FORSCHUNGSZENTRUM JULICH GmbH
Zentralinstitut fur Angewandte Mathematik
D-52425 Julich, Tel. (02461) 61-6402

Interner Bericht

SVM Support in the Vienna Fortran Compiling System

Peter Brezany*, Michael Gerndt, Viera Sipkova*

KFA-ZAM-IB-9401

Januar 1994
(Stand 27.01.94)

(*) Department of Statistics and Computer Science, University of Vienna

SVM Support in the Vienna Fortran Compilation System *

Peter Brezany
University of Vienna
brezany@par.univie.ac.at

Michael Gerndt
Research Centre Jülich(KFA)
m.gerndt@kfa-juelich.de

Viera Sipkova
University of Vienna
sipka@par.univie.ac.at

January 27, 1994

Abstract

Vienna Fortran, a machine-independent language extension to Fortran which allows the user to write programs for distributed-memory systems using global addresses, provides the forall-loop construct for specifying irregular computations that do not cause inter-iteration dependences. Compilers for distributed-memory systems generate code that is based on runtime analysis techniques and is only efficient if, in addition, aggressive compile-time optimizations are applied. Since these optimizations are difficult to perform we propose to generate shared virtual memory code instead that can benefit from appropriate operating system or hardware support. This paper presents the shared virtual memory code generation, compares both approaches and gives first performance results.

Keywords: distributed-memory systems, shared virtual memory systems, compile-time optimization, runtime analysis, Fortran language extensions

1 Introduction

Massively parallel computers (MPP) offer an immense peak performance. Current architectures consist of hundreds of nodes with physically distributed memory and are either pure distributed memory systems (Paragon), hardware-supported shared memory systems (KSR,Cray T3D), or software-based shared virtual memory machines (IVY, iPSC2/Koan).

Shared virtual memory (SVM) machines provide a global address space on top of physically distributed memory via an extension of the virtual memory paging mechanism [5]. Page faults on one processor are served by another processor which currently has the page of the global address space in its private memory. On such systems message passing and shared memory are both available for data exchange among processors.

The two main challenging problems of MPP systems are to increase the sustained performance of real applications on such systems by hardware and software enhancements and to develop programming models and tools making programming of these machines easier.

Recently, High Performance Fortran (HPF) was developed based on Vienna Fortran [20] and Fortran D [13] with the goal to ease programming MPP systems for data parallel applications. The current version supports regular grid-based applications via data partitioning annotations. Algorithms for the compilation of such a programming model have been developed and implemented in some research tools and products, e.g. Superb [4], Fortran D environment [15], xHPF [1].

Current research is going into an extension of HPF supporting irregular computations. Vienna Fortran and Fortran D provide language constructs, such as irregular distribution formats and parallel loops with work distribution specification. Both projects are working on the message passing implementation of these constructs. The implementation has to perform aggressive compile-time optimizations to reduce the storage and runtime overhead. These optimizations are difficult to perform for applications where the problem structure depends on runtime data or is dynamic.

The goal of the work described in this paper is to study the compilation of Vienna Fortran for SVM machines exploiting the strength of both concepts, message passing and shared memory communication. Language extensions for regular grid-based applications are still translated into message passing code

*The work described in this paper was carried out as part of the ESPRIT 3 project "Performance-Critical Applications of Parallel Architectures (APPARC)", the P2702 ESPRIT research project "An Automatic Parallelization System for Genesis" funded by the Austrian Ministry for Science and Research (BMWF), and the research project "High-level Programming Support for Parallel Systems" funded by the Austrian Science Foundation (FWF).

since the compiler can generate efficient code based on compile-time information. The compiler is able to translate language features for irregular computations into both, message passing code and shared virtual memory code. Shared memory code generation facilitates compile-time optimizations, implicitly exploits locality due to page migration among processors, and can use runtime optimizations such as data prefetching. Our long term goal is to find criteria for the selection of the better code generation strategy for a given input code fragment.

Section 2.1 presents the Vienna Fortran language extensions. The basic message passing compilation scheme is introduced and evaluated in Section 2.2. In Section 2.3 we present the basic SVM compilation and Section 3 describes the algorithm used to optimize the interface among message passing and SVM code generation. Section 4 gives some first performance comparisons.

