Interprocedural Data Flow Based Optimizations for Distributed Memory Compilation

Gagan Agrawal and Joel Saltz
UMIACS and Department of Computer Science
University of Maryland
College Park, MD 20742
(301)-405-2756
{gagan, saltz}@cs.umd.edu

Abstract

Data parallel languages like High Performance Fortran (HPF) are emerging as the architecture independent mode of programming distributed memory parallel machines. In this paper, we present the interprocedural optimizations required for compiling applications having irregular data access patterns, when coded in such data parallel languages. We have developed an Interprocedural Partial Redundancy Elimination (IPRE) algorithm for optimized placement of runtime preprocessing routine and collective communication routines inserted for managing communication in such codes. We also present three new interprocedural optimizations: placement of scatter routines, deletion of data structures and use of coalescing and incremental routines. We then describe how program slicing can be used for further applying IPRE in more complex scenarios. We have done a preliminary implementation of the schemes presented here using the Fortran D compilation system as the necessary infrastructure. We present experimental results from two codes compiled using our system to demonstrate the efficacy of the presented schemes.

1 Introduction

In recent years, there have been major efforts in developing language and compiler support for programming distributed memory machines. High Performance Fortran (HPF) consists of Fortran 90 extensions designed to allow users to specify parallelism and data distributions in a high level manner. The first round of HPF language definition has been completed [28] and many commercial HPF compiler development projects are currently underway. Efforts are also underway in the High Performance Fortran Forum to increase the scope of HPF for compiling a wider range of applications.

Traditionally, data parallel programming languages like HPF are considered to be most suited for compiling regular or structured mesh applications, in which loop partitioning and communication can be statically determined by the compiler. However, significant effort has also been made to compile applications having irregular and/or dynamic data accesses (possibly with the help of additional language support) [7, 14, 23, 29, 30, 31, 34, 38]. For such codes, the compiler can analyze the data access pattern and insert appropriate communication and communication preprocessing routines.

Recent work has demonstrated that sophisticated compilation techniques can play a crucial role in optimizing performance obtained from irregular codes [14, 24]. Thus far, experiences and experimental results reported have been from small code templates. We anticipate that the ability to apply optimizations across procedure boundaries will prove to be extremely important in generating efficient parallel code in large applications.

In this paper, we discuss the interprocedural analysis and optimizations for compiling irregular applications. Specifically, we concentrate on applications in which data is accessed using indirection arrays. Such codes are common in computational fluid dynamics, molecular dynamics, in Particle In Cell (PIC) problems and in numerical simulations [11].

The commonly used approach for compiling irregular applications is the inspector/executor model [29]. Conceptually, an *inspector* or a *communication preprocessing statement* analyzes the indirection array to determine the communication required by a data parallel loop. The results of communication preprocessing is then used to perform the communication. CHAOS/PARTI library provides a rich set of routines for performing the communication preprocessing and optimized communication for such applications [35]. The Fortran D compilation system, a prototype compiler for distributed memory machines, initially targeted regular applications [26] but has more recently been extended to compile irregular applications [14, 23]. In compiling irregular applications, the Fortran D compiler inserts calls to CHAOS/PARTI library routines to manage communication [14, 22].

An important optimization required for irregular applications is placement of communication preprocessing and communication statements. Techniques for performing these optimizations within a single procedure are well developed [18, 24]. The key idea underlying these schemes is to do the placement so that redundancies are reduced or eliminated. These schemes are based upon a classical data flow framework called Partial Redundancy Elimination (PRE) [16, 32]. PRE encompasses traditional optimizations like loop invariant code motion and redundant computation elimination.

We have developed an Interprocedural Partial Redundancy Elimination framework (IPRE) [1, 2] as a basis for performing interprocedural placement. In this paper, we discuss various practical aspects in applying interprocedural partial redundancy elimination for placement of communication and communication preprocessing statements. We also present a number of other interprocedural optimizations useful in compiling irregular applications, this includes placement of scatter operations, deletion of data structures constructed at runtime and use of incremental and coalescing routines. While none of these optimizations can be directly achieved by the basic IPRE scheme, they can be achieved through extending the IPRE scheme or by using a variation of the IPRE analysis. We then discuss how the notion of program slicing can be used for increasing the scope of IPRE. We also discuss a related issue of ordering application of IPRE on various candidates within a single procedure.

We have carried out a preliminary implementation of the schemes presented in this paper, using the existing Fortran D compilation system as the necessary infrastructure. We present experimental results from the codes compiled using the prototype compiler to demonstrate the effectiveness of our methods.

While several details and examples presented in this paper specifically concentrate on codes which use indirection arrays, the general ideas broadly apply to all applications in which communication preprocessing calls are inserted and/or collective communication routines are used. We have shown in our previous work how communication preprocessing is useful in regular applications in which data distribution, strides and/or loop bounds are not known at compile-time [3, 5, 4, 36] or when the number of processors available for the execution of the program varies at runtime [17].

The rest of the paper is organized as follows. In Section 2, we discuss the basic IPRE framework. In Section 3, we present several new optimizations required for compiling irregular applications. In Section 4, we discuss modifications and extensions required in IPRE framework, in applying it for placement of communication preprocessing statements in some more complex scenarios. An overall compilation algorithm is presented in Section 5. We present experimental results in Section 6. We briefly compare our work with related work in Section 7 and conclude in Section 8.

2 Partial Redundancy Elimination

Most of the interprocedural optimizations required for irregular applications involve some kind of redundancy elimination or loop invariant code motion. Partial Redundancy Elimination (PRE) is a unified framework for performing these optimizations intraprocedurally [16, 32]. It has been commonly used intraprocedurally for performing optimizations like common subexpression elimination and strength reduction. More recently, it has been used for more complex code placement tasks like placement of communication statements while compiling for parallel machines [18, 24]. We have extended an existing intraprocedural partial redundancy scheme to be applied interprocedurally [1, 2]. In this section, we describe the functionality of the PRE framework, key data flow properties associated with it and briefly sketch how we have extended an existing intraprocedural scheme interprocedurally.

Consider any computation of an expression or a call to a pure function. In the program text, we may want to optimize its placement, i.e. place the computation so that the result of the computation is used as often as possible and, redundant computations are removed. For convenience, we refer to any such computation whose placement we want to optimize as a *candidate*. If this candidate is an expression, we refer to the operands of the expression as *influencers* of the candidate. If this candidate is a pure function, we refer to the parameters of the pure function as the *influencers* of the candidate.

There are three type of optimizations which are performed under PRE:

- *Loop invariant Code Motion:* If the influencers of a candidate are all invariant in the loop, then the candidate can be computed just once, before entering the loop.

(a)

```
A = ... A ...
Do i = 1, 20
    ...
    R = A * B
    ...
Enddo

↓

A = ... A ...
H = A * B
Do i = 1, 20
    ...
    R = H
    ...
Enddo
```

(b)

```
A = ... A ...
R = A * B
...
S = A * B
...

↓

A = ... A ...
H = A * B
R = H
...
S = H
...
```

(c)

```
A = ... A ...
R = A*B
...
If foo
    A = ... A ...
Endif
...
S = A*B

↓

A = ... A ...
H = A*B
R = H
...
If foo
    A = ... A ...
    H = A*B
Endif
...
S = H
...
```

Figure 1: Examples of functionality of Partial Redundancy Elimination. (a): Loop invariant code motion, (b): Redundant code elimination, (c): Supressing partial redundancies

- *Redundant Computation Elimination:* We may find two consecutive occurrences of a computation, such that none of influencers of the candidate are modified along any control flow path from the first occurrence to the second occurrence. In this case, the second occurrence is redundant and is deleted as part of the PRE framework.

- *Suppressing Partial Redundancies:* We may find two consecutive occurrences of a computation such that one or more influencers are modified along some possible control flow path (but not all flow paths) from the first occurrence to the second occurrence. In this case, the second occurrence of the candidate is called partially redundant. By placing candidates along the control flow paths associated with the modification, the partially redundant computation can be made redundant and thus be deleted.

Figure 1 explains the functionality of PRE through small code templates. In 1(a), if the influencers A and B are not modified inside the loop, then the computation $A*B$ is loop invariant and can be placed before entering the loop. In 1(b), if the influencers A and B are not modified between the two computations of $A*B$, then the second computation is redundant and can be replaced. In 1(c), the second computation of $A*B$ is partially redundant. This is because if foo is true, then the influencer A is modified, and the second computation of $A*B$ is not redundant (since this computation will give different answer than the first computation). If foo is not true, then A is not modified, and the second computation is redundant. In this case, additional placement of the computation $A*B$ can be carried out to make the partially redundant occurrence fully redundant. This is termed as suppressing partial redundancies.

We now introduce the key data flow properties that are computed as part of this framework. We use these terms for explaining several new optimizations later in the paper. The properties are:

- *Availability:* Availability of a candidate C at any point p in the program means that C lies on each of the paths leading to point p and if C were to be placed at point p, C will have the same result as the result of the last occurrence on any of the paths.

- *Partial Availability:* Partial availability of a candidate C at a point p in the program means that C is currently placed on at least one control flow path leading to p and if C were to be placed at the point p, C will have the same result as the result of the last occurrence on at least one of the paths.

- *Anticipability:* Anticipability of a candidate C at a point p in the program means that C is currently placed on all the paths leading from point p, and if C were to be placed at point p, C will have the same result as the result of the first occurrence on any of the paths.

