Forschungszentrum Jülich

Zentralinstitut für Angewandte Mathematik

Interner Bericht

Proceedings of the Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'01)

Jörg Striegnitz, Kei Davis et. al. (Eds.)*

FZJ-ZAM-IB-2001-14

FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Proceedings of the Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'01)

Jörg Striegnitz, Kei Davis et. al. (Eds.)*

FZJ-ZAM-IB-2001-14

Dezember 2001

(letzte Änderung: 17.12.2001)

(*) Modelling, Algorithms, and Informatics Group, CCS-3 MS B256
Los Alamos National Laboratory
Los Alamos, NM 87545, USA

Preface

This report comprises the Proceedings of the Workshop on Parallel / High Performance Object-Oriented Scientific Computing (POOSC'01) that was held at the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) in Tampa, Florida, USA, on 14 October 2001. The workshop was a joint organization of Research Centre Jülich, Los Alamos National Laboratory and Indiana University.

Today scientific programming has reached an unprecedented degree of complexity. Sophisticated algorithms, a wide range of hardware environments, and an increasing demand for system integration and portability have shown that language-level abstraction must be increased without loss of performance.

Work presented at previous POOSC workshops has shown that the OO approach provides an effective means for the design of complex scientific systems, and it is even possible to design abstractions and applications that have to fulfill strict performance constraints.

However, OO still isn't embraced in high performance computing and there is still demand for research and discussions. Previous POOSC workshops have proven that a workshop is an ideal venue for this.

Usually, there are no OOPSLA proceedings for workshops. However, since this years contributions have underlined the variety and multidisciplinary character of this research field, we as organizers of POOSC, decided to publish a collection of selected papers ourselves.

We thank all the contributors, referees, attendees and the OOPSLA workshop organizers for helping in making this workshop a highly successful event.

December 2001

Jörg Striegnitz

Kei Davis

Workshop Organizers

Kei Davis Modelling, Algorithms, and Informatics Group, CCS-3 MS B256
Los Alamos National Laboratory
Los Alamos, NM 87545, USA
kei@c3.lanl.gov

Jörg Striegnitz Research Centre Jülich
John von Neumann Institute for Computing (NIC)
Central Institute for Applied Mathematics (ZAM)
42425 Jülich, Germany
J.Striegnitz@fz-juelich.de

Programme Committee

Kei Davis Los Alamos National Laboratory, Los Alamos, NM, USA
Andrew Lumsdaine Indiana University, Bloomington, IN, USA
Bernd Mohr Research Centre Jülich, Germany
Jeremy Siek Indiana University, Bloomington, IN, USA
Jörg Striegnitz Research Centre Jülich
Todd Veldhuizen Indiana University, Bloomington, IN, USA

Table of Contents

Implementing a High Performance Tensor Library
W. Landry

Dynamic Compilation of C++ Template Code
M. J. Cole, S. G. Parker

Parallelization of an Object-Oriented Particle-In-Cell Simulation
S. Pinkenburg, M. Ritt, W. Rosentiel

OoLaLa: Transformations for Implementations of Matrix Operations at High Abstraction Levels
M. Lujan, J. R. Gurd, T. L. Freeman

Parallel Code Generation in MathModelica / An Object-Oriented Based Simulation Environment
P. Aronsson, P. Fritzson

Structured Exception Semantics for Concurrent Loops
J. Winstead, D. Evans

Design Patterns for Library Optimizations
D. Gregor, S. Schupp, D. Musser

An Interactive Environment for Supporting the Paradigm Shift from Simulation to Optimization
C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch

Generic Programming for High Performance Scientific Applications
L. Lee, A. Lumsdaine

Implementing a High Performance Tensor Library

Walter Landry
University of Utah
landry@physics.utah.edu

October 2, 2001

Abstract

Template methods have opened up a new way of building C++ libraries. These methods allow the libraries to combine the seemingly contradictory qualities of ease of use and uncompromising efficiency. However, libraries that use these methods are notoriously difficult to develop. This article examines the benefits reaped and the difficulties encountered in using these methods to create a friendly, high performance, tensor library. We find that template methods mostly deliver on this promise, though requiring moderate compromises in either usability or efficiency.

1 Introduction

Tensors are used in a number of scientific fields, such as geology, mechanical engineering, and astronomy. They can be thought of as generalizations of vectors and matrices. Consider the rather prosaic task of multiplying a vector P by a matrix T, yielding a vector Q

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \end{pmatrix} = \begin{pmatrix} T_{xx} & T_{xy} & T_{xz} \\ T_{yx} & T_{yy} & T_{yz} \\ T_{zx} & T_{zy} & T_{zz} \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix}.$$

If we write out the equations explicitly then

$$\begin{aligned} Q_x &= T_{xx}P_x + T_{xy}P_y + T_{xz}P_z, \\ Q_y &= T_{yx}P_x + T_{yy}P_y + T_{yz}P_z, \\ Q_z &= T_{zx}P_x + T_{zy}P_y + T_{zz}P_z. \end{aligned}$$

Alternatively, we can write it as

$$\begin{aligned} Q_x &= \sum_{j=x,y,z} T_{xj}P_j \\ Q_y &= \sum_{j=x,y,z} T_{yj}P_j \\ Q_z &= \sum_{j=x,y,z} T_{zj}P_j \end{aligned}$$

or even more simply as

$$Q_i = \sum_{j=x,y,z} T_{ij} P_j,$$

where the index i is understood to stand for x, y, and z in turn. In this example, P_j and Q_i are vectors, but could also be called rank 1 tensors (because they have one index). T_{ij} is a matrix, or a rank 2 tensor. The more indices, the higher the rank. So the Riemann tensor in General Relativity, R_{ijkl}, is a rank 4 tensor, but can also be envisioned as a matrix of matrices. There are more subtleties involved in what defines a tensor, but it is sufficient for our discussion to think of them as generalizations of vectors and matrices.

Einstein introduced the convention that if an index appears in two tensors that multiply each other, then that index is implicitly summed. This mostly removes the need to write the summation symbol $\sum_{j=x,y,z}$. Using this Einstein summation notation, the matrix-vector multiplication becomes simply

$$Q_i = T_{ij} P_j.$$

Of course, now that the notation has become so nice and compact, it becomes easy to write much more complicated formulas such as the definition of the Riemann tensor

$$R^i_{jkl} = dG^i_{jkl} - dG^i_{lkj} + G^m_{jk} G^i_{ml} - G^m_{lk} G^i_{mj}.$$

There are some subtle differences between tensors with indices that are upstairs (like T^i), and tensors with indices that are downstairs (like T_i), but for our purposes we can treat them the same. Now consider evaluating this equation on an array with N points, where N is much larger than the cache size of the processor. We could use multidimensional arrays and start writing lots of loops

```
for(int n=0;n<N;++n)
  for(int i=0;i<3;++i)
    for(int j=0;j<3;++j)
      for(int k=0;k<3;++k)
        for(int l=0;l<3;++l)
          {
            R[i][j][k][l][n]=dG[i][j][k][l][n]
                             - dG[i][l][k][j][n];
            for(int m=0;m<3;++m)
              R[i][j][k][l][n]+=G[m][j][k][n]*G[i][m][l][n]
                               - G[m][l][k][n]*G[i][m][j][n];
          }
```

This is a dull, mechanical, error-prone task, exactly the sort of thing computers are supposed to do for you. This style of programming is often referred to as C-tran, since it is programming in C++ but with all of the limitations of Fortran 77. We would like to write something like

```
R(i,j,k,l)=dG(i,j,k,l) - dG(i,l,k,j)
           + G(m,j,k)*G(i,m,l) - G(m,l,k)*G(i,m,j);
```

and have the computer do all of the summing and iterating over the grid automatically.

There are a number of libraries with varying amounts of tensor support ([1][2][3][4][5][6][7]). With one exception, they are all either difficult to use (primarily, not providing implicit summation), or they are not efficient. GRPP [6] solves this conundrum with a proprietary mini-language, making it difficult to customize and extend. With expression templates, it is possible to create a library within the C++ language which is both efficient and relatively easy to use.

2 Implementations

2.1 The Easy-to-Implement, Inefficient Solution with Nice Notation

The most straightforward way to proceed is to make a set of classes (Tensor1, Tensor2, Tensor3, etc.) which simply contains arrays of doubles of size N. Then we overload the operators +, - and * to perform the proper calculation and return a tensor as a result. The well known problem with this is that it is slow and a memory hog. For example, the expression

$$A_i = B_i + C_i \left(D_j E_j \right),$$

will generate code equivalent to

```
double *temp1=new double[N];
for(int n=0;n<N;++n)
  for(int i=0;i<3;++i)
    temp1[n]=D[i][n]*E[i][n];
double *temp2[3]
temp2[0]=new double[N];
temp2[1]=new double[N];
temp2[2]=new double[N];
for(int n=0;n<N;++n)
  for(int i=0;i<3;++i)
    temp2[i][n]=C[i][n]*temp1[n];
double *temp3[3]
temp3[0]=new double[N];
temp3[1]=new double[N];
temp3[2]=new double[N];
for(int n=0;n<N;++n)
  for(int i=0;i<3;++i)
    temp3[i][n]=B[i][n]+temp2[i][n];
for(int n=0;n<N;++n)
  for(int i=0;i<3;++i)
    A[i][n]=temp3[i][n];
delete[] temp1;
delete[] temp2[0];
```

```
delete[] temp2[1];
delete[] temp2[2];
delete[] temp3[0];
delete[] temp3[1];
delete[] temp3[2];
```

This required three temporaries ($temp1 = D_j E_j$, $temp2_i = C_i * temp1$, $temp3_i = B_i + temp2_i$) requiring 7N doubles of storage. None of these temporaries disappear until the whole expression finishes. For expressions with higher rank tensors, even more temporary space is needed. Moreover, these temporaries are too large to fit entirely into the cache, where they can be quickly accessed. The temporaries have to be moved to main memory as they are computed, even though they will be needed for the next calculation. With current architectures, the time required to move all of this data back and forth between main memory and the processor is much longer than the time required to do all of the computations.

2.2 The Hard-to-Implement, Somewhat Inefficient Solution with Nice Notation

This is the sort of problem for which template methods are well-suited. Using expression templates [9], we can write

```
A(i)=B(i)+C(i)*(D(j)*E(j));
```

and have the compiler transform it into a something like

```
for(int n=0;n<N;++n)
  for(int i=0;i<3;++i)
    {
      A[i][n]=B[i][n];
      for(int j=0;j<3;++j)
        A[i][n]+=C[i][n]*(D[j][n]*E[j][n]);
    }
```

The important difference here is that there is only a single loop over the N points. The large temporaries are no longer required, and the intermediate results (like `D[j][n]*E[j][n]`) can stay in the cache. This is a specific instance of a more general code optimization technique called loop-fusion. It keeps variables that are needed for multiple computations in the cache, which has much faster access to the processor than main memory.

This will have both nice notation and efficiency *for this expression.* What about a group of expressions? For example, consider inverting a symmetric, 3x3 matrix (rank 2 tensor) `A`. Because it is small, a fairly good method is to do it directly

```
det=A(0,0)*A(1,1)*A(2,2) + A(1,0)*A(2,1)*A(0,2)
    + A(2,0)*A(0,1)*A(1,2) - A(0,0)*A(2,1)*A(1,2)
```

```
    - A(1,0)*A(0,1)*A(2,2) - A(2,0)*A(1,1)*A(0,2);
I(0,0)= (A(1,1)*A(2,2) - A(1,2)*A(1,2))/det;
I(0,1)= (A(0,2)*A(1,2) - A(0,1)*A(2,2))/det;
I(0,2)= (A(0,1)*A(1,2) - A(0,2)*A(1,1))/det;
I(1,1)= (A(0,0)*A(2,2) - A(0,2)*A(0,2))/det;
I(1,2)= (A(0,2)*A(0,1) - A(0,0)*A(1,2))/det;
I(2,2)= (A(1,1)*A(0,0) - A(1,0)*A(1,0))/det;
```

Through the magic of expression templates, this will then get transformed into something like

```
for(int n=0;n<N;++n)
  det[n]=A[0][0][n]*A[1][1][n]*A[2][2][n]
         + A[1][0][n]*A[2][1][n]*A[0][2][n]
         + A[2][0][n]*A[0][1][n]*A[1][2][n]
         - A[0][0][n]*A[2][1][n]*A[1][2][n]
         - A[1][0][n]*A[0][1][n]*A[2][2][n]
         - A[2][0][n]*A[1][1][n]*A[0][2][n];
for(int n=0;n<N;++n)
  I[0][0][n]= (A[1][1][n]*A[2][2][n]
               - A[1][2][n]*A[1][2][n])/det[n];
for(int n=0;n<N;++n)
  I[0][1][n]= (A[0][2][n]*A[1][2][n]
               - A[0][1][n]*A[2][2][n])/det[n];
for(int n=0;n<N;++n)
  I[0][2][n]= (A[0][1][n]*A[1][2][n]
               - A[0][2][n]*A[1][1][n])/det[n];
for(int n=0;n<N;++n)
  I[1][1][n]= (A[0][0][n]*A[2][2][n]
               - A[0][2][n]*A[0][2][n])/det[n];
for(int n=0;n<N;++n)
  I[1][2][n]= (A[0][2][n]*A[0][1][n]
               - A[0][0][n]*A[1][2][n])/det[n];
for(int n=0;n<N;++n)
  I[2][2][n]= (A[1][1][n]*A[0][0][n]
               - A[1][0][n]*A[1][0][n])/det[n];
```

Once again, we have multiple loops over the grid of N points. We also have a temporary, `det`, which will be moved between the processor and memory multiple times and can not be saved in the cache. In addition, each of the elements of `A` will get transferred four times. If we instead manually fuse the loops together

```
for(int n=0;n<N;++N)
  {
```

```
    double det=A[0][0][n]*A[1][1][n]*A[2][2][n]
               + A[1][0][n]*A[2][1][n]*A[0][2][n]
               + A[2][0][n]*A[0][1][n]*A[1][2][n]
               - A[0][0][n]*A[2][1][n]*A[1][2][n]
               - A[1][0][n]*A[0][1][n]*A[2][2][n]
               - A[2][0][n]*A[1][1][n]*A[0][2][n];
    I[0][0][n]=(A[1][1][n]*A[2][2][n]
                - A[1][2][n]*A[1][2][n])/det;
    // and so on for the other indices.
    .
    .
    .
}
```

then `det` and the elements of `A` at a particular `n` can fit in the cache while computing all six elements of `I`. After that, they won't be needed again. For N=100,000 this code takes anywhere from 10% to 50% less time (depending on architecture) while using less memory. This is not an isolated case. In General Relativity codes, there can be over 100 named temporaries like `det`. Unless the compiler is omniscient, it will have a hard time fusing all of the loops between statements and removing extraneous temporaries. It becomes even more difficult if there is an additional loop on the outside which loops over multiple grids, as is common when writing codes that deal with multiple processors or multiple resolutions of the same grid (as happens with adaptive grid methods).

As an aside, the Blitz library [1] uses this approach. On the benchmark page for the Origin 2000/SGI C++ [8], there are results for a number of loop kernels. For many of them, Blitz compares quite favorably with the Fortran versions. However, whenever there is more than one expression with terms common to both expressions (as in loop tests #12-14, 16, 23-24) there are dramatic slow downs. It even mentions explicitly (after loop test #14) "The lack of loop fusion really hurts the C++ versions."

The POOMA library [7] uses an approach which should solve some of these problems. It defers calculations and then evaluates them together in a block. However, it still requires storage for named temporaries.

Does all this mean that we have to go back to C-tran for performance?

2.3 The Hard-to-Implement, Efficient Solution with only Moderately Nice Notation

The flaw in the previous method is that it tried to do two things at once: implicitly sum indices and iterate over the grid. Iterating over the grid while inside the expression necessarily meant excluding other expressions from that iteration. It also required temporaries to be defined over the entire grid. To fix this, we need to manually fuse all of the loops, and provide for temporaries that won't be defined over the entire grid. We did this by making two kinds of tensors. One of them just holds the elements (so a Tensor1 would have three doubles,

and a Tensor2 has 9 doubles). This is used for the local named temporaries. The other kind holds pointers to arrays of the elements. To iterate over the array, we overload operator++. A rough sketch of this tensor pointer class is

```
class Tensor1_ptr
  {
     mutable double *x, *y, *z;
   public:
     void operator++()
       {
         ++x;
         ++y;
         ++z;
       }
     \\ Indexing, assignment, initialization operators etc.
  }
```

Making it a simple `double *` allows us to use any sort of contiguous storage format for the actual data. The data may be managed by other libraries, giving us access to a pointer that may change. In that sense, the Tensor is not the only owner of the data, and all copies of the Tensor have equal rights to access and modify the data.

We make the pointers `mutable` so that we can iterate over `const Tensor1_ptr`'s. The indexing operators for `const Tensor1_ptr` returns a `double`, not `double *` or `double &`, so the actual data can't be changed. This keeps the data logically `const`, while allowing us to look at all of the points on the grid for that `const Tensor1_ptr`.

We would then write the matrix inversion example as

```
for(int n=0;n<N;++N)
  {
    double det=A(0,0)*A(1,1)*A(2,2) + A(1,0)*A(2,1)*A(0,2)
               + A(2,0)*A(0,1)*A(1,2) - A(0,0)*A(2,1)*A(1,2)
               - A(1,0)*A(0,1)*A(2,2) - A(2,0)*A(1,1)*A(0,2);
    I(0,0)= (A(1,1)*A(2,2) - A(1,2)*A(1,2))/det;
    I(0,1)= (A(0,2)*A(1,2) - A(0,1)*A(2,2))/det;
    I(0,2)= (A(0,1)*A(1,2) - A(0,2)*A(1,1))/det;
    I(1,1)= (A(0,0)*A(2,2) - A(0,2)*A(0,2))/det;
    I(1,2)= (A(0,2)*A(0,1) - A(0,0)*A(1,2))/det;
    I(2,2)= (A(1,1)*A(0,0) - A(1,0)*A(1,0))/det;
    ++I;
    ++A;
  }
```

An example which mixes the pointer and non-pointer tensor classes is

```
void f(const Tensor2_ptr T, const Tensor1_ptr P,
       Tensor1_ptr Q, const int N)
{
  for(int n=0;n<N;++n)
    {
      Tensor2 T_symmetric;
      T_symmetric(i,j)=(T(i,j)+T(j,i))/2;
      Q(i)=T_symmetric(i,j)*P(j);
      ++Q;
      ++T;
      ++P;
    }
}
```

This function symmetrizes the matrix `T` and multiplies it by `P`, putting the result in `Q`. The body inside the loop can also be simplified to remove the named temporary `T_symmetric`

```
Q(i)=(T(i,j)+T(j,i))*P(j)/2;
++Q;
++T;
++P;
```

This solution is not ideal and has a few hidden traps, but is certainly better than C-tran. It requires a manually created loop over the grid, and all relevant variables have to be incremented. Care must also be taken not to attempt to iterate through a grid twice. That is why this function `f()` passes the tensors by value, and not by reference. Otherwise, the pointers in `Q`, `T`, and `P` would have been iterated to the end. The parent function calling `f()` would then not be able to use its copies of `Q`, `T`, and `P`.

In practice, these were not serious problems, because most of the logic of our program is in the manipulation of local named variables. Only a few variables (the input and output) need to be explicitly iterated. Even those that were iterated were locally constructed from global arrays given by another library that managed multiprocessor computations.

However, this may not be the right kind of solution for generic arrays. They correspond to rank 0 tensors (tensors without any indices). It is a win for higher rank tensors because most of the complexity is in the indices. But for generic arrays, there are no indices. A solution like this would look almost identical to C-tran.

3 How Well Does it Work?

We have implemented the first (inefficient) method and the third (efficient) method [11]. We did not attempt to implement the second method, because it was clear that it could not be as efficient as the second method, while still being a difficult chore to implement. We have also not attempted a direct comparison with other tensor libraries, because most do not support

Compiler/Operating System	Compiles efficient library?
Comeau como 4.2.45.2 + libcomobeta14/Linux x86 Compaq cxx 6.3/Tru64 GNU gcc 2.95.2/Linux x86, 2.95.3/Solaris, 2.95.2/AIX KAI KCC 4.0d/Linux x86, 4.0d/AIX	Yes
IBM xlC 5.0.1.0/AIX	Yes with occasional ICE's
SGI CC 7.3.1.1m/Irix	Somewhat-no <cmath> and can't override template instantiation limit
Intel icc 5.0/Linux x86	Somewhat-uses excessive resources and can't override template instantiation limit
Portland Group pgCC 3.2/Linux x86	No, can't handle long mangled names, no <cmath>
Sun CC 6.1/Solaris Sparc	No, doesn't support partial specialization with non-type template parameters

Table 1: Compiler Comparison

implicit summation and none of them support the wide range of tensor types needed (ranks 1, 2, 3 and 4 with various symmetries). This makes replicating the functionality in the tests extremely time consuming.

We found that, when compiled with KAI's KCC compiler on an IBM SP2, the efficient library runs about twice as fast and uses a third of the memory of the inefficient library. When compiled with GNU gcc or IBM's xlC, the efficient library code was 10-20% slower than when compiled with KCC.

However, not all compilers support enough of the standard to compile the efficient library, while the inefficient method works with almost any compiler. A comparison of twelve combinations of compiler and operating system is shown in Table 1.

Surprisingly, not all of the compilers made <cmath> available, although it is easy to work around that. IBM's compiler seems to be immature, with a remarkable number of cases of Internal Compiler Errors (ICE's). The Portland Group compiler had trouble with the long mangled names produced by the compiler. Intel's compiler would routinely use much more memory and CPU time than other compilers on the same machine. The Sun compiler seems to be useless for these kinds of template techniques.

The C++ standard specifies that compliant programs can only rely on 17 levels of template instantiation. Otherwise, it would be difficult to detect and prevent infinite recursion. However, the intermediate types produced by template expression techniques can exceed this limit. Most compilers allowed us to override the limit on the number of pending instantiations, with the exception of the SGI, Intel and IBM compilers. The SGI and Intel compilers would not compile any program with too many levels. The IBM compiler did not honor the standard and happily compiled programs with more than 50 levels of template instantiation.

We made two small benchmarks to test how well the compilers could optimize away the overhead of the expression templates. The first is a simple loop

```
Tensor1 x(0,1,2), y(3,4,5), z(6,7,8);
for(int n=0;n<1000000;n++)
  {
    Index<'i'> i;
    x(i)+=y(i)+z(i);
      +(y(i)+z(i))-(y(i)+z(i))
      +(y(i)+z(i))-(y(i)+z(i))
      +(y(i)+z(i))-(y(i)+z(i))
    .
    .
    .
  }
```

The complexity of the expression is determined by how many `(y(i)+z(i))-(y(i)+z(i))` terms there are in the final expression. Note that since we're adding and subtracting the same amounts, the essential computation has not changed. We also coded a version of the code that used normal arrays instead of the Tensor1 class, and compared the execution speeds of the two versions.

For large expressions, KCC was the only compiler that could fully optimize away the overhead from the expression templates, although we had to turn off exceptions in order to do it. This is good evidence that we didn't make any serious optimization errors in implementation. For the other compilers, the slowdown increased with the number of expressions, becoming more than 100 times slower than the version using arrays.

This benchmark may be deceiving, though. The versions that use normal arrays all run at the same speed regardless of how many terms we added. This suggests that the compiler can remove the identically zero subexpressions. This wouldn't be possible in a most production codes, so the relative slowdown may not be as great.

To reduce this effect, we created a second benchmark. It computes the sum

```
Tensor1 y(0,1,2);
Tensor1 a1(2,3,4);
Tensor1 a2(5,6,7);
Tensor1 a3(8,9,10);
Tensor1 a4(11,12,13);
Tensor1 a5(14,15,16);
.
.
.
for(int n=0;n<1000000;++n)
  {
    const Index<'i'> i;
```

```
    const Index<'j'> j;
    const Index<'k'> k;
    const Index<'l'> l;
    const Index<'m'> m;
    y(i)+=a1(i)
          + 2*a2(i)
          + 3*a1(j)*a2(j)*a3(i)
          + 4*a1(j)*a3(j)*a2(k)*a2(k)*a4(i)
          + 5*a1(j)*a4(j)*a2(k)*a3(k)*a5(i)
          .
          .
          .
    a1(i)*=0.1;
    a2(i)*=0.2;
    a3(i)*=0.3;
    a4(i)*=0.4;
    a5(i)*=0.5;
    .
    .
    .
}
```

with complexity determined by how much we fill in the ellipses. Again, we also coded versions that use simple arrays and compared the execution speed of the two. Here, the story is much different. Figure 1 shows the relative execution times of the the Tensor1 and ordinary arrays versions against the number of operators (+ and *) in the expressions.

The specific compiler options used to create this plot are listed in the appendix. The performance of some of the compilers may be a little overstated since they don't optimize the ordinary array case as well as some other compilers. On Linux x86, the fastest compiler for the ordinary array code was the Intel compiler, and on AIX, it was IBM's xlC. So in Figures 2 and 3 we plot the relative execution time of the fastest ordinary array codes versus the various Tensor codes for Linux and AIX.

The first thing to notice in these plots is that none of the compilers can optimize the expressions completely. Even for the best compilers, the Tensor class can run as slow as 50% as ordinary arrays. However, it is unclear how much this matters in real codes. Writing the equivalent of our General Relativity code using ordinary arrays would be extremely tiresome and error-prone. However, as noted earlier, the differences between KCC, xlC, and gcc for our General Relativity code were only 10-20%. Unfortunately, looking at Figure 3, this might just be because they are all terrible at optimizing. Your mileage may vary.

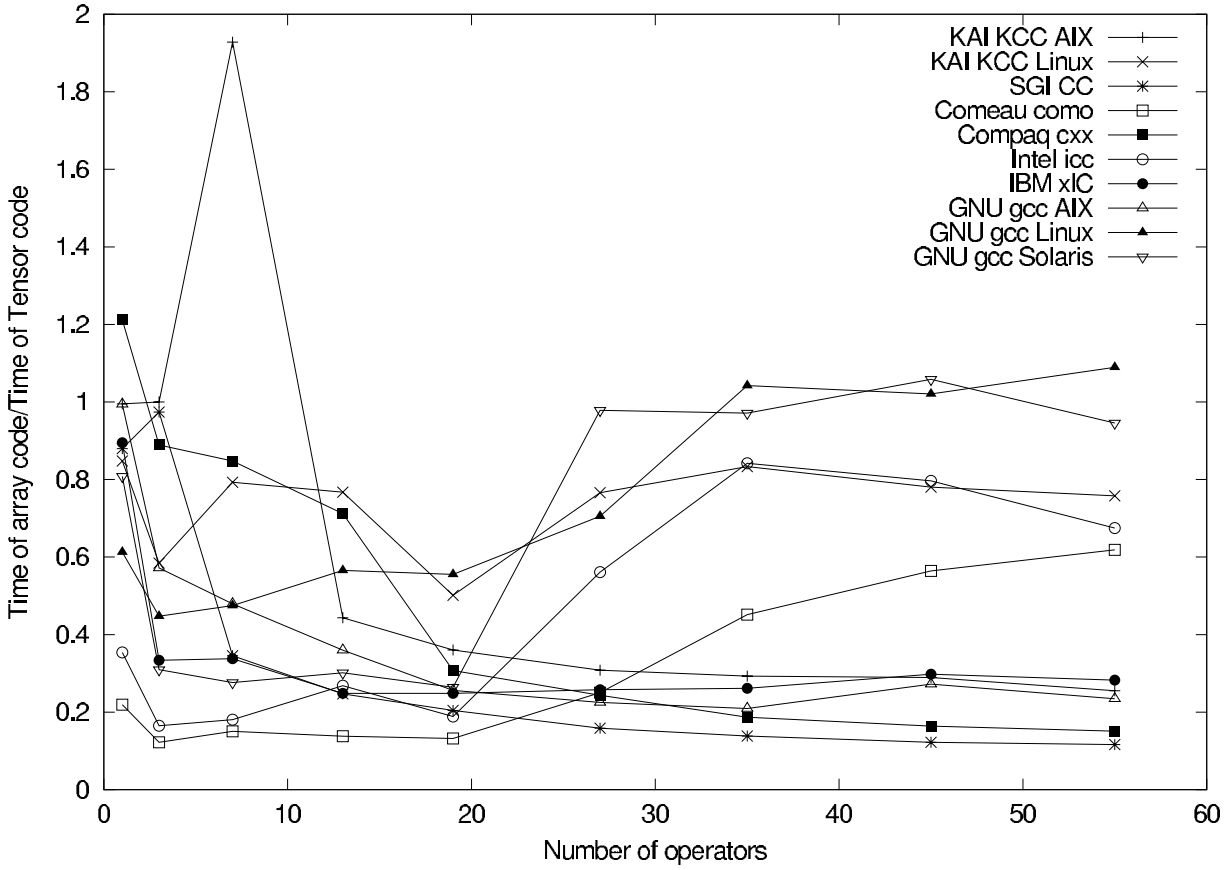

Figure 1: Relative execution times of arrays and Tensor1's

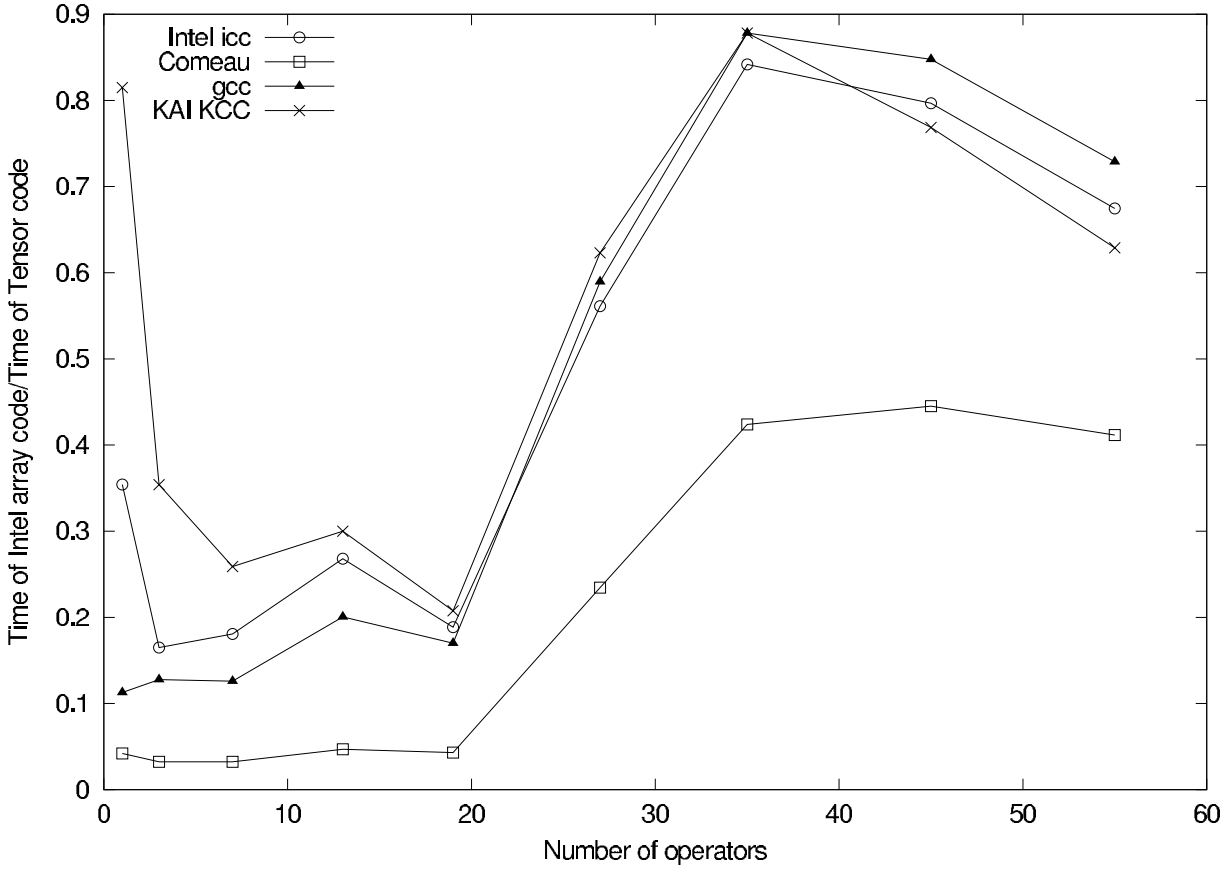

Figure 2: Relative execution times of fastest arrays and Tensor's on Linux x86

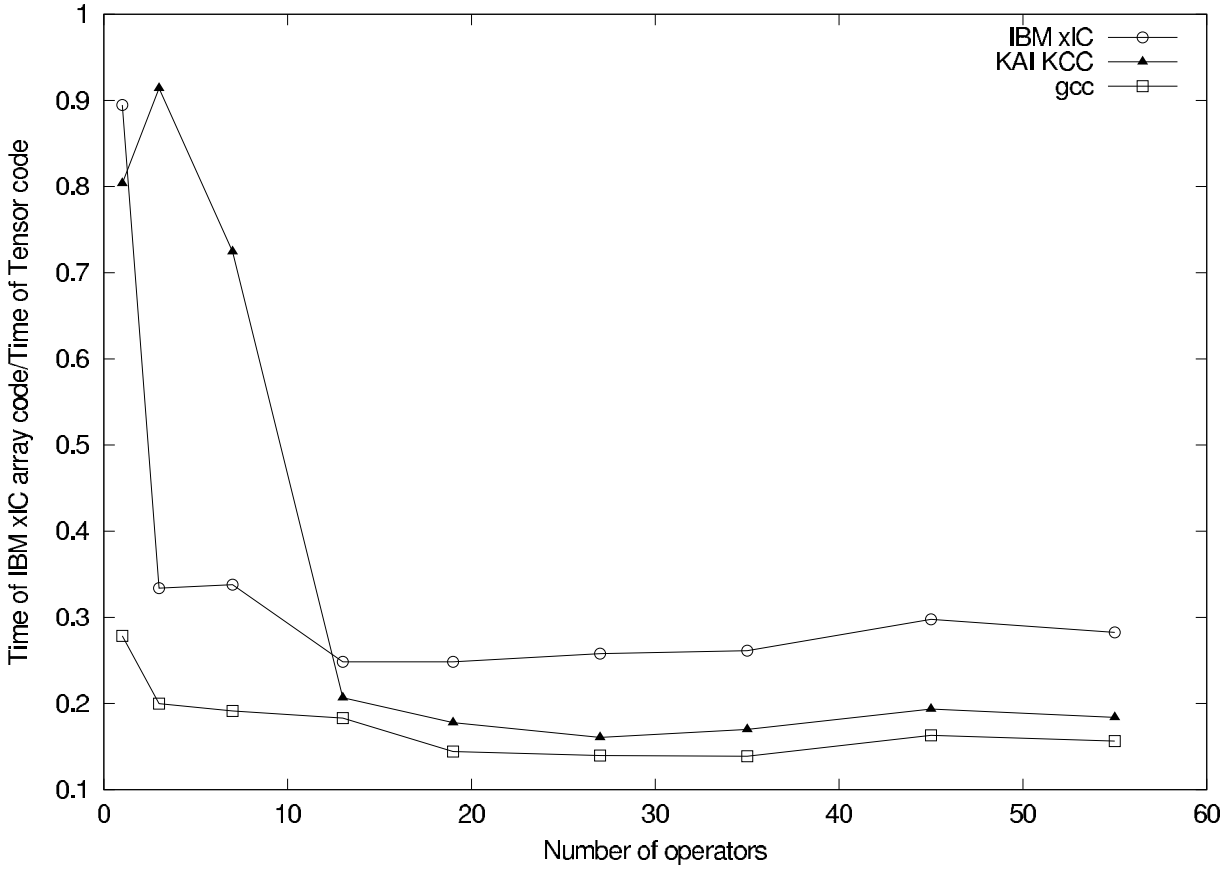

Figure 3: Relative execution times of fastest arrays and Tensor's on AIX

4 Extending the Library

A reader with foresight may have looked at the rough declaration of `Tensor1_ptr` and thought that hard coding it to be made up of `double*` is rather short sighted. It is not so difficult to envision the need for tensors made up of `int`'s or `complex<double>`. It might also be nice to use two or four dimensional tensors (so a Tensor1 would have 2 or 4 elements, a Tensor2 would have 4 or 16 elements). The obvious answer is to make the type and dimension into template parameters. We then specialize for each dimension and whether the type is a pointer

```
template<class T, int Dim> class Tensor1;
template<class T> class Tensor1<T,2> {
  T x, y;
  .
  .
  .
}
template<class T> class Tensor1<T*,2> {
  mutable T *x, *y;
  .
  .
  .
}
template<class T> class Tensor1<T,3> {
  T x, y, z;
  .
  .
  .
}
```

and so on. We can even make the arithmetic operators dimension agnostic with some template meta-programming [10]. Then, if you're trying to follow Buckaroo Banzai across the 8th dimension, you only have to define the Tensor1, Tensor2, Tensor3, etc. classes for eight dimensions, and all of the arithmetic operators are ready to use.

We have implemented this generalization [12]. It uncovers a deficiency in the template support by gcc, so it can't compile it. Also, KCC can't fully optimize complicated expressions in the first benchmark as it could with the simpler version of the library, leading to code that runs hundreds of times slower. Interestingly enough, the TinyVector classes in Blitz [1] are also templatized on type and dimension, and complicated expressions can not be fully optimized in that kind of benchmark as well.

However, the performance in the second benchmark is not affected in the same way. Figure 4 shows the relative execution times for arrays versus the more general Tensor's, and Figure 5 directly compares the two versions of the Tensor library.

The results are generally mixed, although KCC seems to do better while como does worse. However, the overall conclusions from the last section are unchanged. This is nice, in the sense that using a more general library hasn't caused another hit in performance.

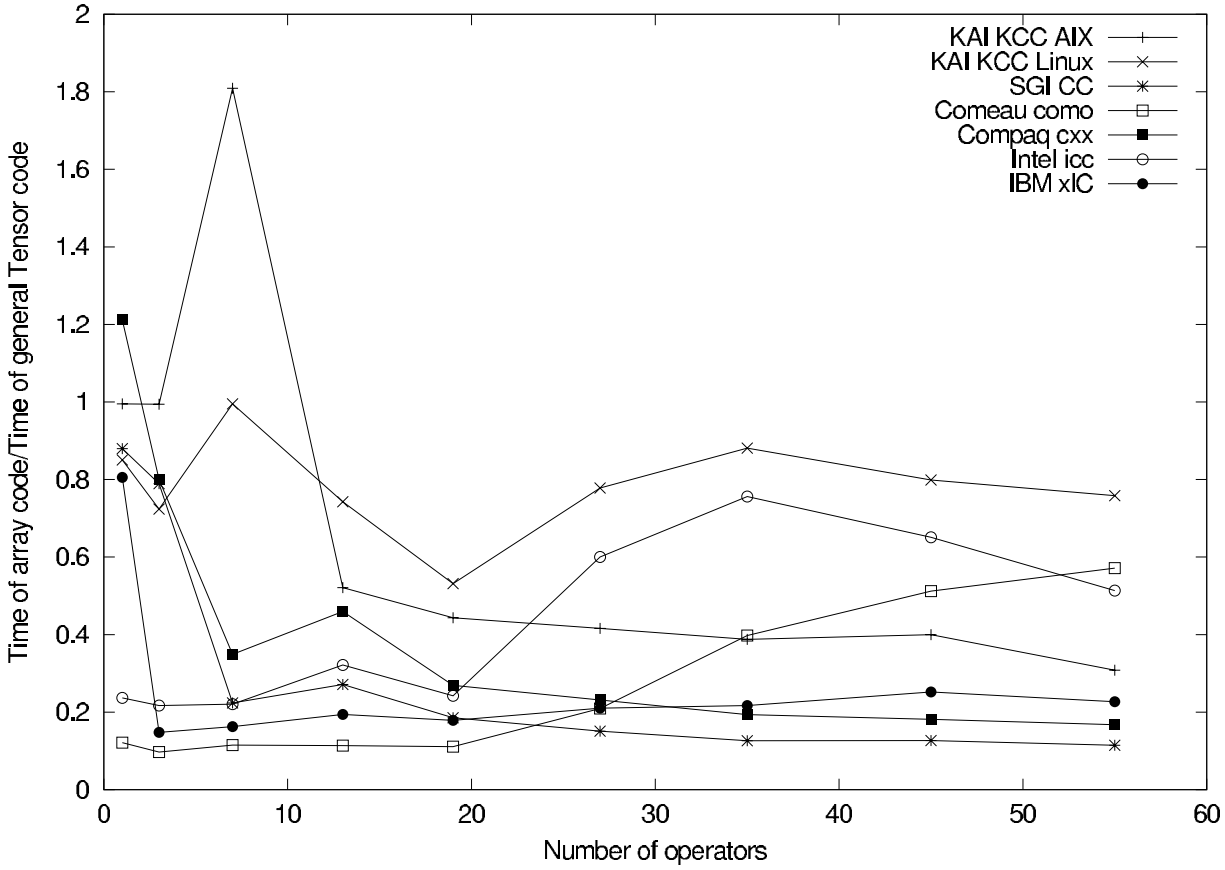

Figure 4: Relative execution time of arrays and more general Tensor's

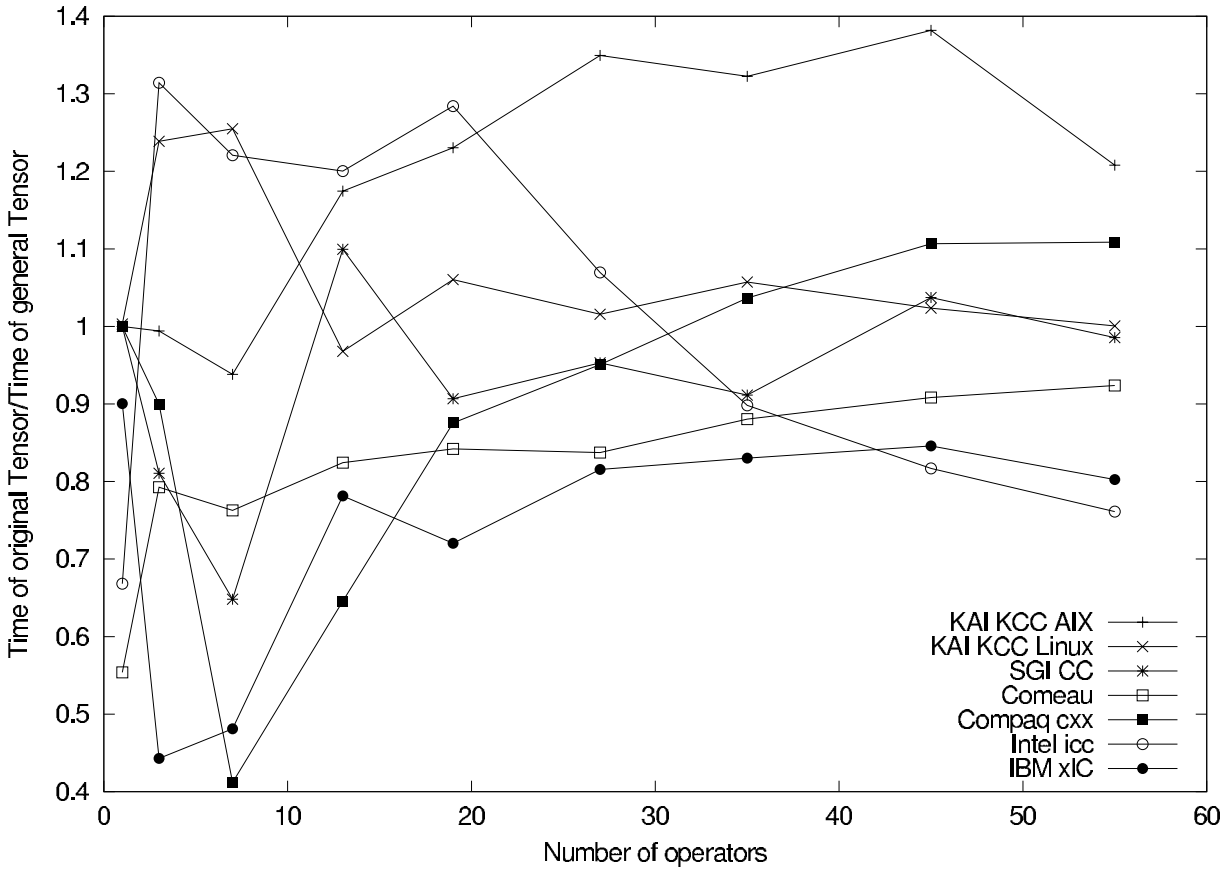

Figure 5: Relative execution time of simple and more general Tensor's

5 Conclusion

The original promise of expression templates as a way to get away from C-tran is not completely fulfilled. Although the syntax is much improved, there are still cases where a programmer must resort to at least some manual loops in order to get maximum performance. Even with this work, there are still performance penalties which vary from problem to problem.

