

We first introduce some notations and definitions that we use in this paper. We define our model of a program, its dependences and finally the affine partitions that we are seeking.

3.1 Programs

Throughout the paper, we use the notation \( v_i \) to represent the \( i \)th element of the vector \( v \), and \( v_{ij} \) to represent the subvector from the \( i \)th to the \( j \)th element of vector \( v \). (\( v_{ij} \) is the empty vector if \( i > j \)).

The domain of our algorithm is the set of sequential programs consisting of arbitrary nestings of loops and sequences of loops, whose array accesses are affine functions of outer loop indices or loop-invariant variables. Other constructs such as conditional statements and non-affine array accesses can be handled by treating them conservatively.

**Definition 3.1** A program is \( P =< S, \delta, D, F, \omega, \eta > \), where

- \( S \) is the set of instructions where an *instruction* is an indivisible unit such as a simple arithmetic operation on program variables. Instruction \( s \) appears lexically before instruction \( s' \) iff \( s \prec_p s' \).
- \( \delta_s \) is the *depth*, or the number of surrounding loops, of instruction \( s \).
- \( D_s(i) = D_s(i)^{\delta_s} \), derived from the loop bounds, is an affine expression such that \( i \) is the loop index of an instance of \( s \) iff \( D_s(i) \geq \delta \).
- \( F_{sr}(i) = F_s(i) + f_{sr} \) is an affine expression that maps an iteration \( i \) to the array index computed by the \( r \)th access function in instruction \( s \) to array \( x \).
- \( \omega_{sr} \) is true iff the \( r \)th access function in instruction \( s \) to array \( x \) is a write operation.
- \( \eta_{sr} \) is the number of common loops shared by instructions \( s \) and \( s' \).

3.2 Data Dependences

The access patterns in a program define the constraints of program transformations. Informally, there is a *data dependence* from an access function \( F \) to another access function \( F' \) if and only if some instance of \( F \) uses a location which is subsequently used by \( F' \), and one of
the accesses is a write operation. A data dependence set of a program contains all pairs of data dependent access functions in the program.

**Definition 3.2** Let \(<\prec, \prec', \omega, \eta >\) be the lexicographical less than operator for program \(P = G, \delta, D, F, \omega, \eta >\) such that \(\tilde{t} \prec_{ss'} \tilde{t}'\) is true if and only if iteration \(\tilde{t}\) of instruction \(\tilde{s}\) is executed before iteration \(\tilde{t}'\) of \(\tilde{s}'\) in \(P\). We define \(\prec\) as

\[
\forall \tilde{s}, \tilde{s}' \in S \quad \forall \tilde{t} \in \mathbb{Z}^{d}, \tilde{t}' \in \mathbb{Z}^{d'} \text{ s.t. } D_{s} (\tilde{t}) \geq \delta \land D_{s'} (\tilde{t}') \geq \delta,
\]

\[
\tilde{t} \prec_{ss'} \tilde{t}' \equiv \bigvee_{m=0}^{\eta_{ss'}} L_{ss'}^{m} (\tilde{t}, \tilde{t}')
\]

where \(L^{m}\) is defined as

\[
L_{ss'}^{m} (\tilde{t}, \tilde{t}') = \left\{ \begin{array}{l}
\tilde{t}_{1:m} = \tilde{t}_{1:m} \land \tilde{t}_{m+1} \leq \tilde{t}_{m+1}^{\eta_{ss'}} \quad m = \eta_{ss'} \\
\tilde{t}_{1:m} = \tilde{t}_{1:m} \land \tilde{t}_{m+1} \leq \tilde{t}_{m+1} \quad 0 \leq m < \eta_{ss'} \end{array} \right.
\]

\(L^{m}\) is true if and only if iteration \(\tilde{t}\) is less than iteration \(\tilde{t}'\) at depth \(m\) and the two iterations are identical up to depth \(m-1\).

**Definition 3.3** The data dependence set of a program \(P = G, \delta, D, F, \omega, \eta >\) is

\[
R = \left\{ < F_{ssr}, F_{ss'r'} > \mid (\omega_{ssr} \lor \omega_{ss'r'}) \land (\exists \tilde{t} \in \mathbb{Z}^{d}, \tilde{t}' \in \mathbb{Z}^{d'} | (\tilde{t} \prec_{ss'} \tilde{t}') \land (D_{r} (\tilde{t}) \geq \delta \land D_{r'} (\tilde{t}') \geq \delta) ) \right\}
\]

### 3.3 Affine Mappings

All operations of an instruction in a program are identified by their loop index values. Our algorithm uses affine transforms to map a vector of the original loop index values to either a processor identification number or an iteration number in a sequential loop. In the following, we define affine partitions formally and introduce a few properties used in the algorithms.

**Definition 3.4** An \(m\)-dimensional affine partition for instruction \(s\) in program \(P\) is an \(m\)-dimensional affine expression \(\Phi_{s} (\tilde{i}) = G_{s} \tilde{i} + C_{s}\), which maps an instance of instruction \(s\), indexed by \(\tilde{i}\), to an \(m\)-element vector.\(^1\) An \(m\)-dimensional affine partition for a program \(P\) is \(\Phi = [\Phi_{1}, \Phi_{2}, ... \Phi_{k}]\), where \(k\) is the number of instructions in \(P\).\(^2\)

**Definition 3.5** The rank of an affine partition for an instruction \(s\), \(\Phi_{s} (\tilde{i}) = G_{s} \tilde{i} + C_{s}\), is the rank of matrix \(G_{s}\). The rank of an affine partition \(\Phi\) for a program is the maximum of the ranks of the affine partitions for its instructions.

**Definition 3.6** Two one-dimensional affine partitions for instruction \(s\), \(\Phi_{s} (\tilde{y}) = C_{s} \tilde{y} + C_{s}'\), and \(\Phi'_{s} (\tilde{y}) = C'_{s} \tilde{y} + C'_{s}'\), are linearly dependent if and only if \(C_{s}\) and \(C'_{s}\) are linearly dependent. Two one-dimensional affine partitions \(\Phi\) and \(\Phi'\) for program \(P\) are linearly dependent iff \(\Phi_{s}\) and \(\Phi'_{s}\) are linearly dependent for all instructions \(s\) in \(P\).

### 4 Synchronization-Free Parallelism

We first consider the problem of finding parallelism that requires no synchronization. This is formulated as dividing the operations of a program into partitions, such that all data

\(^{1}\)Symbolic constants are incorporated into our algorithm by treating them as additions to the set of loop variables.

\(^{2}\)The coefficients of the affine partition are rational numbers. Each dimension or row of a partition for a program can be scaled to have integral values when necessary, as in code generation.
dependent operations belong to the same partition. By assigning each partition to a different processor, no synchronization is needed between processors.

In the following, we first define the necessary and sufficient constraints for an affine partition to be synchronization-free. We next describe an algorithm that solves the constraints and finds a partition that yields the maximum degree of synchronization-free parallelism, which is equivalent to having the highest rank among all the possible solutions.

4.1 Constraints

Definition 4.1 (Space Partition Constraints) Let \( R \) be the data dependence set for program \( P = \langle S, \delta, D, \mathcal{F}, \omega, \eta \rangle \), an affine partition \( \Phi \) for \( P \) is \textit{synchronization-free} if\(^3\)

\[
\forall \langle F_{xsr}, F_{xs' r'} \rangle \in R, \quad \forall \vec{t}, \vec{t}^\prime \in Z^{D_+}, \vec{t}^\prime \in Z^{D'_+} \text{ s.t.}
\]

\[
D_s(\vec{t}) \geq \vec{0} \AND D_s(\vec{t}^\prime) \geq \vec{0} \AND F_{xsr}(\vec{t}) - F_{xs' r'}(\vec{t}^\prime) = \vec{0}, \quad \Phi_s(\vec{t}) - \Phi_s(\vec{t}^\prime) = \vec{0} \quad (2)
\]

Let

\[
D_s(\vec{t}) = D_s \vec{t} + \vec{d}_s, \quad F_{xsr}(\vec{t}) = F_{xsr} \vec{t} + \vec{f}_{xsr}, \quad \Phi_s(\vec{t}) = C_s \vec{t} + \vec{c}_s
\]

The unknown \( X = \begin{bmatrix} -C_s & C_s' & (c_{s'} - c_s) \end{bmatrix} \), representing a row of a synchronization-free affine partition for instructions \( s \) and \( s' \), is constrained by each data dependence \( \langle F_{xsr}, F_{xs' r'} \rangle \in R \) as

\[
\forall \vec{y} \text{ s.t. } G \begin{bmatrix} \vec{y} \\ \vec{1} \end{bmatrix} \geq \vec{0} \AND H \begin{bmatrix} \vec{y} \\ \vec{1} \end{bmatrix} = \vec{0}, \quad X \begin{bmatrix} \vec{y} \\ \vec{1} \end{bmatrix} = \vec{0} \quad (3)
\]

with

\[
\vec{y} \equiv \begin{bmatrix} \vec{t} \\ \vec{t}^\prime \end{bmatrix}, \quad G = \begin{bmatrix} D_s & 0 & \vec{d}_s \\ 0 & D_{s'} & \vec{d}_{s'} \end{bmatrix}, \quad H = \begin{bmatrix} F_{xsr} & -F_{xs' r'} & (\vec{f}_{xsr} - \vec{f}_{xs' r'}) \end{bmatrix}
\]

4.2 Solving the Constraints

Algorithm 4.1 Find a highest-ranked synchronization-free affine partition for program \( P \).

Step 1: Construct the space partition constraints on a one-dimensional partition in the form of (3), each from a data dependence in \( R \). Then rewrite the constraints as a system of linear equalities in the form of \( A \vec{x} = \vec{0} \), using the affine form of the Farkas lemma[5].

Step 2: If we construct a multi-dimensional affine partition \( \Phi \) where all the rows differ from each other only in their constant terms \( (c_s) \), the rank of \( \Phi \) is 1. Thus, a highest-ranked partition needs only consist of one representative among partitions that differs only in \( c_s \)'s. Using a variant of Gaussian elimination, we eliminate all the unknowns \( c_s \)'s from \( A \vec{x} = \vec{0} \). Let the simplified system be \( A' \vec{x}' = \vec{0} \) where \( \vec{x}' \) represents the unknown coefficients \( C_s \)'s.

Step 3: Find the solutions to \( A' \vec{x}' = \vec{0} \), expressed as \( \mathcal{V} \), a set of basis vectors spanning the nullspace of \( A' \).

Step 4: Construct an \( r \)-dimensional affine partition \( \Phi \) for the program, where \( r \) is the number of basis vectors in \( \mathcal{V} \). Each basis vector forms one row of \( \Phi \). The coefficients \( C_s \)'s

\(^3\)For synchronization-free parallelism, it is not necessary to distinguish between the direction of the data dependences in \( R \). The redundancy is included in the constraints to emphasize the similarity with the definition in Section 6. It is a matter of simple optimization to eliminate the redundant constraints.
Figure 1: Iteration space and data dependences for Example 5.

are given the values of the basis vector, and the constant terms $c_i$'s are derived from the coefficients $C_i$'s using $AX = 0$.

**Theorem 4.1** Algorithm 4.1 finds a highest-ranked, synchronization-free affine partition for a program.

### 4.3 Maximum Synchronization-Free Parallelism

**Lemma 4.2** Let $r$ be the rank of a synchronization-free affine partition $\Phi$, assigning operation $i$ of instruction $s$ to processor $\Phi_s(i)$ yields $r$ degrees of synchronization-free parallelism.

**Theorem 4.3** Let $\Phi$ be a highest-ranked, synchronization-free affine partition for program $P$, assigning operation $i$ of instruction $s$ to processor $\Phi_s(i)$ yields the maximum degree of synchronization-free parallelism.

Given an affine space partition, it is straightforward to generate the parallel code. As the computations on different partitions are completely disjoint, the code would be correct as long as the operations mapped to each partition execute in their original sequential execution. Our algorithm is based on Ancourt and Irigoine's polyhedron-scanning code generation technique [1]; details and examples can be found in [11].

### 4.4 Example

We now introduce an example that we use to illustrate the steps in Algorithm 4.1.

**Example 5:**

for $l_1 = 1$ to 100 do  
for $l_2 = 1$ to 100 do  
\[ q[l_1, l_2] = q[l_1, l_2] + b[l_1 - 1, l_2]; \] \[ b[l_1, l_2] = a[l_1, l_2 - 1] \ast b[l_1, l_2]; \]

The iteration space for this 2-deep loop nest is shown in Figure 1, with each loop having 4 iterations. Each iteration of the loop has two operations, the white node representing an operation of instruction 1 and the black node an operation of instruction 2. The arrows represent dependences between the operations. It is easy to see from the figure that there is an affine partition that will give one degree of synchronization-free parallelism.

---

4 Sometimes, $r$ is greater than the rank of the affine partition. For example, completely disjoint computation would be mapped to different dimensions in the partition. A postpass step is used to eliminate such anomalies.
For the given code, we have

\[ \bar{q} \equiv \begin{bmatrix} l_1 \\ l_2 \end{bmatrix}, \quad \bar{p} \equiv \begin{bmatrix} l'_1 \\ l'_2 \end{bmatrix}, \quad \bar{y} \equiv \begin{bmatrix} \bar{q} \\ \bar{p} \end{bmatrix} \]

\[ D_1 = D_2 = D = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \bar{d}_1 = \bar{d}_2 = \bar{d} = \begin{bmatrix} -1 \\ 100 \\ 100 \end{bmatrix} \]

\[ F_{a11} = F_{a12} = F_{b11} = F_{b21} = F_{b22} = F_{c21} = F = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \]

\[ \bar{f}_{a11} = \bar{f}_{a12} = \bar{f}_{b21} = \bar{f}_{b22} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \bar{f}_{b11} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \quad \bar{f}_{c21} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \]

The first step is to construct the space partition constraints, each from a pair of data dependent access functions. From data dependence \(< F_{a11}, F_{a12} >\) and \(< F_{b21}, F_{b22} >\), the constraints derived are tautologous.

Let \( X_1 = X_2 = \begin{bmatrix} -C_1 & C_2 & c_2 - c_1 \end{bmatrix} \). The constraint derived from \(< F_{a11}, F_{a21} >\) is

\[ \forall \bar{y} \text{ s.t. } \begin{bmatrix} D & 0 & \bar{d} \\ 0 & D & \bar{d} \end{bmatrix} \begin{bmatrix} \bar{y} \\ 1 \end{bmatrix} \geq 0 \land \begin{bmatrix} F & -F & \bar{f}_{a11} - \bar{f}_{a21} \\ -F & F & \bar{f}_{b11} - \bar{f}_{b21} \end{bmatrix} \begin{bmatrix} \bar{y} \\ 1 \end{bmatrix} = 0 \] (4)

and the constraint imposed by \(< F_{b11}, F_{b21} >\) is

\[ \forall \bar{y} \text{ s.t. } \begin{bmatrix} D & 0 & \bar{d} \\ 0 & D & \bar{d} \end{bmatrix} \begin{bmatrix} \bar{y} \\ 1 \end{bmatrix} \geq 0 \land \begin{bmatrix} F & -F & \bar{f}_{b11} - \bar{f}_{b21} \end{bmatrix} \begin{bmatrix} \bar{y} \\ 1 \end{bmatrix} = 0 \] (5)

Rewriting (4) and (5) into a system of linear equalities

\[ \begin{bmatrix} 100 & -1 & 99 & 1 \\ 100 & -100 & 99 & 100 \\ 2 & -1 & 1 & 1 \\ 2 & -2 & 1 & 2 \\ 2 & -99 & 1 & 99 \\ 2 & -100 & 1 & 100 \\ 1 & -1 & 1 & 2 \\ 100 & -1 & 100 & 2 \\ 1 & -99 & 1 & 100 \\ 2 & 100 & 2 & 100 \\ 99 & -99 & 88 & 100 \\ 100 & -99 & 100 & 100 \end{bmatrix} \begin{bmatrix} -C_1 & C_2 & c_2 - c_1 \end{bmatrix}^T = 0 \] (6)

Step 2 of the algorithm eliminates the constant term \( c_2 - c_1 \) using the expression

\[ c_2 - c_1 = \begin{bmatrix} -100 & 1 & -99 & -1 \end{bmatrix} \begin{bmatrix} -C_1 \\ C_2 \end{bmatrix}^T. \]

Next, the algorithm finds the solutions to the reduced system by solving for the nullspace of the matrix of constants. From step 3, we compute the solution space as that spanned by the vector \( \begin{bmatrix} -1 & 1 & 1 & -1 \end{bmatrix} \). The corresponding \( c_2 - c_1 \) is 1. Let \( c_2 = 0 \), we assign -1 to \( c_1 \). Using the spanning vector, we construct a one-dimensional synchronization-free partition \( \Phi \) with \( \Phi_1 = l_1 - l_2 - 1 \) and \( \Phi_2 = l'_1 - l'_2 \).

Given the space partition \( \Phi \), we generate the SPMD code

\[
\text{if} \ (p = -99) \ \text{then} \\
\quad a[1, 100] = a[1, 100] + b[0, 100]; \\
\text{if} \ (-98 \leq p \leq 99) \ \text{then} \\
\quad \text{if} \ (1 \leq p \leq 98) \ \text{then}
\]

\[ b[p, 1] = a[p, 0] \times b[p, 1]; \]

for \( l_1 = \max(1, 1 + p) \) to \( \min(100, 99 + p) \)

\[ a[l_1, l_1 - p] = a[l_1, l_1 - p] + b[l_1 - 1, l_1 - p]; \]

\[ b[l_1, l_1 - p + 1] = a[l_1, l_1 - p] \times b[l_1, l_1 - p + 1]; \]

if \((-98 \leq p \leq 0) \) then

\[ a[100 + p, 100] = a[100 + p, 100] + b[99 + p, 100]; \]

if \((p = 100) \) then

\[ b[100, 1] = a[100, 0] \times b[100, 1]; \]

5 Parallelism With \(O(1)\) Synchronizations

To find parallelism that requires only a constant amount of synchronization, we define a program dependence graph and show how the concept of strongly connected components can be used to find parallelism with \(O(1)\) synchronizations.

**Definition 5.1** Let \( R \) be a data dependence set for program \( P = \langle S, \delta, D, F, \omega, \eta \rangle \). The dependence graph for \( P \) is \( G = (V, E) \), where node \( v_s \in V \) represents instruction \( s \), and \( <v_s,v_t> \in E \) if and only if \( \exists s,r,r' \text{ s.t. } <F_{xs},F_{xsr},F_{xsr'}> \in R \).

We first observe that executing several strongly connected components (SCC) in a dependence graph in parallel will not affect the degree of parallelism in a program. Therefore, we only need to extract parallelism within each component. Instructions belonging to the same component must be mapped to the same partition; otherwise, the number of synchronizations required to satisfy data dependences between the instructions is at least \(O(n)\) where \( n \) represents the number of iterations in a loop. We apply our synchronization-free parallelization algorithm to each SCC to maximize parallelism. We insert a barrier at the end of each parallelized SCC, and execute the SCCs sequentially in topological order to ensure data dependences between them are satisfied.

**Algorithm 5.1** Find all parallelism requiring \(O(1)\) synchronizations for computation that has no synchronization-free parallelism.

**Step 1:** Construct a dependence graph for the computation and partition the instructions into strongly connected components.

**Step 2:** Apply Algorithm 4.1 to each strongly connected component to get a space partition that maximizes synchronization-free parallelism.

**Algorithm 5.2** Find all parallelism requiring at most \(O(1)\) synchronizations.

**Step 1:** Find all synchronization-free parallelism using Algorithm 4.1.

**Step 2:** Apply Algorithm 5.1 to each of the parallel partitions found in step (1).

**Theorem 5.1** Algorithm 5.2 finds the maximum degree of parallelism that uses at most \(O(1)\) synchronizations.

6 Parallelism Nested Within Sequential Loops

We now consider those programs whose nodes in a program dependence graph make up one strongly connected component, and there is no available synchronization-free parallelism. Forming one strongly connected component, there must exist an outermost loop that surrounds all the instructions in the program. Having no synchronization-free parallelism, any transformed program will have an outermost loop, whose iterations have to be executed sequentially. We call such a program an outer sequential program.

Our goal is to find an affine mapping from the loop index values of each operation to an iteration of a loop, such that it is legal to execute the loop sequentially and operations
within each iteration have the largest degree of parallelism. If such parallelism can be found, we introduce barrier synchronization between each iteration of the outermost loop to ensure that the dependences are satisfied.

In the following, we first define a legal affine partition such that executing the partitions sequentially does not violate any data dependence constraints. We then calculate the maximally independent solutions to the constraints. From these solutions, we derive an affine time partition that exposes all the degrees of parallelism with only one degree of synchronization. Finally, we present our algorithm for finding the maximum degree of parallelism in a program.

6.1 Time Partitions

The problem is to partition the operations into iterations such that executing the iterations sequentially does not violate any data dependence. In other words, if operation \( v \) depends on operation \( u \), \( u \) must either execute in the same iteration as \( v \), or \( u \) is executed in an iteration that comes before the iteration that executes \( v \). Thus, the constraints for time partitions are simply a more relaxed version of the space partition constraints.

6.1.1 Constraints

**Definition 6.1** (Time Partition Constraints) Let \( R \) be the data dependence set for program \( P = \langle S, \delta, \mathcal{D}, F, \omega, \eta \rangle \), a one-dimensional affine partition \( \Phi \) for \( P \) is legal iff

\[
\forall < F_{xs}, F_{xs'}, > \in R, \ \forall \vec{i}, \vec{i}' \in Z^d, \vec{i} < \vec{i}', s.t. \\
F_{xs}(\vec{i}) \geq \delta \land F_{xs'}(\vec{i'}) \geq \delta \land F_{zs}(\vec{i}) - F_{zs'}(\vec{i'}) = 0, \\
\Phi_x(\vec{i}) - \Phi_x(\vec{i'}) \geq 0 \tag{7}
\]

Using Defn 3.2, we can rewrite the Time Partition Constraints as

\[
\forall < F_{zs}, F_{zs'}, > \in R, \ \forall \vec{i}, \vec{i}' \in Z^d, \vec{i} < \vec{i}', \forall m \in \{0, \ldots, n_{zs'}\} s.t. \\
F_{zs}(\vec{i}, m) \land F_{zs'}(\vec{i'}) \geq \delta \land \Phi_{zs}(\vec{i}) \land F_{zs}(\vec{i}) - F_{zs'}(\vec{i'}) = 0, \\
\Phi_{zs}(\vec{i}) - \Phi_{zs}(\vec{i'}) \geq 0 \tag{8}
\]

Thus, for each data dependence \(< F_{zs}, F_{zs'}, >\), we derive a system of constraints for each level \( m \).

**Lemma 6.1** Let \( \Phi \) be a legal affine partition for a program \( P \). It is possible to find an execution order that satisfies all the data dependences such that operation \( i \) of instruction \( s \) is executed before operation \( i' \) of instruction \( s' \) whenever \( \Phi_x(i) < \Phi_x(i') \).

**Lemma 6.2** There exists at least one legal affine partition for a program.

6.1.2 Solving the Constraints

**Definition 6.2** A set of legal one-dimensional affine partitions \( B = \{ \Phi_1, \Phi_2, \ldots, \Phi_n \} \) is said to be maximally independent iff the set is minimal and any legal one-dimensional affine partition \( \Phi \) can be expressed as a linear combination of the partitions in \( B \). That is,

\[ \exists a, k_1, \ldots, k_n \in Q s.t. \Phi_x(\vec{y}) = k_1 \Phi_1^x(\vec{y}) + k_2 \Phi_2^x(\vec{y}) + \cdots + k_n \Phi_n^x(\vec{y}) + a \]

**Algorithm 6.1** Find a set of maximally independent, legal affine partitions to an outer sequential program \( P \).
Step 1: Similar to step (1) in Algorithm 4.1, we use the affine form of Farkas lemma to rewrite each system of constraints from (8) as \( A\bar{x} \geq \bar{0} \). Let the size of matrix \( A \) be \( m \times n \). Given that the program has no synchronization-free parallelism, the rank of \( A \) must be \( n \).

Step 2: We introduce a set of new variables \( \bar{b} \), one for each row of \( A \), such that

\[
A\bar{x} \geq \bar{0} \iff \begin{bmatrix} A & -I_{m \times m} \end{bmatrix} \begin{bmatrix} \bar{x} \\ \bar{b} \end{bmatrix} = \bar{0}, \quad \bar{b} \geq \bar{0} \tag{9}
\]

The columns of \( \begin{bmatrix} I_{n \times n} \\ \bar{b} \end{bmatrix} \) forms a basis \( \mathcal{V} \) for all the solutions to

\[
A \begin{bmatrix} \bar{x} \\ \bar{b} \end{bmatrix} = \bar{0} \tag{10}
\]

Step 3: Any solution\(^5\) to (9) must also be a solution for (10) and is in the vector space spanned by \( \mathcal{V} \). Using our legal partition algorithm \([12]\) we derive from \( \mathcal{V} \) a maximal set of linearly independent solutions \( (\bar{x}) \) to (9).

Theorem 6.3 Given a program that has a single strongly connected component and has no synchronization-free parallelism, Algorithm 6.1 finds the maximally independent legal affine partitions for an outer sequential program.

6.2 Parallelism With \( O(n) \) Synchronizations

Theorem 6.4 Let \( B = \{ \Phi^1, \ldots, \Phi^k \} \) be a set of maximally independent legal affine partitions for an outer sequential program \( P \). Then any positive linear combination of the partitions such as \( \Phi = \sum_{i=1}^{k} \Phi^i \) is a legal affine partition for \( P \), whose individual partitions have the largest degree of parallelism requiring at most \( O(n) \) synchronizations. Moreover, each partition of \( \Phi \) has at least \( k - 1 \) degrees of synchronization-free parallelism.

Algorithm 6.2 Find all parallelism requiring \( O(n) \) synchronizations for an outer sequential program.

Step 1: Apply Algorithm 6.1 to the outer sequential program. Let \( B \) be the set of maximally independent partitions returned. We derive an optimal time partition as \( \Phi = \sum_{b \in B} \bar{b} \).

Step 2: Find the maximum degree of nested parallelism that requires at most \( O(1) \) synchronizations by applying Algorithm 5.2 to the code representing a partition of \( \Phi \).

Theorem 6.5 Algorithm 6.2 finds the maximum degree of coarse-grain parallelism with at most one degree of synchronization in an outer sequential program.

6.3 Maximum Degrees of Parallelism

Algorithm 6.3 Find the maximum degrees of coarse-grain parallelism in a program.

Step 1: Find maximum degree of synchronization-free parallelism: Apply Algorithm 4.1 to the program.

---

\(^5\)If a loop nest has constant bounds, one legal solution to the equations is to have affine partitions with coefficients larger than the number of iterations in the loop. Such a solution is equivalent to serializing the iterations in the loop nest. Our algorithm takes an extra step to exclude such solutions.
Step 2: Find maximum degree of parallelism that requires $O(1)$ synchronizations: Apply Algorithm 5.1 to each of the space partitions found in step (1).

Step 3: Find maximum degree of parallelism that requires $O(n)$ synchronizations: Apply Algorithm 6.2 to each of the partitions found in step (2).

Step 4: Find maximum degree of parallelism with successively greater degrees of synchronization: Recursively apply step (3) to computation belonging to the same partition until all parallelism is found.

In practice, it is not necessary to find all the degrees of parallelism. Once we have located enough parallelism, we can terminate the algorithm at any step.

Theorem 6.6 Algorithm 6.3 finds the maximum degree of parallelism in a program. The degrees of parallelism found are at their coarsest granularity.

7 Related Work

There has been a lot of research on loop transformations and we limit our comparison to work most closely related to ours. Loop interchanges, reversals, and skewing, and combinations thereof have been modeled as unimodular transformations[3, 14, 15], and various algorithms based on this framework have been developed. Under this framework, the desired combination of loop transformations is achieved by finding the suitable unimodular matrix, which can then be used to generate the desired SPMD code in a straightforward manner. There are several major limitations to the unimodular loop transformation approach: imprecise dependence abstractions such as distance and direction vectors are used, operations within each iteration are treated as indivisible, and unimodular transformations do not apply to non-perfectly nested loops. Unimodular transformations cannot achieve the effects that can be obtained via loop fission, fusion, scaling, reindexing, or statement reordering.

Feautrier used his affine framework to address the problem of finding a piece-wise affine schedule that minimizes the overall execution time of a program[5, 6]. His approach is to schedule the instructions to maximize parallelism first, then minimize communication[7]. He finds for each instruction a piece-wise affine mapping to a multi-dimensional time index. The algorithm uses a heuristic to minimize the dimensionality of time using parametric integer programming. Minimizing the dimensionality of a time schedule corresponds to maximizing the degree of parallelism. By solving for the coarsest granularity of parallelism, we minimize synchronization as we find parallelism. By focusing only on the degree of parallelism, rather than the shortest free schedule as in Feautrier's case, we can use simple algorithms for finding null spaces rather than parametric integer programming techniques. Code generation in our framework is straightforward because the schedules we find are more regular.

Kelly and Pugh presented a framework for unifying iteration reordering transformations, and described a way to find an affine mapping for each instruction based on performance estimation[9, 8]. Recently, they described an algorithm to minimize communication while preserving parallelism[10]. They estimate the parallelism and communication cost for all legal loop permutations and reversals for each instruction. They then build a conflict graph and search for the best mapping with the minimum cost. Their algorithm only finds one dimension of parallelism, rather than all the possible degrees of parallelism. By restricting their search to only loop permutations and reversals for each instruction, they cannot find parallelism that requires skewing the loops.

This paper extends our previous work on finding affine partitions to maximize the degree of communication-free parallelism[11]. This paper presents a simpler and more direct algorithm, made possible by the application of the Farkas lemma. In general, programs cannot be parallelized without any synchronization. This paper shows how the algorithm for synchronization-free parallelism can be used as part of an algorithm that introduces synchronization where necessary.
8 Conclusion

This paper defines two fundamental program transformation constraints: the space partition constraints and the time partition constraints, where the latter is a relaxation of the former. We show that the problem of finding all the degrees of parallelism can be reduced to finding solutions to these constraints alternately. The parallelism found is at the coarsest granularity possible.

The algorithm to this problem is made simple using two ideas. First, we use the Farkas lemma to reduce the constraints to a set of linear inequalities. Second, with the degrees of parallelism as the metric, it is not necessary to use complex optimization algorithms. Instead, simple techniques that find the solution space to linear inequalities are adequate. This observation applies not only to finding the space partitions but also, perhaps more surprisingly, to finding the time partitions. The key insight is that a set of maximally independent legal partitions can be combined linearly to form a time partition that yields maximum parallelism.

References


Optimal loop parallelization under register constraints *

Christine Eisenbeis
Antoine Sawaya
INRIA Rocquencourt†

Abstract

This paper addresses the interaction between instruction level parallelization and register allocation, in the case of straight line code and in the case of loops. This problem is at the heart of code optimization in microprocessors with instruction-level parallelism. Usual solutions use heuristics based on a decoupled approach. We propose here a formulation by linear integer programming, that allows dependence, resource and register constraints to be integrated in the same framework. By varying the parameters, a number of optimization problems can be solved exactly (maximization of the throughput, minimization of the number of registers). We report on examples of computation timings that turn out to be prohibitive in some specific cases, but tractable on average. We give examples of applications as well as a comparison of our approach with another work addressing the buffer size minimization problem.

1 Introduction

The problem of how to orchestrate register allocation [1] and instruction scheduling is one of the most important challenges in optimizing compilers for today’s high performance superscalar microprocessors. First performing instruction scheduling may increase register pressure unacceptably. Conversely, beginning by register allocation may introduce artificial data dependencies that limit parallelism. Many authors have tackled this difficult problem, either by systematic experiments [2, 3] or by more theoretical considerations on respective influence of scheduling and register allocation on each other [4, 5], or by heuristics [6, 7].

In a previous paper, Eisenbeis et al. have suggested an exact Integer Linear Programming (ILP) formulation for managing register allocation and instruction scheduling in a common framework. Their work applies only to basic blocks composed of forests of unary/binary trees, where one variable is allowed to serve as operand of a next operation only once. In this paper, we improve their results in two directions. First, we show how to cope with more than one variable reuse, hence extending the formulation to general Directed Acyclic Graphs (DAGs). Second, we explain how looping can be handled in that framework, hence extending the formulation to general loops without branches. By applying any off-the-shelf ILP solver, we are able to exactly solve instructions scheduling and register allocation simultaneously.

We have implemented our ILP formulation of software pipelining, including an ILP formulation of resource constraints [8]. We explain which kind of applications this formulation can help to handle (architectural as well as code optimization concerns). We

*This work was partially supported by ESPRIT Project COMPARE
†INRIA Rocquencourt, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, FRANCE
also compare our register constraints formulation to the formulation of [9] of “buffer size minimization”. We show that our formulation is more precise than the latter since it results always in a least register pressure.

We first recall basic definitions in Linear Programming, then introduce our different formulations for respectively forests, DAGs and loops. Experimental results are given in section 4.

Our result is not new in itself since some similar approaches have been proposed more or less recently in the literature. How these different approaches differ is described in the section 5.

2 Problem formulation and definitions

In this section, we give the general formulation of our scheduling problem. No attempt is made here to minimize the number of equations or variables. The purpose is just to give an idea of how it works.

2.1 Basic definitions and notations

Linear Programming (LP) : A linear programming problem can be formulated as:
find a vector $x$ with non-negative subscripts, such that $Ax \preceq b$, with $cx$ is minimal. $A$ is a matrix, $b$ a vector called the right hand side, and $c$ a vector specifying the objective function. The sign $\preceq$ means that every row of $A$ (specifying one constraint) can have its own (in)equality $\leq$, $\geq$ or $\approx$. Some number of the subscripts of $x$ may be required to be of type integer in the declaration part of the program. General LP solvers use the simplex algorithm and sparse matrix methods [10].

Dependency graph : A simple loop $B = \{ \text{do } i=1,k \text{ op}_1 \ldots \text{op}_n \text{ end do} \}$ is represented by its dependency graph $G = (O, E, \delta, \lambda)$. $O$ is the set of the vertices which are the statements of the loop body, $|O| = n$, the number of operations in the loop body. $E$ is the set of dependency edges; $|E| = m$. An edge is drawn between two statements if these are related by a flow, anti or output dependency. Data dependencies represent a set of constraints between statement execution. Satisfying data dependencies is sufficient to preserve the semantics of the original sequential loop after any transformation. To each edge $e = (\text{op, op'})$ of the graph are associated two non-negative integers the latency $\delta(e)$ and the dependency distance $\lambda(e)$. They mean that op' can be issued only $\delta(e)$ cycles after the operation op of the $\lambda(e)^{th}$ previous iteration.

Loop schedule : Loop scheduling is performed by software pipelining [11]. A loop schedule is a mapping function from $O \times \mathbb{N}$ to $\mathbb{N}$ (non-negative integer set). $\sigma(\text{op, i})$ denotes the execution cycle where the instance of operation op of the $i^{th}$ iteration is issued. A periodic loop schedule is a schedule of the form $\sigma(\text{op, i}) = \sigma_{op} + hi$. The initiation interval $h$ is considered to be a non-negative integer as well as $\sigma_{op}$ which is a mapping from $O$ to $\mathbb{N}$. $\sigma_{op}$ can be viewed as the schedule of the $0^{th}$ iteration. In the general case, when considering unrolling, we can look for rational numbers $h$ and $\sigma_{op}$, and the loop scheduling function will be defined by $\sigma(\text{op, i}) = [hi + \sigma_{op}]$ [12]. The minimum initiation interval $h_0$ is the least $h$ satisfying dependency constraints. In graph theory, $h_0$ is known as the value of the critical cycles of the graph and is computed as follows:

$$h_0 = \max_{\forall C \subseteq G} \frac{\delta(C)}{\lambda(C)}$$

$$\delta(C) = \sum_{\forall e \in C} \delta(e) \text{ and } \lambda(C) = \sum_{\forall e \in C} \lambda(e)$$

There exist algorithms that compute that quantity in time $O(nm\log n)$ [13].
Dependency constraints: Dependency constraints follow directly from the dependency graph. The following equations express linearly dependency constraints for a schedule $\sigma$:

$$\forall (op, op') \in E \quad \sigma_{op} + \delta \leq \sigma_{op'} + \lambda \cdot h$$

(2)

There are $O(m)$ different equations of this type.

Binary variables: We consider a margin of $L$ time units in which all the operations of a same iteration must be issued. A variable $x_{op,t} \in \{0, 1\}$ will be associated with each operation $op \in O$ and time unit $t \in [1, L]$. The value of $x_{op,t}$ is set to 1 if and only if operation $op$ of the first iteration is scheduled at time $t$ (i.e. $\sigma_{op} = t$). Note that for every operation $op \in O$, only one element $x_{op,t}$ must be set to one, all the others to zero, so that the constraints can be formulated by the two following inequalities:

$$\forall t \in [1, L], \forall op \in O, \quad x_{op,t} \geq 0$$

(3)

These constraints are implicit in the linear programming problem.

$$\forall op \in O, \sum_{t=1}^{L} x_{op,t} = 1$$

(4)

There are $n$ different equations of this type and $O(n \cdot L)$ variables $x_{op,t}$.

It is easy to pass from binary variables $x_{op,t}$ to schedule times $\sigma_{op}$ by the following equations:

$$\forall op \in O, \quad \sigma_{op} = \sum_{t=1}^{L} t \cdot x_{op,t}$$

(5)

There are $n$ such equations.

Earliest and latest schedule: Given the dependency graph of a basic block (acyclic graph) and a margin of $L$ units time, we may want to schedule the operations as soon as possible, while satisfying the dependency constraints. $L_{\sigma, \text{early}(op)}$ denotes the earliest possible time at which we can schedule $op$ without transgressing any data dependency constraint. $L_{\sigma, \text{early}(op)}$ can be computed as being the length of the longest chain from an artificial source to $op$ in the acyclic graph valued with $\delta$ (i.e., the dependency graph corresponding to one iteration). Similarly, $L_{\sigma, \text{late}(op)}$ denotes the latest time at which $op$ can be executed in a schedule with a margin $L$. Note that $L_{\sigma, \text{late}(op)}$ depends on $L$ while $L_{\sigma, \text{early}(op)}$ does not. In the scope of $L$, each operation $op$ can be scheduled between $L_{\sigma, \text{early}(op)}$ and $L_{\sigma, \text{late}(op)}$ (figure 1).

$$L_{\sigma, \text{early}(op)} \leq \sigma_{op} \leq L_{\sigma, \text{late}(op)}$$

(6)

Hence, we can reduce the bounds of the sum in (equation 5).

2.2 Problem to be solved

Depending on the application, we may want to solve different kinds of problems. In this paper, we are interested in the two following ones:

- $P$: Given a dependency graph $G$, an initiation interval $h$ and a margin $L$, find a schedule $\sigma$ that minimizes the number of registers $R$ necessary to execute the source code.

- $Q$: Given a dependency graph $G$ and the three parameters $L$, $R$ and $h$, answer the question whether there exists a schedule $\sigma$ with an initiation interval $h$, satisfying dependencies of $G$, of length lower than $L$ and using no more than $R$ registers.
In the next section, we will try to solve the problem \((P)\) and answer the question \((Q)\). In fact, solving \((Q)\) is the central problem. Any optimization problem related to the three parameters \(R, L\) and \(h\) can be solved by a dichotomic search on these parameters. Particularly, \((P)\) can be solved from \((Q)\) by a dichotomic search on \(R\). The reason why we have distinguished \((P)\) is because \(R\) can be expressed as a linear objective function (see section 3).

3 Register allocation

In this section we express the register constraints as linear equations, in order to be integrated in the whole linear programming model. A register edge corresponds to a data flow dependence carried in a register. The register edges are a subset of \(E\), the data dependence edges. In fact, some dependence edges may not have any corresponding register edge. This would be the case for a dependence between a store and a load to an ambiguous or the same memory location. For simplification, in this paper, dependence and register edges are assimilated. Consider the register edge \((op, op')\). The result of \(op\) is written in a variable \(v\) that will be used later by the operation \(op'\). When there is no confusion, we identify \(v\) to the operation \(op\) defining it. \(v\) must stay alive between the scheduling date of operation \(op\) and the scheduling date of the last operation \(op'\) using it. In addition, at any time step \(t\), we have to ensure that the number of variables \(v\) alive does not exceed the number of physical registers of the machine. Hence, the register constraint is:

\[
\forall t \geq 0 \quad \text{Card}\{op \mid \exists(op, op') \in V, \exists i, \sigma_{op} + hi \leq t < \sigma_{op'} + (\lambda + i)h\} \leq R
\]

This equation describes the fact that no more than \(R\) variables are alive at any time. The reason why it is equivalent to the fact than no more than \(R\) registers are required in the register allocation phase relies on properties of interval graphs. In the case of straight line code (basic block), the resulting interference graph is an interval graph that can be
colored with a number of colors equal to the maximum number of overlapping intervals at each time. In the case of loop code, it has been proved that a register allocation can be performed with a number of registers equal to the maximum number of overlapping intervals at each time, at the cost however of a possible unrolling of the loop [14, 16].

The latter equation is unfortunately not linear because of the function Card. We will show how to linearize this constraint, by considering three different cases of graphs.

3.1 Binary tree

This case has been studied in [16]. It concerns a data dependence graph where each variable is used by at most one operation. We note \( \text{out}(op) \) the number of outgoing edges of node \( op \). \( \text{out}(op) \in \{0,1\} \). Classically, basic blocks contain unary/binary operations, thus \( \text{in}(op) \in \{0,1,2\} \) where \( \text{in}(op) \) is the number of incoming edges of node \( op \). For \( t \in [1, L] \) let \( R_t \) be the number of registers required by the schedule \( L_{\text{out}, \text{late}}(op) \). \( R_t \) is the number of registers required by the schedule where every operation is completed as late as possible, but no operation is completed later than time \( L \). Then, by considering the effects of shifting an operation upward, [16] proves that the constraint on the number of registers can be formally written as:

\[
\forall t \in [1, L] \quad R_t + \sum_{op \in O} \sum_{t' = L_{\text{out}, \text{late}}(op)}^{t-1} \left( \text{out}(op) - \text{in}(op) \right) \cdot x_{op,t'} \leq R
\]

(8)

The key point in this formula is that any variable is used only once. Therefore, it is dead as soon as the operation using it is issued.

There are \( L \) such equations.

3.2 Acyclic graph

In the case of a directed acyclic graph (DAG), the previous formulation is not valid anymore. In fact, the upward shifting of an operation using a variable \( v \) may not necessary imply that \( v \) is not alive anymore. A variable dies only when all operations using it have been shifted upward. To model that, we introduce a new variable \( c_{v,t} \) which counts the number of operations using \( v \) and not yet issued at time step \( t \). How are we going to compute the value of \( c_{v,t} \)? Clearly, \( c_{v,t} \) can be non zero if the operation \( op \) defining \( v \) is scheduled at time step \( t' \leq t \), otherwise \( c_{v,t} = 0 \) as the operation defining \( v \) is not yet issued. Its value can be set to the number of outgoing edges from operation \( op \), decreased by the number of operations \( op' \) using \( v \) and issued at a time step \( t' \leq t \). This is expressed in the next linear equation in which \( v \) is identified to \( op \).

\[
\forall v \in O, \forall t \in [1, L] \quad c_{v,t} = \text{out}(op) \cdot \sum_{t \leq t'} x_{op,t'} - \sum_{op' \in E(t \leq t', t')} \sum_{t \leq t'} x_{op', t'}
\]

(9)

There are \( O(nL) \) equations of this type.

We cannot use this counter directly to compute the total number of variables alive at time step \( t \), because each variable should be counted only once, also when there is more than one sample of that variable alive. That is why we introduce another variable \( e_{v,t} \in \{0,1\} \) which expresses whether the variable \( v \) is alive at time \( t \) (\( e_{v,t} = 1 \)) or not (\( e_{v,t} = 0 \)). The value of \( e_{v,t} \) is computed from \( c_{v,t} \): \( e_{v,t} = 0 \) if and only if \( c_{v,t} = 0 \), otherwise \( e_{v,t} = 1 \).

This can be expressed by the following linear equation:

\[
\forall v \in O, \forall t \in [1, L], \quad e_{v,t} \leq c_{v,t} \leq \text{out}(op) \cdot e_{v,t}
\]

(10)
The number of equations of this type is in $O(2nL)$. Finally, the register constraint that the number of alive variables must not exceed the number of physical registers of the machine can be easily written as:

$$\forall t \in [1, L] \sum_{v \in G} e_{v,t} \leq R \quad (11)$$

There are $L$ such inequalities.

**Theorem 1** In the case of a DAG ($h = 0$), for a fixed margin $L$, the problem (P) is equivalent to the linear program composed of the equations (3), (2), (4), (5), (9), (10) and (11), the objective function being “minimize $R$”.

**Proof** Variables $x_{op,t}$ have been defined so that we can write linear register constraints. Equations (3), (4) and (5) express the coherence of this definition as well as the schedule $\sigma_{op}$ in terms of these new variables. It is clear that equation (2) is a linear definition of dependence constraints while equations (9), (10) and (11) are the linear expression of a DAG register constraints. In conclusion, a solution $\sigma$ verifying all these equations and minimizing the physical number of registers $R$ is nothing else than the solution to the problem (P).

**Theorem 2** In the case of a DAG, for fixed $L$ and $R$, the question (Q) is equivalent to the existence of a solution of the linear program composed of the equations (3), (2), (4), (5), (9), (10) and (11), for a fixed $R$.

**Proof** The equations are the same as for theorem 1 except that the value of $R$ is fixed in equation (11); it is a datum of the problem. In this case, our goal is to find a solution, if one exists, for the given $L$, $h$ and $R$.