- *Transparency* Transparency of a basic block with respect to a candidate means that none of the influencers of the candidate are modified in the basic block.

A *basic block* of code in a procedure is a sequence of consecutive statements in a procedure in the flow enters at the beginning and leaves at the end without possibility of branching expect at the end [6].

If a candidate is placed at a point p in the program, and if it is available at the point p, then the occurrence of the candidate at the point p is redundant. If a candidate is placed at a point p in the program, and if it is partially available at the point, then it is considered to be partially redundant. Anticipability of a computation is used for determining if the placement will be *safe*. A *Safe* placement means that at least one occurrence of the candidate will be made redundant by this new placement (and will consequently be deleted). Performing safe placements guarantees that along any path, number of computations of the candidate are not increased after applying optimizing transformations.

By solving data flow equations on the Control Flow Graph (CFG) of a procedure, the Availability, Partial Availability and Anticipability properties are computed at the beginning and end of each basic block in the procedure. Transparency is used for propagating these properties, e.g. if a candidate is available at the beginning of a basic block and if the basic block is transparent with respect to this candidate, then the candidate will be available at the end of the basic block also.

Based upon the above data flow properties, another round of data flow analysis is done to determine properties PPIN (possible placement at the beginning) and PPOUT (possible placement at the end). These properties are then used for determining final placement and deletion of the candidates. We do not present the details of data flow equations in the paper.

Our interest is in applying the PRE framework for optimizing placement of communication preprocessing statements and collective communication statements. The first step in this direction was to extend the existing PRE framework interprocedurally. For applying this transformation across procedure boundaries, we need a full program representation. We have chosen a concise full program representation, which will allow efficient data flow analysis, while maintaining sufficient precision to allow useful transformations and to ensure safety and correctness of transformations.

2.1 Program Representation

In traditional interprocedural analysis, program is abstracted by a *call graph* [19, 20]. In a call graph $G = (V, E)$, V is the set of procedures and directed edge $e = (i, j)$ ($e \in E$) represents a call site in which procedure i invokes procedure j. The limitation of call graph is that no information is available about control flow relationships between various call sites within a procedure. We have developed a new program representation called Full Program Representation (FPR). In this subsection we describe how this structure is constructed for any program.

We define a basic block to consist of consecutive statements in the program text without any procedure calls or return statements, and no branching except at the beginning and end. A procedure can then be partitioned into a set of basic blocks, a set of procedure call statements and a set of return statements. A return statement ends the invocation of procedure or subroutine call.

In our program representation, the basic idea is to construct *blocks of code* within each procedure. A block of code comprises of basic blocks which do not have any call statements between them. In the directed graph we define below, each edge e corresponds to a block of code $B(e)$. A block of code is a unit of placement in our analysis, i.e. we initially consider placement only at the beginning and end of a block of code. The nodes of the graph help clarify the control flow relationships between the blocks of code.

Full Program Representation: (*FPR*) is a directed multigraph $G = (V, E)$, where the set of nodes V consists of an entry node and a return node for each procedure in the program. For procedure i, the entry node is denoted by s_i and the return node is denoted by r_i. Edges are inserted in the following cases:

1. Procedures i and j are called by procedure k at call sites cs_1 and cs_2 respectively and there is a path in the CFG of k from cs_1 to cs_2 which does not include any other call statements. Edge (r_i, s_j) exists in this case. The block of code $B(e)$ consists of the basic blocks of procedure k which may be visited in any control flow path p from cs_1 to cs_2, such that the path p does not include any other call statements.

2. Procedure i calls procedure j at call site cs and there is a path in the CFG of i from the *start* node of procedure i to cs which does not include any other call statements. In this case, edge (s_i, s_j) exists. The block of code $B(e)$ consists of the basic blocks of procedure i which may be visited in any control flow path p from start of i to cs, such that the path p does not include any other call statement.

```
      Program Example
      Real X(nnodes), Y(nnodes)
      Real Z(nedges), W(nedges)
      Integer IA(nedges), IB(nedges)

C     Input data ...
      do 10 i = 1, 20
      Call Proc_A(X,Y,Z,IA,IB)
      if (nt .gt. 0) then
        Call Proc_B(X,W,IA)
      endif
      do 50 j = 1, nedges
      IB(j) = .. IB(j) ..
50    continue
10    continue
      end

      Subroutine Proc_A(A,B,C,D,E)
      do 20 i = 1, nedges
      C(i) = C(i) + A(D(i))
20    continue
      do 30 i = 1, nedges
      C(i) = C(i) + B(E(i))
30    continue
      do 35 i = 1, nnodes
      B(i) = ...
35    continue
      end

      Subroutine Proc_B(X,W,IA)
      do 40 i = 1, nedges
      W(i) = W(i) + X(IA(i))
40    continue
      do 45 i = 1, nnodes
      X(i) = ...
45    continue
      end
```

Figure 2: An Irregular Code

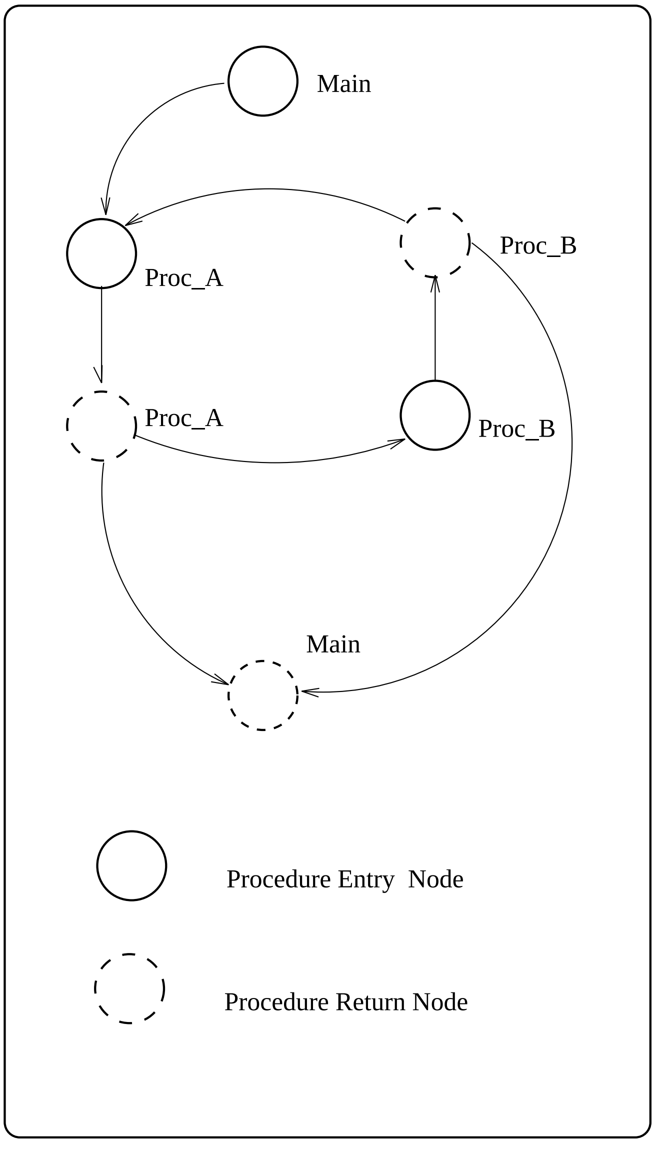


Figure 3: *FPR* for the example program

3. Procedure j calls procedure i at call site cs and there is a path in the CFG of j from call site cs to a return statement within procedure j which does not include any other call statements. In this case, edge (r_i, r_j) exists. The block of code $B(e)$ consists of the basic blocks of procedure j which may be visited in any control flow path p from cs to a return statement of j, such that the path p does not include any call statements.

4. In a procedure i, there is a possible flow of control from start node to a return statement, without any call statements. In this case, edge (s_i, r_i) exists. The block of code $B(e)$ consists of the basic blocks of procedure i which may be visited in any control flow path p from start of i to a return statement in i, such that the path p does not include any call statements.

In Figure 2, we show an example program (which involves irregular accesses to data). The program represenation FPR for this program is shown in Figure 3.

For performing partial redundancy elimination on the full program, we apply data flow analysis on FPR, rather than the CFG of a single procedure. Instead of considering transparency of each basic block, we consider transparency of each edge or the block of code. The data flow properties are computed for the beginning and the end of each edge in the FPR program representation. The details of the data flow analysis required for computing the above properties and then determining placement and deletion based on these has been given elsewhere [1, 2]. There are several difficulties in extending the analysis interprocedurally, this includes renaming of influencers across procedure boundaries, saving the calling context of procedures which are called at more than one call sites and further intraprocedural analysis in each procedure to determine final local placement. These details have been presented elsewhere and are not the focus of this paper.

We are only interested in placement of communication preprocessing statements and collective communication statements. A particular invocation of a communication preprocessing statement or a collective communication statement is considered for hoisting out of the procedure only if none of the influencers is modified along any path from the start of the procedure to this invocation of the statement and the statement is not enclosed by any conditional or loop.

2.2 Applying IPRE for Communication Optimizations

We briefly show how partial redundancy elimination is used for optimizing placement of communication preprocessing calls and collective communication routines. We use the example presented in Figure 2 to show the communication preprocessing inserted by initial intraprocedural analysis, and the interprocedural optimizations that can be done.