2 Code Generation for Irregular Data-Parallel Loops

2.1 Forall Loops

Vienna Fortran supports explicitly parallel loops called **FORALL**[1]-loops which are mainly used for specification of irregular computations. A precondition for the correctness of this loop is the absence of loop-carried dependences except accumulation type dependences.
The general form of the forall-loop is as follows:

FORALL $(I_1 = s\vec{e}c_1, \ldots, I_n = s\vec{e}c_n)$ [*work_distr_spec*]
forall-body
END FORALL

where for $1 \leq j \leq n$, I_j are index names and $s\vec{e}c_j$ are subscript triplets of the form l_j, u_j, s_j which must not contain references to any index name. If the value of s_j equals one, s_j can be omitted. If j equals one, the header parenthesis can also be left out. *work_distr_spec* denotes an optional work distribution specification.

The iterations of the loop may be assigned explicitly to processors by work distribution annotations, i.e. an optional *on-clause* specified in the forall-loop header.

[1] Note that the *forall-loop*, as introduced here, is not the forall-loop proposed during the development of Fortran 90 and in HPF.

```
FORALL i = 1,N ON OWNER(C(K(i)))
      Y(K(i))=X(K(i))+C(K(i))*Z(K(i))
END FORALL
```

(a) Parallel loop with indirect accesses

```
FORALL i = 1, N ON P1D(NOP(i))
     REAL T
        ...
        REDUCE( ADD, X, A(Y(i)))
        ...
END FORALL
```

(b) Summing values of a distributed array

```
FORALL i = 1, N
        ...
   REDUCE( ADD, B(X(i)), D(Y(i))*E(Z(i)))
        ...
END FORALL
```

(c) Accumulating values onto a distributed array

Figure 1: Parallel loops.

The *on-clause* in the example shown in Figure 1 (a) specifies that the i-th iteration ($1 \leq i \leq N$) of the loop is executed on the processor which owns *C(K(i))*. In the second parallel loop (Figure 1(b)) the *on-clause* directly refers to the processor array *P1D*. The ith iteration is assigned to the kth processor where k is the current value of the array element which is denoted by *NOP(i)*. T is declared private in this forall loop. Logically, there are N copies of the variable T, one for each iteration of the loop. Assignments to private variables do not cause loop-carried dependences.

A reduction statement may be used within forall-loops to perform global operations, such as global sums; the result is not available until the end of the loop. The forall-loop in Figure 1(b) computes the sum of some of the array elements of A and places the result in variable X. In each iteration of the forall-loop in Figure 1(c), elements of D and E are multiplied, and the result is used to increment the corresponding element of B. In general, all arrays B, D, E, X, Y, and Z can be distributed. In this example the user presumes the compiler will automatically derive the work distribution.

Moreover, the language enables the user to specify *Block* or *Cyclic* work distribution strategy.

2.2 Message Passing Code Generation

The strategy used generates two code phases for forall-loops, known as the **inspector** and the **executor** [16, 18]. They both are supported by the runtime libraries [11, 12] [2]. The inspector analyzes the communication patterns of the loop, computes the description of the communication, and derives translation functions between global and local accesses, while the executor performs the actual communication and executes the loop iterations.

The inspector consists of three steps. The *first step* performs the construction of translation tables for irregularly distributed arrays[3] occurring in a forall-loop. The translation table is constructed from the mapping array, recording the owner of each datum and its local index. To economize storage, one translation table is constructed for all arrays with the same shape and the same distribution.

The *second step* computes the *execution set exec(p)*, i.e. the set of loop iterations to be executed on processor p . The work distribution specification, if introduced, determines the execution set. For the example in Figure 1(a), the execution set is given by: $exec(p) = \{i \mid K(i) \in local^C(p) \wedge 1 \leq i \leq N\}$ where $local^C(p)$ denotes the set of indices of elements of array C that are stored on processor p.

The *third step* performs a dynamic loop analysis. Its task is to describe necessary communication by a set of so called *schedules* that control runtime procedures moving data between processors in the subsequent executor phase. The dynamic loop analysis also establishes an appropriate addressing scheme to access local elements and copies of non-local elements on each processor.

Information needed to generate a schedule, to allocate a buffer, and for the global to local index conversion for a given distributed array reference *A(f(i))* can be produced from the *global reference* list $global\_ref_A(p)$, along with the knowledge of the distribution of A. The global reference list is computed from the subscript function *f(i)* and from *exec(p)*.