### 3.3 General case

The general case corresponds to loops; the Loop Dependence Graph (LDG) may be cyclic, due to data dependences between operations from distinct iterations. In order to compute register requirements in this case, we have to handle two facts. On the one hand, some variables may stay alive during more than one iteration ($\lambda > 0$). On the other hand, since iterations may overlap, we must cumulate for a given time step all the variables alive defined in the different iterations. Since each iteration has the same schedule, we consider here only the schedule of the first iteration and the lifetime of variables of the first iteration. We keep in mind that each variable gives rise to corresponding variables in the successive iterations.
The first problem can be solved by controlling the variables defined by the first iteration along a time interval greater than $L$. Let $\lambda_{max} = \text{Max}(\lambda)$. In the worst case, a variable can only be alive before time $L + \lambda_{max}h$. Hence, we adapt the formula (9) for computing the new $c_{v,t}$:

$$\forall v \in O, \forall t \in [1, L + \lambda_{max}h]$$

$$c_{v,t} = \text{out}(op) \cdot \sum_{1 \leq t' \leq t} a_{op,t'} - \sum_{op'\in V} \sum_{1 \leq t' \leq \text{Min}(t-\lambda h, L)} x_{op', t'}$$  \hspace{1cm} (12)

There are $O(n(L + \lambda_{max}h))$ such equations.

Similarly, the inequalities (10) must be extended to the interval $[1, L + \lambda_{max}h]$:

$$\forall v \in O, \forall t \in [1, L + \lambda_{max}h] \quad e_{v,t} \leq c_{v,t} \leq \text{out}(op) \cdot e_{v,t}$$  \hspace{1cm} (13)

There are $O(2n(L + \lambda_{max}h))$ such inequalities.

To take into account the overlapping of many iterations, note that the juxtaposition of different iterations on a time interval $h$ is equivalent to the decomposition of one iteration into slots of length $h$ and putting them side by side. Hence, in the steady state, the number of variables alive at time step $t$ ($t \in [1, h]$) is the sum of variables $e_{v,(t+kh)}$ for $1 \leq t + kh \leq L + \lambda_{max}h$.

$$\forall t \in [1, h] \quad \sum_{k=0}^{\lfloor \frac{t}{h} + \lambda_{max} \rfloor} \sum_{v \in O} e_{v,(t+kh)} \leq R$$  \hspace{1cm} (14)

Theorem 1' In the case of an LDG, for a fixed Initiation Interval $h$ and a fixed margin $L$, the problem $(P)$ is equivalent to the linear program composed of the equations (3), (2), (4), (5), (12), (13) and (14), the objective being “minimize $R$”

Theorem 2' In the case of an LDG, for fixed $h$, $L$, $R$, the question (Q) is equivalent to the existence of a solution for the linear program composed of the equations (3), (2), (4), (5), (12), (13) and (14), for a fixed $R$.

These two theorems are the generalization of Theorem 1 and Theorem 2 in the case of an LDG. The difference is that equations (9), (10) and (11) have been substituted by equations (12), (13) and (14) adapted to the loop case. The proofs are similar.

Complexity issues and the minimization of the number of equations and variables are discussed in a technical report.

4 Experiments

4.1 Execution time

In order to validate our approach, we have implemented a generator of linear programs from a loop and architecture specification. The first goal of our experiments was to test the feasibility of this approach by measuring the time spent before a solution is given, as well as the practical influence of the margin $L$ and the initiation interval $h$ on the minimum register requirement $R$.

We applied our system to 12 benchmark loops corresponding to various scientific programs. The latencies for the various instructions are shown in Table 1 and are integrated in the LDG representation of the loop. All these examples as well as the latencies of corresponding operations are taken from [18]. Graphically, for a given edge, the dependence distance is equal to 1 if the corresponding latency is encircled, otherwise it is equal to 0.
<table>
<thead>
<tr>
<th>Instructions</th>
<th>Add</th>
<th>Subtract</th>
<th>Negate</th>
<th>Multiply</th>
<th>Divide</th>
<th>Load</th>
<th>Store</th>
<th>Copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Clock cycle(s)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>17</td>
<td>2-13</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Latencies of instructions

The machine model is supposed to have enough functional units so that no resource constraints need be generated. Of course, it is an idealistic hypothesis, but we stress that our aim in these experiments is to validate our approach and not to have a realistic schedule.

Using the LDG, our program generates the ILP formulation which is solved using the Mixed Integer Program solver lp_solve. To start up our experiments, let us treat the following Livermore fortran loop (kernel9):

```plaintext
for i=1 to n do
    s = s + a[i]
    a[i] = s * s * a[i]
enddo
```

The assembly code of the above loop body can be written as:

1. t0 = t0 + 1
2. t1 = a[t0]
3. s = s + t1
4. t2 = s * s
5. t3 = t1 + t2
6. a[t0] = t3

By applying our system for a margin L = 20 we generated 180 variables and 200 linear equations. For an initiation interval h = 2, a solution is given for a minimum register requirement R = 21. By increasing h to 6, a solution was possible for R = 8. See figure (3). For the latter case, register lifetimes are shown in figure (4). Our tests enable us to plot the curve of the minimum register requirement R in terms of the initiation interval h, see figure (5).

A subset of our tests on the remaining 12 examples represented by their LDG is recapitulated in table (2) and figure (6).

Table (2) reports the parameters L, h and R used in our experiments, the answer to the existence of a solution, as well as the computing time required to obtain the answer. When nothing is indicated, this means that the computing time was too short to be measured. In the other cases, time is given in hh:mm:ss, where hh stands for hours, mm for minutes and ss for seconds. Although the size of automatically generated programs may be very large, the computing complexity remains acceptable in many cases. Only for three tests, did the solver collapse or was interrupted after a long time.

Clearly our solver could not be incorporated directly in a compiler, due to the uncertainty on termination. There are however different ways it can be used. We can for instance set an upper bound of computing time, stop the computation at that time and keep the solution obtained so far by the solver itself or by a heuristic method. Our solver can also be used for optimizing library code, compiled once for all. It should be noted that the solver lp_solve we use is not at all specifically targeted to our problem. By adapting the underlying algorithm to the specificities of our problem, computing time may be shortened. Giving a solution as a starting point to the LP solver would also probably improve the performance.
Figure 3: Schedule and interaction between $h$ and $R$

Further experiments still need to be done to identify the critical parameters of the solver and their influence on the computing time. Also a driver that orchestrates the search of adequate $h$, $R$, and $L$, still needs to be analyzed and designed.

4.2 One possible application

Another example application of our solver is the analysis of architectural parameters. For instance, we have computed the minimal register requirements of some test loops, as a function of Initiation Interval $h$ (figure (6)). From these results, we can deduce that 16 registers are enough to execute our test loops at the optimal execution rate (minimal $h$). If only 8 registers are available in the machine, then one loop (spec - spice - loop2) can execute at best at the rate of 3 clocks cycles per iteration, instead of 1 clock cycle in the optimal case. With 8 registers, no schedule was found for the loop livloop23, so that variables spilling should be considered.

4.3 Comparison with buffer size minimization

In [9] a linear formulation for register allocation was suggested, based on the total size of buffers required for storing the variables with lifetimes spanning more than one iteration. In this paper, it was argued that the criterion of buffer size was a good approximation of the number of registers required. A number of experiments have been done in [19] for estimating the buffer size.

For verifying the validity of that assertion, we have extended our solver by incorporating the resource constraints as described in [8].

We have run experiments on the same set of 27 test loops, with the same architectural constraints (1 adder, 1 multiplier, 1 divider, 1 load/store unit with latencies respectively equal to 1 cycle, 2 cycles, 17 cycles, 2 cycles for load and 1 cycle for store). We first determine the minimal $h$ and then determine the minimal number of registers required for executing the loop at that rate. The results are presented on the figure (7), where $h$ is the Initiation Interval, $R$ the number of registers required and $B$ the total buffer size. The first column describes our results whereas the other columns are taken from [19], where they compare their method with other heuristics for software pipelining: Huff
Figure 4: Register lifetimes for the case $h = 6$, $R = 8$

Slack Scheduling [6], DESP FRLC [20] and modified list scheduling of [21].

In average, the optimal schedules that we have generated require 3.6 registers less than when only counting the buffer size. In some cases, it can be observed that the other methods result in a lower buffer size. This is because they find a greater $h$ and thus a lower execution speed.

5 Related work

ILP formulation for loop software pipelining is a convenient way of handling this optimization problem globally. Probably because of the progress made in the ILP solvers, a number of such formulations have been suggested recently.

When the simple problem with no resource or register/memory constraints is addressed, the problem is trivially linear with respect to the issue dates, when the initiation interval is fixed.

As for the resource constraints, C. Hanen [12] suggested an interesting formulation for describing disjunction constraints between two operations: specifying that two operations can not share the same resource amounts to finding a slot such that the second operation is scheduled between two iterations of the first one. The complexity of that formulation remains low since the unknowns are the issue dates of the first iteration and one variable per disjunction constraint. However, this formulation can only handle the case of one single resource of each type. More recently, Peautier [22] used binary variables for expressing the constraint that no more than $p$ functional units of a given type are available. [18] used also binary variables for specifying more complex resource constraints such as reservation tables. Also in the domain of VLSI design, binary variables are used for specifying complex resource constraints.

The problem of registers/memory constraints formulation was first addressed by [9] where the total buffer size was taken as the optimization criterion. In the domain of VLSI design [23], binary variables are used for expressing memory (cache) constraints. Instead of using our $c_{v,t}$ counter variable, they model $e_{v,t}$ directly by inequalities specifying that $e_{v,t}$ is non zero while some operation $op$ using $v$ is not yet scheduled. This replaces $2nL = L \sum_{op} 2$ inequalities by $L \sum_{op} out(op)$ ones and therefore increases slightly
the complexity that now depends on the number of successors of operations. A similar approach is given in [17], but they consider the steady state of the loop (h consecutive cycles) instead of the scheduling of one iteration. Hence the latency $L$ of one iteration does not appear in the complexity, that depends only on $h$. Based on the same idea as Decomposed Software Pipelining [24], the issue date of an operation is determined by its cycle number (row-number) in the steady state and a shifting factor expressing to which original iteration it corresponds (the column number). One drawback of this formulation is that the search space for the unknowns is unbounded. This may increase the computation time when solving the system. A last formulation was suggested by [25]. They decouple the problem into two steps. First find the column numbers by retiming [26], then fix the column numbers and solve the ILP on the steady state. A branch-and-bound framework is proposed for globally solving the problem. Finally, in his thesis, E. Altman presents a linear formulation that globally models loop software pipelining and register assignment. Whereas in our framework, the condition is that no more than $R$ variables are alive simultaneously (and the register assignment phase is postponed, at the price of a possible loop unrolling), Altman uses variables that model the fact that a variable $v$ be allocated to a (named) register $R_p$. The number of these variables is $nR$. This results in a far higher complexity.

6 Conclusion

In this paper, we have proposed a linear integer programming formulation to solve exactly the problem of scheduling and register allocation together in the same framework. This exact formulation can serve for many applications in compilers as well as in architecture design. On the one hand, we can construct the software pipelined schedule that requires the minimum number of registers for a given initiation rate, at least in theory. On the other hand, we can generate the optimal schedule and the minimum initiation interval
<table>
<thead>
<tr>
<th>Loop</th>
<th>h_min</th>
<th>L_min</th>
<th>L_max</th>
<th>L</th>
<th>h</th>
<th>R</th>
<th>Solution</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>liv-kernel</td>
<td>12</td>
<td>20</td>
<td>72</td>
<td>20</td>
<td>1</td>
<td>49</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>1</td>
<td>39</td>
<td>Ne</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>2</td>
<td>21</td>
<td>Yes</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>2</td>
<td>20</td>
<td>Ne</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>84</td>
<td>20</td>
<td>3</td>
<td>20</td>
<td>Yes</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>3</td>
<td>18</td>
<td>Yes</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>3</td>
<td>14</td>
<td>Ne</td>
<td>03:06</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>87</td>
<td>20</td>
<td>4</td>
<td>13</td>
<td>Yes</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>4</td>
<td>10</td>
<td>No</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>5</td>
<td>10</td>
<td>Yes</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>5</td>
<td>9</td>
<td>Ne</td>
<td>01:12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>17</td>
<td>4</td>
<td>Ne</td>
<td>00:53</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>18</td>
<td>4</td>
<td>Yes</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>18</td>
<td>3</td>
<td>Ne</td>
<td>00:11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>20</td>
<td>3</td>
<td>Ne</td>
<td>00:15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hiddot</td>
<td>11</td>
<td>5</td>
<td>11</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>No</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>No</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>Yes</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>No</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>2</td>
<td>3</td>
<td>Ne</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>Yes</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>No</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>Ne</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>4</td>
<td>2</td>
<td>No</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>liv-loop</td>
<td>8</td>
<td>9</td>
<td>50</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>No</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>4</td>
<td>4</td>
<td>No</td>
<td>14:36</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>9</td>
<td>5</td>
<td>5</td>
<td>Yes</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>9</td>
<td>5</td>
<td>5</td>
<td>Yes</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>5</td>
<td>4</td>
<td>Ph</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>9</td>
<td>6</td>
<td>4</td>
<td>Yes</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>8</td>
<td>3</td>
<td>Ne</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>9</td>
<td>6</td>
<td>3</td>
<td>Ne</td>
<td>00:36</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>74</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>Yes</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>7</td>
<td>3</td>
<td>Ne</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>82</td>
<td>9</td>
<td>8</td>
<td>4</td>
<td>Yes</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>70</td>
<td>8</td>
<td>3</td>
<td>Ne</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Corresponding to a given number of physical registers.

Our preliminary experiments show that our approach is tractable in complexity in general, although the size of generated linear programs may be very large.

Simultaneously to this work, a number of similar approaches have been designed. We believe these different approaches should be experimentally compared and possibly combined. The fact that a lot of efforts are done in the domain of Integer Linear Programming lets us inclined to think that these approaches will become very popular in the immediate future.

References


<table>
<thead>
<tr>
<th>Loop</th>
<th>$L_{min}$</th>
<th>$L_{max}$</th>
<th>$L$</th>
<th>$h$</th>
<th>$R$</th>
<th>Solution</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>liv-loop</td>
<td>2</td>
<td>5</td>
<td>22</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>3</td>
<td>2</td>
<td>No</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>4</td>
<td>2</td>
<td>No</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>4</td>
<td>2</td>
<td>Yes</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>4</td>
<td>1</td>
<td>No</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>liv-loop2d</td>
<td>8</td>
<td>13</td>
<td>192</td>
<td>13</td>
<td>8</td>
<td>11</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>8</td>
<td>11</td>
<td>Yes</td>
<td>05:23</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>8</td>
<td>10</td>
<td>Yes</td>
<td>22:46</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>9</td>
<td>10</td>
<td>Yes</td>
<td>09:46</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>9</td>
<td>9</td>
<td>No</td>
<td>32:36</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>9</td>
<td>9</td>
<td>Pb</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>10</td>
<td>9</td>
<td>Yes</td>
<td>53:07</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>10</td>
<td>8</td>
<td>No</td>
<td>2:06:45</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>11</td>
<td>8</td>
<td>Pb</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>11</td>
<td>8</td>
<td>Pb</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>23</td>
<td>108</td>
<td>22</td>
<td>2</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>22</td>
<td>6</td>
<td>Yes</td>
<td>13:44</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>22</td>
<td>4</td>
<td>No</td>
<td>08:52</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>22</td>
<td>4</td>
<td>No</td>
<td>22:08</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>24</td>
<td>4</td>
<td>Pb</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>24</td>
<td>3</td>
<td>Yes</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>24</td>
<td>2</td>
<td>No</td>
<td>00:17</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>22</td>
<td>2</td>
<td>No</td>
<td>00:23</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>24</td>
<td>2</td>
<td>No</td>
<td>00:35</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>26</td>
<td>113</td>
<td>21</td>
<td>3</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>20</td>
<td>3</td>
<td>No</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>304</td>
<td>26</td>
<td>20</td>
<td>3</td>
<td>Yes</td>
<td>00:16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>21</td>
<td>2</td>
<td>No</td>
<td>05:34</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>21</td>
<td>3</td>
<td>Yes</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>35</td>
<td>21</td>
<td>2</td>
<td>No</td>
<td>05:35</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>35</td>
<td>24</td>
<td>2</td>
<td>No</td>
<td>10:04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>spec-fp-loop</td>
<td>29</td>
<td>21</td>
<td>150</td>
<td>21</td>
<td>20</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>22</td>
<td>1</td>
<td>No</td>
<td>00:09</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>Yes</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>No</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>Yes</td>
<td></td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Tests


Figure 6: variation of the minimum register requirement $R$ in terms of $h$


<table>
<thead>
<tr>
<th>Branches</th>
<th>h</th>
<th>R</th>
<th>Geo</th>
<th>Hoff</th>
<th>Dep</th>
<th>Gasteroni</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Float</td>
<td>6</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Float</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Float</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Float</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Float</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>Float</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Float</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Float</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Float</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Float</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Float</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
</tbody>
</table>

Figure 7: Comparing with [19]


SALTO: System for Assembly-Language Transformation and Optimization

Erven Rohou, François Bodin, André Seznec
{erohou,bodin,seznec}@irisa.fr

Abstract

On critical applications the performance tuning requires multiple passes. SALTO (System for Assembly Language Transformation and Optimization) is a retargetable framework for developing all the spectrum of tools that are needed for performance tuning on low-level codes (assembly-languages). SALTO enables the building of profiling, tracing and optimization tools. The user is responsible for giving a machine description of the target architecture, which includes instruction-set of the processor, precise hardware configuration and reservation-tables for all instructions. High-level functions are provided for writing any tool corresponding to his needs. Moreover SALTO will be a part of a global solution for manipulating assembly-code to implement low-level code restructurization as well as to provide a high-level code restructurer with useful information collected from the assembly code and from instruction profiling.

1 Introduction

The increasing usage of high-performance embedded systems based on RISC/VLIW architectures has highlighted the need for tools that allow the easy implementation of fine-grain parallelism optimizations, assembly-code profiling and instrumentation, along with an accurate description of the target architecture and high retargetability.

SALTO is a first step towards the availability of a retargetable framework for developing the whole spectrum of tools that manipulate assembly-language. Building these tools typically implied a time consuming development phase (rewriting from scratch large pieces of code was often needed) and required much investment. The objective of such a system is to provide the user with a single environment that will automatically perform boring and repetitive tasks and allow him to implement the tools needed for performance tuning on low-level codes.

SALTO overcomes many limitations of previous solutions: it does not implement any algorithm by itself, but its scope is broad enough to allow implementation of either profiling or optimization techniques. A large amount of work needed while working at the assembly-code level (code parsing, construction of the dependence graph, etc.) is performed automatically by the system. To the user, SALTO provides an object-oriented interface to deal with assembly-code. The objects contain a complete description of the control-flow graph (CFG) of the program and a model of the target architecture. They are easily accessible through the user interface and provide a convenient way to implement algorithms without having to worry about infrastructure.

Section 2 reviews related works and points out how our tool provides a more general and integrated solution for building tracers, profilers and optimizers. An
overview of the features of the system is given in section 3. Section 4 shows an example and section 5 concludes this paper.

2 Related Work

To our knowledge no available single environment deals with the whole spectrum of code restructuring and execution profiling and tracing. Moreover most tools are not retargetable.

Much work has been done in the field of low-level code optimization and analysis of dynamic program execution. Optimization can be done at different levels of granularity. They range from code transformations for improving the parallelism in each basic-block to procedure motion for improving the instruction-cache behavior. Dynamic program execution can be analyzed by profiling (particularly interesting for determining critical sections on which to focus) whereas tracing also furnishes the order of instructions and addresses.

2.1 Analysis

Many techniques for analyzing program behavior depend on instrumentation of either the source code or the executable file. PIXIE [20] is an instrumentation utility running on MIPS machines for executables. The instrumented program, which contains additional code, counts the execution of each basic-block. A counter is added in each block. Ball and Larus [3] studied how to improve the technique by minimizing the number of counters required. They build a CFG of the procedure being instrumented and compute the maximal spanning tree before deciding where to add code. They proved this technique to be optimal. They implemented their strategy in a tool called QPT [3].

A more flexible solution called ATOM [9, 21] was proposed for the DEC Alpha. ATOM also depends on code instrumentation and provides common analysis and performance tools. A partial list given in [9] contains instruction profiling, system-call summary, memory checker, and many others. The major advantage is the ability for the user to implement instrumentations in the C language enabling a high degree of flexibility. ATOM can be seen as a library of predefined functions that ease the instrumentation of the code.

Larus and Schnarr [18] have built a library called EEL (Executable Editing Library). EEL enables the building of tools to analyze and edit compiled programs without having to worry about the underlying machine and operating system. EEL analyzes an executable and builds some abstractions like routine, CFG, instruction and snippets. A snippet is a piece of code to be included in the original executable for instrumentation purpose (when saving the registers state, calling a foreign function and restoring the previous values). This code is necessarily specific to each particular machine and cannot be avoided when adding code to an executable. It is similar to the architecture dependent code we add to an assembly source code.

Gordon Irlam's Spy [16] runs on Sparc architectures under SunOs 4.x. Spy exploits special features of the Sparc microprocessor [8] to fetch the instruction to be executed. Spy provides as output a trace composed of the instruction and data addresses. Other tools achieve analysis via instruction-set architecture (ISA) emulation.
Cmelik and Keppel chose another strategy with SHADE [7]: they dynamically compile each instruction of the program, i.e., they build a block semantically equivalent to each assembly instruction of the original code as if it were a complex instruction and then execute the block. This approach enables them to simulate an ISA on a different architecture (they can currently simulate Sparc and MIPS code on Sparc systems).

A survey of trace-driven memory simulation including trace collection, trace reduction and trace processing can be found in [23].

2.2 Code Generation and Optimization

Extensive work has been done on fine-grain optimization [1, 2, 4, 6, 10, 11, 14, 17] but very few on frameworks for enabling a fast implementation of such optimizations. Furthermore, we are not aware of a retargetable system that enables the implementation of assembly-code scheduling (with accurate resource usage models), partial-register allocation and instrumentation. Being able to “redo” partial-register allocation is a major capability that is necessary for achieving good performance when using software-pipeline techniques [2].

The main drawback of having SALTO as a separate tool is that it separates the code-generation and the optimization. Ideally, we want these phases to happen simultaneously [6] to be able to generate optimal code. In practice, mixing code-generation and scheduling (especially software-pipeline and global-scheduling techniques) is a very difficult task. Having code generation and scheduling performed at the same time, generally, results in having a non-optimized code-generator and a weak code scheduler that does not implement anything beyond basic-block scheduling. In the remainder of this section we overview works that we believe are close to SALTO.

Code layout optimization is performed by CORD on MIPS based computers. CORD rearranges procedures in an executable to reduce paging and achieve better instruction-cache mapping. It uses a feedback file generated at run-time by an executable instrumented by Pixie.

The university of Karlsruhe has developed the BEG system (Back-End Generator) [12]. It produces a code-generator from rule based declarative descriptions. Basic blocks are reordered before and after global register allocation. The system is based on a simulation of the pipeline during instruction reordering. Pipelines that schedule more than one instruction per cycle are also supported. BEG can also produce local register allocators suitable for a great variety of target machines.

MARION [5] is a code-generator generator been developed at the University of Washington. The MIPS R2000, Motorola 88000 and i860 architectures have been described with the specific language Maril. The hardware description includes registers, functional units, multiple issue and pipeline stages. Instructions are given properties that influence reordering. The description is based on reservation tables and on delays required to obtain the result of an instruction. MARION has been mainly used to study the interaction between reordering and register allocation [6]. Compared to SALTO, no user interface is provided to modify assembly-code and we believe that it cannot be used with compilers other than cc.

GCC is a C compiler maintained by the Free Software Foundation, Inc. [22]. It uses the internal representation RTL. The good performance achieved by GCC comes from its ability to apply the right optimization at the right moment within the compilation
process. It is improved by a last optimization specific to the target architecture at the end of the process. Instruction reordering within the basic-blocks is performed twice, once before and once after the register allocation stage. The machine description includes the size of delay slots and the annulation conditions, the time needed to compute the result of an instruction and resource conflicts between instructions.

A compiler system named SUIF [15] (which stands for Stanford University Intermediate Format) is being developed at the Stanford University. It is built on top of a kernel that preprocesses source code and produces a representation of the program called an intermediate format. This format contains low-level description of the structures without losing useful high-level information like arrays, if-then-else structures and loops. Many passes are performed on this representation, implementing a single optimization or transformation at a time. The compiler performs several passes on this representation, but insertion of new passes can easily be made. Features are provided to allow implementation of algorithms ranging from data-dependence analysis or register allocation to symbolic analysis or detection of parallelism.

3 System Overview

SALTO is based on the former tool OCO developed at IRISA which was written using parts of the code parser from the GNU assembler GAS. A more complete description of SALTO can be found in [19]. SALTO is composed of three parts: a kernel, a machine description file and an optimization or instrumentation algorithm. Figure 1 illustrates the organization of these three components.

1. The kernel performs common required tasks that the user doesn’t want to worry about like parsing the assembly-code and the machine-description file or the construction of the internal representation (see section 3.1). This internal representation is available via the user interface (see section 3.3).

2. The machine description file contains the hardware configuration and the complete description of the ISA with reservation-tables. Section 3.2 details the format of this file.
3. The optimization or instrumentation algorithm is supplied by the user. Once
the system has read the machine-description file and the assembly-code, an
internal representation is built and control is given to the user through the call
to a predefined function.

In this section we summarize the main features of SALTO. It is built on three com-
ponents, the data structures for representing the program, the machine description
for declaring resources usage and finally the user interface that enables the writing of
instrumentation or scheduling algorithms.

3.1 Data Structures

The data structures used in SALTO are divided into two groups, depending on their
role. The first group represents the control flow of the program, the second group
describes resource usage and data dependencies between instructions.

3.1.1 Control Flow Structures

While parsing the code, SALTO builds the list of the procedures it encounters. Each
procedure has a list of basic-blocks, and each block “knows” its list of instructions.
Additionally, the CFG is built for each procedure. The vertices are the basic-blocks
and the edges denote the execution order of the basic-blocks. Edges are labeled to
indicate if they correspond to the taken or not-taken branch. See figure 2 for an
element of the CFG corresponding to a simple procedure written in C language and
compiled on a Sparc workstation. A known limitation is due to the static analysis
of the assembly-code. If the target address of a branch instruction is a computed
register value, SALTO assumes every block of the current procedure can potentially
be the target. However, the basic-blocks are constructed assuming computed branches
can only target labels (this may lead to incorrect basic block computation in some
circumstances).

3.1.2 Hardware Resources and Data Dependencies

The second part of the data structures provided by SALTO gives information about
the resources needed by an instruction to complete execution. A resource is usually
a register, a functional unit or the memory, but it could be any piece of hardware
needed to describe the behavior of the machine (see section 3.2 for an explanation on
how to define resources). Each instruction needs a resource during a number of cycles
with a particular access mode: read, write or use.

Each instruction is described by a reservation-table, which indicates the list of
resources it needs and the mode and cycle a resource is accessed. This information is
used when determining the type of dependence between two instructions: RAW (read
after write), WAW (write after write), WAR (write after read).

The memory is currently seen as a unique resource and all memory accesses are
considered to be to the same memory location. SALTO is conservative when checking
data dependencies and thus two memory accesses, one of which is a write, always
leads to a dependence. However, the functions of the user interface make it possible
for the user to write his own alias analysis algorithm to detect such situations and to
implement a link with the compiler data dependence analysis.
/* Computes the GCD of two integers using Euclid's method */

int gcd(a, b) {
    int a, b;
    
    int result;
    
    if (a>b) a^=b, b^=a, a^=b;
    if (a%b) result = gcd(a-b, b);
    else result = b;
    return result;
}

Figure 2: Control Flow Graph of a Simple Program

3.2 Machine Description

SALTO is designed to be a retargetable tool. Thus, the target machine must be described in a flexible way, permitting an accurate description while retaining the ability to easily modify parameters. This is achieved with a Lisp-like language based on the reservation-tables formalism. The description file is parsed by SALTO and an internal representation is built using RTL (Register Transfer Language) [22]. The machine description contains:

1. the syntax of the assembly-language used;
2. all the resources needed for the computation of data dependencies;
3. the list of the instructions recognized by the assembler with the applicable formats and the associated reservation-tables;
4. semantical information to warn SALTO about special features implemented in the processor like bypass-mechanism or delayed branch.

The following paragraphs describe more precisely the different steps involved in writing a machine-description file.

3.2.1 Definition of the Available Resources

The language describes the available resources as precisely as desired. A high-level description including only an integer unit, a memory access unit and a floating point unit may be enough to perform local reordering techniques. A lower-level description, with bus access and cache memories is also possible. Three types of resources
are currently recognized: registers, memory and functional units. SALTO also supports symmetric superscalar architectures: any functional unit can be replicated any number of times. Figure 3 shows an example of such a description for the Sparc architecture [8].

3.2.2 Instruction Set

The definition of an ISA includes a declaration of the instruction names, all the applicable formats, and the associated reservation-tables. These reservation-tables are used when resolving data dependencies and resource conflicts. They store information about which functional units are needed by an instruction, the first and last cycle it is required and when the result is written.

Although the reservation-table formalism we choose to use to describe instructions is almost always sufficient, it has some limitations which prevent it from describing special features of particular processors. The push pipeline instruction of the Intel i860 is an example of such a restriction. The tool can still be used, however in this case the particular behavior is handled at the scheduling algorithm level.

Figure 4 is extracted from our machine description file for the Sparc architecture. The tables are first defined as macros and then used.
3.2.3 Semantical Information

Some instructions of an ISA might have a particular behavior. "Semantic" properties can be added to these instructions. In the case of the Sparc processor, this concerns delayed branch: the instruction immediately following a branch is speculatively executed, and thus is part of the same basic-block.

3.3 User-Interface

The object-oriented user interface provides a flexible way to deal with the internal data structures. Features of SALTO include:

- access to the code at three different levels: procedure, basic-block or instruction; with possibility of modification: insertion, deletion;
- unparsing;
- access to the reservation-tables and computation of dependencies and delays between instructions caused by pipeline stalls;
- addition of attributes: attributes are a flexible way to put any kind of information on a particular basic-block or instruction.

The most important classes defined by the interface are: CFG, BB, INST which represent respectively a procedure, a basic-block and an instruction, RES which represents a user-defined hardware resource, and TRES which implements a reservation table. The additional class SaltoAttribute permit any kind of information to be attached to any particular block or instruction. SALTO attributes are similar to the attributes in the Sage++ system [13].

Figure 5 illustrates the use of the user interface with a simple example. A more complex example is given in section 4.

// Renames register reg.s into reg.d for instruction inst.
// regs and reg.d are obtained for example using RES *getResbyName(char *)
void rename(INST *inst, RES *reg.s, RES *reg.d) {
    int ninput, noutput, i;
    ninput = inst->numberOfInput();
    for(i=1; i <= ninput; i++)
        if (inst->getInput(i) == reg.s)
            inst->setInput(i, reg.d);
    ...

Figure 5: Use of the User-Interface
4 An Example: Local Reordering with SALTO

To illustrate SALTO usage, this section presents an application in greater detail: a local scheduler. This example gives an idea of the range of tools that can be implemented using SALTO, but is not intended to be representative of state-of-the-art scheduling techniques. As an example of local reordering we have implemented a list-scheduling algorithm using SALTO. Our program is shown in appendix A. The main function is reorder. It builds the dependence matrix dep[i][j] for the instructions of the current block. The main loop computes the scheduling cycle for each instruction until a branch is seen: verify_predecessors checks if all instructions that have a data dependence have already been scheduled. earliest_cycle then computes the delays before all needed results are obtained. IsConflict is used to detect resource conflicts. The branch instruction, if it exists, and the delay slot are processed afterwards. The blocks are effectively reordered by orderAccordingToCycles and nops are added if necessary by addNecessaryNops to fill the empty slots.

5 Conclusion

Performance tuning of critical applications is a major issue that cannot be done by a one-pass compiler. For many applications, e.g. embedded applications, one can afford to pay long performance tuning costs across a multiple-pass code generation.

SALTO is a framework for developing a large range of tools dealing with performance tuning and optimizations of low-level codes. The major advantages of SALTO compared with already existing tools is its ability to generate performance-analysis tools as well as optimization tools and its retargetability towards any ISA.

The main goals of SALTO are to be part of a global solution for manipulating assembly-code to implement low-level code restructuring as well as to provide a high-level code restructurer with useful information collected from the assembly code and from instruction profiling. To achieve this, we implemented several features that ease integration within a compilation process:

- a high-level user interface which allows powerful transformations (ability to analyze, profile and optimize a program) of the assembly-code in a simple manner using a single tool;
- the possibility to feed gathered information to an analyzer with a simple procedure call, which is far more effective than using temporary files and overcomes the problem of large traces (which typically measure in gigabytes);
- to give the user full control over the modifications he adds to the original program;
- to provide a highly-retargetable tool by means of accurate description of the hardware and of the assembly-language used, giving full access to the resources needed by an instruction.

The two examples presented in the paper and a variety of others have shown the usage of our system. SALTO is yet robust enough to be used on real applications. We expect it to be available soon by FTP at address ftp.irisa.fr. We strongly
believe that a SALTO-like framework is a major software piece that is required in hardware/software codesign of dedicated versions of processors where not only time-to-market, but also software development cost is critical.

References


A  Example: Optimization with List Scheduling Algorithm

#include "salto.h"

int verify_predecessors(int verif, int **dep, INST **inst) {
    for (int i=0; i<verif; i++)
        if ((dep[verif][i]) && (inst[i]->getCycle() < 0)) return 0;
    return 1;
}

int earliest_cycle(Int s, int **dep, INST **inst) {
    int i, z, max = 0;
    for (i=0; i<s; i++)
        if (dep[s][i]) {
            z = inst[i]->getCycle() + inst[s]->getDelay(inst[i]);
            if (z>max) max = z;
        }
    return max;
}

void build_dep_matrix(int **dep, INST **inst, int n) {
    for (int i=0; i<n; i++)
        dep[i][i]=0;
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            dep[i][j] = (inst[i]->IsDep(inst[j]) != 0);
}

INST **instr;
int **dep;

void reorder(BB *bb) {
    int i, to_be_scheduled, cycle_min, nasm, offset, branch_seen, brindex, last_cycle;
    TRES *res_table = new TRES;  // need a reservation table
nasm = bb -> numberOfAsm();
instr = new (INST *)[nasm]; // to avoid calling getAsm each time we need it
for (i=0; i<nasm; i++) instr[i] = bb -> getAsm(i+1);

build_dep_matrix(dep, instr, nasm); // build the dependence matrix
branch_seen = last_cycle = 0;

to_be_scheduled = nasm; // number of instructions to be scheduled
while (to_be_scheduled && !branch_seen) { // before a branch instruction
    for (i=0; i < nasm; i++) {
        if (instr[i] -> isCTI()) {
            brindex = i;
            branch_seen = 1;
            break;
        }
    }
    // Is this instruction already scheduled?
    if ( instr[i] -> getCycle() < 0 ) {
        // Are all the predecessors scheduled?
        if (verify_predecessors(i, dep, instr)) {
            // Wait for data dependencies to be resolved
            cycle_min = earliest_cycle(i, dep, instr);
            // Now wait for resources to be available...
            offset = 0;
            while (res_table -> IsConflict(instr[i], cycle_min + offset)) offset++;
            // Mark resources occupancy into reservation table
            res_table -> markRes(instr[i], cycle_min + offset);
            if (cycle_min + offset > last_cycle) last_cycle = cycle_min + offset;
            // Specify the cycle
            instr[i] -> setCycle(cycle_min + offset);
            to_be_scheduled--;
            break;
        }
    }
    // The case of the branch instruction
    if (branch_seen) {
        instr[brindex] -> setCycle(last_cycle + 1);
        to_be_scheduled--;
    }
    // The delay slot instruction, if any
    if (to_be_scheduled) instr[nasm-1] -> setCycle(last_cycle + 2);

    // Reorder according to values specified by setCycle()
    bb -> orderAccordingToCycles();
    bb -> addNecessaryNops();
    delete res_table;
    delete instr;
}
Data and Loop Transformations
Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs

Alain Darte and Frédéric Vivien *
Laboratoire LIP, URA CNRS 1398
Ecole Normale Supérieure de Lyon
F - 69364 LYON Cedex 07
{darte,fvivien}@lip.ens-lyon.fr

Abstract

This paper proposes an optimal algorithm for detecting fine or medium grain parallelism in nested loops whose dependences are described by an approximation of distance vectors by polyhedra. In particular, it is optimal for direction vectors, which generalizes Wolf and Lam's algorithm to the case of several statements. It relies on a dependence uniformization process and on parallelization techniques related to system of uniform recurrence equations.

1. Introduction: a motivating example

Consider the following simple code:

Example 1

DO i = 1, n
   DO j = 1, n
      a(i, j) = a(j, i) + a(i, j - 1)  (Statement S)
   ENDDO
ENDDO

The exact dependence relations are the following:

\[
\begin{align*}
  &\text{if } 1 \leq i \leq n, 1 \leq j < n \quad S(i, j) \xrightarrow{\text{flow}} S(i, j + 1) \\
  &\text{if } 1 \leq i < j \leq n \quad S(i, j) \xrightarrow{\text{flow}} S(j, i) \\
  &\text{if } 1 \leq i < j \leq n \quad S(j, i) \xrightarrow{\text{anti}} S(i, j)
\end{align*}
\]

Let us apply well known parallelization algorithms to this code, Allen and Kennedy's algorithm (that uses levels of dependences), Wolf and Lam's algorithm (that uses direction vectors), Darte and Robert's algorithm, and Feautrier's algorithm (that both use exact dependences). Figure 1 shows the corresponding (reduced)

dependence graphs when dependence edges are labeled respectively with levels and direction vectors.

![Reduced Dependence Graph](attachment:image)

Figure 1. Reduced Dependence Graph for Example 1: (a) by levels, (b) by direction vectors.

Allen and Kennedy [1] The basic technique of the algorithm is the decomposition of the reduced dependence graph into strongly connected components. Here, the levels of the three dependences are respectively 2, 1, and 1. There is a dependence cycle at depth 1 and at depth 2. Therefore, no parallelism is detected.

Wolf and Lam [17] The algorithm is based on a cone separation technique adapted to the case of direction vectors. Here, the respective dependence vectors are (0, 1), (+, -), and (+, -). In the second dimension, the I and the -- prevents to detect two levels of fully permutable loops. Therefore, the code remains unchanged. No parallelism is detected.

Darte and Robert [3, 4] Darte and Robert look for an affine schedule for each statement that satisfies all dependences. Exact dependence analysis is needed, and a quite large linear system (obtained by the duality theorem of linear programming) has to be solved. This technique leads to the valid schedule \( T(i, j) = 2i + j - 3 \). One level of parallelism is detected.

*Supported by the CNRS-INRIA project ReMaP.
Feautrier [10, 11] Feautrier’s technique is similar to Darte and Robert’s technique for the one-dimensional case (except that the linear program is obtained by the affine form of Farkas’ lemma). Here, the same schedule is found, \( T(i, j) = 2i + j - 3 \). However, Feautrier’s algorithm is more general, since it is able to derive multi-dimensional affine schedule when no one-dimensional schedule exists. So far, Feautrier’s algorithm is indeed the most powerful algorithm for parallelism detection in nested loops.

In this particular example, the representation of dependences by level and by direction vectors is not accurate enough to reveal parallelism, this is the reason why Allen and Kennedy’s algorithm, and Wolf and Lam’s algorithm are not able to detect any parallelism. Exact dependence analysis, associated to linear programming methods that require to solve large parametric linear programs to be solved, reveals some degree of parallelism. The corresponding parallelized code is:

```
DO j = 1, 3n
  DOPAR i = max(1, \( \lfloor \frac{j - 2}{3} \rfloor \)), min(n, \( \lfloor \frac{j - 2}{3} \rfloor \))
    do(i, j - 2i) = a(j - 2i, i) + a(i, j - 2i - 1)
  END
ENDDDO
```

However, there is a large gap between the complexity of the two first algorithms and the complexity of the two last algorithms, both in terms of dependence abstractions and in terms of running complexity. The goal of this paper is to fill this gap and to propose an intermediate algorithm, thus of medium complexity but still optimal for all classical approximations of dependences, namely approximations of distance vectors by polyhedra.

In Example 1, exact dependence analysis is indeed not necessary to reveal maximal parallelism. One has just to notice that there is one uniform dependence \( d = (0, 1) \) and a set of distance vectors \( \{(j - i, i - j) = (j - i)(1, -1) \mid 1 \leq j - i \leq n - 1 \} \) that can be approximated by the set \( P = \{(1, -1) + \lambda(1, -1) \mid 0 \leq \lambda \} \). \( P \) is a polyhedron with one vertex \( v = (1, -1) \) and one ray \( r = (1, -1) \).

Suppose that, as in the above algorithms, we are looking for a linear schedule \( T(i, j) = x_1i + x_2j \). For \( T \) to be a valid schedule, we impose that \( X(0, 1) \geq 1 \) and \( Xp \geq 1 \) for all \( p \in P \), where \( X = (x_1, x_2) \). Instead of using Farkas’ lemma to solve the second constraint, we can remark that it is equivalent to \( Xv \geq 1 \) and \( Xr \geq 0 \). Therefore, one has just to solve the three inequalities:

\[
X_1 \geq 1 \quad X_2 \geq 1 \quad X_r \geq 0
\]

The number of inequalities and variables is related to the number of constraints that define the validity domain of each dependence relation.

which leads, as above, to \( X = (2, 1) \). In other words, we have “uniformized” the constraints and transformed the underlying affine scheduling problem into a simple scheduling problem where all dependences are uniform \((d, v, r)\). However, compared to the classical framework of uniform loop nests, there are two fundamental differences:

- the uniform dependence vectors are not necessarily lexicographic (a ray equal to \((0, -1)\) for example is possible). This makes the problem a lot more difficult, but it can be solved by techniques related to the problem of computability of a system of uniform recurrence equations.
- the constraint imposed for a ray \( r \) is weaker: the constraint is \( Xr \geq 0 \) instead of \( Xr \geq 1 \). This freedom must be taken into account when deriving the parallelization algorithm.

To better understand this “uniformization” principle, think in terms of dependence path: actually, we consider an edge \( e \) labeled by the distance vector \( p = v + \lambda r \) as a path \( \phi \) that uses once the “uniform” dependence vector \( v \) and \( \lambda \) times the “uniform” dependence vector \( r \). However, we consider that the use of the dependence \( r \) counts for \( 0 \) instead of \( 1 \) (constraint \( Xr \geq 0 \) instead of \( Xr \geq 1 \)) when defining the length of the path \( \phi \) so that \( e \) and its equivalent path \( \phi \) have same length. This simulation is summarized in Figure 2: we introduce a new node \( S' \) that simulates \( \phi \) and a null-weight edge to come back to the initial node \( S \).

This “uniformization” principle is the underlying idea of the loop parallelization algorithm proposed in this paper. This algorithm has the following properties:

- It does not require exact dependence analysis, but it is optimal for dependence graph whose edges are labeled by a polyhedral approximation of distance vectors. In particular, it is optimal for
level of dependences and direction vectors. Actually, it behaves exactly as Allen and Kennedy’s algorithm when dependences are expressed by dependence levels and it generalizes Wolf and Lam’s algorithm [17] to the case of multiple statements when dependences are expressed by direction vectors (Wolf and Lam’s algorithm is optimal if there is only one statement).

- It points out precisely which dependences prevent the parallelization or are responsible for a loss of parallelism. This enlightens the link between the maximal degree of parallelism that can be detected and the accuracy of dependence abstractions. See [8] for a complete study about this question.

- By construction, it can be naturally adapted to the search for maximal sets of fully permutable loops which is, in theory, an equivalent problem, and, in practice, a way to exploit medium grain parallelism.

The paper is organized as follows. In Section 2, we recall generalities on dependence analysis and dependence graphs. We formally define what we call polyhedral reduced dependency graphs, and we demonstrate the expressive power of this dependence abstraction.

In Section 3, we give an overview of the different steps of the parallelization algorithm, for perfect nested loops. Unfortunately, because of a lack of space, we can not give the full proofs of correctness and optimality of our algorithm. We refer to [8] in which all proofs are detailed.

Nevertheless, in Section 4, we summarize our results by showing how they are related to techniques developed for systems of uniform recurrence equations [6]. We illustrate the algorithm on a quite complicated example.

Finally, in Section 6, we discuss some implementation strategies that permit to reduce the complexity of the algorithm and to clean up the solution for code generation. Then, we briefly show how extending the algorithm to non perfect loop nests and we conclude in Section 7.

2. Dependence abstractions

For the sake of clarity, we restrict ourselves to the case of perfectly nested DO loops with affine loop bounds. Non perfectly nested loops are considered in Section 6.2. With this restriction, we can identify, as usual, the iterations of \( n \) nested loops (\( n \) is called the depth of the loop nest) with vectors in \( \mathbb{Z}^n \) (called the iteration vectors) contained in a finite convex polyhedron bounded by the loop bounds (called the iteration domain). The \( i \)-th component of an iteration vector is the value of the \( i \)-th loop counter in the nest, counting from the outermost to the innermost loop. In the sequential code, the iterations are therefore executed in the lexicographic order of their iteration vectors.

In the next sections, we denote by \( D \) the polyhedral iteration domain, by \( I \) and \( J \) \( n \)-dimensional iteration vectors in \( D \), and by \( S_j \) the \( j \)-th statement in the loop nest. We write \( I \preceq J \) if \( I \) is lexicographically greater than \( J \) and \( I \preceq J \) if \( I \preceq J \) or \( I = J \).