Appendix: Compiler Options

gcc	-O3
	-ftemplate-depth-100
	-Drestrict=__restrict__
	-finline-functions
	-finline-limit-1000000
	-ffast-math
	-fno-rtti
	-fno-exceptions
como	-Drestrict=
	-O3
	–remove_unneeded_entities
	–pending_instantiations=100
icc	-restrict
	-O3
	-tpp6
	-xi
KCC 4.0d/Linux	+K3
	–restrict
	–no_exceptions
	–inline_auto_space_time=750000000000000
	–inline_implicit_space_time=200000000000000
	–inline_generated_space_time=40000000000000.0
	–inline_auto_space_time=100000000000000.0
	–max_pending_instantiations 100
KCC 4.0d/AIX	+K3
	–restrict
	–no_exceptions
	–inline_auto_space_time=750000000000000
	–inline_implicit_space_time=200000000000000
	–inline_generated_space_time=40000000000000.0
	–inline_auto_space_time=100000000000000.0

	–max_pending_instantiations 100
	-qmaxmem=100000
Compaq cxx	-std ansi
	-model ansi
	-nousing_std
	-noexceptions
	-nortti
	-Drestrict=__restrict
	-assume noptrs_to_globals
	-assume whole_program
	-assume noaccuracy_sensitive
	-inline all
	-fast
	-O5
	-non_shared
	-tune host
	-pending_instantiations 1000
	-nocleanup
SGI CC	-LANG:std
	-LANG:restrict=ON
	-64
	-O3
	-LANG:exceptions=OFF
	-IPA:space=1000000000
	-IPA:plimit=1000000000
	-OPT:unroll_times_max=100000
	-OPT:unroll_size=1000000
	-INLINE=all
	-IPA:alias=ON

Acknowledgements

We gratefully acknowledge the help of Comeau computing in providing a copy of their compiler for evaluation. This work was supported in part by NSF grant PHY 97-34871. An allocation of computer time from the Center for High Performance Computing at the University of Utah is gratefully acknowledged. CHPC's IBM SP system is funded in part by NSF Grant #CDA9601580 and IBM's SUR grant to the University of Utah.

References

[1] Todd Veldhuizen, Blitz, http://www.oonumerics.org/blitz

[2] Neil Gaspar, http://www.openheaven.com/believers/neil_gaspar/t_class.html

[3] Boris Jeremic, nDarray, http://civil.colorado.edu/nDarray/

[4] Wolfgang Bangerth, Guido Kanschat, Ralf Hartmann, Deal.II, http://gaia.iwr.uni-heidelberg.de/~deal/

[5] Robert Tisdale, SVMT, http://www.netwood.net/~edwin/svmt/

[6] TensorSoft, GRPP, http://home.earthlink.net/~tensorsoft/

[7] POOMA, http://www.acl.lanl.gov/pooma/

[8] http://oonumerics.org/blitz/benchmarks/Origin-2000-SGI/

[9] Todd Veldhuizen, "Expression Templates," *C++ Report*, Vol. 7 No. 5 (June 1995), pp. 26-31

[10] Todd Veldhuizen, "Using C++ template metaprograms," *C++ Report*, Vol. 7 No. 4 (May 1995), pp. 36-43.

[11] http://www.physics.utah.edu/~landry/FTensor.tar.gz

[12] http://www.physics.utah.edu/~landry/FTensor_new.tar.gz

Dynamic Compilation of C++ Template Code

Martin J. Cole, Steven G. Parker
mcole@cs.utah.edu, sparker@cs.utah.edu
SCI Institute
University of Utah

Abstract

Generic programming using the C++ template facility has been a successful method for creating high-performance, yet general algorithms for scientific computing and visualization. However, the use of templated code typically leads to propagation of more template code. Compiling all possible expansions of these templates can lead to massive template bloat. Furthermore, compile-time binding of templates requires that all possible permutations be known at compile time, limiting the runtime extensibility of the generic code. We present a method for deferring the compilation of these templates until an exact type is needed. This dynamic compilation mechanism will produce the minimum amount of compiled code needed for a particular application, while maintaining the generality and performance that templates innately provide. Through a small amount of supporting code within each templated class, the proper templated code can be generated at runtime without modifying the compiler. We describe the implementation of this goal within the SCIRun dataflow system. SCIRun is freely available online for research purposes.

Problem Description

SCIRun[1] is a scientific problem solving environment that allows the interactive construction and steering of large-scale scientific computations [1–3]. A scientific application is constructed by connecting computational elements (modules) to form a program (network). This program may contain several computational elements as well as several visualization elements, all of which work together in orchestrating a solution to a scientific problem. Geometric inputs and computational parameters may be changed interactively, and the results of these changes provide immediate feedback to the investigator. SCIRun is designed to facilitate large-scale scientific computation and visualization on a wide range of machines from the desktop to large shared-memory and distributed-memory supercomputers.

At the heart of any general visualization system is the data model. The data model is responsible for representing a wide range of different data representation schemes in a uniform fashion. In the case of SCIRun, the core piece of our data model is the field library [4, 5], where a field is simply a function represented over

[1] Pronounced "ski-run." SCIRun derives its name from the Scientific Computing and Imaging (SCI) Institute at the University of Utah.

some portion of 3D space. In most cases, that function is represented by some discrete approximation, such as a tetrahedral grid (i.e. a Finite Element Mesh) or a 3D rectangular grid (i.e. a Finite Difference mesh, or the product of a 3D medical scan such as Computed Tomography or Magnetic Resonance Imaging). Representing each of these fields in the most general form possible would lead to a number of inefficiencies, including a massive data explosion.

Therefore, we turn to C++ for mechanisms of providing access to these different field types in a uniform way. Typical operations include computing the minimum or maximum value in the field, iterating over discrete data points, and interpolating the value at a specified point in space. In C++, we can use inheritance and virtual functions to maintain a uniform interface.

We compared the runtime performance in a simple yet representative test program. The test times virtual method calls vs. template method calls of an identical function. The results show that there is a performance penalty to using a virtual interface. On Linux, the virtual method timed at 30.43 seconds, and the template version at 10.1 seconds, on Irix the same test ran at 10.21 seconds, and 2.84 seconds respectively. These results have led us away from a virtual interface and so for this task, we turn to generic programming. Table 1 shows these results in tabular form.

Table 1. Performance comparison for typical visualization queries using virtual functions and templates

Machine	Compiler	Processor	Virtual function time	Template time
SGI Origin 2000	MIPSPro 7.3.1.2	250 Mhz R10000	10.21 s	2.84 s
Linux PC Pentium III	GNU g++ 3.0	750 Mhz	30.43 s	10.1 s

Generic programming using the C++ template facility has been a successful method [6–8] for creating high-performance, yet general algorithms for scientific computing and visualization. Generic programming relies on the compiler to generate specialized instances of particular algorithms that are tailored to the underlying data representation. However, the use of templated code typically leads to propagation of more template code. Compiling all possible expansions of these templates can lead to massive template bloat. Furthermore, compile-time binding of templates requires that all possible permutations be known at compile time, limiting the runtime extensibility of the generic code.

As an example, consider the following realistic example from the SCIRun field library. Consider the set of different field classes: TetVol (TetraHedral Volume Grid), LatticeVol (3D Rectangular Lattice Grid), ContourField (A set of countour lines), and TriSurf (A 3D triangulated surface). On each of these fields, we can hold several different types of data, such as double, int, char, unsigned char, unsigned short, short, bool, Vector (3 doubles indicating a direction), and Tensor (6 doubles).

For these four field types, and these nine primitive types, the compiler would be required to generate a total of 36 different field combinations. Now consider these 36 types that are used with a computational or visualization algorithm that is parameterized on one field type. The compiler would also generate 36 versions of this algorithm. However, if the algorithm required two fields, and was therefore parameterized on two different field types, the compiler would be required to generate $36^2 = 1296$ different versions of that algorithm. For an algorithm with three different field types, $36^3 = 46656$ fully instantiated classes would be generated. These numbers grow as more field types, data types, and algorithms are supported.

Our compilers did indeed have problems compiling a fully instantiated version of our code. The compiler itself ran out of 32 bit address space during a global optimization pass. At this point, the template bloat moved from an annoyance to a critical bug.

Since SCIRun is an interactive system, any of these combinations could be used at any time. However, a typical user will use only a handful of different field types while using SCIRun. SCIRun is also extensible at run-time through the dynamic loading of new modules. In particular, new field types can be created by loaded modules, and these fields can be sent to other, pre-compiled modules. With a pure template-based approach, modules that were compiled without support for the new field would not be able to operate.

A different design of the field classes could easily solve this problem, but would have other weaknesses. If we used virtual functions instead of generic programming to access the different types of fields, the system would not suffer from the combinatoric explosion of templated types. However, this design would suffer a different drawback, namely performance. Virtual function calls are costly, and therefore prohibit fine-grained access to data elements. Furthermore, virtual functions thwart many of the optimizations performed by compilers, leading to substantially reduced performance over the template-based approach. As SCIRun is designed for computation and visualization of large-scale scientific datasets, we have found the virtual function solution to be unacceptable in many design situations.

Proposed Solution

Through the use of C++ templates, the compiler creates multiple versions of the code specific to particular data structures, primitive types, and algorithms. Each module that needs to work on one of the above mentioned classes, implements an algorithm templated on the exact field type. It is this algorithm that gets compiled when it is needed. For the purposes of illustration, consider templates of this form:

```
template<class Field1, class Field2> class Algorithm;
```

The system operates in the following simple steps:

1. Use C++ RTTI and additional run-time information to determine which field classes are in use. The calling module also specifies which algorithm is to be applied to these fields.
2. Generate a small amount of C++ code to instantiate the correct algorithm with the discovered field types.
3. Fork a process to compile the C++ code into a shared library.
4. Dynamically link this shared library into the running process, and locate the function that will create the instantiated object.
5. Call this function to create an instance of the specialized algorithm.
6. Make a single virtual function call to the algorithm, passing Field1 and Field2 and a generic base class.
7. Since the algorithm knows the concrete type of Field1 and Field2, it uses dynamic_cast to get a pointer to the specific type.
8. Finally, the algorithm performs its operation on the data.

To accomplish this, the algorithm, and the templated classes need to provide some information to the DynamicLoader so that it can create the C++ file that needs to be compiled. Below we explain the mechanisms that are necessary in the code to support these operations.

Related Work

Kennedy and Syme [9] describe their implementation of generics in the .NET Common Language Runtime. Their work provides a similar solution to the problem of bloat. They use JIT compilation to produce the object at run time, an option enabled by control over the virtual machine. This control enables a faster compile time, as well as the fact that the mechanism is hidden from the user. Essentially we have implemented a crude JIT compilation mechanism for C++. Our compilation/link/load takes longer, but since future runs need not compile and link, the cost is amortized over multiple executions of the SCIRun environment.

POOMA [10] is a high-performance C++ toolkit for parallel scientific computation that depends heavily on C++ templates for achieving high performance code. However, with POOMA, all required templates are instantiated at compile/link time instead of dynamically. Since POOMA is not an interactive system, it does not suffer from some of the same problems as SCIRun; the compiler only generates template instantiations that are required by the scientific program instead of every possible combination. Nevertheless, many POOMA compiles can take considerable time, and some of the template instantiations may never get executed. POOMA does provide constructs beyond what are required for SCIRun, including semi-automatic data parallelism for array expressions and other features. It is possible that our mechanisms could be combined with the expression template engine (PETE) from POOMA in order to provide dynamic compilation of complex scientific simulations.

Implementation

Through a small amount of supporting code within each templated class, the proper templated code can be generated at runtime. The system generates a small amount of C++ code that includes:

- All C++ header files required to compile the algorithm.
- Namespace satisfaction statements.
- A creation function that returns an instance of the desired algorithm.

An example of such code is shown in Figure 1.

Fig. 1. An example of the small automatically generated C++ code to instantiate the proper templated class. This will generate the RenderField algorithm, with the field of type TetVol<double>.

```
// This is an automatically generated file, do not edit!
#include "../src/Core/Datatypes/TetVol.h"
#include "../src/Core/Algorithms/Visualization/RenderField.h"
using namespace SCIRun;

extern "C" {
RenderFieldBase* maker() {
  return scinew RenderField<TetVol<double> >;
}
}
```

Algorithm structure

A templated algorithm inherits from an algorithm base class. This class defines the interface that the algorithm should have. Each templated algorithm provides the underlying implementation for the pure virtual. The interface has no restrictions, save that it be virtual. All access to the interface happens at the algorithm base class level. Typically the interface is a single pure virtual method with arguments that satisfy the passing of data from the calling module. This allows the entire algorithm to be executed with a single virtual method call. All such algorithm base classes inherit from a common base class that the DynamicLoader maps to the string representation of the exact type for an algorithm.

TypeDescription

Each object that supports dynamic compilation, must provide a TypeDescription object. This object holds strings that describe its type, the namespace that

it belongs to, and the path to the .h file that declares it. The latter is frequently provided by simply returning the value of the standard __FILE__ preprocessor macro. Most of this internal type information is not available through the standard C++ RTTI facility, so TypeDescription provides that augmented internal information. This object can also recursively contain the TypeDescriptions for sub types. For example a foo<bar, foobar<int> > has a TypeDescription that has a both bar, and foobar TypeDescriptions. The foobar TypeDescription, has the int TypeDescription. A recursive traversal of this object allows us to output a string that matches the exact type for the object.

CompileInfo

The exact type of an algorithm is composed of the algorithm and a data type. A module that wants to create such an algorithm can not have an instance of the algorithm until after dynamic compilation. For this reason, the CompileInfo object is needed, which provides information similar to the TypeDescription, but without the mapping to an underlying object. This object is also the structure that ultimately holds the strings that get written to the .cc file in preparation for compilation. The CompileInfo gets filled with its information when it is passed along to each TypeDescription object that makes up the data type, as well as to the algorithm. The completed CompileInfo object is passed to the DynamicLoader when the calling module requests an instance of the specialized algorithm.

DynamicLoader

The DynamicLoader is the interface for a module to get a handle on the algorithm it needs. Its interface is simple: A module builds up a CompileInfo for the module, and asks the DynamicLoader for a handle to algorithm object. The DynamicLoader then looks up the algorithm in an internal cache. If it does not exist, it uses the CompileInfo to write a small C++ file to disk in a predefined directory. This directory has a makefile that knows how to build a shared library from that C++ file. The DynamicLoader then forks a shell and builds the desired library. Once the shared library is compiled, it is loaded and stored in the internal cache. Each dynamically compiled library has a uniformly named creation function, maker(), which returns a pointer to the algorithm base class. This function pointer is stored in the hash table, and called each time an algorithm is requested by a module, giving each module a separate instance of the algorithm, including unique state for each algorithm instance. Since SCIRun is a multi-threaded program, the DynamicLoader has synchronization code designed such that threads block waiting for a unique type, but it can compile an unlimited number of distinct algorithms concurrently.

Calling Module

The calling module knows of the DynamicLoader, and has a Field base class that needs to be compiled into an algorithm. The algorithm base type is known, as it

is integral to the module's function. The exact algorithm will be templated on the exact Field type. This is only known to the module through strings, not types. The module fetches the CompileInfo from the algorithm base class, by feeding it the input Field's TypeDescription object, then asks the DynamicLoader for an algorithm that matches the CompileInfo. No instance of the exact types are instantiated until runtime when they are asked for by the module.

Performance

SCIRun is currently supported on Irix and Linux. The runtime compilation and linking depends of course on the complexity of the algorithm, but it is typically on the order of seconds. For a user who is not modifying the code that the algorithm depends upon, this is a one time operation. The library remains on disk, so that upon the next run the library can simply be reloaded after a makefile-based dependency check, skipping the compile step.

For a commonly used library in the SCIRun system, the initial compile and link step takes about 7 seconds on Linux, and about 40 seconds on Irix. It should be noted that the longer Irix compilation and linking often produces better optimized code.

Since the compilation only happens once, the system rapidly amortizes the cost of the compilation from the increased performance during execution of the algorithm. Furthermore, the system facilitates more rapid development cycles, as the typical developer does not need to wait for the compiler to instantiate a multitude of template classes at link time.

Disadvantages

This system is not built into the language, so it requires source code, and a C++ compiler on the system. There is additional code maintenance required. The information that RTTI lacks, and we require, needs to be added to each new datatype that is added to the system. The libraries are all created in a single directory, so users sharing a build must have write permissions in the directory. The runtime compilation step can be time consuming for a large network of modules the first time through. Developers may not see compile errors until runtime, when the actual instantiation of the exact algorithm gets compiled.

Future work

The SCIRun dynamic compilation framework has been used for instantiating classes that could have been known at compile time. We could achieve higher performance in some cases by using even more run-time information in the dynamic compilation phase. For example, array dimensions or repeatedly used constant values could be compiled into the template instance to achieve higher performance.

The current system does not provide a mechanism for specifying special libraries that an algorithm or field class may need. As a result, the makefiles link the shared object against several known libraries, many of which may not be needed. This deficiency could be overcome by requiring the developer to specify required libraries in the TypeDescription and CompileInfo objects.

SCIRun operates under a shared-memory parallel environment. In this case, we are only required to synchronize demand-compilation within a single process. Future versions of SCIRun will operate in a distributed-memory parallel environment, which will require that multiple processors synchronize to avoid race conditions when generating code on a single shared filesystem. In this case, the locking mechanism mentioned above will be extended to use filesystem-based locks.

Summary

We have provided a mechanism for compiling only the template instantiations that are needed as opposed to compiling all possible combinations of instantiations. This solution minimizes the biggest problem with using template code, namely bloat: compiling all possible combinations of template code, increases total space and compilation time requirements. This reason has been enough to overshadow the benefits that templates provide in generality and execution time. The deferred compilation scheme makes the use of templates practical for an interactive, general purpose system such as SCIRun. This mechanism also allows SCIRun modules to operate on data types that it knows nothing about at the time the module is compiled.

References

1. S.G. Parker, D.M. Beazley, and C.R. Johnson. Computational steering software systems and strategies. *IEEE Computational Science and Engineering*, 4(4):50–59, 1997.
2. Steven Gregory Parker. *The SCIRun Problem Solving Environment and Computational Steering Software System*. PhD thesis, University of Utah, 1999.
3. S.G. Parker and C.R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Supercomputing '95*. IEEE Press, 1995.
4. S.G. Parker, D.M. Weinstein, and C.R. Johnson. The SCIRun computational steering software system. In E. Arge, A.M. Bruaset, and H.P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 1–44. Birkhauser Press, 1997.
5. P.J. Morgan and C. Henze. Large field visualization with demand-driven calculation. In *Visualization '99*. IEEE Press, 1999.
6. Todd L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Berlin, Heidelberg, New York, Tokyo, 1997. Springer-Verlag.
7. S. Haney, J. Crotinger, S. Karmesin, and S. Smith. Pete: Portable expression template engine, 1998.

8. E. Johnson and D. Gannon. Programming with the hpc++ parallel standard template library, 1997.
9. Andrew Kennedy and Don. Syme. Design and implementation of generics for the .net common language runtime. In *Programming Language Design and Implementation*, pages 1–12. ACM Press, 2001.
10. S. Atlas, S. Banerjee, J. Cummings, P. Hinker, M. Srikant, J. Reynders, and M. Tholburn. Pooma: A high performance distributed simulation environment for scientic applications, 1995.

Martin Cole

Martin Cole is the Software Manager for the BioPSE development effort. BioPSE is a software tool built within the SCIRun Software System, for the purpose of bioelectric field modeling, simulation, and visualization. He received his B.S. in Computer Science in 1994, and went on to work for Parametric Technology Corporation, specifically on the 3DPaint and CDRS Software systems.

Steven Parker

Steven Parker is a Research Assistant Professor in Scientific Computing and Imaging (SCI) Institute in the School of Computing at the University of Utah. His research focuses on problem solving environments, which tie together scientific computing, scientific visualization, and computer graphics. He is the principal architect of the SCIRun Software System, which formed the core of his Ph.D. dissertation, and is currently the chief architect of Uintah, a software system designed to simulate accidental fires and explosions using thousands of processors. He was a recipient of the Computational Science Graduate Fellowship from the Department of Energy. He received a B.S. in Electrical Engineering from the University of Oklahoma in 1992, and a Ph.D. from the University Utah in 1999.

OoLaLa: Transformations for Implementations of Matrix Operations at High Abstraction Levels

Mikel Luján*, John R. Gurd, and T.L. Freeman

Centre for Novel Computing, University of Manchester
Oxford Road, Manchester M13 9PL, United Kingdom
{mlujan, jgurd, lfreeman}@cs.man.ac.uk

Abstract. OoLaLa is an object oriented linear algebra library designed to reduce the effort of software development and maintenance. In contrast with traditional (Fortran-based) libraries, it provides two high abstraction levels that significantly reduce the number of implementations necessary for particular linear algebra operations.
Initial performance evaluations of a Java implementation of OoLaLa show that the two high abstraction levels are not competitive with the low abstraction level of traditional libraries. One of the high abstraction levels is consistently slower than the low abstraction, while the other is slower in most cases.
These performance results motivate the present contribution – the characterisation of the set of storage formats (data structures) and matrix properties (special features) for which a set of standard transformations are able to map implementations at the two high abstraction levels into efficient implementations at the low abstraction level.

1 Introduction

Object oriented software construction can be used to simplify the interface of numerical linear algebra libraries and thus make them easier to use. It can also be used to arrive at several fundamentally different implementations of linear algebra operations. The objective is to reduce the effort of developing and maintaining these libraries; thus a suitable design should significantly reduce the number of implementations necessary for each particular linear algebra operation, compared with the large numbers encountered in traditional, Fortran based, libraries. OoLaLa [30,31] is a novel Object Oriented Linear Algebra LibrAry which embodies such a design.

In contrast with traditional linear algebra libraries, OoLaLa provides two higher abstraction levels at which matrix operations are implemented. Traditional libraries sacrifice abstraction (when provided by the programming language) as a trade-off for performance. In these libraries, the implementations of matrix operations have embedded knowledge about the data structures (*storage*

* ML acknowledges the support of a research scholarship from the Department of Education, Universities and Research of the Basque Government.

formats) and special characteristics (*matrix properties*) of the matrices passed in as parameters. These implementations are said to be at *Storage Format Abstraction* level (SFA-level). OoLaLa adopts the opposite approach; abstraction first, then performance. The first higher abstraction level, *Matrix Abstraction* level (MA-level), provides random access to matrix elements. The second abstraction level, *Iterator Abstraction* level (IA-level) (based on the iterator pattern [19]), provides sequential access to matrix elements. Both abstraction levels reduce the number of implementations for any given matrix operation [31].

Initial performance evaluations of a Java implementation of OoLaLa show that implementations at MA-level and IA-level are not competitive with those at SFA-level. This motivates the two questions addressed in this paper:

- how can implementations of matrix operations at MA-level and IA-level be transformed into efficient implementations at SFA-level? and
- under what conditions? i.e. for which sets of storage formats and matrix properties can this be done automatically?

These questions are addressed as follows. From the perspective of a library developer Section 2 describes part of OoLaLa and its implementation in Java. Section 3 presents a performance evaluation of the Java implementation of OoLaLa which compares implementations at the three abstraction levels for norm1 and matrix-matrix multiplication. Section 4 characterises (defines) subsets of storage formats and matrix properties, and illustrates different sequences of standard optimising transformations for the defined subsets. These sequences transform implementations at MA-level and IA-level into efficient implementations at SFA-level. Note that cache optimising transformations to obtain blocked [14, 17] or recursive implementations [24, 4] are not included in these sequences. Those transformations are not included because previous work (IBM's Ninja group [35]) has already enabled compilers to apply these transformations to implementations at SFA-level. Related work and conclusions are presented in Sections 5 and 6 respectively.

2 Overview of OoLaLa

The design of OoLaLa covers issues ranging from the representation of matrices (matrix properties and storage formats) to the representation of matrix operations. It also includes abstraction levels for implementing these operations, and enables matrices to be described as compositions or sections of other matrices. This section presents an overview of the Java implementation of OoLaLa and describes it from the perspective of a library developer. A more complete description of OoLaLa appears in [30, 31].

The generalised class diagram[1] of OoLaLa (see Figure 1) can be read as – "a given matrix can have different matrix properties and, as a function of these

[1] UML class diagram where the syntax to define attributes and methods has been relaxed and replaced by pseudo code.

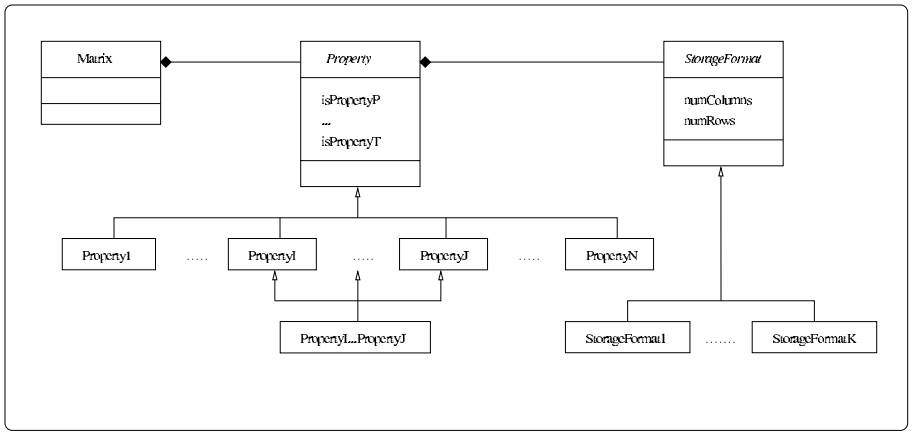

Fig. 1. Generalised class diagram of OoLaLa.

properties, can be represented in different storage formats". The matrix properties (represented as attributes of class `Property` or as subclasses of `Property`) and storage formats (represented as subclasses of `StorageFormat`) are not fixed. This means that, when operated on, the properties and storage format of an instance of class `Matrix` can be modified.

Since multiple inheritance is not included in Java, OoLaLa modifies its class diagram and represents each matrix property, that has a multiple inheritance relation in Figure 1, as a `final` subclass of `Property`. Every class and method is either `abstract`or `final`.Ideally, generic classes would be used so as to develop only one version of OoLaLa, independent of the data type of the matrix elements. But, given that Java does not currently support generic classes, that the official plans[2] to incorporate generic classes do not support primitive types (`float`, `double`, `int`, etc.), and that emulating generic classes with inheritance ([33] App. B.4) delivers poor performance [12, 9], OoLaLa is implemented by developing a version for each numerical data type. OoLaLa represents two-dimensional arrays by mapping them to one-dimensional arrays in a column-wise form (as in the Array Package [34] and JLAPACK [9]). In this way, a two-dimensional array is stored continuously by columns in memory (as in Fortran) and the number of exception tests (array index out of bounds and null object) is halved.

The operation norm1 ($||A||_1$) is used to illustrate and understand the differences among implementations at the three abstraction levels. The implementations presented illustrate the final phase into which OoLaLa divides a matrix operation. Before executing the implementations presented, OoLaLa has checked the correctness of the parameters (e.g. null references, coherent matrix

[2] The specification of the JSR-014 Adding Generics to the Java Programming Language is available at http://jcp.org/aboutJava/communityprocess/review/jsr014

dimensions, etc.), predicted matrix properties (and, if necessary, modified matrix properties or storage formats), and selected the appropriate implementation for the matrix operation. This paper does not describe these three phases, but they appear in [30].

Figure 2 presents the storage formats and matrix properties used in the examples. The bottom three storage formats reduce the memory requirements by not storing the zeros in the matrix. For example, the packed format uses an one-dimensional array to consecutively store in column order the nonzero elements of the matrix. The coordinate format requires three arrays; two of integers and the other of the same data type that the matrix elements. The two arrays of integers store the indices and the third array the associated value of the matrix element. The band format uses a two-dimensional array but the number of rows is the bandwidth of the matrix. Figure 3 presents definition of $\|A\|_1$, used in the subsequent examples.

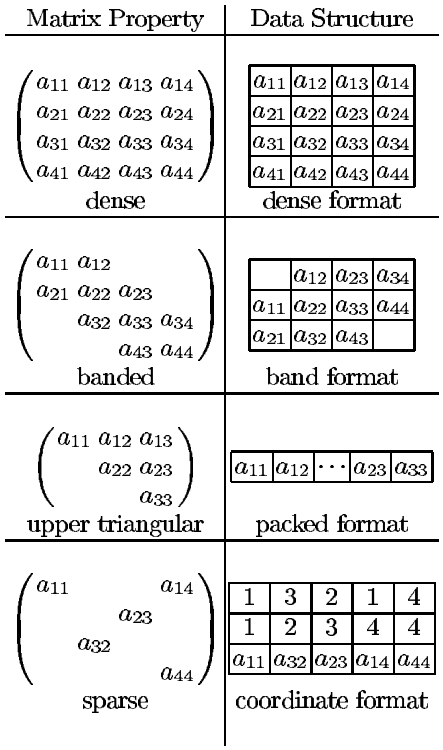

Fig. 2. Examples of matrix properties and storage formats.

```
norm1 ← 0
for j = 1 to n
  aux ← 0
  for i = 1 to m
    aux ← aux + |a_ij|
  end for
  norm1 ← max(aux,norm1)
end for
```

Fig. 3. Algorithm of $||A||_1$.

2.1 Storage Format Abstraction Level

An implementation is said to be at SFA-level if changing the storage format of any of the parameters implies that the implementation is invalid. These implementations know the representations of the storage formats and make explicit usage of this information in order to access elements and to achieve good performance.

```
public static double denseNorm1 (double a[], int m, int n)
{                                // a[m][n]
  int ind=0;
  double sum;
  double max=0.0;

  for (int j=0; j<n; j++)
  {
    sum=0.0;
    for (int i=0; i<m; i++)
    {
      sum+=Math.abs(a[ind]); // a[i][j]
      ind++;
    }// end for
    if (sum>max) {max=sum;}
  }// end for
  return max;
}
```

Fig. 4. Implementation of $||A||_1$ at SFA-level where A is a dense matrix stored in dense format.

Figure 4 presents an implementation of $||A||_1$ where A is a dense matrix stored in dense format[3], while Figure 5 presents an equivalent implementation where A is an upper triangular matrix stored in packed format. The main difference is the inner loop (i-loop) which is shortened because it is known that when `i>j` the matrix elements are zeros and, therefore, these iterations are redundant.

[3] Recall that one-dimensional arrays are used instead of a two-dimensional arrays and that the mapping is, as defined in Fortran, by columns.

```
public static double upperNorm1 (double aPacked[], int m, int n)
{
  int ind=0;
  double sum;
  double max=0.0;

  for (int j=0; j<n; j++)
  {
    sum=0.0;
    for (int i=0; i<=j; i++)
    {
      sum+=Math.abs(aPacked[ind]);
      ind++;
    }// end for
    if (sum>max){max=sum;}
  }// end for
  return max;
}
```

Fig. 5. Implementations of $||A||_1$ at SFA-level where A is an upper triangular matrix stored in packed format.

When, for example, A is an upper triangular matrix stored in dense format, the implementation in Figure 5 does not compute $||A||_1$ correctly. On the other hand, the implementation in Figure 4 computes $||A||_1$ correctly, but it is highly inefficient.

Implementations at SFA-level do not take any advantage of the class structure of OoLaLa. They simply import the techniques to implement and obtain high performance from traditional libraries.

2.2 Matrix Abstraction Level

The MA-level resembles the mathematical definition of a matrix; it provides random access to matrix elements. From a programming point of view, a matrix is a two-dimensional container of numbers. The basic operations are to obtain an `element` of the matrix and to `assign` a value to an element of the matrix. An element is determined by its (unique) position: number of row `i` and column `j`. The two access methods, `element(i,j)` and `assign(i,j,value)`, constitute the MA-level interface.

`Matrix`, `Property` and `StorageFormat` provide these two methods. `Matrix` implements the methods by delegating to its attribute that is an instance of a subclass of `Property`. The subclasses of `Property`[4] resolve the elements that are known due to the specific matrix property that they represent. Otherwise, each subclass of `Property` delegates to its attribute that is an instance of a subclass of `StorageFormat`[5]. Subclasses of `StorageFormat` return the element specified by two integers, or throw an exception. The exception is thrown by sparse storage

[4] `Property` is an `abstract` class, and both `element` and `assign` are `abstract` methods.

[5] `StorageFormat` is also an `abstract` class due to `element` and `assign` being `abstract` methods.

formats when the element is not found. In this way, the subclasses of `StorageFormat` do not decide what value the matrix element has when it is not stored. This increases their reusability by the different subclasses of `Property`. Instead, by catching the exception, each subclass of `Property` determines the value of the matrix element in accordance with the matrix property that it models.

```
public static double norm1 (DenseProperty a)
{
  double sum;
  double max=0.0;
  int numColumns=a.numColumns();
  int numRows=a.numRows();

  for (int j=1; j<=numColumns; j++)
  {
    sum=0.0;
    for (int i=1; i<=numRows; i++)
    {
      sum+=Math.abs(a.element(i,j));
    }// end for
    if (sum>max){max=sum;}
  }// end for
  return max;
}
```

Fig. 6. Implementations of $||A||_1$ at MA-level where A is a dense matrix.

```
public static double norm1 (UpperTriangularProperty a)
{
  double sum;
  double max=0.0;
  int numColumns=a.numColumns();
  int numRows=a.numRows();

  for (int j=1; j<=numColumns; j++)
  {
    sum=0.0;
    for (int i=1; i<=j; i++)
    {
      sum+=Math.abs(a.element(i,j));
    }// end for
    if (sum>max){max=sum;}
  }// end for
  return max;
}
```

Fig. 7. Implementations of $||A||_1$ at MA-level where A is an upper triangular matrix.

Figures 6 and 7 present the implementations of $||A||_1$ at MA-level, where A is a dense matrix and A is an upper triangular matrix, respectively. Implementations at MA-level are independent of the storage format, but dependent on matrix properties.

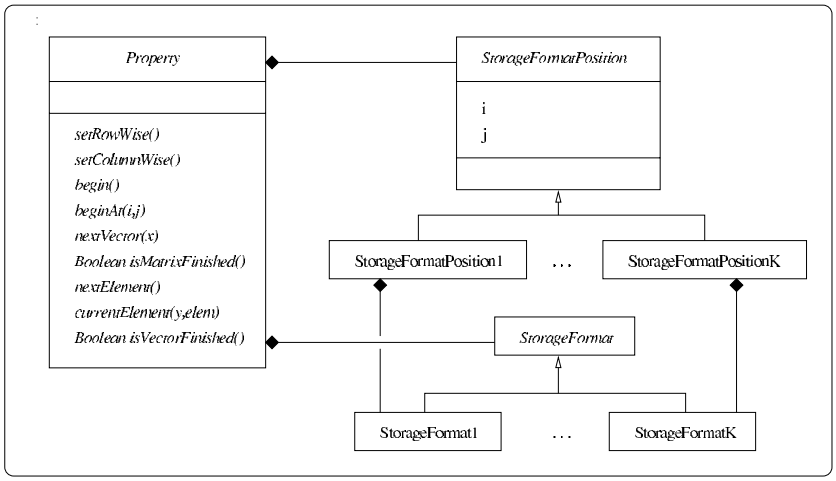

Fig. 8. Class diagram for IA-level.

Note that implementations at MA-level, as with implementations at SFA-level, the loops that traverse the matrix operands are for-loops of the form

```
for (index=L; index<=U; index+=S),
```

where `L` (lower bound), `U` (upper bound), and `S` (stride) are constant integer expressions. In other words, they are classic Fortran do-loops. Hereafter, *for-loop* is used to refer to loops of this form.

2.3 Iterator Abstraction Level

The iterator pattern presents a way to traverse different kinds of containers using a unique interface. The iterator pattern as described by Gamma *et al.* [19] traverses and accesses sequentially the elements in any container. The methods of an iterator (a) set it to a starting position in the container, (b) test if there are any more elements to be accessed, (c) advance one position, and (d) return the current element.

Figure 8 gives a class diagram for `Property` with the set of methods that constitute the IA-level; an adapted iterator for two-dimensional containers. `Matrix` provides the same methods, but their implementations delegate to its attribute that is an instance of a subclass of `Property`. The methods `nextElement` and `nextVector` are defined so that only nonzero elements are accessed. The figure also presents `StorageFormatPosition`; the class that holds the current position.

Figure 9 presents the implementation of $||A||_1$ at IA-level. Implementations at IA-level are independent of storage formats and of those matrix properties based on structures of nonzero elements. Depending on the matrix properties of the instance `a`, `currentElement`, `nextVector` and `nextElement` access different

matrix elements. Implementations at IA-level implicitly determine the elements to be accessed, while implementations at MA-level make this explicit.

```
public static double norm1 (Property a)
{
  double sum;
  double max=0.0;

  a.setColumnWise();
  a.begin();
  while (!a.isMatrixFinished())
  {
    sum=0.0;
    a.nextVector();
    while (!a.isVectorFinished())
    {
      a.nextElement();
      sum+=Math.abs(a.currentElement());
    }// end while
    if (sum>max) {max=sum;}
  }// end while
  return max;
}
```

Fig. 9. Implementation of $||A||_1$ at IA-level.

3 Performance Evaluation

The objective of the experiments reported below is to investigate the relative performances delivered by each of the three abstraction levels. In some cases, performance is compared with the equivalent Fortran 77.

The test cases used for the experiments are norm1 ($||A||_1$) and matrix-matrix multiplication ($C = AB$). The test cases use the `double` data type. The following combinations of matrix properties and storage formats have been implemented in Java: dense matrix in dense format (dp-df), banded matrix in dense format (bp-df), banded matrix in band format (bp-bf), upper triangular matrix in dense format (up-df), upper triangular matrix in packed format (up-pf).

The storage formats are organised column-wise and the implemented algorithms traverse matrices column-wise (as in the Fortran BLAS downloaded from `www.netlib.org`). The matrices used in the experiments for $C = AB$ are square matrices of dimensions 200×200, 400×400, 600×600, 800×800 and 1000×1000, and only the last three for $||A||_1$. The upper bandwidth and lower bandwidth of the banded matrices are one quarter of the matrix dimension.

The machine on which the tests were executed is a Sun Ultra-5 at 333 Mhz with 256 Mb of memory and Solaris 5.8. It runs Sun's Java SDK 2 Standard Edition version 1.3.1. The Fortran 77 compiler is the Sun Workshop version 5.0.

The timing results have an accuracy of milliseconds and each value is the minimum time observed after four invocations of the methods that implement the matrix operations.

The JVM was invoked with the standard `-server` and non-standard flags `-Xbatch -Xms64Mb -Xmx128Mb`. The compiler javac was invoked with the `-O` flag and the Fortran compiler with the `-O2` flag; the `-O2` flag enables basic block optimisations, but not loop optimisations (unrolling, interchange, etc.). This was chosen to make a "fair" comparison with current Java technology which is constrained by the strict Java exception model and may not apply such optimisations.