To build $global\_ref_A(p)$, the forall-body is partially interpreted. The *slicing* ([19]) of the forall-body reduces the code duplication to the level needed to evaluate $f(i)$. Let V be the set of all variable references occuring in $f(i)$. The slice of the forall-body with respect to statement S and the variable references in V consists of all statements and predicates of the forall-body that might affect the values of elements of V in statement S. However, some temporary variables must be allocated to avoid overwriting live data. After introducing these temporary variables, the reference $A(f(i))$, originally occuring in S, is transformed into $A(f'(i))$ in the slice. $f'(i)$ results from $f(i)$ by substituting some original variables with temporary variables. Figure 2 shows a scheme for computing $global\_ref_A(p)$. In this figure, the function *value* computes the value of $f'(i)$, **nil** denotes the empty list, and **append** appends an item to the end of the list.

$global\_ref_A(p)$ = **nil**
foreach $i \in exec(p)$ **do**
  ··· slice execution ···
  $global\_ref_A(p)$ = $global\_ref_A(p)$ **append** $value(f'(i))$
end foreach

Figure 2: Computation of $global\_ref_A(p)$.

The list $global\_ref_A(p)$ and the distribution of A are used to determine the communication pattern, i.e. a *schedule*, the size of the communication buffer to be allocated, and the local reference list $local\_ref_A(p)$ which contains results of global to local index conversion. The declaration of array A in the message passing code allocates memory for the local segment holding the local data of A and the communication buffer storing copies of non-local data. The buffer is appended to the local segment. An element from $local\_ref_A(p)$ refers either to the local segment of A or to the buffer. A local index in $local\_ref_A(p)$ is derived from the corresponding element of $global\_ref_A(p)$.

The executor is the second phase in the execution of the forall-loop; it performs communication described by schedulers, and performs the actual computations for all iterations from *exec(p)*. In general, no synchronization at the beginning and the end of the loop is needed. The schedulers control communication in such a way that execution of the loop using the local reference list accesses the correct data in local segments and buffers. Non-local data needed for the computation on processor p is gathered from other processors by the runtime communication procedure *Gather*. It accepts a schedule, an address of the local segment, and an address of the buffer as input parameters. On the other hand, the non-local data computed on processor p are moved by the procedure *Scatter* into their permanent addresses.

Several experiments demonstrated that the above techniques can be used for dealing effectively with

[2] The Parti library [12] has been the first system constructed to support the handling of irregular computations. On top of the Parti an extended library [11] is being built to support all Vienna Fortran and HPF distributions and alignments.

[3] In this case, array elements are arbitrarily assigned to processors as it is specified by a mapping array.

some typical irregular computations. However, some difficulties can occur when attempting to adapt an irregular code for a distributed-memory system.

In the case, when an indirect indexed array A occuring in the forall loop is aligned with an array B such that array element $A(i)$ is aligned to array element $B(k * i + o)$ and array B is distributed using a $CYCLIC(M)$ distribution, the global to local index conversion is very costly. It has to be computed element-wise, using techniques introduced in [10], for example. This computation can be also done via a table. However, the translation table generated for an array with n elements contains $2 * n$ items and, therefore, it is distributed as well and communication is potentially needed to get some information from the table.

Extents of execution sets, lengths of global reference and local reference sets and sizes of communication buffers are unknown at compile time. In the standard Fortran 77 environment, shapes of arrays that implement the above concepts must be overestimated to guarantee the safe computation as much as possible. Therefore processors can quickly run out of memory space.

Generation of schedules can also cause high runtime overhead. There are a variety of situations in which the same data need to be accessed by multiple loops. The runtime support makes it possible to reuse non-local data copies by generating incremental schedules that include only non-local data accesses not included in the preexisting schedules. The program analysis has to determine when it is safe to use incremental schedules and reuse the schedules generated. The dataflow framework described in [14] aims at providing information about which data are needed at which points in the code, along with information about which live non-local data are available. However, it only considers the intraprocedural level and it works under many simplified assumptions. A simple conservative runtime method is proposed in [17] that allows to reuse the results from inspectors in many cases. In this method a record is maintained whether a distributed array is modified. In such a case, values have to be exchanged again. When using this method, the compiler generates code that at runtime maintains a record of when a forall-loop may have written to a distributed array. However, there is a high runtime overhead in some cases.

2.3 SVM Code Generation

Message passing code generated for forall-loops with irregular accesses has to compute remote accesses precisely at runtime. In SVM systems instead, data movement is implicit when a process references a datum in the global address space also called *shared memory*. The Vienna Fortran Compilation System therefore translates forall-loops for SVM systems in a different way. Distributed arrays accessed in the forall loop are copied to the global address space such that references are resolved automatically. When referencing a variable that is not available in the physical memory of the processor, the appropriate page or cache line is automatically transferred to that processor.

