Automatic Parallelization of the Conjugate Gradient Algorithm

Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill

Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract. The conjugate gradient (CG) method is a popular Krylov space method for solving systems of linear equations of the form $Ax = b$, where A is a symmetric positive-definite matrix. This method can be applied regardless of whether A is dense or sparse. In this paper, we show how restructuring compiler technology can be applied to transform a sequential, dense matrix CG program into a parallel, sparse matrix CG program. On the IBM SP-2, the performance of our compiled code is comparable to that of handwritten code from the PETSc library at Argonne.

1 Introduction

Sparse matrix computations are ubiquitous in computational science applications because they arise naturally when numerical techniques like the finite element and finite difference methods are used to solve partial differential equations approximately. In most applications, these matrices are very large and very sparse. For example, one of the Gordon Bell prizes in 1994 was won by a team that implemented a chemical vapor deposition (CVD) code in which the matrices had roughly one million rows and five hundred non-zeros per row [SHM+94]. Since there is also a lot of parallelism in these computations, it is natural to use parallel computers in this context.

Unfortunately, parallelizing sparse matrix codes is much more challenging than parallelizing dense matrix codes. One problem is keeping the computation to communication ratio high. For example, in a matrix-vector product $M * v$, the number of floating-point operations that are performed if M is sparse is a small fraction of the corresponding number when M is dense. Unless there are commensurate reductions in the communication overhead, parallel performance will suffer. A second problem is *fill* — during the computation, zero entries in matrices may become non-zero either temporarily or permanently. Since one tries not to store zeros explicitly, fill must be handled by techniques like dynamic storage

[1] This research was supported by an NSF Presidential Young Investigator award CCR-8958543, NSF grant CCR-9008526, ONR grant N00014-93-1-0103, and a grant from Hewlett-Packard Corporation.

allocation. Fill is not an issue for dense matrix programs, of course. Furthermore, from the viewpoint of automatic parallelization, sparse matrix programs are challenging because array subscripts are usually not affine functions of loop indices, so standard tools for dependence analysis, which are based on integer linear programming, break down.

In this paper, we focus on the *conjugate gradient method* (CG), a Krylov space method for solving systems of linear equations of the form $Ax = b$ where A is a symmetric positive-definite matrix, b is a vector of constants and x is the vector of unknowns. From our perspective, this algorithm has three attractions. First, it is very popular; for example, it is the primary solver in Macsyma's PDEase finite element package [Mac94]. Therefore, this problem is of real interest. Second, matrix computations in CG do not cause fill. Finally, there are many handwritten versions of this code available for study. The CG algorithm can be applied regardless of whether A is dense or sparse. *Our objective in this paper is to use restructuring compiler technology to transform a sequential, dense matrix CG program into a parallel, sparse matrix CG program for a distributed memory machine (the IBM SP-2).* We do not assume that the non-zeros have any particular structure like bandedness (this is usually the case for finite element problems in two or more dimensions, for example).

How does our work compare to other efforts in this area? The 'inspector-executor' approach of Saltz and co-workers generates parallel sparse matrix code from sequential sparse matrix programs [WDS$^+$95]. They do not have our problem of 'sparsification' — that is, converting dense matrix code into sparse matrix code. However, they give up the ability to analyze and restructure codes at compile-time. Wijshoff and co-workers have studied the problem of sparsification in the context of sequential and uniform memory access computers [BW93a]. In contrast, we deal with distributed memory parallel machines in which locality of data is the driving concern.

How well do we actually do? The performance of our compiled code on the IBM SP-2 is comparable to that of handwritten code in the PETSc library from Argonne [SG94]. This is encouraging, but much more work is needed to solve the sparse matrix parallelization problem in general.

2 Generating parallel, sparse CG by hand

When writing parallel, sparse matrix code, the programmer is responsible for deciding on how data and work are distributed to processors, and for choosing a proper data structure to represent the sparse matrices. Once these decisions have been made, the programmer must manually specialize the original code for the particular parallelism and sparsity.

A stylized sequential, dense CG is shown in Figure 1. The key operations in this code are matrix-vector multiply, and vector scaling and summation. We have not shown the entire algorithm, just one of several scaling loops and the matrix-vector multiply (MVM). Also, we have written this code as one might write it in FORTRAN. In particular, the MVM loop nest accesses A in a column-major fashion, and the reference to $x[j]$ is invariant to the inner loop.

```
while not converged do
   for i = 1...n do
      x[i] = f(y[i])
   for j = 1...n do
      for i = 1...n do
         y[i] += A[i,j] × x[j]
```

Fig. 1. Sequential, dense CG

To specialize the program for sparsity, the programmer must select a compressed data structure for storing the non-zero elements of A, and then modify the code to iterate over the data structure. A data structure that is appropriate for this sequential code is the Compressed Column Storage layout, shown in Figure 2. The non-zero entries of each column of A are compressed and stored in a single vector, *values*. The row index of each value is stored in the vector *rowindex*, and the index of the first non-zero entry for each column is stored in *colindex*.

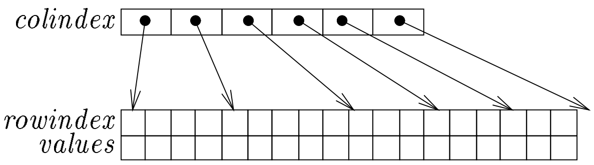

Fig. 2. The Compressed Column Storage of A

The MVM loop nest must be modified to walk over this data structure, as shown below. The i index variable is renamed to ii to emphasize that it is no longer the row index of the element being accessed.

```
while not converged do
   for i = 1...n do
      x[i] = f(y[i])
   for j = 1...n do
      for ii = colindex[j]...colindex[j+1] − 1 do
         y[rowindex[ii]] += values[ii] × x[j]
```

Fig. 3. Sequential, sparse CG

How should we parallelize this code? Most of the work is performed in the MVM loop nest, but effective parallelization of this loop nest is hindered by the fact that the outer loop of the loop nest is not parallel. A better strategy is to change the original dense code by interchanging the i and j loops so that the parallel loop is outermost. To sparsify this code, we observe that in this new code, the matrix A is being traversed by rows, so a different layout is called for. We use Compressed Row Storage (CRS) instead. The resulting code is shown below.

```
while not converged do
   for i = 1...n do
      x[i] = f(y[i])
   for i = 1...n do
      for jj = rowindex[i]...rowindex[i+1] - 1 do
         y[i] += values[jj] × x[colindex[jj]]
```

Fig. 4. Sequential, sparse CG

The meaning of the *colindex* and *rowindex* vectors in CRS is reversed from that in CCS storage: *colindex* holds the column indices of the *values*, and *rowindex* holds the index of the first non-zero entry for each row.