Table 1 contains the execution times for $C = AB$ with dense matrices. It includes figures for both Fortran and Java implementations. Figure 10 presents the ratios between the execution times of the Java implementations. The following observations can be made:

- for big matrices (800-1000), SFA-level is faster than non-optimised and competitive with `-O2` Fortran implementations;
- for small matrices (200-600), SFA-level outperforms the non-optimised Fortran, by a factor of about 3, but is less efficient than `-O2` Fortran;
- IA-level is consistently and significantly slower than SFA-level; the slow-down factor is between 7 and 14 for $||A||_1$ and between 5.3 and 13.6 for $C = AB$;
- MA-level is between 3 and 7 times slower than SFA-level for $||A||_1$ and $C = AB$, except for $||A||_1$ and $C = AB$ with de-df, and $||A||_1$ with up-df. Tables 2 and 3 present the details of these exceptions.

The next section explores the reasons for the poor performance of IA- and MA-level compared with SFA-level.

	Fortran	Fortran -O2	Java SFA-level
200	1.330	0.357	0.367
400	10.666	2.841	3.835
600	40.453	11.656	12.54
800	127.367	32.439	31.079
1000	184.131	64.236	60.89

Table 1. Times in seconds for $C = AB$ with dense matrices on the Sun Ultra-5.

	SFA-level	MA-level
200	0.367	0.416
400	3.835	4.264
600	12.54	14.310
800	31.079	34.787
1000	60.89	71.121

Table 2. Times in seconds for $C = AB$ at MA-level and SFA-level for the case de-df.

	SFA-level (de-df)	MA-level (de-df)
600	0.017	0.017
800	0.031	0.030
1000	0.049	0.046

	SFA-level (up-df)	MA-level (up-df)
600	0.009	0.011
800	0.016	0.020
1000	0.025	0.030

Table 3. Times in seconds for $\|A\|_1$ at MA-level and SFA-level for the cases de-df and up-df.

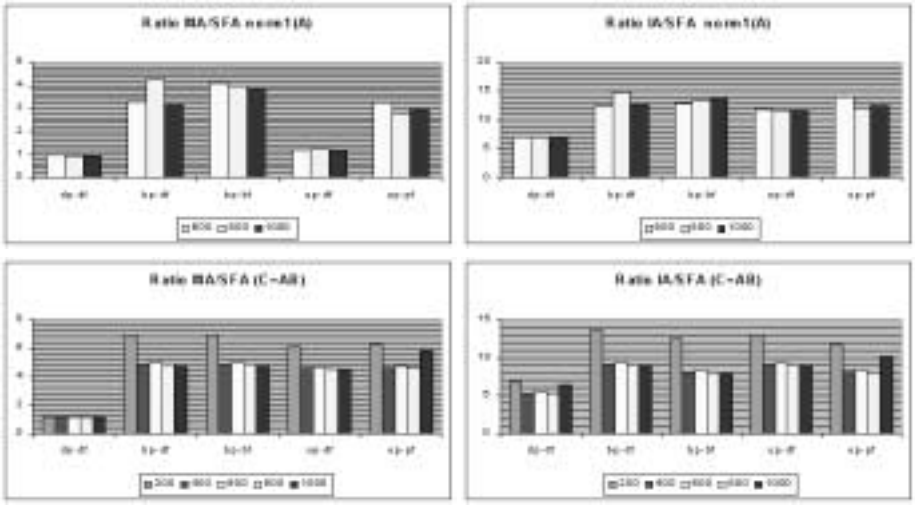

Fig. 10. Ratios for $C = AB$.

4 Transforming Implementations of Matrix Operations

The experimental results of Section 3 provide a motive to search for circumstances in which it is possible to automatically transform code at MA- or IA-level into efficient code at SFA-level. This section describes how known transformation techniques can be applied systematically in certain circumstances to achieve the desired effect.

Section 4.1 delimits the extent of the transformations that a *general purpose compiler* might apply. It also defines specific subsets of matrix properties and storage formats which are sufficient to permit the automatic transformation of implementations at MA- and IA-level into efficient implementations at SFA-

level. Section 4.2 presents the transformations for MA-level implementations, while Section 4.3 presents the transformations for IA-level implementations.

The classes used as examples are summarised in the sequence diagrams of Figures 11, 14 and 22.

4.1 Preliminaries

An aspect to clarify first, is the subdivision of matrix properties. A *matrix property* is a characteristic of certain matrices that has been identified because (a) it appears in real world applications and (b) the implementations of certain matrix operations can exploit it. The implementations might exploit matrix properties to reduce execution time, to reduce memory requirements, or to increase the accuracy.

A subset of these matrix properties, namely the *mathematical relation* properties, are properties of the whole matrix, rather than of the individual elements of the matrix. Examples of mathematical relation properties are symmetry (A=A^T), and orthogonality (A^T=A^{-1} or $AA^T = I$); see [20] for more examples and rigorous formulation of such properties. Mathematical relation properties enable linear algebra researchers to formulate different algorithms for a given matrix operation.

A second subset of matrix properties, namely *nonzero elements structure* properties, is based on the structure of the nonzero matrix elements. Examples of this subset are upper triangular property, banded property, block diagonal property and sparse property, [20]. This latter group of matrix properties specialises algorithms by elimininating steps of the algorithm which are known to be redundant or unnecessary (e.g. `x=x+zero` and `y=y*one`).

The Role of a General Purpose Compiler General purpose static and dynamic compilers could transform implementations following the nonzero elements structure specialisation. However, such compilers cannot transform implementations using the mathematical relation properties; this is the reserve of domain-specific compilers. In other words, a general purpose compiler could specialise the implementation of $||A||_1$ where A is a dense matrix to the case where A is an upper triangular matrix. But, it cannot transform LU-factorisation (an algorithm used for solving systems of linear equations $Ax = b$, where A is a general matrix) into Cholesky-factorisation (another algorithm used for solving the same problem, but only applicable when A is a symmetric, positive definite matrix).

Definitions We now define (a) the subset of matrix properties *linear combination matrix properties* (LCMP), (b) the subset of storage formats *constant time element access storage formats* (CTSF), and (c) the subset of matrices *linear combination and constant time* (LCCT) used in the generalisation of the transformations (Sections 4.2 and 4.3).

The set LCMP is defined as the subset of matrix properties such that every matrix property is based on a boolean expression involving linear combinations of the indices i and j to determine whether an element a_{ij} might be a nonzero element. An example of a LCMP is the upper triangular property: a_{ij} is zero when $i > j$ and might be nonzero when $i <= j$. The general sparse property and the orthogonality property are examples of matrix properties that are not LCMP.

The set CTSF is defined as the subset of matrix storage formats such that are based on arrays and that every valid and stored matrix element can be accessed using an algorithm of complexity $O(1)$. An example of an CTSF is the dense (or conventional) storage format: a $m \times n$ matrix A is stored in a two-dimensional array of the same dimensions so that an element a_{ij} is stored in `a[i-1][j-1]`. Coordinate (an example is shown in Figure 2) is an example of storage formats that are not CTSF.

The set LCCT is defined as the subset of matrices such that their properties belong to LCMP and they are stored in a storage format member of CTSF.

4.2 Matrix Abstraction Level

Dense Case Figure 11 presents the sequence diagram for the execution of the implementation of $||A||_1$ at MA-level when A is dense and stored in dense format (see also Figure 6). Compared with the equivalent implementation at SFA-level (see Figure 4), the main source of overhead at MA-level is the dynamic binding of `a.element(i,j)`.

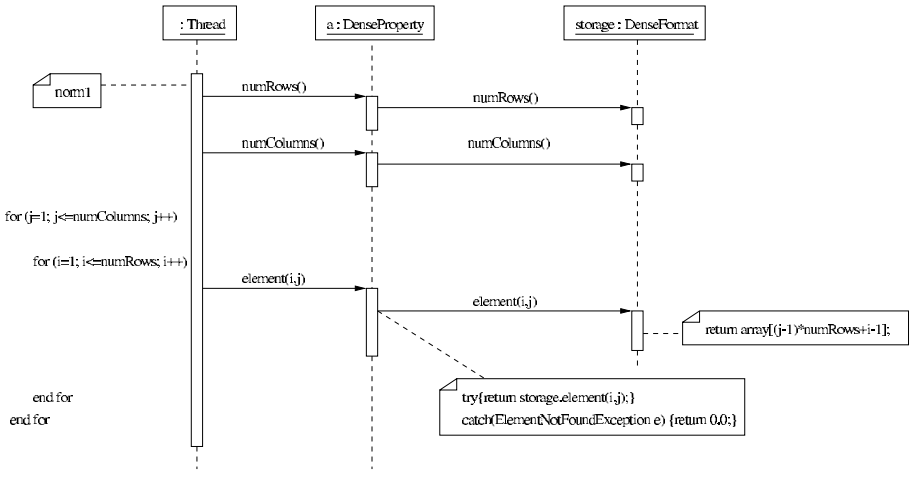

Fig. 11. Sequence diagram for $||A||_1$ implemented at MA-level with A dense and stored in dense format.

Method inlining [13, 21, 15, 18, 2, 6, 16, 25, 37] can be applied to eliminate the invocations by inserting the code of the invoked methods into the invoking methods. After applying method inlining,[6], the statements inside the nested loop appear as shown in Figure 12.

```
if (a instanceOf DenseProperty)
{// first guard
  double aux;
  try
  {
    if (a.storage instanceof DenseFormat)
    // second guard
    {aux=a.storage.array[(j-1)*a.storage.numRows+i-1];}
    else
    {aux=a.storage.element(i,j);}
  }
  catch(ElementNotFoundException e){aux=0.0;}
  sum+=Math.abs(aux);
}
else
{sum+=Math.abs(a.element(i,j);}
```

Fig. 12. Statements inside the nested loop after applying method inlining to the code in Figure 6.

An analysis of the loops reveals that, for every iteration, `a` and `a.storage` are instances of `DenseProperty` and of `DenseFormat`, respectively. It also reveals that none of the statements inside the `try` clause can `throw` an instance of `ElementNotFoundException`.[7] Under these circumstances,

(a) the first guard can be removed because `a` is a parameter of the `final` class `DenseProperty`,
(b) the second guard for the inlined methods can be placed surrounding the nested loops, and
(c) the `try-catch` can be removed leaving the code shown in Figure 13.

Once these optimisations have been applied, the code obtained is almost identical to the hand-written implementation at SFA-level (see Figure 4). The only difference is the index for accessing `array`. The implementation at SFA-level uses an index, `ind`, that is initialised to zero and increases in each iteration. The compiler optimisation technique known as *strength reduction* [5] is able to transform[8] `(j-1)*a.storage.numRows+i-1` to achieve the implementation at SFA-level.

[6] We present method inlining with guarded class tests for the sake of clarity and because, although it is not the most efficient [16, 25] it generates the most general code.

[7] `ElementNotFoundException` is a subclass of `Exception` defined by OoLaLa. Instances of this are thrown by `element(i,j)` in certain subclasses of `StorageFormat` (sparse storage formats) when the matrix element, a_{ij}, is not found.

[8] Note that the upper bound of the i-loop, `numRows`, is a local variable initialised to `a.numRows()`, i.e. `a.storage.numRows` after method inlining.

```
if (a.storage instanceof DenseFormat)
{
  double aux;
  for (j=1; j<=numColumns; j++)
  {
    sum=0.0;
    for (i=1; i<=numRows; i++)
    {
      aux=a.storage.array[(j-1)*a.storage.numRows+i-1];
      sum+=Math.abs(aux);}
    if (sum>max){max=sum;}
  }// end for
}// end then
else
{
  // original implementation
}// end if
```

Fig. 13. Statements inside the nested loop after removing the guards and the `try-catch` clause from the code in Figure 12.

Upper Triangular Case When considering the implementation of $||A||_1$ with A upper triangular and stored in packed format, the main source of overhead is again the dynamic binding (see the sequence diagram in Figure 14). Figure 15 presents the body of the inner loop resulting from applying method inlining to the code in Figure 7.

Applying the same optimisations as in the dense case, the `try-catch` clause can be removed leaving only the statements inside `try{...}`, the first guard can also be removed, and the second guard can be moved to surround the loops.

This upper triangular case introduces an `if-then-else` structure in the nested loops. The inner loop invariant expresses that `i <= j` which is exactly the condition of the `if` clause. Hence, the condition always evaluates to `true` and can be substituted with its true-branch. The resulting code, again, is almost identical to the hand-written code at SFA-level (see Figure 5), except for the index for `array`. Again this difference can be overcome by applying strength reduction.

Generalisation With the given definitions, the following paragraphs argue that it is possible to transform the implementation of a matrix operation at MA-level so that the code is similar to the implementation at SFA-level provided that its operands are members of LCCT.

Given a general form of a matrix operation at MA-level (see Figure 16), any invocation of the method `element` in an instance of a subclass of `Property` can be inlined, since OoLaLa's design guarantees that every class is `abstract` or `final`. For simplicity `PropertyX` is assumed to be a `final` class. Method inlining generates the code in Figure 17. This code has first an `if` clause to guard the inlined statements of `x.element(i,j)`. The inlined statements are only executed when the guard is `true`. When this guard is `false`, the implementation will execute the original invocation.

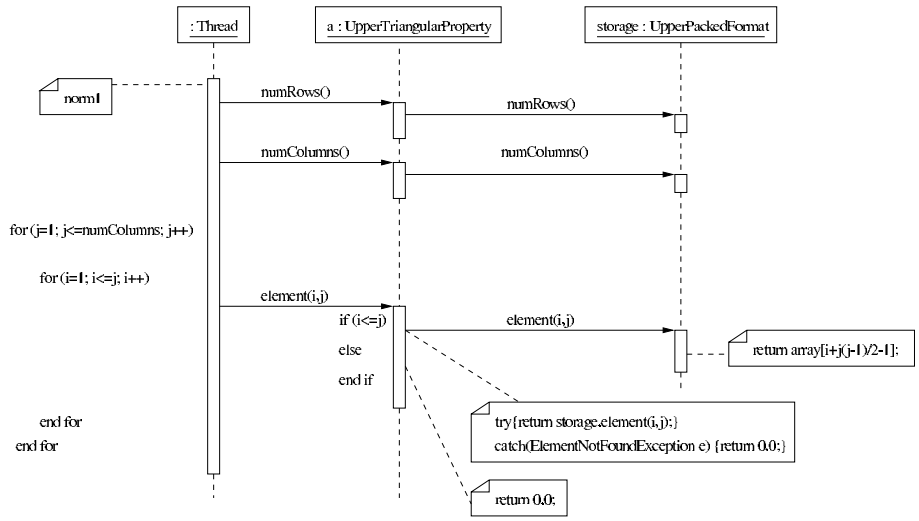

Fig. 14. Sequence diagram for $||A||_1$ implemented at MA-level with A upper triangular and stored in packed format.

```
if (a instanceof UpperTriangularProperty)
{// first guard
  double aux;
  try
  {
    if (i<=j)
    {
      if (a.storage instanceOf UpperPackedFormat)
      // second guard
      {aux=a.storage.array[i+j*(j-1)/2-1];}
      else
      {aux=a.storage.element(i,j);}
    }
    else {aux=0.0;}
  }
  catch(ElementNotFoundException e){aux=0.0;}
  sum+=Math.abs(aux);
}// end then
else
{
  sum+=Math.abs(a.element(i,j));
}// end if
```

Fig. 15. The body of the inner loop resulting from applying method inlining in Figure 7.

Since `PropertyX` represents a property in LCMP, the inlined statements contain an `if (condition)` which is a boolean expression involving linear combinations of `i` and `j`. The `condition` determines when the element is known due to the LCMP or `PropertyX` delegates to `x.storage`. A `try-catch` clause surrounds this invocation; the signature declares that it may throw an instance of

```
ClassOrPrimitiveDataType matrixOperation(..,PropertyX x,..)
{
  ...
    x.element(i,j);
  ...
}
```

Fig. 16. General form of matrix operations implemented at MA-level.

`ElementNotFoundException`. Method inlining is also applied to this invocation and the statements together with the guard appear inside the `try{...}`. Because `StorageFormatY` is a storage format in CTSF, the inlined statements implement an algorithm of $O(1)$ complexity.

```
ClassOrPrimitiveDataType matrixOperation(..,PropertyX x,..)
{
  ...
    double aux;
    if (x instanceof PropertyX) // first guard
    {
      if (condition) // linear combination using i and j
      {
        try
        {
          if (x.storage instanceof StorageFormatY)
          // second guard
          {aux= ...;} // order one algorithm
          else
          {aux=x.storage.element(i,j);}
        }
        catch (ElementNotFoundException e){aux=0.0;}
      }
      else
      {aux=0.0;}
    }
    else
    {aux=x.element(i,j);}
  ...
}
```

Fig. 17. General form of matrix operations implemented at MA-level after method inlining.

The second step in optimisation is to relocate the `try-catch` clause. Every statement that can throw an instance of `ElementNotFoundException` is inside the `try{...}`. Otherwise, the signature of the method `element` in the subclasses of `Property` should declare it; OoLaLa's design avoids this. Each of the statements that `throws` the exception, except for `x.storage.element(i,j)`, can be substituted with the statement (or statements) inside the `catch{...}`, as long as the flow of the program is semantically equivalent.[9] Once the `throws` statements have been removed, only the else-branch of the second guard can throw

[9] For example, the flow of the program can be maintained using Java labelled `if`'s and `break`'s. The `if (condition)` has a label `exception` and just after the state-

an instance of `ElementNotFoundException`. Hence, the `try-catch` clause can be moved inside the else-branch (see Figure 18).

```
...
  if (condition) // linear combination of i and j
  {
    if (x.storage instanceof StorageFormatY)
    // second guard
    {
      aux= ...;
      // order one algorithm without throws statements
    }
    else
    {
      try {aux=x.storage.element(i,j);}
      catch (ElementNotFoundException e){aux=0.0;}
    }
  }
  else
  {aux=0.0;}
...
```

Fig. 18. General form of matrix operations implemented at MA-level after applying method inlining and moving the `try-catch` clause.

Because OoLaLa divides a matrix operation into four phases and specifies an order for them, `x` and `x.storage` remain, throughout the execution of the method, instances of `PropertyX` and `StorageFormatY`, respectively. Thus, an analysis of the method should reveal this and enable compilers to place the guards around the code. The resulting code is presented in Figure 19. The remaining `try-catch` clause is removed when the guards are moved.

The next step in optimisation is to remove `if (condition)`. When this `if` clause is not in a loop,[10] the performance overhead is negligible. However, when the `if` is repeatedly evaluated, it can become a performance problem. The simplest case is when the statements in the loop do not modify the values `i` and `j` and, thus, the `condition` always evaluates to the same value. Because of this, *loop unswitching* [5] can split the loop into two; one with the true-branch and the other with the false-branch. The `condition` is evaluated only once in an `if` which has as true-branch the first loop and as false-branch the second loop.

The more general case, when the statements in the loop modify the values `i` and `j`, may be addressed by applying first *induction variable elimination* [5] and then *index set splitting* [5]. The first transformation ensures that `i` and `j` are loop indices, or functions of loop indices, and, therefore, have upper and lower bounds. Then, index set splitting transforms the loop into multiple adjacent loops where each loop performs a subset of the original iterations. These multiple adjacent

ments substituting a `throws new ElementNotFoundException()`, we introduce `break exception;`. This is a brute force approach, but it is always applicable.

[10] A single loop is assumed for the sake of clarity. The described transformations are also applicable to nested loops.

```
ClassOrPrimitiveDataType matrixOperation(.., PropertyX x,..)
{
  if (x instanceof PropertyX &&
      x.storage instanceof StorageFormatY)
  {
    ...
      double aux;
      if (condition) // linear combination using i and j
      {
        aux= ...;
        // order one algorithm without throws statements
      }
      else
      {aux=0.0;}
    ...
  }// end then
  else
  {
    // original implementation
  }// end if
}
```

Fig. 19. General form of matrix operations implemented at MA-level after method inlining and removing exceptions `ElementNotFoundException`.

loops are selected so that the iterations performed by each loop evaluates the `condition` to the same value.

```
for (j=l1; j<=u1; j++)
{
  for (i=l2; i<=u2; i++)
  {
    if (i<=j)
    {aux= ...;}
    else
    {aux=0.0;}
  }// end for
}// end for
```

Original code

```
for (j=l1; j<=u1; j++)
{
    for (i=l2; i<=Math.min(j,u2); i++)
    {aux= ...;}
    for (i=Math.min(j+1,u2+1); i<=u2; i++)
    {aux=0.0; }
}// end for
```

Transformed code

Fig. 20. An example of applying index set splitting.

For example, consider the code in Figure 20. The condition `i<=j` is eliminated and the i-loop is divided into two loops. The first new loop performs the iterations for which `i<=j` is `true` and, thus, performs the true-branch of the condition, while the second new loop performs the false-branch. Index set splitting can be applied when the `condition` in the loop is a linear combination of the loop indices; which is the case for matrices in LCCT.

The final step in optimisation is to apply strength reduction to the $O(1)$ algorithm for accessing a matrix element in the storage format.

Discussion In the first case, $||A||_1$ with A dense matrix stored in dense format, no condition due to the matrix property has been encountered. In the second

case, $||A||_1$ with A upper triangular matrix stored in packed format, the condition `i<=j` has been removed because the loop invariant implied that the condition would be always `true`. Thus, index set splitting is not required in either case.

```
...
if (a instanceof UpperTriangularProperty &&
    a.storage instanceof DenseFormat)
{
  for (j=1; j<=numColumns; j++)
  {
    sum=0.0;
    for (i=1; i<=numRows; i++)
    {
      double aux;
      if (i<=j)
      {aux=a.storage[(j-1)*a.storage.numRows+i-1];}
      else
      {aux=0.0;}
      sum+=Math.abs(aux);
    }// end for
    if (sum>max) {max=sum;}
  }// end for
}// end then
else
{
  // original implementation
}// end if
...
```

Fig. 21. Implementation of $||A||_1$ at MA-level where A is an upper triangular matrix stored in dense format using an algorithm for dense matrices. The code has been transformed applying method inlining, the `try-catch` clause has been removed and the guards for the inlined methods have been moved surrounding the loops.

Suppose that the implementation in Figure 6 is changed so that the parameter `a` is of class `Property` and that this implementation is invoked with an upper triangular matrix stored in dense format. Figure 21 presents the code after applying method inlining, removing the `try-catch` clause and moving the guards for the inlined methods outside of the loop. This code can now be transformed using index set splitting (a similar example transformation is presented in Figure 20).

The second loop

```
for (i=Math.min(j+1,numRows+1); i<=numRows; i++)
{
  double aux=0.0;
  sum+=Math.abs(aux);
}// end for
```

can be removed completely, *dead code elimination* [5], since *constant propagation* [5] and then *algebraic simplification* [5] remove the statements inside this loop. Index set splitting, together with constant propagation, algebraic simplification and dead code elimination, are the final step in specialising an implementation

at MA-level of a general algorithm[11] into an implementation at SFA-level for matrices in LCCT.

When the matrices are in LCCT, the optimisations described transform implementations of matrix operations at MA-level into code that resembles the code at SFA-level. The optimisation can be applied both to implementations of specialised algorithms that take into account the matrix properties of the operands and, also, to implementations of general algorithms.

4.3 Iterator Abstraction Level

Upper Triangular Case Figure 22 presents the sequence diagram for the implementation of $||A||_1$ at IA-level where A is an upper triangular matrix stored in dense format (see also Figure 9). Apart from the overhead of the dynamic binding, this implementation suffers the repeated execution of certain computations. The strategy to transform this implementation into an efficient implementation at SFA-level follows these steps:

- inline the methods invoked in `a`, `a.currentPosition`, and `a.storage`,
- move the guards for the inlined methods to surround the statements of the method,
- remove `try-catch` clauses,
- make local copies of the accessed attributes,
- disambiguate aliases,
- remove redundant computations, and
- transform the while-loops into for-loops.

For a representative selection of the methods invoked in `a` Table 4 gives the resulting code after applying method inlining. Since the code becomes verbose when including the inlined statements with guards, the table presents them without the guards. The table also excludes the else-branch for the `if (columnWise)` statements.

The optimisation of moving the guards to surround the code has been described in the context of implementations at MA-level. The circumstances that enable compilers to perform this optimisation are guaranteed by OoLaLa's design and, also apply to implementations at IA-level. OoLaLa's design divides the implementation of any matrix operation into different phases and specifies an order among the phases. Thus, OoLaLa ensures that `a`, `a.storage` and `a.currentPosition` remain, throughout the execution of the method `norm1`, instances of `UpperTriangularProperty`, `DenseFormat` and `DenseFormatPosition`.

Another optimisation described in the context of optimisations at MA-level and needed for this implementation, is the removal of `try-catch` clauses. The method `currentPosition` (see Table 4) does not `throw` any exception. It catches instances of `ElementNotFoundException` and, in accordance with the corresponding property, resolves the value of the element. The optimisation substitutes the statements that `throw new ElementNotFoundException()` with the

[11] An algorithm that considers all matrix operands to be dense matrices.

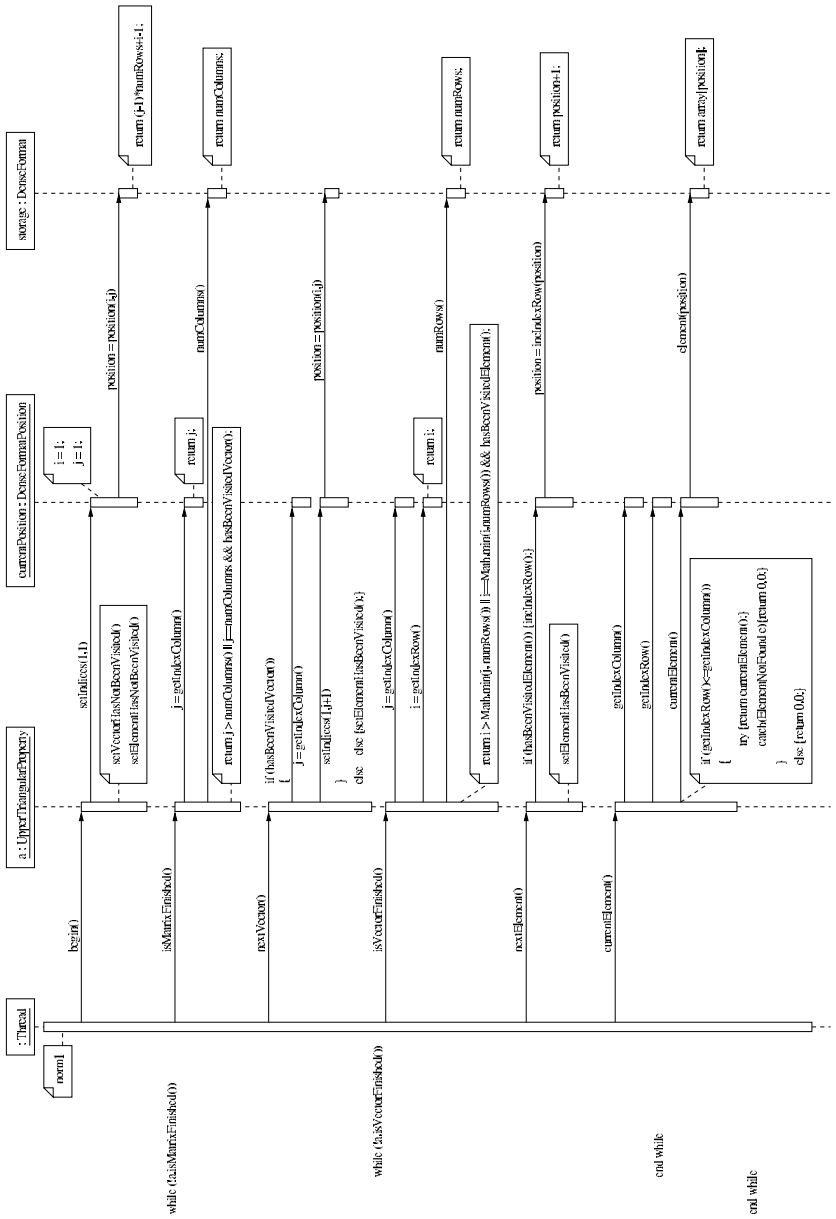

Fig. 22. Sequence diagram for $\|A\|_1$ implemented at IA-level with A upper triangular matrix stored in dense format.

statements inside the `catch{...}` and maintains an equivalent flow of the program.

a.isVectorFinished();	boolean aux;
	DenseFormatPosition currentPosition=a.currentPosition; // R
	int i=currentPosition.i; // R
	int j=currentPosition.j; // R
	DenseFormat storage=a.storage; // R
	int numRowsStorage=storage.numRows; // R
	int numColumnsStorage=storage.numColumns; // R
	boolean columnWise=a.columnWise; // R
	boolean elementHasBeenVisited=a.elementHasBeenVisited; // R
	
	if (columnWise)
	{
	  aux=(i>Math.min(j,numRowsStorage) ||
	  i==Math.min(j,numRowsStorage)) && elementHasBeenVisited;
	}
	else {...aux=...}
	return aux;

a.nextElement();	DenseFormatPosition currentPosition=a.currentPosition; // R
	int i=currentPosition.i; // R
	int j=currentPosition.j; // R
	int position=currentPosition.position; // R
	DenseFormat storage=a.storage; // R
	boolean elementHasBeenVisited=a.elementHasBeenVisited; // R
	boolean columnWise=a.columnWise; // R
	
	if (columnWise)
	{
	  if (elementHasBeenVisited)
	  {
	  i++;
	  currentPosition.i=i;
	  position++;
	  currentPosition.position=position;
	  }// end if
	}
	else {...}
	elementHasBeenVisited=true;
	a.elementHasBeenVisited=elementHasBeenVisited;

a.currentElement();	double aux;
	DenseFormatPosition currentPosition=a.currentPosition; // R
	int i=currentPosition.i; // R
	int j=currentPosition.j; // R
	int position=currentPosition.position; // R
	boolean elementHasBeenVisited=a.elementHasBeenVisited; // R
	
	elementHasBeenVisited=true;
	a.elementHasBeenVisited=elementHasBeenVisited;
	if (i<=j)
	{
	  DenseFormat storageCurrentPosition=currentPosition.storage; // R
	  try {aux=storageCurrentPosition.array[position];}
	  catch (ElementNotFoundException e){aux=0.0;}
	}
	else
	{aux=0.0;}

Table 4. Resulting code after applying method inlining to Figure 9.

Some statements in Table 4 have the line-comment `// R`. This indicates that these statements are repeated, at least once more, among the statements inlined due to the other methods. These statements make local copies of the attributes in `a`, `a.storage` and `a.currentPosition`; this is the third step in optimisation.

The fourth step is to remove these repeated statements by declaring and initialising local copies at the beginning and only writing back to the attributes at the end. This step also removes all tests of the form `if (columnWise)`, leaving only the true-branch. The computations involving `elementHasBeenVisited` can also be removed, but the details are omitted for the sake of clarity.

The fifth step, alias disambiguation, notes that `a.storage` and `a.currentPosition.storage` are both references to the same object. Thus in Figure 23, the local variables `numRowsStorage` and `numRowsCurrentPosition` are copies of the same attribute. Both variables can now be renamed as `numRows`. When considering other matrix operations which involve more than one matrix, alias disambiguation would also be used for the local references to Java arrays.

```
if (a instance of UpperTriangularProperty &&
    a.storage instanceof DenseFormat &&
    a.currentPosition instanceof DenseFomatPosition)
{
  double sum;
  double max=0.0;
  // local copies
  ...

  j=1;
  while (j<=numColumnsStorage)
  {
    sum=0.0;
    i=1;
    position=(j-1)*numRowsCurrentPosition+i-1;
    while (i<=Math.min(j,numRowsStorage))
    {
      double aux;
      if (i<=j) {aux=array[position];}
      else {aux=0.0;}
      sum+=Math.abs(aux);
      i++;
      position++;
    }// end while
    if (sum>max) {max=sum;}
    j++;
  }// end while

  // write back
  ...

  return max;
}// end then
else
{
  // original implementation
}// end else
```

Fig. 23. Implementation of $||A||_1$ at IA-level obtained by applying the optimisation steps 1 to 4.

By now, the structure of the nested while-loops is almost the structure of nested for-loops and standard control-flow and data-flow techniques [1] can be used to recognise this.

Finally, the loop invariant of the inner loop is `i <= j && i <= numRows` which implies that the `if` will always take the true-branch. The resulting code (see Figure 24) is almost identical to the code at SFA-level, except for the statement `position = (j-1) * numRows`. This difference can be eliminated by applying strength reduction.

```
if (a instance of UpperTriangularProperty &&
    a.storage instanceof DenseFormat &&
    a.currentPosition instanceof DenseFomatPosition)
{
  double sum;
  double max=0.0;
  // local copies
  ...

  for(j=1; j<=numColumns; j++)
  {
    sum=0.0;
    position=(j-1)*numRows;
    for (i=1; i<=Math.min(j,numRows); i++)
    {
      sum+=Math.abs(array[position]);
      position++;
    }// end for
    if (sum>max) {max=sum;}
  }// end for

  // write back
  ...

  return max;
}// end then
else
{
  // original implementation
}// end else
```

Fig. 24. Implementation of $||A||_1$ at IA-level obtained by elemininating redundant computations from the code in Figure 23.

Generalisation With the given definitions, the following paragraphs argue that it is possible to transform the implementation of a matrix operation at IA-level so that the resulting code is similar to the implementation at SFA-level provided that the operands belong to LCCT.

Table 5 presents for a selection of the methods that constitute the IA-level interface the effect that method inlining would have. The table assumes that the methods are invoked in an instance `x` of the `final` class `PropertyX` and that `x.storage` is an instance of the `final` class `StorageFormatY`. `PropertyX` represents a LCMP and `StorageFormatY` represents an CTSF; i.e. `x` is a matrix

`x.isVectorFinished();`	`boolean aux;`
	`StorageFormatYPosition currentPosition=x.currentPosition; // R`
	`int i=currentPosition.i; // R`
	`int j=currentPosition.j; // R`
	`StorageFormatY storage=a.storage; // R`
	`int numRowsStorage=storage.numRows; // R`
	`int numColumnsStorage=storage.numColumns; // R`
	`boolean elementHasBeenVisited=a.elementHasBeenVisited; // R`
	
	`aux=conditionVectorFinished(i,j,numRowsStorage,`
	`numColumnsStorage,...);`
	`// linear combination involving i and j and other constant`
	`// characteristics of the matrix`

`a.nextElement();`	`StorageFormatYPosition currentPosition=x.currentPosition; // R`
	`int i=currentPosition.i; // R`
	`int j=currentPosition.j; // R`
	`StorageFormatY storage=x.storage; // R`
	`int numRowsStorage=storage.numRows; // R`
	`int numColumnsStorage=storage.numColumns; // R`
	`boolean elementHasBeenVisited=x.elementHasBeenVisited; // R`
	
	`if (elementHasBeenVisited)`
	`{`
	`i=functionNextElementI(i,j,numRowsStorage,numColumnsStorage,...);`
	`// function derived from a condition LCMP`
	`currentPosition.i=i;`
	`}// end if`
	`elementHasBeenVisited=true;`
	`x.elementHasBeenVisited=elementHasBeenVisited;`

`a.currentElement();`	`double aux;`
	`StorageFormatYPosition currentPosition=x.currentPosition; // R`
	`StorageFormatY storageCurrentPosition=currentPosition.storage; // R`
	`int i=currentPosition.i; // R`
	`int j=currentPosition.j; // R`
	`int position=currentPosition.position; // R`
	`boolean elementHasBeenVisited=a.elementHasBeenVisited; // R`
	
	`elementHasBeenVisited=true;`
	`a.elementHasBeenVisited=elementHasBeenVisited;`
	`if (condition) // linear combination involving i and j`
	`{`
	`try {aux=...;} // order one algorithm`
	`catch (ElementNotFoundException e){aux=value;}`
	`}`
	`else`
	`{aux=value;}`

Table 5. Resulting code after applying method inlining to an implementation of a matrix operation at IA-level.

in LCCT. For simplicity, we consider a column-wise traversal and present only statements for this traversal.

A summary of the information presented in the table follows:

- `isVectorFinished` simply returns the result of a function that takes as parameters constant characteristics of the matrix (e.g. number of rows, number of columns, etc.) and the indices of the current position;
- `nextElement` modifies the indices for the next position using a function that also take as parameters constant characteristics of the matrix (e.g. number of rows, number of columns, etc.) and the indices of the current position;
- `currentElement` implements an $O(1)$ algorithm for finding the element indicated by the current position indices.

The functions, either boolean (for `isVectorFinished`) or integer (for `nextVector` and `nextElement`), are derived from a LCMP condition. In the most simple form, these functions are constants. In the most complex form, they are linear combinations involving the indices of the current position and constant characteristics of the matrix.

The first optimising steps (method inlining for `x`, `x.currentPosition` and `x.storage`, movement of the guards for the inlined statements, removal of `try-catch` clauses, to make local copies of the accessed attributes of `x`, and alias disambiguation) do not present any problems and have been described in previous cases. The description of how to eliminate the computations involving the local variable `elementHasBeenVisited` is omitted for the sake of conciseness.

Figure 25 presents the code after applying the preceding optimisations to the general case. Now, it can be shown that the indices `i` and `j` have lower bounds and upper bounds, since independently of the matrix and traversing column-wise, the condition for `isMatrixFinished()` always includes a term so that `j` is in the range 1 to `numColumns`. A similar argument applies to `i`. Thus, the while-loops can be transformed into a form of for-loops (see Figure 26). Finally, this form of for-loops can be transformed into the traditional form by dividing the iteration spaces so that:

- `conditionMatrixFinished(...)` and `conditionVectorFinished(...)` are simplified to the form `index <= CONSTANT`, and
- `functionNextVectorJ(...)`, `functionNextVectorI(...)`, and `functionNextElementI(...)` are simplified to the form `index +=CONSTANT`.

5 Discussion and Related Work

IBM's Ninja group has developed the base optimisations for high performance numerical computing in Java [35]. They have developed techniques for finding exception free regions in for-loops which perform calculations accessing multi-dimensional arrays. These techniques enable a compiler to eliminate the tests associated with array accesses and, also, to apply classical loop reordering optimisations to the exception free regions. Our transformations take object oriented numerical computations and transform them into for-loops, in most cases involving arrays. Apart from the obvious benefit, their work enables us to respect the exact Java exception model. Our transformations, step by step, do not respect

```
if (x instance of PropertyX &&
    x.storage instanceof StorageFormatY &&
    x.currentPosition instanceof StorageFomatYPosition)
{
  ...
  // local copies
  ...
  i=iI;
  j=jJ;
  while (!conditionMatrixFinished(i,j,numRows,numColumns,...))
  {
    ...
    i=functionNextVectorI(i,j,numRows,numColumns,...);
    while (!conditionVectorFinished(i,j,numRows, numColumns,...))
    {
      ...
      double aux;
      if (condition) // linear combination involving i and j
      {
        aux= ...;
        // order one algorithm
      }
      else
      {aux=value;}
      ...
      i=functionNextElementI(i,j,numRows,numColumns,...);
    }// end while
    ...
    j=functionNextVectorJ(i,j,numRows,numColumns,...);
  }// end while

  // write back
  ...

}// end then
else
{
  // original implementation
}// end else
```

Fig. 25. Implementation of a matrix operation at IA-level obtained by applying the described steps except the transformation of while-loops into for-loops.

the Java exception model. However, if they are applied as a whole and it can be proved that the generated for-loops are exception free (i.e. successfully apply Ninja's technique), then the exception model is not violated.

The Bernoulli compiler [32] incorporates transformation techniques for C++ linear algebra programs implemented at MA-level for dense matrices. The compiler takes these programs and descriptions of sparse storage formats, and generates efficient SFA-level code using the passed in sparse storage formats. Their work is motivated because MA-level (random access to an element) is not efficient when dealing with sparse storage formats. Their transformation techniques complement the transformations described in this paper, since they cover general sparse matrices and sparse storage formats (which we do not cover) but they do not cover LCCT. The main difference is that the Bernoulli compiler is a domain-specific compiler, while our transformations are for general purpose compilers.

```
...
for (j=jJ;
     j>=1 && j<=numColumns && !conditionMatrixFinished(...);
     j=functionNextVectorJ(...))
{
  ...
  for (i=functionNextVectorI(...);
       i>=1 && i<=numRows && !conditionVectorFinished(...);
       i=functionNextElementI(...))
  {
      ...
  }// end for
    ...
}// end for
...
```

Fig. 26. Equivalent for-loops to the while-loops presented in Figure 25.

One of the transformations applied to implementations at MA-level and IA-level are the removal of `try-catch` clauses. General techniques for optimising Java programs in the presence of exceptions are described in [23, 29].

Several object oriented linear algebra libraries are surveyed and classified in [31], and information about numerical computing in Java can be found in the JavaGrande Forum reports [27, 28] and in [7, 10, 38, 12, 9, 34, 36, 22, 3, 26, 35, 11].

OoLaLa as a Generator of Specialised Implementations From an alternative point of view, the described optimisations automatically generate the specialised implementations of most matrix operations covered by the dense and banded BLAS (Chapter 2 of [8]) and LAPACK. The generation process simply needs implementations at MA-level of the general algorithms. The implementations that cannot be generated in this way are those that exploit mathematical relation properties.

Thread Safety The Java language provides threads as part of the language. In general, it is not possible to prove whether two or more threads will be accessing the same data, but the Java memory memory model enables threads to keep private copies of shared data between synchronisation points.

Note that OoLaLa is a sequential library and that the portions of the code, to which the transformation have to be applied, do not have synchronisation points. Thus, in order to make the transformations thread safe, all the attributes of the instances accessed simply have to be copied into private local variables. This has been included in the transformations for IA-level implementations, but it is not presented in the case of MA-level implementations.

Block and Recursive Algorithms Given the trend towards more complex and deeper computer memory hierarchies, block and recursive algorithms [14, 17, 24, 4] have been proposed for numerical linear algebra. These algorithms have as their main advantage that they have a better utilisation of all and each level of the memory hierarchy. As mentioned earlier, compiler transformations that

pursue these objectives are out of the scope of this paper; the work by IBM's Ninja group [35] has enabled Java compilers to apply such transformations to the code produced by the sequences of transformations presented in this section.