Section 2.1 recalls the different concepts of dependence graphs: expanded dependence graphs (EDG), reduced dependence graphs (RDG), apparent dependence graphs (ADG) and the notion of distance sets. In Section 2.2, we formally define what we call polyhedral reduced dependence graphs (PRDG), i.e. reduced dependence graphs whose edges are labeled by polyhedra. Finally, in Section 2.3, we show how the model of PRDG generalizes classical dependence abstractions of distance sets.

2.1. Dependence graphs and distance sets

Dependence relations between operations are defined by Bernstein’s conditions [2]. Briefly speaking, two operations are considered dependent if both operations access the same memory location and if at least one of the accesses is a write. The dependence is directed according to the sequential order, from the first executed operation to the last. Depending on the order of write(s) and/or read, the dependence corresponds to the so called flow dependence, anti dependence or output dependence. We write: \( S_j(I) \rightarrow S_j(J) \) if statement \( S_j \) at iteration \( J \) depends on statement \( S_j \) at iteration \( I \). The partial order defined by \( \rightarrow \) describes the expanded dependence graph (EDG). Note that \( (J - I) \) is always lexicographically non negative when \( S_j(I) \rightarrow S_j(J) \).

In general, the EDG can not be computed at compile-time, either because some information is missing (such as the values of size parameters or even worse, precise memory accesses), or because generating the whole graph is too expensive. Instead, dependences are captured through a smaller (in general) cyclic directed graph, with \( s \) vertices, called the reduced dependence graph (RDG) (or statement level dependence graph).

\[ ^3 \text{In some cases, output and anti dependences can be removed by data renaming and/or expansion. See for example [9].} \]
The RDG is a compression of the EDG. In the RDG, two statements $S_i$ and $S_j$ are said dependent (we write $S_i \rightarrow S_j$) if there exists at least one pair $(I, J)$ such that $S_i(I) \rightarrow S_j(J)$. Furthermore, the dependence $S_i \rightarrow S_j$ is labeled by the set $\{(I, J) \in D^2 \mid S_i(I) \rightarrow S_j(J)\}$, or by an approximation $D_r$ that contains this set. The precision and representation of this approximation makes the power of the dependence analysis.

In other words, the RDG describes, in a condensed manner, an iteration level dependence graph, called (maximal) apparent dependence graph (ADG), that is a superset of the EDG. The ADG and the EDG have the same vertices, but the ADG has more edges, defined by:

$$(S_i, J) \implies (S_j, J) \text{ (in the ADG) } \Leftrightarrow \exists e = (S_i, S_j) \text{ (in the RDG) } \text{ such that } (I, J) \in D_e.$$ 

For a certain class of nested loops, it is possible to express exactly this set of pairs $(I, J)$ (see [9]): $I$ is given as an affine function $f_{i,j}$ of $J$ where $J$ varies in a polyhedron $P_{i,j}$:

$$\{(I, J) \in D^2 \mid S_i(I) \rightarrow S_j(J)\} = \{(f_{i,j}(J), J) \mid J \in P_{i,j} \subset D\} \quad (1)$$

In most dependence analysis algorithms however, rather than the set of pairs $(I, J)$, one computes the set $E_{i,j}$ of all possible values $(J - I)$. $E_{i,j}$ is called the set of distance vectors, or distance set:

$$E_{i,j} = \{(J - I) \mid S_i(I) \rightarrow S_j(J)\}$$

When exact dependence analysis is feasible, Equation 1 shows that the set of distance vectors is the projection of the integer points of a polyhedron. This set can be approximated by its convex hull or by a more or less accurate description of a larger polyhedron (or a finite union of polyhedra). When the set of distance vectors is represented by a finite union, the corresponding dependence edge in the RDG is decomposed into multi-edges.

Note that the representation by distance vectors is not equivalent to the representation by pairs (as in Equation 1), since the information concerning the location in the EDG of such a distance vector is lost. This may even be the cause of a loss of parallelism. However, this representation remains important, especially when exact dependence analysis is either too expensive or not feasible.

Classical representations of distance sets (by increasing precision) are:

- **level of dependence**, introduced for Allen and Kennedy’s parallelizing algorithm [1],
- **direction vector**, introduced by Wolfe [18] and used in Wolf and Lam’s parallelizing algorithm [17],
- **dependence polyhedron**, introduced in [12] and used in Irigoin and Triplet’s supernode partitioning algorithm [13].

In the rest of the paper, we explore this last representation. We now first define formally reduced dependence graph whose edges are labeled by dependence polyhedra and we show the expressive power of this model.

### 2.2. Polyhedral Reduced Dependence Graphs

We first recall the mathematical definition of a polyhedron and its decomposition into vertices, rays and lines.

**Definition 1 (Polyhedron, polytope)**

A set $P$ of vectors in $\mathbb{R}^n$ is called a (convex) polyhedron if there exists an integral matrix $A$ and an integral vector $b$ such that:

$$P = \{x \mid x \in \mathbb{R}^n, Ax \leq b\}$$

* A polytope is a bounded polyhedron.

A polyhedron can always be decomposed as the sum of a (convex) polytope and of a polyhedral cone (for more details see [16]). A polytope is defined by its vertices, and any point of the polytope is a non-negative barycentric combination of the polytope vertices. A polyhedral cone is finitely generated and can be defined by its rays and lines. Any point of a polyhedral cone is the sum of a non-negative combination of its rays and of any combination of its lines.

Therefore, a convex dependence polyhedron $P$ can be equivalently defined by a set of vertices (denoted by $\{v_1, \ldots, v_w\}$), a set of rays (denoted by $\{r_1, \ldots, r_p\}$), and a set of lines (denoted by $\{l_1, \ldots, l_q\}$). Then, $P$ is the set of all vectors $p$ such that:

$$p = \sum_{i=1}^{w} \mu_i v_i + \sum_{i=1}^{p} \nu_i r_i + \sum_{i=1}^{q} \xi_i l_i \quad (2)$$

with $\mu_i \in \mathbb{Q}^+$, $\nu_i \in \mathbb{Q}^+$, $\xi_i \in \mathbb{Q}$, and $\sum_{i=1}^{w} \mu_i = 1$.

We now define what we call a polyhedral reduced dependence graph (or PRDG), i.e. a reduced dependence graph labeled by dependence polyhedra. Actually, we will be interested only in integral vectors that belong to the dependence polyhedra, since dependence distance are indeed integral vectors.
Definition 2 A polyhedral reduced dependence graph (PRDG) is a RDG, for which each edge is labeled by a dependence polyhedron $P(e)$ that approximates the set of distance vectors: the associated ADG contains an edge from instance $I$ of node $S_i$ to instance $J$ of node $S_j$ if and only if $(J - I) \in P(e)$.

In the rest of the paper, to avoid a possible confusion between the vertices of a dependence graph and the vertices of a dependence polyhedron, we call the first one nodes and the second one vertices.

2.3. Simulation of classical dependence representations

We now come back to more classical dependence abstractions: level of dependence and direction vector. We recall their definition and show that RDGs labeled by direction vectors or levels of dependence are actually particular cases of polyhedral reduced dependence graphs.

2.3.1 Direction vectors

When the set of distance vectors is a singleton, the dependence is said uniform and the only distance vector is called a uniform dependence vector. Otherwise, the set of distance vectors can still be represented by a $n$-dimensional vector (called the direction vector), whose components belong to $\mathbb{Z} \cup \{1\} \cup \mathbb{Z} \times \{+, -, -\}$). Its $i$-th component is an approximation of the $i$-th components of all possible distance vectors: $z$ if the dependence is uniform in this dimension with unique value $z$, $z+$ (resp. $z-$) if all $i$-th components are greater (resp. smaller) than or equal to $z$, and $*$ if the $i$-th component may take any value. In general, $+$ (resp. $-$) is used as shorthand for $1+$ (resp. $-1-$).

A direction vector is nothing but an approximation by a polyhedron, with a single vertex and whose rays and lines, if any, are canonical vectors. For example, the direction vector $(2, +, *, -)\alpha$ defines the polyhedron with one vertex $(2, 0, -1, 3)$, two rays $(1, 0, 0, 0)$ and $(0, 0, -1, 0)$, and one line $(0, 1, 0, 0)$.

2.3.2 Level of dependences

The representation by level is the less accurate (though powerful [7]) dependence abstraction. In a loop nest with $n$ nested loops, the set of distance vectors is approximated by an integer $l_i$, in $\{1, \ldots, n\} \cup \{\infty\}$, defined as the largest integer such that the $l - 1$ first components of the distance vectors are null, or $\infty$ if all components are null.

A dependence level is also a representation by a polyhedron. For example, level 2, in a 3-dimensional loop nest, means direction vector $(0, 1, +, *)$ which corresponds to the polyhedron with one vertex $(0, 1, 0)$, one ray $(0, 1, 0)$ and one line $(0, 0, 1)$.

3. Overview of the parallelization algorithm

Our parallelization algorithm consists of two main steps, a “uniformization” step presented in Section 3.2 and a “scheduling” step summarized in Section 3.3. We illustrate both steps with Example 2 below.

3.1. Illustrating example

We will work out the following example, assuming that in the reduced dependence graph, edges are labeled by direction vectors.

Example 2

\[
\begin{align*}
\text{DO } i & = 1, n \\
\text{DO } j & = 1, n \\
\text{DO } k & = i, j \\
\quad & a(i, j, k) = c(i, j, k - 1) + 1 \\
\quad & b(i, j, k) = a(i - 1, j + i, k) + b(i, j - 1, k) \\
\quad & c(i, j, k + 1) = c(i, j, k) + b(i, j - 1, k + 1) + a(i, j - k, k + 1) \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

The graph depicted in Figure 3 has been found by the dependence analyzer Tiny [10].

![Figure 3. Reduced dependence graph with direction vectors, for Example 2.](image)

The reader can check that neither Allen and Kennedy's algorithm, nor Wolf and Lam's algorithm,
is able to find the full parallelism for this code: the third statement seems to be purely sequential.

However, the parallelism detection algorithm that we propose in the next sections is able to build the following multi-dimensional schedule: \((2i + 1, 2k)\) for the first statement, \((2i, j)\) for the second statement and \((2i + 1, 2k + 3)\) for the third statement. This schedule corresponds to the code with explicit parallelism given below (but in which no effort has been made so as to remove “if” tests). For each statement, one level of parallelism has been detected.

This code has been generated by the the procedure “codegen” of the Omega Calculator delivered with Petit [15], thanks to Bill Pugh’s team. We point out that it is a “virtual” code in the sense that it only reveals hidden parallelism. We do not claim that it must be implemented as such.

DOSEQ \(i = 1, n\)

DOSEQ \(j = 1, n\)

DOPAR \(k = 1, j\)

\(b(i, j, k) = a(i - 1, j - 1, k) + b(i, j - 1, k)\)

ENDDO

ENDDO

DOSEQ \(k = 1, n + 1\)

IF \((k \leq n)\)

DOPAR \(j = k, n\)

\(a(i, j, k) = c(i, j, k - 1) + 1\)

ENDDO

ENDIF

IF \((k \geq 2)\)

DOPAR \(j = k - 1, n\)

\(c(i, j, k) = c(i, j, k - 1) + b(i, j - 1, k + 1) - 1\)

ENDDO

ENDDO

3.2. Uniformization step

We first show how PRDGs (polyhedral reduced dependence graphs) can be captured into an equivalent (but simpler to manipulate) structure, the structure of uniform dependence graphs, i.e. graphs whose edges are labeled by constant dependence vectors. This uniformization scheme is achieved by the translation algorithm, given below.

The initial PRDG that describes the dependences in the code to be parallelized is called the original graph and denoted by \(G_o = (V, E)\). The uniform RDG, equivalent to \(G_o\), built by the translation algorithm, is called the uniform graph or the translated of \(G_o\) and is denoted by \(G_u = (W, F)\).

The translation algorithm builds \(G_u\) by scanning all edges of \(G_o\). It starts from \(G_u = (W, F) = (V, \emptyset)\), and for each edge \(\epsilon\) of \(G\), it adds to \(G_u\) new nodes and new edges depending on \(P(\epsilon)\). We call virtual nodes the nodes that are created as opposed to actual nodes which correspond to nodes of \(G_o\).

We follow the notations introduced in Section 2.2: we denote respectively by \(\omega, \rho, \Lambda\) and \(\lambda\) the number of vertices \(v_i\), rays \(r_i\), and lines \(l_i\) of the polyhedron \(P(\epsilon)\).

Translation Algorithm

- Let \(W = V \cup F = \emptyset\)
- For \(e = (x_e, y_e) \in E\) do
  1. If \(\rho = 0\) and \(\lambda = 0\) (the polyhedron is reduced to a polytope)
     - Add to \(F\) \(\omega\) edges of weights \(v_1, v_2, \ldots, v_\omega\) directed from \(x_e\) to \(y_e\).
  2. If \(\rho \neq 0\) or \(\lambda \neq 0\)
     - Add to \(W\) a new virtual node \(n_e\).
     - Add to \(F\) \(\omega\) edges of weights \(v_1, v_2, \ldots, v_\omega\) directed from \(x_e\) to \(n_e\).
     - Add to \(F\) \(\rho\) self-loops around \(n_e\) of weights \(r_1, r_2, \ldots, r_\rho\).
     - Add to \(F\) \(\lambda\) self-loops around \(n_e\) of weights \(l_1, l_2, \ldots, l_\lambda\).
     - Add to \(F\) a null weight edge directed from \(n_e\) to \(y_e\).

Back to Example 2. The uniform dependence graph associated to the PRDG of Example 2 (see Figure 3) is depicted in Figure 4. It has three new nodes (i.e. virtual nodes) that correspond to the symbol “+” and the two symbols “…” in the initial direction vectors.

3.3. Scheduling step

The scheduling step takes as input the translated dependence graph \(G_u\) and builds a multi-dimensional schedule for each actual node, i.e. for each node of \(G_u\) that corresponds to a node of \(G_o\). \(G_u\) is assumed to be strongly connected (otherwise, the algorithm has to be called for each strongly connected component of \(G_u\)).

It is a recursive algorithm. Each step of the recursion builds a subgraph \(G'\) of the current graph \(G\) being processed: \(G'\) is the subgraph of \(G\) generated by all edges of \(G\) that belong to at least one multi-cycle (i.e. union of cycles) of null weight.

\(G'\) can be built by one linear programming resolution. Indeed, it has been shown (see [0]) that the edges
3. Select a vector $X$ and, for each node $S$ in $G$, a constant $\rho_S$ such that:

$$
\begin{align*}
\{ & e = (x_v, y_v) \in G' \text{ or } x_v \text{ is a virtual node} \\
& \Rightarrow Xw(e) + \rho_{x_v} - \rho_{y_v} \geq 0 \\
& e = (x_v, y_v) \not\in G' \text{ and } x_v \text{ is an actual node} \\
& \Rightarrow Xw(e) + \rho_{y_v} - \rho_{x_v} \geq 1
\end{align*}
$$

For all actual nodes $S$ of $G$, let $\rho_S^k = \rho_S$ and $X_S^k = X$.

4. If $G'$ is empty or has only virtual nodes, return.

5. If $G'$ is strongly connected and has at least one actual node, $G$ is not computable (and the initial PRDG $G_0$ is not consistent), return.

6. Otherwise, decompose $G'$ into its strongly connected components $G_i$ and for each $G_i$ that has at least one actual node, call DARTE-VIVIEN($G_i$, $k + 1$).

Remarks

- Step (2) is necessary only for general PRDGs: for example, it could be removed for RDGS labeled by direction vectors. In this case, Steps (1) and (3) can be solved by a single linear program resolution.

- In Step (3), we do not specify, on purpose, how the vector $X$ and the constants $\rho$ are selected, so as to allow various selection criteria, depending if fine or medium grain parallelism is desired. For example, a maximal set of linearly independent vectors $X$ can be selected if the goal is to derive fully permutable loops (see [6]).

Back to Example 2 Consider the uniform dependence graph of Figure 4. There are two elementary cycles of weights $(1, 0, 1)$ and $(0, 1, 1)$, and five self-loops of weights $(0, 0, 1), (0, 0, -1), (0, 1, 0), (1, 0, 0)$, and $(0, -1, 0)$. Therefore, all edges except the edges that only belong to the cycle of weight $(1, 0, 1)$ belong to a multi-cycle of null weight. The subgraph $G'$ is depicted in Figure 5.

The constraints coming from edges in $G'$ make that $X = (x, y, z)$ must be orthogonal to the weight of all cycles of $G'$. Therefore, $y = z = 0$. Finally, considering the other constraints, we find the solution $X = (2, 0, 0)$, $\rho_{y_3} = \rho_{y_4} = 1$ and $\rho_{y_5} = 0$.

In $G'$, there remains four strongly connected components, and two of them are not considered since they only have virtual nodes. The two other components have no null weight multi-cycle. The strongly
connected component with the single node $S_3$ can be scheduled with the vector $X = (0, 1, 0)$, whereas studying the other strongly connected component leads, among other solutions, to $X = (0, 0, 2)$, $\rho_{S_1} = 0$, and $\rho_{S_3} = 3$.

Finally, summarizing the results, we find, as claimed in Section 3.1, the 2-dimensional schedules: $(2i, j)$ for $S_2$, $(2i + 1, 2k)$ for $S_1$ and $(2i + 1, 2k + 3)$ for $S_3$.

4. Properties of the parallelization algorithm

Note the particular structure of $G_u$: it is a graph with edges labeled by integral vectors, i.e. a uniform dependence graph. However, the weights of the edges are not necessarily lexicographically positive which makes a huge difference with classical uniform dependence graphs of nested loops.

Actually, $G_u$ is very similar to a reduced dependence graph associated to a system of uniform recurrence equations. The only difference is that some nodes of $G_u$ are virtual nodes. This difference is small and this is why we can still use (with slight modifications) all techniques we previously developed for systems of uniform recurrence equations [6].

The correctness and optimality of our algorithm comes from this strong link between the "uniformized" graph $G_u$ and systems of uniform recurrence equations. In particular, we have the following results, whose proofs are detailed in [6] and [8].

4.1. Correctness

Theorem 1 (Computability Condition)

$G_u$ is computable if and only if $G_u$ contains no null weight cycle with at least one actual node.

Furthermore, $G_u$ has no cycle of null weight containing an actual node if and only if $DA(G_u) = \text{TRUE}$, where $DA$ is the decomposition algorithm given below. Therefore, a PRDG $G_u$ is computable if and only if $DA(G_u) = \text{TRUE}$.

Algorithm $DA$ is a modified version of the seminal decomposition of Karp, Miller, and Winograd [14]. $\land$ denotes the logical AND.

Boolean $DA(G)$

1. Build $G'$ the subgraph of null weight multi-cycles of $G$.

2. Compute the strongly connected components of $G'$ and let $G'_1, G'_2, \ldots, G'_t$ be the $s$ components that have at least one actual node.

   - If $G'$ is empty or has only virtual nodes, return $\text{TRUE}$.
   - If $G'$ is strongly connected and has at least one actual node, return $\text{FALSE}$.
   - Otherwise, return $\bigwedge_{i=1}^{t} DA(G'_i)$.

This decomposition is related to the computability problem. However, considering at each step the dual of linear program 3 establishes the link with the scheduling problem. This is, without entering the details, what makes correct our scheduling algorithm. Algorithm $DA$ is indeed the skeleton of the algorithm of Section 3.3.

4.2. Optimality

In the scheduling algorithm, each statement $S$ is scheduled by a $d_S$-multi-dimensional schedule. $d_S$ is called the depth of $S$. It is equal to the number of recursive calls (counting the first one) needed to remove $S$ from the graph, except if $S$ do not belong to a cycle, in which case $d_S = 0$. $d$, the depth of the graph, is the maximal $d_S$.

For systems of uniform recurrence equations, the depth of a graph is a measure of the degree of parallelism it describes. This result still holds for PRDGs and the depth $d$ defined above. Indeed, we have the following results:

Theorem 2 If $G_u = (V, E)$ is computable, the multi-dimensional scheduling function $T$:

$$V \times D \longrightarrow Z^d$$

$$(S, I) \rightarrow (\ldots, [X_S^I + p_S], \ldots)$$
defines a valid multi-dimensional schedule for \( G_o \).

Furthermore, this schedule is optimal, in the sense that for each statement \( S \) (i.e. for each node of \( G_o \)), the number of instances of \( S \) that have been sequentialized by \( T \) is of the same order as the number of instances of \( S \) that are inherently sequentialized by the dependences.

More precisely:

**Theorem 3** Assume that the iteration domain \( D \) is contained in a \( n \)-dimensional cube of size \( O(N) \) and contains a \( n \)-dimensional cube of size \( \Omega(N) \). Then, the latency of the schedule is \( O(N^d) \) and the length of the longest dependence path is \( \Omega(N^d) \). More precisely, after generation, each statement \( S \) is surrounded by exactly \( d \) sequential loops and these loops are inherently sequential.

5. Yet another example

We illustrate our technique with a third example, in which the maximal parallelism can be detected only if dependences are approximated by a more accurate PRDG than a RDG labeled by direction vectors. After parallelization, \( S_1 \) is surrounded by a single sequential loop and \( S_2 \) by two.

**Example 3**

```plaintext
DO i = 1, n
  DO j = 1, n
    DO k = 1, n
      a(i, j, k) = a(i, j - 1, k + j) + b(j, i - 1, k)
      b(i, j, k) = b(i, j, k - 1) + a(i, j, k)
    ENDDO
  ENDDO
 ENDDO
ENDDO

The graph \( G_o \) depicted in Figure 6 has been found by the dependence analyzer Tiny [19]. The uniformization step transforms \( G_o \) into \( G_u \) which is depicted in Figure 7.

There is a multi-cycle of null weight generated by all edges whose weight is orthogonal to \( (1, 0, 0) \) (see Figure 7). In \( G' \), the strongly connected component that contains \( S_1 \) and \( S_2 \) still has a multi-cycle of null weight that visits an actual node (\( S_2 \)). \( S_1 \) is removed at depth 2 but \( S_2 \) is removed at depth 3. \( S_2 \) is purely sequential, whereas one degree of parallelism is detected for \( S_1 \). The multi-dimensional schedules are \((i, 2j)\) for \( S_1 \) and \((i, 2j + 1, k)\) for \( S_2 \).

**Figure 6. Reduced dependence graph with direction vectors, for Example 3.**

The resulting code is therefore the following:

```
DOSEQ j = 1, 3, n
  DOPAR k = 1, n
    DOPAR i = \text{max}(0, 1 - \frac{\text{min}(n/2, j)}{2}), \text{min}(n, 1 - \frac{\text{min}(n/2, j)}{2})
    a(i, j - 2i, k) = a(i, j - 2i - 1, k + j - 2i) + b(i + 2, i - 1, k)
  ENDDO
 ENDDO
ENDDO
```

Note that this is exactly what Allen and Kennedy’s algorithm would find. However, if direction vectors are refined by more accurate dependence tests, one can find that the dependences can be approximated by the PRDG of Figure 8.

The reference to array \( k \) generates indeed two dependences, a flow dependence whose dependence polyhedron has one vertex \((0, 1, 0)\) and one ray \((1, -1, 0)\), and an anti dependence whose dependence polyhedron has one vertex \((1, -2, 0)\) and the same ray \((1, -1, 0)\). Note in Figure 8 how this modification changes the structure of \( G' \). \( S_1 \) is now removed at depth 1 and \( S_2 \) at depth 2. For both statements, one more level of parallelism has been detected. The multi-dimensional schedules are \((4i + 2j)\) for \( S_1 \) and \((4i + 2j + 1, k)\) for \( S_2 \).

The resulting code is therefore the following:

```
DOSEQ j = 3, 3n
  DOPAR k = 1, n
    DOPAR i = \text{max}(0, 1 - \frac{\text{min}(n/2, j)}{2}), \text{min}(n, 1 - \frac{\text{min}(n/2, j)}{2})
    b(i, j - 2i, k) = b(i, j - 2i - 1, k - 1) + a(i, j + 2i, k)
  ENDDO
 ENDDO
ENDDO
```

```
6. Implementation strategies

6.1. Reasoning on cycles

Consider the constraints of Step (3). It can be shown that they are equivalent to the constraints \( X u(C) \geq l(C) \) for all elementary cycles \( C \) where \( l(C) \) denotes the number of edges \( e = (x, y) \) of \( C \) that do not belong to \( G' \) and for which \( x \) is an actual node. Furthermore, once the constraints on cycles are satisfied by \( X \), the different constants \( \rho \) can be computed by a technique similar to the Bellman-Ford algorithm, less expensive than a linear programming resolution.

This remark suggests to adopt a two-step strategy: compute \( X \) first by a linear programming approach, and then compute the constants \( \rho \) by a graph-based technique. Unfortunately, it would increase the number of constraints, since the number of elementary cycles in the graph may be exponential in the number of edges. Therefore, computing the weight of all cycles in the PRDG is not reasonable.

To avoid this problem, we propose to use a basis of cycles, instead of considering all cycles, which simulate all constraints on cycles with only \( |E| - |V| + 1 \) cycles. This technique permits us to keep small the number of constraints and variables in the linear programs to solve. Furthermore, it guarantees that the constants \( \rho \) are simple if the vector \( X \) is simple. This property is highly desirable for code generation. Full details can be found in [8].

6.2. Extension to non perfectly nested loops

As proved in the previous sections, our scheduling algorithm is perfectly adapted to a description of distance vectors. When the loops are non perfectly nested, the distance vector \( J - I \) between two statements \( S_1 \) and \( S_2 \) is defined only for the first dimensions that correspond to common loops, i.e., loops that surround both \( S_1 \) and \( S_2 \).

Therefore, a natural way of extending the algorithm to non perfect loop nests is to ignore, in each strongly connected component that appears during the decomposition, all dimensions that are not common dimensions. In other words, at a given depth of the algorithm, we truncate all vectors to the same dimension and we apply on the truncated vectors the same technique as for perfectly nested loops. Finally, we complete each vector \( X \) derived with null components so that they fit the right dimension.

It turns out that this strategy remains optimal, as long as no information is given on the non common dimensions. However, if at each level, the code is non
perfect then this algorithm is not more powerful than Allen and Kennedy’s algorithm, since there is only one common dimension at each step.

To avoid this problem, we suggest another approach that exploits the information on non common dimensions, and to benefit from the power of our algorithm for perfectly nested loops. We first transform the code into perfectly nested loops, by loops fusions or more complex techniques, possibly introducing “if” tests. Then, the scheduling algorithm is applied on the transformed nest, reasoning on its dependence graph. Here is an example, borrowed from the examples proposed in Petit [15].

Example 4

\[
\begin{align*}
\text{DO } & i = 2, n \\
& s(i) = 0 \\
\text{DO } & j = 1, i-1 \\
& s(i) = s(i) + a(j, i) \ b(j) \\
\text{ENDDO} \\
& b(i) = b(i) - s(i) \\
\text{ENDDO} \\
\end{align*}
\]

In this example, the dependence graph has two strongly connected components, one with \( S_1 \), the other one with \( S_2 \) and \( S_3 \). We can thus apply a loop distribution to separate \( S_1 \) from \( S_2 \) and \( S_3 \). Furthermore, we integrate \( S_3 \) into the second loop, so as to obtain only perfect loops nests. We get:

\[
\begin{align*}
\text{DOPAR } & i = 2, n \\
& s(i) = 0 \\
\text{ENDDO} \\
\text{DO } & i = 2, n \\
\text{DO } & j = 1, i \\
\text{IF } & (j \leq i - 1) \text{ THEN} \\
& s(i) = s(i) + a(j, i) \ b(j) \\
\text{ENDIF} \\
\text{IF } & (j = i) \text{ THEN} \\
& b(i) = b(i) - s(i) \\
\text{ENDIF} \\
\text{ENDDO} \\
\text{ENDDO} \\
\end{align*}
\]

The reduced dependence graph, with direction vectors, for the two last statements is depicted in Figure 9. It is easy to see that the corresponding uniformalized dependence graph has no multi-cycle of null weight. Therefore, there is some parallelism. Indeed, applying our scheduling algorithm, we find that the vector \( X = (x, y) \) has to satisfy the constraints \( y \geq 1, x + y \geq 2, x \geq 0, \) and \( y \geq 0 \). We find \( X = (0, 2) \) and \( \rho_{S_2} = 1 \) and \( \rho_{S_3} = 0 \) which corresponds to the following code (once again without any effort to remove if test):

\[
\begin{align*}
\text{DOPAR } & i = 2, n \\
& s(i) = 0 \\
\text{ENDDO} \\
\text{DO } & j = 1, n \\
\text{IF } & (j \geq 2) \text{ THEN} \\
& b(i) = b(i) - s(j) \\
\text{ENDIF} \\
\text{DOPAR } & i = j+1, n \\
& s(i) = s(i) + a(j, i) \ b(j) \\
\text{ENDDO} \\
\text{ENDDO} \\
\end{align*}
\]

7. Conclusion

We have presented an original scheduling algorithm to parallelize loops whose dependences are described through polyhedral reduced dependence graphs, i.e. reduced dependence graphs whose edges are labeled by an approximation of distance vectors by polyhedra. This representation of dependences is a generalization of direction vectors.

Our algorithm is nearly optimal, in the sense that it detects the maximal number of parallel loops that can be found, as long as the only information available is the polyhedral reduced dependence graph. In particular, our algorithm is optimal for direction vectors, which generalizes Wolf and Lam’s algorithm to the case of multiple statements.

We illustrated the practical efficiency of our algorithm on several examples, examples with direction vectors as well as examples with more general polyhedral representations of distance vectors. All examples have been derived automatically with the algorithm we implemented and with the help of tools such as Tiny or Petit.

It remains some work to do for handling non perfectly nested loops. Our algorithm is indeed well adapted for perfectly nested loops, or for common loops in non perfect codes. However, to better exploit information on non common loops, a promising approach
is to develop a method to transform non perfect loop nests into perfect loop nests. This transformation remains to be fully automatized.

References


Non-Singular Data Transformations: 
Definition, Validity and Applications

M.F.P. O'Boyle
Department of Computation
UMIST
Manchester M60 1QD
United Kingdom
mob@rsa.co.umist.ac.uk

P.M.W. Knijnenburg
Department of Computer Science
Leiden University
Niels Bohrweg 1, 2333 CA Leiden
the Netherlands
peterk@cs.leidenuniv.nl

Abstract

This paper describes a unifying framework for non-singular data transformations. It shows that a wide class of existing transformations may be expressed in this framework, allowing compound transformations to be performed in one step. Validity conditions for such transformations are developed as is the form of the transformed program and data. Constructive algorithms to generate data transformations for different applications are described and applied to example programs. It is shown that they can have a significant impact on program performance and may be used in situations where traditional loop transformations are inappropriate.

1 Introduction

Recent years have seen a great improvement in loop transformation theory. By using an affine representation of loops, several loop transformations have been incorporated into one single framework [24]. In [2], Banerjee shows that loop interchange, reversal and skewing can be described as unimodular transformations of the iteration space. In [18], Li and Pingali extend the class of transformations to nonsingular matrices, thereby incorporating loop scaling. Furthermore, this framework has been extended to statement-wise transformations [14] and integrated with loop alignment [15]. From this treatment, general validity tests have been derived. This enables compound transformations to be applied in one go rather than step by step [22, 23]. In [10], the class of transformations is further extended and presented in a unifying framework based on schedules.

Loop transformations, however, are not the only program transformations of interest when optimising programs for parallel execution. In this paper we describe a unifying framework for data transformations, similar in approach to the framework used for loop transformations. Data transformations can be loosely defined as those transformations concerned with the layout, storage and access of array data, rather than reordering the program control flow. Data transformations have a number of applications, including global index reordering, data alignment, data partitioning, access normalisation, false sharing elimination and storage reduction. For instance, in [15, 21], copying of data is used to eliminate cache conflicts and in [9] the use of data transformations to reduce false sharing in parallel C programs is explored. In [5, 6, 16, 19], the relative alignment of arrays to reduce to communication overhead is examined. Data transformations can be used in those cases where data dependence prevents loop transformations and where access to other arrays should not be altered. They are unaffected by control-flow structure and may be used in the presence of imperfectly nested loops and arbitrary control-flow graphs. For a good survey of loop and data transformations, see [13].

Despite the wide applicability of such transformations, there has been little work done to incorporate them into a single framework. This is probably due to the loop oriented approach to parallelisation and dependence testing as developed for shared memory computing. However, data oriented approaches, necessitated by distributed memory computing and NUMAs, in general, highlight the need for a framework concerned with data rather than control structure. Although there has been little work in developing a framework, it has recently received some attention, most notably in [8]. In [8], the effect of row and column-major storage of arrays combined with loop transformations is investigated. Many of the points raised here are discussed, such as data layout being independent of (flow) dependences and how loop and data transformations are related. Their work, however, is preliminary in that the only data transformation discussed is linearisation, the mapping of n-dimensional arrays to 1-dimension. However, the loop transformations are full rank and dimension preserving. Hence there is an asymmetry in the treatment of loop and data transformations, weakening the unification theme in the paper. The transformations presented in this paper allow the row to column-major reordering described in [8] and describe a much larger set of transformations. In [1] it is proposed that arrays be viewed as polyhedral and unimodular transformations be used for changing array layout, though they restricted their attention to permutations.

In this paper we provide a general framework for data transformations, encompassing unimodular transformations. We show that this wide class of transformations is useful by giving an algorithm for constructing a data transformation that improves spatial locality and that leads us beyond simple permutations.

This paper makes the following contributions: it
• Describes a unifying framework for a large class of data transformations.
• Shows how existing transformations may be expressed within the framework.
• Defines the application of data transformations on program structure and code generation.
• Presents new validity tests for data transformations.
• Describes the symmetry between non-singular loop and data transformations.
• Provides sample algorithms to constructively generate data transformation and show their applicability in program optimization.

The paper is organised as follows. In the next section a motivating example is presented, outlining the case for data transformations. Section 3 defines the notation used throughout the paper, while section 4 defines the form of transformations under consideration. Section 6 presents validity criteria for data transformations and section 8 describes how non-singular data transformation can be integrated with loop transformations. Section 7 describes the application of data transformations. Constructive algorithms for three applications are defined and are applied to example programs and tested on existing architectures. Finally, section 8 concludes this paper.

2 Example

In this section, a simple example is used to illustrate the effect of data transformations on programs and why they are useful for compiler optimisation. Stride 1 access of arrays is important in architectures with some form of cache structure as it allows exploitation of spatial locality. Loop interchange is frequently used to ensure this for simple access functions [11]. For general affine accesses, a non-singular loop transformation [18] is first normalised the access is required, followed by loop interchange [17].

Loop transformations that improve spatial locality on one access may adversely affect others and even then may not be valid due to data dependences. The example in column 1 of figure 1 illustrates both these limitations with loop transformations. Fortran stores arrays in a column-major fashion. This is highlighted by the vertical lines within the diagrams of arrays A and B. The accessed region for each array is shaded grey, with arrows representing the traversal order imposed by the loops. At present there is bad stride access on both arrays A and B. Loop interchange would improve the access to A but is illegal due to the dependence vector [1, −1]. Furthermore, while access to A would improve, the access stride of B would still be poor. Changing column to row-major storage, as described in [8] and shown in column 2, would improve stride access to A but would again be of no use in improving access to B. This can be readily seen in the diagram of column 2.

Non-singular data transformations can be used as

• they are not restricted by data dependences, and
• they do not affect other array accesses within that loop nest.

Although they must be globally propagated, this impact can be reduced as described in section 8.

Applying the appropriate data transformations gives the new program in column 3. Since arrays in Fortran need to be rectangular, the new B array is larger than the original. We have to take the bounding box to generate the new array bounds. The dark region represents unaccessed elements introduced by the need to restrict arrays to rectangular form. The skewed access region is mapped to a square one, and stride 1 access is now possible for access to both arrays. The form of the non-singular data transformation, how it is deduced, applied and its impact on actual machine performance is described later in section 7.1.

3 Notation

In this section we briefly describe the notation used to develop the data transformation framework. It is based on a linear algebraic representation of program constructs.

3.1 Iteration Spaces

The Iterators in a Fortran program surrounding any statement can be represented as an \( m \times 1 \) column vector

\[
J = [j_1, j_2, \ldots, j_m]^T
\]

where \( m \) is the number of enclosing loops or iterators. The loop range or affine bounds of the iterators can be described by a system of inequalities defining the polyhedron

\[
AJ \leq b
\]

where \( A \) is a \((\ell \times m)\) integer matrix and \( b \) an \((\ell \times 1)\) vector. The integer values taken on by \( J \) define the iteration space of the iterators.

3.2 Index Domains

The data storage of an array \( A \) can also be viewed as a rectangular polyhedron. We introduce formal indices \( I \) for the array to describe the array index domain

\[
I = [i_1, i_2, \ldots, i_D(j)]^T
\]

where \( D(A) \) is the dimension of array \( A \). The formal indices have a certain range which describe the size of the array, or index space, as follows:

\[
\lambda \leq I \leq \mu
\]

where \( \lambda = [\lambda_1, \ldots, \lambda_D(j)]^T \) and \( \mu = [\mu_1, \ldots, \mu_D(j)]^T \) are \( D(A) \times 1 \) vectors. Due to the restrictions on the bounds of arrays in Fortran (or other languages), only constant vectors are required to describe array bounds.

3.3 Array Accesses

The subscripts in a reference to an array \( A \) represent a function that maps the values of the iteration space to the index domain. If \( J \) is the iteration vector, we assume in this paper that these subscripts can be written as an affine mapping that has the following form

\[
J + u
\]
where $U$ is a $D(n) \times m$ matrix and $u$ is a $D(n) \times 1$ vector.
Thus the access to $B$ in figure 1, $B(i+j, j)$, is denoted by

$$U_j + u = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The subscripts in a reference to an array are normally referred to by the access matrix/vector pair $(U, u)$.

4 Data Transformations

This section describes the application of data transformations and the range of transformations contained with the class of non-singular matrices. Data transformations on an array update all access to that array and also change the way the data is stored.

Definition A data transformation for an array $A$ of dimension $n$ consists of two parts:

1. A (non-singular) $n \times n$ matrix $A$, and
2. A $n \times 1$ vector $a$.

It is denoted by $(A, a)$.

When the shift vector $a$ equals $0$, we write $(A, a)$ simply as $A$. Given an access matrix/vector pair $(U, a)$ for an array $A$ occurring in the program, application of a data transformation $(A, a)$ to this access results in a new access $(U', u')$ given by

$$(U', u') = (AU, Au + a) \tag{2}$$

$A$ is a left-hand acting transformation, in contrast to loop transformations which are right-hand acting. Unlike loop transformations, which are local to a loop nest but affect all contained references, data transformations affect just one array but affect all accesses to the array globally. Thus the transformation $(A, a)$ must be applied to every occurrence of array $A$ throughout the program. In addition the actual storage of the array must be updated. Thus the index domain $Z$ of array $A$ must also be updated.

The application of $(A, a)$ is straightforward: all accesses are updated according to equation (2) and the index domain is updated as described in section 4.1 below. The validity of such transformations is discussed in section 5. No change need be made to the loop or control structure of the program.

In the example in figure 1 the following transformations are applied to the arrays in column 1 to give the transformed form shown in column 3.

$$A_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \text{ and } a_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \text{ and } a_2 = \begin{bmatrix} 0 \\ 4 \end{bmatrix} \tag{3}$$

Applying both transformations, we obtain

$$U_1' = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$u_1' = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

so $A(i, j)$ is mapped to $A(j, i)$.

$$U_2' = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$u_2' = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

so $B(i+j, j)$ is mapped to $B(j, i+j)$.

In section 7.1, a polynomial algorithm is presented which determines the correct transformation to obtain stride-1 access.
4.1 Transforming the Index Space

As well as transforming the accesses to an array, the actual storage for that array needs to be modified. The index space is transformed as follows.

Suppose \((A, a)\) is a data transformation for an array \(A\) with index space given by

\[
\lambda \leq i \leq \mu
\]

(4)

Then we need to compute a new index space \(\lambda' \leq i' \leq \mu'\). The new indices \(i'\) are related to the old indices \(i\) as

\[
i' = A_i + a
\]

Hence

\[
i = A^{-1}i' - A^{-1}a
\]

(5)

Substituting expression (5) in equation (4), we obtain

\[
\lambda + A^{-1}a \leq A^{-1}i' \leq \mu + A^{-1}a
\]

since both \(A\) and \(a\) are constant vectors.

If \(A\) is solely a permutation matrix, this is sufficient: \(A^{-1}i'\) is a permutation of the vector \(i\). Hence we can easily obtain \(\lambda' \leq i' \leq \mu'\). However, in the general case further work is necessary: we must solve for \(i'\) and take the bounding box of the transformed index space, as arrays in Fortran have to be rectangular.

To determine the new array bounds, Fourier-Motzkin elimination is applied to each index to find its range. That is, for an index we project away more outer indices to obtain the minimum and maximum values of an index.

For example, consider the access to array \(B\) in figure 1. \(A_3\) and \(a_3\) are given in equation (3) and

\[
A_3^{-1} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}
\]

This gives the new bounds

\[
\begin{align*}
1 & \leq i_3' \\
5 - i_3' & \leq i_2' \leq 4 \\
i_2' & \leq 12 - i_3'
\end{align*}
\]

After applying Fourier-Motzkin elimination we obtain the new bounds \(B(1:4,1:11)\).

4.2 Forms of \(A\)

Within this paper we restrict our attention to non-singular forms of \(A\). This is analogous to concentrating on non-singular loop transformations. It follows that \(A\) always has an inverse though this inverse is not necessarily integer. For the purposes of showing the usefulness of such transformations, we can subdivide them and illustrate their effect on the program segment in figure 2. Figure 3 summarises the forms of the transformations considered.

Unimodular

Unimodular matrices have been extensively studied and their properties will not be expounded here. They encompass permutation, reversal and skewing of the access matrix. They have the property that the convex hull of the transformed space will have the same number of points as the original domain. However as arrays must be rectangular in Fortran, additional storage will be needed to "pad" skewed arrays. Uses of such transformations include global index reordering and restricted forms of data alignment. Column 1 of figure 3 summarises the effect of unimodular transformations.

Do \(i = 1, n\)

Do \(j = 2, n\)

\(A(i+1,i+j) = i+j\)

Do \(i = 1, n\)

Do \(j = 2, n\)

\(A(2*i, j) = B(1, j)\)

Figure 2: Example

Integer determinant > 1

Integer matrices with a determinant greater than one are a generalisation of unimodular matrices. They can be considered to add another functionality to data storage, namely scaling or expansion. Such a scheme is useful when aligning one array of determinant one (i.e., a dense access pattern) to another of determinant two or greater (i.e., a regular sparse access pattern). Another use would be to expand small arrays so that when partitioned over a large template, the array is evenly distributed across all the processors. Scaling the first index by a factor of 2 is shown in figure 3.

Rational determinant

Rational non-singular transformations allow compression of the access and hence of data space. This is useful when space is critical, especially for reducing the sizes of "windows" in caches. Regular sparse accesses can be compressed so as to reduce the overall data allocation for that array. The example in the table shows compression in the first index by a factor of 2.

Shift

This is a very simple transformation where the arrays are shifted by a constant value. This can prove useful when concerned with aligning data on coherency unit boundaries or to remove problems associated with cache hardware mapping. The effect of a shift of \(-1\) is shown in figure 3.

4.3 Existing Transformations

While there has been much effort to show that loop transformations can be unified, there has been little work with regard to data transformations. Several existing transformations can be considered special cases of non-singular data transformations.

Global index reordering of a particular array corresponds to a simple permutation data transformation matrix.

Padding

array declarations to avoid cache conflicts corresponds to an identity data transformation plus a shift vector.

Data Partitioning

is in fact a rank deficient data transformation matrix whose non-null rows correspond to partitioned array dimensions.

Data Alignment

reorients data with respect to a template. Non-singular data transformations cover all the linear alignment directives used in HPF.

\(*\) the legality of which is not considered in this paper
All data layout schemes can be described as data transformations. Thus transformations from a wide range of applications can be viewed within one framework.

5 Validity

This section describes when data transformations can be legally applied. Data storage transformations can be considered to be an advanced form of renaming variables. Rather than just renaming a variable, e.g., z becomes y, we rename the index values as well. For example, A(1, 2) will now become B(3, 4). Renaming or changing the symbol of a variable is legal as long as the renaming occurs for all occurrences of the variable and there does not exist a variable that is legal for (A, a) are similar.

First of all, we must guarantee that all references are updated and secondly, that no clashes between two references occur. For example, if elements A(1, 2) and A(3, 4) are both accessed, they should not both be stored at point B(1, 2). We must ensure that no values are lost in this new storage scheme. As an additional check we should ensure that all data dependences are preserved. In fact, it will be shown that non-singular data transformations always preserve data dependence.

Fortran allows arrays to change dimension throughout a program especially when passing across subroutine boundaries. We assume that such a change in dimension does not occur in the program under consideration.

Let R be the total number of references to some array A. Let, for 1 ≤ i ≤ R, the ith reference to A be given by access matrix U_i and vector u_i. Let (A, a) be a data transformation for A. All references are updated, that is, for all 1 ≤ i ≤ R,

\[ U_i \rightarrow \mathcal{M} U_i \] and \[ u_i \rightarrow \mathcal{M} u_i + a \]

The first requirement we impose on the data transformation matrix A is that after application array index functions map iterators to integer values. That is, for every point J in the iteration space,

\[ \mathcal{M} U_i + u_i + a \in 2^R(A) \]

where \( R(A) \) denotes the number of rows in A.