The transformed code is better suited for parallelization. It has been demonstrated in [Bas95] that an effective and scalable means of parallelization is to distribute sets of rows of A and the corresponding elements of x and y to processors. Once this is done, the scaling loop can be executed in parallel, without communication. However, the reference to x in the MVM loop will access non-local values and communication must be placed to handle these references. It is pointless to broadcast all values of x to all processors, since a processor needs $x(j)$ only if $A(i,j)$ is non-zero for some row i mapped to it. If the matrix is not available to the compiler, the exact set of values that need to be communicated cannot be determined statically. Therefore, the programmer will insert code immediately before the "while" loop to examine A and schedule the communication. The resulting code is shown in Figure 5.

```
determine what needs to be communicated
while not converged do
   for i = 1...n_local do
      x[i] = f(y[i])
   communicate x
   for i = 1...n_local do
      for jj = rowindex[i]...rowindex[i+1] - 1 do
         y[i] += values[jj] × x[colindex[jj]]
```

Fig. 5. Parallel, sparse CG

Parallelizing programs by hand is a time consuming and error prone process. Specializing for sparsity makes this difficult task even worse. As a result, many programmers chose not to spend the effort developing parallel, sparse codes, and use prewritten codes, or libraries, instead. An example of such a library is KSP, which is part of PETSc ([SG94]). This library provides "templates" of algorithms, which, when combined with implementations of the sparse data structures, produce complete Krylov space solvers. The library writers have taken care of the details of these solvers and have provided implementations of important sparse layouts, but the user of the library remains responsible for selecting the data structures and distributing the data across the processors.

3 Using a parallelizing compiler

To simplify the process of developing parallel, sparse codes, the programmer could take the sequential, sparse code and try to parallelize it using a parallelizing compiler. The technology for parallelizing sparse codes is not nearly as well-established as that for parallelizing dense code, but enough work has been done to draw some conclusions about this approach.

Some examples (not meant to be exhaustive) of this approach are found in work that has been done on the Rice and Syracuse Fortran D compilers in combination with the PARTI and Chaos runtime systems ([DUSH94], [PSC$^+$93], [WDS$^+$95], [SPM$^+$94]). In this work, the programmer gives the compiler code, similar to that shown in Figure 3 or Figure 4, augmented with alignment and distribution information. Because of the indirect references (e.g., $x[colindex[jj]]$) traditional dependence analysis is impossible on this code, so the programmer is additionally required to mark the parallel loops using the "doall" construct. The responsibility for determining and scheduling efficient communication is left to the runtime system, as the requisite information about references is not present until runtime.

This approach will not generate good code if the original sequential, sparse program is not well-suited for parallelization. For example, if the programmer started from the code that used the CCS layout (Figure 3), the parallelizing compiler would fail to produce good parallel code for two reasons. First, the compiler would not be able to perform loop transformations in order to exploit the parallelism present in the code. Second, and more importantly, the compiler would not be able to transform the code to use the CRS layout, which is more appropriate for the transformed loops.

The inability to perform code or layout transformations is a profound failing, not just because parallel performance suffers, but also because the compiler is unable to address other performance issues, like cache locality. It has been shown that sparse computations can exhibit extremely poor cache/memory locality ([SW90]), and it has been suggested that blocking techniques developed for dense codes might increase the cache locality of sparse codes as well ([KW94]). It is unclear how to apply these techniques automatically to sparse codes like the ones in Figure 3 or Figure 4.

In additional to blocking for cache locality, there are other situations where transforming loops and data layout might be appropriate. One such transformation is the Jagged Diagonal Storage layout ([Saa89]), which increases the number iterations of the inner loop of MVM and might be appropriate for a machine with vector FPU's.

So, we are back to where we started: the programmer is still saddled with the responsibility of making transformations for parallelism, locality, and vectorization. Using a parallelizing compiler has saved the programmer some effort, but not much. Most of the parallelizing technology that the compiler can bring to bear on dense codes goes unused.

4 Using a parallelizing sparse compiler

In an attempt to kill these two birds with one stone, our compiler specializes for sparsity (we call this "sparsification") as well as parallelizes. This makes the programmer's job easier because they no longer have to worry about the details of sparsity or parallelism. Also, by relieving the programmer of the burden of sparsification, the compiler is able to do analysis and code restructuring that is otherwise impossible.

Another way to put this is the following. Our compiler assumes responsibility for parallelization and sparsification, and in return it gets the ability to do dependence analysis and loop transformations. We will see that the issues of parallelization and sparsification are intertwined, and that it makes of sense to delegate both to the compiler, instead of just one.

Our compiler takes sequential, dense code and programmer-supplied annotations as input, and produces parallel, sparse code as output. That is, we start with code that contains "for" loops and dense arrays, and annotations that describe the sparsity of the arrays. In the case of conjugate gradient, the sequential, dense code might look something like the code shown in Figure 1, and there is an annotation telling the compiler that A is the only sparse array.

Our compiler has the following four phases,

1. Dependence Analysis,
2. Loop Transformations,
3. Alignment, and
4. Code Generation.

We do not discuss the first two, because they are conventional parallelizing compiler problems and very well studied. This is one of the big advantages of the parallelizing sparse compiler approach over the parallelizing compiler approach: since we are starting with sequential, dense code, we are able to do dependence analysis and loop transformations in a conventional manner and without intervention from the programmer.

In the case of CG, dependence analysis tells us that the dependence vector for the MVM loop is $[1\ 0]^T$. Loop transformations can be used to move the parallel loop (i.e., the i loop) in the MVM loop nest outermost. The resulting code is,

```
while not converged do
   for i = 1...n do
      x[i] = f(y[i])
   for i = 1...n do
      for j = 1...n do
         y[i] += A[i,j] × x[j]
```

5 Alignment and Distribution

We use the following framework for assigning work and data to processors,

A "template" is a space of virtual processors.
An "alignment function" is a linear (actually, affine works too) function from iterations or array elements to template points. Alignment maps are represented using integer matrices. Computation (loop) alignments are denoted using C, and data (array) alignments using D.
A "distribution function" is a function from template points to physical processors. M is used to denote distributions. Distributions can be non-linear.

The meaning of these is given by these two properties,

- Processor p executes iteration i of a loop iff $M(Ci) = p$.
- Processor p stores element a of an array iff $M(Da) = p$.

Given an array access Fi in a loop i, the access is "aligned" if $Ci = DFi$, or more simply if $C = DF$. In this case, the data being accessed by the processor executing the iteration is local. That is, no communication is required to resolve the reference. A reference that is not aligned is called "misaligned". A reference that is misaligned *may* require communication. Communication is only required iff $M(Ci) \neq M(DFi)$.