On the other hand, the generalisations from the examples do not impose any constraint by which block and recursive algorithms implemented at MA- and IA-level cannot be transformed into SFA-level. In addition, none of the transformations change the order in which matrix elements are accessed and, thus, the memory access pattern remains unmodified. In other words, block and recursive algorithms implemented at MA- and IA-level are transformed into block and recursive implementations at SFA-level.

6 Conclusions

MA-level and IA-level are offered as alternatives to the SFA-level imported from traditional (Fortran-based) libraries. MA-level provides library developers with an interface to access matrices independent of the storage format and with random access to matrix elements. IA-level provides library developers with an interface to access matrices also independent of the storage format but with sequential access to the non-zero elements.

The performance evaluations of a Java implementation of OoLaLa show that MA- and IA-level are not competitive with the SFA-level. IA-level is consistently slower than the low abstraction (between 5.3 and 13.6 times slower), while MA-level is slower in most cases (between 3 and 7 times slower). A Java implementation of matrix-matrix multiplication at SFA-level is competitive with its Fortran 77 equivalent compiled with `-O2`.

The core of the paper is the characterisation of the set of matrix properties and storage formats together with the transformations that enable implementations at MA- and IA-level to become efficient implementations at SFA-level. The transformations can be applied when the operands have certain matrix properties and certain storage formats. These matrix properties and storage formats are not too restrictive since they cover the dense and banded BLAS and part of LAPACK.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison Wesley, 1985.
2. Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In *Proceedings of the 10th European Conference on Object-Oriented Programming – ECOOP'96*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166. Springer-Verlag, 1996.
3. G. Almasi, F. G. Gustavson, and J. E. Moreira. Design and evaluation of a linear algebra package for Java. In *Proceedings of the ACM 2000 Java Grande*, pages 150–159, 2000.

4. Bjarne Stig Andersen, Fred Gustavson, Alexander Karaivanov, Jerzy Wasniewski, and Plamen Y. Yalamov. Lawra – linear algebra with recursive algorithms. In *Proceedings of the Conference on Parallel Processing and Applied Mathematics*, 1999.
5. David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *Computing Surveys*, 26(4):345–420, 1994.
6. David F. Bacon and Peter F. Sweeny. Fast static analysis of C++ virtual function calls. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications – OOPSLA'96*, pages 324–341, 1996.
7. Aart J. C. Bik and Dennis B. Gannon. A note on native level 1 BLAS in Java. *Concurrency: Practice and Experience*, 9(11):1091–1099, 1997.
8. BLAS Technical Forum. *Document for the Basic Linear Algebra Subprograms Standard*, August 2001.
9. Brian Blount and Siddhartha Chatterjee. An evaluation of Java for numerical computing. *Scientific Programming*, 7(2):97–110, 1999.
10. Ronald F. Boisvert, Jack J. Dongarra, Roldan Pozo, Karin A. Remington, and G. W. Stewart. Developing numerical libraries in Java. *Concurrency: Practice and Experience*, 10(11-13):1117–1129, 1998.
11. Ronald F. Boisvert, José E. Moreira, Michale Philippsen, and Roldan Pozo. Java and numerical computing. *IEEE Computing in Science and Engineering*, 3(2):18–24, 2001.
12. Zoran Budimlić and Ken Kennedy. The cost of being object-oriented: A preliminary study. *Scientific Programming*, 7(2):87–96, 1999.
13. Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *ACM SIGPLAN'94 Symposium on Principles of Programming Languages*, pages 397–408, 1994.
14. S. Carr and Ken Kennedy. Blocking linear algebra code for memory hierarchies. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, 1989.
15. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy. In *Proceedings of the 9th European Conference on Object-Oriented Programmin – ECOOP'95*, Lecture Notes in Computer Science, pages 77–101. Springer-Verlag, 1995.
16. David Detlefs and Ole Agesen. Inlining of virtual methods. In Rachid Guerraoui, editor, *Proceedings of the 13th European Conference on Object-Oriented Programming – ECOOP'99*, volume 1628 of *Lecture Notes in Computer Science*, pages 258–278. Springer-Verlag, 1999.
17. Jack J. Dongarra and David W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, June 1995.
18. May F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 103–115, 1995.
19. Erich Gamma, Richard Helm, Ralph Johson, and John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.
20. Gene H. Golub and Charles F. van Loan. *Matrix Computations*. John Hopkins University Press, 3^{th} edition, 1996.
21. David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. In *Proceedings of the ACM Object Oriented Programming Systems, Languages and Applications – OOPSLA'95*, pages 108–123, 1995.

22. Edwin Günthner and Michael Philippsen. Complex numbers for Java. *Concurrency: Practice and Experience*, 12(6):477–491, 2000.
23. Manish Gupta, Jong-Deok Choi, and Michael Hind. Optimizing Java programs in the presence of exceptions. In Elisa Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming – ECOOP 2000*, volume 1850 of *Lectures Notes in Computer Science*, pages 422–446. Springer-Verlag, 2000.
24. Fred Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):757–756, 1997.
25. Kazuaki Ishizaki, Motohiro Kawashito, Toshiaki Yassue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a Java Just-In-Time compiler. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications – OOPSLA'00*, pages 294–310, 2000.
26. Shigeo Itou, Satoshi Matsuoka, and Hirokazu Hasegawa. AJaPACK: Experiments in performance portable parallel java numerical libraries. In *Proceedings of the ACM 2000 Java Grande*, pages 140–149, 2000.
27. Java Grande Forum. *Making Java Work for High-End Computing*, November 1998. Available at http://www.javagrande.org/reports.htm.
28. Java Grande Forum. *Interim Java Grande Forum Report*, June 1999. Available at http://www.javagrande.org/reports.htm.
29. Seungll Lee, Byung-Sun Yang, Suhyun Kim, Seongbae Park, Soo-Mook Moon, Kemal Ebcioğlu, and Eric Altman. Efficient Java exception handling in just-in-time compilation. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 1–8, 2000.
30. Mikel Luján. Object oriented linear algebra. Master's thesis, Department of Computer Science, University of Manchester, December 1999.
31. Mikel Luján, T. L. Freeman, and John R. Gurd. OoLaLa: an object oriented analysis and design of numerical linear algebra. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications – OOPSLA'00*, pages 229–252, 2000.
32. Nikolay Mateev, Keshav Pingali, Vladimi Kotlyar, and Paul Stodghill. Next-generation generic programming and its application to sparse matrix computation. In *ACM International Conference on Supercomputing*, 2000.
33. Bertrand Meyer. *Object Oriented Software Construction*. Prentice Hall, 2^{th} edition, 1997.
34. José E. Moreira, Samuel P. Midkiff, and Manish Gupta. A standard Java array package for technical computing. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.
35. José E. Moreira, Samuel P. Midkiff, and Manish Gupta. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems*, 22(2):265–295, 2000.
36. George W. Stewart. *The Jampack Owner's Manual*, 1999. Available at ftp://thales.cs.umd/pub/Jampack/AboutJampack.html.
37. Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications – OOPSLA'00*, pages 264–280, 2000.
38. Peng Wu, Samuel Midkiff, José E. Moreira, and Manish Gupta. Efficient support for complex numbers in Java. In *Proceedings of the ACM 1999 Java Grande*, pages 109–118, 1999.

Parallelization of an Object-Oriented Particle-in-Cell Simulation*

Simon Pinkenburg, Marcus Ritt, and Wolfgang Rosenstiel

Wilhelm-Schickard-Institut für Informatik
University of Tübingen, Department of Computer Engineering
Sand 13, 72076 Tübingen
{pinkenbu,ritt,rosen}@informatik.uni-tuebingen.de

Abstract. We describe our experience made in parallelizing a Particle-in-Cell simulation. The project was part of our efforts to apply object-oriented methodologies to the development of parallel physical simulations. Unlike earlier projects, which were developed and parallelized in cooperation with physicists, the goal was to parallelize a sequential simulation code written in C++ without having support from its developers. Our interest was to analyze the structure of the original code and the possibilities ofadding the parallelization after sequential development. Also, since the parallelization was targeted to distributed memory architectures, we wanted to test the deployment of the object-oriented message-passing library developed in our working group.
Based on a static and dynamic analysis, we describe several general parallelization strategies and the implementation of one of them. We give a introduction to our message-passing library and detail its extension to collective communication, which was necessary to implement the parallel algorithm. Runtime measurements made on two different architectures are compared. We conclude with a discussion of the findings made in course of the project.

Keywords: Object-orientation, Message-Passing, Simulation, Particle-in-Cell

1 Introduction

This work is part of a government funded collaboration of physicists, mathematicians and computer scientists to develop large-scale physical simulations for massive parallel computers. Our group is concerned with the development of adequate runtime environments and libraries to parallelize these simulations effectively.

While in industry object-oriented techniques and programming languages are widely used, the scientific computing community still does not employ them in

* This project is funded by the Deutsche Forschungsgemeinschaft as a part of the Collaborative Research Center 382 (Methods and Algorithms for the Simulation of Physical Processes on High Performance Computers)

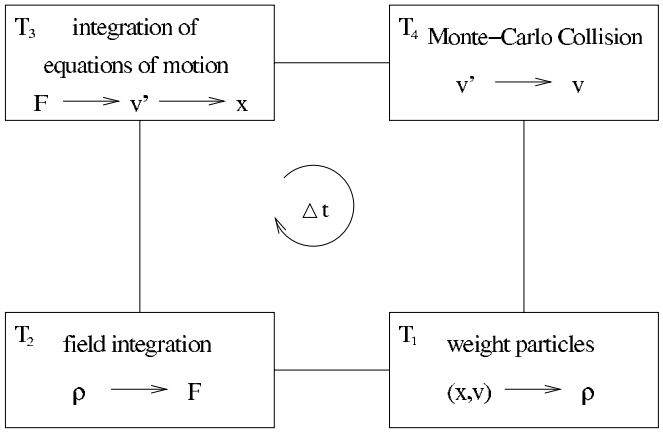

Fig. 1. Steps in a Particle-in-Cell simulation with MCC

the majority of the applications. In recent years, some efforts to enable object-orientation for parallel computing have been made, mainly in the C++ community. Efficient and standard-conforming compilers, mathematical base libraries reduced the performance gap between classical approaches and object-oriented codes [14]. Parallel programming standards like the Message-passing interface provide support for object-oriented languages and efforts to establish object-oriented frameworks for parallel computing are ongoing [15, 8, 11].

In this paper, we investigate how object-oriented methods can help in the parallelization of object-oriented codes. A focus is the reusability of the object-oriented design of a sequential code. In section 2 we give a overview over the Particle-in-Cell method used in this application. We explore parallelization strategies in section 3 and discuss their implementation in the next section. In section 4, we also present an object-oriented message-passing library supporting collective communications, which helped to parallelize the application on distributed memory machine at a reasonable level of abstraction. Measurements of the parallel execution are presented and discussed in section 5. We conclude in section 6.

2 Particle-in-Cell simulation with Monte-Carlo collisions

In a Particle-in-Cell simulation, the medium under consideration is represented by a large number of macro particles, each describing the physics of an ensemble of real particles. The macro particles reside in a simulation space of finite geometric boundaries. In contrast to other particle methods, the particle interactions are calculated using a discrete grid. The grid divides the simulation space into usually regular subregions or cells (hence the name of the method). Characteristic physical properties of the particles are weighted on the grid using a kernel

function. The momentum equations on the grid are solved with some standard method (for example finite differences). Based on these results, the forces are calculated and interpolated to the particles positions to solve the equations of motion.

PiC simulations use grid points to reduce drastically the amount of computation necessary to produce good approximations to the actual behavior of the underlying physical phenomena. To do this they make use of the common physical property that the influence of any two macro particles upon one another quadratically decreases as the geometric distance between them increases. The computational reduction is accomplished by weighting the contributions of the particles on the grid cells, using a kernel function of finite domain. In this way, the computational complexity of simulating n macro particles drops from $O(n^2)$ to $O(n)$.

In this specific application, the PiC method is used to simulate the phenomena in the electrostatic plasma of a direct current glow discharge in a tube. A plasma medium is an ionized gas, which may be regarded as a collection of ions and electrons interacting through their mutual electric and magnetic fields. PiC methods are often used to simulate plasma phenomena by solving Maxwell's Equations in a numeric manner. This involves computing the dynamics of a large number of electrons and ions in the plasma and the influence of the self-consistent electromagnetic fields. Each macro particle represents about $2 \cdot 10^9$ real particles. Since the problem to be solved shows a cylindrical symmetry and the radius components were of no interest, the simulation is effectively one-dimensional. In order to make a model of the direct particle interactions in the plasma, the method is extended by Monte-Carlo collision (MCC) processes. The collision processes prevent regarding the whole physics of the scattering process, but randomly approximate elastic scattering and the stimulation or ionization of neutral gas atoms by electrons.

The sequential PiC code in C++ was the result of an effort creating a framework for PiC simulations called Open Particle Framework (OPAR) [2]. While designed as an extensible framework, it currently implements only the classes required to simulate the direct current glow discharge. The object-oriented CASE tool *together* was used for the static analysis.

The application provides two major abstractions: a task concept and a diagnosis subsystem. Tasks define the execution of the simulation. Each physical entity is encapsulated in a task (by deriving them from class `CTask`) and implements its simulation code. The task classes follow the Composite design pattern and thus a hierarchical execution structure can be defined. Task objects have an external representation for configuring the simulation. This configuration file makes it possible to define the tasks, their parameters and the static dependencies (associations) between them. On start of the simulation, the static object model is reconstructed from this configuration file. Thus, the user is able to construct arbitrary simulation setups based on the set of implemented physical entities without recompiling the application.

The diagnosis subsystem contains classes for producing output from simulation runs. Any diagnosis output (f.ex.particle trajectories or the plasma density distribution) is implemented in its own diagnosis task. This task can be attached as a subtask to the physical entity to observe, and produces the desired output.

Since *together* does not support dynamic analysis by producing an UML sequence diagram from a concrete run, the dynamic behavior was analyzed by manually tracing some executions. In each timestep, the execution engine runs all configured tasks in the order given by the configuration file. The configuration supports one-time tasks, which can be used for additional setup, and tasks running the entire simulation time.

3 Parallelization strategies

After initializing the particles, each simulation timestep executes four substeps: T_1 weights the particles on the grid, T_2 calculates the solution on the grid, T_3 solves the equations of motion of the particles and T_4 applies the Monte-Carlo processes to the particles.

A parallelization has to give a decomposition of the tasks and consider the data dependencies between the parallel computations. Regarding time complexity, tasks T_1, T_3 and T_4 are linear in the number of particles and T_2 is linear in the number of grid points. In T_1, T_3 and T_4, the computation on different particles is independent, but there is a data dependency for T_1 and T_3 to all grid points (an output dependency in case of T_1, and an input dependency in T_3). In T_2, the computations on a grid point depend on its neighbors. Therefore, T_1, T_3 and T_4 can be decomposed on the number of particles, and T_2 on the number of grid points. For the parallel execution of T_1 and T_3, either the data dependencies on the grid have to be resolved, or concurrent accesses to the grid have to be synchronized. The same holds for T_2 in respect to neighboring grid points.

3.1 Decomposition

A parallel Particle-in-Cell algorithm has the choice of divide up the work done on the particles, the grid or both. The next sections explore and compare these different approaches.

Domain decomposition The first possibility, dividing up the grid into subgrids of the same size where each of them remains on one processor, is also the most common way used for particle methods. Each processor thereby keeps one subgrid and all particles in its local memory.

In step T_1 – computing the density – the processors weight the particles on their part of the grid. Next, the electric fields, potentials and force fields are locally solved in parallel under the consideration of data dependencies between the boundary values of each subgrid. Now, the equations of motion can be solved setting the new positions and velocities of all particles. A synchronization follows, updating the positions and velocities of the particles on all processors to

regain redundancy. Thereby, only the moved particles are exchanged by a scatter function in order to reduce the communication overhead. The used function enables every processor to have all new positions and velocities. Finally, the Monte-Carlo processes are executed in parallel.

The whole communication overhead, neglecting communication of boundary values, depends on the number of particles, making this strategy only useful for problems with a large computation on the grid and a small amount of particles.

Particle decomposition The second strategy divides up the particles on each processor and keeps the grid redundant in memory to resolve data dependencies. After weighting the particles on the local grid, a global reduction operation, which sums up all local grids and distributes the result back to all processors, leaves the application with a redundant global grid. The use of a reduction operation is legal because of the additivity of the physical quantities weighted on the grid. Now, the processors can locally compute the electric field, the potential and the forces on the grid in parallel and subsequently advance their particles by setting their new positions and velocities. After simulating the Monte-Carlo collisions the cycle can be repeated.

This strategy parallelizes only the operations on the particles. The computation on the grid, while executed in parallel, is the same on all processors and thus adds to the sequential overhead. The additional communication overhead depends on the number of grid points transmitted in the reduction operation. This approach is efficient for problems with small grids and a great large number of particles.

Domain and particle decomposition The third possibility is to combine both alternatives: divide up the particles and the grid points. Steps T_1, T_3 and T_4 – dealing with particles – should be divided by particles and step T_2 by grid points.

If the memory is large enough, an implementation could keep the grid redundant and simply merge the steps of the particle and domain decomposition. In step T_1, each processor weights its particles on the local grid. After the grid is subdivided into equal parts, each processor sums up the local contributions of the subgrid now is responsible for from all other processors. This can be done in a single reduction operation. In the next step, the processors compute the fields on each subgrid in parallel, considering the data dependencies between boundary values. Afterwards each processor broadcasts its local grid, bringing all local grids up-to-date, which makes in turn possible to proceed locally with the computation of new positions and velocities of the particles and the Monte-Carlo collisions.

This implementation involves a lot of communication overhead. If tight memory resources prohibit the redundancy of some data structures, like in the algorithm above, this overhead gets even worse. Such an approach is only efficient for problems with large grids and a large number of particles.

Conclusion The choice of a parallelization strategy for an application depends on the percentages of computation spent in updating the particles and the grid. As a requirement of the PiC method, the number of particles must exceed the number of grid points. Usually the computation on the grid contributes less than 10% to the total running time, depending on the complexity of the algorithm. In our case, the number of particles exceeds the number of grid points by a magnitude of 2 to 4, therefore the computation on the grid contributes less than 1%. Thus we decided to decompose the computation only by the number of particles. If the simulation would be extended to three dimensions, which increases the number of grid points and requires more complex algorithms for the solution of the field equations, a combined domain and particle decomposition approach would be most promising.

3.2 Load balancing

Another important point was to add a load balancing component. Initially, the particles are distributed equally to all processors. Due to the Monte-Carlo collisions and ionizations done in the last step of the algorithm, the distribution of the particles changes in course of the simulation. The resulting load imbalance can reduce the speedup significantly. The load balancing component checks the particles distribution periodically and averages the number of particles on the processors. Since the running time of a full timestep is very small, the load balancer must be carefully tuned to add little overhead. This was achieved by checking the balance only every few steps and executing the averaging process, which implies the communication of the particles, only if the imbalance exceeds a configurable limit. Further, the averaging process is stopped, if a sufficient balance is reached. For this application, the load balancer starts rebalancing at 20% imbalance and stops at 5% imbalance. This can be usually done in a single communication step, which keeps the overhead small.

4 Implementation

A goal of the implementation was to parallelize the application without substantially rewriting the sequential code. The implementation had to modify three parts of the code: At initialization the particles must be distributed equally to all processors, between T_1 and T_3, the redundant grid must be updated, and in after some number of timesteps, the particles must be load balanced. Where it is possible, the implementation follows the task model. Initialization of the particles is done in tasks of class `CGeneration`. Since the particles are not initialized from external values but created on startup, the initial distribution of particles to processors could be done by modifying this task (by inheritance) to assign newly created particles to the processors in a round-robin fashion. This step requires no communication.

The summation of the local grids could be implemented transparently as a subtask of class `CDensity`, which is responsible for weighting the particles to the

grid. The subtask executes the reduction operation after the local updates are done.

Since the load balancer depends on the particles, it has been implemented as a subtask of class `CSpecies`, the representation of the particles. After the integration of the equations of motion, this subtask is responsible for checking and, if necessary, averaging the particle distribution.

4.1 TPO++

The implementation of the communication was done in TPO++ [3], an object-oriented message-passing library developed in our group. In this section, we give an overview of point-to-point communication in TPO++. For this application, TPO++ has been extended to collective communication, which is discussed in more detail in the next subsection.

TPO++ implements an object-oriented interface for the functionality of the well-known MPI 1.2 message-passing standard [11]. It is intended to fill the semantic gap between current the message-passing standard MPI and the object-oriented programming paradigm. This includes a type-safe interface with a data-centric rather than a memory-block oriented view and concepts for inheritance of communication code for classes. Other goals were to provide a light-weight, efficient and thread-safe implementation, and, since TPO++ is targeted to C++, the extensive use of all language features that help to simplify the interface. A distinguishing feature compared to other approaches [6,7,1,12] is the tight integration of the Standard Template Library (STL). TPO++ is able to communicate STL containers and adheres to STL interface conventions. All communication methods provide the same orthogonal interface for specifying the data objects to communicate. A sender has two options: provide a single datatype (basic or object) or a range of data elements by using a pair of STL iterators. A receiver has the third option to provide a special back inserter (in analogy to the STL back inserters) that allocates the memory on the receiving side automatically.

The following code examples illustrate some of the features of TPO++. Figure 2 shows two classes enabled for transmission in TPO++. For types with a trivial copy constructor like `Point`, a single declaration is sufficient to achieve this. For more complex types like `Circle`, the user has to provide two marshalling methods named `serialize` and `deserialize`. Figure 3 shows the communication of a single object and a collection of objects of class `Circle` from process 0 to process 1.

For further details on TPO++ and a comparison with other object-oriented message-passing systems see [3].

4.2 Collective communication in TPO++

The implementation of collective communication in TPO++ covers the functionality of MPI 1.2. MPI provides three groups of collective primitives:

```
class Point {
public:
  Point() : x(0), y(0) {}
private:
  int x, y;
};
TPO_TRIVIAL(Point);
```

```
class Circle {
public:
  Circle() : radius(0.0) {
    center = new Point(0.0);
  }
  ~Circle() { delete center; }
  void
  serialize(Message_data& m) const {
    m.insert(*center);
    m.insert(radius);
  }
  void
  deserialize(Message_data& m) {
    m.extract(*center);
    m.extract(radius);
  }
private:
  Point* center;
  double radius;
};
TPO_MARSHALL(Circle);
```

Fig. 2. Examples of two classes enabled for transmission in TPO++.

```
using namespace TPO;
if (CommWorld.rank() == 0 ) { // sender
  Circle c;
  CommWorld.send(c, 1);

  vector<Circle> vc(20);
  CommWorld.send(vc.begin(), vc.end(), 1);
} else { // receiver
  Circle c;
  CommWorld.receive(c);

  vector<Circle> vc(20);
  CommWorld.receive(vc.begin(), vc.end());
}
```

Fig. 3. Transmission of user-defined objects.

```
vector<double> vd(10);
TPO::CommWorld.bcast(vd.begin(), vd.end(), 2);
```

Fig. 4. Broadcast of a container of floating-point values in TPO++ rooted at process 2.

- The basic collective primitives `broadcast` and `barrier`.
- Four data-exchange primitives (`scatter`, `gather`, `allgather` and `alltoall`).
- Four combination primitives (`reduce`, `allreduce`, `reduce_scatter` and `scan`).

The goal of the TPO++ implementation of collective communication was to provide a interface consistent with its point-to-point interface and the STL conventions. Of course, the implementation should show a reasonable performance compared to MPI. The existing interfaces suggest an implementation of collective operations as methods of class `TPO::Communicator`. The methods should accept the different kind of data structures discussed in section 4.1. For the details of the interface, several characteristics different from the point-to-point communication are relevant:

Five primitives (`broadcast` and the combination primitives) are *rooted operations*, i.e. they are asymmetrical in respect to one process. The root must be somehow given for these operations. Since it is likely to change in subsequent calls, we chose to add it to the methods parameters. Figure 4 gives an example of broadcasting a vector of floating point values.

The data-exchange and combination primitives are operations on different input and output data structures, given in the same call. With two variants to send and three variants to receive date, we have a maximum of six possible overloaded methods for these operations. Looking more closely, not all possibilities are reasonable, for example a scatter operation, which sends only one element, is impossible[3]. Table 1 summarizes the overloaded methods of these operations.

The data-exchange primitives also come in a vector-variant, which does not require the data structures of the participating processes to be of the same size. MPI allows the clients for p processes to pass p memory blocks of different size and offset relative to a common base address. An obvious generalization in C++ is to allow the clients to pass a number of element ranges, each given by a begin and end iterator. Several different realizations of such an interface are imaginable. TPO++ favors a container of pairs of begin and end iterators. This simplifies the interface, since only one additional parameters is needed, and enhances readability and safety due the explicit pairing of corresponding iterators. For convenience in the common case of adjacent element ranges of different sizes, TPO++ provides another interface, where only a single container of $p+1$ iterators defining the p segments of different length is required.

[3] Except in the boundary case of one process, in which no communication is needed.

The four combination operations require another parameters, the operation to be applied to the elements given by the processes. Standard operations defined in MPI include for example the sum or the minimum of the values. In conformance with the STL, TPO++ allows the client to pass an arbitrary function object. Unlike the STL, MPI further requires the client to state the commutativity of the operation, which is realized by a boolean class attribute of the function operators type. Figure 5 gives an example of an user-defined function operator and a reduction in TPO++.

The implementation of function operators cannot hide entirely the MPI layer and some deviations of the STL semantics. Unlike STL function operators, the function operators in TPO++ obviously cannot be state-dependent, since the operations are executed on different hosts in different instances. Moreover, TPO++ requires the data objects used in reduction operators to provide a default constructor and to be of constant size. These restrictions result from the constraints of MPI user-defined combination operations.

	Receiver		
Sender	Single	Collection	Back-inserter
Single	Broadcast, Reduce, Scan, Allreduce	(All)Gather	(All)Gather
Collection	Scatter	All operations	All operations

Table 1. Overload collective communication methods. The sender can pass *single* objects or arbitrary *collections* of elements in STL containers. The receiver can get *single* objects, *collections* of elements in STL containers or allocate the space automatically using a *back-inserter*.

```
// user-defined operator
// for combination operations
template <class T>
class sum {
  public:
  static bool commute;
  void operator()(T& inout,
    const T& in) {
    inout += in;
  }
};

template <class T>
bool sum<T>::commute = true;
```

```
vector<double> source(20);
vector<double> result(20);

// reduction
CommWorld.reduce(source.begin(),
                 source.end(),
                 result.begin(),
                 result.end(),
                 sum<double>(),
                 0);
```

Fig. 5. Example of an user-defined reduction operation `sum` and its application in the reduction of a container of floating-point values rooted at process 0.

Related work To our knowledge, three object-oriented message-passing systems, mpi++ [6,7], para++ [1] and OOMPI [12] implement collective communication primitives, which significantly differ from the MPI C++ bindings [9, 10].

Para++ implements the concept of C++ IO streams for message-passing and provides only a broadcast and multicast primitive for collective communication[4]. Obviously, having only broadcast is not sufficient for most applications.

mpi++ implements the full set of MPI 1.2 collective communication primitives. In mpi++ collective communication primitives are implemented in two template classes, `Collective` and `Reduction`. `Collective` is parameterized with the type information about the data to send and receive, its methods implement all basic and data-exchange primitives, and its attributes hold the communicator to use and possibly the root of the collective communication. The derived class `Reduction` implements the combination primitives and encapsulates additional information about the type of operation to apply. Different to TPO++, mpi++ reintroduces, in analogy to MPI, its own type system, but does not support the STL. Also, the interface is built around the operations, which are reified as classes, and avoids method parameters in favor of attributes.

OOMPI implements the full set of MPI 1.2 collective communications. One of basic abstractions in OOMPI, class `Port` is used to specify the master for rooted operations. All other operations are implemented in the communicator class `Intra_comm`. Class `OOMPI_Op` is a simple wrapper for MPI operators. The constructor also accepts user-defined MPI operators. Unfortunately, at this point, the underlying MPI layer is visible for the user.

5 Performance measurements

The performance of the Particle-in-Cell code has been measured on two different architectures, the Cray T3E, a conventional supercomputer installed at the German supercomputer center in Stuttgart [4] and Kepler, a self-made clustered supercomputer[6] based on commodity hardware [13]. The T3E has 512 nodes, each equipped with a DEC Alpha EV5 21164 processor running at 450 Mhz and 128 MB of RAM. The interconnect organizes the nodes in a three-dimensional torus ($8 \times 8 \times 8$), and every connection to the 6 neighbors provides a nominal bidirectional bandwidth of about 500 MB/s. Measurements of MPI application to application performance gives a bandwidth of about 300 MB/s and 15μs latency. Keplers 96 nodes are running two Pentium III processors at 650 Mhz and have 1 GB of total memory, or 512 MB per processor. Kepler has two interconnects, a fast ethernet for booting the nodes and administration purposes and a Myrinet network for parallel applications. The latter has a multi-staged hierarchical switched topology organized as a fat tree. The nominal bandwidth of 133

[4] In MPI, the multicast is realized by creating a new communicator containing only a subset of all processes.

[6] As of July 2001 Kepler occupies rank 290 at the TOP 500 list of supercomputers.

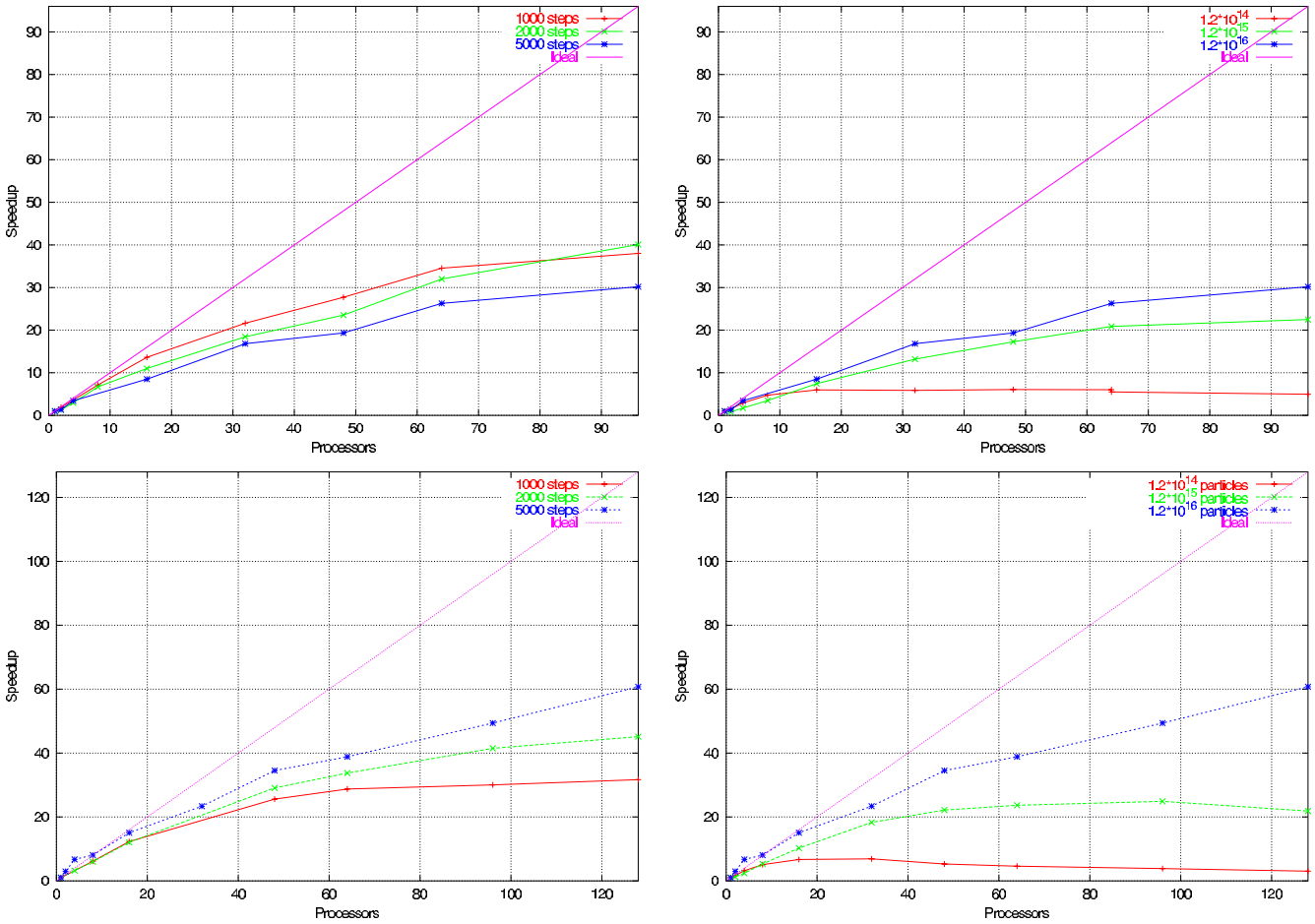

Fig. 6. Measurements on Kepler cluster. Upper row: Load balancing disabled. Lower row: Load balancing enabled. Left column: $1.2 * 10^{16}$ particles for different numbers simulation steps. Right column: 5000 simulation steps for different number of particles.

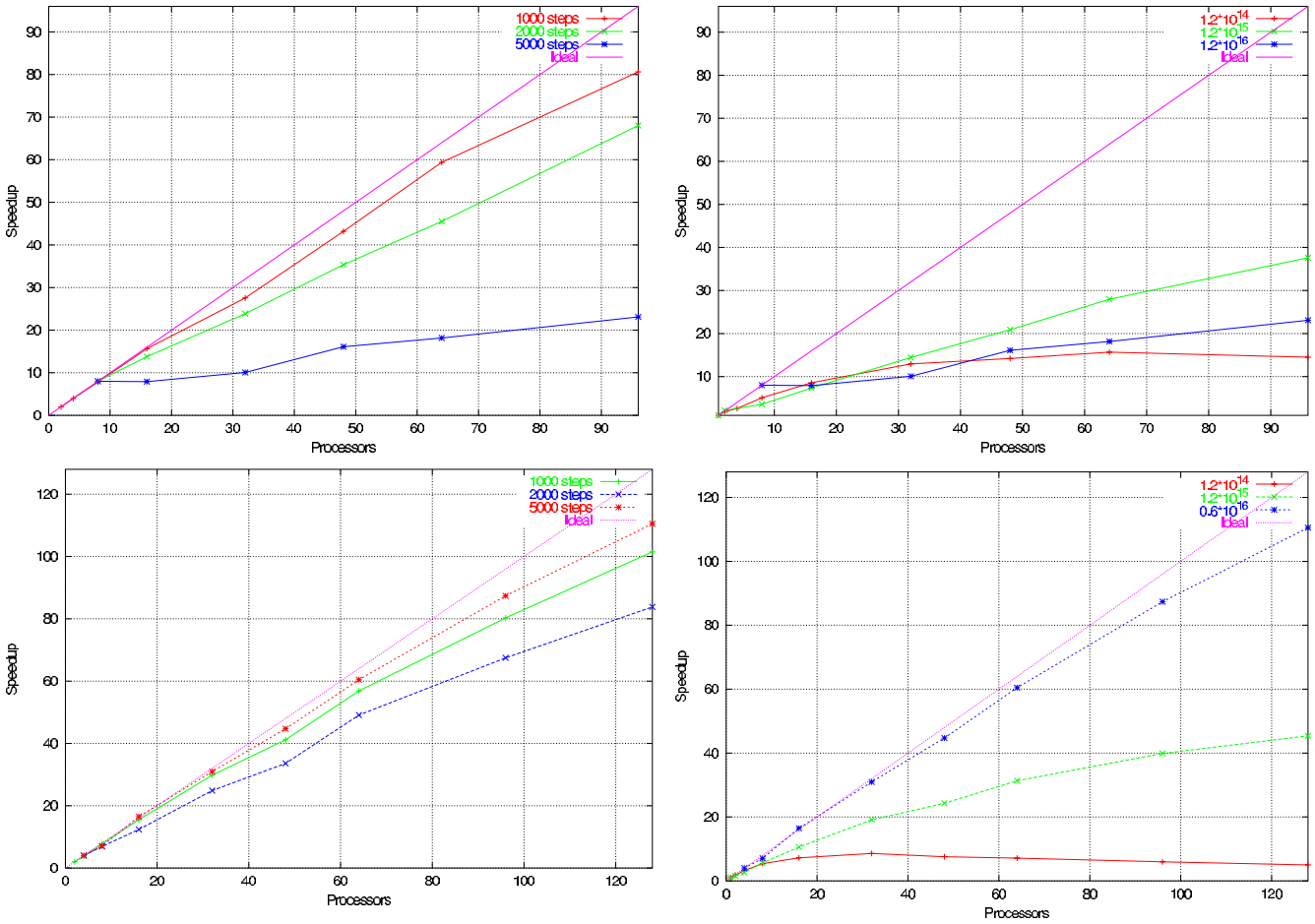

Fig. 7. Measurements on Cray T3E. Upper row: Load balancing disabled. Lower row: Load balancing enabled. Left column: $1.2 * 10^{16}$ particles for different numbers simulation steps. Right column: 5000 simulation steps for different number of particles.

MB/s is the maximum PCI transfer rate, measurements give about 115 MB/s bandwidth and 7μs latency.

The Particle-in-Cell application has been run with couple of different parameters. Three different numbers of particles have been used to observe the effects of a varying computation to communication ratio. Three different numbers of simulation steps show the growing imbalance without load balancing and the improvements after load balancing. All measurements have been made on up to 128 processors.

Figure 6 shows the results for the Kepler cluster, figure 7 for the Cray T3E. Note that the speedups of the large runs on Cray T3E with $1.2 * 10^{16}$ particles and 5000 simulation steps are based on the runtimes for 8 processors, which improves these speedups artificially. The problem did not fit in a smaller number of processors. For the same reason, the largest runs on Cray T3E use only $0.6 * 10^{16}$ simulation particles.

The different number of simulation steps show the effect of a growing imbalance if load balancing is disabled, which results in a decreasing performance with increasing number of simulation steps. With load balancing enabled, the picture is reversed. The speedups improve and the application can profit from the increasing number of particles created in course of the simulation. For smaller number of simulation particles, the load balancing is not able to improve the speedups, in case of $1.2 * 10^{14}$ particles they even decrease. Regarding the initial number of simulation particles, the results clearly show a break-even between $1.2 * 10^{15}$ particles, showing moderate speedups, and $1.2 * 10^{16}$ particles, with very good performance. For $1.2 * 10^{14}$ particles, the application can gain only limited runtime improvement.

6 Conclusions

While the task model allows flexible combination of physical entities, the configuration is restricted to a sequential execution model. In this particular domain, an extension could provide parallel execution primitives. A shortcoming of this implementation is the lack of a consistent organization beyond the task abstraction. For example, the objects defining the geometry and representing the particles are used globally, and therefore are tightly coupled to all other classes. As a consequence of this, it is impossible to configure the application without knowing the classes in detail. The lack of documentation makes it even more complicated.

The use of a design tool simplified the reverse-analysis of the application substantially as well as the understanding of the class dependencies, which otherwise would have not been possible. Regarding the parallelization, static and dynamic analysis helped to select from the theoretical approaches the most reusable strategy, which is not necessarily the most efficient one. In this case, both strategies were the same, which simplified the parallelization substantially. The subsequent addition of a parallelization without modification of the sequential code

was possible due to the flexibility of the task model. The tool also helped a lot documenting the code.

References

1. O. Coulaud and E. Dillon. Para++: C++ bindings for message-passing libraries. In *EuroPVM Users Meeting*, September 1995.
2. T. Daube and H. Schmitz. OPAR: Open architecture C++ plasma simulation code. Ruhr-Universität Bonn, 1998.
3. T. Grundmann, M. Ritt, and W. Rosenstiel. TPO++: An object-oriented message-passing library in C++. pages 43–50. IEEE Computer society, 2000.
4. High performance computing center Stuttgart. *Cray T3E-900/512*. Online. URL: http://www.hlrs.de/hw-access/platforms/crayt3e (January 2001).
5. M. Hipp, S. Hüttemann, M. Konold, M. Klingler, P. Leinen, M. Ritt, W. Rosenstiel, H. Ruder, R. Speith, and H. Yserentant. A parallel object-oriented framework for particle methods. In E. Krause and W. Jäger, editors, *High Performance Computing in Science and Engineering '99*, pages 483–495. Springer-Verlag, 1999.
6. D. Kafure and L. Huang. mpi++: A C++ language binding for MPI. In *Proceedings MPI developers conference*, Notre Dame, IN, June 1995.
7. D. Kafure and L. Huang. Collective communication and communicators in mpi++. Technical report, Department of Computer Science Virginia Tech, 1996.
8. Los Alamos National Laboratory. *POOMA*, 2000. Online: http://www.acl.lanl.gov/PoomaFramework.
9. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230, Computer Science Department, University of Tennessee, Knoxville, TN, May 1994.
10. Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.
11. M. Snir and W. Gropp. *MPI: The complete reference*. MIT Press, 1998.
12. J. M. Squyres, B. C. McCandless, and A. Lumsdaine. Object Oriented MPI: A Class Library for the Message Passing Interface. In *Proceedings of the POOMA conference*, 1996.
13. University of Tübingen. *Kepler cluster website*. Online. URL: http://kepler.sfb382-zdv.uni-tuebingen.de.
14. T. Veldhuizen. Arrays in blitz++. In D. Caromel, R. Oldehoeft, and M. Tholburn, editors, *Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, pages 223–231, 1998.
15. G. V. Wilson and P. Lu, editors. *Parallel Programming using C++*. The MIT Press, Cambridge, 1996.