The second requirement on A is that all distinct elements referenced in the original program are mapped to distinct points. Let \( (U_i, u_i) \) be an access matrix/vector pair occurring in a loop \( J_i \) and let \( (U_k, u_k) \) be an access matrix/vector pair occurring in another loop \( J_k \). Then we require that for every iteration \( J_i \) and \( J_k \) in these loops,

\[ U_i J_i + u_i = U_k J_k + u_k \quad \text{iff} \quad A U_i J_i + u_i + a = A U_k J_k + u_k + a \]

Note that we do not require that every element in the array is mapped to a distinct element, but rather that every element referenced in the program is mapped to a distinct element. This distinction is important since we want to consider storage compression for arrays that are only partially filled. Since in this paper we are only considering data transformation matrices that are non-singular and hence bijective, this requirement is automatically fulfilled.

In future research we hope to consider singular data transformation matrices also. In this case the requirement is not trivial and needs some careful consideration. A desirable application of a singular data transformation is given by transforming a high dimensional temporary array into a low dimensional one, where we reuse storage. A second application is given by strip-mining an array, in which case a low dimensional array is transformed into a high dimensional one. In fact, in [1] it has been shown that strip-mining (a singular data transformation) together with permutation (a non-singular data transformation) can be used for obtaining data layout automatically. This result shows that a general framework of singular data transformations, encompassing non-singular transformations, is important for obtaining efficient code for parallel machines.

Next we consider the validity of non-singular data transformations. In particular, we will show that non-singular data transformations do not violate data dependences.

**Lemma 1** Let \( (A, a) \) be a non-singular data transformation for some array \( A \). Suppose \( (U_1, u_1) \) and \( (U_2, u_2) \) be two access matrix/vector pairs for \( k \) in some loop. Then there exists a dependence from iteration \( J_1 \) to iteration \( J_2 \) caused by these accesses before the application of \( (A, a) \) to \( k \) if and only if there exists a dependence from \( J_1 \) to \( J_2 \) after the application of \( (A, a) \) to \( A \).

**Proof** Trivial, as \( A \) is bijective. \( \square \)
This Lemma immediately leads to the following proposition. 

Proposition 2 The application of a non-singular data transformation is always valid.

Note that in case of singular data transformations the above proposition does not hold. For example, if we want to check that transforming a high dimensional temporary array into a low dimensional one is valid, we need to check that every assignment to an element of the second array is not live anymore before we reassign that element. In general, sophisticated data flow analysis is required for deciding the validity of singular data transformations. In future work we hope to give a precise characterisation of the validity requirements for these data transformations.

6 Unifying Loop and Data Transformations

There has been references to and parallels drawn with loop transformations within this paper. In fact, there is a strong symmetry between data and loop transformations. This can be really seen if we look at the effect of either transformation from an access rather than a loop or data storage point of view.

First, let us recapitulate some of the issues involved. For the sake of clarity, we ignore shifts in the discussion below. Given a loop $L$ and an array $A$, we denote the iteration space of $L$ by $P(L)$ and the index space of $A$ by $P(A)$. Then an access can be seen as mapping $(i,n) : P(L) \rightarrow P(A)$. If we apply a transformation $T$ to $L$ we obtain a new loop $L'$, and we can view the transformation as a mapping $T : P(L) \rightarrow P(L')$. On the other hand, if we apply a transformation $A$ to $A$ we obtain a new array $A'$, and we can view the transformation $A : P(A) \rightarrow P(A')$. In order to obtain the bounds for $L'$ and $A'$, we need to use $T^{-1}$ and $A^{-1}$, respectively, for expressing the new iterators or indices in terms of the old iterators and indices, respectively. However, for transforming accesses, in case of a loop transformation we need to go from $P(L')$ to $P(L)$, using the inverse of the loop transformation, $T^{-1}$, and then to $P(A)$, using the access matrix. Hence $T^{-1}$ is a right hand operating transformation. In case of a data transformation, we need to go from $P(L)$ to $P(A)$, using the access matrix, and then to $P(A')$, using the data transformation $A$. So $A$ is a left hand operating transformation. Thus a loop and a data transformation together have the following effect upon a data access matrix:

$$ U \rightarrow A U T^{-1} $$

Frequently, both transformations can have the same effect on one particular access. In such a situation the choice of a data or loop transformation will depend on legality criteria being satisfied and any side effect on the rest of the program.

One very useful outcome of these observations is that the global impact of data transformations can frequently be resolved locally by loop transformations. Consider the example in figure 1. The initialisation of B is not shown here. Let us assume it is of the form shown in column 1 of figure 4. On propagating the alignment transformation the initialisation code will be transformed into that of column 2. Unfortunately this has very poor access stride, so we have shifted the problem from one place to another. However, the access stride can be improved locally by a loop transformation $T = U^{-1} A U$ and $t = c$ to give the code in column 3. In this case there are no issues due to data dependence and conflicting access requirements.

By considering unimodular loop and data transformations at the same time, we can apply both elementary row and column operations to an access. This enables us, for example, to bring accesses into diagonal form assuming data dependences allow for it. This means that every subscript uses one distinct loop variable. It can be advantageous to obtain such a form as there are alignment and distribution algorithms that require this form of access.

Further investigation of the relationship between loop and data transformations is beyond the scope of this paper and will be the subject of future work.

7 Applications

In this section the use of data transformations in two applications is outlined. Each subsection will describe the optimisation goal and a constructive algorithm to generate the appropriate transformation. Experimental results on the SGI Challenge and KSR-1 are presented to illustrate the usefulness of data transformations by comparing them to other techniques.

7.1 Spatial Locality

Stride-1 access of arrays is important in architectures with some form of cache structure since this allows exploitation of spatial locality. Assuming a column-major ordering of arrays, the innermost iterator should ideally access only the first index, if any. Accessing the second or subsequent index should be avoided as it will destroy spatial locality [11]. Assuming that the dimension of the referenced array is $n$ and that the nesting depth of the surrounding loops is $m$, the desired structure of an access matrix $U$ for spatial locality is therefore

$$ U = \begin{bmatrix} y & 1 & 0_1 \\ 1 & 0_2 & 0 \end{bmatrix} $$

(6)

where $y$ is a $1 \times \ell$ row matrix, $Y$ an $(n-1) \times \ell$ matrix, $o_1$ an $1 \times (m-\ell-1)$ null row matrix, $o_2$ an $(n-1) \times 1$ null column matrix and $O$ a $(m-1) \times (m-\ell-1)$ null matrix. The $1$ is the right-most non-null entry in row one and corresponds to the most inner iterator referenced by the first index. For good spatial locality, that iterator must not be referred to by any other index, i.e., there should be just one column entry for that iterator. This implies that $o_2$ must be null. More than one entry will imply diagonal or skewed access.

For example, assuming an iterator ordering $k,j,i$, then $A(k,j+k)$ and $B(i+j,j)$ are good accesses, while $C(i+k,j)$ is not. This can be seen from their access matrices:

$$ U_k = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad U_j = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad U_i = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} $$

where only the access matrix of $C$ does not have the desired structure. In both $A(k,j+k)$ and $B(i+j,j)$ the innermost iterator is only referenced in the first index while the access $C(i+k,j)$ refers to $i$ in both indices.

7.1.1 Algorithm

To achieve the desired structure of the access matrix requires reordering of the matrix by elementary row or column operations which directly correspond to left and right hand
transformations, i.e., data and loop transformations. The
algorithm in figure 6 describes a constructive technique for
optimising spatial locality using data transformations. It
attempts to create a matrix with an anti-diagonal structure
using rowwise operations. Such a matrix has the desired
property of equation (6). Reducing an array to row-echelon
form creates an array with diagonal elements. We wish an
anti-diagonal structure which is achieved by first post-
multiplying the access matrix with the reverse permutation
T, where

\[
T = \begin{bmatrix}
0 & 0 & \cdots & 0 & 1 \\
0 & 1 & \cdots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
1 & \cdots & \cdots & \cdots & 0 \\
\end{bmatrix}
\]  

(7)

1. Let T be the reverse permutation of identity, given in
equation 7.
2. Apply rowwise operations X to reduce LT to integer
row echelon form.
3. Let A = X and let (A, 0) be the initial data
transformation.
4. Determine lower bounds \( a_k \) and upper bounds \( b_k \)
of each index \( i_k \) after applying \( (A, 0) \), using Fourier-
Matzkin elimination.
5. Let \( a_k = -b_k + 1 \) for all \( k \), yielding the shift vector
a. This shift vector guarantees that all lower bounds become
1, as is the default in Fortran.
6. Apply the data transformation \( (A, a) \) to the array.

Figure 5: Spatial Locality Algorithm

To illustrate this algorithm consider the access to B in
figure 1 and assume that the array B is of size \( (2n \times n) \). Applying
the algorithm, we obtain the initial data transformation

\[
A = \begin{bmatrix}
0 & 1 \\
1 & -1 \\
\end{bmatrix}
\]

This gives the following lower and upper bounds after
applying A:

1 \( \leq i_1 \leq n \) \quad and \quad -n + 1 \( \leq i_2 \leq 2n - 1 \)

The new lower bounds of the array are negative after applying
A, so the shift vector a is applied to restore the indices to
positive values, where

\[
a = \begin{bmatrix}
0 \\
n \\
\end{bmatrix}
\]

The updated program for \( n = 4 \) is given in column 3 of
figure 1.

This algorithm is applicable to more complex references
such as \( A(i+j+k, i+k, i-j+k) \). If the above algorithm is applied
we have

\[
X = \begin{bmatrix}
0 & 1 & 0 \\
1 & -1 & 0 \\
1 & -2 & 1 \\
\end{bmatrix}
\]

Hence \( A(i+j+k, i+k, i-j+k) \rightarrow A(2i+j+k, j, 0) \), where the
innermost iterator k has stride 1 access on k. Although
such references are unlikely to be as complex in practice, it
demonstrates the generality of the above scheme.

Rank Deficient Accesses If data transformations to improve
spatial locality on a particular access are restricted to
permutations then they can be seen as a formulation of
global index reordering or a change from row to column-
major storage. In the general case for arbitrary affine ac-
cesses they can be seen as the data dual of access normalisation
[17]. Data transformations can be shown to be strictly
more powerful than loop transformations in enhancing spa-
tial locality, as there exists accesses, namely rank deficient
accesses, where no loop transformation can improve its spa-
tial locality, but a data transformation can.

Rank deficient accesses, where two rows are dependent, can-
ot be removed by column operations (loop transforma-
tions). This causes a problem if the two dependent rows
contain an entry in the innermost iterator column.

For example, consider the access \( A(i+j, i+j) \). A loop trans-
formation may map

\[
A(i+j, i+j) \rightarrow A(i', i')
\]

while a data transformation may have the following effect:

\[
A(i+j, i+j) \rightarrow A(i+j, 0)
\]

In this case data transformations can remove diagonal access
causing references which cannot be removed by loop
transformations. Column/row-major reordering is also unable
to remove the poor stride access in rank deficient accesses.
Figure 6: Spatial Locality

7.1.2 Results

To illustrate the benefit of the above algorithm the three versions of the program shown in figure 1 were executed and the results shown in figure 6 on one processor of an SGI Challenge. Parallel versions could not be evaluated due to the serialising data dependences within the program. Changing from column to row-major layout provides a small improvement on large data sizes as is improved access to array A. The poor stride on B, however, prevents significant performance improvement. The data transformation approach, based on the algorithm in figure 5, improves the stride access to both arrays and shows up to a factor of 5 improvement over the existing methods.

7.2 False Sharing and Page Replication

In this section we will look at false sharing and page replication issues in Cache-only Memory Architectures (COMAs). We consider an SFMD execution model with the owner computes rule, where data is partitioned across the processors such that each processor has part of the data resident. Note that different parts of a single page may be owned by distinct processors.

Partitioning of arrays along the first index can lead to false sharing and page replication in SVM systems. Let the dimension of the first array index be n, the number of processors p and the size of the coherence unit c. If each processor writes to a distinct contiguous region, false sharing will occur if \( \frac{n}{p} < c \). That is, two or more processors will write to the same page, causing the coherence unit to ping-pong between processors even though there is no data dependence [4].

Difficulties still arise in COMAs when \( \frac{n}{p} > c \) for 2- or higher dimensional arrays, if partitioned on the first index. This means that every processor is allocated a collection of rows of the array. Data is allocated column-wise to pages and each time an element is referenced, the page must be first allocated in memory by the Operating System before it may be accessed. Partitioning on the first index of a 2-dimensional array implies that potentially every column will be accessed.

The figure shows time against varying data sizes for three layouts, rather than temporal performance (1/time) against processor number which is used in the other (parallel) applications.

by every processor and thus all pages must be allocated in each processors attraction memory. For large arrays, there will be extensive cache conflict and performance degradation. For this reason, partitioning by columns is preferred by programmers [20]. However, in those cases where parallelism is only available in the first array index existing loop based approaches cannot alleviate this problem without loss of parallelism.

7.2.1 Algorithm

Real \( A(200,400) \), \( B(200,400) \)

\[
\text{Do } k = 1, n
\]
\[
\text{Do } i = 1, n
\]
\[
A(i,k) = A(i,k-1) + B(1,k)
\]
\[
\text{Enddo}
\]
\[
\text{Enddo}
\]

Figure 7: ADI fragment: before transformation

Consider the program fragment from the ADI example [7] shown in figure 7. Here parallelism is available only in the first index. Interchanging loops will not change this but will actually lead to worse stride access. If, however, the storage of all arrays were transposed, leading to a columnwise data partitioning throughout the program, then page replication is avoided. However, this is at the expense of stride access as the innermost iterator now strides through the second array index. This is easily rectified by loop interchange and is shown in figure 8.

Real \( A(400,200) \), \( B(400,200) \)

\[
\text{Do } i = 1, n
\]
\[
\text{Do } k = 1, n
\]
\[
A(k,i) = A(k-1,i) + B(k,i)
\]
\[
\text{Enddo}
\]
\[
\text{Enddo}
\]

Figure 8: ADI fragment: after transformation

The algorithm in figure 9 performs a data storage transformation program wide if partitioning is upon the first index. It takes as input a partitioning matrix \( P \). With each partitioning matrix \( P \) a serialising matrix \( S \) is associated such that

\[
P + S = I
\]

Given the formal indices \( L, PL \) and \( SL \) define those indices to be partitioned and serialised, respectively. A full rank \( P \) implies that all the indices will be partitioned; a null matrix means the array is not distributed.

The partitioning matrix \( P \) is checked to see if it partitions the first index and if so, whether there are other indices which could be partitioned instead. This can only be true if
\( \mathcal{P} \) is not full rank. If successful, the partitioning matrix, all array accesses and data storage are updated. In the algorithm \( E_{1,j} \) denotes the identity matrix \( I \) with columns \( 1 \) and \( j \) interchanged. Its application swaps the first row of a matrix with the \( j \)th, hence the \( j \)th row will now be partitioned rather than the first.

if \( \mathcal{P} \) is not full rank and \( \mathcal{P} \mathcal{I}[1] \neq 0 \)
then
Let \( j \) be the first null row of \( \mathcal{P} \)
Let \( \mathcal{A} = E_{1,j} \)
Apply \( \mathcal{A}, \mathcal{O} \) to all accesses
Let \( \mathcal{P}' = \mathcal{A} \mathcal{P} \mathcal{A}^{-1} \)
endif

Figure 9: Page Replication Algorithm

In the example given in figure 7, we have before transformation the following values of \( \mathcal{P} \) and access matrix/vector pairs, where we number accesses from left to right

\[
\mathcal{P} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}
\]

\( (\mathcal{U}_{1}, u_{1}) = \left( \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right) \)

\( (\mathcal{U}_{2}, u_{2}) = \left( \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right) \)

\( (\mathcal{U}_{3}, u_{3}) = \left( \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right) \)

Given the formal indices \( \mathcal{I} \), \( \mathcal{P} \mathcal{I} \) defines those indices which will be partitioned. In this case, \( \mathcal{P} \mathcal{I}[1] \), the first element of \( \mathcal{P} \mathcal{I} \), equals 6. Hence the first index is to be partitioned. \( \mathcal{P} \) is not full rank as one of its rows is null; it is a 2-dimensional matrix of rank 1. If \( \mathcal{P} \) were full rank, this would imply that all indices were to be partitioned and there would be no point in reordering indices. In the case of a rank deficient partition matrix, other indices can be chosen to partition.

As there is only one null row in \( \mathcal{P} \), \( j = 2 \) and \( \mathcal{A} \) is readily found:

\[
\mathcal{A} = E_{1,2} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}
\]

Application of the data transformation \( \mathcal{A}, \mathcal{O} \) gives the new accesses

\( (\mathcal{U}_{1}'', u_{1}'') = \left( \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right) \)

\( (\mathcal{U}_{2}'', u_{2}'') = \left( \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix} \right) \)

\( (\mathcal{U}_{3}'', u_{3}'') = \left( \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right) \)

The new index domain (storage) for both arrays is

\( \mathcal{I}' = [i_{1}', i_{2}] = [i_{1}, i_{1}] \)

and

\[
1 \leq i_{1} \leq 400, 1 \leq i_{2} \leq 200
\]

Finally, the new partitioning matrix \( \mathcal{P}' \) is formed where

\[
\mathcal{P}' = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}
\]

Therefore, the new index to partition on is given by

\[
\mathcal{P}' \mathcal{I}' = \begin{bmatrix} 0 \\ i_{2}' \end{bmatrix}
\]

which is the second index, avoiding the problem of false sharing and page replication\(^4\). After loop interchange is performed, the program in figure 8 is found.

This algorithm was implemented in the MARS compiler\([3]\) after the data partitioning decision is made. Although it may also be necessary to update loop structures using loop interchange in cases where bad stride occurs, it is not necessary to include this in the present transformation, as there already is an existing module in MARS which reorders loop for locality after partitioning and mapping has occurred\([3]\).

7.2.2 Results

To demonstrate the importance of this algorithm we compared it with the existing MARS compiler and KAP, an existing commercial auto-parallelising compiler. We applied KAP, MARS and the new MARS compiler in which this transformation is implemented to the ADI code fragment and ran the resulting parallel programs on the KSR over data sizes 128, 256, 512, 1024 and 2048 and number of processors 1, 2, 4, 8, 16 and 32. The results are shown in figures 10, 11, 12, 13 and 14, respectively. Applying the data transformation can give dramatic performance improvements by up to a factor of 50 improvement over the old version of MARS and a factor of up to 60 over KAP. As the data size increase, so does the amount of page replication and the number of cache conflicts in the attraction memory leading to severe performance degradation in the case of \( n = 2048 \); it is here that the transformation gives the greatest improvement.

\(^{4}\)The local node programs after partitioning are not shown here.
8 Conclusion

In this paper we have described the use of non-singular data transformations in optimising program performance. We have described their impact on program structure and developed validity tests for their application. Constructive algorithms have been developed to generate transformations that can significantly improve program performance outperforming existing techniques. Non-singular data transformations unify existing transformations and are strongly connected with non-singular loop transformations. Future work will investigate the connection between these classes of transformation and extend the class of transformations examined beyond non-singular matrices.

References


Reshaping Access Patterns
for Improving Data Locality

Aart J.C. Bik
Computer Science Department, Indiana University
Lindley Hall 215, Bloomington, Indiana 47405-4101, USA
ajcbik@cs.indiana.edu

Peter M.W. Knijnenburg
Computer Science Department, Leiden University
Niels Bohrweg 1, 2333 CA Leiden, the Netherlands
peterk@cs.leidenuniv.nl

Abstract

In this paper, we present a method to construct a loop transformation that simultaneously reshapes the access patterns of several occurrences of multidimensional arrays along certain desired access directions. First, the method determines a direction through the original iteration space along which these desired access directions are induced. Subsequently, a unimodular transformation is constructed that changes the iteration space traversal accordingly. Finally, data dependences are accounted for. In particular, this reshaping method can be used to improve data locality in a program.

1 Introduction

It is well known that the performance of programs heavily depends on the degree of locality exhibited by the accesses in the program. If a program exhibits good temporal and spatial locality, most memory references will be to data present in the cache. Since the speed of processors and main memory tend to grow apart more and more, good locality is of ever greater importance to performance. It is therefore important to be able to restructure the program in such a way that locality is improved. The traditional way to improve locality is by blocking or tiling iteration spaces. In this approach the iteration space is cut up into pieces so that each piece only refers to a subset of the array that fits into the cache. Several approaches try to estimate the amount of locality in inner-loops, for example by means of windows [10], to drive heuristic strategies for applying blocking, permutation and fusion of loops [10, 8, 7, 11]. Recently, sophisticated multilevel blocking algorithms have been devised that take into account registers and multilevel caches [15]. Other approaches try to construct unimodular loop transformations. In the context of NUMA architectures, Li and Pingali [14] construct a transformation for access normalization in order to localize array references; in this approach (affine) index functions are mapped onto target loop indices in such a way that inner target loops only refer to array elements local to the processor. Wolff and Lam [19] present an advanced method incorporating estimates of locality that includes unimodular transformations and tiling. Recently, algorithms that compute both computation and data decomposition have been proposed [1, 2]. These algorithms, however, focus on communication minimization rather than cache behavior optimization, like the present paper. More recently, approaches that try to rearrange the layout of arrays for improving data locality have been proposed [16]. Note that this last approach is different from the previously cited ones in that the loop structures in the program remain unaffected.

In this paper, we present a novel approach to improving spatial locality, namely, we want to reshape access patterns of arrays. The objective is to reshape access patterns so that they become either column-wise or row-wise, depending on whether column-major (e.g., Fortran), or row-major storage of arrays is used. In this approach, the iteration space of a loop nest is transformed in such a way that the innermost loop
will access arrays along columns or rows, respectively. In case of column-major storage, this means that the first index expression in an array reference should contain the loop index of the innermost loop, and the other index expressions should not. If we are able to reshape access patterns in this way, then obviously the locality of a program will be improved. We present a general method to construct a valid loop transformation that simultaneously reshapes the access pattern of several occurrences of (possibly different) multi-dimensional arrays along certain desired access directions. The method can be successively applied to each individual loop nest, thereby improving the efficiency of the program as a whole.

As far as the authors are aware, the present approach is new. In a number of cases, the proposed construction yields a loop interchange, like the strategies reported in [8]. The difference is that in our case this interchange is computed, whereas in [8] is selected by a heuristic. The construction in [14] constructs a transformation based on affine access functions present in the loop, like our approach. However, in the latter work an entire index expression is mapped onto one loop index in the target loop. This means, in general, that all index expressions in one array reference in the target loop may refer to this loop index. This should be contrasted to our approach in which only one index expression will refer to the innermost loop index, namely, the leftmost expression in case of column-major storage. This is obviously preferable for locality purposes. An interesting extension of the present work is to study how the techniques of blocking and reshaping access patterns can be combined. Reshaping access patterns could be used as a preprocessing transformation for the blocking algorithms cited above: the transformed loop will have compact reference windows in inner loops.

Improving data locality is only one application of access pattern reshaping. We are able to present the construction of the transformation in a general setting: the transformation resizes each access pattern along an arbitrary preferred access direction. This means that we are able to reshape access patterns so that they become column-wise, row-wise, along a diagonal, or along an arbitrary other line through the array. There are several reasons for taking this approach. First, enforcing column-wise or row-wise access are just special instances of the same technique. Therefore, this technique can be incorporated in compilers for arbitrary languages. Second, in [16] a notion of data transformation is discussed that changes the layout of an array and propagates this change throughout the program. Such a data transformation is given by a non-singular matrix \( D \). In that paper, it has been observed that loop transformations and data transformation are, in a certain sense, the dual of each other: given a data transformation \( D \) and a loop transformation \( U \), an access matrix \( W \) is transformed to \( DWU^{-1} \). Since we are able to transform a loop so that the accesses to an array will be diagonal-wise, for instance, after this transformation the layout of the array can be skewed by means of a data transformation so that the accesses become column-wise in the transformed array. In this way, the access to an array can be made column-wise, even if a single loop or data transformation cannot be found that does the job. The precise connection between data and loop transformations as well as how they can be used together for improving locality will be the topic of future research. Finally, the reshaping method has been used in a completely different context, namely, in the conversion of dense to sparse code [8]. Because the entries in a sparse storage scheme can usually only be generated efficiently along one particular direction, here it is also important to be able to change the access direction along arbitrary preferred directions. In [6], a preliminary version of the reshaping method of this paper (restricted to two-dimensional arrays) has been used for this purpose.

The outline of the paper is as follows. Preliminaries are given first in section 2. The reshaping method is presented in section 3. In section 4 we explore how this reshaping method can be used to enhance data locality in perfectly nested loop and we provide some measurements. Finally, we state some conclusions in section 5.

2 Preliminaries

In this section, we first give some definitions. Thereafter, we discuss the effect of applying a unimodular transformation to a perfectly nested loop on the order in which the iterations of this loop are executed.

2.1 Definitions

We will use the following definitions [3, 20, 21]. The relation \( \prec_k \) on \( \mathbb{Z}^d \) is defined as follows, where \( \vec{i}, \vec{j} \in \mathbb{Z}^d \) and \( 1 \leq k \leq d \):

\[
\vec{i} \prec_k \vec{j} \iff i_1 = j_1, \ldots, i_{k-1} = j_{k-1}, i_k < j_k
\]
The lexicographical order \( \prec \) on \( \mathbb{Z}^d \) is defined in terms of this relation:

\[
\vec{i} \prec \vec{j} \Leftrightarrow \exists 1 \leq k \leq d : i_k < j_k
\]

We have \( \vec{i} \preceq \vec{j} \), if either \( \vec{i} \preceq \vec{j} \) or \( \vec{i} = \vec{j} \). The relations \( \succ \)'s, \( \preceq \)'s and \( \succeq \)'s are defined similarly.

In this paper, we will use the framework of unimodular transformations [3, 9, 13]. Given a perfectly nested loop with stride-1 DO-loops, index vector \( \vec{i} = (i_1, \ldots, i_d) \), and iteration space \( IS \subseteq \mathbb{Z}^d \), any combination of loop interchanging, loop skewing and loop reversal (see e.g. [17, 20, 21]) that transforms this loop into another loop with index vector \( \vec{i}' = (i'_1, \ldots, i'_d) \) and iteration space \( IS' \subseteq \mathbb{Z}^d \) can be modeled by a linear transformation \( \vec{U} : IS \rightarrow IS' \) that is defined by a \( d \times d \) unimodular matrix \( U \).

This means that an iteration \( \vec{i} \in IS \) is mapped to an iteration \( \vec{i}' \in IS' \) as follows:

\[
\vec{i}' = U \vec{i}
\]

We assume that all data dependences occurring in the original loop are represented by a set of distance vectors \( D \subseteq \mathbb{Z}^d \), where \( d \in D \) implies that a statement instance executed during iteration \( \vec{j} \in IS \) depends on an instance executed during iteration \( \vec{i} \in IS \), where \( \vec{j} = \vec{i} + \vec{d} \). Note that \( \vec{d} \geq \vec{0} \) is holds for all \( \vec{d} \in D \).

Application of a loop transformation defined by \( U \) is valid if and only if \( U\vec{d} \geq \vec{0} \) holds for all \( \vec{d} \in D \).

Converting the original loop with index vector \( \vec{i} \) into the target loop with index vector \( \vec{i}' \) using a unimodular transformation \( U \) is implemented by (i) rewriting the loop-body of the original loop according to the equation \( \vec{i} = U^{-1} \vec{i}' \), and (ii) generating new loops that induce a lexicographical traversal of the target iteration space. Under the assumption that the original iteration space \( IS \) can be represented by an integer system \( A \vec{i} \leq \vec{b} \), the second step consists of generating loop bounds that induce a lexicographical traversal of all discrete points in \( AU^{-1} \vec{i}' \leq \vec{b} \). This generation is usually accomplished using Fourier-Motzkin elimination.

Fourier-Motzkin elimination [3, 20] can be used to test the consistency of a reasonably small system of linear inequalities \( A \vec{x} \leq \vec{b} \), or to convert this system into a form in which the lower and upper bounds of each variable \( x_i \) are expressed in terms of the variables \( x_1, \ldots, x_{i-1} \) only. Consistency indicates that the system has at least one rational solution, and can be used as a necessary (but not sufficient) condition under which an integer solution exists. An extension to Fourier-Motzkin elimination that can test for the existence of integer solutions is given by the Omega-test [18].

2.2 Iteration Space Traversal

Because iterations in both the original and target iteration space are traversed in lexicographical order, a unimodular transformation effectively changes the order in which iterations are executed. Since the index vectors of the original and target loop are related as \( \vec{i}' = U \vec{i} \), we can make the following observations:

(a) Let \( \vec{u} \in \mathbb{Z}^d \) denote the first row of \( \vec{U} \). Then, for fixed \( i'_1 = i_1 \), the more inner DO-loops of the target loop execute iterations in \( IS' \) that correspond to all iterations of \( IS \) in the hyperplane \( \{ \vec{i} \in \mathbb{Z}^d | \vec{u} \cdot \vec{i} = 0 \} \).

(b) Let \( \vec{u}' \in \mathbb{Z}^d \) denote the last column of the inverse matrix \( \vec{U}^{-1} \). Then, for fixed \( i'_d = i_d \), the innermost DO-loop of the target loop successively executes iterations in \( IS' \) that correspond to iterations of \( IS \) along a single straight line with direction \( \vec{u}' \).

Inner loop concurrentization methods [4, 12, 20] exploit the first observation by enforcing a successive traversal of hyperplanes in the original iteration space in successive iterations of the outermost DO-loop of the target loop such that all iterations in each individual hyperplane are completely independent. The second observation, on the other hand, forms the basis of the reshaping method presented in this paper.

**EXAMPLE:** Consider the following matrices:

\[
U = \begin{pmatrix}
1 & 1 & 1 \\
1 & 0 & 0 \\
0 & 1 & 0
\end{pmatrix}
\quad \quad
U^{-1} = \begin{pmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
1 & -1 & -1
\end{pmatrix}
\]

These unimodular matrices define the following loop transformation:

\[
\begin{array}{l}
D \quad i_1 = 0, \quad 5p \\
D \quad i_2 = 0, \quad 5p - 1 \\
D \quad i_3 = 0, \quad 5p \\
B \quad (i_1, i_2, i_3) \rightarrow + \quad (i'_1, i'_2, i'_3) \\
\end{array}
\quad
\begin{array}{l}
D \quad i'_1 = 0, \quad 10p \\
D \quad i'_2 = 0, \quad \text{max}(5p, i'_1) \\
D \quad i'_3 = \text{max}(5p, i'_1 - i'_2 - 5p), \\
\end{array}
\quad
\begin{array}{l}
\text{ENDDO} \quad (i'_1, i'_2, i'_3) \\
\text{ENDDO} \quad (i'_1, i'_2, i'_3) \\
\text{ENDDO} \quad (i'_1, i'_2, i'_3)
\end{array}
\]

\(1\) An integral matrix \( U \) that satisfies \( \det(U) = \pm 1 \) is called a unimodular matrix.
The transformation of the original iteration space IS into the target iteration space IS' is illustrated in figure 1. In figure 2, we illustrate how the original iteration space IS is effectively traversed after the transformation.

Observation (a) reveals that because the first row of \( U \) is \((1, 1, 1)\), for fixed \( I_1' = i_3' \) the target loop executes all iterations corresponding to iterations of IS in the plane \( I_1 + I_2 + I_3 = i_3' \). For fixed \( I_1' = i_1' \) and \( I_2' = i_2' \), the target loop executes all iterations in the intersection of the planes \( I_1 + I_2 + I_3 = i_3' \) and \( I_1 = i_1' \).

Observation (b) reveals that this intersection forms a straight line with direction \((0, 1, -1)\), i.e., the last column of \( U^{-1} \). For example, the iterations \((0, 0, 1)\) and \((0, 1, 0)\) in IS are mapped to the successively executed iterations \((1, 0, 0)\) and \((1, 0, 1)\) in IS'.

3 Reshaping Access Patterns

In this section, we first state the objective of reshaping. Thereafter, we present the reshaping method in detail.

3.1 Objective of Reshaping

Suppose that the subscripts of an occurrence of a \( d \)-dimensional array in a perfectly nested loop with index vector \( \mathbf{i} = (I_1, \ldots, I_d) \) are represented by the following affine transformation \( F: \mathbb{Z}^d \rightarrow \mathbb{Z}^n \):

\[ F(\mathbf{i}) = \mathbf{v} + \mathbf{W} \mathbf{i} \]

The last column of \( \mathbf{W} \) is called the access direction \( \mathbf{v} \in \mathbb{Z}^n \) of this occurrence: the array is traversed along lines with direction \( \mathbf{f} = \mathbf{W}(0, \ldots, 0, 1)^T \). Any loop transformation defined by a \( d \times d \) unimodular matrix \( U \) changes this access direction by replacing the original subscripts \( F(\mathbf{i}) = \mathbf{v} + \mathbf{W} \mathbf{i} \) with \( F'(\mathbf{i}) = \mathbf{v} + \mathbf{W} U^{-1} \).

We say that a loop transformation reshapes the access patterns of this occurrence along a preferred access direction \( \mathbf{s} \in \mathbb{Z}^n \), if this latter vector and the resulting true access direction \( \mathbf{s}' \in \mathbb{Z}^n \) are linearly dependent. This implies that the following equation holds for some \( \lambda \).
\[ \mathbf{F}' = WU^{-1}(0, \ldots, 0, 1)^T = \lambda \cdot \mathbf{s} \]

We can use the following proposition, which says that the collection of all vectors that are linearly dependent on \( \mathbf{s} \) are given by the kernel of \( \mathcal{S}(\mathbf{s}) \).

**Proposition 3.1** Let \( \mathbf{r}, \mathbf{s} \in \mathbb{Z}^n \), where \( \mathbf{s} \neq 0 \). Let \( k \) be the least index such that \( s_k \neq 0 \). Then \( \exists \lambda, \mathbf{r} = \lambda \cdot \mathbf{s} \) if \( \mathcal{S}(\mathbf{s}) \mathbf{r} = 0 \), where \( \mathcal{S}(\mathbf{s}) \) is the following \((n-1) \times n\) matrix:

\[
\mathcal{S}(\mathbf{s}) = \begin{pmatrix}
s_k & -s_1 & \cdots & -s_{k-1} & s_k \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
0 & \cdots & 0 & s_k & -s_n
\end{pmatrix}
\]

**Proof** \( \Rightarrow \) Suppose \( \mathbf{r} = \lambda \cdot \mathbf{s} \) or \( r_1 = \lambda \cdot s_1, \ldots, r_n = \lambda \cdot s_n \). Then necessarily \( \lambda = r_k/s_k \) since by assumption \( s_k \neq 0 \). Substituting \( r_k/s_k \) for \( \lambda \) and rewriting yields the second equation.

\( \Leftarrow \) Conversely, suppose \( \mathcal{S}(\mathbf{s}) \mathbf{r} = 0 \). This can be rewritten to \( s_k \cdot r_i - s_i \cdot r_k = 0 \) for all \( 1 \leq i \leq n \). Hence, for all \( i \), \( r_i = (r_k/s_k) \cdot s_i \). Hence, by setting \( \lambda = r_k/s_k \), we infer that there exists \( \lambda \) such that for all \( i \), \( r_i = \lambda \cdot s_i \).

Note that we allow reshaping that results in 'scalar-wise' access patterns (which will improve temporal locality), since the equation is trivially satisfied for the vector \( \mathbf{r} = 0 \).

### 3.2 Method of Reshaping

In this section, we present a general method to construct a valid unimodular transformation that simultaneously reshapes the access patterns of \( m \) occurrences of, possibly different, multi-dimensional arrays in a single perfectly nested loop with index vector \( \mathbf{l} = (I_1, \ldots, I_d)^T \) along certain preferred access directions. We assume that the ith occurrence is a \( n_i \)-dimensional array with the subscript function \( F_i(\mathbf{l}) = \mathbf{i} + W_i \mathbf{l} \) (where \( W_i \) is a \( n_i \times d \) matrix) and that the preferred access direction for this occurrence is given by \( \mathbf{s}_i \in \mathbb{Z}^{n_i} \). Moreover, we assume the data dependence structure of this loop is represented by a set \( D \subseteq \mathbb{Z}^d \) of dependence distance vectors.

#### 3.2.1 Preferred Iteration Directions

According to proposition 3.1, a loop transformation defined by a \( d \times d \) unimodular matrix \( U \) reshapes the access patterns of the ith occurrence in the loop along the preferred access direction \( \mathbf{s}_i \in \mathbb{Z}^{n_i} \), if the last column \( \mathbf{a} \in \mathbb{Z}^d \) of the inverse matrix \( U^{-1} \) satisfies the following objective:

\[ \mathcal{S}(\mathbf{s}_i) \cdot W_i \mathbf{a} = 0 \]

After application of such a transformation, the original iteration space is effectively traversed along straight lines with the direction \( \mathbf{a} \in \mathbb{Z}^d \) (cf. observation (b) in section 2). Therefore, this vector is referred to as the **preferred iteration direction**. All access patterns are reshaped simultaneously if \( \mathbf{a} \in \mathbb{Z}^d \) satisfies \( \mathcal{S}_a = 0 \) for the following objective matrix:

\[
S = \begin{pmatrix}
S(s_1) \\
\vdots \\
S(s_m)
\end{pmatrix}
\begin{pmatrix}
W_1 \\
\vdots \\
W_m
\end{pmatrix}
\]

Since the elements in each row or column of a unimodular matrix must be relatively prime, any integer solution of \( S\mathbf{a} = 0 \) with \( \gcd(a_1, \ldots, a_d) = 1 \) may be used as preferred iteration direction, which gives rise to the following set \( \Delta \subseteq \mathbb{Z}^d \):

\[ \Delta = \{ \mathbf{a} \in \mathbb{Z}^d \mid S\mathbf{a} = 0 \text{ and } \gcd(a_1, \ldots, a_d) = 1 \} \]

Using the integer echelon reduction algorithm of [3, p32-39] to compute the unimodular matrix \( R \) such that \( RST \) is in echelon form (yielding \( r = \text{rank}(S) \) as side-effect), all integer solutions of the homogeneous integer system \( S\mathbf{a} = 0 \) are given by the following formula for arbitrary \( \lambda_i \in \mathbb{Z} \) [3, p59-66]:

\[
\mathbf{a} = [\underbrace{0, \ldots, 0}_{r}, \underbrace{\lambda_{r+1}, \ldots, \lambda_d}_{d-r}] \mathbf{R}^T
\]

This observation provides a simple condition for the existence of a (possibly invalid) unimodular transformation that performs the preferred reshaping:

**Proposition 3.2** There exists a \( d \times d \) unimodular matrix \( U \) for which the last column \( \mathbf{a} \in \mathbb{Z}^d \) of \( U^{-1} \) satisfies \( S\mathbf{a} = 0 \) for some \( n \times d \) matrix \( S \) if and only if \( \text{rank}(S) < d \).
If \( r = d \), then the reshaping method fails. If, on the other hand, \( r < d \) holds, then the last \( d - r \) rows of \( R \) form a basis of the linear subspace consisting of all solutions of the homogeneous system.

We now have to choose a vector \( \vec{a} \in \Delta \) for constructing a unimodular matrix \( U^{-1} \) having \( \vec{a} \) as its last column. It is not immediately obvious which \( \vec{a} \) to choose: there may be infinitely many solutions with components that are relatively prime, and we want to find one such that eventually we can construct a corresponding valid transformation. We have found that for practical cases, it is sufficient to restrict our attention to the set of basis vectors:

\[
\Delta' = \{ \vec{a} = [0, \ldots, 0, 1, 0, \ldots, 0] R \}^T \mid r < k \leq d \}
\]

If none of these can be used to construct a valid transformation (see the next section), the reshaping method fails.

However, there remain cases in which this heuristic fails because access patterns can only be reshaped using an \( \alpha \in \Delta - \Delta' \).

3.2.2 Construction of Valid Transformation

For any \( \vec{a} \in \Delta' \), we can use a completion method [3][5, p15-19] to construct a \( d \times d \) unimodular matrix \( U \) for which the last column of \( U^{-1} \) consists of this preferred iteration direction. Given these unimodular matrices, we can exploit the fact that for any \((d - 1) \times (d - 1)\) unimodular matrix \( Y \) and integer \( \lambda \in \{-1, +1\} \), the following \( V \) is still a unimodular matrix for which the last column of the inverse is \( \pm \vec{a} \):

\[
V = \begin{pmatrix}
Y & \cdots & 0 \\
0 & \cdots & 0 & z \\
0 & \cdots & 0 & z
\end{pmatrix}
\]

\[
V^{-1} = U^{-1}
\begin{pmatrix}
Y^{-1} & \cdots & 0 \\
0 & \cdots & 0 & z \\
0 & \cdots & 0 & z
\end{pmatrix}
\]

Consequently, if for a given \( \vec{a} \in \Delta' \) and corresponding \( U \) we can construct a unimodular matrix \( Y \) and integer \( \lambda \in \{-1, +1\} \) that define a matrix \( V \) with \( \vec{v} \geq \vec{0} \) for all \( \vec{d} \in \mathcal{D} \), then a valid transformation performing the preferred reshaping has been found. Otherwise, another \( \vec{a} \in \Delta' \) is tried until either this construction is successful, or the set \( \Delta' \) has been exhausted. In the latter case, the reshaping method fails.

For \( \mathcal{D} = \emptyset \), the construction is trivial, since we can use the loop transformation defined by \( Y = I \) (viz. \( Y = I \)). Otherwise, we define the following set \( \mathcal{D} \subseteq \mathcal{D} \):

\[
\mathcal{D} = \{ \vec{d} \in \mathcal{D} \mid U\vec{d} = (0, \ldots, 0, \lambda)^T, \lambda \in \mathbb{Z} \}
\]

The set of dependence distance vectors \( \mathcal{D} \) can be partitioned into \( \mathcal{D} = \mathcal{D} \cup \mathcal{D} - \mathcal{D} \). We will see that dependence vectors in the former set determine the selection of the integer \( \lambda \in \{-1, +1\} \), whereas dependence distance vectors in the latter set must be dealt with during the construction of the matrix \( Y \).

Lemma 3.3 Given a \( d \times d \) unimodular matrix \( U \) and a set \( \mathcal{D} \subseteq \mathbb{Z}^d \) where each \( \vec{d} \in \mathcal{D} \) satisfies \( \vec{d} \geq \vec{0} \) and \( U\vec{d} = (0, \ldots, 0, \lambda)^T \) for some \( \lambda \in \mathbb{Z} \), then either:

1. for all \( \vec{d} \in \mathcal{D} \) we have \( U\vec{d} \geq \vec{0} \), or
2. for all \( \vec{d} \in \mathcal{D} \) we have \( U\vec{d} \leq \vec{0} \).

Proof Assume that there are \( \vec{d}_1, \vec{d}_2 \in \mathcal{D} \) such that

- \( U\vec{d}_1 = (0, \ldots, 0, \lambda_1)^T \) for \( \lambda_1 > 0 \), and
- \( U\vec{d}_2 = (0, \ldots, 0, \lambda_2)^T \) for \( \lambda_2 < 0 \).

Obviously, \( \vec{d}_1 = U^{-1}(0, \ldots, 0, \lambda_1)^T \) implies that both \( \vec{d}_1 = \lambda_1 \cdot \bar{v} \) and \( \vec{d}_2 = \lambda_2 \cdot \bar{v} \) hold for a fixed \( \bar{v} \neq \vec{0} \), namely, for \( \bar{v} \) equal to the last column of \( U^{-1} \). This is in contradiction with the assumption that both \( \vec{d}_1 \) and \( \vec{d}_2 \) are lexicographically positive. Consequently, either:

1. for all \( \vec{d} \in \mathcal{D} \), \( U\vec{d} = (0, \ldots, 0, \lambda)^T \) for some \( \lambda \geq 0 \), i.e., \( U\vec{d} \geq \vec{0} \), or
2. for all \( \vec{d} \in \mathcal{D} \), \( U\vec{d} = (0, \ldots, 0, \lambda)^T \) for some \( \lambda \leq 0 \), i.e., \( U\vec{d} \leq \vec{0} \).

This lemma provides a convenient method to select a suitable integer \( \lambda \in \{-1, +1\} \).

Proposition 3.4 Given a \( d \times d \) unimodular matrix \( U \) and a set \( \mathcal{D} \subseteq \mathbb{Z}^d \) where each \( \vec{d} \in \mathcal{D} \) satisfies \( \vec{d} \geq \vec{0} \).
and \( U\vec{d} = (0, \ldots, 0, \lambda)^T \) for some \( \lambda \in \mathbb{Z} \), then for any \((d - 1) \times (d - 1)\) matrix \( Y \) there exists an integer \( z \in \{-1, +1\} \) such that (2) defines a matrix \( V \) with \( V \vec{d} \geq \vec{0} \) for all \( \vec{d} \in \mathcal{D} \).

**Proof** Since the first \( d - 1 \) components of \( V \vec{d} \) are zero for any \( \vec{d} \in \mathcal{D} \) and any matrix \( Y \), the proposition follows directly from Lemma 3.3: We select \( z = -1 \) if \( U\vec{d} < \vec{0} \) for any \( \vec{d} \in \mathcal{D} \), and \( z = 1 \) otherwise.

Using Proposition 3.4, we have found a method for finding the required \( z \in \{-1, +1\} \). In order to construct the unimodular matrix \( Y \), only the dependence distance vectors in \( \mathcal{D} - \mathcal{D} \) need to be used. Let \( \vec{U} \) consist of the first \( d - 1 \) rows of the unimodular matrix \( U \):

\[
\vec{U} = \begin{pmatrix}
1 & \cdots & 0 \\
\vdots & \ddots & \vdots \\
1 & \cdots & 0
\end{pmatrix}
\begin{pmatrix}
y_1 \\
\vdots \\
y_{d-1}
\end{pmatrix} \leq \begin{pmatrix}
-1 \\
\vdots \\
-1
\end{pmatrix}
\]

We use Fourier-Motzkin elimination to test the consistency of this system. The construction of \( Y \) fails if the system is inconsistent (e.g., we have \( 1 \leq y_1 \) and \( y_1 \leq -1 \) for the example above). If the system is consistent, however, any rational solution \( \vec{y}^* \) (which can be obtained as side-effect of the elimination) can be scaled to an integer solution of which the components are relatively prime (viz. \( \vec{y} = \lambda \cdot \vec{y}^* \) for \( \lambda \) equal to the least common multiple of the denominators). Therefore, a completion method as described in [3][5, p15-19] can be used to construct a unimodular matrix \( Y \) with \( \vec{y} \in \mathbb{Z}^{d-1} \) as its first row. Obviously, \( Y\vec{U} \vec{d} \geq \vec{0} \) holds for all \( \vec{d} \in \mathcal{D} - \mathcal{D} \).