The alignment equations for the transformed program are the following.

$$\begin{array}{cc} C_i = D_x[1] & C_i = D_y[1] \\ C_{i,j} = D_x[0\ 1] & C_{i,j} = D_y[1\ 0] \\ \multicolumn{2}{c}{C_{i,j} = D_A I} \end{array} \tag{1}$$

5.1 Computing an alignment

We wish to relieve the programmer of the burden of deciding alignment by computing it automatically. In particular, we seek to find alignment functions that,

- maximize parallelism, and
- minimize the communication caused by the misaligned references.

This is a constraint problem, where there is a constraint of the form, $C = DF$, for each array reference for which alignment is desired. Such systems always have a solution because we can map all data and computations to a single processor, thereby satisfying all constraints. However, this trivial solution exhibits no parallelism. We are interested only in solutions that exhibit some degree of parallelism, or non-trivial solutions. We have developed an algorithm for solving such systems of constraints and producing a solution with the largest degree of parallelism[BKK+94]. If a non-trivial solution is found, we are done. Otherwise, we have an overconstrained system and we must drop one or more constraints to obtain a non-trivial solution. Each removed constraint represents a set of references that will be misaligned. Therefore, eliminating constraints will result in communication in the final program. The problem of determining which constraints to remove to produce the best alignment solution is NP-hard. In practice, heuristics are used to select the constraints to remove [CGS93, Fea92, AL93].

5.2 Alignments for CG in Handwritten Code

There are two alignments that are used for conjugate gradient. The one with the highest degree of parallelism is an alignment with a 2D template space. In other words, the first two constraints on $C_{i,j}$ in Equations 1 are eliminated, and the remaining equation for $C_{i,j}$ is satisfied by setting $C_{i,j} = D_A = I$, the identity matrix in two dimensions. This is the alignment that is required by the NAS CG benchmark ([BBLS91]). However, this alignment is most appropriate when either A is dense or when the non-zeros are uniformly distributed in A. These assumptions are not met in sparse matrices arising in practice from finite element problems and the like, and a 1D alignment turns out to be superior [Bas95]. To obtain this alignment, the first constraint on $C_{i,j}$ is dropped, and the remaining equations are solved to obtain the following alignment:

$$D_x = [1] \; D_y = [1] \; D_A = [1 \; 0]$$
$$C_i = [1] \; C_{i,j} = [1 \; 0]$$

That is, elements of x and y and rows of A are mapped across the template, and iterations of the first loop nest and iterations of the i loop in the second nest are assigned across the template. The only misaligned reference is $x[j]$ in the second loop nest.

5.3 Our alignment heuristic

A completely satisfactory heuristic is not yet available even for dense matrix computations, let alone for sparse matrix computations. However, a simple heuristic is to give preference to 2D arrays over 1D vectors, and to prefer aligned writes over aligned reads. This yields the 1D alignment discussed above. Although this is fine for CG, it does not find the most appropriate alignment for other codes like GMRES. Alignment heuristics require much more work.

5.4 Distribution

After computing alignments, we have,

- a set of template points that must be assigned to physical processors, and
- a set of misaligned references.

We wish to compute a distribution function mapping template points to physical processors that

- minimizes the amount of communication, and
- minimizes load imbalance.

In dense codes, block/cyclic distributions are used and do a good job of minimizing these two. But, in unstructured sparse problems, these regular distributions do not do as good a job as irregular distributions. In unstructured sparse problems, the best distributions are usually found by explicitly constructing a graph representing the communication, and using node partitioning techniques like recursive spectral bisection [MTTV93, LRT79, PSL90] to compute the distribution function.

However, the irregularity of the resulting distribution creates some problems for the compiler. The first is that, because the resulting distribution is irregular, it is unlikely to have a nice linear closed form, like the block/cyclic distributions do. As a result, the distribution function must be stored explicitly. Moreover, this function is known only at runtime. Therefore, either the programmer must provide the distribution function explicitly, or the compiler must generate code to compute it at runtime. We have not implemented the later approach, and, at present, we require that the programmer provide the distribution function explicitly.

The second problem with irregular distribution functions is the generation of efficient code. We address this problem in the next section.

6 Code Generation

At this point, we have performed dependence analysis and loop transformations, and have determined the alignment of computation and data. It remains for us to

- specialize the computation and data for parallelism,
- specialize the computation and data for sparsity, and
- insert communication to preserve data dependences to misaligned references.

In this section we will develop a general technique for generating code that is specialized in these ways. We will also point out improvements that can be made. We will discuss regular and irregular distributions and structured and unstructured sparsity patterns.

6.1 Naively specializing for sparsity and parallelism

We first discuss an extremely simple method for specializing the code and data for sparsity and parallelism. This method is not actually used in the compiler, but it serves as a platform upon which the optimizations and improvements used by our compiler are built.

The Runtime System We assume the existence of a runtime system (abbreviated to RTS) whose primary responsibility is handling data storage and communication. In particular, the runtime system will ensure the following.

- Zero array elements are not explicitly stored (more precisely, zero elements, as specified by the array's sparsity, are not stored).
- Only those array elements that have been assigned to a processor by alignment and distribution are stored on that processor.
- References to elements that do not reside on the local processor result in communication with the processors on which the values resides.

We view the runtime system as managing a collection of objects. Each array in the program will be implemented using a set of objects, one object residing on each processor. Each object will be responsible for storing only the non-zero elements of the array that are assigned by alignment and distribution to the processor on which it resides. The methods implemented by these objects are,

`double &global_ref(integer` $\imath$`)` Returns a reference to the $\imath$th element of the array.

`boolean nonzero?(integer` $\imath$`)` Returns true iff the $\imath$th element of the array is non-zero.

When a reference is made to an array element that does not reside on the local processor, the local object must determine in which remote object the reference does reside. The local object must then communicate with the remote object in order to resolve the reference. The overhead of this type of communication will be high, because short messages are used, and because the remote processor will be interrupted during other computation to handle the communication.

Specializing the code We wish to transform code such that

- exactly those iterations that have been assigned to a processor by alignment and distribution, are executed by that processor, and
- exactly those iterations that contribute to the final result, in the presence of sparsity, are executed, and
- zero elements of sparse arrays are not directly accessed.

Parallelism The first requirement is most easily accomplished by placing guards, or conditionals, into the code in order to govern which iterations are executed. For instance, if a particular loop has an alignment of C and distribution of M, then we can ensure that iterations of the loop are executed on their assigned processor by placing the appropriate guard around the body of the loop.

for $\imath = \ldots$
    if $M(C\imath) =$ `#me` then
        ... body ...