Parallel Code Generation in MathModelica / An Object Oriented Component Based Simulation Environment

Peter Aronsson, Peter Fritzson
(petar,petfr)@ida.liu.se

Dept. of Computer and Information Science,
Linköping University, SE-581 83 Linköping, Sweden

Abstract. Modelica is an a-causal, equation based, object oriented modeling language for modeling and efficient simulation of large and complex multi domain systems. The Modelica language, with its strong software component model, makes it possible to use visual component programming, where large complex physical systems can be modeled and composed in a graphical way. One tool with support for both graphical modeling, textual programming and simulation is MathModelica.
To deal with growing complexity of modeled systems in the Modelica language, the need for parallelization becomes increasingly important in order to keep simulation time within reasonable limits.
The first step in Modelica compilation results in an Ordinary Differential Equation system or a Differential Algebraic Equation system, depending on the specific Modelica model. The Modelica compiler typically performs optimizations on this system of equations to reduce its size. The optimized code consists of simple arithmetic operations, assignments, and function calls.
This paper presents an automatic parallelization tool that builds a task graph from the optimized sequential code produced by a commercial Modelica compiler. Various scheduling algorithms have been implemented, as well as specific enhancements to cluster nodes for better computation/communication tradeoff. Finally, the tool generates simulation code, in a master-slave fashion, using MPI.

Keywords: Object Oriented Modeling, Visual Programming, Components, Scheduling, Clustering, Modelica, Simulation, Large Complex System

1 Introduction

Modelica is an a-causal, object-oriented, equation based modeling language for modeling and simulation of large and complex multi-domain systems [15, 8] consisting of components from several application domains. Modelica was designed by an international team of researchers, whose joint effort has resulted in a general language for design of models of physical multi-domain systems. Modelica has influences from a number of earlier object oriented modeling languages, for instance Dymola [7] and ObjectMath [9].
The four most important features of Modelica are:

- Modelica is based on equations instead of assignment statements. This permits a-causal modeling that gives better reuse of classes since equations do not specify a certain data flow direction. Thus a Modelica class can adapt to more than one data flow context.
- The possibility of having model components of physical objects from several different domains such as e.g. electrical, mechanical, thermodynamic, hydraulic, biological and control applications can be described in Modelica.
- Modelica is an object-oriented language with a general class concept that unifies classes, generics (known as templates in C++) and general subtyping into a single language construct. This facilitates reuse of components and evolution of models.
- The strong software component model in Modelica has constructs for creating and connecting components. Thus the language is ideally suited as an architectural description language for complex physical systems, and to some extent for software systems.

The class, also called model, is the building block in Modelica. Classes are instantiated as components inside other classes, to make it possible to build a hierarchical model of a physical entity. For instance, an electrical DC motor can be composed of a resistor, an inductance and a component transforming electricity into rotational energy, see Figure 1. The model starts with two import statements, making it possible to use short names for models defined in the **Modelica.Electrical.Analog.Basic** package (the first import statement) and using the short name for the **StepVoltage** model defined in the Modelica package **Modelica.Electrical.Analog.Sources** (the second import statement). The import statements are followed by several component instantiations, where modification of components is used. The modification of components is a powerful language construct that further increases the possibility of reuse. The next part of the model definition is the `equation` section, see Figure 1. It can consist of arbitrary equations, involving the declared variables of a model. It can also contain `connect` statements, which are later translated into equations that couple variables in different components together.
One tool for developing models, for simulation and for documentation is MathModelica [14]. It integrates Modelica with the mathematical engineering tool *Mathematica* and the diagram editor *Visio* to allow the user to work with models both in a powerful computer algebra system and by using component based modeling in a drag and drop/connect fashion in a graphical environment, see Figure 2. In MathModelica a model can also be entered in a Mathematica notebook document as ordinary text. For example, the `dcmotor` model can be simulated by:

```
Simulate[dcmotor,{t,0,50}];
```

The MathModelica environment then compiles the Modelica model into C, which is then linked with a solver into an executable file. The result from running the simulation consist of a number of variables changing over time, i.e. they are functions of time. In MathModelica these variables are directly available and can be, for instance, plotted, see Figure 3.
When a model described in Modelica is to be simulated the involved models, types and classes are first fed to a compiler. The Modelica compiler flattens the

```
model dcmotor
  import Modelica.Electrical.Analog.Basic.*;
  import Modelica.Electrical.Analog.Sources.StepVoltage;
  Resistor R1(R=10);
  Inductor L(L=0.01);
  EMF emf;
  Ground G;
  StepVoltage src;
equation
  connect(src.p,R1.p);
  connect(R1.n,L.p);
  connect(L.n,emf.p);
  connect(emf.n,G.p);
  connect(G.p,src.n);
end dcmotor;
```

Fig. 1. A simple model of a DCmotor described in the Modelica modeling language.

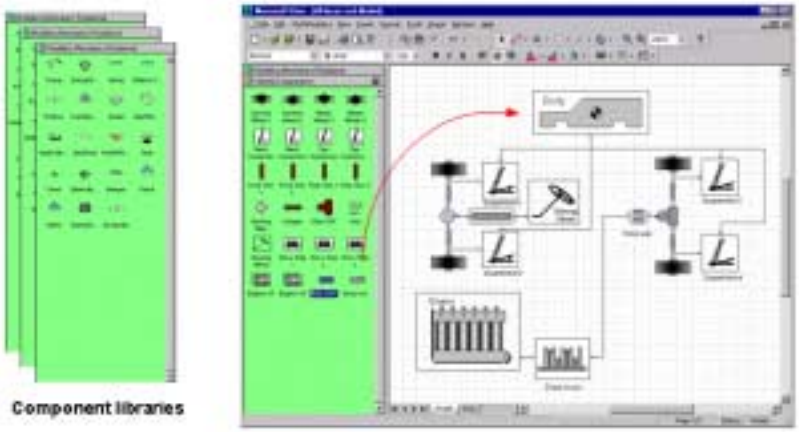

Fig. 2. Visual component based modeling in MathModelica.

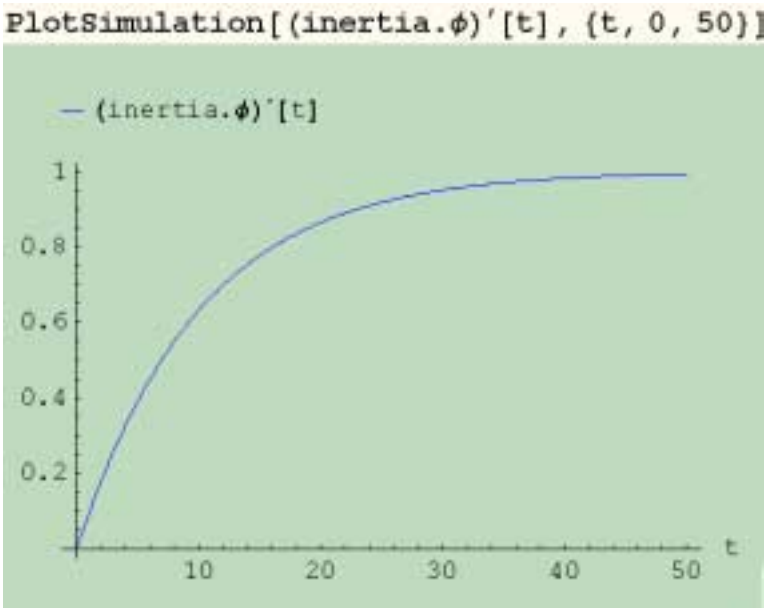

Fig. 3. A Notebook document containing a plot command.

object oriented structure of a model into a system of differential algebraic equations (DAE) or a system of ordinary differential equations (ODE), which during simulation is solved using a standard DAE or ODE solver. This code is often very time consuming to execute, and there is a great need for parallel execution, especially for demanding applications like hardware-in-the-loop simulation.

The flat set of equations produced by a Modelica compiler is typically sparse, and there is a large opportunity for optimization. A simulation tool with support for the Modelica language would typically perform optimizations on the equation set to reduce the number of equations. One such tool is Dymola [6], another is MathModelica [14].

The problem presented in this paper is to parallelize the calculation of the states (the state variables and their derivatives) in each time step of the solver. The code for this calculation consists of assignments of numerical expressions, e.g. addition or multiplication operations, to different variables. But it can also contain function calls, for instance to solve an equation system or to calculate sin of a value, which are computationally more heavy tasks. The MathModelica simulation tool produces this kind of code. Hence we can use MathModelica as a front end for our automatic parallelization tool. The architecture is depicted in Figure 4, showing the parallelizing tool and its surrounding tools.

To parallelize the simulation we first build a task graph, $G = (V, E)$ where each task $v \in V$ corresponds to a simple binary operation, or a function call. A data dependency is present between two tasks v_1, v_2 iff v_2 uses the result from v_1. This is represented in the task graph by the edge $e = (v_1, v_2)$. Each task

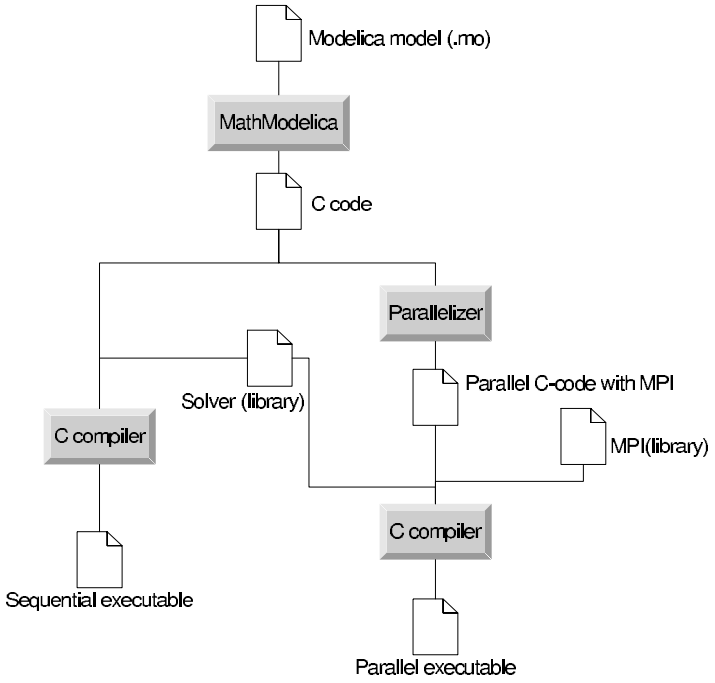

Fig. 4. The architecture of a Modelica simulation environment

is assigned an execution cost which corresponds to a normalized execution time of the task, and each edge is assigned a communication cost corresponding to a normalized communication time between the tasks if they execute on different processors. The goal is to minimize the execution time of the parallel program. This often means that the communication between processors must be kept low, since interprocessor communication is very expensive. When two tasks execute on the same processor, the communication cost between them is reduced to zero.

Scheduling and partitioning of such task graphs described above has been studied thoroughly in the past three decades. There exists a plethora of different scheduling and partitioning algorithms in the literature for different kinds of task graphs, considering different aspects of the scheduling problem. The general problem of scheduling task graphs for a multi-processor system is proven to be NP complete [16].

The rest of the paper is organized as follows: Section 2 gives a short summary of related work. Section 3 presents our contribution of parallelizing simulation code. In section 4 we give some results of our contribution, followed by a discussion and future work in section 5.

2 Related Work on Multiprocessor Scheduling

A large number of scheduling and partitioning algorithms have been presented in the literature. Some of them use a *list scheduling* technique and heuristics [1, 4, 5, 10, 12, 13], some have been designed specifically for simulation code [20]. A list scheduler keeps a list of tasks that are ready to be scheduled, i.e. all its predecessors have already been scheduled. In each step it selects one of the tasks in the list, by some heuristic, and assigns it to a suitable processor, and updates the list.
Another technique is called *critical path scheduling* [17]. The critical path of a task graph (DAG) is the path having the largest sum of communication and execution cost. The algorithm calculates the critical path, extracts it from the task graph and assign it to a processor. After this operation, a new critical path is found in the remaining task graph, which is then scheduled to the next processor, and so on. One property of critical path scheduling algorithms is that the number of available processors is assumed to be unbounded, because of the nature of the algorithm.
An orthogonal feature in scheduling algorithms is *task duplication* [11, 17, 21]. Task duplication scheduling algorithms rely on task duplication as a mean of reducing communication cost. However, the decision if a task should be duplicated or not introduces additional complexity to the algorithm, pushing the complexity up in the range $O(n^3)$ to $O(n^4)$ for task graphs with n nodes.

3 Scheduling of Simulation Code

Simulation code generated from Modelica mostly consist of a large number of assignments of expressions with arithmetic operations to variables. Some of the variables are needed by the DAE solver to calculate the next state, hence they must be sent to the processor running the solver. Other variables are merely temporary variables whose value can be discarded after the final use.
The simulation code is parsed, and a fine grained task graph is built. This graph, which has the properties of a DAG, can be very large. A typical application (a thermo-fluid model of a pipe, discretisized to 100 pieces), with an integration time of around 10 milliseconds, has a task graph with 30000 nodes. The size of each node can also vary a lot. For instance, when the simulation code originates from a DAE, an equation system has to be solved in each iteration if it can not be solved statically at compile time. This equation system can be linear or non-linear. In the linear case, any standard equation solver could be used, even parallel solvers. In the non-linear case, fixed point iteration is used. In both cases, the solving of the equation system is represented as a single node in the task graph. Such a node can have a large execution time in comparison to other nodes (like an addition or a multiplication of two scalar floating point values).
The task graph generated from the simulation code is not suitable for scheduling to multiprocessors, using standard scheduling algorithms found in literature. There are several reasons for this, the major reason is that the task graph is too fine grained and contains too many dependencies for getting good results on standard scheduling algorithms. Many scheduling algorithms are designed for coarse grained tasks. The granularity of a task graph is the relation between the communication cost between tasks and the execution cost of tasks. The large amount

of dependencies present in the task graphs also makes it necessary to allow task duplication in the scheduling algorithm. There are several scheduling algorithms that can handle fine grained tasks as well as coarse grained tasks. One such category of algorithms is non-linear clustering algorithms [18, 19]. A cluster is a set of nodes collected together to be executed on the same processor. Therefore all edges between two nodes that belong to the same cluster has a communication cost of zero. The non-linear clustering algorithms consider putting siblings[1] into the same cluster to reduce communication cost. But these algorithms does not allow task duplication. Therefore they are not producing well on this kind of simulation code.

A second problem with the task graphs generated is that in order to keep the task graph small, the implementation does not allow a task to contain several operations. For instance, a task can not contain both a multiplication and a function call. The simulation code can also contain Modelica `when` statements, which can be seen as a `if` statement without `else` branch. These need to be considered as one task, since if the condition of the `when` statement is true, all statements included in the `when` clause should be executed. An alternative would be to replicate the guard for each statement in the `when` clause. This is however not implemented yet, since usually the when statements are small in size and the need of splitting them up is low.

To solve the problems above, a second task graph is built, with references into the original task graph. The implementation of the second task graph makes it possible to cluster tasks into larger ones, thus increasing the granularity of the task graph. The first task graph is kept, since it is needed later for generating code. The two task graphs are illustrated in Figure 5.

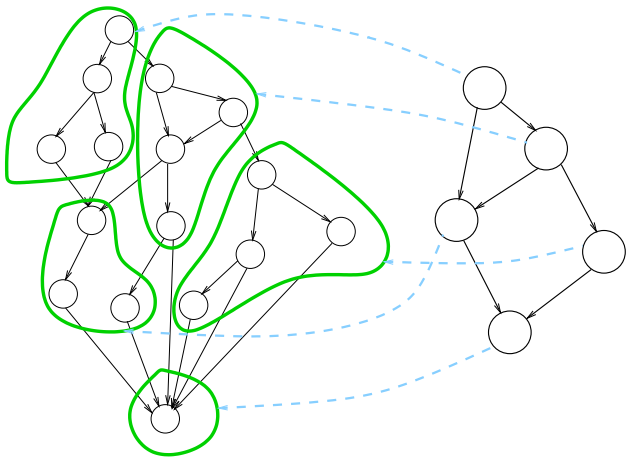

Fig. 5. The two task graphs built from the simulation code.

[1] A sibling *s*, to a task *n* is defined as a node where *n* and *s* has a common predecessor.

In this framework we have investigated several scheduling algorithms. Our early approaches found that the a task duplication algorithm called TDS [3] did not produce well. The main reason for this was the task granularity. The TDS algorithm can produce the optimal schedule if the task graph has certain properties, however fine grained task graphs as produced by our tool do not possess these properties.
We have also partially implemented a non-linear clustering algorithm called DSC [18], but measurements from this were not satisfying either. The clusters produced by the algorithm were too small, giving a parallel execution time much larger than the sequential execution time.
The combination of large fine grained task graphs and many dependencies makes it hard to find a low complexity scheduling algorithm that produce well. However, an approach that actually did produce speedup in some cases is a method we call *full task duplication*. The idea behind full task duplication is to prevent communication in a maximum degree, and instead of communicating values duplicate the tasks that produces the values. Figure 6 illustrates how the full task duplication algorithm works. For each node without successors, the complete tree of all its predecessors are collected into a cluster. Since all predecessors are collected no communication is necessary. The rationale of this approach is that, given a large fine grained task graph with many dependencies, it is cheapest to not communicate at all, but instead duplicate. The method also works better if the height of the task graph is low in relation to the number of the nodes, since the size of each cluster is dependent of the height of the task graph and the number of successors of each node.

Fig. 6. By duplicating all predecessors, each cluster forms a tree

When all nodes without successors have been collected into clusters, a reduction phase is started to reduce the number of clusters until the number of clusters

matches the number of available processors. First, clusters are merged together as long as they do not exceed the size of the largest cluster. In this way, the clusters are load balanced since each cluster will be limited by the maximum cluster size. If the number of clusters is still larger than the number of available processors, the algorithm selects two clusters with the highest number of common nodes and merge them together. This is then repeated until the number of clusters matches the number of available processors.

4 Results

The early attempts of implementing standard scheduling algorithms did not produce speedup at all. Therefore, the full task duplication method was invented. The results we present here are theoretical results achieved by running the algorithm and measuring the parallel time of the program by calculating the size of the largest cluster. In the future, when we have fully implemented code generation with MPI calls, we will run the simulation code on different parallel architectures like Linux clusters and SMP (Shared Memory Processors) machines.

Figure 7 gives some theoretical results for a couple of different models. The *Pressurewave* examples are a thermo-fluid application where hot air is flowing through a pipe. The pipe is discretisized into 20 and 40 elements in the two examples. The *Robot* example is a multi-body robot with three mechanical joints with electrical motors attached to each joint. As shown in Figure 7, the results are better for the discretisized models than for the robot model. For the robot model, new scheduling methods are surely needed to get a speedup at all. The full task duplication algorithm could not produce better speedup than 1.27 for two processors.

The speedup figures are calculated as $speedup = P_{seq}/(P_{par} + CommCost)$ where P_{seq} is the sum of the execution cost of all nodes and P_{par} is the maximum cluster size. $CommCost$ is for simplicity reasons assumed to be zero, even if it is a large cost. Since these figures are only measurements on the scheduled task graph and not real timing measurements from a running application, we can make this simplification. The focus here is on explaining the complexity and special needs of scheduling simulation code from code generated from Modelica models.

Speedup	n=2	n=4	n=6	n=8
Pressurewave20	1.54	2.62	3.60	4.45
Pressurewave40	1.52	2.82	3.21	5.13
Robot	1.27	-	-	-

Fig. 7. Some theoretical speedup results for a few examples.

5 Discussion and Future Work

Simulation code generated from equation based simulation languages is highly optimized and very irregular code. Hence, it is not trivial to parallelize. The scheduling algorithms found in literature are not suited for fine grained task graphs of the magnitude produced by our tool. Therefore, new clustering techniques with task duplication are needed.
Due to the large task graphs, caused by the large simulation code files, the clustering algorithm must be of low complexity. The large number of dependencies in the task graphs also requires us to use task duplication to further decrease the parallel time of a task graph. Therefore, future work includes finding new scheduling algorithms that are suited for large fine grained task graphs with many dependencies and that use task duplication, and still have a low complexity.
The full task duplication algorithm can be further improved by cutting trees of at certain points and instead introduce communications. This will be further investigated in the near future to see if it is a fruitful approach to further reduce the parallel time of a simulation program.
Future work also includes parallelization of code using inline solvers and mixed mode integration [2]. This means that the system can be partitioned into parts, each with its own inline solvers, which reduces the dependencies between these parts. This will hopefully reveal more parallelism in the task graph, which will improve our results.

6 Acknowledgments

This work started with support from the Modelica Tools project in Nutek, Komplexa Tekniska System and continues in the EC/IST project RealSim.

References

1. A. Radulescu, A. J.C. van Gemund. FLB:Fast Load Balancing for Distributed-Memory Machines. Technical report, Faculty of Information Technology and Systems, Delft University of Technology, 1999.
2. A. Schiela, H. Olsson. Mixed-mode Integration for Real-time Simulation. In P. Fritzson, editor, *Proceedings of Modelica Workshop 2000*, pages 69–75.
3. S. Darbha, D. P. Agrawal. Optimal Scheduling Algorithm for Distributed-Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, vol. 9(no. 1):87–94, January 1998.
4. C. Hanen, A. Munier. An approximation algorithm for scheduling dependent tasks on m processors with small communication delays. Technical report, Laboratoire Informatique Theorique Et Programmation, Institut Blaise Pascal, Universite P.et M. Curie, 1999.
5. C.Y. Lee, J.J. Hwang, Y.C. Chow, F.D Anger. Multiprocessor Scheduling with Interprocessor Communication Delays. *Operations Research Letters*, vol.7(no. 3), 1988.
6. *Dymola, http://www.dynasim.se.*
7. H. Elmqvist. *A Structured Model Language for Large Continuous Systems.* PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978.

8. P. Fritzson, V. Engelson. Modelica - A Unified Object-Oriented Language for System Modeling and Simulation. In *Proceedings of the 12th European conference on Object-Oriented Programming*, 1998.
9. P. Fritzson, L. Viklund, J. Herber, and D. Fritzson. High-level mathematical modeling and programming. *IEEE Software*, vol. 12(no. 4):77–87, July 1995.
10. G. Sih and E. Lee. Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*, vol. 4(no. 2), 1993.
11. G.L. Park, B. Shirazi, J. Marquis. DFRN: A New Approach for Duplication Based Scheduling for Distributed Memory Multiprocessor Systems. In *Proceedings of Parallel Processing Symposium, 1997.*
12. J.J. Hwang, Y.C. Chow, F.D. Anger, C.Y. Lee. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *Journal on Computing*, vol. 18(vol. 2), 1989.
13. M. Y. Wu, D. Gajski. Hypertool: A Programming Aid for Message-Passing Systems. *Transactions on Parallel and Distributed Systems*, vol. 1(no. 3), 1990.
14. *MathModelica, http://www.mathcore.se.*
15. *The Modelica Language, http://www.modelica.org.*
16. R.L. Graham, L.E. Lawler, J.K. Lenstra and A.H. Kan. Optimization an Approximatioin in Deterministic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics*, pages 287–326, 1979.
17. S. Darbha, D. P. Agrawal. Optimal Scheduling Algorithm for Distributed-Memory Machines. *Transactions on Parallel and Distributed Systems*, vol. 9(no. 1), 1998.
18. T. Yang, A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *Transactions on Parallel and Distributed Systems*, vol. 5(no. 9), 1994.
19. V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, MA, 1989.
20. B. E. Wells. A Hard Real-Time Static Task Allocation Methodology for Highly-Constrained Message-Passing Environments. *The International Journal of Computers and Their Applications*, II(3), December 1995.
21. Y-K. Kwok, I. Ahmad. Exploiting Duplication to Minimize the Execution Times of Parallel Programs on Message-Passing Systems. *Transactions on Parallel and Distributed Systems*, vol. 9(no. 9), 1998.

Structured Exception Semantics for Concurrent Loops

Joel Winstead and David Evans
{jaw2u,devans}@cs.virginia.edu

University of Virginia, Department of Computer Science

Abstract. Concurrent languages have offered parallel loop constructs for some time to allow a parallel computation to be expressed in a simple and straightforward fashion. Modern programming languages include exceptions to allow for clean handling of errors or unexpected conditions, but few concurrent languages incorporate exception handling into their models for parallel loops. As a result, programmers that use parallel loops cannot use exceptions to simplify their programs. We present a semantics for handling exceptions in parallel loops that is predictable and that reduces to the familiar semantics for sequential loops. This semantics provides guarantees about the behavior of parallel loops even in the presence of exceptions, and facilitates the implementation of parallel algorithms. A Java library implementation of this semantics is presented, along with a description of a source-to-source translation.

1 Introduction

Exceptions generated in parallel loops create problems that can be difficult to resolve. Parallel loop semantics allow more than one iteration of a loop to be executed at a time. Because any statement can generate an exception, this makes it possible for more than one unhandled exception to be raised concurrently in the same loop, a situation that does not occur in sequential loops. However, the exception semantics of most languages do not allow more than one exception to be raised at a time. It is not clear how to deal with this situation in a consistent way. It may not be consistent with the language's exception model to propagate both exceptions; if only one exception is allowed, one must be chosen and the others ignored, and there may be consequences for ignoring an exception generated by the program. Although sequential loops simply stop executing once an exception occurs, there are several ways abnormal termination could be handled in the parallel case, and one must be chosen that is reasonable and understandable.

Exception handling should be well-integrated with other parts of the language and exceptions should be handled in a consistent way regardless of the context in which they are generated. One proposed way of dealing with the problem of exceptions in parallel loops is simply to forbid them, and treat any exception that reaches the top level of a parallel loop as a fatal error. This is an unsatisfactory solution because exceptions in the context of a parallel loop are not handled in a

manner that is consistent with the way exceptions in other contexts are handled, and it does not allow them to be caught by handlers higher in the call chain than the parallel loop, even if matching handlers exist. Ideally, an uncaught exception generated in a parallel loop should propagate out of the loop like any other exception, and be handled in a way that is consistent with the way exceptions are handled in other contexts.

Even in the event of exceptions, a loop may produce partial results that are still useful, so it is important to be able to make strong assertions about the state of the program after a parallel loop has terminated exceptionally. A good semantics for exception handling in parallel loops should provide a way to assert strong postconditions for both normal and exceptional loop terminations.

We propose a semantics for exceptions in parallel loops that solves these problems by always propagating the exception that occurs structurally first in the loop, not the exception that occurs first in time. This removes the nondeterminism that results when more than one iteration of the loop throws an exception, and allows exceptions to be handled in a manner that is consistent with the way exceptions are handled in sequential loops. It allows stronger postconditions to be asserted about the result of the loop even in the case when an exception is thrown, because it allows the exception handler to know that all structurally prior iterations of the loop have completed without exceptions.

The semantics presented here should be useful for scientific computing and for applications where parallelism in the program is used to improve performance, and where partial results are useful. It allows a large class of sequential loops to be parallelized with predictable and consistent semantics even in the presence of exceptions. It may be useful for debugging these kinds of applications, even where partial results are not useful, because it allows determinism in the case of multiple exceptions. It may also be useful in libraries where exceptions are caused not by programming errors in the library, but by bad data passed to the library. Because it integrates the exception semantics with the concurrency constructs, a programmer need not know whether a method in a library uses parallel loops in order to write correct code that uses exceptions.

Our semantics is most likely not useful for systems programming or real-time systems where concurrency is an inherent part of the problem to be solved, rather than a means to better performance. These kinds of computations do not benefit from parallel loops, and are better expressed as separate routines executing concurrently rather than one loop executing in parallel. The kind of concurrency normally provided by Java Threads [1] or Ada tasks [2] is probably more appropriate for this kind of system.

2 Parallel Loops

A parallel loop is a `for` loop in which the iterations of the loop execute concurrently rather than sequentially. When the program reaches a parallel loop it spawns multiple threads to execute the iterations of the loop, and requires that all threads complete before the program continues with the next statement. Each

thread within the loop receives its own copy of the iteration variable so that the threads can perform independent computations. For example, the following loop computes the vector sum of two arrays in parallel:

```
parfor (int i=0; i<N; i++)
    a[i] = b[i] + c[i];
```

Each iteration of the loop has its own value for i and operates on a different part of the data. Assuming that a does not share storage with b or c, all iterations of the loop can execute simultaneously. The program does not return to the surrounding block and continue with the next statement until all iterations have completed. This particular loop can be executed efficiently in parallel because it has no data dependencies between iterations.

Parallel loops may be either synchronous or asynchronous. Synchronous loops imply synchronization between statements in different iterations of the loop; the form of the implied synchronization varies from language to language, but all generally require each iteration to execute the same statement at the same time. A Fortran `forall` loop containing a single assignment statement as its body, for example, requires that the value of the right-hand sides of the assignments must be computed first for every iteration of the loop before any actual assignments are made, thus avoiding potentially harmful effects. This semantics is particularly well-suited to vector machines.

Asynchronous loops do not have any implicit synchronization between statements in different iterations. This allows the iterations of the loop body to proceed independently of one another. Explicit synchronization statements can be used to provide a stronger ordering on loops that require it. Asynchronous loop semantics are well-suited to systems that implement parallelism as threads that are scheduled independently. Asynchronous loops have appeared in Compositional C++ [3], Modula-3* [4], and other parallel languages.

Some implementations, such as the `forall` loops in Fortran 95 [5], require that the number of iterations of the loop, and their index values, be known when the loop starts; this simplifies the implementation and allows all iterations to start immediately, at the cost of flexibility. Other parallel loop constructs, such as the `parfor` construct in Compositional C++, do not have this restriction, and have the control portion of the loop execute sequentially while spawning threads to execute the body of the loop [3]; this allows loops where the number of iterations is not known in advance, or that do not have integer indices.

3 Exception Semantics for Parallel Loops

Although parallel loops have been implemented in a number of languages, few have been implemented in languages that have exceptions, and even fewer have attempted to address the semantics of an uncaught exception that occurs within a parallel loop. Exceptions in parallel loops introduce several difficult situations that must be addressed, especially when exceptions occur within more than one iteration of the loop. When an exception occurs within a sequential loop, the

loop simply stops executing and the exception is propagated to the calling block; because of sequential loop semantics, the handler may assume that this was the only exception that occurred, and that all iterations up to the exception completed normally. When an exception occurs within a parallel loop, the iteration that generated the exception necessarily stops executing, but it is less clear what to do about the other iterations, and how or if the exception should be propagated. The question of what to do if more than one iteration of the loop throws an exception is a difficult issue as well, because the exception semantics of most languages do not allow two exceptions to be raised simultaneously.

3.1 Goals for Exception Semantics

A simple approach to the problem of exceptions in parallel loops is to ignore uncaught exceptions in concurrent code, or to forbid them altogether. The designers of Ada, when confronted by the issues exceptions raise in a concurrent context, chose to ignore uncaught exceptions that reach the top level of a task. If an uncaught exception occurs in an Ada task, the task terminates without handling the exception or passing it on to an outer block that can handle it; this policy was chosen to prevent the problems that would result if the exception was passed to a parent task asynchronously [6]. The designers of pSather chose to forbid exceptions in a concurrent context [7] [8] [9]. These solutions deny concurrent programmers the expressive and robustness benefits of exceptions.

Programmers should be able to use exceptions in concurrent constructs as they would normally in the programming language, and be able to reason about their behavior. A good semantics for uncaught exceptions in parallel loops should be consistent. If the exception which is propagated from the loop is chosen nondeterministically, this could make the program difficult to reason about. Some forms of nondeterminism are unavoidable and even desirable in an asynchronous loop, because the parallel loop semantics impose only a partial order on the statements in the loop; however, the ability to predict and reason about the exception, if any, produced by the loop would make programs easier to write.

In order to recover from an exception in a loop, it is important to know the state of the program after loop termination. A good exception semantics should allow a strong postcondition to be asserted about the state of the program after an exception has been thrown. This postcondition can then be assumed by the exception handler, which can use the information in its recovery.

An exception semantics should also enable a clear termination condition for the loop. If an iteration within a loop terminates with an exception, and other threads within the loop depend on the normal completion of that iteration, a deadlock could result. A good exception semantics should provide a way to prevent this sort of problem from occurring whenever possible.

3.2 Exception Propagation

Java, C++, and other object-oriented languages which have exceptions can only raise one exception at a time. In a parallel loop, however, it may be possible to

raise multiple exceptions within the same block of code. It is important to have a clear semantics for what should happen in this case.

One possibility is to handle each exception in the loop as it happens, while allowing other iterations in the loop to continue, possibly generating more exceptions. This solution would create consistency problems because it would allow code within the loop to run at the same time as code in outer blocks, and could even result in destroying stack frames that are part of the loop's context. This would create code safety problems, and is inconsistent with the concurrent loop semantics defined in the previous section, because the semantics state that all iterations of the loop must complete before any outside exception handler begins.

Another possibility is to merge all of the exceptions together into a single exception that contains an array storing the exceptions generated by each iteration of the loop. This allows all exception information to be kept. The cost of this approach is that each iteration of the loop must be allowed to run to completion in order to determine which exceptions will be thrown. This could result in deadlock if there are dependencies between events in different iterations of the loop. Code written to catch and handle exceptions would need to be more complicated, because either `catch` expressions would need to be able to match patterns in the array, or some mechanism would be required to apply the chain of exception handlers to each exception in the array. This would be particularly complex if some exceptions were handled higher in the call chain, while others were handled at lower levels.

A scheme for collapsing multiple exceptions into a single exception in a Modula-3* parallel loop is described by Heinz [10]. If a single exception is generated by the loop, or if all exceptions in the loop are identical, a single copy of the exception is propagated out of the loop. If the exceptions are not all identical, a special exception is generated to note the inconsistency. This approach allows multiple exception cases to be handled in languages that can only raise a single exception at a time. However, it requires all iterations of the loop to complete, and requires programmers to handle the case of inconsistent exceptions.

In many cases, if multiple exceptions occur in a parallel loop, the exceptions are all related to the same problem, and it is only necessary to propagate one of the exceptions to ensure a proper recovery; other exceptions can be discarded. Philippsen and Blount describe an implementation of asynchronous parallel loops in Java that uses this approach [11] [12]. The first exception that occurs in time is propagated out of the loop; other iterations are canceled, and any exceptions they generate are ignored. This approach loses some information and introduces nondeterminism in the exception that is returned, but keeps within the single-exception model of the language, and does not require every iteration of the loop to complete.

Our approach is to return not the first exception that occurs in time, but the exception that occurs structurally first in the loop. This removes the nondeterminism in the exception that is propagated, and loops can be written in a way that guarantees that the most important exception, if any, will be the one that is propagated out of the loop. Our semantics requires structurally prior

iterations of the loop to complete so that any structurally earlier exceptions can be obtained. Structurally later iterations, however, can be canceled, thus eliminating deadlock in situations where no iteration of the loop waits for an event in a structurally later iteration.

3.3 Cancellation and Loop Termination

In order to determine what exception semantics is most appropriate for parallel loops, the termination condition imposed by each exception semantics must be considered. A straightforward solution is to require all iterations of the loop to terminate, even in the event of an exception. This approach is used by Heinz in Modula-3* loops [10]. In cases where some iterations of the loop wait or depend on events that occur in other iterations, the loop may deadlock. In particular, if an iteration of the loop terminates with an exception before creating an event that some other iteration is waiting for, the loop will deadlock because the second iteration is waiting for an event that will never occur. This situation can be created by the use of mutual exclusion for a shared resource, which is common in asynchronous parallel programs. Requiring all iterations of the loop to terminate even in the event of an exception is practical if it is known that there is no communication or explicit synchronization between iterations, but many loops do not satisfy this condition, so this solution cannot be used for a general parallel loop.

Another solution is to cancel some iterations of the loop in the event of an exception in order to guarantee that the loop terminates. One approach is to cancel all iterations in the loop other than the one that generated the exception. This approach guarantees that the loop will terminate in the event of an exception. Because it cancels some iterations, this cancellation strategy is not consistent with any approach to exception propagation that requires all iterations to complete. In addition, because this approach cancels both structurally prior and structurally later iterations, it cannot be used if the semantics requires the structurally first exception from the loop to be propagated. This approach is consistent with an exception propagation semantics that requires the first exception in time to be propagated.

Because canceling structurally prior iterations is inconsistent with a semantics that propagates the structurally first exception, we adopt a cancellation strategy that cancels structurally later iterations only. Because it requires some iterations of the loop to complete, it can result in deadlock if an iteration waits for an event that is generated by a structurally later iteration; however, if no iteration waits for an event generated by a later iteration, this cancellation strategy guarantees that the loop will terminate in the event of an exception, provided that the structurally prior iterations themselves terminate. This approach is consistent with a semantics that propagates the structurally first exception out of the loop.

3.4 Loop Postconditions

In order to write correct programs, it is important be able to make assertions about the state of the program after loop termination, even in the event of an exception. The exception semantics directly affects the kinds of postconditions that can be asserted about a loop that terminates with an exception.

Because different iterations of a parallel loop may share some context and data, and there is no explicitly defined order between events in one iteration of the loop and events in another, the postcondition that can be asserted of one iteration of the loop is not necessarily the same as the strongest postcondition that could be asserted of the same sequence of statements if executed in a sequential context. The postcondition for each iteration of the loop must be true for any possible interleaving of the different iterations of the loop and any potential interactions between them, and may not assume anything about the progress of other iterations. For example, consider the following loop:

```
parfor (int i=0; i<100; i++)
    A[i] = i*i;
```

If this loop were executed sequentially, the postcondition of iteration $i = 0$ would be $A[0] = 0$ with all other elements of A unchanged, or $A[0] = 0 \wedge \forall i : i > 0 : A[i] = A_{pre}[i]$. It it were executed concurrently, however, the strongest postcondition that can be asserted for iteration $i = 0$ is $A[0] = 0 \wedge \forall i : 0 < i < 100 : (A[i] = A_{pre}[i] \vee A[i] = i^2) \wedge \forall i : i >= 100 : A[i] = A_{pre}[i]$. This weaker postcondition takes into account the fact that the other iterations may not have completed by the time iteration $i = 0$ has. If different iterations of the loop attempted to write different values to the same variable without explicit synchronization, the postconditions for all iterations of the loop must account for each possible ordering of the writes.

The loop guarantee is the strongest condition that can be asserted of the loop at all times, and must allow for every possible interleaving of iterations, and every possible partial result. For this loop, this is $(\forall i : 0 \leq i < 100 : A[i] = A_{pre}[i] \vee A[i] = i^2) \wedge (\forall i : i \geq 100 : A[i] = A_{pre}[i])$, expressing the fact that any combination of the iterations may have completed.

The postcondition of the loop as a whole is the conjunction of the loop guarantee and the postconditions of all iterations of the loop that are known to have completed. For the above example, this is $(\forall i : 0 \leq i < 100 : A[i] = i^2) \wedge (\forall i : i \geq 100 : A[i] = A_{pre}[i])$, assuming that all iterations complete. The terms in the individual iteration postconditions that expressed uncertainty about the progress of the other iterations are absorbed when the fact that each iteration has completed is included in the postcondition.

If one or more iterations of the loop terminate with exceptions, the postconditions for these iterations may not be true, and they cannot be used in the loop postcondition. The cancellation strategy may prevent further iterations from completing, and the postconditions of these iterations may not be used either. In addition, although the exception propagation policy does not affect which iterations will complete, it may limit the knowledge the exception handler has

of the completion status of the loop. For any iteration that does not complete, the strongest postcondition that can be asserted is the uncertainty about how much progress the iteration made, which includes each potential partial result of the iteration; this condition of uncertainty is included in the loop guarantee.

If the first exception thrown in time is propagated from the loop, and other exceptions are ignored, an exception handler may not assume anything about the termination of other iterations of the loop, regardless of the termination strategy used. Even if the cancellation strategy allows other iterations of the loop to complete, it is possible that some of them may terminate exceptionally, and it would not be safe for the exception handler to assume anything about them. In this case, no iteration of the loop can be assumed to have completed, and no postcondition can be asserted of the loop that is stronger than the loop guarantee.

If the structurally first exception is propagated from the loop, and other exceptions are ignored, then all iterations structurally prior to the one that terminated abnormally are known to have completed. However, nothing can be asserted about the completion status of any other iterations, because any of them may have terminated exceptionally. If this exception propagation policy is used, the strongest postcondition that can be asserted of the loop as the exception handler is entered is the conjunction of the loop guarantee with the postconditions of all iterations structurally prior to the one that terminated with the exception. In the above example, if an exception occurred in iteration $i = 50$, the exception handler could assume $(\forall i : 0 \leq i < 50 : A[i] = i^2) \wedge (\forall i : 50 \leq i < 100 : A[i] = A_{pre}[i] \vee A[i] = i^2)$.

Because propagating the structurally first exception in the loop yields the most useful loop postcondition, and because canceling structurally later threads in the loop increases the class of problems for which the loop terminates in the exception case, we propose a semantics for parallel loops that propagates the structurally first exception. Under this semantics, the effect and postcondition of a loop that terminates with an exception is determined by the structure of the loop, not by the accident of its execution.

4 Extending Java with Parallel Loops

We chose to implement this semantics for parallel loops using Java as the base language. Although Java is not as commonly used for scientific computing as C++, its language specification includes concurrency, and its class hierarchy in which all exceptions are derived from a single `Throwable` type makes it possible to implement the extensions using a source-to-source translation. Because the C++ specification does not offer these features, a C++ implementation could probably not be done as a source-to-source translator. While we chose Java for its simplicity, we believe these ideas could be implemented in C++ or other object-oriented languages.

Concurrency in Java is expressed using objects, rather than through a control flow construct. Each thread of control in the program is implemented as a sep-

arate object that either extends the `Thread` class or implements the `Runnable` interface and has a controlling `Thread` object. Every object in Java has its own associated monitor which is used for synchronization. There are no explicit parallel constructs similar to parallel loops. This model is well suited for writing programs with separate, unrelated tasks that execute concurrently, such as having one thread compute in the background while another thread handles the user interface. This technique is used mainly to separate the tasks in an application and to reduce latency in user interfaces. This thread model is not well suited for data parallelism, where many threads are to perform the same or similar operations on different pieces of data as part of the same task, because the constructs for declaring, starting, and joining tasks are not control flow constructs that can easily be used in an algorithm description.