Phase 1: Work Distribution
Compute *exec(p)*.
Phase 2: Array Globalization
Copy the local part of distributed arrays read or written in the forall-loop to the shared memory.
Phase 3: Execute Local Iterations
Execute the iterations in exec(p).
Phase 4: Array Localization
Copy local parts of modified arrays back to process private memory.

Figure 3: SVM Code

The code generated for individual forall-loops is outlined in Figure 3. The first phase computes the execution set according to the specified work distribution. This phase is implemented in the same way as in the message passing version. The second phase copies the distributed arrays accessed in the forall-loops to the global address space (array globalization). The third phase executes the iterations in the local execution set, and in the last phase the modified arrays are copied back to the process's private memories (array localization). Global synchronizations have to be performed after phase 2, to ensure that no process accesses a datum in the global address space before it is initialized, and prior to phase 4, to ensure that all assignments have been performed before data are copied back. The forall-loop is thus translated in a classical parallel loop with a pre-phase of globalize operations and a following phase of localize operations.

The amount of shared memory needed for distributed arrays accessed in the forall-loops of a unit is dynamically allocated at the start of the program unit and deallocated at the end. For each such distributed array a new array is declared with the origi-

nal shape. The new arrays are allocated dynamically in the shared memory segment (cf. Section 3.4).

3 SVM Code Optimization

The main drawback of this code generation scheme is that the globalize and localize operations are performed for each forall-loop although in a sequence of forall-loops only the first has to globalize an array and the last loop has to localize it again. In addition, if no globalization has to be performed the synchronization before the loop can be eliminated.

Therefore, we designed and implemented an optimization that eliminates redundant globalize and localize operations. The optimization is based on the assumption that globalize and localize operations can only be performed at the start and end of the unit and before and after each forall-loop. We do not allow such operations to be inserted arbitrarily.

The overall algorithm consists of a top-down and a bottom-up phase that are repeated until no changes occur. Each phase solves a dataflow problem which depends on the results of the previous phase. The top-down problem is called the *reaching distributed arrays problem* (REDAP) and the bottom-up problem is the *required distributed arrays problem* (RQDAP). The result of this optimization is the information which arrays have to be globalized at the start of the unit, which arrays have to be localized at the end, and which arrays have to be globalized or localized for individual forall-loops.

Both problems, REDAP and RQDAP, can be described using the following notation:

- G($\mathcal{N}$,$\mathcal{E}$,entry,exit) is the program graph where $\mathcal{N}$ is the set of nodes representing the statements, $\mathcal{E}$ the set of arcs, *entry* the first statement, and *exit* the last statement. In the program graph entire forall-loops are condensed to a single node.
- $\mathcal{A}$ is the set of arrays distributed via language annotations and accessed in a forall-loop.
- $\mathcal{L}$=$\mathcal{P}$($\mathcal{A}$) is the data flow information set.
- SH: $\mathcal{N} \rightarrow \mathcal{L}$ is a function that defines for each forall-loop the set of accessed arrays from $\mathcal{A}$. These arrays are shared in the SVM implementation.
- TOPDOWN, BOTTOMUP is the direction of the data flow problem, which defines whether the effect of joining paths at a node is the union of the predecessors or the successors.