This method of using guards to assign work to processors is called "runtime resolution" ([RP89]).

Sparsity We can use a similar method to specialize the code for sparsity. The technique of introducing sparsity guards and simplifying the code using algebraic properties has been developed by Bik and Wijshoff ([BW93b]). In the MVM loop nest, we get the following code.

for $i = 1 \ldots n$ do
    for $i = 1 \ldots n$ do
        $y[i] += A[i,j] \times x[j]$

Since the increment to $y[i]$ is non-zero only when $A[i,j]$ is non-zero, we can introduce a guard that will perform the operation only when this is so,

for $i = 1 \ldots n$ do
    for $j = 1 \ldots n$ do
        if $A[i,j] \neq 0$ then
            $y[i] += A[i,j] \times x[j]$

The data references In order to interface the resulting code with the runtime system, each array reference, $A[i,j]$ should be changed to a method invocation. For instance, the code,

if $A[i,j] \neq 0$ then
    $\ldots = \ldots A[i,j] \ldots$

becomes,

if A`.nonzero?(`i`,`j`)` then
    $\ldots = \ldots$ A`.global_ref(`i`,`j`)` ...

Distinguishing between local and non-local accesses Recall that a reference is aligned when $C = DF$, where F is the access function of the reference, and that a reference is misaligned when it is not aligned. Since aligned references always access local data, we can have a method just for accessing local data,

`double &ref(integer` $\imath$`)` Returns a reference to the $\imath$th element of the array. The element is required to be local to the processor.

An invocation of `ref` is cheaper than an invocation of `global_ref` because the check to see if the reference is local or not is not required.

Taking advantage of regular distribution or structured sparsity Much has been done to address the problem of specializing code and data for regular distribution (e.g., block or cyclic). These sorts of distributions can be expressed in terms on integer linear inequalities, which can be folding directly into the bounds of loops using such methods as Fourier-Motzkin elimination. This approach is referred to as "compile-time" resolution.

Since some forms of structured sparsity can also be expressed as integer linear inequalities (e.g., "$A[i, j] \neq 0$ where A is tridiagonal" can be expressed as $-1 \leq i - j \leq 1$), these techniques could also be used to fold the sparsity guards into the bounds of loops. When the sparsities predicates can be expressed as integer linear inequalities, sophisticated dependence analysis (e.g., the Omega test ([Pug91])) may be able to exactly determine the dependences in the program, and because the alignment and distribution functions can be expresses as integer linear inequalities, it is possible to express the communication requirements in a closed form. It is then possible to insert efficient communication code into the program.

We have investigated these ideas briefly, and they appear promising, but, since we are more interested in handling irregular distributions and unstructures sparsity, we have not pursued them.

6.2 Using invariance to handle unstructured sparsity and distribution

The problem with irregular distributions and unstructured sparsities is that they make it is impossible to fold guards into the bounds of the loops. This is not only because these distributions and sparsities cannot be expressing in a nice linear manner, but also because they are often not known until runtime. As was pointed out earlier in this paper, these sorts of programs are very common in scientific computing, and it is these programs that we are most interested in generating code for.

However, as is the case in compiling dense matrix programs, it is useful to lift communication out of loops and block it wherever possible. This is possible in many scientific codes because the sparsity of the arrays and the distribution of data is fixed for large portions of the program. In a sense, the guards being evaluated are "invariant", and can be lifted so that the cost of evaluating them can be amortized over many executions of the loop. This is the key idea behind the inspector/executor approach of Saltz and co-workers. The idea is to determine the non-local references that a loop will make, to use this information to efficiently schedule communication, and to use the resulting communication schedule for many executions of the original loop nest.

To do this, a copy of a loop containing a misaligned reference is made. Then, the body of the clone loop is modified so that the loop no longer reads or writes to memory, or produces any other sort of side-effect, but the computation of array references is still performed. Procedure calls are inserted so that these references are passed to the runtime system. Thus, when this clone loop is executed, the runtime system is given the entire stream of references that the original loop would have made. This clone loop is called the "inspector".

With the stream of references, it is possible for the runtime system to determine exactly which non-local locations are referenced. The runtime system can group these references by their owner processor, and schedule a few large messages to resolve the references. This message aggregation results in less communication overhead than the many small messages that were previously required. Also, communication schedules can be shared with the data's owner processors. Then, the owner can send the data without the processor making the reference having to request it.

The inspector loop should be placed at the earliest point in the program where all of its controlling predicates are valid. That is, if the inspector loop contains a sparsity guard on, say, A, then the inspector cannot be placed any earlier than the last write than might change the sparsity of A. If the programmer indicates that A's sparsity cannot change, then perhaps the inspector can be placed very early in the program.

Next, dependence analysis can be used to determine the earliest point in the program at which the values to be communicates are available. At this point, a call to the runtime system can be placed to cause the communication to occur. Since this communication is explicitly present in the resulting code, processors don't have to worry about handling unexpected, asynchronous communication at runtime. Also, since the values are sent earlier than they are required, there is a greater chance that they will be available when they are needed. This reduces the chance that a processor will stall waiting for data.

To make this concrete, suppose that the original code is,

```
while ... do
    for i = ...
        x.ref(i) = ...
    for i = ...
        for j = ...
            if A.nonzero?(i,j) then
                y.ref(i) += A.ref(i,j) × x.global_ref(j)
```

The only misaligned reference and, therefore, the only reference that might required communication is, $x[j]$, in the second "for" loop nest. After creating and placing the inspector loop and placing the communication, we have,

```
for i = ...
    for j = ...
        if A.nonzero?(i,j) then
            pass the reference "x[j]" to the RTS.
RTS can compute the communication schedule
        and allocate non-local buffers for "x[j]".
while ... do
    for i = ...
```

```
    x.ref(i) = ...
  Communicate "x[j]".
  for i = ...
    for j = ...
      if A.nonzero?(i,j) then
        y.ref(i) += A.ref(i,j) × x.global_ref(j)
```

Using inspector loops to precompute sparsity guards and local object addresses There are still severe performance problems remaining. Consider, for instance, a program, like the one above, that accesses an aligned array with arbitrary sparsity, for which the distribution function is only known at runtime. The evaluation of the sparsity predicate A.`nonzero?`(i,j) is expensive because it requires performing a search within A's local object. Furthermore, the overhead of enumerating all possible values of $< i, j >$ in order to test the sparsity predicate is also very high.

But all is not lost. We have shown that it is possible to preschedule the communication; it should be clear that we can precompute and cache the sparsity predicate, as well as the location of references within the local object. As with communication, once these have been computed, it is often possible to use these cached results many times.