In order to define a `parfor` loop for Java that uses the structured exception semantics, we must modify the Java language semantics for a `for` loop. The Java Language Specification [13] defines a sequential `for` loop in terms of a condition *Expression*, a *ForUpdate* statement, and a contained body *Statement*:

- If the *Expression* is present, it is evaluated, and if evaluation of the *Expression* completes abruptly, the `for` statement completes abruptly for the same reason. Otherwise, there is then a choice based on the presence or absence of the *Expression* and the resulting value if the *Expression* is present:
  - If the *Expression* is not present, or it is present and the value resulting from its evaluation is `true`, then the contained *Statement* is executed. Then there is a choice:
    - If execution of the *Statement* completes normally, then the following two steps are performed in sequence:
      - First, if the *ForUpdate* part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If evaluation of any expression completes abruptly for some reason, the `for` statement completes abruptly for the same reason; any *ForUpdate* statement expressions to the right of the one that completed abruptly are not evaluated. If the *ForUpdate* part is not present, no action is taken.
      - Second, another `for` iteration step is performed.
    - If execution of the *Statement* completes abruptly, see §14.13.3 below.
  - If the *Expression* is present and the value resulting from its evaluation is `false`, no further action is taken and the `for` statement completes normally.

In order to achieve parallel execution of the loop, we must modify this definition so that instead of executing the *Statement* directly, it starts a thread which executes the *Statement* concurrently. The *Statement* thread receives its own copy of the iteration variable so that it may execute independently of the loop control and the other iterations. In addition, we must add a rule that states that when the loop completes, either normally or abruptly, it must wait for all concurrent *Statement* threads to complete.

These changes allow the loop to execute concurrently, but further changes are needed to handle abrupt termination in the event of an exception. Section 14.13.3 of the specification describes abrupt termination of a `for` loop in the case of a `break` or `continue` statement, and states that "If execution of the *Statement* completes abruptly for any other reason, the `for` statement completes abruptly for the same reason." For simplicity, we do not allow `break`, `continue`, or `return` statements in `parfor` loops, although they could be introduced in ways consistent with our semantics. Hence, the only form of abrupt termination we consider here is through exceptions. In the event of an abrupt termination, the control portion of the loop should complete immediately, and no further *Statement* threads should be started. Any running *Statement* threads that were started after the *Statement* that generated the exception should be interrupted, which will allow them to terminate cleanly. Once all *Statement* threads have completed, the loop should terminate by propagating the structurally earliest exception.

Together, these changes produce the following specification for a Java `parfor` loop:

- First, the `parfor` control loop is executed:
  If the *Expression* is present, it is evaluated, and if evaluation of the *Expression* completes abruptly, the `parfor` control completes abruptly for the same reason. Otherwise, there is then a choice based on the presence or absence of the *Expression* and the resulting value if the *Expression* is present:
  - If *Expression* is not present, or it is present and the value resulting from its evaluation is `true`, and no existing *Statement* thread has terminated exceptionally, then a thread is started to execute the contained *Statement* concurrently with the `parfor` control. A copy is made of the iteration variable, the new thread uses the copy.
    * Then the `parfor` control performs the following two steps in sequence:
      · First, if the *ForUpdate* part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If evaluation of any expression completes abruptly for some reason, the `parfor` control completes abruptly for the same reason; any *ForUpdate* statement expressions to the right of the one that completed abruptly are not evaluated.
      · Second, another `parfor` control step is performed.
  - If the *Expression* is present and the resulting value of its execution is `false`, no further action is taken and the `parfor` control completes normally.
- If a *Statement* thread terminates abruptly, all running *Statement* threads that were started after the one that terminated abruptly are interrupted. The `parfor` control then completes without starting any new *Statement* threads.
- When the `parfor` control completes, it must wait for all of its *Statement* threads to complete before terminating. If any errors or exceptions occurred

in the execution of either the `parfor` control or any *Statement* threads, the error or exception from the earliest iteration is propagated as the reason for the `parfor` statement's termination.

To illustrate the semantics, consider the array assignments example introduced in §3.4. If A is an array of size 50 instead of size 100, any attempt to access elements 50 through 99 will result in an `ArrayBoundsException`. Because the iterations of the loop execute in parallel, however, it is not possible to know in advance which iteration will cause an exception to be thrown first, nor is it possible, once that statement has been reached, to know which iterations of the loop have already completed or how much of the array has been filled.

With the proposed semantics for abnormal loop termination, although it cannot be known which iteration will generate the first exception, we do know that the structurally first exception will be the one that is propagated, and that all lower iterations will complete before the loop terminates. This means that even though we cannot know whether iteration $i = 50$ or iteration $i = 60$ will generate an exception first, we do know that the `ArrayBoundsException` generated by iteration $i = 50$ will be the one propagated by the loop, and the handler for that exception may safely assume that iterations $i = 0..49$ have completed and that the array up to the point the exception occurred has been filled, and this can be asserted as a postcondition of the loop.

The result of an exception with this semantics is consistent with the semantics of a normal sequential `for` loop. In a normal `for` loop, only one iteration of the loop executes at a time, and the iterations execute in order. If an exception is thrown in one iteration of the loop, the loop terminates immediately and propagates the exception to the calling block, and no further iterations of the loop are executed. The program may safely assume that all iterations up to the iteration that threw the exception have completed normally, and the handler for the exception can use this information when recovering. Even if there are additional problems that would have caused higher-numbered iterations of the loop to throw exceptions, only the first such problem is caught and reported. Our proposed exception semantics for parallel loops shares these properties: the lower-numbered iterations of the loop are guaranteed to have terminated normally, and the exception that is propagated is the one from the lowest-numbered iteration that threw an exception.

4.1 Handling Exceptions from Parallel Loops

In order for an exception handler to be able to make use of the loop postcondition described above, it must know how much of the loop completed and what iteration it was that generated the exception. To support this, we allow `catch` and `finally` clauses to be attached directly to `parfor` statements. The iteration variable for the loop remains in scope within these clauses. Because more than one copy of the iteration variable exists in the parallel loop semantics, we must specify what the iteration variable in the exception handlers refer to.

If an exception occurs within the body of the parallel loop, the value of the iteration variable in the handler should be the same as the value of the iteration variable in the instance of the body that raised the exception. If an exception is generated in the sequential control portion of the loop, then the handlers should see the current value of the iteration variable from the control portion when it stopped. This allows the handler to know which iteration caused the exception, so that it can use this information when recovering from the exception.

If an exception occurs in the initialization statement of the loop, which declares the iteration variable, the variable is undefined, and the `catch` and `finally` clauses of the loop cannot be allowed to execute because the result would be undefined and could introduce a code safety issue. For these purposes, the initialization statement can be considered to be outside of the loop, and outside of any attached `catch` and `finally` clauses. Exceptions generated by the initialization statement can still be handled, but they must be handled outside of the loop.

If no `catch` clause attached to the loop handles the exception, the exception will propagate up the call chain normally, and the information about which iteration caused the exception will be lost. An alternative might be to include the information about which loop and which iteration threw the exception in the exception object itself. However, this is undesirable. It would require a fundamental change to Java's exception model which would affect much more than just loops, and that would make the parallel exception semantics inconsistent with all normal Java programs. Furthermore, it is unlikely that the information about which iteration caused the exception would be useful outside of the context of the loop.

Although an exception handler inside the loop body, rather than outside the loop, would also have access to the value of iteration variable, such a handler would not have the knowledge that all iterations of the loop have completed. An exception handler outside the loop body, but without access to the iteration variable, would not be able to take advantage of the fact that all structurally prior iterations have completed normally. The possibility that an exception handler could use both the value of the iteration variable and the knowledge that all loop iterations have completed (and structurally prior iterations completed normally) justifies allowing a special handler to be attached to the loop which preserves the value of the iteration variable.

4.2 Safe Thread Cancellation

The semantics require that all iterations structurally prior to an iteration that causes an exception must be allowed to complete, because they may also throw exceptions, and we want to propagate the exception that is structurally first, not first in time. Any higher iterations that have not yet started will never be started, and higher iterations that are already running are interrupted. The purpose of this is to cancel those iterations that occur logically after the exception and are therefore invalid.

Cancellation of running threads is difficult to do safely in an object-oriented language [7]. Simply terminating a thread with no mechanism to allow cleanup is unsafe and deadlock-prone, because the thread may be holding locks or may have temporarily placed an object in an inconsistent state. For this reason, a mechanism to allow the thread to exit cleanly is needed.

One mechanism for terminating a thread provided by early versions of Java allowed an exception to be delivered to a thread asynchronously, though the `Thread.stop()` method. This is dangerous, however, because it can deliver an exception to code that was not designed to handle that particular exception. Although it is theoretically possible to write code that can handle asynchronous exceptions, it is very cumbersome to read and very difficult to get correct, particularly when the exception is delivered in a `finally` clause or `synchronized` block. In any case, the mechanism is not guaranteed to cause the thread to stop: the thread's code could be written in a way to trap such exceptions and ignore them. Because of these problems, Sun chose to deprecate this feature of Java [14].

Instead, Sun recommends using the `Thread.interrupt()` mechanism to cancel running threads, which is safe because it causes the thread to be interrupted only at well-defined points in the code. Certain library routines can generate an `InterruptedException`, which is a checked exception which must be caught or declared. Other code can explicitly check to see if it has been interrupted by using the `Thread.interrupted()` call, and it is the responsibility of the programmer to make sure that the thread exits cleanly. Although it is possible for code to ignore the interruption and continue running, this is not worse than the `Thread.stop()` method, which cannot provide this guarantee either, and is much harder to use safely [1].

For these reasons, we attempt to terminate running iterations of the loop using the interrupt mechanism. Because poorly written code could ignore the interrupt mechanism, the semantics cannot provide a guarantee that canceled iterations will terminate promptly, or even that they will terminate at all; however, correct Java code should terminate when interrupted. Because the loop semantics require all iterations of the loop to terminate before the loop itself terminates, the semantics cannot guarantee for arbitrary loops that the loop as a whole terminates either, even when a loop iteration raises an exception. Care must be taken to make sure that loops are written such that they will terminate cleanly, even in the presence of exceptions. This is already the case for sequential loops, and for all Java code, so this limitation of the parallel loop semantics is not worse than the limitations of the exception semantics for any other construct of Java.

4.3 Example

Suppose we wished to use a computer to render frames of an animation. Because each frame is independent of other frames in the video, they can be rendered in parallel, saving computation time. A procedure to do this is:

```
void render_frames(Frame[] frames,int start,int end) {
    parfor (int i=start; i<end; i++) {
        images[i] = frames[i].render();
    }
}
```

This loop would render the separate frames of the sequence in parallel. However, if an exception occurred while rendering the frames, the loop would stop. Because many frames have already been rendered at this point, and because rendering is a computationally expensive task, we would prefer to keep the partial results even in the event of an exception, and use these when recovering from the exception. For example, if the machine runs out of memory while rendering the frames, we would like to be able to recover from this situation by saving the completed frames to disk, and then continuing where the program left off. It is advantageous to save a contiguous block of frames together because this maximizes the effectiveness of a compression algorithm. Our proposed exception semantics allows this to be expressed easily:

```
void render_frames(Image images[],Frame[] frames,
                   int start,int end) {
    parfor (int i=start; i<end; i++) {
        images[i] = frames[i].render();
    } catch (OutOfMemoryException e) {
        // Delete frames that come after the exception
        for (int j=i; j<end; j++) {
            images[j] = null;
        }
        // Save earlier, completed images
        save_images(images,start,i-1);
        // Delete completed images to recover memory
        for (int j=0; j<i; j++) {
            images[j] = null;
            frames[j] = null;
        }
        // Render remaining images through recursive call
        render_frames(images,frames,i,end);
    }
}
```

This example recovers from an exception by deleting all rendered images that come logically after the frame that caused the exception, and saving all images that come logically before the exception. Images that come logically after the exception are not saved, because they may not be complete, and would not be contiguous with the earlier frames. Because the postcondition guarantees that the exception thrown is the structurally first exception and that all prior iterations have completed, these images can be saved to disk together and do not

need to be recomputed. Because these images are in a contiguous block without holes, it should be possible to compress them easily; this would not be possible if all completed images were saved, including any completed frames logically after the image that caused the exception.

5 Implementation

The proposed language extensions can be implemented using a library of new classes that encapsulate the loop and exception semantics, and a source-to-source translator that translates the extensions into standard Java code that uses the library. The resulting code can then be compiled by any Java compiler. This chapter describes the implementation of the library, and the design of a source-to-source translator. The source-to-source translator for our extensions has not yet been implemented, but similar extensions have been implemented as source-to-source translators before, and implementation of our extensions would be straightforward.

5.1 Strategy

An extension to Java could be implemented as a Java library, a source-to-source translator, a full compiler generating standard JVM bytecode, or a compiler generating special bytecode for a modified JVM.

A modified JVM would allow the most flexibility in what can be implemented, but would come at the cost of portability and interoperability with existing Java code. The JVM's built-in `Thread` class could be redefined so that uncaught exceptions generated by the thread would be passed to the parent thread upon termination; the parent thread could receive the exceptions when it attempts to `join` the child threads. This implementation technique would require a compiler to recognize the `parfor` loops and generate the code needed to implement them.

A full compiler has the ability to perform some transformations that generate valid JVM bytecode but cannot be expressed in Java in source form, but is also complex and can introduce interoperability problems. A source-to-source translator cannot perform as many transformations as a full compiler, but because its output is standard Java source code, it does not introduce any portability or interoperability problems. In addition, the code generated by a source-to-source translator is readable, which makes debugging and analysis easier. For these reasons, we chose to implement the extensions as a library along with a description of a source-to-source translation. A more detailed analysis of these options for implementing parallel loops in Java is provided by Philippsen [11].

The implementation described here uses an inner class to represent each parallel loop, and a set of methods that spawn threads, execute the loop, and wait for the threads to terminate. If an exception is generated by the loop, the loop class waits for structurally prior iterations to complete, and re-throws the structurally first exception to the calling code. This allows the parallel loop

constructs to be implemented without changing the virtual machine, and allows code with parallel loops to work transparently with standard Java code.

Although we have not implemented an automatic source-to-source translator for our proposed extensions, we describe here how the extended features could be translated mechanically into normal Java code. Other researchers have implemented automatic translators that generate standard Java code for parallel loops using similar techniques [12] [11] [15] [16], and it should be straightforward to modify one of these translators to use our proposed exception semantics.

5.2 Loop Translation

To implement `parfor` loops, the `ParforLoop` class manages a single parallel loop and is responsible for creating threads, waiting for them to complete, catching exceptions, interrupting running threads, and propagating exceptions back to the caller. `ParforLoop` has undefined abstract methods for the condition expression, update statement, and loop body; each method takes an `Object` parameter that holds the current value of the loop's iteration variable. The class is not instantiated directly: in order to use the class, a child class must be derived from it that defines these methods for the particular loop that is to be executed.

A `parfor` loop is translated as an inner class that extends the `ParforLoop` class and defines the abstract methods, and executed by calling the ParforLoop class's `loop` method. This transformation requires a few fairly simple substitutions.

The loop is described by overriding the abstract `condition`, `update`, and `body` functions to contain the loop's condition expression, update statement, and body implementation, respectively. These functions each take a reference to the iteration variable as a parameter. The loop is actually executed by a call to the `ParforLoop.loop` method, which is passed the initial value of the iteration variable. This method executes the control portion of the loop and starts threads to execute the body of the loop, and does not return until the loop has terminated.

Each iteration of the loop is executed by an instance of the `StructuredThread` class. This class calls the `ParforLoop`'s `body` method, and catches any exceptions that occur. If an exception occurs, it passes it to the `ParforLoop` object's `catchException` method, which actually implements the exception semantics by storing the exception along with its iteration number in the `ParforLoop` object, and interrupting any structurally later threads that are still running.

Once all iterations have stopped executing, `ParforLoop.loop` propagates any exception that occurred by re-throwing it to the calling block. If the loop has any extended `catch` or `finally` clauses attached, for which the iteration variable is to remain in scope, the handlers described there are made part of a `try..catch` block surrounding the call to `loop`. The value of the iteration variable is retrieved by calling the `getIteration()` method of the `ParforLoop` object, so that the handlers may use this information in their recovery.

5.3 Performance

The overhead associated with our exception semantics should not significantly affect the performance of concurrent Java programs. The cost of setting up the loop and starting threads is linear in the number of iterations of the loop. In the normal case in which there are no exceptions, the cost of joining the threads in the loop and cleaning up is also linear in the number of iterations of the loop, and introduces little overhead.

If the loop terminates exceptionally, the running time may be longer than other methods because our semantics requires all earlier iterations of the loop to complete when an exception is thrown, while other semantics allow all threads to be terminated immediately in this situation. However, the running time of this should not be any worse than that of normal execution of the loop. It is possible that a loop which uses concurrency control constructs incorrectly or which ignores the `InterruptedException` may deadlock or enter an infinite loop if an exception occurs; however, this is the result of an incorrect program and not the result of the translation.

The translation does increase code size. Any program using the translation must use the `ParforLoop` library. The translation itself creates a new inner class and several methods for each loop, and creates additional code to handle each type of exception that could be expected from the loop. The increase in code size is linear in the number of translated `parfor` loops. Relaxation of the Java requirement that all exceptions must be caught or declared (except for errors and run-time exceptions) would make the translation simpler and would reduce the size of the translated code.

The library described here uses the simple approach of spawning a separate Java thread to execute each iteration of the body of the loop. Other methods of implementing parallel loops in Java achieve higher performance by not using a one-to-one mapping of loop iterations to threads; see [15] for a comparison. However, such a strategy must deal with the possibility that there are dependencies between different iterations, and it can be difficult to determine when two iterations can be executed sequentially and when they must be executed concurrently. The mechanisms needed to deal with this situation are complex. Because our purpose here is to demonstrate our proposed exception semantics, rather than how to achieve optimum performance, we chose the simpler one-to-one mapping of iterations to threads. Our exception semantics could be incorporated into an implementation that uses a more efficient mapping of iterations to threads.

6 Conclusions

The goal of our semantics for exceptions in parallel loops is to facilitate the writing of parallel programs by making them more predictable and understandable, while allowing exceptions to be used in a consistent way with other language features. Under our semantics, exceptions can be used in any context, and have a well-defined meaning even in parallel loops. Exceptions in parallel loops behave

in a manner analogous to exceptions in sequential loops, because in both cases the structurally first exception thrown is the one that is propagated out of the loop. The attempt to cancel later iterations through interruption is consistent with the fact that later iterations in a sequential loop do not run. The proposed exception semantics are a generalization of the standard Java semantics for sequential loops, and remain consistent with them.

Our semantics allow strong postconditions to be asserted in the presence of exceptions, because the exception that is propagated is chosen deterministically, based on the structure of the program rather than on the accident of its execution. This also provides repeatability for some kinds of errors in concurrent programs, which facilitates debugging. Other semantics for exceptions in parallel loops, such as propagating the first exception that occurs in time, do not have this property. Of course, there are other sources of nondeterminism in concurrent programs, and this semantics does not eliminate all of them; indeed, that would not be possible without giving up the benefits of parallelism. But it does eliminate some nondeterminism in a way that should make program behavior more predictable and easier to understand and reason about.

The semantics also allows stronger assertions about what partial results have been computed in the exception case. Keeping partial results in the case of an exception can save time when recomputing or recovering from the exception. Partial results may be useful in some cases even when the program does not attempt to repair and recompute after the exception has been handled. Other choices for an exception semantics do not allow strong postconditions to be asserted.

For some loops, it does not matter which exception is structurally first, or what partial results have been computed, and for these loops, propagating the first exception in time and canceling or interrupting all remaining threads in the loop may make more sense. One possibility is to use one keyword to indicate parallel loops for which partial results are useful and deterministic exception semantics is desired, in which our exception semantics would be used, and another keyword to indicate parallel loops which express inherently concurrent algorithms for which it is not important which exception is structurally first.

Our semantics is presented in the context of Java, but it could be applied to other languages. The partial results feature may be particularly helpful in implementing the `retry` keyword in languages that have one, such as Eiffel [17]. This would allow the loop to restart at the point the logically first exception occurred, while keeping previously computed partial results.

The ability to express concurrency in a clean and natural way is an important feature to have in modern programming languages. Parts of programs that express concurrency or parallelism should be well integrated with the rest of the language, and it is important for language designers to consider how concurrency interacts with exception handling in particular. We have presented a concurrency construct for Java that provides exception semantics that are consistent and integrated with the language, and that can be implemented with modest overhead.

Acknowledgements

This work was supported by NSF CCR-0092945 and NASA NRC 99-LaRC-4. The authors thank Paul Reynolds and John Thornley for helpful comments and suggestions.

References

1. Sun Microsystems: Java 2 Platform, Standard Edition, v 1.2.2 API Specification. (1999) http://java.sun.com/products/jdk/1.2/docs/api/index.html.
2. Barnes, J.G.P.: An overview of Ada. Software - Practice and Experience **10** (1980) 851–887
3. Carlin, P., Chandy, K.M., Kesselman, C.: The Compositional C++ language definition. Technical Report 1993.cs-tr-92-02, Department of Computer Science, California Institute of Technology (1993)
4. Heinz, E.: Modula-3*: An efficiently compilable extension of modula-3 for explicitly parallel problem-oriented programming. In: Joint Symposium on Parallel Processing, Tokyo, Waseda University (1993) 269–276
5. Adams, J.C.: Fortran 95 Handbook. MIT Press, Cambridge, MA (1997)
6. Ichbiah, J.D., Heliard, J.C., Roubine, O., Barnes, J.G.P., Krieg-Brueckner, B., Wichmann, B.A.: Rationale for the design of the ADA programming language. ACM SIGPLAN Notices **14** (1979) 1–247
7. Fleiner, C., Feldman, J., Stoutamire, D.: Killing threads considered dangerous (1996)
8. Murer, S., Feldman, J.A., Lim, C., Seidel, M.: pSather: Layered extensions to an object-oriented language for efficient parallel computation. Technical Report TR-93-028, International Computer Science Institute, Berkeley, CA (1993)
9. Stoutamire, D., Omohundro, S.: The pSather 1.1 manual and specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, CA (1996)
10. Heinz, E.A.: Sequential and parallel exception handling in Modula-3*. In Schulthess, P., ed.: Advances in Modular Languages: Proceedings of the Joint Modular Languages Conference, Ulm, Germany (1994) 31–49
11. Philippsen, M.: Data parallelism in Java. In Schaefer, J., ed.: High Performance Computing Systems and Applications. Kluwer Academic Publishers, Boston, Dordrecht, London (1998) 85–99
12. Blount, B., Chatterjee, S., Philippsen, M.: Irregular parallel algorithms in JAVA. In: Parallel and Distributed Processing, 6th International Workshop on Solving Irregularly Structured Problems in Parallel. Number 1586 in Lecture Notes in Computer Science, Puerto Rico, Springer Verlag (1999) 1026–1035
13. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Second edn. Addison-Wesley (1997)
14. Sun Microsystems: Java 2 Platform, Standard Edition, v 1.2.2 API Specification. (1999) http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html.
15. Oliver, J., Ayguade, E., Navarro, N.: Towards an efficient exploitation of loop-level parallelism in Java. In: Java Grande. (2000) 9–15
16. van Reeuwijk, C., van Gemund, A., Sips, H.: Spar: A programming language for semi-automatic compilation of parallel programs. Concurrency: Practice and Experience **9** (1997) 1193–1205

17. Meyer, B.: Eiffel: The Language. Prentice Hall International, Hemel Hempstead, UK (1992)

Design Patterns for Library Optimization*

Douglas Gregor, Sibylle Schupp, and David Musser

Computer Science Department
Rensselaer Polytechnic Institute
{gregod,schupp,musser}@cs.rpi.edu

Abstract. We apply the notion of design patterns to optimizations performed by designers of software libraries, focusing especially on object-oriented numerical libraries. We formalize three design patterns that we have abstracted from many existing libraries and discuss the role of these formalizations as a tool for guiding compiler optimizers. These optimizer operate at a very high level that would otherwise be left unoptimized by traditional optimizers. Finally, we discuss the implementation of a design pattern-based compiler optimizer for C++ abstract data types.

1 Introduction

Design patterns have been widely accepted as an invaluable tool for the design of software systems. They represent abstract notions of the behavior of code without collapsing under the weight of implementation details, and therefore serve as an efficient method of communicating design. Design patterns are not synthesized but instead are abstracted from commonalities in design found amongst many successful software systems. As abstractions, these design patterns must be customized for any specific task at hand, but any instance retains the properties of the design pattern(s) applied.

Design patterns need not be limited to high-level design. Techniques employed by designers of high-performance software libraries to enable code optimizations also constitute design patterns. Especially in object-oriented libraries, there are standard ways for example to minimize the number of temporaries, to manipulate the evaluation of an expression, or to choose among functionally equivalent expressions. It is essentially because of these optimization patterns that libraries in higher level programming languages such as C++ or Java have become competitive with those written in C or Fortran. Often, however, the price for using these patterns is code clarity.

In an object-oriented numeric library, for example, it is often possible to directly express mathematical formulae by using operators on user-defined types, but these operator expressions are known to cause a large number of extraneous temporary values to be computed and stored. While these temporaries may be inexpensive for fundamental integer or floating-point types, or even small

* This work was supported in part by the National Science Foundation (NSF) NGS Grant 0131354.

user-defined types, such as complex numbers, temporaries for large user-defined types, such as arbitrary-length integers, arbitrary-precision floating point numbers, or matrices, can become very costly. Programmers have reacted to these extra costs by reverting from the more natural operator-centric representation of mathematical expressions to the use of procedure calls that require fewer temporaries and result in better overall performance.

Design patterns for optimization provide a new perspective on the ways in which library authors design code for maximal performance. These optimization patterns offer the same benefits as traditional design patterns in that they succinctly communicate design, but have additional value in that they can be directly transformed into optimization opportunities for compilers. They are based on the observation that the transformation of, e.g., an operator-centric expression to an equivalent procedural form is a largely mechanical task for the programmer, which, however, cannot be automated as long as the programmer cannot communicate to the compiler the kind of transformation it should perform. What is needed for automation is an optimization scheme a programmer can refer to and a categorization of related optimizations, including the semantic conditions under which they can be applied.

Optimization patterns help make the process of specifying such transformations manageable by defining an abstract form that these transformations may be derived from. Assuming a compiler supports a particular optimization pattern, a user (i.e., library designer) can refer to this pattern and identify the characteristics that make a given transformation an instance of this design pattern. Conversely, an optimization that is given in the form of an optimization pattern has been proved to be applicable across several libraries, and thus has established itself as an optimization methodology. It is therefore worthwhile to develop compiler optimizers based on design patterns. Our Simplicissimus project [15] has already produced one such compiler optimizer that can handle optimization patterns; we hope that other open compilation environments will follow.

We have surveyed several C++ object-oriented numerics libraries and abstracted design patterns that are common amongst these libraries. In this paper we introduce three patterns that are important, but not restricted to numerical applications: the Replacement pattern, the Assignment Replacement pattern, and the Temporary Removal pattern. As it turns out, the Assignment Replacement pattern can be understood as a direct refinement of the Replacement pattern, while the Temporary Removal is a subpattern of the Replacement pattern that adapts instances of the Assignment Replacement pattern.

We begin the presentation with examples of optimizing designs gathered from C++ object-oriented numerics libraries in Section 2 that motivate the abstraction that underlies each pattern. In sections 3 and 4 we formalize, discuss, and illustrate the Replacement pattern and the two subpatterns Assignment Replacement and Temporary Removal. Section 5, finally, summarizes the implementation of optimization patterns within the Simplicissimus framework and briefly show its integration into the GNU C++ compiler. The emphasis of the paper, however, is

on the concept of a design pattern for optimization, and the main purpose of the paper is to initiate the identification and refinement of these patterns.

2 Optimization Methods used by Library Designers

We surveyed several object-oriented numerics libraries, including LiDIA [16], the Matrix Template Library (MTL) [14, 13], the Number Theory Library (NTL) [12], and the Basic Linear Algebra Subprograms (BLAS) [9], searching for design patterns commonly used to facilitate optimizations that could be leveraged by a compiler optimizer instead of relying on the library user. The most common technique is the use of procedures or functions in lieu of operator expressions. These functions can be placed into roughly three categories: shorthand functions, operations that write their result directly to a target, and functions that combine several operations into one call. Each of these categories will be further described with examples from the aforementioned libraries.

Throughout this paper, by *semantic equivalence* of two expressions we mean equivalence of the observable behavior of the expressions. Expressions e_1 and e_2 have the same observable behavior if replacing an instance of one with the corresponding instance of the other will not change a program barring exceptional conditions (e.g., memory allocation failure). We denote this relation by $e_1 \equiv e_2$.

In addition to standard mathematical notation, we use the infix copy assignment operator ':=' that replaces the value of the left-hand operand with the result of computing the right-hand operand. The result of this operation is the left-hand operand.

2.1 Shorthand Functions

Shorthand functions often encapsulate operations that are expressible by common operations but may be computed more efficiently within a single function. Such operations include complex conjugation, inverses, and taking the square of a value. Figure 1 illustrates examples of shorthand functions in NTL and LiDIA.

Library	Operation	Semantics
NTL	Inverse(x)	$1/x$
LiDIA	x.AssignZero()	$x := 0$
LiDIA	x.EqualsOne()	$x = 1$

Fig. 1. Shorthand operations

2.2 Targeted Operations

The return value of an operation is often the cause of unwanted temporaries. Even in simple assignments, such as $y := a \times x$, a temporary is generated by the

multiplication $a \times x$ and must be copied into y. As a reaction to this, library authors create procedures that store the result directly into one of its operands. Figure 2 illustrates some examples of this pattern.

Library	Operation	Semantics
LiDIA	`add(x, y, z)`	$x := y + z$
LiDIA	`multiply(x, y, z)`	$x := y \times z$
NTL	`Sub(x, y, z)`	$x := y - z$
NTL	`Inverse(x, y)`	$x := 1/y$
MTL	`transpose(A, B)`	$B := A^T$

Fig. 2. Targeted operations

2.3 Composite Operations

Certain sets of operations are often used in conjunction. Library authors have used this as an opportunity to introduce new functions that perform all operations in one step without the creation of temporaries and with efficiency that would otherwise not be achieved using separate functions. The most obvious implementation of this technique is in the BLAS libraries, where operator expressions are not included but instead complicated general-purpose routines are supplied. Some examples of composite functions are listed in Figure 3.

Library	Operation	Semantics
MTL	`mult(A, x, y, z)`	$z := A \times x + y$
MTL	`mult(A, B, C)`	$C := A \times B + C$
BLAS	`AXPY(a, x, y)`	$y := a \times x + y$
BLAS	`GEMM(a, A, B, b, C)`	$C := a \times A' \times B' + b \times C'$, where $x' = x, x^T$, or x^H

Fig. 3. Composite operations. The BLAS library's GEMM subroutine has been simplified from its original thirteen arguments for brevity.

3 The Replacement Pattern

Shorthand, targeted, and composite operations often have semantics that are expressed via mathematical formulas. In the majority of object-oriented numerics libraries, these mathematical formulas are also directly expressible, but come at a cost in efficiency. The programmer is expected to transform the mathematical formulas into a set of function or procedure calls to evaluate them. Informally speaking, the Replacement pattern is a natural abstraction of this expression transformation for optimization and can be likened directly to a rewriting system

where the left-hand side of a rewrite rule denotes the mathematical expression and the right-hand side denotes the equivalent, more efficient, procedure call. In the rest of this section we formalize the Replacement pattern in terms of sets of rewrite rules.

3.1 Definitions and Notation

Expressions are finite tree structures built from a given finite set F of function symbols and a denumerably infinite set V of variable symbols; the set of all such expressions is denoted $T(F,V)$. An equation is a pair of such expressions, say (t_1,u_1), usually written $t_1 = u_1$, and the equality rules of inference are captured in the notion of *rewriting* a subexpression of an expression using an equation as a *rewrite rule*. Specifically, a pair of expressions (l,r) is a rewrite rule if l is not just a variable and the variables that appear in r also appear in l. We usually write the rule as $l \to r$, and l is called the left-hand side and r the right-hand side of the rule. Note that in some cases an equation $t = u$ could be used as a rewrite rule as either $t \to u$ or $u \to t$.

A *substitution* is a mapping σ from expressions to expressions that is determined entirely by its value on a finite number of variables; a substitution is denoted by an expression of the form $\{t_1/v_1,\ldots,t_k/v_k\}$, read "substitute t_1 for v_1, ..., t_k for v_k." The $k \geq 0$ variable symbols $v_1,\ldots,v_k$ must be distinct, and the case $k = 0$ is the identity substitution ι such that $\iota(t) = t$ for all expressions t. Following convention we write an application of a substitution as $t\sigma$ rather than $\sigma(t)$.

To define rewriting precisely we also need some notion of position of an occurrence of a subexpression s within an expression t. One way to do this to introduce an extra variable symbol $\Box$ and the concept of a *box expression*: an expression in $T(F,V \cup \{\Box\})$ with a single occurrence of $\Box$. Then an ordinary expression t in $T(F,V)$ can be described as some box expression t_1 with a subexpression s replacing the box, which we make precise as an application of a substitution: $t = t_1\{s/\Box\}$.

For a given rewrite rule $l \to r$, a relation on pairs of expressions, *t rewrites to u*, can be defined as: for some subexpression s of t and box expression t_1 such that $t = t_1\{s/\Box\}$, there is a substitution σ such that $s = l\sigma$ and $u = t_1\{r\sigma/\Box\}$. We write this as $t \to u$ using $l \to r$, overloading the use of the symbol $\to$. For a given set of rewrite rules $\mathcal{R}$, we say $t \to u$ using $\mathcal{R}$ if $t \to u$ for some rule $l \to r$ in $\mathcal{R}$.

These definitions can be extended to *conditional rewriting*: a conditional rewrite rule is a triple of expressions (l,r,p) where (l,r) is a rewrite rule and p is a predicate expression whose variables also appear in l. We usually write the rule as $l \to r$ (if p). For a set of such conditional rules $\mathcal{R}$ the rewriting relation $t \to u$ using $\mathcal{R}$ is defined by $t \to u$ if there is a rule $l \to r$ (if p) in $\mathcal{R}$ such that $t \to u$ using $l \to r$ and $p\sigma$ is true, where σ is the same substitution used in the rewrite.

The nature of the condition on a rewrite rule depends partly on the programming language used and its type system, partly on the program transformation

in which the expression e takes place. Conditions include conceptual or type requirements as well as the specification of computational behavior, e.g., freedom from side-effects of a functional expression, or anti-aliasing of pairs of variables. We want to emphasize, however, that especially in the examples listed the validity of a condition cannot (efficiently) be deduced in an automated way. What can be automatically checked, however, are assertions of properties, including the logical implications of these assertions. We therefore assume that the pattern designer asserts certain properties of variables and other subexpressions, and that a condition is then checked against these declarations. Likewise is it the pattern designer, and not a program, that claims the semantic equivalence of two expressions.

3.2 The Replacement Pattern

We assume there is a *cost* function available from expressions to reals (or any totally ordered domain) so that costs of expressions can be compared. We also recall the relation of semantic equality, $\equiv$, as introduced in Section 1.

Definition. Let L and R be expressions and P be a predicate expression. A *Replacement pattern* is a triple

$$(L, R, P)$$

such that

1. $L \equiv R$ whenever P holds
2. $\text{cost}(R) \leq \text{cost}(L)$

Operationally speaking, a Replacement pattern can be implemented in a framework of conditional rewrite rules. Some patterns can be implemented as a single conditional rewrite rule, $L \rightarrow R$ (if P). This is the case, for example, with the shorthand operation Inverse in Figure 1, with the rewrite rule

$$1/x \rightarrow \text{Inverse}(x)$$

where there is no condition required. Similarly, the Replacement patterns for the other two shorthand operations in Figure 1 can each be implemented with a singe rule. More generally, the implementation of a Replacement pattern can require several rules if there are expressions that are semantically equivalent to L that are not instances of L in the strict syntactic sense of matching defined by the rewrite system. Consider, for example, the Replacement pattern instance that targets the BLAS AXPY routine in Figure 3, which is commonly used for manipulation of vectors. Formally, this instance is

$$(y := a \times x + y,\ \texttt{AXPY}(a, x, y), P(a, x, y))$$

(To simplify the discussion in this section we do not spell out the constraints represented by the predicate P; details of such constraints in several examples

are however discussed in Section 5.) We can use this triple first of all to form the rewrite rule

$$(y := a \times x + y) \rightarrow \texttt{AXPY}(a, x, y) \quad (\text{if } P(a, x, y)),$$

but if we want the same optimization in the case $a = 1$ we also need the rule

$$(y := x + y) \rightarrow \texttt{AXPY}(1, x, y) \quad (\text{if } P(1, x, y)),$$

since $y := x + y$ doesn't syntactically match $y := a \times x + y$ (because it lacks an occurrence of the multiplication operator, $\times$). Similarly, to reflect the role of commutativity of $+$ in semantic equivalence of expressions, we need two more rules

$$(y := y + a \times x) \rightarrow \texttt{AXPY}(a, x, y) \quad (\text{if } P(a, x, y)),$$
$$(y := y + x) \rightarrow \texttt{AXPY}(1, x, y) \quad (\text{if } P(1, x, y)).$$

(We could get by without such additional rules if we were implementing in terms of a more powerful form of rewriting, such as so-called associative-commutative rewriting.)

Thus, in general, to implement the Replacement pattern (L, R, P) we require a set of n rewrite rules $l_i \rightarrow r_i$ (if p_i) such that $l_i \equiv r_i \equiv L\sigma_i$ and $p_i \equiv P\sigma_i$ for some substitution σ_i, for $i = 1, \ldots, n$.

Given that pattern designers are responsible for determining the semantic equality of left- and right-hand side as well as for identifying the constraints that hold for instances of L, the rewrite framework is left with three tasks. First, it performs the syntactic match between an actual subexpression s and a left-hand side, l, of one of the rewrite rules $l \rightarrow r$ (if p), obtaining a substitution σ such that $l\sigma = s$. Second, it checks the constraints $p\sigma$ by inferring whether or not declaratively asserted properties of the actual subexpression s preserve the constraints. If the constraints are satisfied, it applies the rewrite rule and replaces s with the corresponding instance $r\sigma$.

Note that for a set of rewrite rules and a given actual expression the selection of an appropriate rewrite rule is not necessarily unique: the actual expression can match, and satisfy the conditions, of more than one left-hand side. The cost function associated with each rule can then be used to compute the locally optimal selection.

All examples we have seen so far can be considered as instances of the Replacement pattern. Some share additional characteristics, however, and furthermore refer to expression schemes that occur sufficiently frequently to establish patterns on their own, or, more precisely: *subpatterns* of the Replacement pattern. A subpattern inherits all properties of its superpatterns but may add properties, both to the left-hand side of its superpattern and to its right-hand side. Any optimization for a given pattern is valid for any subpatterns of that pattern, and the subpattern relationship is necessarily transitive. The next section presents two examples of subpatterns.

4 Assignment Replacement and Temporary Removal

As the survey in Section 2 has shown, a great deal of emphasis within numerical computing is placed on the removal of temporaries. Therefore, many instances of the Replacement pattern within numerics libraries are designed specifically to remove extraneous temporaries. In this section we first introduce the Assignment Replacement pattern, an abstraction from the targeted operation discussed earlier, then the Temporary Removal adaptor that further eliminates temporaries by adding appropriate expressions. While the Assignment Replacement pattern is a syntactic refinement of the Replacement Pattern, the Temporary Removal adaptor applies to Replacement patterns and generates new instances of Assignment Replacement patterns (which are then eligible for optimization with existing pattern instances).

4.1 The Assignment Replacement Pattern

As motivation we again consider the Replacement pattern for the BLAS routine, $(y := ax + y, \texttt{AXPY}(a, x, y), P)$. We consider here just the first of the four rewrite rules that implement this pattern as discussed in the previous section.

$$(y := a \times x + y) \rightarrow \texttt{AXPY}(a, x, y) \quad (\text{if } P).$$

If we consider the naive computation of the expression $y := a \times x + y$, three loops are required for evaluation: one for the scalar multiplication, one for the vector addition, and one for the vector copy. For each of the two temporaries created by this expression, memory for the vector's storage must be allocated and later freed by the destruction of the temporary. On the other hand, the procedure call $\texttt{AXPY}(a, x, y)$ requires no temporaries and a single loop. Since the discussion of targeted expressions in Section 2.2 has shown that copy assignments are a frequent source of temporaries (see Figure 2) the introduction of a separate optimization pattern for copy assignments seems to be appropriate.

Definition. An *Assignment Replacement* pattern is a Replacement pattern (L, R, P) such that the root of L is a binary function (operator) that represents an assignment to its left operand.

As with the Replacement pattern, instances of the Assignment Replacement pattern may vary greatly in generality and scope. The LiDIA routine add may only be useful for the expression listed in Figure 2, whereas the BLAS routine GEMM has many possible instances, as is illustrated in the form of rewrite rules in Figure 4.

What, however, happens if an actual expression does not quite match, even semantically, the left-hand side of an Assignment Replacement pattern?

$$\begin{array}{|rl|}\hline (C := a \times A \times B + b \times C) & \rightarrow \texttt{GEMM}(a, A, B, b, C) \\ (C := A \times B + b \times C) & \rightarrow \texttt{GEMM}(1, A, B, b, C) \\ (C := a \times A \times B) & \rightarrow \texttt{GEMM}(a, A, B, 0, C) \\ (C := C + a \times A \times B) & \rightarrow \texttt{GEMM}(a, A, B, 1, C) \\ \hline \end{array}$$

Fig. 4. Rewrite system for optimizing to the GEMM function

4.2 The Temporary Removal Adaptor

Consider an expression $z := a \times x + y$ that is similar to the semantic specification of AXPY, but is not semantically equivalent. In this case, two temporaries will be generated. It is possible, however, to remove one of these temporaries by executing $z := y$ followed by the procedure call $\texttt{AXPY}(a, x, z)$. Similarly, the expression $a \times x + y$ may be optimized into a call to AXPY depending on the nature of y. If y is a temporary value, overwriting it with another temporary value is reasonable assuming that y is not reused. In fact, the semantics of most programming languages does not support the direct reuse of temporaries, making this a reasonable assumption. An expression such as $a \times x + b \times y$ can therefore be optimized into $t := b \times y$ followed by a call to $\texttt{AXPY}(a, x, t)$. Generalizing the two examples, we introduce the Temporary Removal adaptor.