Initial intraprocedural analysis inserts one communication preprocessing call and one gather (collective communication routine) for each of the three data parallel loops in the program shown in Figure 4. We have omitted several parameters to both the communication preprocessing routines and collective communication routines for keeping the examples simple. Consider the

```
      Program Example
      Real X(nnodes), Y(nnodes)
      Real Z(nedges), W(nedges)
      Integer IA(nedges), IB(nedges)
C     Input data ...
      do 10 i = 1, 20
      Call Proc_A(X,Y,Z,IA,IB)
      if (nt .gt. 0) then
        Call Proc_B(X,W,IA)
      endif
      do 50 j = 1, nedges_local
      IB(j) = .. IB(j) ..
50    continue
10    continue
      end

      Subroutine Proc_A(A,B,C,D,E)
      Sched1 = Irreg_Sched(D)
      Call Gather(A,Sched1)
      do 20 i = 1, nedges_local
      C(i) = C(i) + A(D(i))
20    continue
      Sched2 = Irreg_Sched(E)
      Call Gather(B,Sched2)
      do 30 i = 1, nedges_local
      C(i) = C(i) + B(E(i))
30    continue
      do 35 i = 1, nnodes_local
      B(i) = ...
35    continue
      end

      Subroutine Proc_B(X,W,IA)
      Sched3 = Irreg_Sched(IA)
      Call Gather(X,Sched3)
      do 40 i = 1, nedges_local
      W(i) = W(i) + X(IA(i))
40    continue
      do 45 i = 1, nnodes_local
      X(i) = ...
45    continue
      end
```

```
      Program Example
      Real X(nnodes), Y(nnodes)
      Real Z(nedges), W(nedges)
      Integer IA(nedges), IB(nedges)
C     Input data ...
      Sched1 = Irreg_Sched(IA)
      do 10 i = 1, 20
      Call Proc_A(X,Y,Z,IA,IB)
      if (nt .gt. 0) then
        Call Proc_B(X,W,IA)
      endif
      do 50 j = 1, nedges_local
      IB(j) = .. IB(j)..
50    continue
10    continue
      end

      Subroutine Proc_A(A,B,C,D,E)
      Call Gather(A,Sched1)
      do 20 i = 1, nedges_local
      C(i) = C(i) + A(D(i))
20    continue
      Sched2 = Irreg_Sched(E)
      Call Gather(B,Sched2)
      do 30 i = 1, nedges_local
      C(i) = C(i) + B(E(i))
30    continue
      do 35 i = 1, nnodes_local
      B(i) = ...
35    continue
      end

      Subroutine Proc_B(X,W,IA)
      do 40 i = 1, nedges_local
      W(i) = W(i) + X(IA(i))
40    continue
      do 45 i = 1, nnodes_local
      X(i) = ...
45    continue
      end
```

Figure 4: Result of Intraprocedural Compilation (left), and Code after Interprocedural Optimizations (right)

execution of the partitioned data parallel loop on a particular processor. The off-processor elements referred to on this processor are fetched before the start of the loop. A simple memory management scheme is used in the CHAOS/PARTI framework. For each *data array* (i.e. an array whose contents are accessed using indirection arrays), a ghost area is created, contiguous with the local data array. The off-processor elements referred to in the parallel loop are stored in this ghost area. The communication preprocessing routine *Irreg_Sched* takes in the indirection array and information about distribution of the data arrays. Besides computing a communication schedule, it outputs a new local version of the indirection array and the number of off-processor accesses made by the loop. In this new local version of the indirection array, the off-processor references are replaced by appropriate references to the elements in the ghost area. The collective communication calls also need the starting position of the ghost area as one of the parameters. For simplicity, this detail is omitted in all the examples.

In Figure 4, we also show the program after interprocedural optimization of communication preprocessing routines and gather routines. We refer to loop in the main of the program (which encloses the calls to the routines Proc_A and Proc_B) as the *time step* loop. Initially, interprocedural partial redundancy elimination is applied for communication preprocessing statements. Since the array IA is never modified inside the time step loop in the main procedure, the schedules $Sched1$ and $Sched3$ are loop invariant and can be hoisted outside the loop. Further, it can be deduced that the computation of $Sched1$ and $Sched3$ are equivalent (since their influencers, after renaming across procedure boundaries, are the same). So, only $Sched1$ needs to be computed, and the gather routine in Proc_B can use $Sched1$ instead of $Sched3$. For simplicity, $Sched1$ is declared to be a global variable, so that it does not need to be passed along as parameter at different call sites. After placement of communication preprocessing statements is determined, we apply the IPRE analysis for communication routines. The gather for array IA in routine Proc_B is redundant because of the gather of array D in routine Proc_A. Note that performing IPRE on communication preprocessing statements before applying IPRE on communication statements is critical, since it is important to know that $Sched3$, one of the influencers of gather for array IB can be replaced by $Sched1$.

2.3 Discussion

In the rest of this paper, we concentrate on three issues:

- We discuss three new optimizations that are useful in compilation of irregular applications. These three optimizations are: placement of scatter operations, deletion of runtime data structures and the use of incremental and coalescing routines. While none of these optimizations can be directly achieved by the IPRE scheme we have so far described, they can be achieved by extending the IPRE scheme or using a variation of the basic IPRE analysis.

- We extend the applicability of IPRE, by considering slices of candidates and performing motion of the entire slice. We also discuss the related issue of determining the order in which IPRE is to be applied over different candidates from the same procedure.

- We describe the implementation of the IPRE framework and the extensions mentioned above using the Fortran D compilation system as the necessary infrastructure. We also report experimental results which demonstrate the efficacy of our methods.

3 Other Optimizations for Compiling Irregular Problems

In this section, we discuss three new interprocedural optimizations which are useful in compiling irregular applications. These optimizations are: placement of scatter operations, deletion of runtime data structures and use of incremental and coalescing routines. While none of these optimizations can be directly achieved by the interprocedural partial redundancy elimination scheme we have so far described, they can be achieved through extending the IPRE scheme or using a variation of the basic IPRE analysis.

3.1 Placement of Scatter Operations

Collective communication routines can be broadly classified to be of two kinds: *gathers* and *scatters*. By gather, we mean a routine which, before entering a data parallel loop, collects the off-processor elements referred to in the loop. By scatter, we mean a routine which, after a data parallel loop, updates the off-processor elements modified by the loop.

In distributed memory compilation, a commonly used technique for loop iteration partitioning is *owner computes rule* [26]. In this method, each iteration is executed by the processor which owns the left hand side array reference updated by the iteration. If the owner computes rule is used, then no communication is required after the end of a data parallel loop, since no off-processor element is modified by the loop.

Owner computes rule is often not best suited for irregular codes. This is because of two reasons: Use of indirection in accessing left hand side array makes it difficult to partition the loop iterations according to the owner computes rule, secondly, because of the use of indirection in accessing right hand side elements, total communication may be reduced by using heuristics other than the owner computes rule.

If a method other than owner computes is used for loop partitioning, there is need for routines *scatter_op*, which will perform an *op* on the off-processor data, using the values computed in the loop. In Figure 5, we show an example of a code requiring scatter_op routines. In the two data parallel loops, loop iteration i is executed by processor owning $Z(i)$ and $W(i)$ respectively. Further, suppose that the arrays W, X and Z are identically distributed. Array element X(IA(i)) is modified (an addition operation is performed) in such an iteration, and in general, this can be an off-processor reference. The communication preprocessing routine generates a new local version of the array IA, in which the references to the off-processor elements are changed to references to the elements in the ghost area. Modifications to the off-processor references are stored in the ghost area. (Before the loops, the elements of the ghost area need to be initialized to 0, this detail is omitted from our example). After the end of the loop, the collective communication routine *scatter_add* is used to update the off-processor elements.

```
      Program Example
      Real X(nnodes)
      Real Z(nedges), W(nedges)
      Integer IA(nedges)
C     Input data ...
      do 10 i = 1, 20
      Call Proc_A(X,Z,IA)
      Call Proc_B(X,W,IA)
10    continue
      end

      Proc_A(A,B,C)
      do 20 i = 1, nedges
      A(C(i)) = A(C(i)) + B(i)
20    continue
      end

      Proc_B(X,W,IA)
      do 40 i = 1, nedges
      X(IA(i)) = X(IA(i)) + W(i)
40    continue
      do 45 i = 1, nnodes
      X(i) = ...
45    continue
      end
```

```
      Program Example
      Real X(nnodes)
      Real Z(nedges), W(nedges)
      Integer IA(nedges)
C     Input data ...
      do 10 i = 1, 20
      Call Proc_A(X,Z,IA)
      Call Proc_B(X,W,IA)
10    continue
      end

      Proc_A(A,B,C)
      Sched1 = Irreg_Sched(C)
      do 20 i = 1, nedges_local
      A(C(i)) = A(C(i)) + B(i)
20    continue
      Call Scatter_add(A, Sched1)
      end

      Proc_B(X,W,IA)
      Sched2 = Irreg_Sched(IA)
      do 40 i = 1, nedges_local
      X(IA(i)) = X(IA(i)) + W(i)
40    continue
      Call Scatter_add(X,Sched2)
      do 45 i = 1, nnodes_local
      X(i) = ...
45    continue
      end
```

```
      Program Example
      Real X(nnodes)
      Real Z(nedges), W(nedges)
      Integer IA(nedges)
C     Input data ...
      Sched1 = Irreg_Sched(IA)
      do 10 i = 1, 20
      Call Proc_A(X,Z,IA)
      Call Proc_B(X,W,IA)
10    continue
      end

      Proc_A(A,B,C)
      do 20 i = 1, nedges_local
      A(C(i)) = A(C(i)) + B(i)
20    continue
      end

      Proc_B(X,W,IA)
      do 40 i = 1, nedges_local
      X(IA(i)) = X(IA(i)) + W(i)
40    continue
      Call Scatter_add(X,Sched1)
      do 45 i = 1, nnodes_local
      X(i) = ...
45    continue
      end
```

Figure 5: Compilation and optimization of a code involving scatter operations: Original sequential code (left), Result of Intraprocedural Compilation (center), and Code after Interprocedural Optimizations (right)

In the example presented in Section 2, the collective communication routine involved were the gather operations. For performing optimized placements, gather operations were treated in the same way as the communication preprocessing routines. We now discuss what kind of analysis is required to determine optimized placement of scatter_ops.