### 3.3 Summary

Summarizing, the following steps are applied. First, we construct the objective matrix \( S \) using the subscripts and preferred access directions. If \( \text{rank}(S) = d \), then the reshaping method fails (Proposition 3.2).

Otherwise, for each preferred iteration direction \( \vec{d} \in \Delta^t \), a corresponding \( d \times d \) unimodular matrix \( U \) of which the last column of \( U^{-1} \) is equal to this direction is constructed and the following steps are applied until either the method succeeds or this set has been exhausted, in which case the reshaping method fails:

- Select \( z = -1 \) if \( U\vec{d} < \vec{0} \) for any \( \vec{d} \in \mathcal{D} \) or select \( z = 1 \) otherwise (Proposition 3.4).
- Find a \( \vec{y} \in \mathbb{Z}^{d-1} \) with \( \gcd(y_1, \ldots, y_{d-1}) = 1 \) such that \( \vec{y} \cdot \vec{U} \vec{d} > 0 \) for all \( \vec{d} \in \mathcal{D} - \mathcal{D} \).

If this \( \vec{y} \in \mathbb{Z}^{d-1} \) exists, then a completion method [3][5, p15-19] is used to construct a unimodular matrix \( Y \) with this vector as first row. The resulting matrix \( Y \) together with \( Y^{-1} \) and integer \( z \in \{-1, +1\} \) define matrices \( V \) and \( V^{-1} \) according to (2) such that \( V\vec{d} \geq \vec{0} \) holds for all \( \vec{d} \in \mathcal{D} \). Hence, application of the loop transformation defined by this matrix is valid and reshapes the access patterns of each \( t \)th occurrence along the preferred direction \( \vec{s}_t \in \mathbb{Z}^2 \).
EXAMPLE: Consider the following triple loop:

\[
\begin{align*}
\text{DO } & I_2 = 1, 3 \\
\text{DO } & I_3 = 10, 16 \\
\text{DO } & I_4 = 10, 15 \\
A(I_2 + I_3) & = \ldots \\
B(2I_2 + I_3, 10I_2 + I_3) & = \ldots \\
C(I_2 - 3I_3) & = \ldots
\end{align*}
\]

We assume that the data dependencies of this loop are represented by the following set:

\[ D = \{(1, 0, 0)^T, (0, 1, 0)^T, (0, 0, 1)^T, (3, 1, -6)^T\} \]

Now, suppose that row-wise access patterns are desired for these three occurrences, i.e., \( z = (0, 1)^T \) for \( 1 \leq i \leq 3 \). Reshaping the access patterns accordingly seems to be a non-trivial task at first sight. However, the reshaping method proceeds as follows.

First, the objective matrix \( S \) is constructed:

\[
S = \begin{pmatrix}
1 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
1 & 3 & 1 \\
2 & 0 & 1 \\
0 & 2 & 0 \\
0 & 0 & 1
\end{pmatrix}
\]

Echelon reduction of \( S^T \) yields the following form for \( E = RS^T \):

\[
\begin{pmatrix}
1 & 1 & 0 \\
0 & 1 & 1 \\
0 & 0 & 0
\end{pmatrix} = \begin{pmatrix}
0 & 0 & 1 \\
1 & 0 & -1 \\
0 & 1 & 0
\end{pmatrix} \begin{pmatrix}
1 & 2 & 1 \\
3 & 0 & -3 \\
1 & 1 & 0
\end{pmatrix}
\]

Because \( \text{rank}(S) = 2 \), all integer solutions of the homogeneous system \( S\tilde{a} = \tilde{0} \) are given by \( \tilde{a} = ((0, 0, \lambda_3) \tilde{a})^T \) for arbitrary \( \lambda_3 \in \mathbb{Z} \). Hence, we have \( \Delta' = \{(3, 1, -6)^T\} \). The following matrices are constructed with a completion method as described in [3][5, p15-19]:

\[
U = \begin{pmatrix}
-1 & 3 & 0 \\
0 & 6 & 1 \\
0 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
-1 & 0 & 3 \\
0 & 0 & 1 \\
0 & 1 & -6
\end{pmatrix}
\]

Let \( \bar{U} \) denote the matrix consisting of the first 2 rows of \( U \). Then \( \bar{D} = \{\tilde{d} \in D | \tilde{d} \bar{U} = \tilde{0}\} \) is equal to \( \{(3, 1, -6)^T\} \). Since \( U(3, 1, -6)^T = (0, 0, 1)^T \), we select \( z = 1 \).

Finding an integer vector \( \tilde{y} \in \mathbb{Z}^2 \) such that \( \tilde{y} \cdot \bar{U}\tilde{d} > 0 \) for all \( \tilde{d} \in D - \bar{D} \) is equivalent to solving the following system of linear inequalities:

\[
\begin{pmatrix}
-1 & 0 \\
3 & -6 \\
0 & -1
\end{pmatrix} \begin{pmatrix}
y_1 \\
y_2
\end{pmatrix} \leq \begin{pmatrix}
-1 \\
-1 \\
-1
\end{pmatrix}
\]

Obviously, this system is consistent, and a solution \( (1, 1)^T \) can be used directly for \( \tilde{y} \). The following matrices results:

\[
Y = \begin{pmatrix}
1 & 1 \\
-1 & 0
\end{pmatrix} \\
Y^{-1} = \begin{pmatrix}
0 & -1 \\
1 & 1
\end{pmatrix}
\]

These matrices and \( z = 1 \) give rise to the following \( V \) and \( V^{-1} \):

\[
V = \begin{pmatrix}
1 & 3 & 1 \\
0 & -1 & 3 \\
0 & 1 & 0
\end{pmatrix} \\
V^{-1} = \begin{pmatrix}
0 & 0 & 1 \\
1 & 1 & -6
\end{pmatrix}
\]

Application of the loop transformation defined by \( V \) yields the following target loop in which row-wise access patterns result for all occurrences:

\[
\begin{align*}
\text{DO } & I_1 = 23, 39 \\
\text{DO } & I_2' = \text{MAX}(1I_2, [1I_2 + (15, 16, 17, 1)], \text{MIN}(-1I_2, [-20, 33, -1I_2]), \text{MIN}([1I_2 + 16I_3/6], [(1I_2 + 10I_3/3)], [1I_2 + 15I_3/3])) \\
\text{DO } & I_3' = \text{MAX}((1I_3 + 16I_2/6), [(1I_3 + 10I_3/3)], [1I_3 + 15I_3/3]) \\
\text{DO } & I_4 = \text{MAX}(1I_4, 3I_4 + 1I_3) \\
\text{DO } & I_5 = \text{MAX}(1I_5 + 2I_4) \\
A(I_1', I_2', I_3', I_4') & = \ldots \\
B(2I_1' - I_2', 2I_2') & = \ldots \\
C(I_2' - I_3', I_3' + I_1' - 5I_2') & = \ldots
\end{align*}
\]

4 Improving Data Locality

Because Fortran uses column-major storage of arrays, accessing arrays along columns enhances the spatial locality of a program. Hence, a straightforward approach to improve the performance of a program is to apply the reshaping method to each perfectly nested loop in a Fortran program, where we set the preferred access direction of each multi-dimensional array to \( (1, 0, \ldots, 0)^T \). If the reshaping method fails for a particular loop nest, either because the reshaping is impossible or because data dependences prohibit the reshaping, then we can simply continue with the next perfectly nested loop in the program, or we can re-apply the reshaping method to the same loop with a smaller set of occurrences, for example, by discarding an occurrence with unique subscripts before occurrences having identical subscripts. If the reshaping method still fails, another occurrence is discarded.
until reshaping is successful or all occurrences have been discarded.

Below, we present some experiments that have been conducted on an SGI Indy 4600. All fragments are compiled by the native FORTRAN compiler using the flag -02. Although in the experiments the value of N varies, the actual shape of each array is kept constant (and this shape envelops the parts that may be accessed for all values).

**EXAMPLE:** Consider the following implementation of the operation C ← C + AB.

```fortran
DO I1 = 1, N
  DO I2 = 1, N
    DO I3 = 1, N
      C(I1, I3) = C(I1, I3) + A(I1, I3) * B(I3, I2)
    ENDCCDO
  ENDCCDO
ENDCCDO
```

First, we construct the objective matrix S using the preferred access direction \( \vec{s} = (1,0)^T \) for all occurrences:

\[
\begin{pmatrix}
0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0
\end{pmatrix}
\begin{pmatrix}
W_C \\
W_C \\
W_A \\
W_B
\end{pmatrix}
\]

Hence, the objective matrix has the following form:

\[
S = \begin{pmatrix}
0 & 1 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0
\end{pmatrix}
\]

Echelon reduction of \( S^T \) yields the following form for \( E = R S^T \):

\[
\begin{pmatrix}
1 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0
\end{pmatrix}
= \begin{pmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
0 & 0 & 0
\end{pmatrix} S^T
\]

Because \( \text{rank}(S) = 2 \), we obtain the set \( \Delta' = \{(1,0,0)^T\} \). Under the assumption that the data dependencies due to the accumulation of elements in array C may be ignored, the following matrices can be used directly to apply the appropriate reshaping:

\[
U = \begin{pmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
1 & 0 & 0
\end{pmatrix}
\quad U^{-1} = \begin{pmatrix}
0 & 0 & 1 \\
1 & 0 & 0 \\
0 & 1 & 0
\end{pmatrix}
\]

This transformation gives rise to a simpler loop permutation where \( t_1 \) becomes the index of the outermost DO-loop. In Figure 3, we show the execution times of the original and reshaped version for varying values of \( N \).

**EXAMPLE:** Consider the following triple loop in which some three-dimensional arrays and some two-dimensional arrays are referenced:

```fortran
DO I1 = 1, N
  DO I2 = 1, N
    DO I3 = 1, N
      C(I1, I2, I3) = B(I1, I2, I3) + A(I1, I2)
    ENDCCDO
  ENDCCDO
ENDCCDO
```

The objective matrix \( S \) is constructed as illustrated below:

\[
\begin{pmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
0 & 0 & 1 \\
0 & 0 & 0
\end{pmatrix}
\begin{pmatrix}
W_C \\
W_C \\
W_A \\
W_B
\end{pmatrix}
\]

Hence, the resulting objective matrix has the following form:

\[
S = \begin{pmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
0 & 0 & 1 \\
0 & 0 & 0
\end{pmatrix}
\]

Echelon reduction of \( S^T \), however, reveals that \( \text{rank}(S) = 3 \), indicating that making all access patterns column-wise is impossible. However, after we...
Figure 4: Performance of first Triple Loop

discard the occurrence of A from consideration (which is simply implemented by discarding the last row of S), then the rank(S) becomes 2 and the reshaping method can be applied successfully. In this case, the same loop permutation as in the previous example results, making all except the access patterns of A column-wise. In figure 4, we show the performance improvement obtained by this transformation.

Finally, we briefly sketch an extension to our approach that is given by iterating the construction. First observe that, in case of column-wise reshaping, all reshaped access matrices become of the form, where \( w_{1d} \neq 0 \):

\[
W = \begin{pmatrix}
  w_{11} & \cdots & w_{1d-1} & w_{1d} \\
  w_{21} & \cdots & w_{2d-1} & 0 \\
  \vdots & \ddots & \vdots & \vdots \\
  w_{c1} & \cdots & w_{c,d-1} & 0
\end{pmatrix}
\]

or:

\[
W = \begin{pmatrix}
  w_{11} & \cdots & w_{1,d-1} & 0 \\
  w_{21} & \cdots & w_{2,d-1} & 0 \\
  \vdots & \ddots & \vdots & \vdots \\
  w_{c1} & \cdots & w_{c,d-1} & 0
\end{pmatrix}
\]

Ideally, in the later case we would like to make \( w_{2,d-1} = 0, \ldots, w_{c,d-1} = 0 \) as well. In order to obtain this form we re-apply the reshaping method to all occurrences for which \( W \) has a zero last column. First, we construct a \((d-1) \times (d-1)\) unimodular transformation \( V \) using the method from section 3 with as input the reduced access matrices of these occurrences, formed by dropping the last column from the original access matrices. Subsequently, this loop transformation is applied to the \( d-1 \) innermost loops. Likewise, we construct a \((d-2) \times (d-2)\) unimodular transformation \( V' \) for handling the next iterator, and so on. It is easy to see that in this way we can obtain an 'ideal' access matrix, as described above.

**EXAMPLE:** Consider the following triple loop:

\[
\begin{align*}
\text{DO } I_1 & = 1, N \\
\text{DO } I_2 & = 1, N \\
& \text{DO } I_3 = 1, N \\
A(I_1+I_2, I_2, I_1-I_3) & = B(I_3, I_2) \times C(1-I_3, I_2)
\end{align*}
\]

Making the access patterns of all occurrences column-wise gives rise to the following objective matrix:

\[
S = \begin{pmatrix}
0 & 1 & 0 \\
1 & 0 & -1 \\
0 & 1 & 0 \\
0 & 1 & 0
\end{pmatrix}
\]

Echelon reduction of \( S^T \) yields the following form for \( E = RS^T \):

\[
\begin{pmatrix}
1 & 0 & 1 & 1 \\
0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
= \begin{pmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
1 & 0 & 1
\end{pmatrix}
\]

Because rank(S) = 2, we obtain the set \( \Delta' = \{(1, 0, 1)^T\} \). Since there are no data dependences in the loop, the following matrices can be used directly to apply the reshaping:

\[
U = \begin{pmatrix}
1 & 0 & -1 \\
0 & 1 & 0 \\
1 & 0 & 0
\end{pmatrix}
U^{-1} = \begin{pmatrix}
0 & 0 & 1 \\
0 & 1 & 0 \\
-1 & 0 & 1
\end{pmatrix}
\]

Applying the loop transformation defined by these matrices results in the fragment shown below:

\[
\begin{align*}
\text{DO } I_1 & = 1, N \\
\text{DO } I_2 & = 1, N \\
& \text{DO } I_3 = \max(I_1+1, I_2), \min(I_1+1, N) \\
A(I_3+I_2, I_1, I_1-I_3) & = B(I_3, I_2) \times C(1-I_3, I_2)
\end{align*}
\]

The resulting access matrices reveal that column-wise access patterns have been obtained for both the occurrences of A and B:
\[ W_k = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}, \quad W_k = \begin{pmatrix} -1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \]

The access patterns of \( C \), on the other hand, have become scalar-wise. In fact, this occurrence is accessed along rows in the outermost two loops:

\[ W_C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \]

Applying the reshaping method to the reduced access matrix gives rise to the following objective matrix:

\[ S = [0, 1] \]

Hence, we obtain \( \Delta' = \{(1,0)^T\} \), which gives rise to a simple loop interchanging of the outermost two DO-loops:

\[ U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad U^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \]

In figure 5, we show the execution time of the original, first reshaped, and final reshaped fragment for varying values of \( k \). This experiment clearly shows how an extra iteration of reshaping can be used to further improve the performance of a loop. Moreover, the experiment shows that the overhead associated with evaluating MIN and MAX functions is out-weighted by the improvements obtained with respect to the memory hierarchy.

Figure 5: Performance of second Triple Loop

this method can be combined with methods that are aimed at enhancing temporal locality, like loop blocking. Moreover, future research will be aimed at combining the reshaping method of this paper with the framework of data structure transformations presented in [16].

5 Conclusions

In this paper, we have presented the construction of a valid transformation of a perfectly nested loop that simultaneously reshapes the access patterns of several occurrences of multi-dimensional arrays along certain desired access directions. In particular, we have shown how this reshaping method can be used to improve data locality in programs. If for a particular loop nest the reshaping method fails, the compiler re-tries the reshaping method with a smaller collection of occurrences. Moreover, we have shown that in some cases, iterating the reshaping method can further improve performance.

Note that in this paper we have focused on improving spatial locality by making loops refer to neighboring elements in successive iterations. Obviously,


The Loop Parallelizer LooPo

Martin Griebl and Christian Lengauer

Fakultät für Mathematik und Informatik
Universität Passau, D–94030 Passau, Germany
email: {griebl,lengauer}@fmi.uni-passau.de
WWW: http://www.uni-passau.de/~lengauer

Abstract. We report on a prototype for testing different methods of space-time mapping loop nests. LooPo admits perfect or imperfect loop nests in a number of imperative languages, takes data dependences from the user or derives them itself from the source code, provides a choice of strategies for scheduling and allocating the loop nest’s iterations, and produces synchronous or asynchronous parallel target code for shared-memory or distributed-memory machines.

1 Why LooPo?

LooPo is not meant to be yet another parallelizing compiler. It is a prototype system whose purpose is to assist us in the research on and evaluation of space-time mapping methods for loop parallelization. To that end, it implements the complete path from executable source code to executable target code, with switches for choosing alternative methods. At present, we provide several inequality solving methods, several schedulers and several methods of code generation, one dependence analyzer (we are working on a second) and one allocator.

LooPo is in the public domain and uses only freely available software to ensure easy distribution. It runs on Sun workstations under SunOS 4.1.x and Solaris 2.x, and on PCs under Linux.

2 Theoretical Background

The space-time mapping methods we study are based on the polytope model of nested loops, and on some extensions of it.

The polytope model supports the static parallelization of nested for loops [18]. The model represents the atomic iteration steps of $d$ perfectly nested for loops as the points of a polytope in $\mathbb{Z}^d$; each loop defines the extent of the polytope in one dimension. The faces of the polytope correspond to the loop bounds; they are all known at compile time. This enables the compile-time discovery of maximal parallelism—relative to the choices available in the method, which are limited by the data dependences of the source program and by the requirement that the space-time mapping defining the parallelism be affine.
for $i := 0$ to $n$ do
for $j := 0$ to $i + 2$ do
$A(i, j) := A(i - 1, j) + A(i, j - 1)$

for $t := 0$ to $2n + 2$ do
forall $p := \max(0, t - n)$ to $\min(t, \lfloor t/2 \rfloor + 1)$ do
$A(t - p, p) := A(t - p - 1, p) + A(t - p, p - 1)$

Fig. 1. Loop parallelization in the polytope model.

2.1 The basic polytope method

Parallelization in the polytope model proceeds as follows (Figure 1).

First, one transforms $d$ perfectly nested source loops into a $d$-dimensional polytope, where each loop defines the extent of the polytope in one dimension. We call this polytope the index space. Each point of the index space represents one iteration step of the loop nest. The coordinates of the point are given by the values of the loop indices at that step; the vector of these coordinates is called the index vector.

Then one applies an affine coordinate transformation, the space-time mapping, to the polytope and obtains another polytope in which some dimensions enumerate space and the others enumerate time. We call the transformed polytope the target space. The standard techniques for space-time mapping in the polytope model require perfectly nested loops (all statements belong to the innermost loop body) and uniform dependences [14]. In this case, all points of the index space have identical dependence vectors which are independent of the problem size. LooPo accepts also imperfect loop nests and affine, not only uniform dependences.

Finally, one translates this polytope back to a nest of target loops, where each space dimension becomes a parallel loop and each time dimension becomes a sequential loop.

2.2 Extension to affine-by-statement scheduling

The basic model represents only perfectly nested loops. This severe restriction can be relaxed by applying the basic method to every single statement separately rather than to the body as a whole. Thus, every statement has its own index space, index vector, space-time mapping and target space [5, 9, 15]. An
operation in the program is identified by a statement together with its index 
vector. Of course, the feature of statementwise space-time mapping complicates 
the generation of target code significantly (cf. Section 3.7).

2.3 Extension to loop nests containing while loops 

One of the main future goals of LooPo is to be able to parallelize loop nests 
containing while loops. This entails a rather serious departure from the polytope 
model [19]. The theory and method have been worked out [12, 13], but the 
implementation in LooPo is still under way.

3 The Structure of LooPo 

LooPo traverses a sequence of steps which transform the source program to an 
executable parallel target program. There are modules for scanning and parsing, 
(in)equality solving, dependence analysis, scheduling, allocation and target code 
generation. A front end provides the user with a graphical interface by which he/she can control LooPo. There is also a graphical tool for displaying index 
spaces and iteration dependence graphs of loop nests.

Subsequently, we give a very brief overview of the system. See our Web pages 
on LooPo for more details.

3.1 The Front End 

The front end enables the user to invoke modules in sequence by mouse clicks 
on buttons (Fig. 2). Many modules provide a choice of methods via an options 
window, e.g., the Scheduler module (Fig. 3). Each module takes as input the 
output of the previous module, which is stored in a file. The user can intervene 
at any point by specifying hand-picked solutions in these files.

3.2 The Input to LooPo 

LooPo accepts (possibly imperfect) loop nests in C, Fortran or a number of 
abstract loop notations, and declarations of functions, procedures and symbolic 
constants. By stating explicit dependences, one can experiment with the space-
time mapping of non-executable programs, i.e., programs with incomplete loop 
odies.

3.3 The Inequality Solvers 

There are several methods for parametric linear programming, which is the cen-
tral mathematical problem of the polyhedron model. We considered the following 
methods for use in LooPo:

Fourier-Motzkin. This is the standard doubly exponential method of polytope 
projection [1].
**Loop Parallelization in the Polytope Model**

**Source Program**

```
CONSTANT n;
DO i=0, n
  DO j=0, i+2
    A(i,j) = A(i, i+1) + A(i,j-1)
  END DO
END DO
```

**Target Program**

```
C Loopo program
C FORTRAN
DO k1=0,2*(n+2)
  DOPAR pi=CELL(MAX(0,ti-n)), FLOOR(MIN(2,ti+2))
  A(ti+1,pi)=A(ti-1,pi)+A(ti+1,pi-1)
ENDDO
ENDDO
C end of Loopo program
```

---

**Fig. 2.** LooPo’s main window.

**PIP.** This is Feautrier’s system for parametric integer programming [7]. It proceeds indirectly by transforming the original system of inequations into a dual system and solving that. In principle, it is an extension of the well-known simplex algorithm.

**Weispfenning.** This is another direct method which is only singly exponential [22]. It performs better than Fourier-Motzkin on problems with more than four variables.

**Omega.** The Omega library [21] by Pugh solves linear programs on the basis of Presburger formulas (affine constraints, the usual logical connectives, and existential and universal quantifiers), with efficient heuristics for parallelization.
The current implementation is largely based on PIP; the dependence module offers a choice of PIP or Fourier-Motzkin. Omega will be integrated as an alternative for PIP and Fourier-Motzkin in all modules as soon as possible.

### 3.4 The Dependence Analyzer

At present, LooPo features the dependence analysis method described by Banerjee [1], which makes no distinction between flow, anti and output dependences. We are working on the implementation of Feautrier's method [8], which allows potentially more parallelism, since it eliminates anti and output dependences first and then considers only the remaining flow dependences.
3.5 The schedulers

Presently, LooPo provides three different automatic schedulers:

**Lamport.** The hyperplane method by Lamport [2, 16] can handle perfectly nested for loops with uniform dependences. It yields a one-dimensional affine schedule for the complete loop body and, as allocation, a projection onto the source axes such that the space-time mapping formed by the combination of schedule and allocation is unimodular.

**Feautrier.** The Feautrier scheduler [9, 10] determines an optimal (concave) schedule for imperfectly nested for loops with affine dependences, at the cost of a longer computation time based on the necessity of dealing with parametric integer linear programming [7]. The resulting schedule for every statement can be multidimensional and piecewise affine.

For a comparison with Lamport’s method, one can call the Feautrier scheduler by iteration (in the case of a perfectly nested input program), which enforces the same schedule for all statements in the loop body.

**Darte/Vivien.** Darte and Vivien proposed a fast scheduler with reasonably good results [6], which can schedule arbitrary loop programs with uniform and non-uniform dependences. It uses a less precise dependence description than the Feautrier scheduler. Therefore, the quality of its schedules is somewhere between that of Lamport’s and Feautrier’s schedules.

3.6 The Allocator

Presently, LooPo provides only Feautrier’s allocation method [11] which determines the placement of operations on virtual processors. It is based on the “owner computes rule” and tries to “cut” dependences by mapping the depending operations to the same processor, starting with dependences in the highest dimensions (greedy heuristic). The allocator does not inspect the schedule, and may therefore generate an allocation in which some dimensions are linearly dependent. Because of LooPo’s modularity, other allocators can be added easily.

3.7 The Target Generator

The target generator consists of two modules: one derives a first, raw form of the target loop nest(s), the other refines it with communications for synchronization and communication.

**The target loops** The loops of the transformed source program are constructed from the index spaces, the dependences, the schedule and the allocation. Note that transformations can be individual for every statement in the source program. The target loops are represented as a parse tree which does not contain any synchronization or communication statements.

The construction of the parse tree takes place in two phases. First, the statements of the source program are transformed individually. In order to achieve
that, the transformation matrices are computed, the target index spaces are constructed, strides and offsets are calculated, and finally all expressions in every statement are re-indexed according to their transformed coordinates. Second, the transformed statements in their respective index spaces (parts) are combined to a single target program. Aside from the two options of synchronous or asynchronous code\(^1\), three merging strategies are available [23]:

- The parts are simply combined with a parallel operator, i.e., there are several separate loop nests which are assumed to be executed in parallel.
- All index spaces of the parts are merged to one index space to construct a perfect loop nest. The statements inside the body are guarded by ifs to determine at run time whether or not they are to be executed.
- To avoid ifs at run time, one determines at compile time precisely in which regions which statements must be executed, and enumerates these regions successively. This is only implemented for the outermost dimension; the others are merged by the run time method. The problem is that determining the relative order of \(n\) parameters in the loop bounds requires a case distinction of order \(O(n!)\) in the target program!

**Synchronization and communication** The parse tree representing the target loops is then translated to one of various possible output languages, e.g., parallel C, Fortran, or Parix-C\(^2\).

Synchronization and communication is added if the user so desires. The target program (with communications) is executable on any PARIX machine. We plan to add a code generator for the MasPar.

4 Related Work

There are several other systems which support space-time mapping in the polytope model:

**ALPHA** [17] allows the space-time mapping of affine recurrence equations. The target code consists of uniform space-time recurrences, i.e., a representation of the target polyhedron as a functional program.

**Bouclettes** [3] parallelizes uniform loop nests. The target code is in High-Performance Fortran (HPF).

**PAF** (not present in the literature as of yet) is being developed by Feautrier and implements his recent space-time mapping methods [9, 10, 11]. PAF returns, on input of an (im)perfect loop nest, a space-time mapping in the form of a (piecewise) affine expression and generates (at present, only for unimodular space-time mappings) target code in Fortran for the MasPar, Cray Y-MP and CM-2.

---

\(^1\) In **synchronous** code the sequential loops are outermost, in **asynchronous** code innermost. Banerjee calls this horizontal and vertical parallelism [2].

\(^2\) **PARIX** is an operating system for massively parallel computers with distributed memory by the PARSYTEC company based on the SPMD programming model [20].
There are also larger projects which are not based on the polytope model:

*Crystal* [4] yields target code for a number of machines (iPSC, NCUBE, Connection Machines, iWarp). The system imposes fewer restrictions on the input, and its use requires significantly more interaction from the user than the systems above.

*SUIf* [24] is the most ambitious of all projects listed here. It offers a level of intermediate code, which can be subjected to diverse program transformations; many optimization algorithms are implemented. SUIf gives highest priority not to parallelism but to the locality of data in a multi-level storage hierarchy, based in experiences with the DASH, a multiprocessor prototype.

LooPo can be used as a platform for experimenting with any step of the parallelization process in the polyhedron model; anybody interested in one special aspect of the parallelization can plug his/her own module into LooPo and obtain a customized source-to-source compilation environment. The central data structures (restricting the applicability of LooPo) are—according to the method—polyhedra and piecewise affine functions.

5 Conclusions

LooPo is a platform for comparing various parallelization techniques, based on the space-time mapping idea.

Our first tests showed that there are two main restrictions limiting the applicability of LooPo in practice. The first is the lack of conditional statements in the current version of LooPo. This will be fixed soon.

The other limitation is more deeply connected with the use of the polytope model for space-time mapping. The polytope model offers very precise analysis and scheduling techniques. However, these techniques are based on integer linear programming which is a difficult task. We have had to learn that the (in)equation solvers are the crucial factor of the parallelizer: they do not only consume most of the compilation time but even give up very frequently without computing a solution.

The implementation of the back end module for architecture-specific adaptations is still under way. Hence, we regret to be unable to present performance studies of the various implemented parallelization techniques at this time.

Acknowledgements

We thank the LooPo team: Nils Ellmenreich, Peter Faber, Max Geigl, Robert Günz, Harald Keimer, Radko Kubias, Wolfgang Meisl, Martina Schumergruber, Sabine Wetzel, Christian Wieninger and Alexander Wüst. Their spirit made this project much more than an exercise in software development. We also thank Jean-François Collard, who has been a continuous source of energy and encouragement.

This work is part of the DFG project RecuR and received travel funds from the DAAD exchange program PROCOPE.
References


HPF Implementation Techniques
A Unified Tiling Approach for Out-Of-Core Computations

M. Kandemir
CIS Dept., Syracuse University, Syracuse, NY 13244
mtk@top.cis.syr.edu

R. Bordawekar
CACR, Caltech, Pasadena, CA 91125
rajesh@cacr.caltech.edu

A. Choudhary†
ECE Dept., Northwestern University, Evanston, IL 60208-3118
choudhar@ece.nwu.edu

J. Ramanujam
ECE Dept., Louisiana State University, Baton Rouge, LA 70803
jxr@ee.lsu.edu

Abstract

This paper describes a framework by which an out-of-core stencil program written in a data-parallel language can be translated into node programs in a distributed-memory message-passing machine with explicit I/O and communication. We focus on a technique called Data Space Tiling to group data elements into slabs that can fit into memories of processors. Methods to choose legal tile shapes under several constraints and deadlock-free scheduling of tiles are investigated. Our approach is unified in the sense that it can be applied to both FORALL loops and the loops that involve flow-dependences.

1 Introduction and Related Work

Since, today, almost every processor has some kind of memory hiernachy organized into layers with different costs, compiler optimizations to reduce memory access costs are important. Tiling, one such optimization, was first used by Abu-Sufah et al. [Abu81] in order to optimize loop nests in a paging-memory system. The later applications were generally on cache memories and registers [Vol89, WL91]. In [RS92] a number of loop iterations were aggregated into tiles that execute atomically without any synchronization in a distributed-memory message-passing machine. Irigoin and Triollet [IT88] introduce tiles which are atomic, identical and bounded. Within this context different (and sometimes contradictory) optimization criteria have been offered to choose the best tile shape and size [RS92, SD90, BD93].

The orientation of this paper is different from those of the previous works in one important aspect: We tile the data that reside on disks; that is, we address so called out-of-core problem. The primary data structures for the programs reside on disks and the programs explicitly read from and write into disks. We call the unit of transfer between disk and memory a Data Tile and the technique to schedule read and writes Data

†This work was supported in part by NSF Young Investigator AwardCCR-9357840, NSF CCR-9509143 and in part by the Scalable I/O Initiative, contract number DABT63-94-C-0049 from Defense Advanced Research Projects Agency (DARPA) administered by US Army at Ft. Huachuca. The work of J. Ramanujam was supported in part by an NSF Young Investigator Award CCR-9457763 and by the Louisiana Board of Regents through contract LEQSF(1991-94)-RD-A-09.

A longer version of this paper may be obtained from http://web.ece.nwu.edu/~choudhar.
Space Tiling. We demonstrate the tradeoffs in choosing good tile shapes for FORALL loops[KL94] and the
loops that contain flow-dependences. Extra File I/O is introduced and scheduling techniques to eliminate it
are presented.

The rest of the paper is organized as follows: Section 2 describes the problem and the underlying model. In
Section 3, reuse vectors and chain vectors are discussed. How tiling parameters are determined is discussed
in Section 4. Section 5 studies scheduling of data tiles and we conclude in Section 6.

2 Problem Description and Our Model

The problem addressed in this paper is to compile applications that use very large amount of data on
distributed memory message-passing architectures. A computation is called Out-Of-Core(OCC) if the data
used by it cannot fit in the memory; that is, parts of data reside in files.

In the rest of the paper, a message passing distributed memory machine is assumed. Out-of-core arrays
are divided by the programmer into local out-of-core arrays each of which is stored on a logical disk attached
to processor. We call this model Local Placement Model (LPM). Each processor has its local out-of-core files
stored on the logical disk attached to it, and data sharing is performed by explicit message passing[Bor98].
During the course of program, parts of the local out-of-core file, called tiles or slabs, are fetched into memory,
the new values are computed and the tile is stored back (if necessary) into appropriate locations in the local
file. Computation Volume of a tile is the number of data points it contains.

Our programming model is based on the data parallel programming paradigm. In this model, parallelism
is achieved by decomposing data among processors. We assume a fixed set of distribution patterns e.g.,
row-block, column-block, block-block. Array partitioning results in each processor storing in memory a local
array associated with each array. In an out-of-core program local arrays are also out-of-core.

3 Preliminaries

3.1 Reuse Vectors and Reuse Matrices

We assume that loop bounds and array subscripts are affine functions of the enclosing loop indices, and all
the statements are inside the deepest loop. Data reuse in a program can be expressed by an integer vector
called Data Reuse Vector [WL91, Li94, Wol96]. If \( \ell_1 \) and \( \ell_2 \) are two iterations that access the same data
element, the reuse vector for this access can be defined as \( \ell = \ell_2 - \ell_1 \). Reuse Matrix[Li94] is a matrix every
column of which is a reuse vector.

Suppose that \( X(f(i)) \) is a reference for an array \( X \) on the LHS of an assignment and \( X(g(i)) \) is a reference
for the same array on the RHS of assignment where \( f(i) = A\ell + c_1 \) and \( g(i) = A\ell + c_2 \). Here \( A \) is the
access (reference) matrix and \( c_1 \) and \( c_2 \) are constant vectors. Data reuse between these two references
can be found by solving \( A\ell_1 + c_1 = A\ell_2 + c_2 \). Then, the temporal reuse vector, \( \ell = \ell_2 - \ell_1 \) can be found
from \( A\ell = c_2 - c_1 \). Note that the reuse matrix captures both flow-dependences and anti-dependences.
Since in a FORALL statement all dependences are resolved as anti-dependences, reuse matrix contains only
anti-dependences. It is convenient to represent reuse matrix \( R \) as \( R = [D; S] \) where \( D \) and \( S \) denote
the matrices that contain flow-dependence and anti-dependence vectors respectively. \( D \) is frequently called Data
Dependence Matrix[ZC90, Wol96].
3.2 Chain Vectors and Chain Matrices

An \( r \)-dimensional array defines an \( r \)-dimensional polyhedron. The vectors that define the relation between data points (array elements) are called Chain Vectors [RS89]. For example, the relation between data points for \( X(i) = X(i-1) + X(i+1) \) can be represented by two chain vectors for each value of \( i \). One of them is in direction 1 whereas the other one is in direction -1. This corresponds informally to the statement in order to compute the new value of \( X(i) \) both \( X(i-1) \) and \( X(i+1) \) are needed. It should be emphasized that the chain vectors, in general, can span different arrays.\(^1\) In this paper we only consider stencil applications, and therefore we assume that there is only one array referenced in the nest and all references to it have the same access matrix.\(^2\) In that case chain vectors can be represented in a graph called Data Space Graph (DSG).

Suppose that \( X(\vec{f}(i)) \) and \( X(\vec{g}(i)) \) are two references for the same array \( X \) and the latter occurs on RHS whereas the former occurs on LHS. Let \( \vec{f}(i) = A\vec{r} + \vec{a}_1 \) and \( \vec{g}(i) = A\vec{r} + \vec{a}_2 \), where \( A \) is the access matrix and \( \vec{a}_1 \) and \( \vec{a}_2 \) are constant vectors.

A data reuse exists between two iterations \( \vec{i}_1 \) and \( \vec{i}_2 \) iff \( A\vec{r} + \vec{a}_1 = A\vec{r} + \vec{a}_2 \Rightarrow A(\vec{i}_2 - \vec{i}_1) = \vec{a}_1 - \vec{a}_2 \Rightarrow A\vec{r} = \vec{a}_1 - \vec{a}_2 \) where \( \vec{r} \) is the reuse vector (it may be dependence or anti-dependence vector). On the other hand, we can define a chain vector \( \vec{i} \) for this reference pair as follows: \( \vec{i} = (A\vec{r} + \vec{a}_1) - (A\vec{r} + \vec{a}_2) = \vec{a}_1 - \vec{a}_2 \). From these two equations we obtain an important relation between \( \vec{i} \) and \( \vec{r} \):

\[
\vec{i} = A\vec{r}
\]

where \( A \) is the access matrix, \( \vec{r} \) is the reuse vector and \( \vec{i} \) is the chain vector. If \( \vec{r} \in D \) we call \( \vec{i} \) an Effective Chain Vector. In other words, an effective chain vector is a chain vector implied by a flow-dependence. A Chain Matrix \( T \) is a matrix every column of which is a vector chain. On the other hand an Effective Chain Matrix \( U \) is a chain matrix every column of which is an effective chain vector. We have now the following relation between \( U \) and \( D \):

\[
U = AD
\]

And for every \( \vec{u} \in U \) and \( \vec{d} \in D \) we have

\[
\vec{u} = A\vec{d}
\]

We assume that access matrix \( A \) is square and invertible. That is, the dimensionality of data space is equal to that of iteration space. As an example suppose that a 2-deep nest has the statement \( X(i,i+j) = X(i-1,i+j) + X(i-1,i+j+2) \). Then \( A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \), \( D = \begin{pmatrix} 1 & 1 \\ -1 & -3 \end{pmatrix} \), and \( U = \begin{pmatrix} 1 & 1 \\ 0 & -2 \end{pmatrix} \).

4 Constraints on the Tile Shape and Size

Let \( n \) be the dimension of the iteration (and data) space and \( m \) be the number of columns of the effective chain matrix \( U \). Every tile shape can be succinctly described by one of two ways. Let \( P_d \) be an \( nxn \) matrix every column of which corresponds to a tile boundary in that dimension. It is known that the columns of \( P_d \) constitute an extreme vector set for the effective chain matrix \( U \) [RS92]. The second way to define a tile is a matrix \( H_d \) every column of which is a vector perpendicular to the tile boundary along that dimension. The relation between \( P_d \) and \( H_d \) is \( P_d = H_d^{-1} \) [TT88, BDDR93]. In the rest of the paper \( P_d \) and \( H_d \) are called tiling matrices.

In general, there are four factors that determine shape and size of a tile on data space:

- **Computation Constraint.** Tile shape must be legal in the sense that all effective chain vector traffic between two tiles must be in one direction (i.e. from one tile to the other.)

---

\(^1\)The chain vectors that span different arrays are called Cross Chain Vectors whereas the chain vectors defined entirely on a single array are called Regular Chain Vectors. In this paper we concentrate only on regular chain vectors.

\(^2\)The references defined that way form a Uniformly Generated Reference Set [GJK98, WL91].
• **I/O Constraint.** Tile shape must be compatible with the file layout.

• **Communication Constraint.** The number of chain vectors (effective or not) going from one tile to the others and the number of tiles that a tile communicates with should be minimized.

• **Memory Constraint.** Tile size cannot be larger than the size of the memory of a node.

Of course, for a loop that contains only anti-dependences, computation constraint does not exist.

### 4.1 Computation Constraint (Legality)

Arbitrary clustering of data space points into tiles might result in effective chain vector cycles between tiles. Tile shapes that produce effective chain vector cycles are called illegal tiles. The reason for illegality is that processing of tiles must be atomic in the sense that a tile must take all the data it requires from outside before computation begins, and all the data required by other tiles should be available after computation terminates. Allowing effective chain vector cycles among tiles violates this requirement. As an example, for the data space graph shown in Figure 1(A), tiles (1) and (2) are legal, while tile (3) is illegal. The arrows on the figure represent effective chain vectors.

Let \( h_1, h_2, \ldots, h_n \) be the rows of the \( H_d \) matrix and \( u_1, u_2, \ldots, u_m \) be the columns of the effective chain matrix \( U \). The legality condition can now be stated as follows [IT88, RS92, BDRR93]:

\[
h_i u_j \geq 0
\]

for \( i = 1, 2, \ldots, n \) and \( j = 1, 2, \ldots, m \). We note that \( H_d \) is not necessarily unique.

![Different Tile Shapes](image)

**Figure 1: Different Tile Shapes**

### 4.2 I/O Constraint

I/O cost of a tile is determined by the number of file accesses (I/O calls) required to read it from disk. Under *column-major* layout assumption, in order to minimize the number of file accesses, number of (sub)-column reads should be minimized. This may not always possible in practice as minimization of I/O calls can lead to illegal tiles or some communication requirements may impose lower bounds on some edges of tile. In Figure 1(A), tile (1) and tile (2) have the same computation volume (8 data points); however I/O cost (number of sub-columns) of tile (1) is 4 while that of tile (2) is 2. Everything else being equal, tile (2) is a better choice than tile (1). Note that this constraint is unique to data space tiling.

### 4.3 Communication Constraint

*Communication Volume* [BDRR93] of a tile is the number of chain vectors (effective or not) going from one tile to the others. Communication volume may be reduced if the set of extreme vectors for chain vectors
(i.e. columns of $P_d$) is a subset of the chain vectors. An important observation is that minimizing the communication volume (cost), in some cases, leads to an increase in the I/O cost. This situation is shown in Figure 1:(B) (taken from [BDRR93]). The tile on the left leads to 5 communications whereas the right one leads to 4 communications. On the other hand, the left one needs 2 I/O calls while the one on the right needs 3 I/O calls. This example clearly demonstrates the tradeoff between I/O and communication constraints. There is another aspect of the communication as well. For any tile, the tile size along each dimension must be larger than the magnitude of the maximum of the corresponding components of chain vectors. This will ensure that all the communicating tiles will be neighbors.

### 4.4 Memory Constraint

Tile size cannot be larger than the size of node memory. We can state this constraint as $M \geq V_{\text{comp}}(T)$ where $M$ is the size of the memory of a single processor and $V_{\text{comp}}(T)$ is the computation volume of data tile $T$.

Let $H_i$ and $P_i$ denote iteration space (IS) tiling matrices while $H_d$ and $P_d$ denote data space (DS) tiling matrices. The proofs of the following theorems can be found elsewhere [KBCR96].

**Theorem:** It is always possible to find an $H_d$ and a $P_d$ in order to tile the data space in a deadlock-free manner provided that $A$ is invertible.

**Theorem:** For any legal $H_i$ (or $P_i$) on DSG, there is a corresponding legal $H_d$ (or $P_d$) on the iteration space provided that $A$ is invertible.

**Theorem:** If access matrix $A$ is unimodular, then the number of integer points in the data tile and that of the corresponding iteration tile are equal; overmore there is one-to-one correspondence between the points of these two tiles.

### 4.5 Choosing Tile Shape and Size

Overall cost of a data tile ($T_{\text{cost}}$) has two components: I/O cost ($T_{\text{I/O}}$) and communication cost ($T_{\text{comm}}$). Consider now the tile and the effective chain vector shown in Figure 2:A. The computation volume of this tile $V_{\text{comp}} = [\tilde{p}_i \tilde{p}_j]$ and the communication volume can be approximated $V_{\text{comm}} = [\tilde{u}_i \tilde{u}_j + \tilde{u}_i \tilde{u}_j]$ [BDRR93]. First we need a few definitions.

- $C_{\text{I/O}} = \text{startup cost for an I/O read (write)}, \ t_{\text{I/O}} = \text{cost of reading (writing) an element from (into) a file}$
- $C_{\text{comm}} = \text{startup cost for communication}, \ t_{\text{comm}} = \text{cost of communicating an element, and} \ K = \text{maximum message length of the machine}$

The overall cost (file read and send communication) is

$$T_{\text{cost}} = p_{11} C_{\text{I/O}} + V_{\text{comp}} t_{\text{I/O}} + \left[ \frac{V_{\text{comm}}}{K} \right] C_{\text{comm}} + V_{\text{comm}} t_{\text{comm}}$$

then the optimization problem is to minimize $T_{\text{cost}}$ under the constraints $V_{\text{comp}} \leq M$ and $P_d^{-1}U \geq 0$. Moreover, all entries of $P_d$ are restricted to be integers. This problem in general is difficult to solve. In the following we present a heuristic that works for restricted cases of $U$ (or $U$ in general). Since $C_{\text{I/O}}$ is the most costly term in overall cost expression, we believe that the communication cost of a tile is of secondary importance as compared with its I/O cost.

Recall that the condition for legality of a data tile represented by tiling matrix $H_d$ is

$$H_d U \geq 0$$
where \( H_d \) is the tiling matrix for the data space and \( U \) is the effective chain matrix. We now consider a restricted version of \( U \): We assume that \( \forall \vec{z} \in U \) is lexicographically positive.

**Theorem:** In two-dimensional case if columns of \( U \) are lexicographically positive, it is always possible to choose a tiling matrix \( P_d \) of the form

\[
P_d = \begin{pmatrix}
1 & 0 \\
d & 1
\end{pmatrix}
\]

(3)

where \( d > 0 \) so that \( H_d U \geq 0 \) (see [RS92] for the proof).

---

**Figure 2:** (A) A Data Tile defined by \( \vec{p}_1 \) and \( \vec{p}_2 \) and an Effective Chain Vector \( \vec{u} \). (B) Legal Tile Shapes. (C) Entries of \( P_d \) on the Data Tile. (D) Legal Tile for \( d = 1 \) on the Data Space. (E) Legal Tile Shapes for \( d = 1 \) and \( d \neq 2 \).