Definition. Let (L, R, P) be an Assignment Replacement pattern where L is of the form $y := e$ for some variable y and expression e. We further assume that $e = e_1\{y/\Box\}$ and $R = R_1\{y/\Box\}$ for some box expressions e_1 and R_1. From this pattern the *Temporary Removal adaptor* produces the following new Replacement patterns:

$$\begin{array}{ll} (z := e, & (z := y,\ R_1\{z/\Box\}), \quad P), \\ (e, & (\texttt{var}\ t = y,\ R_1\{t/\Box\}), P). \end{array}$$

where $\texttt{var}\ t = y$ denotes the declaration of a temporary variable t (local to the expression sequence) and its initialization to the value of y.

Applied to the just discussed AXPY Assignment Replacement, for example, the Temporary Removal adaptor generates the following two Replacement patterns:

$$\begin{array}{ll} (z := a \times x + y, & (z := y,\ \texttt{AXPY}(a, x, z)), \quad P), \\ (a \times x + y, & (\texttt{var}\ t = y,\ \texttt{AXPY}(a, x, t)), P). \end{array}$$

In the same way the Assignment Replacement used in the MTL library (see Figure 3)

$$(C := A \times B + C, \texttt{mult}(A, B, C), P)$$

generates the two patterns

$$\begin{array}{ll} (D := A \times B + C, & (D := C,\ \texttt{mult}(A, B, D)), \quad P), \\ (A \times B + C, & (\texttt{var}\ t = C,\ \texttt{mult}(A, B, t)), P), \end{array}$$

and the GEMM Assignment Replacement (see Figure 4)

$$(C := C + a \times A \times B, \texttt{GEMM}(a, A, B, 1, C), P)$$

the two patterns

$$\begin{array}{ll} (D := C + a \times A \times B, & (D := C,\ \texttt{GEMM}(a, A, B, 1, D)), \quad P), \\ (C + a \times A \times B, & (\texttt{var}\ t = C,\ \texttt{GEMM}(a, A, B, 1, t)), P). \end{array}$$

5 Implementation

The implementation of an optimizer for the Replacement pattern and its subpatterns essentially requires the implementation of an expression rewrite system with rewrite rules supplied by the user. An immediate requirement of such a system is that the implementation of expression matching must be generic enough to support any form of expression, including user-defined operators (in the form of overloaded operators or function calls). Additionally, the user must be able to examine an expression to determine the semantics of the expression and its subexpressions to ensure correctness when applying a rewrite rule. Finally, the user must be able to construct new expressions to complete the rewriting step.

5.1 Internal Representation

Simplicissimus' internal representation consists entirely of C++ expression templates, a set of classes representing unary, binary, ternary, and other operations that are parameterized by the operators and operands, in a form similar to functional prefix form. Expression templates were discovered as an optimization technique for numerical computing [19] but have also been used for delayed evaluation and functional composition [8,4]. Simplicissimus' expression templates differ from most in that they have no run-time components: distinct variables and literal values are modeled as types, so that C++ expressions can be fully expressed as C++ types and manipulated at compile time.

Compile-time manipulations of expressions using expression templates have several advantages. They do not exist at run-time, so they incur no run-time overhead. They are also natural to work with within C++, using well-known *template metaprogramming* techniques [17] and especially partial specialization for rule matching, which is further described in Section 5.2. Finally, they are platform- and compiler-independent because they represent C++ with C++; this will be further discussed in Section 5.5.

The form of an expression template is similar to that of function prefix form. An expression $x + y * z$ can be expressed in prefix form as $(+\ x\ (*\ y\ z))$ and, similarly, as the expression template
`Expr<BinaryExpr<Add, X, Expr<BinaryExpr<Mul, Y, Z> > > >`. Here we use the type names `Add` and `Mul` to represent addition and multiplication, respectively. Each operator or function will have a unique type (generally an empty class) that represents it in an expression template. Expressions are wrapped

in class templates that contain the operator name and its operand(s), and are named based on the arity of the operation (`UnaryExpr`, `BinaryExpr`, etc.). The `Expr` class is a wrapper around each expression template that makes all expression templates easily distinguishable from other types.

The leaves of an expression tree—literal values and variables—are each expressed using unique types. The class template `Variable` is parameterized by the type of the variable (e.g., `int`) and by an integer identification number that is unique to that variable. In our example above, `X` may be `Expr<Variable<int,0>>` whereas `Y` could be `Expr<Variable<int,1>>`. Similarly, a class template `Literal` contains literal values, where a literal can be any C++ literal, but the notion has been extended slightly to include user-defined literals for abstract data types.

5.2 Matching Expressions

Expression templates naturally lend themselves to pattern-matching via partial specialization. Partial specialization allows multiple definitions of class templates where each definition specifies the partial type structure of types it will be instantiated with. Expression templates use type structure to express expression evaluation, thus partial specialization can trivially be used to specify and match expressions. Figure 5 illustrates the *primary* template and one specialization of the class template `AXPYMatch`. The template can match any expression template via the primary template (the `valid` member will be `false`) but it can also match an expression $a*x+y$ where $+$ is represented by the type `VectorAdd` and $*$ is represented by the type `VectorScale`, in which case `valid` will be `true` to signify a match.

```
template<typename ExprT> struct AXPYMatch
  { static const bool valid = false; };

template<typename A, typename X, typename Y>
struct AXPYMatch< Expr< BinaryExpr<
                          VectorAdd,
                          Expr<BinaryExpr<VectorScale, A, X> >, Y> > >
  { static const bool valid = true; };
```

Fig. 5. Using partial specialization to perform a syntactic match

5.3 Semantic Constraints

Semantic constraints determine whether or not a particular expression that syntactically matches the left-hand side of a rewrite rule will be semantically equivalent if the expression is rewritten. The check for semantic equivalence relies primarily on traits that describe the computational behavior of expressions, including which operands are modified, whether an operation has side effects

beyond what is reflected in the operands and return value, and whether the operation is applicative (i.e., predictable given a set of operands and regardless of program state).

We will extend the expression matching class template `AXPYMatch` described in Section 5.2 to validate the semantic constraints of the `AXPY` subroutine in addition to matching the structure. This dual purpose is reasonable because semantic constraints are generally expressed as predicates based on the variables bound when matching the expression.

```
template<typename Expr1, typename Expr2> struct SameVariable
  { static const bool value = false; };

template<typename T, int ID>
struct SameVariable<Expr<Variable<T, ID> >, Expr<Variable<T, ID> > >
  { static const bool value = true; };

template<typename ExprT> struct AXPYMatch
  { static const bool valid = false; };

template<typename A, typename X, typename Y>
struct AXPYMatch< Expr< BinaryExpr<
                          VectorAdd,
                          Expr<BinaryExpr<VectorScale, A, X> >, Y> > >
{
  static const bool valid = !SameVariable<X, Y>::value
    && !X::has_side_effects && !Y::has_side_effects;
};
```

Fig. 6. Expressing the semantic requirements of the `AXPY` transformation using traits

Figure 6 illustrates the validation of the semantic constraints on `AXPY`. Altogether three constraints on its parameters x and y, logically connected to the member `valid`, have to be met. For one, neither the evaluation of x nor the evaluation of y may have side effects, because the order of evaluation may change when rewriting an expression as a function call. The compile-time value of the member `has_side_effects` of any expression template is recursively determined using expression and user-defined operation traits. Additionally, x and y may not be the same variable. The `SameVariable` class template of Figure 6 determines if the given expression templates are the same variable in the simplest case. A completely developed version of `SameVariable` is more extensive in that it takes into account user-defined operators that return references to one of their arguments, such as the C++ assignment operator.

5.4 Temporary Removal Adaptor

The optimizations described for temporary removal in Section 4.2 are implemented in Simplicissimus as a class template `InPlaceOperationSimp`. The class template `InPlaceOperationSimp` is instantiated with a class template `T` that implements the functionality specific to an particular instance of the Temporary Removal adaptor. The functionality required by `T` is implemented by three members:

- `valid`: a boolean value that is true iff the syntactic and semantic constraints on the pattern are met;
- `result`: the type of the variable that is the target of the assignment in the underlying Assignment Replacement;
- `rewrite_with_target`: a class template that performs a rewrite of the given expression to the procedural form using the given target expression.

We complete the optimization of the `AXPY` function in Figure 7 with our final implementation of the class template `AXPYMatch`. This class is to be directly used with the `InPlaceOperationSimp` template to generate the rewrite rule class `AXPYSimp` that performs three temporary-removing optimizations within the Simplicissimus system: the `AXPY` Assignment Replacement along with the two optimizations generated by the `AXPYMatch` adaptor.

$$
\begin{aligned}
(y := a \times x + y) &\rightarrow \texttt{AXPY}(a, x, y),\\
(z := a \times x + y) &\rightarrow (z := y,\ \texttt{AXPY}(a, x, z)),\\
a \times x + y &\rightarrow (\texttt{var}\ t = y,\ \texttt{AXPY}(a, x, t)).
\end{aligned}
$$

```
template<typename ExprT> struct AXPYMatch
  { static const bool valid = false; };

template<typename A, typename X, typename Y>
struct AXPYMatch< Expr< BinaryExpr<
                          VectorAdd,
                          Expr<BinaryExpr<VectorScale, A, X> >, Y> > >
{
  static const bool valid = !SameVariable<X, Y>::value
    && !X::has_side_effects && !Y::has_side_effects;
  typedef Y target;

  template<typename Z> struct rewrite_with_target
    { typedef Expr<TernaryExpr<AXPY, A, X, Z> > result; };
};

struct AXPYSimp : public InPlaceOperationSimp<AXPYMatch> {};
```

Fig. 7. Optimizations for the BLAS `AXPY` function based on the Temporary Removal adaptor

Partial specialization is again used to match `AXPY`'s semantic constraints. The `valid` member is true whenever the expression is matched, and the target of the `AXPY` function is identified as Y by the `target` member type. The actual rewriting into the more efficient form using `AXPY` is performed by the class template `rewrite_with_target`, which trivially builds an expression template using the ternary operation `AXPY`.

5.5 Integration in the GNU C++ Compiler

Simplicissimus is a stand-alone optimizer written in the C++ template sublanguage, and is therefore naturally compiler-neutral. Such a design allows optimizations based on Simplicissimus, such as the implementation of the Replacement and its subpatterns, to be portable as well.

Integration of the Simplicissimus optimizer with a new compiler requires a transformation from the compiler's internal representation to Simplicissimus's expression templates for optimization, and then the reverse transformation to utilize the results of the optimization. Within the GNU C++ compiler, approximately 2000 lines of C code were required to perform these transformations.

6 Related Work

Design patterns [5] have been gaining wide acceptance as a tool for the construction and documentation of software systems, but their use does not generally extend beyond that of documentation or guidelines for programmers. The FRED [7] development environment, which extends this limited view of patterns to instead aid the programmer in the specialization of patterns for a particular purpose, thus shares our view that a design pattern is more than documentation or guideline. On the other hand, the goals are radically different from our own.

Tools for applying domain-specific transformations to optimize code, such as TAMPR [2] and Draco [10], enable authors of domain-specific languages to introduce optimizations based on the semantics of a particular domain. However, these general systems do not provide a conceptual framework for generating transformations that are common across multiple domains and multiple languages, that is, they do not take a pattern-based approach that describes optimizations as specializations of well-known, language- and domain-neutral optimization patterns. Constructing new, domain-specific languages that have similar optimization opportunities to other domains therefore causes a large amount of repetition.

Tools that allow library-specific optimizations within general purpose languages, such as the Broadway [6] open compilation system and the CodeBoost [1] source-to-source transformation system, enable users (library designers) to introduce additional semantic information and optimization opportunities for ordinary user code. Like domain-specific transformation, however, these systems give users little direction regarding optimizations that span multiple software libraries. Applying design patterns for optimization to any of these transformation systems would yield the same benefits as in our own Simplicissimus optimizer.

Work in the construction of *active libraries* [20], such as Blitz++ [18] and POOMA [11], has significantly narrowed the gap between library and compiler. Such libraries take an active role in the compilation process, tuning the generated code to specific tasks or specific architectures. Design patterns for optimization—or, specifically, implementations supporting them—can serve as a powerful tool for use by active libraries enabling optimizations that are impossible without such support. The Sophus C++ library [3] integrates with the aforementioned CodeBoost transformation system to apply domain-specific transformations to C++ code that uses the Sophus library. The transformations there are similar to those of the Temporary Removal adaptor.

7 Conclusion

We have surveyed the design of several object-oriented numerics libraries with a strong focus on optimization techniques employed. From these designs we abstracted the common structure and semantics to form the Replacement pattern and two important subpatterns, the Assignment Replacement and the Temporary Removal adaptor. Additional patterns, such as the delayed element-wise transformation used by expression templates in libraries such as Blitz++ [18] and POOMA [11], are also known to exist but have not yet been studied.

Unlike many design patterns, the Replacement pattern and its subpatterns present optimization opportunities at a very high level of abstraction. Once instances of these patterns are identified, a compiler optimizer can attempt to generate better code based on strong, user-supplied assumptions on the semantic behavior of abstract data types. We see these patterns as tools for advanced users and library authors to direct the optimization of high-level constructs that otherwise would be left unoptimized.

The Simplicissimus compiler optimizer implements the three patterns discussed in a compiler-independent manner. By using the strengths of the C++ language, Simplicissimus provides users with the ability to specify optimizations for abstract data types without requiring recompilation or additional extension of the compiler.

References

1. O. Bagge, M. Haveraaen, and E. Visser. CodeBoost: A framework for the transformation of C++ programs. Technical report, Universiteit Utrecht, The Netherlands, October 2000.
2. J. Boyle, T. Harmer, and V. Winter. The TAMPR program transformation system: Design and applications. In E. Arge, A. Bruaset, and H. Langtangen, editors, *Modern Software Tools for Scientific Computing*. Birkhauser, 1997.
3. T. Dinesh, M. Haveraaen, and J. Heering. An algebraic programming style for numerical software and its optimisation. Technical Report SEN-R9844, CWI, December 1998.
4. FACT! - Multiparadigm programming with C++. `http://www.kfa-juelich.de/zam/FACT/start/index.html`, 2001.

5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.
6. S. Z. Guyer and C. Li. An annotation language for optimizing software libraries. In T. Ball, editor, *2nd Conference on Domain-Specific Languages*. Usenix, 1999.
7. M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa. Generating application development environments for Java frameworks. In J. Bosch, editor, *Generative and Component-Based Software Engineering*, volume 2186 of *LNCS*, pages 163–176. Springer, September 2001.
8. J. Järvi and G. Powell. The Lambda library: Lambda abstraction in C++. Technical Report 378, Turku Centre for Computer Science, November 2000.
9. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, Sept. 1979.
10. J. M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–574, September 1984.
11. POOMA. `http://www.acl.lanl.gov/pooma/`, 2001.
12. V. Shoup. NTL: A library for Number Theory, 2001. `http://www.shoup.net/ntl/`.
13. J. G. Siek. A modern framework for portable high performance numerical linear algebra. Master's thesis, Notre Dame, 1999.
14. J. G. Siek and A. Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.
15. Simplicissimus. `http://www.cs.rpi.edu/research/gpg/Simplicissimus`, 2001.
16. The LiDIA Group. Lidia—a C++ library for computational number theory. `http://www.informatik.tu-darmstadt.de/TI/LiDIA/`.
17. T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4), 1995.
18. T. Veldhuizen. Blitz++. `http://www.oonumerics.org/blitz/`, 2001.
19. T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.
20. T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

An Interactive Environment for Supporting the Paradigm Shift From Simulation to Optimization*

Christian H. Bischof, H. Martin Bücker, Bruno Lang, and Arno Rasch

Institute for Scientific Computing
Aachen University of Technology, D–52056 Aachen, Germany

Abstract. Simulation is a powerful tool in science and engineering, and it is nowadays also used for optimizing the design of products and experiments rather than only for reproducing the behavior of physical or other systems. In order to reduce the number of simulation runs, the traditional "trial and error" approach for finding near-to-optimum design parameters should be replaced with efficient numerical optimization algorithms. Done by hand, the coupling of simulation and optimization software is tedious and error-prone. In this note we report on the current version of a software environment that facilitates and speeds up this task by doing much of the required work automatically. Our framework includes support for automatic differentiation, which can provide the derivatives required by many optimizers. We describe the process of integrating the widely used computational fluid dynamics package FLUENT and a MINPACK-1 least squares optimizer into our environment and follow a sample session solving a data assimilation problem.

1 Introduction

Traditionally, simulation software has been used mainly for reproducing—as exactly as possible—the behavior of physical or other systems, thus complementing theory and classical experiment. While this kind of use certainly will remain important, there is increasing demand for embedding the simulation in a larger optimization framework.

One prominent example is *design optimization*, where we seek parameters $\mathbf{x}$ for the system such that some cost function $\mathbf{f}(\mathbf{x})$ is minimized. Replacing experiments with simulation in the optimization process can drastically reduce the number of prototypes to be built before the final product emerges, and thus leads to considerable savings in money and time. Another optimization problem, *data assimilation*, comes from modeling and from the simulation itself. Here we try to adjust the values of certain model or simulation parameters such that the computed values $\mathbf{f}(\mathbf{x})$ best match the data $\mathbf{d}$ obtained from actual experiments.

* This research is partially supported by the Deutsche Forschungsgemeinschaft (DFG) within SFB540 "Model-based experimental analysis of kinetic phenomena in fluid multi-phase reactive systems," Aachen University of Technology, Germany.

Thus, the objective function to optimize is $||\mathbf{f}(\mathbf{x}) - \mathbf{d}||$ in the data assimilation problem, whereas in the design optimization problem it is $||\mathbf{f}(\mathbf{x})||$. In both cases, weighted norms may be used to emphasize selected components.

Up to now, such optimization problems often are "solved" by running the simulation several times with varying parameter sets and selecting the set that gave the best results. While easy to implement from a programming point of view, this procedure is not very efficient with respect to the number of simulation calls, and the selection of "appropriate" parameter sets may require experience. Numerical optimization routines, by contrast, typically make better use of the information and can also be used by non-experts. Unfortunately, optimization software and simulation software often follow different conventions for passing parameters, and therefore it is a tedious and error-prone task to combine these two components. In particular, switching to another optimizer or another simulation package requires rewriting the interfacing software, often from scratch.

This situation was addressed in [4], where we proposed a framework for *automatically* combining large-scale simulation and optimization software via *CORBA* technology. In the present note we report on an extension of the functionality of this environment and demonstrate the use of the software with selected case studies.

The structure of the note is as follows. Sophisticated optimizers make use of derivative information in order to reduce the number of iterations. In Sect. 2 we discuss several methods for computing these derivatives. In particular, the so-called forward mode of automatic differentiation (AD) is reviewed. The structure of our software environment is described in Sect. 3. Section 4 presents some case studies showing the steps that are necessary to integrate new simulation and optimization codes into our framework and to run the optimization, including simple and efficient access to derivatives via AD. This section also includes results from numerical experiments. Our findings are summarized in Sect. 5.

2 Providing Derivative Information

This section addresses the problem of providing derivative information for large-scale simulations, as required by sophisticated optimizers. We will briefly discuss the problems with three well-known methods for computing derivatives, namely analytic, symbolic, and numerical differentiation, and review a less widely known technique called automatic differentiation.

If the function is given by an explicit formula or defined by simple differential or integral equations then often the derivatives of the function can also be described in this way (*analytic differentiation*). Then the mathematical description is turned into code by hand. In the context of simulation, complex effects such as turbulence typically preclude this method.

Due to the sheer size of the simulation packages totaling hundreds of thousands of lines and to the complexity of the codes with their numerous branches, loops, and subroutine calls, tools for *symbolic differentiation*—aimed at automatically producing code for evaluating the derivatives *at arbitrary points* $\mathbf{x}$—

also fail even for highly simplified test cases where the derivatives might be be obtained analytically.

If applicable, analytic and symbolic differentiation yield derivatives that are accurate except for rounding errors. *Numerical differentiation* with *divided differences* (DD), by contrast, always incurs an approximation error, which grows with the step size. Taking the first-order forward divided difference

$$\frac{\partial}{\partial x_j}\mathbf{f}(\mathbf{x}) \approx \frac{\mathbf{f}(x_1,\ldots,x_{j-1},x_j+h,x_{j+1},\ldots,x_n)-\mathbf{f}(\mathbf{x})}{h} \tag{1}$$

as an example, one sees that this problem cannot be solved by using a tiny step size h because then catastrophic cancellation in the numerator of Eqn. (1) reduces the quality of the computed values. As a result, derivatives approximated via DD are often only valid to one half of the available digits, even with an optimal choice for h. The main advantage of the DD approach is that it is independent from the complexity of the function $\mathbf{f}$, which is used only in a black-box fashion to be evaluated at certain points $\mathbf{x}$.

The fourth technique, to be considered in the following, is able to provide highly accurate derivatives even for very complex codes. The term *automatic differentiation* (AD) comprises a set of techniques for automatically augmenting a given computer program with statements for the computation of derivatives. That is, given a computer program C that computes a function

$$\mathbf{f}(\mathbf{x}) = \big(f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x})\big)^T \in \mathbb{R}^m,$$

automatic differentiation generates another program C' that, at any point of interest $\mathbf{x} \in \mathbb{R}^n$, not only evaluates $\mathbf{f}$ but also its Jacobian

$$J_{\mathbf{f}}(\mathbf{x}) := \begin{pmatrix} \frac{\partial}{\partial x_1} f_1(\mathbf{x}) & \ldots & \frac{\partial}{\partial x_n} f_1(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} f_m(\mathbf{x}) & \ldots & \frac{\partial}{\partial x_n} f_m(\mathbf{x}) \end{pmatrix} \in \mathbb{R}^{m\times n}$$

at the same point $\mathbf{x}$.

The AD technology is applicable whenever derivatives of functions given in the form of a high-level programming language, such as Fortran, C, or C++, are required. The reader is referred to the recent book by Griewank [13] and the proceedings of AD workshops [1, 5, 14] for details on this technique. In automatic differentiation the program is treated as a—potentially very long—sequence of elementary statements such as binary addition or multiplication, for which the derivatives are known. Then the chain rule of differential calculus is applied over and over again, combining these step-wise derivatives to yield the derivatives of the whole program. This mechanical process can be automated, and several AD tools are available for transforming a given code C to the new *differentiated* code C' [2, 3, 12, 15]. In this way, AD requires little human effort and produces derivatives that are accurate up to machine precision.

When accumulating the derivatives of elementary operations step by step, the associativity of the chain rule offers several alternatives, all leading to the

same overall derivatives for the whole program, but at different cost with respect to computation and storage. One well-known strategy for applying the chain rule is the so-called forward mode of AD. If one is interested in obtaining derivatives of $\mathbf{f}$ with respect to n scalar variables (called *independent variables* hereafter), a gradient object $\mathbf{u}^\nabla \in \mathbb{R}^n$ is associated to every intermediate scalar variable u involved in the evaluation of the function $\mathbf{f}$. In the sequel, u is called *function part* and its associated $\mathbf{u}^\nabla$ is referred to as *gradient part*. The pair $u_d = [u, \mathbf{u}^\nabla]$ is called a *doublet*. Note that the gradient part of a doublet stores the gradient of the function part with respect to the independent variables.

For every operation of the original code C involving a scalar variable u, there is a corresponding operation on the doublet $u_d = [u, \mathbf{u}^\nabla]$ in the differentiated code C'. For instance, a binary addition statement $u = v + w$ in C is transformed in C' into

$$\begin{aligned} u &= v + w \\ \mathbf{u}^\nabla &= \mathbf{w}^\nabla + \mathbf{v}^\nabla; \end{aligned}$$

that is, the separate addition of function and gradient part. A multiplication statement $u = v \cdot w$ is transformed into

$$\begin{aligned} u &= v \cdot w \\ \mathbf{u}^\nabla &= v\mathbf{w}^\nabla + w\mathbf{v}^\nabla, \end{aligned}$$

where the gradient part is defined in a product rule-like manner. Since any programming language consists of only a small set of operations, the set of corresponding operations on doublets is easily constructed.

Except for very simple cases, the function f cannot be evaluated within a single routine. Instead, evaluating f typically involves a large subtree of the whole program, the "top-level" routine of this tree invoking a multitude of lower-level routines and finally providing the function value. For example the computation of some characteristic number of a stationary flow may involve a nonlinear solver, which in turn calls a preconditioned linear solver, and so on. In such a case automatic differentiation must be applied to the whole subtree, often totaling several hundreds of routines and tens or even hundreds of thousands lines of code, in order to obtain the function's derivatives.

In contrast to the forward mode, the so-called reverse (or backward) mode of AD generates derivatives objects whose length is equal to the number of dependent variables, m, and can be more efficient than the forward mode if $m < n$.

Sophisticated forward mode AD tools are capable of generating code for the computation of $J_f(\mathbf{x}) \cdot S$, where S is a so-called *seed matrix* of appropriate dimension. That is, besides computing the Jacobian matrix explicitly by setting S to the $n \times n$ identity, the seed matrix offers the option to compute any linear column combination of the Jacobian at a cost that is proportional to the number of columns of S. Thus, appropriately initializing S, called *seeding*, is often critical in terms of performance, provided that the full Jacobian J is not needed

explicitly. Reverse mode AD tools typically allow the computation of any linear row combinations $S^T \cdot J_f(\mathbf{x})$ of the Jacobian.

3 The Structure of the Software Environment

To give a better understanding of the structure of our software environment we first review the steps that have to be done in order to solve a typical data assimilation problem. Here, `f` denotes a subroutine evaluating the simulation function, and `opt` is used to refer to some optimization routine. We further assume that `opt` takes as input a user-supplied function `u` which is needed to compute the objective function. For instance, `u` might represent the difference vector $\mathbf{f}(\mathbf{x}) - \mathbf{d}$ or its norm.

- First, a driver for `opt` has to be implemented, which provides an initial guess $\mathbf{x}^0$ and potentially additional control parameters specifying stopping criteria, constraints, etc.
- The function `u` has to be implemented such that `f` is called and then the return values $\mathbf{f}(\mathbf{x})$ are compared with the measurements $\mathbf{d}$, and finally the results are returned. Note that the calling sequence of `u` is usually prescribed by the developers of the optimization software. So the user has to fit his particular optimization problem into a given scheme.
- There is often a similar scheme for the routine `du` computing the derivative of `u`. Therefore a mechanism for evaluating the derivatives of `f` is needed.
- To increase flexibility it is often desired to keep some of the input parameters of `f` at fixed values, i.e., only a subset of the parameters of `f` should be optimized. So there is need for a mechanism to specify fixed parameters $\mathbf{p}$.

The computational scientist often considers various simulation packages and, more importantly, different optimization codes in order to validate the robustness of the numerical solution. Instead of implementing the above requirements several times by hand, we suggest an automated way for easily combining different software packages and experimenting with varying problem configurations.

One step in this direction is the NEOS project [7]. The NEOS environment allows users to solve an optimization problem remotely on an optimization server, which offers a rich variety of optimization algorithms. The user submits an initial guess, possibly some control parameters, and a subroutine for evaluating the objective function to the NEOS server, where the problem is solved with the selected optimizer. This approach is easy to use and highly flexible. However, it is not applicable to our class of problems, where the evaluation of the objective function involves a complete run of a typically very large simulation code. In addition to sheer size, the simulation code might be tuned for a specific architecture, e.g., for a parallel or vector supercomputer, or it is possibly protected by copyright laws and therefore cannot be submitted via the internet. There are many other projects aimed at connecting existing software components in the context of simulation (see, e.g., the contributions in [16]), but most of them do not touch on optimization.

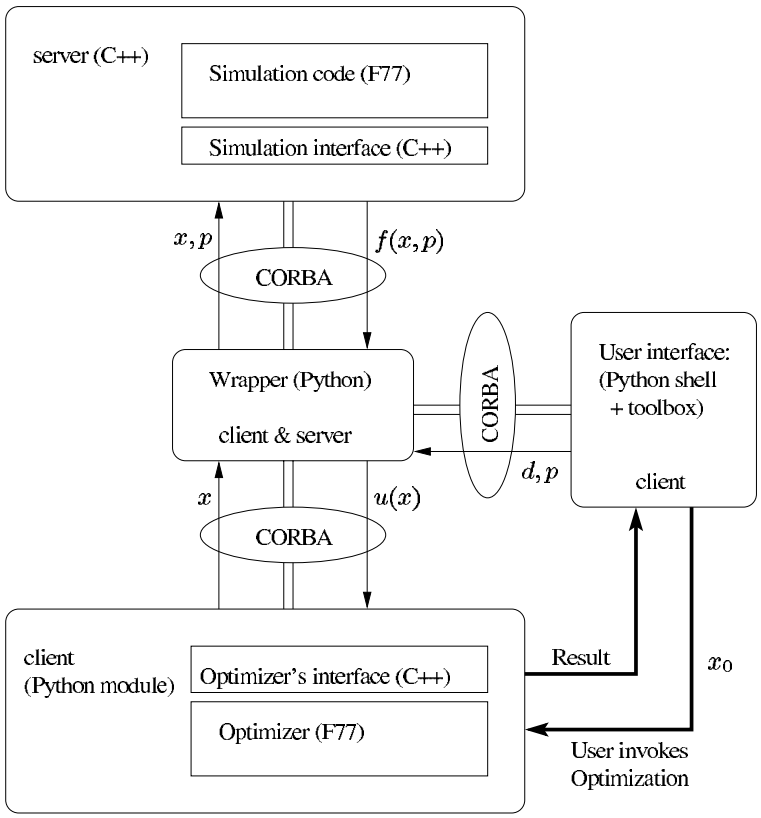

Fig. 1. Overall structure of the CORBA-based environment for rapid prototyping.

In [4] we proposed a software environment especially designed for *automatically* combining optimization routines and large-scale simulation codes. Our approach treats the evaluation of the function `f`, the execution of one iteration of the optimizer `opt`, and the computation of the user-defined function `u` as basic tasks and provides an infrastructure for controlling the interplay of these tasks. To achieve maximum flexibility in supporting different platforms and languages, our environment is based on the *CORBA* technology. The structure of the system is depicted in Fig. 1.

The user specifies the input and output variables of the simulation code as well as the variables needed to optimize the objective function. From this specification, C++ interfaces for the evaluation of the function `f` and its derivatives are generated automatically. They are needed to set up a so-called "simulation server", which is able to drive the evaluation of the simulation function as well as the specified gradient or Jacobian. All requests to the simulation server come from a standard module called "wrapper" hereafter. The wrapper is responsible

for transferring data between the simulation and optimization components and also for the computation of the user-defined function u. Also making use of the specification mentioned above, the wrapper sends a request to the simulation server for evaluating either $\mathbf{f}(\mathbf{x}, \mathbf{p})$ or the derivative of $\mathbf{f}(\mathbf{x}, \mathbf{p})$ w.r.t. $\mathbf{x}$. Here, the variables needed for optimization are denoted by $\mathbf{x}$, whereas $\mathbf{p}$ represents additional fixed parameters of the simulation. The specification of the simulation's input parameters also includes the values for such fixed parameters.

The whole computation is driven by the optimization component which repeatedly calls the user-defined subroutine u (and, possibly, du) for evaluation of the objective function (resp. its derivative). In our framework, these two routines are just stubs that perform no computational work by their own but only call a standardized C++ interface, which in turn makes use of CORBA to send an evaluation request, together with an argument $\mathbf{x}$, to the wrapper. The wrapper then forwards this request to the simulation server, complementing the free simulation parameters $\mathbf{x}$ with the fixed values $\mathbf{p}$.

The complete optimization process is invoked via a user's interface, which also makes the measurements $\mathbf{d}$ available to the wrapper. The user interface is implemented via the *Python* shell and provides some auxiliary functions, e.g., for communication with the wrapper module and for specifying input and output parameters of the simulation. This allows users to experiment with different problem configurations either interactively or through scripts. The actual implementation of our system uses *omniORB3.0* [17] for C++ and *Fnorb* [6] for Python.

4 Case Studies

In this section we describe the integration of particular software components into our system, namely the widely used *FLUENT* computational fluid dynamics (CFD) package [9], and an optimization routine from the *MINPACK-1* library [10]. Then we show the use of the environment by means of a test example taken from the FLUENT tutorial guide [9].

4.1 Integration of the FLUENT Solver

In order to integrate any simulation package into our prototyping environment, we must be able to control the simulation through one single routine, the so-called "top-level routine". This does not mean that the complete simulation code must be contained in one routine but that the input and output values of the simulation are accessible within that routine.

In the case of FLUENT we had to turn off the graphical user interface. The remaining text-based version of FLUENT can be controlled via a so-called "log file" providing a complete specification of the simulation problem. Furthermore, we had to replace the main program by a top-level routine, which takes the relevant input parameters, executes the commands given in the log file, and returns the results.

In our test example based on the example *Flow Through a Filter Cartridge* from the FLUENT tutorial we want to adjust the model parameters $c_{1\varepsilon}, c_{2\varepsilon}, c_\mu, \sigma_k, \sigma_\varepsilon$ of the k-ε turbulence model such that the pressure distribution in the filter best matches some given experimental data. Therefore, the top-level routine `filter` calculating the simulation function has the following structure:

```
      subroutine filter(c1,c2,cmu,eprnd,dprnd,lpressure,pressure)
      integer lpressure
      double precision c1,c2,cmu,eprnd,dprnd
      double precision pressure(lpressure)

c --- open log file: channel 5 (= stdin) is redirected to file.
c --- Problem specification is given in FILTER_LOGFILE
      open(unit=5,file='FILTER_LOGFILE')

c --- original Fluent code

c --- close channel 5 (stdin)
      close(5)
      end
```

Here, the input variables `c1`, `c2`, `cmu`, `eprnd`, and `dprnd` correspond to the turbulence parameters $c_{1\varepsilon}$, $c_{2\varepsilon}$, c_μ, σ_k, and σ_ε. Note that we further changed the original FLUENT code such that it uses the values of these input variables instead of the internal default values for the turbulence parameters.

The next step is to provide a specification of the input and output variables. For simplicity, all variables are assumed to be `double precision` – so the specification just consists of array size information. Furthermore, the top-level routine of the simulation and a set of variables to optimize, which is a subset of the simulation's input variables, must be defined. In our test example we want to enable all input variables for optimization.

The specification is given via the user's interface, and is needed to generate the CORBA/C++ interface connecting the simulation routine to our system. It will be also used later during the optimization process.

The generated CORBA/C++ routine takes a sequence of input variables (possibly vectors) from the wrapper, allocates memory for the output variables, then calls the simulation's top-level function with the input values, and finally returns one or more solution vectors to the wrapper. An excerpt from the CORBA/C++ interface generated for our particular test example is given below:

```
// input:  sequence of input vectors ''x_in''
// output: sequence of solution vectors ''all_solutions''
  double c1,c2,cmu,eprnd,dprnd;
  int ldpressure = 336;
  double *pressure = new double[ldpressure];
  c1 = x_in[0][0];
```

```
  c2 = x_in[1][0];
  cmu = x_in[2][0];
  eprnd = x_in[3][0];
  dprnd = x_in[4][0];
  filter(&c1,&c2,&cmu,&eprnd,&dprnd,&ldpressure,pressure);
// copy pressure to all_solutions[0]
  ...
  delete pressure;
  return all_solutions._retn();
```

If desired, the system additionally creates a control script which can be used by the *ADIFOR* AD tool [2] to transform the simulation source code into new code with additional statements for the computation of the derivatives of the simulation's outputs w.r.t. the input variables selected for optimization.

Based on the particular specification for our test example where the dependent variable `pressure` of the top-level routine `filter` is to be diferentiated w.r.t. five independent variables (`AD_IVARS`), the system generates a control script for ADIFOR containing the following directives:

```
AD_TOP=filter
AD_PMAX=5
AD_IVARS=c1,c2,cmu,eprnd,dprnd
AD_DVARS=pressure
AD_EXCEPTION_FLAVOR=performance
AD_PROG=
```

For a not-too-complicated code fully adhering to the Fortran 77 standard, one would simply insert the name of a "composition file" (which just lists all source files of the simulation) in the field `AD_PROG`, and run ADIFOR. However, as it is usual with huge programs grown over many years, the FLUENT code with its approximately 1.500.000 lines of mostly Fortran 77 required some additional code massaging in order to obtain *standard* Fortran 77, before we could apply ADIFOR to it. A detailed description of this process will be given elsewhere. Note that this preparation of the code has to be done only once, even if many different optimization problems are considered later on.

The calling sequence of the differentiated top-level routine generated by ADIFOR is given below:

```
      subroutine g_filter(g_p_,
     + c1,g_c1,ldg_c1, c2,g_c2,ldg_c2, cmu,g_cmu,ldg_cmu,
     + eprnd,g_eprnd,ldg_eprnd, dprnd,g_dprnd,ldg_dprnd,
     + lpressure,pressure,g_pressure,ldg_pressure)
```

Here, the seed matrix S, introduced in Sect. 2, consists of the input variables `g_c1`, `g_c2`, `g_cmu`, `g_eprnd`, and `g_dprnd` with corresponding leading dimensions `ldg_c1`, `ldg_c2`, `ldg_cmu`, `ldg_eprnd`, and `g_dprnd`. The derivative of the pressure field is returned in `g_pressure`, which is a two dimensional array of size `(ldg_pressure,lpressure)`.

It is easy to see how the calling sequence of this augmented routine is determined by the original top-level routine: All parameters of `filter` reappear in `g_filter`, and each parameter p corresponding to an "independent" or "dependent" variable (as listed in `AD_IVARS` and `AD_DVARS` in the ADIFOR script) is immediately followed by two additional parameters `g_`p and `ldg_`p containing the derivatives of p and the leading dimension of the new array, respectively. Therefore, our system is able to generate the CORBA/C++ interface for this routine, too.

The interface code for the differentiated routine also performs the proper seeding of the independent variables according to the user's specification. For this reason, the CORBA/C++ interface routine for the differentiated code gets not only the simulation's input values from the wrapper but also the actual seed matrix in a condensed representation. The seeding is then done automatically within the interface routine. The other tasks of this routine are similar to the one described above for the simulation–wrapper interface, except that now the *differentiated* top-level routine is called, and the derivatives are returned instead of the corresponding solution vectors. This "automatic seeding" feature has only been added in the current version of our environment. It allows the user to interactively reduce the set of parameters to optimize, even *after* AD code and CORBA interfaces have been generated.

To further illustrate this issue, we consider two basic strategies for choosing parameters for optimization.

Method 1: The user specifies only those parameters that she or he definitely wants to optimize, and generates the differentiated program suitable for this particular case. If, later on, it turns out that more (or other) parameters should be considered for optimization then the AD process has to be carried out again, and the CORBA interface for the new differentiated code must be generated as well. Since both tasks are carried out in a completely mechanical fashion, the only disadvantage of this approach is the processing time to be invested. In the case of very large codes like FLUENT, automatic differentiation and compilation may take several hours.

Method 2: The user specifies all input parameters that might possibly become relevant for optimization, and generates AD code as well as CORBA interfaces for this configuration. The number of parameters to optimize may now be reduced. Since the current set of of optimization parameters is always known to the wrapper module, the seed matrix can be used to filter out the corresponding partial derivatives. Thus, whenever the set of optimization parameters is changed, the seed matrix is changed respectively. This approach does not need any recompilation at all, i.e., the set of optimization parameters can be changed interactively. On the other hand, this approach typically requires more memory at run-time.

Of course, mixing of the two strategies is possible.

In order to build the complete *simulation server* shown in Fig. 1, the simulation function, the differentiated code, and the corresponding CORBA/C++

interfaces have to be linked with a small main program, which is independent from the actual simulation.

4.2 Integration of a MINPACK-1 Optimization Routine

Typical optimization routines require subroutines for the evaluation of the objective function or its derivative. These subroutines must be provided by the user. Thus, these two subroutines provide the means for coupling the optimizers with the simulation software. In our environment, this is done via the wrapper; see Fig. 1.

Go give an example, we discuss the process of integrating the least squares optimization routine `lmder1` from the MINPACK-1 library into our environment. This package is publicly available, e.g., from `http://www.netlib.org`. According to [10] the user-defined subroutine must have the following structure:

```
      subroutine fcn (m,n,x,fvec,fjac,ldfjac,iflag)
      integer n, m, ldfjac, iflag, i,j
      double precision x(n), fvec(m), fjac(ldfjac,n)
      if (iflag .eq. 1) then
c ---    calculate the functions at x
c ---    and return this vector in fvec
      end if
      if (iflag .eq. 2) then
c ---    calculate the Jacobian at x
c ---    and return this matrix in fjac
      end if
      end
```

Usually the user implements the code for the required computations in this routine. In our system, by contrast, the user only needs to put the following subroutine calls at the appropriate places:

```
      call calcfvec(n,x,m,fvec)
```

for the function, and

```
      call calcjac(n,x,m,fjac)
```

for the Jacobian. These external routines perform no computational work, but send a request to the wrapper for evaluating the objective function and the Jacobian, respectively. The routines can be used whenever a function $\mathbf{f} : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ or its Jacobian has to be computed. For convenience, we also provide specialized interface routines taylored to the evaluation of scalar-valued functions and gradients.