There are two differences in dealing with scatters_ops as compared to gathers. We have seen so far, how the placement of a gather operation can be moved earlier, if this can reduce redundant communication. The required condition is that the placement must be done after the last modification of the array whose data is being gathered. Thus, we need to check if the array whose data is being gathered is modified.

In the case of scatter_ops, the placement can be done later, if this can reduce redundancies. The required condition is that the array whose data is being scattered must not be *referred* to or *modified.* If the array being scattered is referred to, then the reference made may be incorrect because the modifications made in an earlier loop have not been updated. Similarly, if the array being scattered is modified, then the updates made later may be incorrect.

Optimization of scatter_ops is therefore done by applying IPRE scheme with three differences:

- We consider a scatter operation for interprocedural placement only if none of the influencers are modified or referred to along any control flow path from the scatter's invocation to the end of the procedure, and if this invocation of scatter operation is not enclosed by any conditional or loop.

- We change the definition of Transparency, to check if the influencers of the candidate are neither referred to nor modified.

- We consider our graph, as defined in Section 2, with the notion of source and sink reversed. Thus, we tend to move the scatter_ops downwards, if there is any redundancy to be eliminated this way.

In Figure 5, the result of interprocedural optimization is shown in the right. In the procedure Proc_A, the scatter operation can be deleted, since this scatter is subsumed by the scatter done later in Proc_B.

Scatter operations have also been used by distributed memory compilers in compiling regular applications [8]. The HPF/Fortran 90D compiler developed at Syracuse University uses scatter operations (called *post-comp writes*) whenever the subscript in the left hand side array reference is a complex function of the index variable. The optimization described above will therefore be applicable in compiling regular applications also.

3.2 Deletion of Data Structures

Runtime preprocessing often results in construction of large data structures, which are used by other routines later. This includes communication schedules which store information about the off-processor elements which need to be gathered/scattered to each other processor. Large scientific applications involve large arrays and consequently, the memory required by the data structures like communication schedules can be large.

In hand parallelizing applications using libraries like CHAOS/PARTI, it is generally useful to free the memory required by these data structures after the last time they are used. Since the large distributed arrays themselves require large memory, it is important not to let these data structures increase the memory usage of the program substantially. This is even more important on machines which do not support virtual memory.

If a compiler does an unoptimized placement of communication preprocessing calls (i.e. placement just on the basis of a single loop level or single procedure level analysis), then data structures can be easily deleted after their use is over. However, this is a non-trivial problem when interprocedural analysis is performed to do optimized placement.

We now describe how to determine the places where the data structures can be deleted. The key idea is to make sure that there must not be any use of the data structure along any control flow path starting from the point where it is deleted. In ensuring this, our method may not delete a data structure ever (which is equivalent to saying that it is deleted at the end of the program). The steps of our method are as follows:

- Interprocedural analysis is done to determine optimized placement of communication preprocessing routines and collective communication routines. None of the schedules are initially deleted.
- We mark a placement of *free(sched)*, immediately after each use of the sched. For the analysis here, we consider these *free(sched)* statements as the candidates for placement.
- We determine optimized placement of these candidates, by applying IPRE analysis on the reversed graph (i.e. FPR with notion of source and sink reversed, as used earlier for determining placement of scatter operations).
- After determining placement of the these candidates, we check if the candidate is partially available at any of the places where it is marked for placement. (The partial availability we use must be computed on the reversed graph.) We actually place a *free(sched)* only if it is not partially available.

The significance of the last step mentioned above is as follows. In placement of candidates, PRE or IPRE analysis can do a placement at a point where the candidate may be partially available. So, if the analysis has determined that an optimized placement of the candidate needs to be done at a point p in the program, there may already be another placement of the candidate at one of the paths leading to the point p. In determining deletion of data structures, we cannot place a deletion if the schedule is going to be used at any path starting from that point.

3.3 Using Incremental and Coalescing Communication Routines

Consider an occurrence of a communication statement. While this communication statement may not be redundant (the same candidate may not be directly available), there may be some other

```
      Program Example
      Real X(nnodes)
      Real Z(nedges), W(nedges)
      Integer IA(nedges), IB(nedges)
C     Input data ...
      do 10 i = 1, 20
      Call Proc_A(X,Z,IA)
      if (nt .gt. 0) then
      Call Proc_B(X,W,IB)
      endif
10    continue
      end

      Subroutine Proc_A(A,B,C)
      do 20 i = 1, nedges
      B(i) = B(i) + A(C(i))
20    continue
      end

      Subroutine Proc_B(X,W,IB)
      do 40 i = 1, nedges
      W(i) = W(i) + X(IB(i))
40    continue
      do 45 i = 1, nnodes
      X(i) = ...
45    continue
      end
```

```
      Program Example
      Real X(nnodes)
      Real Z(nedges), W(nedges)
      Integer IA(nedges), IB(nedges)
C     Input data ...
      Sched1 = Irreg_Sched(IA)
      Sched2 = Irreg_Sched_Inc(IB,IA)
      do 10 i = 1, 20
      Call Proc_A(X,Z,IA)
      if (nt .gt. 0) then
      Call Proc_B(X,W,IB)
      endif
10    continue
      end

      Subroutine Proc_A(A,B,C)
      Call Gather(A,Sched1)
      do 20 i = 1, nedges_local
      B(i) = B(i) + A(C(i))
20    continue
      end

      Subroutine Proc_B(X,W,IB)
      Call Gather(X,Sched2)
      do 40 i = 1, nedges_local
      W(i) = W(i) + X(IB(i))
40    continue
      do 45 i = 1, nnodes_local
      X(i) = ...
45    continue
      end
```

Figure 6: Use of incremental schedules. Original code is shown in left and the SPMD code (after Interprocedural Optimizations) is shown in right

communication statement, which may be gathering at least a subset of the values gathered in this statement. The execution time of the code can be reduced by disallowing redundant gathering of certain data elements.

Consider the program shown in Figure 6. The same data array X is accessed using an indirection array IA in the procedure Proc_A and using another indirection array IB in the procedure Proc_B. Further, none of the indirection arrays or the data array X is modified between flow of control from first loop to the second loop. The set of data elements to be communicated between the processors can only be determined at runtime, however it is very likely that there will be at least some overlap between the set of off-processor references made in these two loops. At the time of schedule generation, the contents of the array IA and IB can be analyzed to reduce the net communication required by these two loops.

PARTI/CHAOS library provides two kinds of communication routines for reducing communication in such situations. *Coalescing* preprocessing routines take more than one indirection array, and produce a single schedule, which can be used for generating the communication required by different loops. In the example mentioned above, a coalescing communication preprocessing routine will take in arrays IA and IB and produce a single communication schedule. If a gather operation is done using this schedule, then all off-processor elements referred to through indirection arrays IA and IB will be gathered. *Incremental* preprocessing routine will take in indirection arrays IA and IB, and will determine the off-processor references made uniquely through indirection array IB and not through indirection array IA (or vice-versa). While executing the second loop, communication using an incremental schedule can be done, to gather only the data elements which were not gathered during the first loop.

Use of both incremental and coalescing routines reduces the net communication volume. The advantage of using coalescing routines over incremental routines is that only one message is required for communication. This further reduces the communication latency involved.

The following analysis is done to determine use of coalescing and incremental communication preprocessing routines. After the placement of communication preprocessing and communication statements has been determined, consider two communication statements $L1$ and $L2$, which do gathers for the same data array.

Recall the definition of *Availability* and *Anticipability*, as presented in Section 2. The communication done by the statements $L1$ and $L2$ can be done by using a single coalescing routine if the following holds:

- The communication done in $L1$ is available at the point $L2$ in the program, **and**
- The communication done in $L2$ is anticipable at the point $L1$ in the program.

In this case, the communication at $L2$ can be deleted and the communication at $L1$ can be replaced by a coalesced communication. The first condition above ensures that the elements communicated at the point $L1$ in the program will still be valid at the point $L2$ in the program. If the communication at $L1$ is replaced by a coalesced communication, then the second condition above ensures that, along any control flow path starting from $L1$, the additional data communicated will be used.