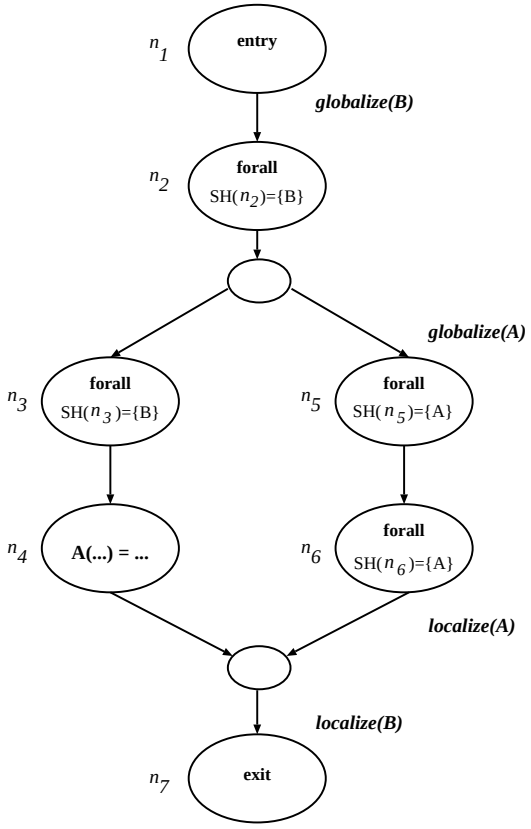

Figure 4: Program Graph

- $\mathcal{F}$ is the set of functions on $\mathcal{L}$ that includes for each node n a function f_n: $\mathcal{L} \rightarrow \mathcal{L}$ describing the effect of that node on the data flow information.

The example shown in Figure 4 illustrates the definitions. The localization of B after n_2 and its globalization prior to n_3 resulting from the basic implementation are redundant. The same is true for A between n_5 and n_6. The assignment to A in node n_4 requires that A is distributed when the statement is executed. The final localize and globalize operations resulting from our optimization are presented in the figure. After the following description of the optimization algorithms we explain how this final result is obtained.

3.1 Reaching Distributed Arrays

The REDAP is the problem of determining, for each node, all arrays in $\mathcal{A}$ that may be distributed when the statement is executed. The solution is described by a function RE: $\mathcal{N} \rightarrow \mathcal{L}$. REDAP is a top-down problem.

$$(a)\quad f_n(X) := \begin{cases} (X \setminus SH(n)) \cup \{A \in \mathcal{A} | A \textit{ is localized}\} & n \textit{ is forall} \\ \{A \in \mathcal{A} | A \textit{ is not globalized}\} & n \textit{ is entry} \\ X & \textit{otherwise} \end{cases}$$

$$(b)\quad f_n(X) := \begin{cases} (X \setminus SH(n)) \cup \{A \in \mathcal{A} | A \textit{ is globalized}\} & n \textit{ is forall} \\ \{A \in \mathcal{A} | A \textit{ is not localized}\} & n \textit{ is exit} \\ X \cup \{A \in \mathcal{A} | A \textit{ is accessed in } n\} & \textit{otherwise} \end{cases}$$

Figure 5: Propagation Functions of REDAP and RQDAP

The propagation functions in $\mathcal{F}_{RE}$ are defined in Figure 5(a). An array accessed in a forall is shared after the loop if it is not explicitly localized after the forall, independent of whether it was distributed prior to the forall. An array is distributed after the entry statement if it is not explicitly globalized by the entry statement since we require that arrays in $\mathcal{A}$ are distributed at start of the unit.

3.2 Required Distributed Arrays

The RQDAP is the problem of determining, for each node, all arrays that have to be distributed after the statement was executed. The solution is described by the function RQ: $\mathcal{N} \rightarrow \mathcal{L}$. RQDAP is a bottom-up problem.

The propagation functions in $\mathcal{F}_{RQ}$ are defined in Figure 5(b). If an array is globalized by a forall this array has to be distributed whenever the forall is executed. An array that is accessed in a forall but not globalized need not be distributed when the forall is executed independent of whether it has to be distributed after the forall.

An array has to be distributed when the exit node is executed if it is not localized by the exit node because all arrays have to be distributed when the unit is finished.

Each array accessed in another statement, i.e. outside of forall-loops, has to be distributed when the statement is executed.

3.3 Optimization Algorithm

Both data flow problems are solved with the basic iterative algorithm shown in Figure 6.
The calls

find_solution(G,$\mathcal{F}_{RE}$,RE,TOPDOWN,entry)
find_solution(G,$\mathcal{F}_{RQ}$,RQ,BOTTOMUP,exit)

```
find_solution(G,F,x,direction,s)
stable=false
while (not stable) do
  stable=true
  foreach n ∈ N\{s} do
    if (direction=TOPDOWN) then
        new = ⋃_{n'∈ pred(n)} f_{n'} (x(n'))
    else
        new = ⋃_{n'∈ succ(n)} f_{n'} (x(n'))
    endif
    if (new≠x(n)) then
      x(n)=new
      stable=false
    endif
  end foreach
end while
end
```

Figure 6: Iterative Solver for REDAP and RQDAP

with the initializations

$$\forall n \in N \;\; RE(n) = RQ(n) = \emptyset$$

solve REDAP and RQDAP respectively.