The header of the main optimization routine is given below:

```
      subroutine lmder1(fcn,m,n,x,fvec,fjac,ldfjac,tol,
     +                  info,ipvt,wa,lwa)
      integer m,n,ldfjac,lwa,info
      integer ipvt(n)
      double precision tol
      double precision x(n), fvec(m), fjac(m,n), wa(lwa)
      external fcn
```

To make this routine available from within Python we employed the *Pyfort* tool [8]. Since Pyfort cannot handle function names in a subroutine's calling sequence, we removed the first argument. After this modification, we utilized Pyfort to create a shared library `minpackmodule.so`, which can be accessed from Python. Note that the resulting optimization module is independent from the actual problem configuration because the evaluation of the objective function (resp. the Jacobian) has been separated from the optimization part. Therefore, the steps described in this subsection have to be carried out only once, and from this point on the generated Python module is usable for solving arbitrary optimization problems within our environment.

4.3 A Sample Optimization Session

In the following we will show how our system can be used to solve a typical data assimilation problem. For this particular test example, *Flow Through a Filter Cartridge*, we consider a turbulent flow at a Reynolds number $\mathrm{Re} \approx 10^5$ using the k-ε turbulence model.

We want to determine values for three of the turbulence parameters, $c_{1\varepsilon}$, $c_{2\varepsilon}$, and c_μ, such that the pressure distribution computed with the FLUENT CFD solver best matches given experimental data at certain grid points. For the time being, the remaining two parameters $\sigma_k = 1.0$ and $\sigma_\varepsilon = 1.3$ are considered fixed. But as we do not know if these two values are correct, they might be included in later optimization problems.

In our example, the test data $\mathbf{d}$ have been generated artificially by running the simulation with the parameter set ($c_{1\varepsilon} = 1.44, c_{2\varepsilon} = 1.92, c_\mu = 0.09, \sigma_k = 1.0, \sigma_\varepsilon = 1.3$) and saving the results for the pressure distribution to file.

The following extract from a Python session shows how we set up the optimization problem:

```
1. c1_eps    = newInputVar(wrapper,"c1",1,[1.44])
2. c2_eps    = newInputVar(wrapper,"c2",1,[1.92])
3. cmu       = newInputVar(wrapper,"cmu",1,[0.09])
4. sigma_k   = newInputVar(wrapper,"eprnd",1,[1.0])
5. sigma_eps = newInputVar(wrapper,"dprnd",1,[1.3])
6. setInputVars(wrapper,[c1_eps,c2_eps,cmu,sigma_k,sigma_eps])
7. pressure = newOutputVar(wrapper,"pressure",336,1)
8. setOutputVars(wrapper,[pressure])
9. setOptVars(wrapper,[c1_eps,c2_eps,cmu,sigma_k,sigma_eps])
```

In lines 1–6 the input variables are specified. For each input variable we indicate its name in the simulation code, the size (if it is an array variable), and its default value(s) for the case that the variable is not optimized, i.e., it is considered to be a "fixed" parameter. In our case, all parameters are scalars, and therefore the size for each variable is set to 1. In lines 7 and 8 we specify the output variable. The solution vector `pressure` is defined as an array of size 336, corresponding to the number of cells in the underlying grid. The last entry is used to define the mapping between simulation output and external test data. In line 9 we specify all the input variables that *might* be optimized later on.

At this point we can generate a control script for ADIFOR as well as the CORBA interfaces for the `filter` routine, and apply the ADIFOR tool to obtain the differentiated routine `g_filter`. Note that, due to our definition of the optimization variables in line 9, `g_filter` will be able to compute derivatives with respect to all five parameters or a subset of them, depending on the actual seeding. After providing the test data **d** to the wrapper and importing the optimizer module we are ready to start the optimization.

Considering σ_k and σ_ε to be fixed, we are going to optimize $c_{1\varepsilon}, c_{2\varepsilon}$, and c_μ. Hence, we redefine the set of optimization variables and provide an initial guess for these 3 variables:

```
setOptVars(wrapper,[c1_eps,c2_eps,cmu])
x = array([1.1,1.2,0.1],Float64)
```

After setting further control parameters, which are not shown here for the sake of brevity, we call the optimization routine `lmder1` from the MINPACK module. In addition, we print the result `x` and the final euclidean norm of the residuals.

```
import minpack
fvec,fjac,info,ipvt=minpack.lmder1(336,3,x,ldfjac,tol,wa,lw)
print x,minpack.enorm(336,fvec)
```

The optimization takes 6 function calls and 5 evaluations of the Jacobian and produces the output

```
[ 1.44000001  1.91999998  0.09000001] 3.39489005512e-06
```

Here, the first 3 numbers are the final approximation for the variables $c_{1\varepsilon}, c_{2\varepsilon}$, and c_μ whereas the fourth number is the Euclidean norm of the residuals. Indeed, the solution shows a good agreement with the parameter set that was used to generate the test data. However, in general users may want to verify the robustness of the solution, e.g., by trying a different optimization package. Thus we let the same problem be solved again by another optimizer, namely the bound-constrained least squares optimization routine `dn2gb` from the PORT [11] library, which has been integrated in our system as well. We import the corresponding Python module, and provide the initial guess as above. The array `bounds` will be used to pass the constraints to the optimization routine. For sake of brevity the initialization of the remaining variables is ommited here.

```
import port
x = array([1.1,1.2,0.1],Float64)
lower_bounds = [0.01]*3
upper_bounds = [2.0]*3
bounds = array([lower_bounds,upper_bounds],Float64)
port.dn2gb(336,3,x,bounds,iv,liv,lv,v,ui,ur)
```

The solution found by this optimizer confirms the above result.

5 Conclusions

We have presented a software environment for facilitating the combination of simulation and optimization software by enabling much of the interfacing work to be done automatically. Our approach treats the evaluation of the simulation function, one iteration of the optimizer, and the computation of the optimizer's objective function as basic tasks and provides an infrastructure for the interplay of these tasks.

A considerable number of languages and tools play together in our software environment. Object-oriented languages are used for managing the data and control flows between the different components at the executable level, whereas we rely on highly optimizable imperative languages, e.g., Fortran, for the computationally intensive tasks.

One reason for selecting CORBA is that it greatly simplifies distributed execution. For example, in our experiments the simulation ran on a SUN Fire 6800 high-end compute server, whereas a standard PC was used for the optimizer and the user interface. The C++ interfaces to the Fortran codes are added to simplify the remote calls via CORBA.

The wrapper and the user interface are written in Python because of its flexibility and ease of use.

Finally, the ADIFOR automatic differentiation tool is employed for automatically augmenting the simulation code such that it computes derivative information together with the function values.

In addition to the FLUENT simulation package and the least squares optimizers from the MINPACK-1 and PORT libraries which are mentioned in this note, we have integrated another large simulation package, SEPRAN [19], as well as optimizers for scalar-valued objective functions like, e.g., L-BFGS-B [20] and UNCMIN [18], thus showing the versatility of our environment. Due to its modular structure, components can easily be replaced with others providing comparable functionality. Thus, experimenting with different simulation packages and/or different optimization algorithms is greatly simplified.

References

1. M. Berz, C. Bischof, G. Corliss, and A. Griewank. *Computational Differentiation: Techniques, Applications, and Tools.* SIAM, Philadelphia, PA, 1996.
2. C. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.
3. C. Bischof, L. Roh, and A. Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software: Practice and Experience*, 27(12):1427–1456, 1997.
4. C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch, and J. W. Risch. A CORBA-based environment for coupling large-scale simulation and optimization software. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2001, Las Vegas, USA, June 25–28, 2001*, volume 1, pages 68–72. CSREA Press, 2001.
5. G. Corliss, A. Griewank, C. Faure, and L. Hascoët, editors. *Automatic Differentiation 2000: From Simulation to Optimization.* Springer, 2001. To appear.
6. CRC for Distributed Systems Technology, The University of Queensland, Australia. *The Python CORBA ORB*, 1999. http://www.fnorb.org.
7. J. Czyzyk, M. P. Mesnier, and J. J. Moré. The network-enabled optimization system (NEOS) server. Technical Report ANL/MCS-P615-1096, Argonne National Laboratory, March 1997.
8. P. F. Dubois and T.-Y. Yang. Extending Python with Fortran. *Computing in Science & Engineering*, 1(5):66–73, 1999.
9. Fluent Inc., Lebanon, NH. *FLUENT Tutorial Guide*, 1995.
10. B. S. Garbow, K. E. Hillstrom, and J. J. Moré. User Guide for MINPACK-1. Report ANL-80-74, Argonne National Laboratory, Argonne, 1980.
11. D. M. Gay. Usage summary for selected optimization routines. Computing Science Technical Report 153, AT&T Bell Laboratories, Murray Hill, 1990.
12. R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Trans. Math. Softw.*, 24(4):437–474, 1998.
13. A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, Philadelphia, PA, 2000.
14. A. Griewank and G. Corliss. *Automatic Differentiation of Algorithms.* SIAM, Philadelphia, PA, 1991.
15. A. Griewank, D. Juedes, and J. Utke. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw.*, 22(2):131–167, 1996.
16. M. E. Henderson, C. R. Anderson, and S. L. Lyons, editors. *Proceedings of the 1998 SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing*, Philadelphia, PA, 1999. SIAM.
17. S.-L. Lo, D. Riddoch, and D. Grisby. *The omniORB Version 3.0 User's Guide.* AT&T Bell Laboratories, Cambridge, May 2000. Available from http://www.uk.research.att.com/omniORB.
18. R. B. Schnabel, J. E. Koontz, and B. E. Weiss. A modular system of algorithms for unconstrained minimization. *ACM Trans. Math. Softw.*, 11:419–440, 1985.
19. G. Segal. *SEPRAN Users Manual.* Ingenieursbureau Sepra, Leidschendam, NL, 1993.
20. C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal. Algorithm 778: L-BFGS-B, Fortan subroutines for large-scale bound constrained optimization. *ACM Trans. Math. Softw.*, 23, 1997.

Generic Programming for High Performance Scientific Applications

Lie-Quan Lee and Andrew Lumsdaine

{llee,lums}@osl.iu.edu
Open Systems Laboratory
Computer Science Department
Indiana University
Bloomington, IN 47405

Abstract. We present case studies that apply generic programming to the development of high-performance parallel codes for solving archetypal PDEs. We examine the overall structure of the example scientific codes and consider their generic implementation. With a generic approach it is a straightforward matter to reuse software components from different sources; implementations with components from Blitz++, A++/P++, MTL, and Fortran BLAS are presented. We compare the generic implementation to equivalent implementations developed with alternative libraries and languages and discuss not only performance but software engineering issues as well.

1 Introduction

Generic programming is an emerging software development paradigm that has simultaneous emphases on both reusability and efficiency. In our previous work, we have used generic programming methodologies to develop libraries in key areas of scientific computing: basic numerical linear algebra [1, 2], sparse matrix ordering [3], and iterative solvers [1, 4]. These libraries, although implemented completely in native C++, exhibit performance comparable to the fastest known alternatives (including vendor-tuned and automatically-tuned libraries).

In this paper we move beyond individual generic libraries and present case studies that apply generic programming to developing a parallel code for solving two archetypal PDEs. Several aspects of using generic programming are examined, including code structure, interface specification, reusability, and, of course, performance. We examine the overall structure of the example scientific codes and consider their generic implementation. We also consider the generic interface specifications requirements for key subsystems (e.g., an iterative linear solver). Since reuse is one important purpose of a generic algorithm interface, we demonstrate how reuse can be accomplished by mapping the interfaces of popular array libraries (Blitz++ [5], A++/P++ [6]), the Matrix Template Library (MTL) [1, 2] and Fortran BLAS [7–9] for use with our generic algorithms. Finally, we compare the generic implemntation to equivalent implementations developed with alternative libraries and languages and discuss not only performance but software engineering metrics as well. Also presented is a novel technique for automatically generating communication patterns for parallel sparse matrix computations and for creating MPI-based communication structures to carry them out.

2 Problems and Algorithm

We perform experiments to solve two archetypal problems — the so-called Bratu problem and the driven cavity flow problem.

The classic nonlinear elliptic PDE Bratu problem originates in solid fuel ignition. We consider a two dimensional unit domain. The governing equation is given by

$$-\nabla^2 u - \lambda e^u = 0$$

Here, λ is a constant known as the Frank-Kamenetskii parameter in the problem context. It is chosen to be 6.0 in our testing cases. We take $u = 0$ at the boundary,

The second problem is thetwo dimensional thermally and dynamically driven cavity flow governed by four coupled nonlinear equations involving unknowns of horizontal and vertical velocity (u and v), vorticity ω, and temperature T. The equations are dimensionless such that the domain becomes a unit square and the temperature varies from zero to unity [10].

$$\begin{aligned} -\nabla^2 u - \frac{\partial \omega}{\partial y} &= 0 \\ -\nabla^2 v + \frac{\partial \omega}{\partial x} &= 0 \\ -\nabla^2 \omega + \frac{\partial (u\omega)}{\partial x} + \frac{\partial (v\omega)}{\partial y} - Gr\frac{\partial T}{\partial x} &= 0 \\ -\nabla^2 T + Pr(\frac{\partial (uT)}{\partial x} + \frac{\partial (vT)}{\partial y}) &= 0 \end{aligned}$$

Here, Gr and Pr are Grashof and Prandtl numbers, respectively.

By definition, vorticity is $\omega = -\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}$. No-slip, rigid-wall Dirichlet conditions are used for u and v. Dirichlet conditions are used for ω, where along each constant coordinate boundary, the tangential derivative is zero. Dirichlet conditions are used for T on the left and right walls, and insulation homogeneous Neumann conditions are used for T on the top and bottom walls.

Both problems lead to nonlinear algebraic equations after central difference discretization. The Newton-Krylov iterative method [11–13] is used to solve them. The outer Newton iteration approximates the solution x of $f(x) = 0$ through a sequence of iterates $x^i = x^{i-1} + \alpha \cdot \delta x^i$ starting with an initial x^0. The update δx^i is an approximate solution of Newton correction linear system

$$f'(x)\delta x = -f(x^{i-1})$$

The above linear system is solved by a Krylov subspace iterative method, such as restarted GMRES [14]. The Jacobian $f'(x)$ is approximated by $f'(x^{i-1})$ or a matrix-free method is used for evaluating the quantity $f'(x)y$.

3 Related Work

The Portable, Extensible Toolkit for Scientific Computation (PETSc) [15] is a well-known library that provides comprehensive functionality for solving scientific problems. PETSc includes basic sparse and dense linear algebra operations, preconditioners, Krylov subspace methods, and nonlinear solvers. However, PETSc is not designed to be generic. For instance, user-defined data structures not provided in PETSc cannot be used in its Krylov subspace methods. Similarly, the Aztec [16] library also requires users to use its own matrix and vector data structures.

One the other hand, Trilinos [17] and the Equation Solver Interface (ESI) [18] use object-oriented techniques in C++ to allow extension through inheritance. Genericity is achieved by defining an abstract interface via base classes with virtual functions specifying the requirements. Generic functions in this context are written in terms of base classes; function calls are dispatched at run-time based on the concrete types of objects.

Providing an abstract interface via base classes does provide the separation of implementation and interface that is an important part of generic programming. However, this approach has several drawbacks which do not exist in C++ template-based approach.[1] First, virtual functions introduce run-time overhead due to

[1] To be precise, the object-oriented approach relies on *subtype polymorphism* whereas the template-based generic programming approach relies on *parametric polymorphism.*

virtual function table lookup. For example, in ESI, to get matrix row and column sizes, a virtual function, `getGlobalSizes(Ordinal& rows, Ordinal& rows)`, is invoked. This incurs a virtual function lookup as well as the function call overhead itself. With a template-based approach, the code to get matrix row or cloumn sizes is easily inlined and will not incur any overhead. Second, as just alluded to, using virtual functions may interfere with compile-time optimization because it is impossible to inline virtual functions in general. Third, it is difficult to express type requirements through base classes. Finally, using member functions in base classes to express requirements may cause the requirements to be more heavyweight than necessary. In particular, the virtual function will have a return type that will induce an interface requirement, even if the return type does not need to be explicitly specified (indeed, it is often the case that one would want to leave this unspecified). In generic programming, *valid expressions* rather than member functions are used to express requirements.

4 Interfaces

We base the iterative solver portion of this case study on routines from the Iterative Template Library (ITL) [1]. It is important to note that the ITL is itself generic and is intended for reuse. In particular, the ITL is intended to be used with libraries other than the Matrix Template Library (MTL).

The following is a function template for the conjugate gradient algorithm [19]. The names of the parameterized types correspond to different *concepts*[2] and convey specific requirements. Any type conforming to the requirements listed in a corresponding concept can be used with the algorithm.

```
template <typename LinearOperator, typename CGVectorX,
          typename CGVectorB, typename HermitianPreconditioner,
          typename Iteration>
int
cg(const LinearOperator& A, CGVectorX& x, const CGVectorB& b,
   const HermitianPreconditioner& M, Iteration& iter) {
  //concept checking here ...

  typedef itl::itl_traits<CGVectorX>       Traits;
  typedef typename Traits::internal_vector TmpVec;
  typename Traits::value_type rho, rho_1;
  TmpVec p(size(x)), q(size(x)), r(size(x)), z(size(x));

  itl::mult(A, itl::scaled(x, -1.0), b, r);
  while (! iter.finished(r)) {
    itl::solve(M, r, z);
    rho = itl::dot_conj(r, z);
    if (iter.first())
      itl::copy(z, p);
    else
      itl::add(z, itl::scaled(p, rho / rho_1), p);
    itl::mult(A, p, q);
    typename Traits::value_type alpha = rho / itl::dot_conj(p, q);
    itl::add(x, itl::scaled(p, alpha), x);
    itl::add(r, itl::scaled(q, -alpha), r);
    rho_1 = rho;
    ++iter;
  }
  return iter.error_code();
}
```

[2] In generic programming, a *concept* is a set of valid expressions, associated types, and semantics used to represent an equivalence class of types in an interface specification.

Table 1 lists the concepts in the above conjugate gradient algorithm and summarizes their valid expressions.

Valid Expressions	Description
LinearOperator	
`mult(A, x, y)`	Linear transformation $w \leftarrow Ax$
CGVector	
`scaled(x, alpha)`	Lazy evaluation of vector scaling
`add(x, y, z)`	Vector addition $z = x + y$
`copy(x, y)`	To copy x to y
`size(x)`	The size of x
`dot_conj(x, y)`	Conjugate dot $x^T \cdot y$
`itl_traits<V>::value_type`	The element value type of V
HermitianPreconditioner	
`solve(M, y, x)`	To solve $Mx = y$
Iteration	
`iter.first()`	To test whether it is the first iteration
`iter.finish(r)`	To test convergence
`++iter`	To increase iteration counter
`iter.error_code()`	To return error code

Table 1. Summary of the concepts used in the conjugate gradient algorithm. A is an object whose type models LinearOperator concept. x, y, z and r are objects whose types model CGVector concept. $alpha$ is a scalar.

It is important to note that the C++ language does not provide a mechanism for the algorithm writer to explicitly state to which concept the user-supplied template argument should conform. The concept-checking technique described in [20] is used to provide compile-time checks for template parameters and verify the concept requirements.

The ITL provides algorithm and data structure interoperability. With a very thin adapter layer, the ITL can use data structures from any library that provides the necessary functionality (as required by the specified concepts). The interface layer is used only to adapt syntax. As an example, we present a thin adapter layer to allow a Blitz++ array to model the CGVector using the technique of external adaptation [21]. External adaption wraps a new interface around a data structure without copying data or placing the data inside the adaptor objects. The ITL is carefully designed to accommodate this type of adaptation.

First we define a structure for lazy evaluation of vector scaling. It wraps a vector and a scalar inside and provides access methods (`vec()` and `alpha()`) to them.

```
template <class Vec, class T>
struct Scaled {
  Scaled(const Vec& v, const T& alpha) : _alpha(alpha), _v(v) { }
  T alpha() const { return _alpha; }
  const Vec& vec() const { return _v; }
protected:
  T _alpha;
  const Vec& _v;
};
```

Next, the function template of lazy evaluation flavor of vector scaling for Blitz++ array, `copy()`, `dot_conj()`, `size()`, and `add()` operations are in the defined. Each function template is one line of code.

```
template <class prec, int dim, class T>
inline Scaled<Array<prec, dim>, T>
scaled(const Array<prec, dim>& v, const T& alpha)
{ return Scaled<Array<prec, dim>, T> (v, alpha); }

template <class Vec>
inline void copy(const Vec& a, Vec& b)
{ std::copy(a.begin(), a.end(), b.begin()); }

template <class VecA, class VecB>
typename VecA::T_numtype
dot_conj(const VecA& a, const VecB& b)
{ return sum(a * conj(b)); }

template <typename T, int dim>
inline Array<T, dim>::T_index
size(const Array<T, dim>& x)
{ return x.shape(); }

template <class Vec, class T>
inline void add(const Vec& v, const Scaled<Vec,T>& sv, Vec& vd)
{ vd = sv.alpha() * sv.vec() + v; }
```

A partial specialization of `itl_traits` is defined to allow Blitz++ array to provide the required `value_type` in the concept CGVector.

```
template <class prec, int dim>
struct itl_traits< Array<prec, dim> > {
  typedef prec value_type;
};
```

To use Blitz++ arrays in all the ITL solvers, a few additional wrapper functions such as variants of addition are needed. Data structures from the A++ library can be used in the ITL with a similar thin adapter layer.

We note that the LinearOperator in Table 1 is not limited to traditional matrices; matrix-free operators can meet the requirements of LinearOperator. The following defines a matrix-free operator that models the LinearOperator concept. The code is parameterized (generic) and can be used with arbitrary vector classes, such as those in MTL or Blitz++.

```
template <class Vector, class NonlinearFunction>
class matrix_free_operator {
public:
  typedef typename itl_traits<Vector>::value_type value_type;
  typedef typename itl_traits<Vector>::size_type size_type;
  matrix_free_operator(NonlinearFunction f,
                       const Vector& x, const Vector& z) :
    f0(f), x0(size(x)), z0(size(z)), tmp0(size(x)) {
      itl::copy(x, x0);
      itl::copy(z, z0);
      sigma = 1.e-6 * itl::two_norm(x) + 1.e-8;
  }
  template<class VectorX, class VectorY>
  void
  apply(const VectorX& x, VectorY& y) const {
    // tmp0 <- x0 + sigma*x
```

```
    itl::add(x0, itl::scaled(x, sigma), tmp0);
    // y <- f(tmp0)
    f0(tmp0, y);
    // y <- y - f(x0)
    itl::add(y, itl::scaled(z0, -1.0), y);
    itl::scale(y, 1.0/sigma);
  }
private:
  NonlinearFunction f0;
  Vector x0;
  Vector z0;
  mutable Vector tmp0;
  double sigma;
};

template <class Vector, class NonlinearFunction,
          class VectorX, class VectorY>
inline void
mult(const matrix_free_operator<Vector, NonlinearFunction>& A,
     const VectorX& x, VectorY& y)
{  A.apply(x, y);  }
```

The following is a code excerpt that uses Blitz++ to solve the Bratu problem. In this case, it is quite natural to use a *two dimensional* Blitz++ array to represent unknowns and function values on the two dimensional descretized grid points. The Blitz++ stencil operator is defined to compute function values and incorporated into the matrix-free method (which is used instead of an explicit Jacobian).

```
//define blitz stencil for the bratu problem
BZ_DECLARE_STENCIL2(Bratu_stencil, U, F)
  F = -Laplacian2D(U) - exp(U) * lamhxhy;
BZ_END_STENCIL

struct apply_op {               // Wrap up applyStencil
  void operator()(Array<double, 2>& U, Array<double, 2>& F) const
  { applyStencil(Bratu_stencil(), U, F); }
};

int main(int argc, char *argv[]) {
  // local variables here ...
  Array<double, 2> U(shape(nx, ny)), F(shape(nx, ny)), DX(shape(nx, ny));
  // Initial guess for U ...

  identity_preconditioner p;
  apply_op f;                   // wrapper of applyStencil
  f(U, F);                      // compute nonlinear function f
  modified_gram_schmidt< Array<double, 2> > orth(restart, size(U));

  // Start Newton solver
  for (k = 0; k < kmax; ++k) {
    matrix_free_operator<Array<double, 2>, apply_op> A(f, U, F);
    basic_iteration<double> iter(F, iter_max, ksp_rtol, ksp_atol);

    DX = 0.0;
    //  Solve J DX = F  with matrix-free GMRES
    gmres(A, DX, F, p, restart, iter, orth);

    U -= DX;
```

```
    f(U, F);
    //check convergence here ...
  }
  //...
}
```

5 Parallelization

Parallelization can be easily introduced by using an adapter layer to provide parallelized version of iterative methods. No changes to the iterative solver algorithms themselves are required. We briefly describe our approach. Without loss of generality, in our illustration we use a a one dimensional row-wise partition of the matrices and vectors. We also use the Message Passing Interface (MPI) [22, 23] for communication.

To accomplish a sparse parallel matrix-vector multiply $w \leftarrow Ax$, we need to communicate data of vector x between processors so that each processor can carry out its local part of the computation. However, since A is sparse not all remote elements in x need to be communicated. We only need to communicate those few elements in x that will be accessed in the multiplication. In general, this will significantly reduce the amount of data that must be communicated from one processor to another as well as the number of processors that must communicate at all. The access pattern into the vector x is determined by the non-zero structure of the sparse matrix A and will not change if the matrix structure does not change.

The exact access pattern can be obtained by traversing all local nonzero elements in the matrix and collecting column indices. Interestingly, this traversal has precisely the same form as the matrix-vector product we wish to parallelize. The access patterns are thus computed by re-using a generic matrix-vector multiply with the sparse matrix to be analyzed along with a special `pattern_finder` class for the "vectors." When invoked with the matrix-vector algorithm, the pattern-finder class records the indexing information necessary to compute the communication access pattern.

```
  //recv_displacements is an array to record indices
  pattern_finder pf(recv_displacements);
  mtl::mult(A, pf, pf);
```

The implementation of `pattern_finder` is straightforward.

```
class pattern_finder {
public:
  // typedefs here ...
  inline pattern_finder(int* p_) : pattern(p_) {}
  inline int operator[](int i) const {
    pattern[i] = 1; //set to be one for elements accessed
    return x;
  }
  // ...
protected:
  int x;
  mutable int* pattern;
};
```

From the access pattern, the size and displacement of elements is needed from each MPI process in order to compute the local linear transformation and can be easily determined. The next step is to create communication structures using MPI datatypes and persistent communication requests.

1. Send each process the size of elements needed
2. Receive the sizes of elements to be sent from other processes

3. Send each process the displacement of elements needed
4. Receive the displacements of elements to be sent from other processes
5. Create MPI datatypes for send and receive
6. Initialize persistent communication

Access pattern analysis and communication structure creation is performed once for a matrix, outside of the Krylov iteration. A straightforward implementation of the parallel linear transformation itself (the matrix-vector product) looks like the following where we reuse the serial version.

```
template <typename Matrix, typename VecX, typename VecY>
void mult(const parallel_matrix<Matrix>& A,
          const VecX& x, VecY& y) {
  // copy x to be ready for persistent send
  std::copy(x.begin(), x.end(), wx.begin()+pos*rank);

  // start communication
  Request::Startall(total_num_send_recv, request_array);
  mtl::mult(A.diagonal_block(), x, y);

  // finish communication
  Request::Waitall(total_num_send_recv, request_array);

  //at this point all necessary entries in wx are there
  mtl::mult(A.off_diagonal_block(), wx, y);
}
```

One optimization to this approach would be to process multiplication operations as the remote data arrive rather than waiting for all of them.

There is an interesting abstraction conflict presented by this interface. On the one hand, we wish to be able to provide a general interface that allows multiplication between A and any x. On the other hand, we also would like to establish persistent communication requests that require persistent knowledge of which data they are going to be communicating. The simple solution (used here) is to copy the data from x to persistent communication buffers. Alternatively, a memorization process could be used to create and reuse persistent requests on demand.

All parallel vector operations will be the same for serial versions except for dot product and conjugate dot product. Parallel (conjugate) dot product is implemented by a serial (conjugate) dot product plus a reduction with summation.

As we know, using preconditioners in Krylov subspace iterative solvers will help convergence dramatically and we want our parallel interface to include parallel preconditioners. Block-wise versions of preconditioners are simple yet effective for most applications. Matrix reordering methods such as reverse Cuthill-McKee method [24, 25] or Self-Avoiding Walk [26] will add more weight for using block-wise preconditioners since they often reorder sparse matrices to have less bandwidth. To implement block-wise preconditioners, we reuse the serial versions with a minor extension. For example, Block Incomplete LU is the same as serial version of ILU with one different argument on construction.

6 Experiments and Results

The experiments for the results shown here were performed on a Sun UltraSPARC cluster connected by 100MB-baseT switched Ethernet. Each cluster node was a dual processor 400MHz UltraSPARC machine with 512MB RAM. We arranged the experiments to run one MPI process per cluster node. LAM/MPI version 6.3.2 [27] was used as the MPI implementation. PETSc [28] release 2.0.28, compiled by Sun WorkShop

6 update 1 with "make BOPT=O", is used in the comparison. Parallel ITL is compiled by KCC-3.4g with the following optimization flags:

```
+K3 -O -fast -fsimple --abstract_pointer \
--backend -fast --backend -x04 --backend -xdepend \
--backend -xtarget=ultra2 --backend -xarch=v8plusa \
--backend -xrestrict=\%all --inline_keyword_space_time=15000.0
```

Fig. 1 and Fig. 2 show the execution time for solving the Bratu problem on a 256×256, 512×512, and 1024×1024 grid using Parallel ITL and PETSc with and without preconditioners. Both packages achieve virtually the same performance. Fig. 3 and Fig. 4 plot the parallel speed-up with various grid sizes. PETSc was invoked as follows:

```
mpirun N ex5 -mx Xgrid -my Ygrid -Nx 1 -Ny Number_of_Node \
-snes_monitor -ksp_type gmres -pc_type PC -ksp_max_it 300 \
-ksp_gmres_restart 60 -ksp_gmres_modifiedgramschmidt \
-snes_max_it 60 -ksp_rtol 0.5 -ksp_atol 0.005 -snes_rtol 0.005
```

where `Xgrid` and `Ygrid` are number of grids in x or y direction, and `PC` is either "none" or "bjacobi".

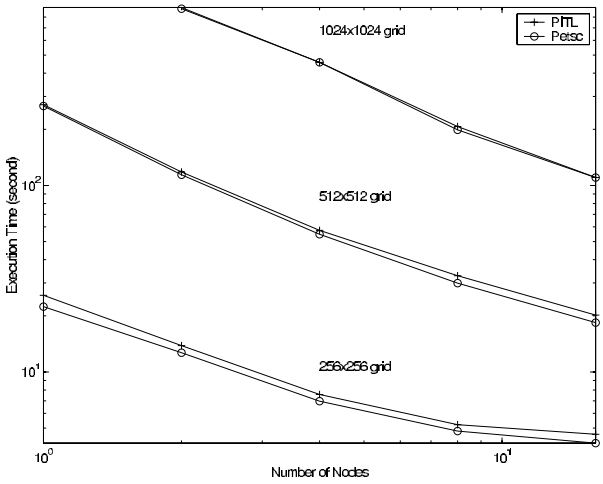

Fig. 1. Performance comparison of Parallel ITL and PETSc. The two-dimensional Bratu nonlinear PDE is solved using Parallel ITL and PESTSc without preconditioner.

For the second set of results, we use a matrix-free method using GMRES with modified Gram-Schmidt orthogonalization for the driven cavity problem. No preconditioner is used in this case. The results are shown in Fig. 5, where we plot the execution time for three different problem sizes as a function of number of processors. Again, performance of the generic approach is identical to that of PETSc. The parallel speed-up of each case is shown in Fig. 6.

One of the most important principles in software engineering is that of *separation of concerns* [29]. This principle states that a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity and to achieve the required engineering quality factors such as adaptability, maintainability, extendability and reusability. Generic programming provides a clean way to reduce coupling between components while it provides a formal mechanism (concepts) to address interfaces between them. In our case studies generic programming enabled the decoupling of iterative solvers from basic linear algebra operations in ITL. We can use any data structures in those libraries providing the necessary functionality in our iterative solvers.

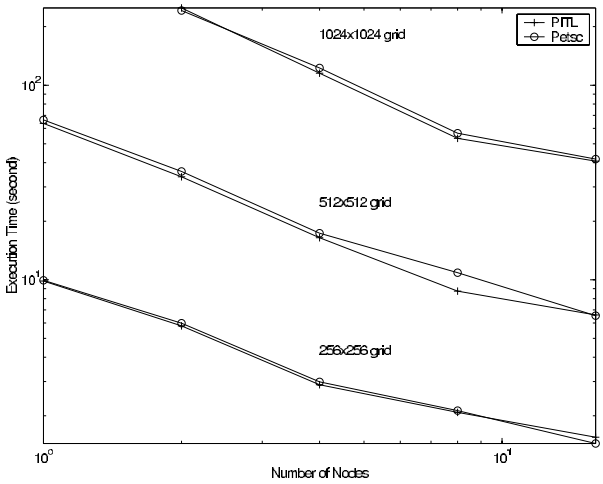

Fig. 2. Performance comparison of Parallel ITL and PETSc. The two-dimensional Bratu nonlinear PDE is solved using parallel ITL and PESTSc with block ILU preconditioner.

The decoupling of iterative solvers and parallelization by a thin non-trivial interface was also enabled by the generic programming paradigm used in the software development. Not only is high performance as exhibited in the above results, but debugging and testing are simpler. For example, each components such as GMRES and the parallel matrix-vector multiplication can be tested independently.

Acknowledgments

The authors are grateful to David E. Keyes for graciously providing references for the thermally driven cavity problem and for his valuable comments. We also acknowledge helpful discussions with Jeremy Siek and Todd Veldhuizen. This work was supported by NSF grant ACI-9982205.

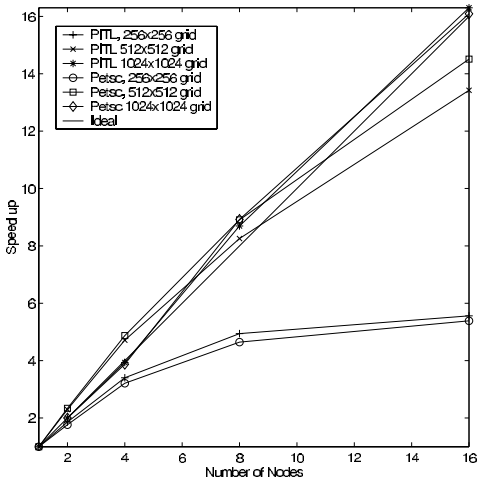

Fig. 3. Parallel speed-up for unpreconditioned Bratu.

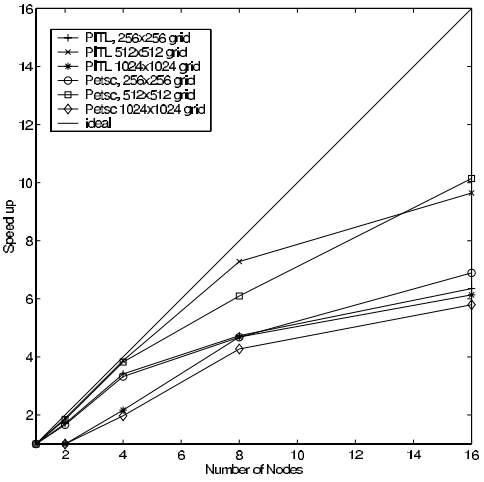

Fig. 4. Parallel speed-up for preconditioned Bratu.

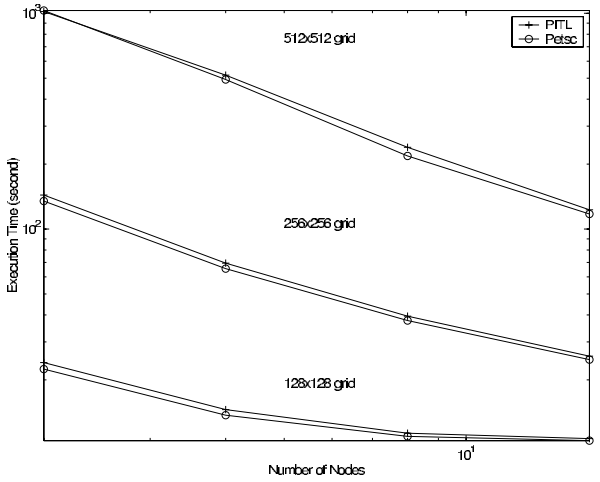

Fig. 5. Performance of the two-dimensional driven cavity problem solved by parallel ITL and PETSc.

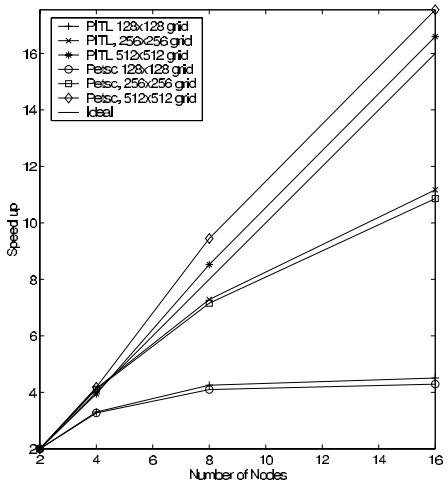

Fig. 6. Parallel speed-up for the driven cavity problem.

References

1. Siek, J., Lumsdaine, A., Lee, L.Q.: Generic programming for high performance numerical linear algebra. In: Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98), SIAM Press (1998)
2. Lumsdaine, A., Siek, J., Lee, L.Q.: The matrix template library home page. (`http://ww.osl.iu.edu/research/mtl`)
3. Lee, L.Q., Siek, J.G., Lumsdaine, A.: Generic graph algorithms for sparse matrix ordering. In: ISCOPE'99. Lecture Notes in Computer Science, Springer-Verlag (1999)
4. Lumsdaine, A., Lee, L.Q., Siek, J.: The iterative template library home page. (`http://ww.osl.iu.edu/research/itl`)
5. Veldhuizen, T.: Blitz++ home page. (`http://oonumerics.org/blitz`)
6. Quinlan, D.: A++/P++ Manual. (Lawrence Livermore National Laboratory)
7. Dongarra, J., Croz, J.D., Hammarling, S., Hanson, R.: Algorithm 656: An extended set of basic linear algebra subprograms: Model implementations and test programs. ACM Transactions on Mathematical Software **14** (1988) 18–32
8. Dongarra, J., Croz, J.D., Duff, I., Hammarling, S.: A set of level 3 basic linear algebra subprograms. ACM Transactions on Mathematical Software **16** (1990) 1–17
9. Lawson, C., Hanson, R., Kincaid, D., Krogh, F.: Basic linear algebra subprograms for fortran usage. ACM Transactions on Mathematical Software **5** (1979) 308–323
10. Bennett, B.A.V., Smooke, M.D.: Local rectangular refinement with application to nonreacting and reacting fluid flow problems. Journal of Computational Physics **151** (1999) 684–727
11. Cai, X.C., Gropp, W.D., Keyes, D.E., Tidriri, M.D.: Newton-Krylov-Schwarz methods in CFD. In Hebeker, F., Rannacher, R., eds.: The International workshop on Numerical Methods for the Navier-Stokes Equations. (1994)
12. Gropp, W.D., Keyes, D.E., McInnes, L.C., Tidriri, M.D.: Globalized Newton-Krylov-Schwarz algorithms and software for parallel implicit CFD. International Journal of High Performance Computing Appolications **14** (2000) 102–136
13. Hayder, M.E., Ierotheou, C., Keyes, D.E.: Three parallel programming paradigms: Comparisons on an archetypal PDE computation. Parallel and Distributed Computing Practices (2000) 35–53
14. Saad, Y., Schultz, M.: GMRES: A generalized minimum residual algorithm for solving nonsymmetric linear systems. SIAM J. Sci. Statist. Comput. **7** (1986) 856–869
15. Gropp, W.D., Smith, B.: PETSc: Portable extensible tools for scientific computation. Technical report, Argonne National Laboratory, Argonne, IL (1994)
16. Tuminaro, R.S., Heroux, M., Hutchinson, S.A., Shadid, J.N.: Official Aztec User's Guide: Version 2.1. (1999)
17. The Trilinos Team: (The Trilinos project) http://www.cs.sandia.gov/~mheroux/Trilinos/doc/Trilinos.html.
18. The ESI technical forum: (Equation Solver Interface (ESI) standards multi-lab working group) http://z.ca.sandia.gov/esi.
19. Hestenes, M.R., Stiefel, E.: Methods of conjugate gradients for solving linear systems. J. Res. Nat. Bur. Standards **49** (1952) 409–436
20. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: First Workshop on C++ Template Programming, Erfurt, Germany. (2000)
21. Boost: (Boost Graph Library) `http://www.boost.org/libs/graph/doc/index.html`.
22. Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI The Complete Reference. MIT Press, Cambridge, MA (1996)
23. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., , Snir, M.: MPI — The Complete Reference: Volume 2, the MPI-2 Extensions. MIT Press (1998)
24. Cuthill, E.H., McKee, J.: Reducing the bandwidth of sparse symmetric matrices. In: Proc. 24^{th} National Conference of the ACM, ACM Press (1969) 157–172
25. Liu, W., Sherman, A.: Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithnms for sparse matrices. SIAM J. Numerical Analysis (1976) 198–213
26. Heber, G., Biswas, R., Gao, G.: Self-avoiding walks over adaptive unstructured grids. In: Parallel and Distributed Processing. Number 1586 in LNCS, Spriger-Verlag (1999) 968–977
27. The LAM Team: Getting Started with LAM/MPI. University of Notre Dame, Department of Computer Science, `http://www.lam-mpi.org/`. (1998)
28. Balay, S., Gropp, W.D., McInnes, L.C., Smith, B.F.: Efficient management of parallelism in object-oriented numerical software libraries. In Arge, E., Bruaset, A.M., Langtangen, H.P., eds.: Modern Software Tools in Scientific Computing. Birkhauser (1997)
29. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976)