The second communication can be replaced by an incremental communication if the following conditions hold:

- The communication done in $L1$ is available at the point $L2$ in the program, **and**
- The communication done in $L2$ is **not** anticipable at the point $L1$ in the program.

In this case, the communication statement at $L1$ remains as it is and the communication at $L2$ can be replaced by an incremental communication. In Figure 6, we show the use of incremental routines. Note that the call to the procedure Proc_B is enclosed inside a conditional, so the second communication is not anticipable at the point of the first communication. If this conditional was not there, then the second communication could be removed all together and the first communication could be replaced by a coalesced communication.

The analysis described above can be performed at two stages. After calls to communication preprocessing routines and communication statements have been inserted by initial intraprocedural analysis, the above analysis can be done intraprocedurally. For this purpose, availability and anticipability must be computed intraprocedurally on the CFG of the single routine. Next, after optimization of communication preprocessing routines and communication statements has been done through IPRE, another round of the analysis described above can be done on the *FPR*. In this case, availability and anticipability is computed on the *FPR*.

The scatter operations can also be optimized further using coalescing and incremental routines. The difference in analysis would be to consider the graph with notion of source and sink reversed and the definition of transparency changed to use both Mod and Ref information instead of just the Mod information.

4 Further Application of IPRE

In this section, we first discuss how program slicing can be used for further applying IPRE in more complex scenarios. We then discuss the related issue of determining the order in which IPRE can be applied to different candidates from the same procedure.

4.1 Use of Slicing

In all the examples presented so far, the parameters of the candidates were formal parameters or global variables. As described in Section 2, such a call to a candidate can be considered for placement across procedure boundaries only if none of the influencers is modified along any path from the start of the procedure to this invocation of the candidate, and if the call by itself is not enclosed by any conditional or loop.

This may not be adequate for performing code motion in several irregular applications, especially the ones in which data is accessed using multiple levels of indirection [14]. For such codes, IPRE can be performed by using *slices* of the call to the candidates.

```
   Program Example
   Real X(n), Real Z(n)
   Integer P(n), Q(n)
C  Input data ...
   do 10 i = 1, 20
   Call Proc_A(X,Z,P,Q)
   do 55 l = 1, n
   Q(l) = ...
55 continue
10 continue
   end


   Proc_A(X,Z,P,Q)
   do 20 j = 1, 20
   Call Proc_B(X,Z,P,Q)
20 continue
   end


   Proc_B(X,Z,P,Q)
   Integer R(n/2), S(n)
   k = 0
   do 30 l = 1, n, 2
   k = k + 1
   R(k) = P(l)
30 continue
   do 35 l = 1, n
   S(l) = Q(R(2*l)) + P(l)
35 continue
   do 40 l = 1, n
   X(l) = X(l) + Z(S(l))
40 continue
   do 45 l = 1, n
   Z(l) = ....
45 continue
   end
```

```
   Program Example
   Real X(n), Real Z(n)
   Integer P(n), Q(n)
C  Input data ...
   do 10 i = 1, 20
   Call Proc_A(X,Z,P,Q)
10 continue
   do 55 l = 1, n_local
   Q(l) = ...
55 continue
   end

   Proc_A(X,Z,P,Q)
   do 20 j = 1, 20
   Call Proc_B(X,Z,P,Q)
20 continue
   end

   Proc_B(X,Z,P,Q)
   Integer R(n/2), S(n)
   k = 0
   do 30 l = 1, n_local, 2
   k = k + 1
   R(k) = P(l)
30 continue
   Sched1 = Irreg_Sched(R)     C1
   Call Gather(Q, Sched1)      C2
   do 35 l = 1, n_local
   S(l) = Q(R(2*l)) + P(l)
35 continue
   Sched2 = Irreg_Sched(S)     C3
   Call Gather(Z,Sched2)       C4
   do 40 l = 1, n_local
   X(l) = X(l) + Z(S(l))
40 continue
   do 45 l = 1, n_local
   Z(l) = ....
45 continue
   end
```

```
   Program Example
   Real X(n), Real Z(n)
   Integer P(n), Q(n)
   k2 = 0
   do 32 l2 = 1, n_local, 2
   k2 = k2 + 1
   R2(k2) = P(l2)
32 continue
   Sched1 = Irreg_Sched(R2)
   do 10 i = 1, 20
   Call Proc_A(X,Z,P,Q,R2)
10 continue
   do 55 l = 1, n_local
   Q(l) = ...
55 continue
   end

   Proc_A(X,Z,P,Q,R2)
   Call Gather(Q, Sched1)
   do 37 l4 = 1, n_local
   S2(l4) = Q(R2(2*l4)) + P(l4)
35 continue
   Sched2 = Irreg_Sched(S2)
   do 20 j = 1, 20
   Call Proc_B(X,Z,P,Q,S2)
20 continue
   end

   Proc_B(X,Z,P,Q,S2)
   Integer R(n/2), S(n)
   Call Gather(Z,Sched2)
   do 40 l = 1, n_local
   X(l) = X(l) + Z(S2(l))
40 continue
   do 45 l = 1, n_local
   Z(l) = ....
45 continue
```

Figure 7: Compilation and optimization of a code involving multiple levels of indirection: Original sequential code (left), Result of Intraprocedural Compilation (center), and Code after Interprocedural Optimizations (right)

Consider the code given in Figure 7. In the procedure Proc_B, the array Q is accessed using array R, which is local within procedure Proc_B. Earlier in the procedure, the array R is computed using array P, which is a formal parameter of the procedure. If the computation of the schedule for communicating Q is to be hoisted up, then the computation of the array R will also need to be moved. For this purpose, we use the notion of program (or procedure) *slices*.

Program Slice. A program (procedure) slice is defined as a program comprising of a set of statements which contribute, either directly or indirectly, to the value of certain variables at a certain point in the program [14, 15, 37]. This set of variables and the point in the program is together referred to as the slicing criterion. For our purpose, the slicing criterion used is the set of parameters of the candidate at the point in the program where the candidate is invoked. We compute the slice of the procedure with respect to the parameters of candidate at the point in the procedure where candidate is called.

We change the definition of influencers of the candidate when we consider entire slice for placement across procedure boundaries. After computing the slice, we identify all global variables and formal parameters of the procedure which contribute, either directly or indirectly, to the value of any of the parameters of the candidate. (These are simply the global variables and formal parameters which appear in the slice). This set of global variables and formal parameters is now called *influencers* of the *candidate*.

An interesting case is the presence of procedure calls in control flow from the start of the procedure to a candidate. For each such procedure call in the control flow path of candidate, we just examine if any of the variables in the slice are modified by the procedure call [12]. If so, we do not consider this candidate for hoisting outside the procedure.

When we use slices of the candidates, additional steps are required in final placement of the candidates. In placing the candidate, the entire slice corresponding to candidate is placed. Note that the slice may include assignments to a number of variables, which may also be referred to later in the procedure (even after the computation of the candidate). While we need to place the entire slice when we hoist the candidate, the entire slice cannot be deleted in the procedure. For this reason, when we place the slice in a new location, all variables written into in the slice (prior to the computation of the candidate) are privatized, i.e., a new name is given to them. While removing the code from the original procedure, only the candidate is removed. After the candidate has been deleted, we can perform dead code elimination to delete the computations which are never used later in the procedure.

4.2 Ordering Application of IPRE

Consider the example shown earlier in Figure 4. In Section 2.2, we had discussed how we need to perform the placement decision for the communication preprocessing statements (i.e. the computation of $Sched1$ and $Sched3$) before we consider the placement of communication statements. This was because the communication statements have the corresponding schedule as one of the influencers. If the influencer is actually computed within the procedure, then the communication statement cannot be considered for interprocedural placement. However, if analysis determines that it is possible to hoist up placement of a communication preprocessing routine, then it is possible that the corresponding communication statement could also be hoisted.

In general, a communication preprocessing routine may use the contents of an array, which by itself is communicated earlier in the procedure. In Figure 7, the result of intraprocedural compilation is shown in the center. There are four candidates in the procedure Proc_B, two communication preprocessing routines (C1 and C3) and two communication statements (C2 and C4). The candidate C3 computes a schedule based upon the contents of array S, array S is computed earlier in the procedure using the array Q. The off-processor references to Q made while computing array R are gathered by the statement C2. When interprocedural placement of the candidate C3 is considered, we need to see if C2 can be hoisted up. The placement of C2, in turn, depends upon placement of C1 and similarly, the placement of C4 depends upon the placement of C3.

Because of the possibility of such dependence between the candidates, there are two important differences in the way we select candidates for placement and apply IPRE.

- While computing the slice of a candidate C_i, we identify all the candidates on whose placement the placement of C_i depends.
- We perform the application of IPRE in such an order, that if the placement of a candidate C_i depends upon the placement of candidates $C_{i1}, \ldots, C_{im}$, then the placement of candidates $C_{i1}, \ldots, C_{im}$, is decided before applying IPRE for placement of C_i.

Computing Slices. Algorithms for computing a slice, given a slicing criterion, have been presented in the literature [37]. We make one important difference in the way slices are computed, since we need to accommodate the fact that some of the statements included in the slice may themselves be candidates for placement. We do not present the modified algorithm formally, but explain the difference with the help of an example.