The optimized interface among message passing code generation and SVM code generation can now be described as the pair of functions RE and RQ. The code generation will then generate localize and globalize operations according to the following rules:

entry statement

$$\forall A \in \mathcal{A} \setminus RQ(entry) \;\; globalize(A)$$

exit statement

$$\forall A \in \mathcal{A} \setminus RE(exit)\ \ localize(A)$$

forall statement

$$\forall A \in SH(forall) \cap RE(forall)\ \ globalize(A)$$

$$\forall A \in SH(forall) \cap RQ(forall)\ \ localize(A)$$

The computation of the optimized interface is performed by the procedure shown in Figure 7.

```
compute_interface(G,RE,RQ)
∀ n ∈ N   RE(n) = RQ(n) = ∅
stable=false
while (not stable) do
  stable=true
  find_solution(G, F_RQ(RE),RQ,BOTTOMUP,exit)
  x=RE
  find_solution(G, F_RE(RQ),x,TOPDOWN,entry)
  if (x≠RE) then
    RE=x
    stable=false
  endif
end while
end
```

Figure 7: Interface Optimizer

The algorithm iteratively solves RQDAP and REDAP until RE is no longer changed. In each iteration the current solution, RE and RQ, determines the propagation functions of the nodes in the next problem according to the rules described above.

For example, if a statement following a forall-loop requires that an array is distributed and that array is accessed in the forall loop and thus is copied to shared memory, a localization has to be performed after the loop. The solution of RQDAP determines that the localization has to be done and this localize operation has to be taken into account when REDAP is solved afterwards.

Therefore, the propagation functions of REDAP and RQDAP have to be parameterized according to the current solution RE and RQ. The parameterized propagation functions are defined in Figure 8.

The algorithm starts with the assumption that all arrays accessed in any forall-loop are globalized at the beginning and keeps them shared as long as possible.

We use the example program graph in Figure 4 to illustrate the algorithm. First, RQDAP is solved. Here the only requirement is that A has to be distributed in n_4. This is propagated in the program graph. The solution of RQDAP is: RQ(n_1)= RQ(n_2)= RQ(n_3)={A} and RQ(n_4)= RQ(n_5)= RQ(n_6)= RQ(n_7)= $\emptyset$.

Now, REDAP is solved taking RQ into account. Since RQ(n_1)={A} A is not globalized by the entry statement. The solution of this step is RE(n_2)= RE(n_3)= RE(n_4)= RE(n_5)= RE(n_7)={A}. Since both solutions of REDAP and RQDAP were changed another iteration has to be performed. This iteration is necessary because the current solution is incorrect. The exit statement will not localize A since RE(exit)={A}. But A is only distributed if the control flow goes through n_3 and not through n_5. The next computation of RQDAP now propagates the information that A has to be distributed to n_6. In the final solution RE is unchanged and RQ(n_5)= $\emptyset$ and for all other nodes RQ(n)= {A}.

The convergence of the interface optimization algorithm results from the proof in [9] for monotone data flow systems and the following properties of the algorithm:

- ($\mathcal{L}$,$\cup$) is a bounded semilattice,
- the propagation functions of the parameterized problems REDAP and RQDAP are distributive functions on $\mathcal{L}$, and
- $\forall n \in N, x \in \mathcal{L}: \quad f_n(x) \leq f'_n(x)$

  where either

  $$f_n \in \mathcal{F}_{RQ}^{next}, f'_n \in \mathcal{F}_{RQ}$$

  or

  $$f_n \in \mathcal{F}_{RE}^{next}, f'_n \in \mathcal{F}_{RE}$$

  and

  $\mathcal{F}^{next}$ is the set of propagation functions in the next iteration of the while-loop in compute_interface.

Due to these properties the solutions RE(n) and RQ(n) for each node n in each iteration in compute_interface include at least those arrays that were included in the previous iteration. Since the number of arrays in $\mathcal{A}$ is fixed the algorithm will terminate.