For instance, in our previous example, the i,j loop nest does not have the sparsity guard folded into the bounds of the loops. As a result, the number of i and j values enumerated by the "for" loops are much greater than the number of times the body of the loop is actually executed. Since we already have an inspector that performs the sparsity test in order to generate the stream of references to x, it should be fairly easy to modify it to generate the stream of valid iterations as well! We will store this stream in a vector S. Then, the original loop nest can be modified to walk S, and thus enumerate the set of valid iterations.

```
n = 0;  S = φ
for i = ...
  for j = ...
    if A.nonzero?(i,j) then
      pass the reference "x[j]" to the RTS.
      S[n] =< i,j >;  n++
RTS can compute the communication schedule
    and allocate non-local buffers for "x[j]".
while ... do
  for i = ...
    x.ref(i) = ...
  Communicate "x[j]".
  for ii = 0 ... n − 1
    < i,j >= S[ii]
    y.ref(i) += A.ref(i,j) × x.global_ref(j)
```

One remaining problem with this code is the searches of the local object required to resolve the references y.ref(i), A.ref(i,j), and x.global_ref(j). Not surprisingly, these can be folded into the inspector as well. In the inspector, we will compute the addresses of the references, and save them as pointers in the vectors $\hat{y}$, $\hat{A}$, and $\hat{x}_j$, respectively. A second inspector in required to precompute the x.ref(i) reference in the i "for" loop nest, and save the addresses into $\hat{x}_i$.

$\hat{x}_i = \phi$
for $i = \ldots$
    $\hat{x}_i[i] = \&x$.ref(i)
$n_2 = 0;\ S_2 = \hat{y} = \hat{A} = \hat{x}_j = \phi$
for $i = \ldots$
    for $j = \ldots$
        if A.nonzero?(i,j) then
            pass the reference "$x[j]$" to the RTS.
            $S_2[n_2] = < i, j >$
            $\hat{y}[n_2] = \&y$.ref(i)
            $\hat{A}[n_2] = \&A$.ref(i,j)
            $\hat{x}_j[n_2] = \&x$.global_ref(j)
            n_2 ++
RTS can compute the communication schedule
        and allocate non-local buffers for "$x[j]$".
while ... do
    for $i = \ldots$
        $^*\hat{x}_i[i] = \ldots$
    Communicate "$x[j]$".
    for $ii = 0 \ldots n_2 - 1$
        $< i, j > = S_2[ii]$
        $^*\hat{y}[ii] \mathrel{+}= {}^*\hat{A}[ii] \times {}^*\hat{x}_j[ii]$

To summarize, the precomputed information about the set of valid iterations of a particular loop nest will consist of a set of vectors, which we call the "iteration storage". The length of these vectors will be the number of valid iterations of the loop. Some vectors will be used to store the loop indices of the valid iterations. Some will be used to store pointers to the array elements accessed by each iteration. The code of the original loop nest if modified to walk these vectors and to use the information stored within them.

Non-local buffers and local storage We have said that the runtime system, with information from the inspector loops, can efficiently schedule communication. In order to do this, it will be necessary for the runtime system to allocate buffers into which non-local values are received, and from which non-local values are sent. The runtime system must now be able to determine whether a reference resides in the local object or one of the non-local buffers

Fortunately, this determination can be made once by the runtime system during the inspector loop. In particular, in the previous example, the reference to $\hat{x}_j[n_2] = \&x$.global_ref(j) in the second loop nest's inspector should be interpreted as a call to the runtime system that returns a pointer to the location in either the local object or a non-local buffer where the reference x.global_ref(j) will eventually be found.

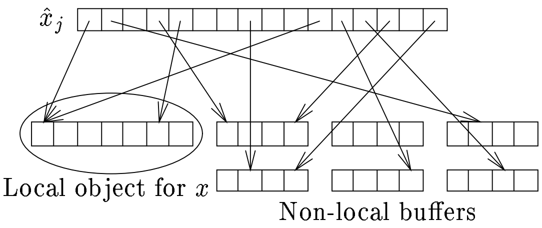

Fig. 6. Iteration storage for the reference $x[j]$

6.3 Reducing the number of inspectors required

As we have presented the technique above, every loop nest in the original code will have its own inspector. But consider the following code.

```
for i = 0...n do
   for j = 0...n do
      if A.nonzero?(i,j) then
         ...A.ref(i,j)...
for i = 0...n do
   for j = 0...n do
      if A.nonzero?(i,j) then
         ...A.ref(i,j)...
```

In this case, it is clear that only one inspector is required.

```
n = 0;  S = φ
for i = 0...n do
   for j = 0...n do
      if A.nonzero?(i,j) then
         S[n] =< i,j >;  n++
for ii = 0...n − 1 do
   < i,j >= S[ii]
   ...A.ref(i,j)...
for ii = 0...n − 1 do
   < i,j >= S[ii]
   ...A.ref(i,j)...
```

The most obvious case where it is possible to reduce the number of inspectors is when two loops have the same bounds, direction, alignment, sparsity guards, etc. What can be done when these conditions are not satisfied? This is an area where more work is needed.

6.4 Handling imperfectly nested loops

Suppose that we have an imperfectly loop nest with sparsity guards, like the following,

```
while ... do
   for i = ... do
      for j = ... do
         if A.nonzero?(i,j) then
            ...
      for k = ... do
         if B.nonzero?(i,k) then
            ...
```

What should the inspectors for this loop nest look like? One possibility is to have one inspector for i,j loop and one for the i,k loop. Then, the original loops would be modified to, first execute all valid iterations of the i,j loop, and then all iterations of the i,k loop.

If there are no dependences between the j and k loops, then this is not a problem. In fact, this is equivalent to distributing the i loop over the j and k loops, resulting in two perfectly nested loops, i,j and i,k.

Since, in general, loop distribution is not possible, we need some method of enumerating the valid $< i,j >$ and $< i,k >$ iterations in their original order. That is, for any fixed value of $\tilde{\imath}$, first all of the valid $< \tilde{\imath},j >$ iterations are enumerated, and then all of the valid $< \tilde{\imath},k >$ iterations. If we do compute two different sets of iteration storage, one for the i,j loop and one for the i,k loop, then we need a way of finding iterations that correspond to a particular value of $\tilde{\imath}$ within each set of iteration storage.