Consider the slice of the statement "*Sched2 = Irreg_Sched(S)*" (candidate C3). The loop for computing contents of the array S will clearly be included in the slice. This loop includes references to array Q, so the statement(s) modifying array Q also need to be included in the slice. The only such statement is the communication statement "*Call Gather(Q, Sched1)*". This statement is a candidate for placement by itself (C2). In this case, we do not further include the statements which modify Q and $Sched1$ in the slice. Any such statement will obviously be included in the slice for candidate C2. Instead, we mark a dependence $C2 \rightarrow C3$. The significance of this dependence is that if C2 is not moved outside procedure, C3 cannot be moved above procedure either. If it is determined where C2 is to be placed, then the block of code where C2 is placed is considered to be the last modification of the array Q and Sched1. Since Q is one of the influencers of C3, C3 cannot be moved beyond the block of code where the placement of C2 is determined.

Once we have constructed the slices for all the candidates using the method described above, we form a dependence graph between the slices. The dependence graph for the candidates in the procedure Proc_B in Figure 7 will be $C1 \rightarrow C2 \rightarrow C3 \rightarrow C4$.

Applying IPRE. We now determine the order in which IPRE is applied to different candidates from the same procedure. We have described how a dependence graph can be constructed for various candidates within the same procedure. For simplicity, we consider only the dependence graphs which are acyclic. Topological sort is done on the dependence graph formed above for determining the order in which IPRE is applied to each individual candidate. This ensures that if the placement of a candidate C_i depends upon the placement of candidates $C_{i1}, \ldots, C_{im}$, then the placement of candidates $C_{i1}, \ldots, C_{im}$ is determined before performing the analysis for determining placement of C_i.

In Figure 7, the code shown in the right is the result of the interprocedural placement of the slices. The candidate C1 can be moved across the enclosing loops in Proc_A and the main, since the array P is never modified. The candidates C2 and C3 can then be moved across the enclosing loop in the procedure Proc_A.

5 Overall Compilation Algorithm

So far we have presented various optimizations required for compiling irregular applications. We now discuss an overall compilation algorithm, to show how the optimizations are applied and how these optimizations interplay with the rest of the compilation process.

There are three phases in our overall compilation method (see Figure 8). The first phase is the intraprocedural compilation as in the existing Fortran D compilation system. During this phase, we collect information about candidates (including their slices and list of influencers) and control flow relationships between the call sites in each procedure. The second phase performs data flow analysis for optimizing placement. This phase uses only the summary information stored about each procedure in the previous phase. In the third phase, each procedure is visited again, and the decisions made about placement of candidates are actually incorporated in the code for each procedure.

First Phase. The initial local compilation phase inserts communication preprocessing and communication statements based upon intraprocedural analysis [26]. This code generation is based upon *reaching decomposition analysis* [21]. Reaching decomposition analysis propagates information about the distribution of arrays from calling procedures to callees. In compiling languages like Fortran D or HPF, the information about data distribution is used by the compiler for determining loop partitioning, communication and to decide upon the appropriate runtime routines to insert. The existing Fortran D compiler uses the call graph of the full program to determine the order in which procedures are compiled. For most of the Fortran programs, the call graph is a directed acyclic graph. If the procedures are compiled in topological order on the

{* *Initial Intraprocedural Compilation* *}
$Global\_Max\_dep = 0$
Foreach Procedure P (In topological order)
    Propagate reaching decomposition information
    Generate code for distributed memory machines
    Create *blocks of code* from basic blocks of P
    Compute *Mod* and *Ref* Information for each block of code
    Foreach Candidate C compute
        - Slice of C within the procedure
        - The list of influencers for C
        - Determine if C can be hoisted at the top of the procedure
    End
    Generate the dependence graph
    $Global\_Max\_dep = max(Global\_Max\_dep, Max\_dep(P))$
    For $i = 1$ to $Max\_dep(P)$
        Store the list of candidates at level i in procedure P
    End
End

{* *Interprocedural Analysis for Placements* *}
Generate *FPR* for the program
Initialize nodes of *FPR* with candidates for placement
For $i = 1$ to $Global\_Max\_dep$
    Apply IPRE for all candidates at level i (in all procedures)
End
Do analysis for using coalescing and incremental routines
Perform analysis for determining deletion of data structures

{* *Addition/Deletion of Candidates based upon Analysis above* *}
Foreach Procedure P (In any order)
    Do addition/deletion of candidates based upon analysis above
End

Figure 8: Overall Compilation Algorithm

call graph, then each calling procedure is compiled before its callee(s) and the information about data distributions is available while compiling each procedure.

Three important pieces of information are collected during this phase which are used during the second phase. We use the control flow graph of the procedure to compute *blocks of code* (see Section 2.1) for the procedure. Then, we traverse the basic blocks in each block of code for determining *Mod* and *Ref* information for the block of code (i.e. the list of variables modified and referred to, respectively, in each block of code). Next, we identify all the candidates for placement in the procedure. We compute the slices of the candidates in the procedure and find the list of influencers of the candidate. We also construct the dependence graph of the candidates from the procedure. As shown in the Figure 8, the variable $Max\_dep(P)$ determines the maximum depth of any candidate in the dependence graph built for the procedure P. We maintain a variable $Global\_Max\_dep$ to store the maximum of $Max\_dep(P)$ over all the procedures in the program. For each depth level i ($1 \leq i \leq Max\_dep$), we store the list of candidates which are at the level i in the dependence graph of the procedure.

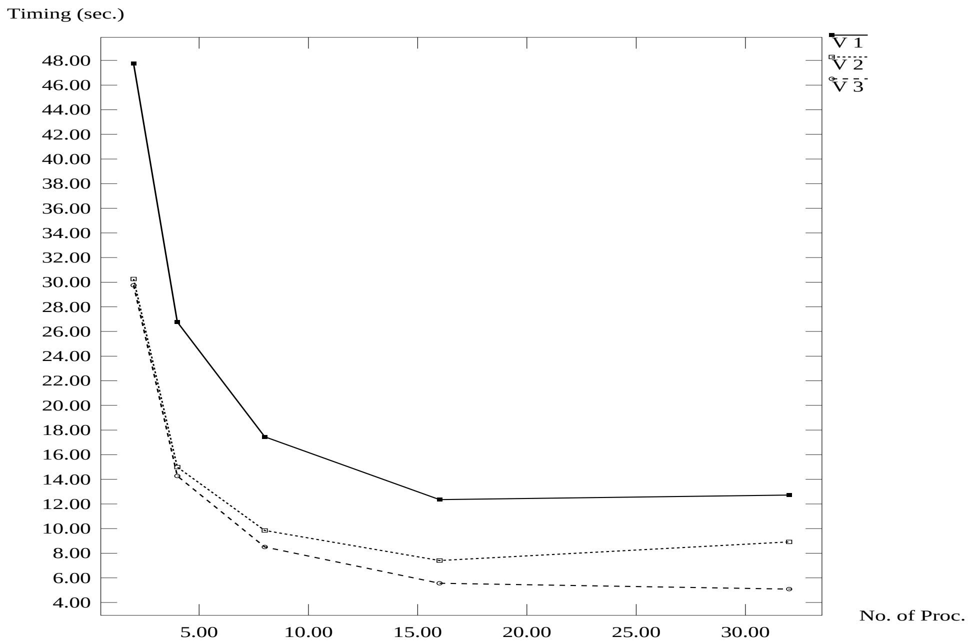
Second Phase. After the initial pass over all the procedures, we perform the data flow analysis for determining placement. The first step is to generate the full program representation (*FPR*) using the summary information computed from each procedure [1]. The procedure entry nodes are then initialized with the candidates for placement.

During the first phase, we have stored the value $Global\_Max\_dep$, the maximum depth level of any candidate in any procedure. We iterate over 1 to $Global\_Max\_dep$, and perform the analysis for placement of all candidates at that depth level, across all the procedures. Next, for each pair of gather routines (or scatter routines), it is checked if communication time can be reduced by using coalescing or incremental routines (Section 3.3). After determining placement of all these routines, analysis described in Section 3.2 is applied to determine where the data structures can be deleted. All information about addition and deletion of statements is just stored in this phase, and no actual change in the code for each procedure is done. This phase uses only the *FPR* constructed in the previous phase, the information associated with blocks of code (Mod and Ref) and the information about candidates. The abstract syntax tree (AST) and other auxiliary structures associated with each procedure are not accessed during this phase.

Final Phase. The final phase of the analysis performs the actual placement or deletion of the routines. Each procedure is visited again, and final addition or deletion of the candidates is done.