In the basic interface (Figure 3) only the modified shared arrays are localized after a forall-loop. A similar optimization can also be applied to the optimized

$$\forall f_n \in \mathcal{F}_{RE}(RQ) \quad f_n(X) := \begin{cases} (X \setminus SH(n)) \cup RQ(n) & n \text{ is } forall \\ RQ(n) & n \text{ is } entry \\ X & otherwise \end{cases}$$

$$\forall f_n \in \mathcal{F}_{RQ}(RE) \quad f_n(X) := \begin{cases} (X \setminus SH(n)) \cup RE(n) & n \text{ is } forall \\ RE(n) & n \text{ is } exit \\ X \cup \{A \in \mathcal{A} | A \text{ is accessed in } n\} & otherwise \end{cases}$$

Figure 8: Parameterized Propagation Functions

interface. The exit and forall statements have to localize only those arrays that were modified since the globalize operation. These arrays can also be determined by a data flow problem, the *Modified Shared Arrays Problem (MSAP)*. MSAP is the problem of determining, for each node, which arrays are shared and may have been modified. MSAP is a TOPDOWN problem and the node propagation function of forall-loops adds shared arrays modified in that loop to the current solution and excludes shared arrays localized after the forall-loop. The code generation rules for the exit statement and forall statements can now be modified according to the solution of MSAP.

3.4 Implementation

In the Vienna Fortran Compilation System data flow problems are expressed in terms of monotone data flow systems [9] and solved by a parameterized tool that implements the basic iterative algorithm. The tool is based on a bitvector representation of the data flow information set and is parameterized by the propagation function and the direction of the specific problem.

Since REDAP and RQDAP can also be formulated as monotone data flow systems the existing tool was applied to solve both problems. The solutions RE and RQ are expressed by bitvectors and the propagation functions implemented as efficient bitvector operations.

The shared memory implementation is based on Unix System V shared segments. At start of a subroutine a shared segment is allocated and each process maps the segment in its virtual address space. The copies of distributed arrays are dynamically allocated in the shared address space.

We use two different strategies to implement dynamic arrays in the generated source code. On Paragon, pointer-based variables known from Cray Fortran are supported by the Fortran compiler. The code generated for a distributed array A(1:N) is shown in Figure 9.

```
POINTER (PA,SA)  ! Declaration of the shared
REAL SA(1:N)     ! array
 . . .
PA=SALLOC(N)     ! SALLOC returns the start
                 ! address of SA in the
 . . .           ! shared address space
DO i=$l,$r       ! globalize(A)
 SA(i)=A(i-$lb+1)
ENDDO
```

Figure 9: Code Generation for Shared Arrays

On the iPSC/2-Koan implementation dynamic arrays are implemented by computing the offset of a statically declared copy and the start address in the global address space. Accesses to the shared copy are linearized and the offset is added such that memory locations in the shared address space are accessed.

The globalize and localize operations are implemented as shown in Figure 9. Each process copies its own part of the distributed array to the shared copy and vice versa. Therefore, if the computation in the forall-loop does provide locality most of the referenced pages will be local.

Reduction operations in forall-loops are implemented via a distributed algorithm if replicated variables are computed. Each process computes the reduction for its local iterations and the results are combined by the appropriate global message passing operations. Reductions, computing array elements of shared arrays, are implemented via appropriate synchronization. On the Koan implementation we use page locking and on the Paragon critical sections.

4 Comparison

The following example illustrates the properties of both code generation techniques. We executed the message passing and the SVM code on the Koan system, an SVM implementation on an iPSC2 at IRISA in France.

```
      INTEGER a(40000)  DIST (BLOCK)
      INTEGER b(40000)  DIST (BLOCK)
      INTEGER id(40000) DIST (BLOCK)
      ...
      DO 100 iter=1,50
       ...
        FORALL 20 i=2,20000-1
          a(2*id(i))=b(2*id(i)-1)+b(2*id(i)+1)
20      CONTINUE
100   CONTINUE
      ...
      END
```

Table 1 gives the execution times of the outermost do-loop for three different program versions. In the SVM code the globalize and localize operations are executed at program start and at the end of the program. In the non-optimized message passing version the inspector is executed for each iteration of the outer do-loop. Due to the memory requirements of the Parti implementation the code could only be executed on 16 processors. The performance of the manually optimized message passing version is much better since the inspector is executed only once. The one-node version in all three cases is the sequential code.

Although the example gives an idea of the relation between both code generation techniques, a comparison is difficult since the performance depends on a large set of parameters, such as the page fault times, the page size, the message passing performance, the locality with respect to the pages, the size of the arrays etc. Therefore, both code generation techniques have to be carefully studied for real applications taking into account new compile-time and runtime optimizations.

nodes	SVM	non-opt. MP	opt. MP
1	4.8	4.8	4.8
2	6.2	*	16.3
4	3.5	*	7.0
8	2.3	*	3.2
16	1.6	50.2	1.5

Table 1: Time (in secs) for loop 100

5 Conclusion

When compiling irregular code written in Vienna Fortran for a distributed memory system that provides somehow a global address space, two code generation strategies are available: generation of message passing code based on the runtime analysis and generation of SVM code. We have discussed both compiling techniques and presented first performance results.