This can be done by recording, for each iteration of $\tilde{\imath}$, the location of the first and last valid iterations of j and k. For instance, we could record in $start_j[\tilde{\imath}]$ the index of the first valid iteration of $< \tilde{\imath},j >$ in S_j, similarly with $start_k[\tilde{\imath}]$ for S_k. The first and last valid iterations of the j loop for iteration $\tilde{\imath}$ of the i loop can be found at index $start_j[\tilde{\imath}]$ and index $start_j[\tilde{\imath}+1]-1$, respectively of the iteration storage. Here is what the resulting inspectors and code might look like.

```
n_j = n_k = 0;  S_j = S_k = φ
start_j[0] = start_k[0] = 0
for i = ... do
   for j = ... do
      if A.nonzero?(i,j) then
         S_j[n_j] = j; n_j ++
```

for $k = \ldots$ do
    if B.`nonzero?`(i,k) then
        $S_k[n_k] = j; n_k$ ++
    $start_j[i+1] = n_j$
    $start_k[i+1] = n_k$
while ... do
    for $i = \ldots$ do
        for $jj = start_j[i] \ldots start_j[i+1] - 1$ do
            $j = S_j[jj]$
            ...
        for $kk = start_k[i] \ldots start_k[i+1] - 1$ do
            $k = S_k[kk]$
            ...

Notice that one more index is stored in $start_j$ and $start_k$ than there are iterations of loop i. The last values mark the end of two sets of iteration storage.

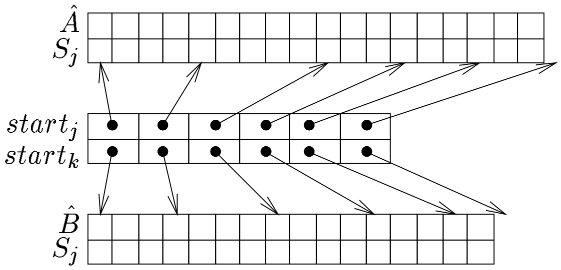

Fig. 7. Iteration storage for imperfect loops

A similar approach can be used to handle other types of imperfectly nested loops. Suppose that we have a loop nest in which one of the guards in invariant to the inner loop, as follows.

while ... do
    for $i = \ldots$ do
        if $M(i) =$ `#me` then
            for $j = \ldots$ do
                if A.`nonzero?`(i,j) then
                    ...

This gets transformed to the following code.

$n_i = n_j = 0;\ S_i = S_j = \phi$
$start_j[0] = 0$
for $i = 0 \ldots n-1$ do

if $M(i) = \texttt{\#me}$ then
  $S_i[n_i] = i$
  for $j = 0 \ldots n-1$ do
    if $A.\texttt{nonzero?}(i,j)$ then
      $S_j[n_j] = j;\ n_j\,{+}{+}$
  $n_i\,{+}{+};\ start_j[n_i] = n_j$
while … do
  for $ii = 0 \ldots n_i - 1$ do
    $i = S_i[ii]$
    for $jj = start_j[ii] \ldots start_j[ii+1] - 1$ do
      $j = S_j[jj]$
      …

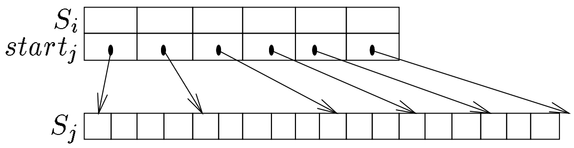

Fig. 8. Iteration storage for invariant guards

6.5 Accessing local storage efficiently

We have already said that aligned references do not require checks to see whether or not they are accessing non-local data. However, we still require that an inspector compute their addresses within a local object, that these addresses be saved for future use, and that the final reference to the location be made by dereferencing these pointers. All of these, but particularly the dereferences which occur within inner loops, incur substantial overhead. It is often possible to improve upon these accesses to local storage.

Suppose for instance, that we have the following code,

while … do
  for $i = \ldots$ do
    $\ldots = \ldots y.\texttt{ref}(i) \ldots$
  for $i = \ldots$ do
    $\ldots = \ldots y.\texttt{ref}(i) \ldots$
  for $i = \ldots$ do
    $\ldots = \ldots y.\texttt{ref}(i) \ldots$

Assuming that the bounds, guards, etc. of each of the i "for" loops is the same, then we insert one inspector and get the following code,

$n = 0;\ \ S = \phi;\ \ \hat{y} = \phi$
for $i = \ldots$ do
  $\hat{y}[n] = \&y$.ref(i); n++
while ... do
  for $ii = 0 \ldots n-1$ do
    $\ldots = \ldots{}^{*}\hat{y}[ii] \ldots$
  for $ii = 0 \ldots n-1$ do
    $\ldots = \ldots{}^{*}\hat{y}[ii] \ldots$
  for $ii = 0 \ldots n-1$ do
    $\ldots = \ldots{}^{*}\hat{y}[ii] \ldots$

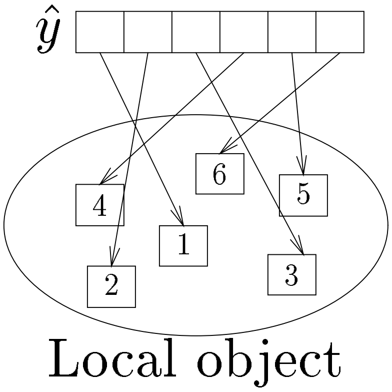

Fig. 9. $\hat{y}$ points into the local object

If

- all of the references to y in the program, or at least this region of the program, are precomputed in the same inspector loop,
- all the references are aligned and have the same access function, and
- the access function of the original reference is invertible (i.e., every iteration of the original i "for" loops access a different location in y),

then the map $\hat{y}$ is a one-to-one and onto map from iterations to all locations of y that are accessed by the local processor. In this case, $\hat{y}$ need not be the addresses of the locations; it can be the actual locations. That is, instead of being pointers to the values of y, $\hat{y}$ can be the values of y.

$n = 0;\ \ S = \phi;\ \ \hat{y} = \phi$
for $i = \ldots$ do
  $\hat{y}[n] = y$.ref(i); n++
while ... do

for $ii = 0 \ldots n-1$ do
  $\ldots = \ldots \hat{y}[ii] \ldots$
for $ii = 0 \ldots n-1$ do
  $\ldots = \ldots \hat{y}[ii] \ldots$
for $ii = 0 \ldots n-1$ do
  $\ldots = \ldots \hat{y}[ii] \ldots$

Since there are no pointers into it, the local object that was used to store y can be eliminated.

$\hat{y}$	1	2	3	4	5	6

Fig. 10. $\hat{y}$ holds the values

Actually, it is possible to relax these restrictions somewhat: we require that all *aligned* references have the same invertible access function, and allow other misaligned references. Since the runtime system already generates addresses for misaligned references, we can simply require that these addresses point into this new iteration storage, instead of into the local object.