A few legal tile shapes that are specified by such a \( P_d \) matrix are shown in Figure 2:B. Re-scaling \( \vec{p}_1 \) by \( \alpha \) and \( \vec{p}_2 \) by \( \beta \) gives the following re-scaled tiling matrix:

\[
P_d = \begin{pmatrix}
\alpha & 0 \\
\alpha d & \beta
\end{pmatrix}
\]

We now impose our constraints on this specific type of \( P_d \) (see Figure 2:C).

- **Legality Constraint.** \( \alpha \geq \max(\max(\frac{u_{1j}}{u_{1i}}, \frac{u_{2j}}{u_{2i}}), 1) \), \( (u_{1i}, u_{2i}) \neq 0 \) where \( \vec{u}_i = \begin{pmatrix} u_{1i} \\ u_{2i} \end{pmatrix} \) is the \( i \)th effective chain vector.
- **I/O Constraint.** \( \alpha \) should be minimized.
- **Communication Constraint.** \( \alpha \geq \max(|t_{1j}|), \beta \geq \max(|t_{2j}|) \) where \( \vec{t}_j = \begin{pmatrix} t_{1j} \\ t_{2j} \end{pmatrix} \) is the \( j \)th chain vector.
- **Memory Constraint.** \( \alpha \beta \leq M \) where \( M \) is the memory size of a node.

Considering I/O and communication constraints together it is clear that \( \alpha = \max(|t_{1j}|) \). Then considering communication and memory constraints together we have

\[
\max(|t_{2j}|) \leq \beta \leq \frac{M}{\alpha}
\]
From this last expression $\beta$ and from the legality constraint $\epsilon$ can be set to appropriate values.

As an example suppose that $U = T = \begin{pmatrix} 4 & 4 \\ 0 & -4 \end{pmatrix}$ and $M = 32$. Using the constraints given above $\alpha = 4$, $\epsilon \geq 1$ and $4 \leq \beta \leq 8$. In order to utilize available memory as much as possible, we should set $\beta = 8$. Now different values for $\epsilon$ give different solutions all of which have the minimum I/O cost. For example if $\epsilon = 1$ we have $P_d = \begin{pmatrix} 4 & 0 \\ -4 & 8 \end{pmatrix}$, if $\epsilon = 2$ on the other hand we obtain $P_d = \begin{pmatrix} 4 & 0 \\ -8 & 8 \end{pmatrix}$. Figure 2:1E shows I/O optimized legal tiles for $\epsilon = 1$ and $\epsilon = 2$; and Figure 2:1D shows the former one on the data space graph of the example.

5 Optimizing I/O in Stencil Codes

5.1 Translated Code

Translated node program of processor $k$ for a 2-deep nest is as follows:\footnote{Translated code for the FORALL statement is very similar, so it is omitted (see [Bor96]).}

\begin{verbatim}
DO IT = LB1T, UB1T, S1T
  DO JT = LB1T, UB1T, S1T
    read data tile DT defined by $f_k(\text{IT}, \text{JT})$ from local file
    compute the corresponding iteration tile $CT$ for data tile $DT$
    handle communication and storage of boundary data
      DO IE = LB1E, UB1E, S1E
        DO JE = LB1E, UB1E, S1E
          perform computation on $DT$ to compute the new data values
        ENDDO IE
      ENDDO JE
    handle communication and storage of boundary data
    write data tile $DT$ defined by $f_k(\text{IT}, \text{JT})$ into local file
  ENDDO JT
ENDDO IT
\end{verbatim}

In the translated loop nest, loops $IT$ and $JT$ are called tiling loops and loops $IE$ and $JE$ are called element loops. Throughout this section we try to find a suitable scheduling function $f_k$ for a given computation such that overall I/O cost is minimized and potential parallelism is exploited. Note that $LB1E$, $UB1E$, $LB1E$, and $UB1E$ depend on the iteration tile $CT$. In other words, out-of-core compiler first fetches the data tile, then computes the corresponding iteration tile, and after that executes the iterations in the iteration tile using the elements of the data tile. Note also that scheduling function $f_k$ may be different for each processor, and this fact is used to find schedules that eliminate all unnecessary I/O and maximize parallelism.

5.2 Extra File I/O Problem

In this subsection we concentrate on how an out-of-core compiler should automatically schedule tile reads/writes to access data in an efficient manner in a distributed-memory environment. First we consider the case with cross-processor anti-dependences only.

Definition: Any file I/O performed for any purpose except local computation is termed as Extra File I/O [Bor96].
Since extra file I/O is pure overhead, it should be eliminated whenever possible. In out-of-core computations, communication cost can, in general, be negligible for most of the cases when compared with the I/O cost. But inappropriate communication methods can cause extra file I/O; in other words, communication may necessitate extra I/O. The proposed technique tries to eliminate the types of communication that cause extra file I/O. Within this context, the efficacy of a scheduling method will be tested by 1) the amount of overhead it incurs, and 2) the amount of extra memory needed to hold the data to be transferred between tiles. The overhead involved can be divided into two groups: extra file I/O cost and communication cost.

Let us now elaborate on extra file I/O cost. During computation when a non-local data is requested by a processor, there are two possibilities: 1) the data has been brought into memory by the owner processor prior to the request, or 2) the data has not been brought into memory yet. In the first case if a processor finds out that a data will later be requested by another processor, it can keep the data in memory till it is requested. This approach does not cause extra file I/O (if there is enough memory), but it requires extra memory allocation for the data to be transferred. In the second case however, the owner processor should read the data requested by issuing I/O call(s) and send it to the requesting processor. Our scheduling strategy must prevent the occurrence of this second case whenever possible.

Consider now Figure 3 to see the overheads of different scheduling strategies for a square tile of size $S \times S$ (dashed arrows represent the direction of anti-dependence whereas solid arrows indicate execution order). Schedule (1) and (4) do not cause extra file I/O, but require extra memories. Schedule (2) leads to extra file I/O. This is because when, for example, processor 1 needs boundary data from processor 0, processor 0 should issue file read requests in order to read the data. The schedules (3) and (5) do not involve extra file I/O and do not require any extra storage for the data to be transferred. Note also that if there were two anti-dependences in the opposite directions instead of one, then the schedules (1) and (4) would involve extra file I/O as schedule (2).

The next section discusses methods to eliminate extra file I/O.

### 5.3 Scheduling Tiles

We assume two-dimensional data sets and two-dimensional processor grid for the illustrative purposes. In such a grid, every processor has a direction vector $v_i$ with respect to each of its neighbors. As an example, consider Figure 5:A. The direction vector for processor 2 with respect to processor 1 is $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$. In general, direction vector for processor $i$ with respect to processor $j$ is denoted by $v_{ij}$.

Each tile can be represented by a size matrix $T_i$. It is a square matrix diagonal elements of which are the sizes of the tile in the corresponding dimensions and all other elements are zero. Let $O$ be a similar diagonal matrix representing the local out-of-core array in terms of its size. These two matrices can be used to compute an important parameter $f$ called Degree of Freedom and a matrix $F$ called Freedom Matrix. Given a $T_i$ and an $O$, the freedom matrix can be computed as $F = \{s \cdot T_i^{-1} \cdot O\}$. The number of the non-unit diagonal elements of $F$ gives the degree of freedom $f$. Scheduling matrix $G_i$, for processor $i$, determines the order in which
the data tiles are brought into memory for processing. For a nested loop that contains only anti-dependences (e.g. a FORALL statement\(^4\)), tile access patterns and their corresponding scheduling matrices for \( f=1 \) and \( f=2 \) are given in Figure 4:A and Figure 4:B respectively. At the heart of the scheduling algorithm is the following theorem the proof which as well as a generalization to arbitrary grid sizes may be found in [BCR95].

**Theorem:** In a two by two processor grid, schedules \( G_i \) and \( G_j \) eliminate extra file I/O between processor \( i \) and \( j \) iff they satisfy the following equality:

\[
G_i^T v_{ij} = G_j^T v_{ji}
\]

Given the theorem above, the scheduling algorithm for a nested loop that contains only anti-dependences consists of three steps:

- assign a symbolic schedule matrix to each processor. The dimension of the schedule matrix will be equal to the degree of freedom of the tiles.
- compute the schedule equations for every processor pair using \( G_i, G_j, v_{ij} \) and \( v_{ji} \).
- initialize the schedule matrix of a processor with an arbitrary schedule from Figure 4 and compute the corresponding schedules of the remaining processors by solving the schedule equations.

Consider the following elliptic solver example that uses five-point relaxation.

\[
\text{FORALL}(i=2:n-1,j=2:n-1) \\
X(i,j)=X(i,j)+X(i+1,j)+X(i-1,j)+X(i,j+1)+X(i,j-1))/5.0 \\
\text{ENDFORALL}
\]

Suppose \( f=2 \) and \( F = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \) as shown in Figure 5:B.

Let \( \begin{pmatrix} a_i \\ c_i \end{pmatrix} \) denote a symbolic schedule matrix for processor \( i \). After obtaining the necessary equations using the preceding theorem, if we let the scheduling matrix for processor \( 0 \) to be \( \begin{pmatrix} a_0 & b_0 \\ c_0 & d_0 \end{pmatrix} = \)

\(^4\)It should be noted that if a FORALL loop contains multiple statements, there may be flow-dependences among different statements, and they should be counted for.
\[
\begin{pmatrix}
-1 & 0 \\
0 & -1 \\
\end{pmatrix},
\]
then the remaining schedules are \( G_1 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \), \( G_2 = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \), and \( G_3 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \).

As shown in Figure 5:C, these values define a scheduling which does not involve any extra file I/O.

**Theorem.** If a loop contains only anti-dependencies, then using LPM, it is always possible to schedule tiles such that all extra file I/O is eliminated [BCR95, Bor96].

Let us now consider the computations that involve flow-dependences, considering only rectangular tiles. A **Tiling Space Graph** (TSG) is defined to indicate the chain vectors among the tiles. We are dealing with the two-dimensional case where each component of the effective chain vectors between two tiles is 0 or 1. As a first step, the tile access patterns that are associated with scheduling matrices are re-defined as shown in Figure 6. The numbers indicate the order of execution. We discuss a technique called *minimal perturbation*. Suppose that an effective chain vector \( u_i \) in the TSG is a member of set span\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \}. The practical significance of this is that any scheduling in the opposite direction \( \begin{pmatrix} -1 \\ 0 \end{pmatrix} \) is not acceptable. In order to avoid this, after all schedule matrices \( G_j \) are obtained (as in the previous case), if the first column of any of them is \( \begin{pmatrix} -1 \\ 0 \end{pmatrix} \), it is converted to \( \begin{pmatrix} 1 \\ 0 \end{pmatrix} \) and the other column is left as it is. So, in order to reach legitimate schedules we apply minimal changes to schedule matrices, hence the name minimal perturbation.

The scheduling algorithm for a nested loop that may contain flow-dependences is the same as that of a loop that contains only anti-dependencies, except that the initial schedule should be *legal* (observe the effective chain vectors) and maximize (pipeline-) parallelism. As a last step of algorithm, we apply minimal perturbation. Consider the following example that illustrates the technique (see Figure 5:D).

![Figure 5: (A) A 2x2 Processor Grid. (B) Data Tiles with f=2. (C) A scheduling that eliminates extra file I/O. (D) Data Tiles with flow-dependence. (E) A scheduling that eliminates extra file I/O for flow-dependence case. (F) A scheduling that leads to extra file I/O. (G) A scheduling that requires extra storage. (H-I) Scheduling that sequentialize computation.](image)

```plaintext
DO i = 2, n-1
   DO j = 1, n-1
      X(i,j) = (X(i+1,j) + X(i,j+1) + X(i-1,j))/3.0
   ENDDO
ENDDO
```

This nest contains an effective chain vector in \( \begin{pmatrix} 1 \\ 0 \end{pmatrix} \) direction. If we write down the schedule equations as in the previous example and initialize \( G_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \) (to maximize pipeline parallelism), after the fourth step of the algorithm we obtain \( G_1 = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \), \( G_2 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \), and \( G_3 = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \).
The schedules for processor 1 and processor 3 are violating the effective chain vector and are not acceptable. So, we apply minimal perturbation to correct them at the last step of the algorithm and we obtain $G_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $G_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $G_2 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$, and $G_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$. These schedules are shown in Figure 5:E. It is easy to see that there is no extra file I/O involved. Also notice that the locality between processors 0 and 2 (similarly between 1 and 3) is exploited. The reason is that the minimal perturbation preserves the localities originating from anti-dependence relations as much as possible. Scheduling in Figure 5:F, on the other hand, leads to extra file I/O (because of the reference $X(i, j + 1)$) and scheduling in Figure 5:G requires extra storage of size $n/2$ per processor. From the discussion above we can conclude the following, proof of which is omitted due to lack of space, but follows closely the preceding discussion.

**Theorem:** By scheduling data tiles appropriately, it is always possible using LPM to eliminate extra file I/O.

![Diagram](image)

**Figure 6:** Tile Access Patterns and Scheduling Matrices for loop nests containing flow-dependences.

In a nested loop that contains only anti-dependences, it is not important what the initial schedule for a processor is. But in a nest that contains flow-dependences, an initial schedule must be legal (not violate any effective chain vector). In addition to that, among the candidate schedules some of them might be preferable over the others, especially when parallelism is considered. To see this, consider Figure 5:H and Figure 5:I. Although both schedules are perfectly legal, neither of them exploits the parallelism available.

### 6 Conclusion and Future Work

In this paper, we have presented a technique to decompose an out-of-core array stored on disk(s) into partitions called data tiles. We have discussed the extra file I/O problem and proven that it is always possible to find tile schedules so that extra file I/O is eliminated completely in stencil computations. We are currently working on the feasibility of data space tiling approach for loop nests that contain arbitrary computations on out-of-core arrays in multicomputers.

### References


Computing the Local Iteration Set of a block-cyclically distributed reference with affine subscripts

Samuel P. Midkiff*

Abstract

This paper presents two methods for computing the local iteration set (LIS) of a reference to a block-cyclically distributed variable. The subscript expression of the reference is assumed to be an affine expression in one or more loop index variables. The first method presented generates the members of the LIS in a different order than they would appear in the iteration space of the surrounding loop. The second method preserves this order at the cost of increased complexity during code generation and at run-time.

1 Introduction

A block-cyclic distribution results from distributing blocks, or adjacent elements, of an array, across processors or nodes in a round-robin fashion. An example of a block-cyclic distribution is shown in Figure 1. Assume a reference $a(c_1 i_1 + c_0)$, (where $c_1$ and $c_0$ are compile time constants and $i_1$ is a loop index variable) to the block-cyclically distributed array. There are two problems that must be solved to compile the program. First, the set of iterations - i.e. values of $i$ - that access elements of $a$ that reside on a processor must be determined. This problem is the problem of finding the local iteration set or LIS. The second problem is determining where in the local buffer space the element $a(c_1 i_1 + c_0)$ is, given a value of $i_1$. The is the access sequence generation problem. For affine subscripts in one variable, such as the one shown here, a variety of good solutions exist [3, 5, 7, 2, 4]. In [8] strong evidence that the overhead of finding the LIS and generating an access sequence for references of the form $a(c_1 i_1 + c_0)$ need not be high relative to the cost of finding these with a regular block distribution. These findings are buttressed by the results of [6].

*The author can be reached by E-mail at midkiff@watson.ibm.com, by mail at I.B.M. T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY, 10598, or by telephone at (914) 945-3018.
A more difficult problem arises when the subscripts are affine functions in two or more variables – for example if the subscript above was \( a(3i_1 + 4i_2 + i_3 + 6) \), where \( i_1, i_2 \) and \( i_3 \) are loop index variables. To our knowledge there are no solutions known to this problem. In this paper we present two solutions. The first is simpler, but does not necessarily generate members of the LIS in the same order as in the original iteration space of the loop. The second is more complicated, and expensive at run-time, but overcomes this limitation.

The difficulty of compiling programs with block-cyclic distributions came to prominence with the inclusion of the distribution in the HPF language standard. Nevertheless, solutions to the block-cyclic problem will be useful for a variety of languages compiled for machines with physically distributed memory with non-uniform latencies. Even if the address space is logically shared, locality and reduction of inter-processor (or inter-node, in the case of clusters of SMPs) communication will be a necessary optimization, just as in message passing machines with distributed address spaces.
2 Computing the LIS and access sequence

The strategy we use to derive the local iteration set and access sequence for general affine subscripts in loop nests is similar to that of [5]. The LIS is first derived, and from the LIS offsets into the local buffer space (i.e. the access sequence) can be formed. The local iteration set is found by symbolically intersecting, at compile-time, the set of elements owned by a particular processor with the set of elements accessed by a reference. By solving for the values of the index variable that access the members of this set, the LIS can be computed. From this the access sequence can be generated by applying the function that maps array elements into the local buffer to the result of applying the subscript function to each member of the LIS.

2.1 Describing the set of elements residing on a processor

We number the blocks distributed onto a processor from zero to the number of blocks minus one, i.e. $L_b \leq b \leq U_b$, where

\[
    L_b = 0 \\
    U_b = \left\lfloor \frac{U_a - s - 1}{r} \right\rfloor.
\]

Block $b$ on processor $p$ contains $bs$ elements, with the first element being $b \cdot r + s$.

Within the block are $bs$ elements. Let $0 \leq e \leq bs - 1$, then

\[
    \mathcal{P} = \{ e : 0 \leq e \leq bs - 1, 0 \leq b \leq U_b | e = b \cdot r + s + e + 1 \}
\]

describes the set of array elements on some processor. This formula assumes a lower bound of 1 for the array, but can be trivially extended to the case where the lower bound is not 1.

2.2 Describing the set of elements accessed by a reference

The set of elements accessed by a reference is the set of values of the subscript expression takes when evaluated at all of the points of the iteration space of the surrounding loop nest. This can be described by

\[
    \mathcal{R} = \{ e : L_{i_j} \leq i_j \leq U_{i_j}, 1 \leq j \leq d | e = c_1 \cdot i_1 + c_2 \cdot i_2 + \ldots + c_d \cdot i_d + c_o \}
\]

2.3 Intersecting $\mathcal{P}$ and $\mathcal{R}$

To solve for $\mathcal{P} \cap \mathcal{R}$, we symbolically solve the diophantine equation

\[
    c_1 \cdot i_1 + c_2 \cdot i_2 + \ldots + c_d \cdot i_d + c_o = b \cdot r + s + e
\]
The variables in this equation are \( i_j, 1 \leq j \leq d, b \) and \( e \). The other terms are constants on a given processor \( p \). Moving the constant terms to the right hand side yields the equation:

\[
c_1 \cdot i_1 + c_2 \cdot i_2 + \ldots + c_d \cdot i_d - b \cdot r - e = s - c_0 + 1
\]

(1)

Substituting \( r \) and \( s \) for the appropriate terms (see Figure 2) gives:

\[
c_1 \cdot i_1 + c_2 \cdot i_2 + \ldots + c_d \cdot i_d - b \cdot r - e = s - c_0 + 1
\]

To solve the equations the techniques described in [1] are used. These involve constructing the array

\[
\begin{bmatrix}
1 & 0 & \ldots & 0 & 0 & 0 & c_1 \\
0 & 1 & \ldots & 0 & 0 & 0 & c_2 \\
\vdots \\
0 & 0 & \ldots & 1 & 0 & 0 & c_d \\
0 & 0 & \ldots & 0 & 1 & 0 & -r \\
0 & 0 & \ldots & 0 & 0 & 1 & -1
\end{bmatrix}
\]

This matrix is formed by concatenating an identity matrix (the first \( d + 2 \) columns) with the transposed vector containing the coefficients of the equation variables (the last column.)

Following the procedure of [1], by successive subtraction and row interchange operations we reduce the last column to contain a "1" in the first row and zeros in all other rows. The same operations are performed on the first \( d + 2 \) columns. The result of this transformation is the matrix

\[
\begin{bmatrix}
0 & 0 & \ldots & 0 & 0 & -1 & 1 \\
1 & 0 & \ldots & 0 & 0 & c_1 & 0 \\
0 & 1 & \ldots & 1 & 0 & c_2 & 0 \\
\vdots \\
0 & 0 & \ldots & 1 & 0 & c_3 & 0 \\
0 & 0 & \ldots & 0 & 1 & -r & 0
\end{bmatrix}
\]

A parametric solution in the variables \( t_1, t_2, \ldots, t_{n+2} \) can be formed with this matrix and the right hand side of the original equation Equation 1. Using the methods of [1], the value of \( t_1 \) is the solution to

\[
[t_1, t_2, \ldots, t_{d+2}] \times [1, 0, \ldots, 0]^T = s - c_0 + 1
\]

or \( s - c_0 + 1 \). Solving for \( i_1, i_2, \ldots, i_d, b \) and \( e \) produces the equations:

\[
i_1 = t_2
\]

\[
i_2 = t_3
\]

\[
\ldots
\]

\[
i_d = t_{d+1}
\]

\[
b = t_{d+2}
\]

\[
e = c_1 \cdot t_2 + c_2 \cdot t_3 + \ldots + c_d \cdot t_{d+1} - r \cdot t_{d+2} + c_0 - s - 1
\]

(3)
<table>
<thead>
<tr>
<th>Variable</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b, t_{d+2}$</td>
<td>$L_b = 0$</td>
</tr>
<tr>
<td></td>
<td>$U_b = \left[ \frac{V_{a+1} - 1}{r} \right]$</td>
</tr>
<tr>
<td>$i_1, t_2$</td>
<td>$L_{i_1} = \left[ \frac{c_1 - c_2 \cdot t_2 - \cdots - c_4 \cdot t_4 + r \cdot b - c_5 + 1}{c_1} \right]$</td>
</tr>
<tr>
<td></td>
<td>$U_{i_1} = \left[ \frac{c_1 - c_2 \cdot t_2 - \cdots - c_4 \cdot t_4 + r \cdot b - c_5 + 1}{c_1} \right]$</td>
</tr>
<tr>
<td>$i_2, t_3$</td>
<td>$L_{i_2} = \left[ \frac{c_2 - c_3 \cdot t_1 - \cdots - c_4 \cdot t_4 + r \cdot b - c_5 + 1}{c_2} \right]$</td>
</tr>
<tr>
<td></td>
<td>$U_{i_2} = \left[ \frac{c_2 - c_3 \cdot t_1 - \cdots - c_4 \cdot t_4 + r \cdot b - c_5 + 1}{c_2} \right]$</td>
</tr>
<tr>
<td>$i_d, t_{d+1}$</td>
<td>$L_{i_d} = \left[ \frac{c_2 - c_3 \cdot t_d - \cdots - c_4 \cdot t_{d-1} + r \cdot b - c_5 + 1}{c_2} \right]$</td>
</tr>
<tr>
<td></td>
<td>$U_{i_d} = \left[ \frac{c_2 - c_3 \cdot t_d - \cdots - c_4 \cdot t_{d-1} + r \cdot b - c_5 + 1}{c_2} \right]$</td>
</tr>
</tbody>
</table>

Figure 3: The initial bounds equations

### 2.4 Solving for $i_j$

With the parametric equations of Equation 2 we can describe the LIS's of the index variables in the loop nest. The LIS's will be described as a set of inequalities, which can be thought of as defining the iteration space of a loop nest that emits the LIS (or forms the local access sequence.)

In the following set of equations, the bounds on $b$ (or $t_{d+2}$) are the bounds on $b$ computed above. The bounds on each $i_j$ ($t_{j+1}$) are found by solving for $i_j$ in Equation 3. The result of this is shown in Figure 3.

There is one problem with this formulation, however. The bounds for $i_k$ contain references to all other index variables $i_1, i_2, \ldots, i_{k-1}, i_{k+1}, \ldots, i_d$. Thus the bounds of a loop $i_k$ cannot be known without knowing the bounds of all outer and inner loops – whose bounds in turn depend on the value of $i_k$.

We solve this problem as follows. Since the goal is to determine the bounds of a loop nest that enumerates the members of the local iteration set for the
do $b = L_b, U_b$
  do $i_1 = L_{i_1}, U_{i_1}$
    do $i_2 = L_{i_2}, U_{i_2}$
      ... 
    do $i_d = L_{i_d}, U_{i_d}$

(b) Representation of the inequalities as a loop nest

do $b = L_b, U_b$
  
  $L_{i_1} = \max \left( L_{i_1}, \left[ \frac{L_a - c_2 \cdot L_{i_3} - \ldots - c_d \cdot L_{i_d} + r \cdot b - c_0 + s + 1}{c_1} \right] \right)$
  
  $U_{i_1} = \min \left( U_{i_1}, \left[ \frac{U_a - c_2 \cdot L_{i_3} - \ldots - c_d \cdot L_{i_d} + r \cdot b - c_0 + s + 1}{c_1} \right] \right)$

  do $i_1 = L_{i_1}, U_{i_1}$
    
    $L_{i_2} = \max \left( L_{i_2}, \left[ \frac{L_a - c_1 \cdot i_1 - c_3 \cdot i_3 - \ldots - c_d \cdot L_{i_d} + r \cdot b - c_0 + s + 1}{c_2} \right] \right)$
    
    $U_{i_2} = \min \left( U_{i_2}, \left[ \frac{U_a - c_1 \cdot i_1 - c_3 \cdot i_3 - \ldots - c_d \cdot L_{i_d} + r \cdot b - c_0 + s + 1}{c_2} \right] \right)$

  do $i_2 = L_{i_2}, U_{i_2}$
    
    ... 
  
  $L_{i_d} = \max \left( L_{i_d}, \left[ \frac{L_a - c_1 \cdot i_1 - c_2 \cdot i_2 - c_4 \cdot i_4 - \ldots + r \cdot b - c_0 + s + 1}{c_d} \right] \right)$
  
  $U_{i_d} = \min \left( U_{i_d}, \left[ \frac{U_a - c_1 \cdot i_1 - c_2 \cdot i_2 - c_4 \cdot i_4 - \ldots + r \cdot b - c_0 + s + 1}{c_d} \right] \right)$

  do $i_d = L_{i_d}, U_{i_d}$

(c) A representation of the inequalities as a loop nest

---

Figure 4: Final bounds equations for loop index variables

loop, the set of inequalities can be represented as a loop nest, as shown in Figure 4a. Therefore, any references to $i_j, j < k$ that are in the bounds of the index variable $i_k$ can use the value of $i_j$ that currently exists in the loop nest formed by the inequalities. This leaves us with the problem of dealing with the references to $i_m, m > k$ in the bounds expressions for $i_k$. To remove these terms containing references to inner loop index variables, i.e. $i_m, m > k$ from the bounds expressions, we will substitute the extreme values possible for these
real a(2000)
!HPFS$PROCESSORS P(8)
!HPFS$DISTRIBUTE a(cyclic(10)) ONTO P

   do \(i_1 = 1, 100\)
   do \(i_2 = 1, 200\)
   do \(i_3 = 1, 300\)
   \(\ldots a(3 \cdot i_1 + 4 \cdot i_2 + i_3 + 5) \ldots\)
   end do
   end do
end do

(a) A Program with a block-cyclically distributed affine reference

   do \(b = 0, \left[\frac{1999 - 10.2}{80}\right]\)

   do \(i_1 = \text{max}(1, \left[\frac{80 \cdot b - 1005}{3}\right]), \text{min}(100, \left[\frac{80 \cdot b - 1}{3}\right]\)

   do \(i_2 = \text{max}(1, \left[\frac{-3 \cdot i_1 + 80 \cdot 3 - 305}{4}\right]), \text{min}(200, \left[\frac{-3 \cdot i_1 + 80 \cdot 4 + 3}{4}\right])\)

   do \(i_3 = \text{max}(1, -3 \cdot i_1 - 4 \cdot i_2 + 80 \cdot b - 5), \text{min}(300, -3 \cdot i_1 - 4 \cdot i_2 + 80 \cdot b + 4)\)

   (a) The LIS for the reference of (a)

---

Figure 5: An example of the first LIS computation method

terms[1]. The resulting loop nest is shown in Figure 3\(^1\)

Figure 5 gives a short example of how to use this method.

The resulting loop bounds are still deficient in one major respect: first all values of \(i_1, i_2, \ldots, i_d\) that access elements of the 0'th block are generated, then all values of \(i_1, i_2, \ldots, i_d\) that access elements of the 1st block are generated, and so forth until all accessed blocks on the processor are scanned. The order than these members of the iteration space are generated is not necessarily the

\(^1\)We briefly explain the method of [1] for finding the maximum (minimum) values of an affine expression. Let \(c \cdot i\) be a term in the affine expression, where \(c\) is a constant and the upper and lower bounds of \(i\) are \(L_i\) and \(U_i\). The maximum (minimum) extreme value is given by \(c^+ \cdot U_i - c^- \cdot L_i\) \((c^+ \cdot L_i - c^- \cdot U_i)\), where \(c^+ = c\) if \(c \geq 0\), and 0 otherwise and \(c^- = c\) if \(c \leq 0\), and 0 otherwise. The sum of the terms gives the extreme values of the linear expression as a whole. For notational simplicity, in the paper we have assumed all constants are positive, but the information above allows these results to be easily extended to the case where the sign of \(c\) is negative or unknown.
same as the order they would be generated in the original iteration space of the loop. This leads to two problems:

1. It cannot be assumed that dependences between two iterations of the loop nest that execute on the same processor are honored by virtue of sequential execution within a processor.

2. When generating the union or intersection of two local iteration sets it is desirable that common elements of the local iteration sets be generated in the same order on different processors. This is not guaranteed with the current technique.

We will now form new bounds on the local iteration set that remove these restrictions. As is often the case, the computation of the new bounds is probably less efficient than the previous bounds computation.

Our basic strategy will be to compute all members of the iteration set for \( i_1 = 1, i_2 = 1, \ldots, i_{d-1} = 1 \), then all members for \( i_1 = 1, i_2 = 1, \ldots, i_{d-1} = 2 \), and so forth until the entire iteration set is scanned. We consider first the outermost loop, and let \( i_1 \) range from \( L_{i_1} \) to \( U_{i_1} \). Using the equation for \( e \) (Equation 3), and the technique of substituting extreme values used before, we can limit the blocks which might be accessed when \( i_1 \) is a particular value \( v_1 \). The expression for this is:

\[
L_{i_1} = \max \left( L_b, \left[ \frac{-U_b + c_1 \cdot i_1 + c_2 \cdot L_{i_2} + \ldots + c_d \cdot L_{i_d} + c_0 - s - 1}{r} \right] \right) \tag{4}
\]

\[
U_{i_1} = \min \left( U_b, \left[ \frac{-L_b + c_1 \cdot i_1 + c_2 \cdot U_{i_2} + \ldots + c_d \cdot U_{i_d} + c_0 - s - 1}{r} \right] \right)
\]

For the \( i_2 \) loop, the range of possible values that \( i_2 \) can take on for some \( i_1 \) value is found by solving for the upper and lower bounds of \( i_2 \) in Equation 3, again substituting extreme values for unknown variables. The resulting expressions are:

\[
L_{i_2} = \max \left( L_{i_2}, \left[ \frac{L_b - c_1 \cdot i_1 - c_3 \cdot U_{i_3} - \ldots - c_d \cdot U_{i_d} + r \cdot b - c_0 + s + 1}{c_2} \right] \right) \tag{5}
\]

\[
U_{i_2} = \min \left( U_{i_2}, \left[ \frac{U_b - c_1 \cdot i_1 - c_3 \cdot L_{i_3} - \ldots - c_d \cdot L_{i_d} + r \cdot b - c_0 + s + 1}{c_2} \right] \right)
\]

Since the range of possible values for \( i_2 \) is more tightly constrained than in Equation 4, we can now tighten the bounds on \( b \) using the tighter bounds for \( i_2 \) in Equation 5:

\[
L_{i_2}^{i_1} = \max \left( L_{i_2}^{i_1}, \left[ \frac{-U_b + c_1 \cdot i_1 + c_2 \cdot L_{i_2} + \ldots + c_d \cdot L_{i_d} + c_0 - s - 1}{r} \right] \right)
\]

\( ^2 \)This can be made more efficient by finding the iteration space implied by the extreme values of the bounds expressions and intersecting it with the actual loop iteration space. In order to simplify the explanation somewhat we have omitted this optimisation.
Figure 6: An example of the second LIS computation method using the program of 5.

These tighter bounds can be used when finding $L_{i_3}, U_{i_3}$. This process continues until bound have been formed for all $i_1, i_2, \ldots, i_d$. Figure 6 shows the loop nest that results from applying this method to the program of Figure 5a.

3 Conclusions

We have described two methods of computing local iteration sets for block cyclically distributed arrays that are referenced with affine subscripts in with one or more index variables. From a member of the LIS, the offset into the
global space of the array, and from there the access into the local storage buffer can be computed. Therefore, our LIS computation scheme forms the basis of a method to generate access sequences for these references. It also has the benefit of not requiring any diophantine equations to be solved at run-time.

References


The PGI-PSI Project:
Preprocessing Optimizations for Existing and New F90 Intrinsics in HPF using Compositional Symmetric Indexing of the $\Psi$ Calculus. *

Lenore R. Mullin       David Eggleston       Luther J. Woodrum
William Rennie
{lenore,]w,rennie}@cs.albany.edu, burnett@cs.wisc.edu
Department of CS, University at Albany, SUNY
Albany, New York 12222

September 16, 1996

Abstract

Optimizations for Fortran 90 Intrinsic expressions are necessary if HPF is to exploit structure of array arguments in mapping (sub)arrays to processors. This paper reports on a research project, between the $\Psi$ Compiler Group, University at Albany, SUNY and The Portland Group, Inc.(PGI). The project attempts to integrate a preprocessing optimizer, based on the $\Psi$ Calculus, into pg/hpf. The optimizer exploits efficient indexing of arrays and array expressions. It also minimizes temporary arrays caused by multiple intrinsics.

Introduction

This paper reports on the Research Project between the Psi($\Psi$) Compiler Group in the Department of Computer Science, University at Albany, and The Portland Group Inc.(PGI). We report our progress to date with a thorough presentation deferred to a subsequent in revision journal paper.

A goal of this project is to show how the $\Psi$ Calculus [13] can provide formal optimizing transformations and reductions to HPF Fortran 90 intrinsic expressions. We discuss how this approach applies to combining alignments and distributions relative to partitioning and we propose how to incorporate a $\Psi$-Reduction Engine in any HPF compiler.

Efficient indexing of arrays is a prerequisite to both High Performance Computing and Communications(HPCC) with origins in sequential programming [6]. As multiprocessors emerged, array indexing was further complicated by communications slow downs incurred by distributed memory and the complicated memory hierarchy that ensued. For scientific applications symmetries in access patterns were exploited by many researchers [10, 4, 1, 3, 5, 9, 14]. But, unless programs were highly structured, concurrency was difficult if not impossible [16] to extract. The focus on a Fortran 90[7] style of programming hoped that structure, implicit in the data, would help exploit the diversity of architectures.

---

*This work was supported by NSF Presidential Faculty Fellow Award CCR-9596184. Eggleston is presently in the Dept. of CS, University of Wisconsin-Madison.
Various algebraic methods use compositional matrices to produce both unimodular and non-unimodular transformations on index sets \([2, 17, 15, 8]\) to exploit symmetries. Loop transformations correspond to loop interchange and permutation, while multiplying a loop by \(-1\) corresponds to loop reversal \([15]\). The \(\Psi\) method is similar and more general since operations and their resultant shapes are defined in terms of argument shapes. The MOA[13] Algebra and proposed extensions to F95 \([11]\) are fundamental to the definition of F90 intrinsics and indexing in general.

The Psi Project

Our initial efforts focused on mechanizing a useful subset of the MOA algebra[13] and its corresponding reduction rules. We viewed our interface to the reduction engine as a preprocessing intermediate form that took valid expressions as input and produced simpler reduced expressions, in terms of indexing followed by code generation to C, as output. Next, we added a HPF front end for an extended subset of F90 intrinsics[11] with output to F90. Our experiments produced optimized results for a small program, similar to the following example. Next, we needed to interface a production compiler and run an experiment using HPF benchmark programs. This phase is the topic of our paper. Our academic research project moved into an \(R\) and \(D\) effort with PGI. We chose to interface PGI’s HPF at the Abstract Syntax Tree(AST) level. We wanted a transparent universal interface that proved useful in other compiler efforts\(^1\). We did not however use existing interfaces because we wanted ASTs to denote F90 monolithic arrays, i.e. arrays as leaves of nodes. We did not want to rely on F90 to F77 to C translators in order to use the SUIF interface, e.g.. An AST link let \(\Psi\) optimizations proceed prior to PGI’s. We chose a program from the NPAC HPF benchmark suite. We then modified these programs to exploit F90 intrinsics and our extensions.

Psi Reduction: Motivating Optimizations

Definitions are given for one dimensional arrays\(^2\) Let \(A\) and \(B\) denote one dimensional vectors such that SHAPE(\(A(4)\)) and SHAPE(\(B(4)\)) is \((4)\). \(A\) is the vector \((0,1,2,3)\) and \(B\) is the vector \((/4,5,6,7/)\).

Definition 1 CONCAT(\(A,B\)) is defined when

\[
\text{SHAPE(CONCAT}(A,B)) = \text{SHAPE}(A) + \text{SHAPE}(B)
\]

and for all valid indices \(i\) such that

\[
1 \leq i \leq (\text{SHAPE}(A) + \text{SHAPE}(B))
\]

\(\text{CONCAT}(A,B)[i] = A[i] \text{ when } 1 \leq i \leq \text{SHAPE}(A) \text{ and } B[i-\text{SHAPE}(A)] \text{ otherwise.}
\]

Definition 2 CSHIFT(\(A,SHIFT=x\)) is defined when \(x\) is an integer and

\[
\text{SHAPE(CSHIFT}(A,\text{SHIFT}=x)) = \text{SHAPE}(A)
\]

and for all valid indices \(i\) such that

\[
1 \leq i \leq \text{SHAPE(CSHIFT}(A,\text{SHIFT}=x))
\]

\(\text{CSHIFT}(A,\text{SHIFT}=x)[i] = A[i+(x+i-1) \mod \text{SHAPE}(A)]\).

\(^1\)http://sulf.stanford.edu

\(^2\)To illustrate compatibility, we use Fortran’s syntax.
An Example

With the definitions above let us simplify:

\[ \text{CONCAT}(\text{CSHIFT}(A, \text{SHIFT}=3), B)(5:8) \]

First, obtain the shape of this expression.

\[ \text{SHAPE}(\text{CONCAT}(\text{CSHIFT}(A, \text{SHIFT}=3), B))(5:8) \text{ is } (4) \]

Therefore, there exists valid indices \( i \) such that \( 1 \leq i < 5 \).

Next, we apply \( \Psi \) Reduction:

\[ \text{CONCAT}(\text{CSHIFT}(A, \text{SHIFT}=3), B)(6:8)[i] \text{ is } \text{CONCAT}(\text{CSHIFT}(A, \text{SHIFT}=3), B)[i+4] \]

Let \( j = 4 + i \). Therefore for \( 5 \leq j < 9 \)

\[ \text{CONCAT}(\text{CSHIFT}(A, \text{SHIFT}=3), B)[j] \text{ is } B[(i+4) - \text{SHAPE}(\text{CSHIFT}(A, \text{SHIFT}=3))] \]

which simplifies to

\[ B[(i+4) - 4] \text{ then } B[i] \]

\[ \text{\textit{qed.}} \]

Notice, that \( j \) can never be less than 5. Therefore, we see from the reduction process that

\[ \text{CONCAT}(\text{CSHIFT}(A, \text{SHIFT}=3), B)(5:8) \text{ is equivalent to } B(1:4). \]

Overview of the two Compilers

Organization of the PGIPSi Project

This project consists of various segments. The first segment traverses abstract syntax trees (ASTs) produced by \textit{pgphf} and passes relevant information to functions provided by the second segment, which creates equivalent abstract syntax trees that the \( \Psi \) compiler can recognize. The third segment traverses a data structure of \( \Psi \) \textit{statements} reducing as needed. The output is passed to functions provided by the fourth segment, which creates equivalent abstract syntax trees that are inserted into the program being processed by the first segment. As we will see, the simplicity or complexity of the data structures between the compilers determines the complexity of the interface.

The introduction of a true interface would cause this program to be broken up into five logical segments. The first segment would convert ASTs of the target compiler into ASTs of this interface. The second segment would convert ASTs of the interface into ASTs of the \( \Psi \) compiler. The third segment would convert reduced ASTs of the \( \Psi \) compiler to ASTs of the interface. The fourth segment would convert ASTs of the interface to ASTs of the target compiler. Finally, a fifth segment would consist of several modules needed to support the interface, including functions to create ASTs and allow symbol table implementation.

PGI Compiler and PGI Representation

One goal of our project was to demonstrate the \( \Psi \) compiler's effectiveness in simplifying array expressions used in traditional benchmarks. We obtained access to the source code for \textit{pgphf}, the High Performance Fortran (HPF) compiler produced by PGI.

We inserted our top level function \textit{PGIPSi} \textit{AST}, in the \textit{pgphf} source code immediately after an HPF program had been parsed and put into AST form, but before any \textit{pgphf} optimizations. This is because the output of the fourth segment could potentially create several loops that could be merged by \textit{pgphf}'s optimizer.
The first segment traverses a PGI statement list and removes assignment statements from this list. It then verifies that this statement is one that can be converted to an equivalent $\Psi$ expression that the $\Psi$ compiler's reduction engine can handle. PGI statements are not nested. For example, a node for a DO statement does not contain a sublist corresponding to the body of the loop. Rather, it just contains the loop index information. The statement list will contain an ENDDO statement after the loop body. This allowed us to process any assignment statement in the program without having to encode this code segment using recursion, particularly in the form of statement list stacks.

In the future, one desirable feature of the interface would be to analyze this statement list and look for reductions that could be applied to several lines. For example, consider the following code segment:

\[
\text{TEMP1} = \text{CHT}\text{HIFT}(B, \text{shift} = 5) \\
\text{TEMP2} = \text{CHT}\text{HIFT}(C, \text{shift} = 3) \\
\text{A} = \text{TEMP1} + \text{TEMP2}
\]

As is, it doesn't eliminate these temporary arrays. If it could inspect the statement list and verify that the specified values of TEMP1 and TEMP2 weren't used as R-values elsewhere in the program, it could eliminate these three statements and then replace this segment with the reduced code for:

\[
\text{A} = \text{CHT}\text{HIFT}(B, \text{shift} = 5) + \text{CHT}\text{HIFT}(C, \text{shift} = 3)
\]

This would allow programmers to not have to bunch up expressions into a single statement. For now, we have had to do this with the benchmarks. In summary, the function that processes single PGI assignment statements (PGI\text{CONV.AST}) has one PGI statement as its input, and produces a PGI statement list as its output.

In this representation, the nodes of the statement list exist primarily to implement a doubly linked list. Each node of a list points to an AST node. Thus, a statement is treated like an expression. Each AST node consists of $n$ 32-bit words. Information is then packed into this node via C macros. To accommodate a wide variety of AST types, a particular piece of the node represents different fields based on the AST's type.

The symbol table is represented in a similar fashion, with each symbol table entry consisting of $n+1$ 32-bit words. Symbol table routines are also implemented with C macros. There are also similar, but distinct, structures for function argument lists, triple lists (e.g., for forall statements), and array subscript information. Although the use of well-named macros made programming for this compiler very easy, with the data structure information well hidden, the homogeneity of the representation made tracing errors a tedious process, as they compiled.

Psi Compiler and Psi Representation

As an academic research project, students programmers were used. The core of the compiler was written without a front end per se. However, a parser was written to handle a subset of the MOA algebra. The core does incorporate a back end, however, and it is this feature that has been the primary impediment to our progress. The code generation is in C, and the programmers of the module that implements the reduction rules of the Psi Calculus clearly had this end in mind.

The input to the reduction engine of the $\Psi$ compiler is an assignment statement. The output is a statement list that corresponds to the procedural steps that would have to be taken in order to complete the assignment. The $\Psi$ Research Group is currently discussing whether and how the reduction engine should be rewritten. One of the primary goals is to keep expressions as expressions,
much as the $\Psi$ Calculus does. In this approach, the output of reducing an expression would be the equivalent simplified expression in normal form. This would have the advantages of:

- aiding in proving the correctness of the reduction engine;
- easing the addition of reduction rules to the engine;
- allowing for a uniform processing of the output.

The $\Psi$ compiler utilizes numerous data structures. The $\text{expr\_t}$ type is used to represent array expressions. The $\text{ident\_t}$ type is generally used to represent a symbol table entry. It contains fields for the name, type, and value of an identifier. The $\text{vect\_t}$ type allows one to construct vector expressions without using the bulky expr.t structure. The $\text{s\_expr\_t}$ type allows one to build scalar expressions. This too allows one to avoid using the expr.t type. The existence of these types may have made it easier for the original programmers to create ASTs, but it makes traversing the results much more difficult. Finally, the $\text{statement\_t}$ type is used as an intermediate representation for code generation. That is, the reduction engine not only applies rules to the incoming expression, but also translates that abstract syntax tree into another language. This type is used to implement a linked list, with each node containing a pointer to a node of some type depending on the type of statement. Usually, the type of a statement is $\text{reduced\_t}$. This field will often have a subfield (of type $\text{assign\_t}$) which represents an assignment similar to the one that was passed to the engine. However, more than one assignment may be returned. Additionally, reduced.t’s can be used to implement forall loops. Inside this node there is a list of reduced.t’s which represent the body of the loop.