6 Experimental Results

We now present experimental results to show the efficacy of the methods presented so far. We measure the difference made by performing interprocedural placement of both the communication preprocessing statements and the collective communication statements. We have used two irregular codes in our study, an Euler solver on an unstructured mesh [13], originally developed



V 1 : Performance before interprocedural optimizations
V 2 : Interprocedural placement of preprocessing stmts.
V 3 : Interprocedural placement of comm. stmts also

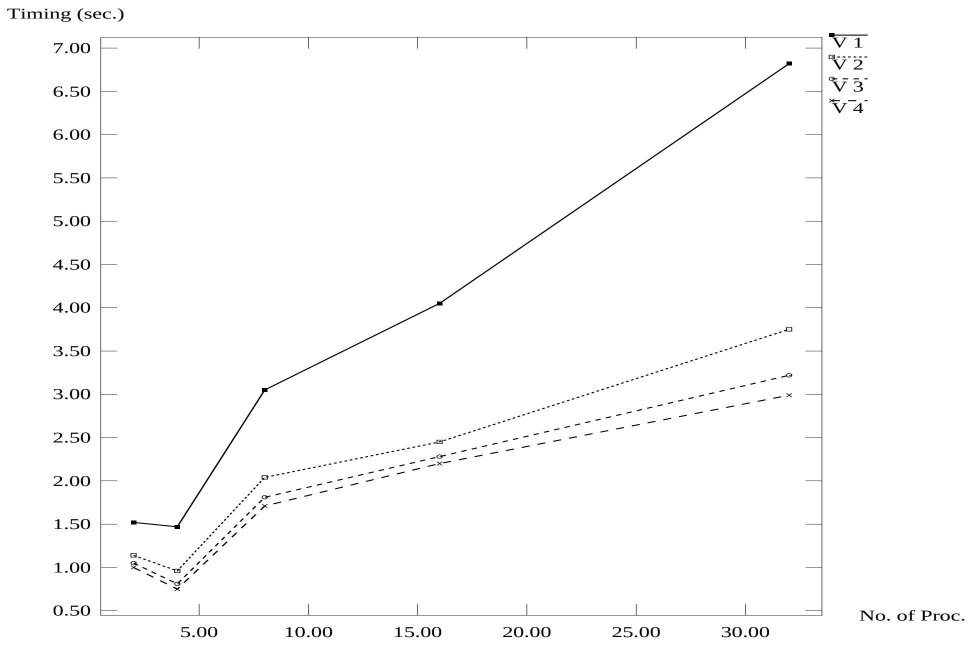
Figure 9: Effect of Optimizations on Euler solver (10K mesh, 20 iterations) on Intel Paragon.

at ICASE by Mavriplis *et al.* and a template taken from CHARMM [9], a molecular dynamics code. We used Intel Paragon at Rice University for performing our experiments.

The Euler solver we experimented with performs sweeps over an unstructured mesh inside the time step loop. The data parallel loops iterate over both the edges and the faces of the unstructured mesh. Indirection arrays are used to store the nodes corresponding to each edge and each face of the mesh. This leads to irregular accesses to data in the major computational loops of the program. The version of the code we worked with comprised of nearly 2000 lines of code across 8 procedures. We used two sets of input data in our experiments, a mesh having 53000 mesh points and 350000 edges, and another mesh having 9500 mesh points and 55000 edges.

The existing Fortran D compiler inserts appropriate communication preprocessing statements and collective communication statements in parallelizing such irregular codes, but (before the work presented here) did not perform any interprocedural placement of these statements.

In Figure 9, we show the performance difference obtained by interprocedural placements of communication preprocessing statements and communication statements. Performance of the different versions of the code is measured for 2 to 32 processors of Intel Paragon. The sequential program took 71 seconds on a single processor of the Intel Paragon. A super-linear speed up was noticed in going from one processors to two processors, we believe happens because on single

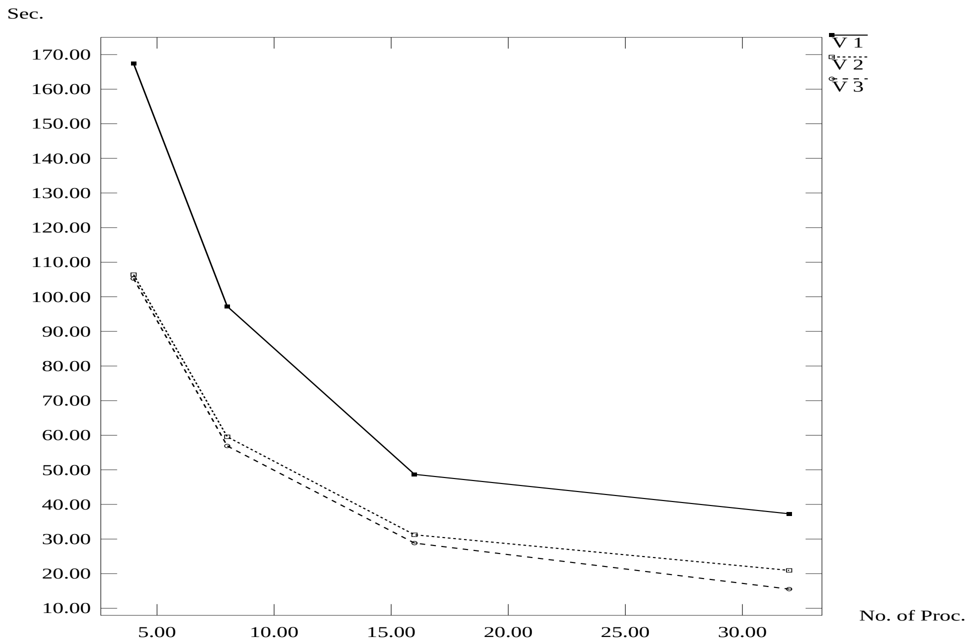


V 1 : Communication time, before optimizations
V 2 : Communication time after IPRE
V 3 : Further use of coalescing gather routines
V 4 : Further use of coalescing scatter routines

Figure 10: Effect of Communication Optimizations on Euler solver (10K mesh, 20 iterations) on Intel Paragon.

processor, all data cannot fit in the main memory of the machine. The first version (V 1) is the code which does not perform any interprocedural placement. In the second version (V 2), interprocedural placement is performed for only communication preprocessing statements. This leads to significant difference in the performance. The third version (V 3) is further optimized by various placement optimizations on communication statements, this includes applying IPRE on communication statements and the use of coalescing gather and scatter routines. On a small number of processors, the total communication time is small, and therefore, the overall performance difference due to the different communication optimizations is not significant. However, when the same data is distributed over a larger number of processors, the communication time becomes a significant part of the total execution time and the communication optimizations make significant difference in the overall performance of the program.

In Figure 10, we further study the impact of different placement optimizations on communication statements. Only the communication time is shown for the various versions of the code. The first version (V 1) does not perform any optimizations on communication statements. The second version (V 2) performs IPRE on communication statements. Figure 10 shows that this results in substantial reduction in the communication time. In the next version (V 3), coalescing



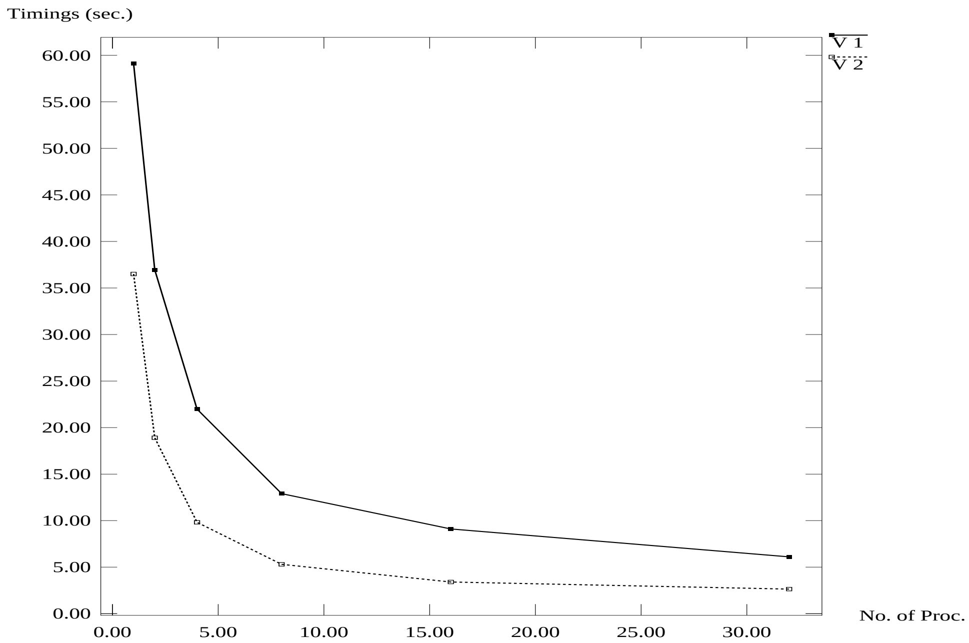
V 1 : Performance before interprocedural optimizations
V 2 : Interprocedural placement of preprocessing stmts.
V 3 : Interprocedural placement of comm. stmts also

Figure 11: Effect of Optimizations on Euler solver (53K mesh, 20 iterations) on Intel Paragon.

of gather operations is performed, this results in some more reduction in the communication time. The last version also includes coalescing of scatter operations, a marginal further reduction in communication time is noticed.

In Figure 11, we show the result of optimizations when this program is run on a larger data set, i.e. a 53000 node mesh. Interprocedural placement of communication preprocessing statements results in significant reduction in the time required by the program. When the number of processors is large, the communication time becomes significant in total execution time of the program and interprocedural optimizations on communication statements also lead to substantial improvement in the performance of the code.

The second code we considered was a template taken from a molecular dynamics code Charmm [9, 27]. The templates we worked with comprised of just 2 procedures, one procedure which computed non-bonded forces between the atoms of the molecules and the other procedure enclosed this procedure in a time step loop. Computation of non-bonded forces involed multiple levels of indirection and we used the methods described in Section 4 for performing interprocedural code motion. We used data from water molecules, which comprised of 648 atoms and nearly 100K interations between the atoms.