For instance, suppose that the previous example had a misaligned reference to y, y`.global_ref(`j`)`,

while $\ldots$ do
  for $i = \ldots$ do
    $\ldots = \ldots y$`.ref(`i`)` $\ldots$
  for $i = \ldots$ do
    $\ldots = \ldots y$`.ref(`i`)` $\ldots$
  for $j = \ldots$ do
    $\ldots = \ldots y$`.global_ref(`j`)` $\ldots$

In this case, it is possible to store the values of y`.ref(`i`)` directly in $\hat{y}_i[ii]$. A second vector, $\hat{y}_j$ will hold the addresses of the references y`.global_ref(`j`)`, which will point into either $\hat{y}_i[ii]$ or the non-local buffers allocated by the runtime system for communication.

$n_i = 0$; $S_i = \phi$; $\hat{y}_i = \phi$
for $i = \ldots$ do
  $\hat{y}_i[n_i] = y$`.ref(`i`)`; n_i ++
$n_j = 0$; $S_j = \phi$; $\hat{y}_j = \phi$
for $j = \ldots$ do
  $\hat{y}_j[n_j] = \&y$`.global_ref(`j`)`; n_j ++
while $\ldots$ do
  for $ii = 0 \ldots n_i - 1$ do

$\ldots = \ldots \hat{y}_i[ii] \ldots$
for $ii = 0 \ldots n_i - 1$ do
  $\ldots = \ldots \hat{y}_i[ii] \ldots$
for $jj = 0 \ldots n_j - 1$ do
  $\ldots = \ldots {}^*\hat{y}_j[jj] \ldots$

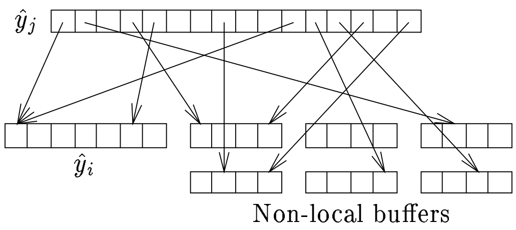

Fig. 11. Iteration storage for the reference y.`global_ref`(j)

6.6 Conjugate gradient

After inserting the guards for sparsity and parallelism and replacing array references with method invocations, we have,

while not converged do
  for $i = 0 \ldots n - 1$ do
    if $M(i) =$ `#me` then
      x.`ref`(i) $= f(y$.`ref`$(i))$
  for $i = 0 \ldots n - 1$ do
    if $M(i) =$ `#me` then
      for $j = 0 \ldots n - 1$ do
        if A.`nonzero?`(i,j) then
          y.`ref`(i) $+=$ A.`ref`(i,j) $\times$ x.`global_ref`(j)

We insert a inspector before the "while" loop to schedule the communication and compute the iteration information. Since all of the aligned references to x, y, and A have the same invertible access functions, the values, and not just the addresses, of these locations can be stored in $\hat{x}_i$, $\hat{y}$, and $\hat{A}$.

$n_i = n_j = 0;\; S_i = S_j = \phi$
$\hat{A} = \hat{y} = \hat{x}_i = \hat{x}_j = \phi$
$start_j[0] = 0$
for $i = 0 \ldots n - 1$ do
  if $M(i) =$ `#me` then
    $S_i[n_i] = i$

```
        x̂_i[n_i] = x.ref(i)
        ŷ_i[n_i] = y.ref(i)
        for j = 0 ... n − 1 do
            if A.nonzero?(i,j) then
                S_j[n_j] = j
                Â[n_j] = A.ref(i,j)
                pass the reference "x[j]" to the RTS.
                x̂_j[n_j] = &x.global_ref(j)
                n_j ++
        n_i ++; start_j[n_i] = n_j
RTS can compute the communication schedule
        and allocate non-local buffers for "x[j]".
while not converged do
    for ii = 0 ... n_i − 1 do
        i = S_i[ii]
        x̂[ii] = f(ŷ[ii])
    Communicate "x[j]".
    for ii = 0 ... n_i − 1 do
        i = S_i[ii]
        for jj = start_j[ii] ... start_j[ii + 1] − 1 do
            j = S_j[jj]
            ŷ[ii] += Â[jj] ×* x̂[jj]
```

6.7 "Peephole" optimizations

There are two additional optimizations that we can apply to this code. First, notice that the inspector loop enumerates all iterations of i in order to test which ones are aligned with template points assigned to the local processor.

```
for i = 0 ... n − 1 do
    if M(i) = #me then
        ...
```

A better approach, in this case, would be to enumerate all of the template points assigned to the local processor, and test whether the corresponding iterations falls within the bounds of the original loop nest.

```
for t ∈ M⁻1(#me) do
    i = t
    if 0 ≤ i ≤ n − 1 then
        ...
```

This results in lower loop overhead in the inspector, and is safe because dependence analysis tells us that the iterations of the original loop can be executed in any order.

There is a similar optimization that can be used to reduce the cost of finding the non-zeros in each row of A.

```
for t ∈ M⁻1(#me) do
   i = t
   if 0 ≤ i ≤ n − 1 then
      for j = 0...n − 1 do
         if A.nonzero?(i,j) then
            ...
```

Instead of testing each entry in the row $A[i,*]$, perhaps the local object could provide the inspector with exactly the non-zero entries in the row. Then the inspector would iterate over this set and tests to see if each element corresponds to an access that occurred within the original loop.

```
for t ∈ M⁻1(#me) do
   i = t
   if 0 ≤ i ≤ n − 1 then
      for a₂ ∈ non-zeros indices of A[i,*] do
         j = a₂
         if 0 ≤ j ≤ n − 1 then
            ...
```

Again, this results in lower loop overhead in the inspector, and is safe because dependence analysis tells us that the iterations of the original loop can be executed in any order.

6.8 The current implementation

Although our current implementation of the code generation techniques is able to compile conjugate gradient into the form described above, it does not implement all of the techniques discussed in this paper. In particular, our implementation does not attempt to generate the general inspectors and then apply optimizations. Instead, it checks to see that certain properties of the code are true, and then generates the optimized inspectors directly. This is not completely satisfying, but it is sufficient to demonstrate that this method of compiling code for parallelism and sparsity is sound and produces good results.

7 Performance

Table 1 shows the performance of the sparse CG code generated by our compiler, compared with the spare CG code found in the PETSc version 2.0 beta 2. Times include the execution of an inspector and 16 iterations of the *while* loop. The codes were run on 1 through 64 nodes of the Cornell Theory Center SP-2. Both codes used the vendor supplied MPI implementation as the communication layer. The matrix A was determined from a 64,000 node 3D mesh. In other words, A had 64,000 rows and columns, and an average of 27 non-zeros per row. A 3D mesh was used so that the template points could be optimally partitioned, thus eliminating the effects of heuristic partitioning techniques. The performance of

Procs.	Compiler	PETSc
1	22.11	—
2	9.95	7.48
4	4.86	4.16
8	2.40	2.20
16	1.26	1.42
32	0.70	1.01
64	0.44	0.47

Table 1. Wall clock times for CG, in seconds

the compiled code is respectable, and can probably be improved by overlapping communication and computation more aggressively. These issues require more work.