PSLPGI / PGLPSI stacks

To allow communication between the two compilers, we implemented a pair of stacks which are manipulated by driver routines. Most parameter passing occurs at the atomic level (constants and identifiers.) Larger expressions obtain the data for subexpressions/operands/arguments from the stack. Occasionally, it was more efficient (and conceptually clearer) to pass constants over. One such context was the dimension for array operations. However, as the $\Psi$ compiler expands and becomes more comprehensive, this will have to be changed. For example, the shift amount of a circular-shift (cshift) should be any scalar expression. While the design of $\Psi$ data structures allow for this expression, the compiler rejects this usage.

The PGLPSI stack nodes are of type $\text{parser\_t}$, which is essentially an ident.t and an expr.t joined together. The PSLPGI stack nodes are of type UINT16 as defined in $\text{pghpf}$. This type is used to represent all PGI objects.

Processing ASTs

We first needed to traverse a PGI representation of a HPF program then pass relevant information to the PGLPSI stack. At the top most level, each node of a linked list of statements is visited. If the node represents an assignment statement, the LHS is visited, followed by the RHS. We then request the PGLPSI stack to form an assignment statement and pass it to the reduction engine. To enforce modularity, a pointer is passed to the current statement node to the PGLPSI modules. These modules will, in turn, pass them back to the PSLPGI modules when requesting to add a new statement to the PGI list. The PGLPSI functions are straightforward. PGI expressions are searched depth-first, so combinators such as binary operators or function calls are handled by first visiting the sub-expressions (and pushing the translations onto the PGLPSI stack), and
then requesting the PGLPSI stack to rebuild the expression. Unlike statements, expressions are traversed right-to-left as a rule. For procedure/function calls, the last argument is processed first. Generally, the PGLPSI routines are ignorant of Ψ notation. For example, when a CSHIFT node is encountered, a request is made to PGLPSICSHIFT; the PGLPSI module will pop the argument off the stack and form an equivalent Ψ expression. In MOA, all arrays have an index origin of 0. In HPF, arrays may have any starting index. So, for each dimension of the array, the number of components (stop - start + 1) is computed then stored in an array. The array is passed to the PGLPSI stack, which treats it as the shape vector for the array.

Presently, the reduction engine of the Ψ compiler can only process a subset of legitimate expressions. Therefore, as the PGI ASTs are traversed, we needed to verify that the expression is Ψ-ready before passing the assignment to psi.reduce. Instead of making a first pass to inspect the expression, we make a single pass, assuming that it will be valid. If we found at some point that the current statement should be skipped, the traversal was aborted immediately and the PGLPSI stack emptied.

The next segment was a set of routines that remove information from the PGLPSI stack and form Ψ expression trees. Before the first statement is converted, a symbol table is created and initialized; this table is used by all statements throughout an entire program.

When one is forming Ψ expressions, there are really two approaches to take. The first is to use basic Ψ operators to try to build the result. The second approach is to indicate how indexing this expression relates to indexing the operands. For example, to build a 2-D TRANSPOSE of an M by N array A, one would have to build an expression using M*(N-1) + (M-1) omega (concat) operators[13]. That is, we isolate the elements within the row and then use omega to concatenate them into a column, which would require N-1 concats for each of the M rows. Then, we would join the M columns together to form an N by M transpose of A. More efficient is to express indexing the TRANSPOSE in terms of indexing A. One can build two forall nodes (both with start < 0 >, one with bound < M >, the other with bound < N >), say with forall indices forall1 and forall2. Then the bottommost node of this expression would be an array access (of A) with index vector < forall2, forall1 >. The AST is much smaller than in the previous approach, and there is virtually nothing left to reduce. We first tried to build the array section by forming an index set. We then multiplied the index set by the stride of the section, adding in the start of the section, and then using omega to index A with the index set. Unfortunately, the Ψ compiler produced complex code for this type of expression. Due to a traditional approach taken by the compiler programmers, an omega node was reduced and immediately sent to code generation. This strategy had the side effect of not knowing that an omega node could be part of an expression that needed further reduction. We then realized it would be much easier (and formally equivalent) if we expressed subscripting not just by using a new forall node, but by actually changing the start/stop/stride information in the expression to be subscripted. The first step was to compute the new shape of the node; the number of components for each dimension i would be (stop[i] - start[i]) / stride[i] + 1. The second step is to compute the new indexing rule for this node; if the node has no prior indexing rule, then it is initialized to (start - 1). If the node does have a prior indexing rule, then a new start must be computed. Recall, that if the start of the section is δ, that refers to the fifth element of the expression. So, the adjusted start is the original start + (section start - 1) * original stride. The stride is set; if there was a stride originally specified, then the new stride is the original stride multiplied by the section stride.

**Example**: Let fish be a vector with shape < 50 >, and goat be a vector with shape < 4 >. The statement (fish(10:40:3))(3:9:2) would be evaluated as follows:
1. The process is bottom-up, so first the array access node is created with un-set index, bound, and stride fields.

2. Next, the section (10:40:3) is processed.

3. The array access node’s index field is set to \( <9 > (10-1) \).

4. The array access node’s bound field is set to \( <11 > ((40-10)/3 + 1) \).

5. The array access node’s stride field is set to \( <3 > \).

6. Next, the section (3:9:2) is processed.

7. The array access node’s index field is set to \( <15 > (9 + (3-1)*3) \).

8. The array access node’s bound field is set to \( <4 > ((9-3)/2 + 1) \).

9. The array access node’s stride field is set to \( <6 > (3 * 2) \).

So, by adjusting the fields, one can more easily create the desired index set; in this case, 15, 21, 27, 33.

When reduction was complete, inverse functions were applied to the PSI ASTs to produce PGI ASTs. They were then re-linked to pgphpf’s ASTs to allow further processing through their compiler.

**Experiments**

**Hardware**

For our experiments we used three machines with four processors connected by a 155 Mb/sec FORE ATM: one Sun 1000 with 128 MB of Main Memory, 506 MB of Virtual memory, 2-60 Mhz SPARC CPUs and, two SUN SPARC 20s with 64 MB of Main Memory, 535 MB of Virtual.

**Software**

We used the ADI (Alternating Direction Implicit method) HPF benchmark program developed at NPAC\(^3\). ADI uses a split operator technique to solve the Poisson and vorticity transport equations that arise from the unsteady Navier Stokes equations for incompressible flow. These equations can be expressed as Poisson equations in spatial terms with the time dependence separated. The computation of the ADI scheme requires the solution of 2 matrix equations for 1 intermediate finite difference step. A code fragment taken from the programs defining ADI follows. Observe the modifications we made to the code which exploit intrinsics and our new grammar. We made changes consistent with these throughout the HPF programs.

```fortran
    DO k=j1+1,jN
      *************
      ! Code for extended compiler
      !
      ! temp(k,j1:jN) = (c/beta)(k-1,j1:jN)
      !
      ! beta(k,j1:jN) = (b - a*temp(k,j1:jN))
      !
      ! u(k,j1:jN) = ((c*shift(u,shift=1,dim=1))/beta)(k,j1:jN)
      ! END DO
      !
      ! temp(N,j1:jN) = (c/beta)(jN,j1:jN)
    ! *************

\(^3\)Northeast Parallel Architectures Center
temp(k,ji:jN) = c/beta(k-1,ji:jN)
beta(k,ji:jN) = b - a*temp(k,ji:jN)
u(k,ji:jN) = (rhs(k,ji:jN) - a*u(k-1,ji:jN))/beta(k,ji:jN)
END DO

Performance

Our first experiment used a CSHIFTed expression similar to our Phase II experiment[11]. Full translation through both compilers succeeded. Input was:

! cshift.pfh
program csh
  implicit none
  real :: fish(30), a5(10)
  a5 = (cshift(cshift(fish, 15), 10))(11:20:1)
end program csh

output from pghpf produced:

program csh
  implicit none
  common /phgpf_0c/ phgpf_0c, /phgpf_1c/ phgpf_1c, /phgpf_0c/ phgpf_0c
  character*1 phgpf_0c
  integer i, j, k, a1(20), b5(30), fish(30), a4(30), a5(10)
  allocatable fishsr(:)
  pointer (fishsr$dp,fishsr$d), (fishsr$dp,fishsr$d)
  data phgpf_01 /841724289,377598286,933323256,406440568,1,
+1261038733,0,-278833561/
  call phgpf_init(0)
  call phgpf_template(fishsr$dp,1,0,1,10)
  call phgpf_instance(fishsr$dp,fishsr$d,27,4,0,0,0)
  allocate(fishsr(1:30))
  ! forall (i=1:15:1) fishsr(i) = fish(i+15)
  do ii = 1, 15
    fishsr(ii) = fish(ii+15)
  enddo
  ! forall (i=16:30:1) fishsr(i) = fish(i-15)
  do ii = 16, 30
    fishsr(ii) = fish(ii-15)
  enddo
  ! forall (ii=1:20:1) b5(ii) = fishsr(ii+10)
  do ii = 1, 20
    b5(ii) = fishsr(ii+10)
  enddo
  ! forall (ii=21:30:1) b5(ii) = fishsr(ii-20)
  do ii = 21, 30
    b5(ii) = fishsr(ii-20)
  enddo
  call pghpf_free(2,fishsr$dp,fishsr$d)
  deallocate(fishsr)
  ! forall (ii=1:10:1) a5(ii) = b5(ii+10)
  do ii = 1, 10
    a5(ii) = b5(ii+10)
  enddo
99999 continue
end

Output from the PGIPS1 compiler produced:

```
program cah
implicit none
common /pghsf_0/pghsf_0, /pghsf_1/pghsf_1, /pghsf_0c/pghsf_0c
+, /pghsf.lineno/pghsf.lineno, /pghsf_01/pghsf_01
integer pghsf_0, pghsf_1, pghsf.lineno, pghsf_01(0)
character*1 pghsf_0c
integer i, j
real j, k, a1(20), b5(30), fish(30), a4(30), a5(10), z,a_0(30)
data pghsf_01 /94172310,23932152,9333256,37958208,1,
+126168733,0,-398445550/
call pghsf_init(0)
!
forall (i=1:10:1) a5(i) = fish(i)+5
do i=1, 10
a5(i) = fish(i)+5
enddo
99999 continue
end
```

Observe the \(\Psi\) reduced program has one loop. We viewed this as a great success. We made a full integration that successfully compiled in a production compiler. Unfortunately, when we ran the benchmark programs we discovered that the MOA subset supported by the \(\Psi\) compiler was insufficient in its support of LHS indexing. We thought it would be straightforward to make modifications but the complexities of data structures made efforts difficult. Other anomalies in the \(\Psi\) compiler were uncovered during our testing. Although theoretically, scalars and vectors are arrays, the implementation treated them differently. For example, there were separate reduction rules for scalars, vectors and arrays instead of one set of rules for all types. Despite numerous setbacks we successfully compiled ADI. At the time of paper submission we successfully ran our benchmark program on a Sun SPARC 20. Results of experiments running on our four processor network will be reported at the workshop. However, preliminary experiments indicated performance speedups. Beginning with a 32 by 32 matrix and increasing to 64 by 64, \(\Psi\) optimizations gave a 40% performance improvement with implications that further performance would occur as the matrix size got larger. We anticipate increased performance over \textit{pghsf} when communications costs are factored into our timings.

Conclusion

Our attempts to integrate the \(\Psi\) compiler in a production compiler were realized. Our research demonstrates feasibility of potential optimizations for HPF when a preprocessing \(\Psi\) reduction engine is used. We have also demonstrated the need to keep the theory and implementation operationally not just semantically equivalent: the numerous data structures comprising an existing \(\Psi\) AST must be integrated into one and reduction rules must be uniformly applied. e.g., scalars must be treated as zero-dimensional arrays. Finally, we need to investigate the attributes of a general AST interface to facilitate the integration of a \(\Psi\) reduction engine for HPF in general. If the interface were very general it could interface to any monolithic array based language, procedural or functional.

We believe that \(\Psi\) reduction can be applied to optimizations for (re)distribution, (re)alignment, and processor directives. If the partitioning and mapping were incorporated directly into a HPF expression by the compiler, that expression could be reduced by the reduction engine as above, potentially reducing communications traffic.
Figure 1: Initial ADI Experiments: pphpf with PGIPS1

References


Compile-Time and Run-Time Support
Near Fine Grain Parallel Processing without Explicit Synchronization on a Multiprocessor System

Wataru Ogata, Akimasa Yoshida, Masami Okamoto, Keiji Kimura and Hironori Kasahara

Department of Electrical, Electronics and Computer Engineering, Waseda Univ.
3-4-1 Okubo, Shinjuku-ku, Tokyo 169, JAPAN
{ogata,yoshida,okamoto,kimura,kasahara}@oscar.elec.waseda.ac.jp

Abstract - In Fortran parallelizing compilers for multiprocessor systems, loop parallelizing scheme has been widely used. However, there still exist loops to which the Do-all and Do-across techniques cannot be effectively applied because of loop carried dependence and conditional branches to the outside of a loop. Also, the compiler do not exploit coarse grain parallelism among subroutines, loops and basic blocks, and near-fine-grain parallelism inside a basic blocks outside loops or in a sequential loop. Therefore, it is important to use coarse-grain parallelism and near-fine-grain parallelism in addition to loop parallelization. Taking into consideration the above facts, the authors have proposed multigrain parallel processing scheme which combines coarse-grain parallel processing (or macro-dataflow processing), loop concurrentization, and near-fine-grain parallel processing hierarchically. Generally, near-fine-grain parallel processing on a multiprocessor system causes relatively large data transfer and synchronization overhead. To minimize the overhead and total processing time, the proposed compilation scheme uses static scheduling algorithm name of CP/DT/MISP (Critical Path/Data Transfer/Most Immediate Successors First) and a very precise code generation scheme which eliminates all synchronization instructions by scheduling data transfer and memory access timing. This scheme has been implemented on a multiprocessor system OSCAR, which was designed to support precise code scheduling at compile time, and evaluated on OSCAR. The performance evaluation shows the proposed near-fine-grain parallel processing without explicit synchronization reduces processing time by 11% to 40% compared with ordinary near-fine-grain parallel processing with synchronization instructions. The proposed compilation technique will be very effective coming multiprocessors on chip.

I. INTRODUCTION

Conventional Fortran parallelizing compilers for multiprocessor systems have been using loop parallelization schemes [1, 2, 3, 4, 6]. However, those compilers can not exploit coarse parallelism among loops and subroutines and parallelism inside a basic block [6] outside a loop or in sequential loop.

To cope with this problems, the authors have proposed the multigrain parallel processing scheme [7, 8] which hierarchically exploits coarse-grain parallelism, near-fine-grain parallelism, and loop parallelism. Coarse-grain parallel processing, or macro-dataflow [9, 10, 11], exploits parallelism among loops, subroutines and basic blocks. Near-fine-grain parallel processing [8, 12, 13, 14] exploits parallelism among statements, or several machine instructions, inside a basic block among processors. However, near-fine-grain parallel processing on a multiprocessor system suffer from some difficulties such relatively large data transfer overhead and synchronization overhead.

To reduce synchronization overhead, Beckmann and Polychronopoulos proposed a fast barrier synchronization hardware [15]. O’Keefe and Diets proposed “Static Barrier MIMD (SBM)” scheme [16] which combines simple synchronization hardware and scheduling algorithm, and more flexible “Dynamic barrier MIMD (DBM)” scheme [17].

To reduce a number of synchronizations, Zaatari, Dietz and O’Keefe proposed a code generation scheme [18] for their synchronization hardware. The scheme decomposes a basic block into several sections and eliminates synchronization instructions within each section by inserting barrier synchronization instructions between sections.

To reduce data transfer overhead and minimize total execution time the authors have proposed a practical heuristic scheduling algorithm, CP/DT/MISP (Critical Path/Data Transfer/Most Immediate
Successors First [8, 12, 13], which schedule a large number of near-fine-grain tasks onto processors to minimize execution time considering data transfer and synchronization overhead can be minimum. Also, the authors have proposed an elimination scheme of redundant synchronization using the result of static scheduling [12, 19].

This paper proposes a scheme which eliminates all synchronization codes inside a basic block by using precise code scheduling with architectural supports which allow the compiler to predict timing of instruction execution and access to remote distribution shared memory through buses accurately.

The proposed scheme has been implemented on a multiprocessor system OSCAR (Optimally Scheduled Advanced Multiprocessor) having centralized and distributed shared memory addition to local memory, which has designed to support compiler optimization. Also the performance is evaluated of this paper.

Section II of this paper describes OSCAR's architecture, which supports near-fine-grain parallel processing without explicit synchronization. Section III explains OSCAR Fortran parallelizing compiler on which proposed scheme was implemented. Section IV proposes the near-fine-grain parallel processing scheme without explicit synchronization. Section V evaluates performance of the proposed scheme on OSCAR.

II. ARCHITECTURAL SUPPORT FOR NEAR-FINE-GRAIN PARALLEL PROCESSING WITHOUT EXPLICIT SYNCHRONIZATION

This section describes architecture of a multiprocessor system OSCAR (Optimally Scheduled Advanced Multiprocessor) [20] designed by the authors in 1986 to support compilation scheme for multigrain parallel processing including near-fine-grain parallel processing without explicit synchronization. OSCAR has been operational since 1987.

Figure 1 shows OSCAR's architecture. As shown in the figure, OSCAR is a multiprocessor system with both centralized and distributed shared memories in addition to local memories. OSCAR has 16
processor elements (PEs) having distributed shared memory (DSM) and local program and data memories. The PEs are uniformly connected to three modules of centralized shared memory (CSM) by three buses.

OSCAR’s PE has a custom made 32-bit RISC processor of 5 MFLOPS with 64 registers, an integer processing unit, floating-point processing unit, a data memory, two banks of program memories, a dual port memory used as a distributed shared memory, a stack memory (SM), and a DMA controller used for overlapping of data transfer among CSMS and DSMs by data pre-loading and post-storing. Also DMA can be used for pre-loading program code from CSMS to a bank of local program memory. The distributed shared memory on each PE is a dual port memory and can be accessed by the PE itself and another PE simultaneously.

OSCAR provides the following three types of data transfer modes by using DSMs and CSMS:

1. One-PE-to-one-PE direct data transfer using DSM
2. One-PE-to-all-PEs data broadcasting using DSMs
3. One-PE-to-several-PEs indirect data transfers through CSMS

Each module of the centralized shared memory (CSM) is a simultaneously readable memory of which the same address or different addresses can be read by three PEs in the same clock.

In order to predict and control the timing of all behavior of system precisely on clock level at compile time, OSCAR provides the following features. The custom-made 32bit RISC processor core on each PE executes every instruction including a single precision floating-point addition and a multiplication in one clock and the other floating operations in fixed numbers of clocks. All PEs and buses operate under a single common reference clock. By using those features, compiler can fully control timings of instruction execution, bus access, remote and local memory access.

III. PARALLELIZING COMPILER FOR NEAR FINE GRAIN PARALLEL PROCESSING WITHOUT EXPLOIT SYNCHRONIZATION

This section describes the proposed compilation scheme for near fine grain parallel processing of a Fortran program without explicit synchronization on a multiprocessor system. The proposed scheme has been implemented in a middle pass and a back end of OSCAR Fortran Multigrain Compiler [7] to
(a) Near fine grain tasks

(b) Task graph

Figure 3: An example of basic block and task graph

process a basic block assigned to a processor cluster by a dynamic scheduler generated by macrodataflow compiler module [7, 9, 10, 11, 21, 22].

A. Near-Fine-Grain Parallel Processing

Figure 3(a) shows an example of basic block to solve a sparse linear equation using loop-free code generation technic. The near-fine-grain parallel processing scheme [12, 23] decomposes it into near-fine-grain tasks, each of which consists of a statement, analyzes data dependencies among tasks and generates task graph as shown in Figure 3(b). Those tasks are assigned to processors by using CP/DT/MISF scheduling [8, 12, 13, 24] at compile time because uncertainties caused by condition branches are handled by dynamic scheduling in upper level macrodataflow processing.

Next, a compiler back end generates codes for near-fine-grain tasks assigned to each PE and inserts instructions for data transfer, data synchronizations and barrier synchronizations. On OSCAR, a synchronization flag for data synchronization is written on DSM of a receive PE by a sender PE and the flag is checked by the receiver PE without external bus access before execution of consumer task [12, 19, 23, 24].

It takes 4 clocks to write a synchronization flag to remote DSM, and it takes at least 3 clocks to check the flag by a receiver PE. Therefore, it takes at least 7 clocks for data synchronization among tasks on different PEs.
Figure 4: Elimination of redundant synchronization instructions and all synchronization instructions.

B. Elimination of Redundant Synchronization

Synchronization overhead on OSCAR, or 7 clocks, is larger than the cost of the execution of each instruction, or 1 clock. Therefore, it is important for efficient near-fine-grain parallel processing to reduce the number of synchronizations.

The authors have already proposed the elimination of redundant synchronization scheme [12, 19] using result of static task scheduling. Figure 4(a) shows an example of result of static task scheduling when four tasks assigned to three processors (task 1 is assigned to PE1, task 2 and 3 to PE0, task 4 to PE2) and data transfers and synchronizations are performed via CSM. In this case, five data transfers and synchronizations shown as thick edges in Figure 4(a) are required. The data from task 2 to task 3 on the same PE is transferred through a register and the synchronization between them is unnecessary. Synchronization between task 1 on PE1 and task 3 on PE0 is removed as a redundant synchronization because task 2 has already checked the completion of data transfer and synchronization flags set on CSM by task 1. The synchronization between task 1 on PE1 and task 4 on PE2 is also unnecessary because task 4 checks the completion of task 3 which is a successor of task 2.

Finally, the number of synchronizations can be reduced to two as shown in Figure 4(b).
C. Elimination of All Synchronization

Furthermore, the compiler can control the timing to start execution (including data load from CSM instructions) of task 2 by inserting NOPs (No Operation instructions) so that task 2 starts execution after task 1 finishes execution, synchronization instructions between task 1 and 3 can be eliminated. Since in OSCAR the compiler can guarantee all data dependencies by controlling all execution timing, all synchronization instructions can be removed as shown in Figure 4(c).

IV. GENERATION OF NEAR-FINE-GRANULE PARALLEL PROCESSING CODES WITHOUT EXPLICIT SYNCHRONIZATION

A code generator in the back end of OSCAR compiler determines the timings of instruction execution and data transfers considering behaviour of hardware.

A. Elimination of Data Synchronization Instructions

Figure 5 shows an example of data transfer and synchronization between task 1 on PE0 and task 4 on PE1. Here, task 4 is data dependent on task 1. In this case, generally, instructions for synchronization flag set on DSM of PE1 by PE0 and flag check by PE1, are inserted after task 1 and before task 4 as shown in Figure 5(a) and 5(c) depending on when task 2 finishes execution.

The code generator of the proposed scheme also inserts a data transfer instruction after task 1 and then estimates completion time of the data transfer as $t_{set-flag(1)}$. On the other hand, completion time of task 2 which is executed just before task 4 is estimated as $t_{check-flag(1)}$.

If

$$t_{set-flag(1)} \leq t_{check-flag(1)},$$

the code generator removes synchronization instruction for set flag of task 1 and instructions for check it before task 4 as shown in Figure 5(b).

If

$$t_{set-flag(1)} > t_{check-flag(1)},$$

the code generator inserts NOPs (no operation instructions) after task 3 to delay the execution of task 4 by $t_{set-flag(1)}$ as shown in Figure 5(d).
(1) In a case task 2 completes after "data write onto DSM on PE1" by task 1

(2) In a case task 2 completes before "data write onto DSM on PE1" by task 1

Figure 5: Elimination of data synchronization instructions
Figure 6: Elimination of barrier-synchronization instructions

B. Elimination of Barrier Synchronization

When tasks 1, 2, 3, and 4 assigned to four processors try to take barrier synchronization by OSCAR’s hardware, it takes 8 clocks after the last task finish the execute as shown in Figure 6.

In this case, the code generator estimates which task will complete execution last in the four tasks and estimates the completion time as $t_{\text{max}}$.

Next, the code generator inserts NOPs (No Operation instructions) after other tasks (namely task 1, 2, and 4) to make all processors reach the synchronization point at $t_{\text{max}}$ together as shown in Figure 6(b). Application of this scheme eliminates 8 clocks of hardware barrier synchronization overhead.

V. EVALUATION OF NEAR-FINE-GRAIN PARALLEL PROCESSING WITHOUT SYNCHRONIZATION

This section shows performance of the proposed scheme on OSCAR.

Figure 7(a) shows a sample FORTRAN sequential loop including 24 statements (near-fine-grain tasks) in the loop-body. Figure 7(b) shows a result of code scheduling applied to a loop body when five processors are used. As shown in Figure 7(b), instruction executions, bus accesses, and shared memory accesses is completely scheduled exactly the same as hardware behaviour of OSCAR.

Figure 8 shows the execution time per iteration plotted against the numbers of processors. Figure 8(a) shows the execution result of the sequential loop in Figure 7 on OSCAR. Upper thin solid line shows execution time with all synchronization instructions. Dotted line represents execution time after elimination of redundant synchronization. Lower thick solid line represents execution time with no synchronization instructions. The figures above the lines show number of synchronizations, that is S means flag set on remote DSM, C means flag check on local DSM.

For three processors, the processing time was reduced from 92.63μs with all synchronization instructions (18 flag sets, 26 flag checks and one barrier synchronization) to 61.76μs without explicit synchronization (32.6% speed up).
INTEGER V9, V10
REAL V1, V2, V3, V4, V5, V6, V7, V8
V1 = 0.0
V6 = 1.0
V7 = 1.0
V8 = 4.0
V3 = 0.0
V5 = 0.0
DO 10 V9 = 1, 100000
V10 = V9 - V9 / 100 * 100
V2 = V2 - 1.0 / (V6 + 2.0)
V2 = V2 + 1.0 / (V6 + 4.0)
V2 = V2 - 1.0 / (V6 + 6.0)
V3 = V3 - V7 / (V6 + 8.0)
V3 = V3 + V7 / (V6 + 10.0)
V3 = V3 - V7 / (V6 + 12.0)
V4 = V7 / (V6 + 14.0)
V5 = V5 + V7 / (V6 + 16.0)
V4 = V4 + V7 / (V6 + 18.0)
V5 = V5 + V7 / (V6 + 20.0)
V5 = V5 + V7 / (V6 + 22.0)
V4 = V4 + V7 / (V6 + 24.0)
V5 = V5 + V7 / (V6 + 26.0)
V4 = V4 + V7 / (V6 + 28.0)
V5 = V5 + V7 / (V6 + 30.0)
V1 = V1 + V8 / (V6 + 32.0)
V1 = V1 - V8 / (V6 + 34.0)
V1 = V1 + V8 / (V6 + 36.0)
V1 = V1 + V8 / (V6 + 38.0)
V2 = V2 + V4
V1 = V1 + 4.0 * V2
V6 = V6 + 40.0
V1 = V1 + 4.0 * V3 - 4.0 * V5
END

(a) Sequential Loop  (b) Result of Code Scheduling

Figure 7: An example program and its scheduled parallel code
Figure 8: Performance of near fine grain parallel processing

Of course, the proposed parallel execution without explicit synchronization gave us the correct calculation results, or the same as the sequential calculation results. This means that the proposed scheme guarantees all data dependencies among tasks without synchronization instructions by precise code scheduling.

Figure 8(b) shows execution time of FORTRAN program for solution of linear equation with randomly sparse coefficient matrix (60 x 60, 2.3% non zero elements) using loop free code. This program includes a large basic block having 96 statements, or 96 near fine grain tasks. In this case, the measured processing time for six processors were reduced from 1.791μs with all synchronization instructions to 1.541μs without explicit synchronization instructions on OSCAR simulator (13.7% speed up).

From the evaluation, it has been confirmed that the proposed near-fine-grain parallel processing without explicit synchronization gave us significant performance improvement.

VI. CONCLUSION

This paper has proposed the near fine grain compilation scheme which eliminates all synchronization instructions inside a basic block by using precise code scheduling scheme in clock level.

Also it has been confirmed that near fine grain parallel processing without explicit synchronization using the proposed scheme reduces processing time by 11% to 40% compared with ordinary near fine
grain parallel processing with synchronization instructions on a real multiprocessor system OSCAR.

OSCAR’s architecture is simple so that some PEs and CSM will be able to integrated into one chip in near future.

The authors are currently researching on optimization algorithms of data transfer order using clock level precise code scheduling and performance estimation of the proposed scheme with multigrain parallel processing on multiprocessors on chip.

ACKNOWLEDGEMENTS

This research was partly supported by the Ministry of Education Grant-in-Aid for Scientific Research No.(C)0780372 and Waseda University Grant for Special Research Projects No. 96B-033.

References


Data Dependence Analysis in Programs with Pointers

W. Amme, E. Zehendner

Computer Science Department
Friedrich-Schiller-University
D-07740 Jena, Germany

We describe a method to derive safe approximations for data dependences in programs with pointers by solving a data flow problem within a monotone data flow framework. In our approach alias information and reaching definitions' information at a program point are simultaneously covered by a single data structure, called A/D graphs. Advantages of our method are improved precision, economical storage use, and reduced analysis time.

1 Introduction

Data dependence analysis is an indispensable task within parallelizing compilers [1], and is also widely applicable towards various other software development techniques [17]; for example, source-level debugging, automated testing, semantic browsing, integration of independently altered versions of programs, and, last but not least, most of the code optimization tools of compilers [21].

There are three different kinds of data dependence, each relating two statements or two different statement instances of a program written in an imperative programming language: A true dependence is purely value-based and relates the producer and the consumer of a data item. An anti-dependence or an output dependence is caused by a memory cell used to store different values in succession; under certain conditions, these dependences can be resolved by renaming, i.e., providing separate memory cells for each data item. The techniques to perform data dependence analysis for each of these kinds of dependence are very similar, so literature usually concentrates on true dependences.

The precision of the performed data dependence analysis directly affects the efficacy of its application. Underestimation, i.e., detecting only a subset of the dependence relation, usually becomes disastrous in context of parallelizaton and other code restructuring techniques that rely on semantics preserving transformations. Overestimation, i.e., determining a proper superset of the dependence relation, is safe albeit it degrades the merits of the analysis. We are thus striving for safe approximations that are as precise as possible. However, it can be expected that a higher precision will require, during the analysis, more time to analyze the code and more storage.

An essential step in data dependence analysis is the calculation of reaching definitions for all statements or statement instances, i.e., the problem of determining, for a specific program point and storage location, all program points where the value of this storage location has been (or could have been) written last. Once the reaching definitions have been determined, we are then able to infer def-use associations; a def-use pair of statements indicates a true dependence between them.

In the presence of pointer variables or when performing interprocedural analysis with reference formals, a storage location might be accessed through several variable names or, more generally, through different access paths. This phenomenon is called aliasing. Alias analysis is an integral part of the reaching definitions calculation but
has also some interest on its own. The basic problem of alias analysis is to determine may aliases, i.e., access paths that during some execution history of a program refer to the same storage cell at the same program point. Thus alias analysis relates access paths at a single program point whereas data dependence analysis yields pairs of different program points. A variant of the may alias problem is the calculation of must aliases; a must alias is an alias that at a specific program point holds on any execution path covering this point. Must aliases can be used to improve the precision of the data dependence analysis; in contrast to may aliases, must alias analysis is safe if it yields an underestimation of the precise must alias relation.

Currently, data dependence analysis is performed by static analysis. During static analysis we abstract from the particular values contained in or pointed to by program variables. Hence we are not able to infer the direction of branches taken during program execution. Therefore, we have to consider all legal paths in the control flow graph of a program, which might be a proper superset of the execution paths actually appearing at run time, thus rendering our analysis inaccurate. Nevertheless, within the framework of static analysis, a dependence relation (or alias relation, or the like) usually is called precise if it is exact with respect to the legal paths in the control flow graph. While being consistent in the case of programs without pointers, this definition introduces the confusing situation in which a precise solution to the static alias analysis problem with pointers may also contain access paths that can never appear in any execution of the program; however, there is no such dilemma with data dependences. By extending static analysis with some kind of symbolic execution, for instance constant propagation, the set of paths under consideration may come closer to the set of proper execution paths but in general will remain a superset of the latter.

Virtually all problems in data dependence analysis resp. alias analysis can be formulated as data flow problems. A data flow problem is described as a set of recurrence equations whose least fixpoint - if one exists - is taken for the solution. This fixpoint can be successively better approximated by a well-known inductive process \([6,9,10]\) starting from a trivial approximation. Under certain conditions of boundedness this iterative algorithm terminates and yields the desired solution of the data flow system \([10]\). Data flow systems therefore have became a standard modelling tool in compiler construction; however, the efficiency of the solution mechanism heavily depends on the chosen implementation techniques.

2 Analysis of programs with pointers is different

In languages like FORTRAN aliases may already be introduced through reference formals. Languages that allow pointers, however, entail situations that can never be due to reference formals alone. As the value of a pointer variable can be defined by assignment, the alias relation may change in any statement of the program; in contrast, when using non-pointer reference formals, the alias relation is constant during the execution of a procedure's body and changes only at entry resp. exit of the procedure. Moreover, when using pointer-valued reference formals or global pointer variables in an interprocedural setting, the calling procedure may inherit some of the changes of the alias relation which occurred inside a called procedure.

As the address of a storage object can appear as the value of a pointer variable and thus may be passed to other pointer variables by assignment, there no longer exists a
fixed relation between variable names resp. access paths used within a program and storage objects handled by the program. In non-recursive programs without pointers, when arrays are treated as units, the size of the alias relation is bounded in the number of variable names and statements appearing in the program; this is also the case for programs with pointers when all storage objects are statically allocated or when only single level indirection via pointers is allowed.

However, when using dynamically allocated storage objects in combination with multi-level pointer indirection, (in principle) we can create data structures whose size is not bounded in any number inferable from the program code. Hence the size of the alias relation also cannot be bounded statically, and the alias analysis has to assume an unbounded number of objects. This leads to data flow frameworks with unbounded information sets that may have no finite fixpoint solution and thus aren’t computable by an inductive approach, so we have to rely on some kind of symbolic treatment of the problem. Our observations seem to be in conflict with the assumption of Pande [18] that major theoretical difficulties in solving alias, reaching definitions, and def-use problems for programs with multi-level indirection are inherent for programs using only single level indirection; however, the latter statement supposedly should be applied to interprocedural analysis, only.

Finally, as a bizarre observation, we note that during static analysis of programs with pointers we can also come across some spurious alias pairs where even the access paths considered are purely fictional and can never appear in any real execution of the program.

3 Previous work and state of the art

The first approach to determine data dependences in programs with pointers using $k$-bounded data structures as representations is due to Jones and Muchnick [8]. Much of the work which followed was based on this technique.

Larus and Hilfinger [15] have constructed a monotonic data flow framework for conflict analysis with programs written in Lisp. The data flow information set they use are alias graphs. Conflict analysis with alias graphs is very storage inefficient, as the analysis is performed in two steps and therefore the alias graphs for all statements must be stored at the same time. Our representation, in some respect, is also based on this work.

Horwitz, Pfeiffer, and Reps [7] proposed a method for data dependence analysis. They perform a data flow analysis in which the data information set is composed of so-called store graphs. Each of the graphs describes various possible structures for storage. This work, being so theoretical in nature, makes a practical implementation appear very difficult.

Conditional alias analysis, as suggested by Landi [12], allows for the treatment of program-point-specific pointer-induced aliases, for each analyzed procedure, and combines the results of the analysis for different procedures afterwards. Based on this work, Pande, Landy, and Ryder [17] developed the first polynomial-time approximate algorithm for interprocedural def-use associations with single level pointers. Landi and Ryder [11] also gave a safe approximate algorithm for may-alias analysis with multi-level indirection.
Deutsch [5] proposed a memoryless model to solve the may alias problem, employing algebraic methods.

Matsumoto, Han, and Tsuda [16] propose a two-stage dependence analysis to handle programs with linearly linked lists or trees. The first stage is a safe alias analysis, where variables are supposed to be aliased except when they definitely point to disjoint data structures. The second stage detects linearly linked lists, and then performs a dependence analysis between pointer references to the same list. The method cannot be used outside the scope of detecting linearly linked lists or trees. Even for handling these kinds of data structures, the accuracy of the method is poor due to very coarse approximations in the different phases that are passed through the whole analysis. For instance, this method is not able to determine the fact that there is no true dependence between the second and the third loop in our example from figure 7.

Sagiv, Reps, and Wilhelm [19] give an algorithm to determine the possible shapes that heap-allocated structures in a program can take on. For certain programs the method is an accurate means to derive the storage shape for cyclic data structures. As in our method, manipulation of their representation is based on sets of access paths, distinguishing this work from all other approaches.

In a previous paper [2] we stated an algorithm for precise data dependence analysis with single level indirection, also based on representations by A/D graphs.

4 A structure for static determination of data dependences

In our method, we perform a data dependence analysis for programs with pointers by means of a data flow analysis. For this purpose we have to generate a control flow graph for the program in advance. During our static analysis we are usually not able to obtain any information on the length of the chained storage objects, for instance a linked list. If we intended to analyze a program that uses recursive data structures, we would have to generate an infinite number of objects in the case of a precise static analysis, thus rendering the analysis impracticable. Therefore we refrain from an accurate static analysis and rely on an approximate static analysis that uses a finite graph structure to describe the store (as, for example, also in [3]).

We use approximate data dependence description graphs (A/D graphs) as data flow information sets (explained below). Each A/D graph denotes, in a finite form, the structure of the store and some aspects of its state at a certain point in program execution. From A/D graphs we can derive essentially the following informations: First, the combination of paths with which we can access a storage cell at one program point (i.e., the alias information); second, the program statements which defined respectively used the contents of an object last (i.e., the reaching definitions). The actual data dependences for the statement in consideration can then be read off by looking for relevant def-use combinations concerning the same storage cell.

The nodes of an A/D graph represent the objects present in the store. A reference of one of these storage objects to another storage object by use of a pointer is expressed by an edge in the A/D graph. There are two types of nodes in an A/D graph: simple and condensed. A simple node represents an object that can be accessed through any of the paths by which we can reach this node in the A/D graph. By the introduction of condensation nodes, we achieve the finiteness of our graphs. In A/D graphs a
condensation node is self-labelled by an outgoing edge to itself. A condensation node stands for all objects that can be accessed through the infinite set of paths targeting the condensation node in the A/D graph.

The use of A/D graphs for our data dependence analysis may lead to some loss of accuracy during the analysis. Nevertheless we get an approximation that is safe, i.e., all data dependences are detected but the analysis may report some false dependences, too. In general this kind of inaccuracy cannot be avoided when performing a static analysis, only.

Figure 1 shows a program and a corresponding A/D graph that describes the store immediately before execution of statement 11. The nodes of this A/D graph record write accesses to memory cells. The program segment presented here generates a linked list with 10 elements and initializes these as a function of the index value. In the A/D graph 2 list elements are shown as distinct nodes; the further 8 list elements are represented by a condensation node. From this A/D graph we infer a true dependence of statement 11 on statements 1, 5, 7, 2, and 6.

5 Programming model

The data flow analysis presented in the following sections works for properly restricted imperative languages (for our examples we used a subset of Modula-2). In [2] we offer a data dependence analysis with A/D graphs that works well for programs that handle data structures with at most one level of indirection, i.e., pointer variables may not refer to structures that contain other pointer variables. In this paper, we extend our programming model to target programs that also handle acyclic linked lists. Pointer variables can obtain their values via assignment or by allocation; arithmetic on pointers is disallowed. Further we assume that our programs have been fully type-checked and that all pointer dereferences in programs are correct.
Interprocedural analysis is outside the scope of this paper. Nevertheless the data dependence analysis with A/D graphs can be extended to handle programs with procedure calls. In [2] we use a partial inline-expansion [4] which allows a simple, though accurate interprocedural analysis for non-recursive programs. By using one of the known standard approaches (see, for example, [14,20]), we can extend our analysis method to handle recursive programs, too. Clearly such an imprecise treatment of recursion is not sufficient in state-of-the-art compilers, however at the beginning of this work we want to concentrate on the essential novelties of our method.

6 Intraprocedural data dependence analysis with A/D graphs

As already mentioned, we use a data flow analysis for the determination of data dependences. Data flow problems can be solved in a uniform way by using the concept of monotone data flow systems [21]. By doing so, we can use a well-known general algorithm for static analysis [6,9,10] to solve our problem. Data flow problems are modelled by data flow frameworks [21]. A data flow framework is a triple \((L, V, F)\), where \(L\) is the data flow information set, \(V\) is the union operator, and \(F\) is the set of semantic functions. The general algorithm always terminates and calculates a safe solution for a data flow problem if \((L, V)\) forms a bounded semi-lattice with one and zero element and if the semantic functions are monotone.

Let \(S\) be the set of program statements and \(V\) the set of variables. If, to simplify matters, we assume that each node of a linked list contains only one data field, then an A/D graph \(A\) is a quadruple \(A = (N, E, Def, Entry)\), where \((N, E)\) is a directed labelled graph, \(Def \subseteq N \times P(S) \times P(S)\) is the set of node labels, and \(Entry \subseteq V \times N\) is the set of entry points into the graph. \(A\) is called \(l\)-bounded, if there is no path leading to a simple node that is longer than \(l\). Because of well definition, in \(l\)-bounded A/D graphs, there are no outgoing edges from a condensation node except those directed to the same node. Further, for all predecessor nodes of a condensation node, there exists exactly one outgoing edge. For a fixed \(l \in \mathbb{N}\), the set of all \(l\)-bounded A/D graphs is denoted by \(AD_l\).

We use \(l\)-bounded A/D graphs as data flow information set, where \(l \in \mathbb{N}\) can be chosen to guarantee the desired precision of the analysis. If a node of the control flow graph has more than one predecessor, we must integrate all information stemming from these predecessors. In data flow frameworks, the union operator describes the effect of joining paths in the flow graph. Figure 2 describes the union operator \(Union\) on \(l\)-bounded A/D graphs. First, the union operator uses the join operator \(\cup\) on A/D graphs to express the fusion of the contained data flow information. The nodes of

\[
Union(A, B) = \text{Reduce}(A \cup B), \text{ where} \\
\text{Reduce}(A) = \\
\text{while } \exists n_1 \in N_C, n_2, n \in N, n \neq n_1 \text{ and } \exists (n_2, n_1), (n_2, n) \in E \text{ do} \\
\quad N := N \setminus \{n\} \\
\quad E := E \cup \{(n_1, n) \in E\} \cup \{(n, n') \in E\} \\
\quad \cup \{(n', n_1) | \exists (n', n) \in E\} \cup \{(n_1, n') | \exists (n, n') \in E\} \\
\quad D := D \cup \{(n, d, p), (n_1, d_1, p_1) \in D\} \\
\quad \cup \{(n_1, d_1, p_1) | \exists (n_1, d, p) \in D\} \\
\text{od;} \\
\text{return } (N, E, D)
\]

Figure 2: Union operator on A/D graphs.
an A/D graph can be uniquely distinguished by the sets of access paths reaching the nodes. The information from the two A/D graphs can be joined as follows: Nodes that are reachable in both graphs by the same set of paths are collapsed to a single node. To this node we attach the union of the label fields of the collapsed nodes.

Unfortunately the join operator does not always yield an $l$-bounded A/D graph. The function \textit{Reduce} transforms the graph $A \cup B$ into an $l$-bounded graph. For this purpose \textit{Reduce} iteratively merges with a condensation node those nodes that would become reachable through the condensation node or via a predecessor thereof. To the condensation node, we attach the union of the labels of the merged nodes. An application of the union operator on two A/D graphs is shown in figure 3.

We have proven that for a fixed $l \in \mathbb{N}$, the structure $(AD_l, \text{Union})$ is a semi-lattice that is bounded, as the number of $l$-bounded A/D graphs is always finite. For each program we find an one element and a zero element of the semi-lattice depending on the variables used in the program. Figure 4 shows these elements for a program containing only the variables $I_1, I_2, L_1, \ldots, L_4$; $I_1$ and $I_2$ are of type \textit{INTEGER} and $L_1, \ldots, L_4$ are of type \textit{POINTER TO LIST}, where \textit{LIST} is a record with fields \textit{data} and \textit{next}, which describes the structure of the list elements.

In the control flow graph chosen to be used during the analysis, each node stands for a uniquely labelled program statement. Therefore we can unambiguously assign a \textit{semantic function} to each of the nodes; this semantic function will be used to transform the A/D graph when processing the node. Figure 5 specifies the semantic functions used by our method. In this specification, \textit{AD$_{in}$} stands for an A/D graph before the application of the semantic function, and \textit{AD$_{out}$} for the corresponding A/D graph after the application of the semantic function. We limit our description to the case of true dependences and output dependences. These semantic functions have to be adjusted for handling anti dependences, i.e., instead of write accesses we have to note read accesses in the A/D graphs.

![Figure 4: Example of an one element and a zero element.](image)
Assignment to pointer variables

- \( n: X := Y \)
  \[ A_{D_{out}} = \text{Define}(\text{InsertAndComplete}(\text{DeleteEdge}(A_{D_{in}}, X, Y), X, Y), X, n, \text{Field}(X)) \]
- \( n: X := \text{NIL} \)
  \[ A_{D_{out}} = \text{Define}(\text{DeleteEdge}(A_{D_{in}}, X, Z), X, n, \text{Field}(X)), Z \not\in \text{Var} \]
- \( n: \text{New}(X) \)
  \[ A_{D_{out}} = \text{Define}(\text{GenerateNode}(\text{DeleteEdge}(A_{D_{in}}, X, Z), X), X, n, \text{Field}(X)), Z \not\in \text{Var} \]
- \( n: \text{Dispose}(X) \)
  \[ A_{D_{out}} = \text{Define}(\text{DeleteEdge}(A_{D_{in}}, X, Z), X, n, \text{Field}(X)), Z \not\in \text{Var} \]

Assignment to non-pointer variables

- \( n: X := \ldots \)
  \[ A_{D_{out}} = \text{Define}(A_{D_{in}}, X, n, \text{Field}(X)) \]

Boolean expression

- \( n: l := [1 \text{ to } N] \)
  \[ A_{D_{out}} = \text{Define}(A_{D_{in}}, l, n, \text{Field}(X)) \]
- \( n: X \text{ opBool } Y \)
  \[ A_{D_{out}} = A_{D_{in}} \]

Figure 5: Semantic functions.

These semantic functions are composed of auxiliary functions that perform transformations on \( l \)-bounded A/D graphs. \( \text{Define}(A, X, n, f) \) registers the label \( n \) in the field \( f \) of every simple node of the A/D graph \( A \), that is a target of path \( X \). Otherwise, if \( X \) leads to a condensation node, the label of field \( f \) of this node will be joined with \( n \). \( \text{GenerateNode}(A, X) \) first marks all simple nodes in \( A \), that are reachable via a path \( X \). Then \( \text{Generate}(A, X) \) starts a new edge on every marked node that cannot be reached via a path with length \( l \), and attaches a blank node to it. In contrast, all marked nodes, that are reachable via a path with length \( l \), will be transformed into a condensation node, by inserting an edge to itself. \( \text{DeleteEdge}(A, X, Y) \) transforms the \( l \)-bounded A/D graph \( A \) first into an expanded A/D graph—in this graph, there exists no node that can be reached via different paths starting at the same variable. Then we mark all simple nodes in the graph that are reachable via \( X \), but not via a prefix of \( Y^- \). From any marked node, we delete all edges emanating from this node and perform a garbage collection thereafter, i.e., we delete all nodes not reachable via any path starting at a variable. After this, the graph will be compressed—meaning we collapse all nodes that are reachable by the same set of paths in the same way as described previously. Because of well definition we have to finally transform the emerging graph with the function \( \text{Reduce} \) of figure 2 into a \( l \)-bounded A/D graph.

\( \text{InsertAndComplete}(A, X, Y) \) first expands the \( l \)-bounded A/D graph \( A \). Because the function handles nodes that are reachable via path \( X \) and via a prefix of \( Y^- \) resp. nodes that are reachable via \( X \), but not via a prefix of \( Y^- \) in a different way, we can divide the function into two steps. Let \( \text{Var}(Y) \) be the variable, with which the path \( Y \) starts. In a first step we mark all nodes in the graph that are reachable via \( X \) and via a path starting at \( \text{Var}(Y) \), as well as all these node's outgoing edges. For any
marked node \( N \) we insert an edge from \( N \) to all successors of \( N \), that are reachable via \( Y^* \) into the graph, and delete all marked edges emanating from \( N \), thereafter. Finally we perform a garbage collection, i.e., we delete all nodes not reachable via any path starting at a variable. In a second step we have to mark all nodes in the graph, that can be reached via a path \( X \), but not via a path starting at \( Var(Y) \). For any marked node \( N \) we must do the following: Therefore let \( node \) be the set of nodes in the graph which are reachable via path \( Y^* \), but not reachable via a path starting at a variable for which there exist a path to \( N \), starting at the same variable. Then copy each subgraph that starts by an element \( M \in node \) and insert an edge from each predecessor node of \( M \), and from \( N \), to the entry node of this subgraph.

Before compressing the graph and applying the function \( Reduce \), we have to examine whether there is a simple node \( N \) in the graph that can be reached via a path that is longer then \( l \). If this is the case, for any node \( N \) we must insert a condensation node and an edge from each predecessor node of \( N \) to this node into the graph.

The result of applying the function \( InsertAndComplete(A, X, Y) \) can be improved, if there is exactly one node in \( A \) that is reachable via a path \( X \). If this node is not reachable via a path starting at \( Var(Y) \), we can perform an additional garbage collection, i.e., before compressing the emerging graph we delete all successors from any node \( N \in node \) and the nodes itself. If \( A \) contains more than one node reachable via \( X \), then always to consistently perform garbage collection may yield wrong information, because it is unknown whether all objects reachable via \( X \) are always present in storage at the considered program point.

To obtain a better sense of how this semantic function works, figure 6 illustrates step by step the work method of a semantic function which assigns to a node that represents an assignment to a pointer variable.

We have proven that the semantic functions are monotone: Let \( f \) denote a semantic function, and \( A \) as well as \( B \) \( l \)-bounded \( A/D \) graphs, then \( \text{Union}(f(A), f(B)) \leq f(\text{Union}(A, B)) \). \( \leq \) is the partial order induced by the union operator, \( a \leq b \iff \text{Union}(a, b) = a \). Since for all programs considered here \((AD_i, \text{Union})\) forms a

![Figure 6: Application of a semantic function](image-url)
Figure 7: Data dependence analysis with A/D graphs.
bounded semi-lattice with one and zero element, to solve the data flow framework we can use the previously cited general algorithm that always terminates and gives a solution that is safe.

Figure 7 shows an example of a data flow analysis with 2-bounded A/D graphs. For each node of the control flow graph we determined an A/D graph that describes the store immediately before the execution of the corresponding statement. From these graphs we are now able to determine, for a statement, all true and output dependences with respect to other statements.

The number of nodes in a l-bounded A/D graph might grow exponentially with the number of variables. Although this case appears rarely in practice [13], we have ruled out such a storage complexity on principle. For this purpose we use k-l-bounded A/D graphs. We call a l-bounded A/D graph k-l-bounded, if each node in the graph points to no more than k objects (k arbitrary but fixed). By extending the function InsertAndComplete, we can build a data flow framework for the k-l-bounded A/D graph. Therefore in addition to the examination of the path lengths, we must check for nodes with more than k outgoing edges in the graph. If that is the case, for any detected node N we must insert a blank condensation node and an edge from N to this on the graph. Because of this, the resulting graph of the subsequent application of Reduce is a safe approximation of this graph. As before InsertAndComplete will be further monotone. By using k-l-bounding A/D graphs we can limit the storage needed during the analysis to be quadratic in the number of program statements.

7 Conclusions

We have presented a single-pass method to derive data dependences in programs with pointers that is safe, accurate, fast, and storage economical. Our method is based on a representation called A/D graphs, covering both reaching definitions and alias information at the same time. Actually we are investigating slightly different representations with corresponding union operator and semantic functions. The question of tradeoffs between precision, storage use, and analysis time will be clarified after prototype implementations and intensive experimental work.

References

Interoperability of Data Parallel Runtime Libraries with Meta-Chaos *

Guy Edjlali, Alan Sussman and Joel Saltz
Department of Computer Science
University of Maryland
College Park, MD 20742
{edjlali,ais,saltz}@cs.umd.edu

Abstract
This paper describes a framework for providing the ability to use multiple specialized data parallel libraries and/or languages within a single application. The ability to use multiple libraries is required in many application areas, such as multidisciplinary complex physical simulations and remote sensing image database applications. An application can consist of one program or multiple programs that use different libraries to parallelize operations on distributed data structures. The framework is embodied in a runtime library called Meta-Chaos that has been used to exchange data between data parallel programs written using High Performance Fortran, the Chaos and Multiblock Parti libraries developed at Maryland for handling various types of unstructured problems, and the runtime library for pC++, a data parallel version of C++ from Indiana University. Experimental results show that Meta-Chaos is able to move data between libraries efficiently, and that Meta-Chaos provides effective support for complex applications.

1 Introduction
Distributed parallel programs are used to speed up the time to complete an application. To achieve this goal, such programs rely on partitioning data and computation among the available processors. They are considered difficult to write, to maintain and to modify. Many parallel programming paradigms have been developed, with the most important one for large scale scientific applications being data parallelism. Data parallel applications can currently be written using a high level language, for example High Performance Fortran (HPF) [13] or pC++ [3]. Data parallel programs can also be written using a sequential programming language and runtime libraries for performing communication. These libraries can be low level communication libraries such as MPI [17] or PVM [7], or application specific runtime libraries that encapsulate communication into higher level functions, such as ChaoS [12] or LPARX [14]. However, inter-application communication to allow multiple data parallel programs to cooperate to solve a single problem is rare, because such programs are difficult to write and there are few tools available to develop them.

To motivate our work, we present the following simple scenario. A client program, running sequentially or in parallel, requires the services of a parallel server, on the same or another (parallel) machine. The server could provide functionality that is not available in the client, or provide additional computational power to make the client run significantly faster. Let us be more concrete with our example: let the client be a sequential C program and the server be an HPF parallel program, and the two programs exchange one parameter: a matrix of size $M \times N$. The matrix is stored as a two-dimensional array in both the client and the server. In addition, suppose the array is distributed on the server in a block-cyclic fashion, to optimize the server computation. Because of the block-cyclic distribution of data on the server, the client process must communicate with every server process. This operation requires a collective communication operation [17]. The client must determine which parts of the matrix are going to be sent to each of the processes in the server. Similarly, each server process must determine which part of the array it will receive from the client, and the order in which the array elements will arrive. To determine this information, the client process needs to know the data distribution on

*This research was supported by NASA under grant #NAG-1-1485 (ARPA Project Number 8874) and by ARPA under grant #F19628-94-C-0057. The Maryland IBM SP2 and Digital AlphaServer used for the experiments were provided by NSF CISE Institutional Infrastructure Award #CDA#401181 and grants from IBM and Digital Equipment Corporation.
the server. However, only knowing the data distribution is insufficient to perform the communication for this example. The matrix stored in the memory of the client is laid out in row major order (C style), while the matrix stored on each processor of the server is stored in column major order (Fortran style). Therefore, the mapping between the data elements on the client side and on the server side has to be specified. This example illustrates the three points we focus on in this paper: collective communication, data distribution, and data mapping.

In this paper, we present a meta-library approach that achieves direct application to application data transfer. By a meta-library, we mean a runtime library-based system that interacts with the data parallel libraries and languages used to implement the separate applications. The meta-library can handle any data distribution, and can be used for any data parallel library or language construct that distributes data for the sequential or parallel applications. The meta-library can be used to allow the exchange of data between separate (sequential or parallel) programs, and can also be used to allow data transfers between data managed by different data parallel libraries in the same application.

An example that illustrates the utility of the meta-library approach comes from a computational aerodynamics problem. Computational fluid dynamics (CFD) flow solvers often use different types of meshes to represent different physical structures. For example, the space around an airplane body may be modeled with a structured mesh, while the nose, wing and tail may be modeled with an unstructured mesh. The data parallel numerical solution techniques used for the flow fields employ algorithms specifically designed for each type of mesh and often use runtime library support optimized for a particular solution technique. To allow interactions between the different meshes at their shared boundaries, it is necessary for the different parallel libraries that distribute the meshes to exchange data. However, such functionality is not easily achieved for arbitrary libraries. The meta-library framework we describe in this paper provides the mechanisms needed to perform the communication.

In sequential programs, intra-application and inter-application communication is commonly used in a networked environment. Such programs often use low level communication calls (e.g., sockets) to move data between separate address spaces. Building manageable distributed applications with these low level communication calls can be difficult. Therefore many distributed applications instead use the Remote Procedure Call (RPC) [2] paradigm to hide many communication details from the application programmer. RPC extends the notion of a procedure call in a sequential program by allowing the transfer of both data (parameters and return values) and control over a network from one process to another. Two processes are involved in the RPC call: the client and the server. In addition, the CORBA object model [16] provides RPC-like capability for a distributed object model. RPC provides a simple, efficient programming model for heterogeneous sequential applications. Our meta-library framework can be seen as a step towards allowing RPC-like functionality for data parallel programs, by providing the infrastructure needed to pass data between parallel or sequential clients and servers.

Much effort has recently gone into languages and runtime libraries for combining task and data parallelism. For example, the data parallel language Flx [19] extended HPF to support task parallelism. However, none of that work addressed the problem of allowing programs written using different data parallel libraries or languages to inter-operate. All the systems that we will discuss in Section 6 are designed to work only within a single (extended) language or with one data parallel library. In addition, none of those designs allow multiple application-specific data parallel libraries to exchange data within a single program. Our meta-library based system works both for separate data parallel programs written using any data parallel library or language (that exports the required set of inquiry functions) and for a single data parallel program that uses multiple data parallel libraries to optimize performance.

We have developed a prototype implementation based on the meta-library approach. We call our runtime library Meta-Chaos. An implementation currently runs on the IBM SP2 multicomputer, an eight-node Digital Alpha cluster of SMPs and a cluster of workstations. The libraries and languages currently supported by Meta-Chaos include Fortran, C, HPF [18], pC++ [3], Multiblock Parti [1] and Chaos [12]. Our results indicate that this approach is feasible and that the overhead of our general approach is acceptable. Our results also show that the data parallel library extensions required by the meta-library are not difficult to implement, even by someone other than the implementor of Meta-Chaos.

The rest of the paper is organized as follows. Section 2 presents a high level overview of interoperability between two data parallel libraries, while Section 3 discusses the system design of the meta-library. Section 4 describes an implementation of the meta-
library, called Meta-Chaos, and Section 5 presents several experiments designed both to quantify the overhead costs encountered when an application uses Meta-Chaos and to show the benefits that can be obtained from using Meta-Chaos to structure an application. Section 6 compares the meta-library approach to previous work, and we conclude in Section 7. The appendix illustrates the use of Meta-Chaos to allow two HPF programs to exchange data.

2 Basic Concepts

Figures 1 and 2 provide a high-level view of interoperability between two data parallel libraries, for two different scenarios. Suppose we have programs written using two different data parallel libraries named LibX and LibY, and that data structure A is distributed by LibX and data structure B is distributed by LibY. Then the scenario presented in Figure 1 consists of copying multiple elements of A into the same number of elements of B, with both A and B belonging to the same data parallel program. On the other hand, the scenario presented in Figure 2 copies elements of A into elements of B, but A and B belong to different programs. Our system design provides a standard set of techniques for performing the copy operation.

The two examples show the main steps needed to copy data distributed using one library to data distributed using another library. More concretely, these steps are:

1. Specify the elements to be copied (sent) from the first data structure, distributed by LibX.
2. Specify the elements to be copied (received) into the second data structure, distributed by LibY.
3. Specify the correspondence (mapping) between the elements to be sent and the elements to be received.
4. Build a communication schedule, by computing the locations (processors and local addresses) of the elements in the two distributed data structures.
5. Perform the communication using the schedule produced in step 4.

These 5 steps could be performed directly by the application programmer. However this supposes that the application programmer

- knows the locations of the source data A1 on the processors where A is distributed by LibX (Perhaps requiring the programmer to look at internal data structures of LibX),
- knows the location of the destination data B1 on the processors where B is distributed by LibY (Perhaps requiring the programmer to look at internal data structures of LibY),
- can easily determine the mapping between the elements to be sent from A and the elements to be received into B (note that A and B may be in different programs),
- compute a communication schedule,
- and perform the communication.

The application programmer could perform these operations, but that requires the programmer to understand the management of both A and B, even if the management is encapsulated in a data parallel runtime library. The resulting code could be difficult to optimize across multiple parallel platforms, and any changes to the management of the data structures would have to be reflected in the implementation of the algorithm, so code maintenance would be difficult. Also, determining the mapping between elements in A1 and elements in B1 could be complex.

However, the data parallel library writer knows the internal data structures and behavior of the library and can optimize the performance of operations on data distributed by the library. This observation leads us to require the library builder to export some additional services, to support interoperability with other data parallel libraries. The application programmer specifies the data to be moved for both the source and destination. On the other hand, the library writer provides a way to determine the location of the data (processor and local address). The mapping between the source data and the destination data must also be specified. This can be done via a process we call linearization. Providing a linearization is the responsibility of the library writer, and will be described in Section 3.3.

A new piece of software, which we call a meta-library, must be written to compute the communication schedule at runtime. The meta-library uses the specification of the data to be sent and received by the application programmer(s) and the services exported by the data parallel libraries to build the schedule. The meta-library can also provide data transport routines that are appropriate for the execution environment (e.g., message passing). Using the meta-library, the application programmer only defines the source and destination of the data to be transferred, and the meta-library does the rest of the work.

3 System design

3.1 Options

There are at least three potential solutions to provide a mechanism for allowing data parallel libraries to interoperable. The first approach is to identify the unique features provided by all existing data parallel libraries and implement those features in a single in-
integrated runtime support library. The major problem with such an approach is extensibility.

A second approach is to use a custom interface between each pair of data parallel libraries that must communicate. However, if there are a large number of libraries that must interoperate, say n, this method requires someone to write n² communication functions. So this approach also has the disadvantage of being difficult to extend.

The third approach is to define a set of interface functions that every data parallel library must export, and build a meta-library that uses those functions to allow all the libraries to interoperate. This approach is often called a framework-based solution, and is the one we have chosen for our design. This approach gives the task of providing the required interface functions to the data parallel library developer (or a third party that wants to be able to exchange data with the library). The interface functions provide information that allows the meta-library to inquire about the location of data distributed by a given data parallel library. Providing such functions does not prevent a data parallel library developer from optimizing the library for domain-specific needs.

3.2 Data specification

The data to be transferred are specified by the application programmer. The data are distributed and otherwise managed by one or more data parallel libraries. Most such libraries provide a compact way to describe groups of elements in a distributed data structure (e.g., a range of subscripts for a distributed array). We will call this description of a set of elements a Region type. Hence the library builder must specify the Region type for a given library, so that the meta-library will be able to map elements in the source data parallel library to elements in the destination library. For example, High Performance Fortran (HPF) [13] and Multiblock Parti [1] utilize arrays as their main distributed data structure; therefore the Region type for them is a regularly distributed array section. Chaos [12] employs irregularly accessed arrays as its main distributed data structure, either through irregular data distributions or accesses through indirection arrays. For Chaos the Region type would be a set of global array indices.

A Region type is dependent on the requirements of the data parallel library. The library builder must provide a Region constructor for each Region type to create regions and a destructor to destroy the Regions specified for that library.

A single Region is not always sufficient to characterize the data to be moved. Thus multiple Regions to be moved can be specified. Regions are gathered into an ordered group we call a SetOfRegions. A mapping between source and destination data structures therefore specifies a SetOfRegions for both the source and the destination.

3.3 Linearization

Linearization is the method by which the meta-library can define an implicit mapping between the source of a data transfer distributed by one data parallel library and the destination of the transfer distributed by another library. The source and destination data elements are each described by a SetOfRegions.

One view of the linearization is as an abstract data structure that provides a total ordering for the data elements in a SetOfRegions. The linearization for a Region is provided by the library writer.

We represent the operation of translating from the SetOfRegions $S_A$ of $A$, distributed by LibX, to its linearization, $L_{SA}$, by $\varphi_{LibX}$, and the inverse opera-
tion of translating from the linearization to the SetOfRegions as $\ell_{libX}^{-1}$:

$$L_{S_A} = \ell_{libX}(S_A)$$

$$S_A = \ell_{libX}^{-1}(L_{S_A})$$

Moving data from the SetOfRegions $S_A$ of $A$ distributed by libX to the SetOfRegions $S_B$ of $B$ distributed by libY can be viewed as a three-phase operation:

1. $L_{S_A} = \ell_{libX}(S_A)$
2. $L_{S_B} = L_{S_A}$
3. $S_B = \ell_{libY}^{-1}(L_{S_B})$

The only constraint on this three-phase operation is to have the same number of elements in $S_A$ as in $S_B$, in order to be able to define the mapping from the source to the destination linearization (the second operation).

The concept of linearization has several important properties:

- It is independent of the structure of the data, and thus very flexible. Any data structure can be transferred to any other data structure, so long as a mapping can be specified. For example, several elements of a distributed tree data structure can be transferred to several elements of a distributed array data structure as long as:
  - a linearization is provided by the library used to create the distributed tree data structure,
  - a linearization is provided by the library used to create the distributed array structure,
  - and the number of elements specified to be transferred are the same in the tree and the array.

- It does not require the explicit specification of the mapping between the source data and destination data. The mapping is implicit in the separate linearizations of the source and destination data structures.

- It is only an abstract, not a physical object. No space must be allocated for the linearization of a SetOfRegions in the memory of either the source or the destination program. The meta-library can transfer data directly from the source SetOfRegions to the destination SetOfRegions, never building a data structure for the linearization.

- A parallel can be drawn between the linearization and the marshal/unmarshal operations for the parameters of a remote procedure call. Linearization can be seen as an extension of the marshal/unmarshal operations to distributed data structures.

For each object $i$ in the source SetOfRegions:

1. Determine the corresponding object belonging to the destination SetOfRegions using the source and destination linearizations. Call this object $j$.

2. Determine the owner (processor and local address) of object $i$, $P_i$, using the inquiry functions provided by the source data parallel library.

3. Determine the owner of $j$, $P_j$, using the inquiry functions provided by the destination data parallel library.

4. In the schedule for $P_i$, insert a send of object $i$ to $P_j$.

5. In the schedule for $P_j$, insert a receive into object $j$ from $P_i$.

Figure 4: Meta-library schedule computation algorithm

- For optimization purposes, multiple linearizations can be provided by a library writer for the same Region type in a single data parallel library. This capability does not create significant difficulties for the developer of the meta-library.

3.4 Example

Figure 3 shows a data copy from distributed array $A$ into distributed array $B$, with the SetOfRegions defined as shown. For this example, a Region for the array is a regular section, and the order within a section is row major. The SetOfRegions for $A$ and $B$ define the 1-1 mapping between elements for the copy.

3.5 Data parallel library extensions

A communication schedule describes the data motion to be performed for the specified transfer. From the SetOfRegions specified by the application programmer the meta-library can determine the elements to be moved, and where to move them. The meta-library applies the (data parallel library-specific) linearization mechanism to the source SetOfRegions and to the destination SetOfRegions. The linearization mechanism generates a one-to-one mapping between elements of the source SetOfRegions and the destination SetOfRegions. A high-level algorithm for computing the communication schedule is shown in Figure 4.

Implementation of the schedule computation algorithm requires that a set of procedures be provided by both the source and destination data parallel libraries. These procedures are essentially a standard set of inquiry functions that allow the meta-library to perform operations such as:

1) dereferencing an object to determine the owning processor and local address, and a position in the
linearization,

2) manipulating the Regions defined by the library to build a linearization, and

3) packing the objects of a source Region into a communication buffer, and unpacking objects from a communication buffer into a destination Region.

4 Implementation - Meta-Chaos

We have implemented our system design for the meta-library on a network of four-processor SMP Digital Alpha Server 4/2100 workstations, on an IBM SP2, and on a network of Sun workstations. We call the system Meta-Chaos, because it borrows several of its implementation techniques from Chaos.

4.1 Interface to the Library Builder

A major concern in designing the system was to require that relatively few procedures be provided by the data parallel library implementor, to ease the burden of integrating a new library into the Meta-Chaos framework. So far, implementations for several data parallel libraries have been completed, including the High Performance Fortran runtime library, the Maryland Chaos and Multiblock Parti libraries for various types of irregular computations, and the pC++ [3] runtime library, Tulip, from Indiana University. The pC++ implementation of the required functions was performed by the pC++ group at Indiana in a few days, using MPI as the underlying message passing layer, which shows that providing the required interface is not too onerous.

4.2 Communication cost

Once a communication schedule has been computed, using the algorithm from Figure 4, Meta-Chaos uses the information in the schedule to copy data into contiguous communication buffers in each processor managed by the data source parallel library. Similarly, Meta-Chaos uses the information in the schedule to extract data from communication buffers into the memory of each processor managed by the destination data parallel library. The communication buffers are transferred between the source and destination processors using either the native message passing mechanism of the parallel machine (e.g., MPI or MPI on the IBM SP2), or using a standard message passing library on a network of workstations (e.g., PVM or MPI). Messages are aggregated, so that at
most one message is sent between each source and each destination processor.

A set of messages crafted by hand to move data between the source and the destination data parallel libraries would require exactly the same number of messages as the set created by Meta-Chaos. Moreover, the sizes of the messages generated by Meta-Chaos are also the same as for the hand-optimized code. The only difference between the two set of messages would be in the ordering of the individual objects in the buffers. This ordering depends on the order of the bijection between the source objects and the destination objects used by Meta-Chaos (which is based on the linearizations provided by the source and destination data parallel libraries), and the order chosen by the hand-crafted procedure.

The overhead introduced by using Meta-Chaos instead of generating the message passing by hand is therefore only the computation of the communication schedule. Since the schedule can often be computed once and reused for multiple data transfers (e.g., for an iterative computation), the cost of creating the schedule can be amortized.

4.3 Heterogeneous environments

In an heterogeneous environment, data format incompatibility problems may arise. This problem can be solved by using a standard format, such as XDR [18]. Meta-Chaos supports heterogeneous environments by using XDR, with its implicit typing of data objects in a message buffer, as required.

4.4 Non-array aggregate data structures

Meta-Chaos requires only linearizations (i.e. a one to one implicit mapping from the source to the destination SetOfRegions) to be able to exchange data between different data parallel libraries. Therefore Meta-Chaos is not constrained to work only for distributed arrays, but can also be used for other distributed aggregates, including pointer-based structures such as trees and graphs. The only constraint is that each library that supports non-array distributed data structures provide a method for linearizing the data structures it supports. For example, Meta-Chaos could be used to allow two C++ programs parallelized using pC++ constructs [9], each containing compatible pointer-based data structures, to exchange parts of the data structures. More complex scenarios, with programs parallelized using different libraries and exchanging data between non-array data structures, can also be supported.

5 Experimental Results

We present two classes of experiments to evaluate the feasibility of using Meta-Chaos for efficient interaction between multiple data parallel libraries. The first class of experiments, in Sections 5.1 and 5.2, presents a set of application scenarios that quantify the overheads associated with using Meta-Chaos. The second class of experiments, in Section 5.3, is designed to show the benefits that Meta-Chaos can provide by allowing a sequential or parallel client program to exploit the services of a parallel server program implemented in a data parallel language (HPF).

Meta-Chaos is designed to operate efficiently for at least two significantly different patterns of communication. The first pattern, often called irregular communication, requires specifying each element separately (e.g., through an indirection array), while the second pattern is more regular, and usually specifies entire groups of elements in a compact way (e.g., with a regular section). To evaluate the overheads incurred in using Meta-Chaos, we present experiments that generate communication schedules and copy data for both types of communication patterns. These two patterns provide upper and lower bounds on the communication performance of Meta-Chaos, because they represent the least and most compact representations of the data to be moved. We also compare communication cost using Meta-Chaos to the communication cost of highly optimized and specialized data parallel libraries, which in these experiments are the Chaos and Multiblock Party libraries.

5.1 Interaction between a structured and an unstructured mesh in one program

One scenario where communication between two different libraries can occur in the same program is when the program performs a sweep through a regular mesh followed by a sweep through an irregular mesh. Both meshes are defined in the same program, but the regular mesh is distributed regularly (using the Multiblock Party library) while the irregular mesh is distributed irregularly (using the Chaos library).

Meta-Chaos is used to perform a copy operation between the regular and the irregular mesh. The schedule generated by Meta-Chaos is used multiple times, twice per time-step, to perform the data copies. All that must be done is to select the proper source and destination for each data copy, so that Meta-Chaos can generate message sends from the source mesh and receives into the destination mesh.

It is also possible to compute the communication schedule by treating the regular mesh generated by Multiblock Party as an irregular mesh. To do that,
Table 1: Schedule build time (total) and data copy time (per iteration) for regular and irregular meshes in one program on IBM SP2, in msec

<table>
<thead>
<tr>
<th></th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Chaos</td>
<td>schedule</td>
</tr>
<tr>
<td></td>
<td>copy</td>
</tr>
<tr>
<td>Meta-Chaos with cooperation</td>
<td>schedule</td>
</tr>
<tr>
<td></td>
<td>copy</td>
</tr>
<tr>
<td>Meta-Chaos with duplication</td>
<td>schedule</td>
</tr>
<tr>
<td></td>
<td>copy</td>
</tr>
</tbody>
</table>

a Chaos-style translation table has to be created to describe the pointwise data distribution. The translation table can be utilized by Chaos to directly compute a communication schedule for moving data between the regular and irregular meshes. However, the correspondence between the points in the regular mesh and the Chaos representation of the mesh must be stored explicitly.

For this experiment, the parallel programming environment is a 16 processor IBM SP2. The data is a two-dimensional array of double precision floating point numbers of size 256x256, regularly distributed by blocks in both dimensions onto the processors, using the Multiblock Parti data distribution routines. The irregular mesh contains 65536 points, stored into a Chaos array irregularly distributed among the processors. The two data parallel libraries are both called from the same program.

There are two different ways to compute a schedule using Meta-Chaos; in Table 1 they are called cooperation and duplication. The terms refer to the way Meta-Chaos computes schedules. For cooperation, Meta-Chaos computes ownership of the source objects (processor, local address, etc.) in the processors running the source program using the functions provided by the source data parallel library. That information is sent to the processors running the destination program, which also compute the corresponding information for the destination objects using the destination parallel library functions and then compute the complete schedule for both the source and destination processors. The computed schedule is then sent to the source processors. On the other hand, when Meta-Chaos computes schedules with duplication, the source and destination processors first exchange data descriptors for both source and destination distributed data structures. This method assumes that, for two separate programs using Meta-Chaos to exchange data, both the source and destination processors know how to interpret the data descriptors (i.e. both sides have the code for both data parallel libraries). With both sets of data descriptors available, the communication schedule can be computed separately in the processors running the source and destination programs.

The cost of the schedule computation for Chaos is dominated by the calls to the Chaos dereference function, which performs the translation from a global (sequential) array index into a processor number and local address. The Meta-Chaos implementation with cooperation also uses the same Chaos dereference function, which is why the schedule computation costs for the two methods are very similar. On the other hand, the Meta-Chaos implementation with duplication must call the Chaos dereference function twice for each array element to be copied, which explains why the cost of building the schedule with that method costs about twice as much as for the other two implementations.

The major difference in the cost of the data copy using Chaos and using Meta-Chaos is that the Chaos implementation internally requires an extra copy of the data and also an extra level of indirect data access. These extra operations are necessary to implement the correspondence between the regular mesh representation of each array element for Multiblock Parti and the pointwise representation of the same element for Chaos. These factors cause the Chaos data copy to usually cost somewhat more than the Meta-Chaos version. However, the actual communication of the data, in terms of the messages generated by all three methods, is essentially identical: all the methods use the same total number of messages and the messages are the same size.

From this experiment, we see several advantages of Meta-Chaos over trying to use a single data parallel library in a manner for which it was not designed:

- smaller memory requirements (Meta-Chaos does not have to explicitly maintain the mapping between the regular mesh representation of an array element and the pointwise representation - that is done implicitly in the linearization),
- ease of use (no extra memory allocation, no explicit mapping between objects in the two data par-
Table 2: Schedule build time (total) and data copy time (per iteration) for two structured meshes in one program on IBM SP2, in msec

<table>
<thead>
<tr>
<th></th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Block Parti</td>
<td></td>
</tr>
<tr>
<td>schedule</td>
<td>19</td>
</tr>
<tr>
<td>copy</td>
<td>467</td>
</tr>
<tr>
<td>Meta Chaos with</td>
<td></td>
</tr>
<tr>
<td>cooperation</td>
<td>29</td>
</tr>
<tr>
<td>schedule</td>
<td>396</td>
</tr>
<tr>
<td>copy</td>
<td></td>
</tr>
<tr>
<td>Meta Chaos with</td>
<td></td>
</tr>
<tr>
<td>duplication</td>
<td>24</td>
</tr>
<tr>
<td>schedule</td>
<td>396</td>
</tr>
<tr>
<td>copy</td>
<td></td>
</tr>
</tbody>
</table>

In each dimension across all the processors. Half the data in each array was involved in the data copy.

As was explained for the previous experiment, the time to compute the schedule with Meta-Chaos using the cooperation method is around twice the time required when using Multiblock Parti. Neither Multiblock Parti nor Meta-Chaos using the duplication method require any communication to build a communication schedule for this experiment. The overhead for building the schedule using Meta-Chaos is a little higher than for Multiblock Parti, which is not surprising since Multiblock Parti is optimized to build schedules for moving regular sections. On the other hand, the Meta-Chaos cooperation implementation requires some communication, since parts of the schedule are not computed on the processors that use it, so those parts must be sent to the right processors. Even though the cost of the communication for this method is not large, it still causes the schedule build to cost more than for the other two methods.

Since the data copy operations for Multiblock Parti and for both Meta-Chaos implementations are exactly the same (they all effectively generate the same schedule), the times for the data copy are essentially the same for all three methods. The only difference is that Meta-Chaos handles data copies within a processor (when parts of the source and destination mesh are on the same processor) more efficiently than does Multiblock Parti. Meta-Chaos performs a direct copy between the storage for the source and destination, while Multiblock Parti requires an intermediate buffer. This is only an issue for the two-processor case, because a large percentage of the data is copied locally, requiring no communication.

These results are encouraging because they show that the more general Meta-Chaos library is able to generate a communication schedule with very little extra overhead compared to generating the same schedule using a special-purpose data parallel library that has been optimized to generate such schedules.

5.3 Client/server program interaction

This experiment presents the results of client/server-style program interaction. The structure of the data on the client and the server is completely managed by Meta-Chaos, meaning that neither needs to know anything about the structure of the data (e.g., whether or how it is distributed across multiple processors) in the other program. From this point of view, Meta-Chaos provides an analogue of a Unix pipe for the programmer to transfer data between the client and server programs.

To illustrate client/server interaction we have chosen a scenario in which the client uses the server as a
high performance computation engine for performing a matrix vector multiply operation.

Meta-Chaos is used to perform the copy operations for the matrix and the vectors, after computing the required communication schedules. These schedules are computed once and stored for reuse as needed.

We have implemented this scenario on an eight-node Digital Alpha cluster of four-processor SMPs, connected via OC-3 links to a Digital ATM Gigaswitch. The client program is a sequential Fortran program, or a parallel Fortran program parallelized using Multiblock Parti. The client program builds the matrix and multiple vectors, then sends data to the server program and receives the result vector. The server program is an HPF matrix-vector multiply program that distributes the matrix and vector across the processors, gets a matrix from the client, then repeatedly gets a vector from the client, computes the result vector and returns it to the client. The client and server are run on disjoint sets of nodes on the Alpha cluster, with each program allocated its own set of up to four nodes (16 processors). Meta-Chaos accesses the ATM switch via FVM, while HPF uses a high performance Digital implementation of the UDP protocol. The experiment has been performed using a 512x512 matrix of double precision floating point numbers.

Figures 5, 6 and 7 show the times to:

- compute the schedules to copy the matrix and the vectors between the client and the server, measured on the client,
- send the matrix from the client to the server, measured on the client,
- perform the matrix-vector multiply on the server, measured on the server, and
- copy both the operand and the result vectors between the client and the server, computed by measuring in the client the total time to send the operand vector, compute in the server, and receive the result vector and subtracting the time spent in the server (from the previous measurement).

The three figures are for varying numbers of client processes, up to four (one per node). In all these experiments, the server is running on four nodes, with up to four processes per node (one per processor).

As is shown in the figures, the best performance is obtained from a server running with eight processes. This is because that configuration achieves the best balance between communication and computation. The time to compute the communication schedules decreases with increasing numbers of server processes, up to four server processes and increases thereafter, because of contention for the ATM network among multiple server processes on the same node. In addition, building the schedules requires an all-to-all communication between the client and server processes, and a relatively small amount of data is sent, so adding more server processes increases the total number of messages required. The same all-to-all communication is required for copying the matrix and the vectors between the client and server. All these factors lead to the performance behavior shown, namely that, beyond eight server processes, the speedup from running the matrix-vector multiply on more server processors is offset by increased communication overhead. In addition, the HPF server program does not speed up beyond eight processors, because of increased internal communication costs in performing the matrix-vector multiply.

A more realistic determination of the benefits that can be achieved from Meta-Chaos requires performing more computation in the server, to amortize the cost of exchanging data between the client and server. Figures 8 and 9 show the results of performing many matrix-vector multiplies using the same matrix. In that case, the communication schedules must only be computed once and the matrix is only sent once from the client to the server. The figures show the time to compute the schedules to copy the matrix and vector between the client and the server, to send the matrix to the server, to perform the matrix-vector multiply operation and to copy both the operand and the result vectors between the client and the server for varying numbers of vectors and server processes. From the results shown in Figure 8, we can compute that a speedup of 4.5 is achieved when the server is an eight-process program, relative to performing the same computation in the client. Figure 9 emphasizes the point that, in many cases, much of the overhead from using a computation server can be amortized over multiple computations.

Figure 10 shows the number of vectors that must be multiplied by the same matrix to amortize the overhead of using a separate server program rather than computing the matrix-vector multiply within the client processes (e.g., with a library routine). From the results shown in Figure 10, we see that a sequential program could benefit greatly from using a parallel server to perform an expensive computation, using Meta-Chaos to do the communication between the programs. In this experiment, the server is not executing a particularly expensive computation, but performance gains are still possible after only a small number of matrix-vector multiply computations are done to amortize the cost of sending the matrix.
Figure 5: Total time for a sequential client. The server runs on four nodes, with up to four processes per node (at most one per processor).

Figure 6: Total time for a two-process client running on two separate nodes. The server runs on four nodes.

Figure 7: Total time for a four-process client running on four separate nodes. The server runs on four nodes.

Figure 8: Total time for twenty vectors for a one-process client. The server runs on four nodes.

6 Related work

The software tool that provides the closest functionality to that of Meta-Chaos is HPF/MPI [6]. The HPF/MPI library extends the MPI point-to-point message passing functions to allow multiple HPF programs to communicate using MPI calls. In the appendix, we present an example of communication between two programs written in HPF, using Meta-Chaos. The example would be written in the same way using the HPF/MPI library, except for some cosmetic changes. However, there are three main differences between Meta-Chaos and HPF/MPI. The first one is that Meta-Chaos can be easily extended to additional data parallel libraries and languages (so long as the libraries/languages provide the required interface functions), while HPF/MPI is restricted to only HPF programs. The second difference is related to the first one, in that HPF/MPI computes a communication schedule using a mechanism similar to the duplication technique used in Meta-Chaos. HPF/MPI does not provide any mechanism equivalent to the Meta-Chaos cooperation technique for computing a schedule. The cooperation technique is important when it is not possible (for performance reasons) to exchange entire data descriptors between the source and destination of a data transfer. For example, computing a communication schedule to transfer array elements irregularly distributed using the Chaos library using the duplication method would be very expensive. The third major difference in functionality between Meta-Chaos and HPF/MPI is that Meta-Chaos allows multiple data parallel libraries to communicate with one another in the same program. Overall, Meta-Chaos
provide a superset of the functionality of HPF/MPI.

As HPF/MPI and other previous work has shown, integrating task and data parallelism can present significant advantages in both performance and ease of programming. Several research efforts have been working on the enhancement of data parallel languages to integrate data parallelism and task parallelism. Fx [19] adds compiler directives to HPF to specify task parallelism. Opus [11] is a set of HPF extensions that provides a mechanism for communication and synchronization through a shared data abstraction (SDA). Fortran M [4] extends Fortran77 for task parallel computations, and also introduces several data distribution statements. Braid [6] introduces data parallel extensions to the Mentat [9] distributed object programming language. Integrating task and data parallelism within one language is an active area of research, but does not address the problem that there are many existing parallel codes that have been written using different data parallel libraries. The existence of many such data parallel libraries [8, 10, 12, 14, 16], and their related applications, provides strong support for the claim that no single library or language is sufficient for all potential applications. In addition, asking application programmers to write new codes, or rewrite existing codes, using only one data parallel language or library just so they can inter-operate is unlikely to succeed. Therefore a tool like Meta-Chaos is needed to allow existing parallel applications to inter-operate and also to allow new applications, written using whatever data parallel library or language the application programmer feels is appropriate, to inter-operate.

All the described languages and systems are designed to allow communication between separate programs or tasks containing data structures managed by one data parallel library. None of the systems allows communication between data structures managed by multiple data parallel libraries, either between separate (data parallel) programs or within a single program. Our system design, as implemented in Meta-Chaos, provides this capability.

7 Conclusions and Future Plans

In this paper we have addressed the problem of interoperability between different data parallel libraries. With the mechanisms we have described, multiple libraries can exchange data in the same data parallel program or between separate data-parallel programs.

We have implemented the interoperability mechanism in a library called Meta-Chaos and have explored the behavior of the approach for several programs on two different parallel architectures. Our experimental results show that our framework-based approach can be implemented efficiently, with Meta-Chaos exhibiting low overheads, even compared to the communication mechanisms used in two specialized and optimized data parallel libraries. In addition, we exhibited the flexibility of the approach, and good performance, for applications using a client/server execution model.

We plan to make Meta-Chaos publicly available and encourage developers of data parallel libraries to provide the interface functions needed for Meta-Chaos.
to access data distributed using those libraries. We plan to apply the framework-based approach to new application areas, and are currently studying ways to incorporate distributed data parallel objects into the CORBA [16] object model, so that data parallel programs could interoperate with distributed object systems. Meta-Chaos could be used as the underlying mechanism for such an extension.

References


A HPF Meta-Chaos example

Figure 11 illustrates the sequence of calls required to allow two HPF programs to exchange data using Meta-Chaos.

Two programs are shown in Figure 11, a source program and a destination program. Each owns one HPF distributed array, and the programs use Meta-Chaos to copy an array subsection from the source to the destination. The program performs the following operation, in Fortran00 array syntax:


In the HPF programs, the user defines the source and destination array sections with the CreateRegionHPF interface function provided by the implementor of the HPF runtime library interface functions. The Meta-Chaos functions MSGestSetOfRegion and MSGestAddRegion2Set are used to create the SetOfRegions for the source and
destination. Building the communication schedule is then performed by the call to the Meta-Chaos MC_ComputeSched function, which is a collective operation across the processors of both the source and destination program. The Meta-Chaos MC_DataMove calls then perform the communication between the source and destination, using the schedule. This is also a collective operation across both the source and destination, with the source program using a call for sending data and the destination program using a call for receiving data.
Konferenzen des Forschungszentrums Jülich

1 Technik und Gesellschaft
Vorträge eines Seminars veranstaltet von der
Programmgruppe Mensch, Umwelt, Technik
des Forschungszentrums Jülich GmbH
W. Fischer, L. Hennen, W. Kirstein, G. Schleser, G. Stein, Ch. Wandrey,
P.M. Wiedemann (1990)

2 Analytische Glimmentladungs-Spektroskopie
herausgegeben von J. Rottmann (1990)

3 Die digitale Verarbeitung analoger Signale in Theorie und Praxis
KFA-Fortbildungsseminar; Jülich, im April 1989

4 Reduced Enrichment for Research and Test Reactors
Proceedings of the XIIth International Meeting
Berlin, 10.–14. September 1990
Arranged and edited by G. Thamm, M. Brandt (1991)

5 Kontrolle radioaktiver Abfälle
Erfahrungen mit der BMU-Richtlinie
Seminar; Jülich, 23.–24. Oktober 1990
Redaktion M. Laser (1991)

6 Intermetallische Phasen als Strukturwerkstoffe für hohe Temperaturen
Seminar; Hagen, 30.–31. Oktober 1990
herausgegeben von F.J. Bremer (1991)

7 Mikrobruchvorgänge in Al₂O₃-Keramik
DFG-Kolloquium; Jülich, 9.–10. April 1990

8 BMFT-Workshop Klimawirkungsforschung
Auswirkungen von Klimaveränderungen
Tagungsband; Bonn, 11.–12. Oktober 1990
herausgegeben von W. Fischer, G. Stein (1991)

9 5th International Symposium on
Laser-Aided Plasma Diagnostics

10 A Regime to Control Greenhouse Gases:
issues of Verification, Monitoring, Institutions
Proceedings of a Workshop, Bad Neuenahr, June 12–14, 1991
edited by J.C. di Primio, G. Stein (1991)

11 Hartstoffe in Werkzeugen
Redaktion: H. Prasse (1992)
12 Hadronic Processes at Small Angles in Storage Rings
105th International WE-Heraeus-Seminar, Proceedings
Bad Honnef, February 1–3, 1993
edited by E. Rössle, O.W.B. Schult (1993)

13 Economics of the Greenhouse Effect
Modeling Strategies and Impacts
Workshop: Bad Zwischenahn, January 21–22, 1993

14 Greenhouse Gas Verification — Why, How and How Much?
Proceedings of a Workshop, Bonn, April 28–29, 1994

15 Advances in Systems Analysis: Modelling Energy-Related Emissions
on a National and Global Level
edited by J.-Fr. Hake, M. Kleemann, W. Kuckshinrichs, D. Martinsen, M. Walbeck
(1994)

16 Wasser: Nachhaltige Gewinnung und Verwendung eines lebenswichtigen Rohstoffs

17 Die digitale Verarbeitung analoger Signale in Theorie und Praxis
KFA-Fortbildungsseminar; Jülich, im Juni 1996

18 Partielle Differentialgleichungen, Numerik und Anwendungen
Manuskripte der Vorlesungen der Sommerschule vom 2. bis 6. September 1996
im Forschungszentrum Jülich

19 Gesundheitsbelastungen
im Forschungszentrum Jülich
herausgegeben von P. M. Wiedemann, H. M. Seitl (1996)

20 Energieforschung
Manuskripte der Vorlesungen des 2. Jülicher Ferienkurses „Energieforschung"
vom 22. September bis 2. Oktober 1996 im Forschungszentrum Jülich
herausgegeben von J.-Fr. Hake, W. Kuckshinrichs, K. Kugeler,
W. Pfaffenerberger, H.-J. Wagner (1996)

Teil I: Energiebedarf und Energieversorgung; Energietechnik; Energieverteilung;
Energiewirtschaft; Energie und Umwelt

Teil II: Systemanalyse, Energiesysteme; Sicherheit und Risiko von Energieanlagen;
Energiesituation in ausgewählten Ländern; Energieforschung und Energiepolitik

21 Proceedings of the 6. Workshop
Compilers for Parallel Computers
Manuscripts of the papers presented at the 6. Workshop held in Aachen, Germany,
from 11 – 13 December 1996
edited by M. Gerndt (1996)