V 1 : Performance before interprocedural optimizations
V 2 : Interprocedural placement of preprocessing stmts.

Figure 12: Effect of Optimizations on Charmm template (20 iterations) on Intel Paragon.

In Figure 12, we show the result of optimizations. The sequential program took 34.8 seconds on the single processor of Intel Paragon. In the first version (V 1), no interprocedural placement of communication preprocessing statements is done. In the second version (V 2), placement of communication preprocessing statements is optimized interprocedurally. Since this was a relatively small template, no further improvement in performance can be achieved by interprocedural optimization of communication statements. Experiments on hand-parallelization of the entire Charmm code [27, 35] have shown a nearly 20% reduction in the communication time, by using coalescing communication routines.

7 Related Work

The only other effort on interprocedural analysis for distributed memory compilation is by Hall *et al.* [21]. They have concentrated on flow-insensitive analysis for regular applications, including management of buffer space and propagation of data distribution and data alignment information across procedure boundaries. In this work, the Augmented Call Graph (ACG) was introduced as a new program abstraction. This abstraction records any loop(s) enclosing a procedure call. Again, this abstraction does not allow to look for redundant communication preprocessing calls or communication calls in adjacent procedures.

Framework for Interprocedural Analysis and Transforms (FIAT) [20] has recently been proposed as a general environment for interprocedural analysis. This is based upon Call Graph program abstraction and is targeted more towards flow-insensitive interprocedural analysis. Our implementation uses several facilities available from FIAT as part of the Fortran D infrastructure.

Partial redundancy elimination was used interprocedurally by Gupta *et al.* [18] for performing communication optimizations. An interesting feature of their work is the available section descriptor, which facilitates many other optimizations for regular codes. Hanxleden [24] has developed Give-N-Take, a new communication placement framework. This framework extends PRE in several ways, including a notion of early and lazy problems, which is used for performing earliest possible placement of sends and latest possible placement of receive operations. Allowing such asynchronous communication can reduce communication latencies. Our work differs significantly since we consider interprocedural optimizations and present several new optimizations.

Several different program representations have been used for different flow-sensitive interprocedural problems. Myer has suggested the concept of the SuperGraph [33] which is constructed by linking control flow graphs of procedures by inserting edges from call site in the caller to start node in callee. The total number of nodes in the SuperGraph can get very large and consequently the solution may take much longer time to converge. Several ideas in the design of our representation are similar to the ideas used in Callahan's Program Summary Graph [10] and Interprocedural Flow Graph used by Soffa et al. [25].

8 Conclusions

In this paper, we have presented interprocedural optimizations for the compilation of irregular applications on distributed memory machines. In such applications, runtime preprocessing is used to determine the communication required between the processors. We have developed and used Interprocedural Partial Redundancy Elimination for optimizing placement of communication preprocessing and communication statements. We have further presented several other optimizations which are useful in the compilation of irregular applications. These optimizations include placement of scatter operations, deletion of runtime data structures and placement of incremental schedules and coalesced schedules. We have also presented how IPRE can be applied in more complex scenarios, this includes the use of slicing and ordering of the application of IPRE on different candidates.

We have carried out a preliminary implementation of the schemes presented in this paper, using the existing Fortran D compilation system as the necessary infrastructure. We have presented experimental results to demonstrate the efficacy of our schemes.

Acknowledgements

We have implemented our techniques using the existing Fortran D system. Without this essential infrastructure, this work would not have been possible. We gratefully acknowledge our debt to the implementers of the interprocedural infrastructure (FIAT) and the existing Fortran D compiler.

We are grateful to the members of the CHAOS team for providing us the library and helping us numerous times during our experiments. The authors will like to acknowledge extremely useful discussions with Bill Pugh, Anurag Acharya, Raja Das and Paul Havlak.

This work was supported by NSF under grant No. ASC 9213821, by ONR under contract No. N00014-93-1-0158, by ARPA under Scalable I/O Project (Caltech Subcontract 9503) and by NASA/ARPA contract No. NAG-1-1485. The authors assume all responsibility for the contents of the paper.

References

[1] Gagan Agrawal and Joel Saltz. Interprocedural communication optimizations for distributed memory compilation. In *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing*, pages 283–299, August 1994. Also available as University of Maryland Technical Report CS-TR-3264.

[2] Gagan Agrawal, Joel Saltz, and Raja Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 258–269. ACM Press, June 1995. ACM SIGPLAN Notices, Vol. 30, No. 6. Also available as University of Maryland Technical Report CS-TR-3446 and UMIACS-TR-95-42.

[3] Gagan Agrawal, Alan Sussman, and Joel Saltz. Compiler and runtime support for structured and block structured applications. In *Proceedings Supercomputing '93*, pages 578–587. IEEE Computer Society Press, November 1993.

[4] Gagan Agrawal, Alan Sussman, and Joel Saltz. Efficient runtime support for parallelizing block structured applications. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages 158–167. IEEE Computer Society Press, May 1994.

[5] Gagan Agrawal, Alan Sussman, and Joel Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, 1995. To appear. Also available as University of Maryland Technical Report CS-TR-3143 and UMIACS-TR-93-94.

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[7] Aart J. C. Bik and Harry A. G. Wijshoff. Annotations for a sparse compiler. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[8] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, April 1994.

[9] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dy namics calculations. *Journal of Computational Chemistry*, 4:187, 1983.

[10] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988.

[11] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and J. Saltz. Software support for irregular and loosely synchronous problems. *Computing Systems in Engineering*, 3(1–4):43–52, 1992. Papers presented at the Symposium on High-Performance Computing for Flight Vehicles, December 1992.

[12] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the rn programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[13] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives. *AIAA Journal*, 32(3):489–496, March 1994.

[14] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect access to distributed arrays. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 152–168. Springer-Verlag, August 1993. Also available as University of Maryland Technical Report CS-TR-3076 and UMIACS-TR-93-42.

[15] Raja Das, Joel Saltz, Ken Kennedy, and Paul Havlak. Index array flattening through program transformation. Submitted to PLDI '95, November 1994.

[16] D.M. Dhamdhere and H. Patil. An elimination algorithm for bidirectional data flow problems using edge placement. *ACM Transactions on Programming Languages and Systems*, 15(2):312–336, April 1993.

[17] Guy Edjlali, Gagan Agrawal, Alan Sussman, and Joel Saltz. Data parallel programming in an adaptive environment. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 827–832. IEEE Computer Society Press, April 1995.

[18] Manish Gupta, Edith Schonberg, and Harini Srinivasan. A unified data flow framework for optimizing communication. In *Proceedings of Languages and Compilers for Parallel Computing*, August 1994.

[19] Mary Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, October 1990.

[20] Mary Hall, John M Mellor Crummey, Alan Carle, and Rene G Rodriguez. FIAT: A framework for interprocedural analysis and transformations. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 522–545. Springer-Verlag, August 1993.

[21] M.W. Hall, S. Hiranandani, K. Kennedy, and C.-W. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings Supercomputing '92*, pages 522–534. IEEE Computer Society Press, November 1992.

[22] R. v. Hanxleden, K. Kennedy, and J. Saltz. Value-based distributions in Fortran D – a preliminary report. Technical Report CRPC-TR93365-S, Center for Research on Parallel Computation, Rice University, December 1993. Submitted to Journal of Programming Languages - Special Issue on Compiling and Run-Time Issues for Distributed Address Space Machines.

[23] Reinhard v. Hanxleden. Handling irregular problems with Fortran D - a preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993. Also available as CRPC Technical Report CRPC-TR93339-S.

[24] Reinhard von Hanxleden and Ken Kennedy. Give-n-take – a balanced code placement framework. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 107–120. ACM Press, June 1994. ACM SIGPLAN Notices, Vol. 29, No. 6.

[25] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[26] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[27] Yuan-Shin Hwang, Raja Das, Joel Saltz, Bernard Brooks, and Milan Hodoscek. Parallelizing molecular dynamics programs for distributed memory machines: An application of the CHAOS runtime support library. Technical Report CS-TR-3374 and UMIACS-TR-94-125, University of Maryland, Department of Computer Science and UMIACS, November 1994.

[28] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[29] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[30] V. Kotlyar, K. Pingali, and P. Stodghill. Automatic parallelization of sparse conjugate gradient code: A progress report. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[31] Antonio Lain and Prithviraj Banerjee. Exploiting spatial regularity in irregular iterative applications. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 820–826. IEEE Computer Society Press, April 1995.

[32] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[33] E. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth ACM Symposium on the Principles of Programming Languages*, pages 219–230, January 1981.

[34] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. In *Proceedings Supercomputing '93*, pages 361–370. IEEE Computer Society Press, November 1993. Also available as University of Maryland Technical Report CS-TR-3055 and UMIACS-TR-93-32.

[35] 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 *Proceedings Supercomputing '94*, pages 97–106. IEEE Computer Society Press, November 1994.

[36] Alan Sussman, Gagan Agrawal, and Joel Saltz. A manual for the multiblock PARTI runtime primitives, revision 4.1. Technical Report CS-TR-3070.1 and UMIACS-TR-93-36.1, University of Maryland, Department of Computer Science and UMIACS, December 1993.

[37] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, 1984.

[38] Janet Wu, Raja Das, Joel Saltz, Harry Berryman, and Seema Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44(6):737–753, June 1995.