8 Related work

Our ideas on parallel, sparse compiling have been influenced most heavily by the work by Bik, et al. on the Sparse Compiler and Saltz, et al. on PARTI/Chaos.

The Sparse Compiler is introduced in [BW93b]. The input source code is sequential, dense code, as in our case, and the output is sequential, sparse code. The Sparse Compiler has a fairly general annotation language for describing the sparsity of the data structures, and the compiler's job is to introduce sparsity guards into the code, to optimize these guards, and to select and generate an appropriate compressed data layout. In [BKW94], loop transformations are done to try to align array references so that access to the selected compressed data layout is more efficient. Some work has been done to parallelize the sparse code ([BW93a]), but this was done within the context of a UMA multiprocessor, so the parallelization issues are different than those that we considered. More specifically, their problem is to insert and optimize explicit synchronization operations, while our problem is locality.

The idea of using an inspector to pass information to the runtime system, was developed by Saltz, et al.([DPSM91]) in the context of handling communication in the presence of irregularity in Fortran codes. The big difference with our approach, as was observed earlier, is that they assume that the programmer sparsifies the code before compiling. As was noted in Section 2, this approach is somewhat more fragile than ours since it generates good code only if the programmer has written the loops in the right order, chosen the right storage formats *etc.*

9 Conclusions

We have shown that it is possible and advantageous to automate both the processes of sparsification and parallelization for NUMA parallel machines. We have developed a compiler that is able to produce, for the conjugate gradient method, parallel code that is comparable to hand-written code in both function and performance. This is encouraging but much more work remains to be done to solve the problem in general.

References

[AL93] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 112 – 125, June 1993.

[Bas95] Achim Basermann. Parallel sparse matrix computations in iterative solvers on distributed memory machines. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, February 1995. SIAM Press.

[BBLS91] D.H. Bailey, J.T. Barton, T.A. Lasinski, and H.D. Simon. The NAS parallel benchmarks. Technical Report RNR-91-02, NASA Ames Research Center, Moffett Field, CA, January 1991.

[BKK+94] David Bau, Induprakas Kodukula, Vladimir Kotlyar, Keshav Pingali, and Paul Stodghil. Solving alignment using simple linear algebra. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing. Seventh International Workshop.*, LNCS. LNCS, Springer-Verlag, 1994.

[BKW94] Aart J.C. Bik, Peter M.W. Knijnenburg, and Harry A.G. Wijshoff. Reshaping access patterns for generating sparse codes. In *Proceedings of the Seventh Annual Workshop on Languages and Compiler for Parallel Computing*, Ithaca, New York, August 8–10, 1994. Springer-Verlag. LNCS #892.

[BW93a] Aart Bik and Harry Wijshoff. Advanced compiler optimizations for sparse computations. In *Proceedings of Supercomputing 93*, pages 430–439, November 1993.

[BW93b] Aart Bik and Harry Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the International Conference on Supercomputing*. ACM Press, August 1993.

[CGS93] Siddartha Chatterjee, John Gilbert, and Robert Schreiber. The alignment-distribution graph. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing. Sixth International Workshop.*, number 768 in LNCS. Springer-Verlag, 1993.

[DPSM91] Raja Das, Ravi Ponnusamy, Joel Saltz, and Dimitri Mavriplis. Distributed memory compiler methods for irregular problems – data copy reuse and runtime partitioning. ICASE Report 91–73, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Virginia, September 1991.

[DUSH94] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994. Also available as University of Maryland Technical Report CS-TR-3163 and UMIACS-TR-93-109.

[Fea92] Paul Feautrier. Toward automatic distribution. Technical Report 92.95, IBP/MASI, December 1992.

[KW94] P.M.W. Knijnenburg and H.A.G. Wijshoff. On improving data locality in sparse matrix computations. Technical Report 94-15, Department of Computer Science, Leiden University, 1994.

[LRT79] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16:346–358, 1979.

[Mac94] PDEase Programmer's Manual. Macsyma Inc., 20 Academy Street, Arlington, MA 02174., 1994.

[MTTV93] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Automatic mesh partitioning. In A. George, J. Gilbert, and J. Liu, editors, *Graph Theory and Sparse Matrix Computation*, volume 56 of *IMA Volumes in Mathematics and its Applications*. Springer-Verlag, Berlin, 1993.

[PSC+93] Ravi Ponnusamy, Joel Saltz, Alok Choudhary, Yuan-Shin Hwang, and Geoffrey Fox. Runtime support and compilation methods for user-specified data distributions. Technical Report CS-TR-3194 and UMIACS-TR-93-135, University of Maryland, November 1993. To appear in IEEE Transactions on Parallel and Distributed Systems.

[PSL90] A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis and Applications*, 11:430–452, 1990.

[Pug91] William Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings Supercomputing '91*, pages 4–13, Albequerque, New Mexico, November 18–22, 1991.

[RP89] Anne Rogers and Keshav Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 69–80, Portland, Oregon, June 21–23, 1989. Published as ACM SIGPLAN Notices 24(7).

[Saa89] Youcef Saad. Kyrlov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1232, November 1989.

[SG94] B. Smith and W. Gropp. Portable, parallel, reusable krylov space codes. In *Colorado Conference on Iterative Methods*, Colorado, April 1994.

[SHM+94] John N. Shadid, Scott A. Hutchinson, Harry K. Moffat, Gary L. Hennigan, Bruce Hendrickson, and Robert W. Leland. A 65+ gflops/s unstructured finite element simulation of chemically reacting flows on the intel paragon. In *Supercomputing '94*, pages 673–679, Washington, D.C., November 14–18, 1994.

[SPM+94] Shamik D. Sharma, Ravi Ponnusamy, Bongki Moon, Yuan-Shin Hwang, Raja Das, and Joel Saltz. Run-time and compile-time support for adaptive irregular problems. In *Supercomputing '94*, Washington, D.C., November 14–18, 1994.

[SW90] Youcef Saad and Harry A.G. Wijshoff. Spark: A benchmark package for sparse computations. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 239–253, Somewhere, Sometime, 1990.

[WDS+95] Janet Wu, Raja Das, Joel Saltz, Harry Berryman, and Seema Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44(6), 1995.