In the future, we plan to optimize the SVM-based code generation by exploiting advanced runtime features, such as data prefetching and weak coherence. In addition, we will extend the intraprocedural interface optimization towards an interprocedural optimization to efficiently support applications where distributed arrays are accessed in multiple subroutines only in forall-loops.

We plan to evaluate the techniques in the near future on the SVM implementation on Paragon, which is based on OSF/1 shared segments, and on KSR. On Paragon, two SVM implementations will be available, the Intel-supported kernel implementation and an External Mach Server implementation developed at the Technical University of München. Both support the shared segment interface.

Currently, no similar approach towards a combination of message passing code generation and SVM code generation for data-parallel languages is known to the authors. Research related to the message passing code generation, especially its optimization, was already discussed in Section 2.2.

References

[1] Applied Parallel Research, *FORGE 90, Version 8.0, User's Guide*, 1992

[2] G. Fox, S. Hiranadani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, M. Wu, *Fortran D Language Specification*, Rice University, Technical Report COMP TR90-141, December 1990

[3] S. Hiranandani, K. Kennedy, C. Tseng, *Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines*, Proceedings of the Supercomputing Conference 1991, Albuquerque, 86-100, November 1991

[4] M. Gerndt, *Updating Distributed Variables in Local Computations*, Concurrency: Practice and Experience, Vol. 2(3), 171-193, September 1990

[5] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Ph.D. Dissertation, Yale University 1986, Technical Report YALEU/DCS/RR-492

[6] HPFF, *High Performance Fortran Language Specification*, High Performance Fortran Forum, May 1993, Version 1.0, Rice University Houston Texas

[7] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, A. Schwald, *Vienna Fortran - A language Specification Version 1.1*, University of Vienna, ACPC-TR 92-4, March 1992

[8] P. Brezany, K. Sanjari, *Processing Vector Subscripted Accesses to Arrays with Multi-Dimensional Distributions*, Internal Res. Rep., Inst. for Soft. Technology and Parallel Systems, University of Vienna, November 1993.

[9] H.P. Zima, B. Chapman, *Supercompilers for Parallel and Vector Computers*, Addison-Wesley, New York, 1990

[10] S. Benkner, P. Brezany, H.P. Zima, *Processing Array Statements and Procedure Interfaces in the Prepare HPF Compiler*. To appear in the Proc. of the Compiler Construction Conf., Edinburgh, April 1994.

[11] S. Benkner, P. Brezany, K. Sanjari, V. Sipkova,*Processing Vector Subscripted Accesses to Arrays with Multi-Dimensional Distributions*, Internal Res. Rep., Inst. for Soft. Technology and Parallel Systems, University of Vienna, December 1993

[12] R. Das, J. Saltz,*A manual for Parti runtime primitives - revision 2*, Internal Research Report, University of Maryland, 1992

[13] G.Fox, S.Hiranadani, K.Kennedy, C.Koelbel, U.Kremer, C.Tseng, M.Wu, *Fortran D Language Specification*, Rice University, Technical Report COMP TR90-141, Dezember 1990

[14] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz, *Compiler Analysis for Irregular Problems in Fortran D*, Proceedings of the Third Workshop on Compilers for Parallel Computers, Vienna, Austria, July 1992

[15] S.Hiranandani, K.Kennedy, C.Tseng, *Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines*, Proceedings of the Supercomputing Conference 1991, Albuquerque, 86-100, Nov. 1991

[16] C. Koelbel, *Compiling Programs for Nonshared Memory Machines*, Ph.D. Dissertation, Purdue University, West Lafayette, IN, November 1990.

[17] R. Ponnusany, J. Saltz, A. Choudhary, *Runtime-Compilation Techniques for Data Partitioning and Communication Schedule Reuse*, CS-TR-93-32, University of Maryland, College Park, MD, April 1993

[18] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, *Run-time scheduling and execution of loops on message passing machines*, Journal of Parallel and Distributed Computing, 8(2):303–312, 1990

[19] M. Weiser, *Program Slicing*, IEEE Transactions on Softw. Eng., Vol. 10, No. 4, July 1984, 352–357

[20] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, A. Schwald, *Vienna Fortran - A language Specification Version 1.1*, University of Vienna, ACPC-TR 92-4, March 1992