Compiling Stencils in High Performance Fortran

Gerald Roth, John Mellor-Crummey,
Ken Kennedy, and R. Gregg Brickner

CRPC-TR97725-S
November 1997

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted in November 1997; From the *Proceedings of SC'97: High Performance Networking and Computing*, San Jose, CA, November 16-21, 1997

Compiling Stencils in High Performance Fortran [*]

Gerald Roth John Mellor-Crummey Ken Kennedy

Department of Computer Science
Rice University
Houston, TX 77005-1892
{roth|johnmc|ken}@cs.rice.edu

R. Gregg Brickner

Scientific Computing Group
Los Alamos National Laboratory
Los Alamos, NM 87545
rgb@lanl.gov

Abstract

For many Fortran90 and HPF programs performing dense matrix computations, the main computational portion of the program belongs to a class of kernels known as stencils. Stencil computations are commonly used in solving partial differential equations, image processing, and geometric modeling. The efficient handling of such stencils is critical for achieving high performance on distributed-memory machines. Compiling stencils into efficient code is viewed as so important that some companies have built special-purpose compilers for handling them and others have added stencil-recognizers to existing compilers.

In this paper we present a general compilation strategy for stencils written using Fortran90 array constructs. Our strategy is capable of optimizing single or multi-statement stencils and is applicable to stencils specified with shift intrinsics or with array-syntax all equally well. The strategy eliminates the need for pattern-recognition algorithms by orchestrating a set of optimizations that address the overhead of both intraprocessor and interprocessor data movement that results from the translation of Fortran90 array constructs. Our experimental results show that code produced by this strategy beats or matches the best code produced by the special-purpose compilers or pattern-recognition schemes that are known to us. In addition, our strategy produces highly optimized code in situations where the others fail, producing several orders of magnitude performance improvement, and thus provides a stencil compilation strategy that is more robust than its predecessors.

Keywords: stencil compilation, shift optimization, communication unioning, statement partitioning, High Performance Fortran

[*]From the *Proceedings of SC'97: High Performance Networking and Computing*

1 Introduction

High-Performance Fortran (HPF)[14], an extension of Fortran90, has attracted considerable attention as a promising language for writing portable parallel programs. HPF offers a simple programming model shielding programmers from the intricacies of concurrent programming and managing distributed data. Programmers express data parallelism using Fortran90 array operations and use data layout directives to direct partitioning of the data and computation among the processors of a parallel machine.

In many programs performing dense matrix computations, the main computational portion of the program belongs to a class of kernels known as stencils. For HPF to gain acceptance as a vehicle for parallel scientific programming, it must achieve high performance on this important class of problems. Compiling stencils into efficient code is viewed as so important that some companies have built special-purpose compilers for handling them [4, 5, 6] and others have added stencil-recognizers to existing HPF compilers [1, 2]. Each of these previous approaches to stencil compilation had significant limitations that restricted the types of stencils that they could handle.

In this paper, we focus on the problem of optimizing stencil computations, no matter how they are instantiated by the programmer, for execution on distributed-memory architectures. Our strategy orchestrates a set of optimizations that address the overhead of both intraprocessor and interprocessor data movement that results from the translation of Fortran90 array constructs. Additional optimizations address the issues of scalarizing array assignment statements, loop fusion, and data locality.

In the next section we briefly discuss stencil computations and their execution cost on distributed-memory machines. In Section 3 we give an overview of our compilation strategy, and then discuss the individual optimizations. In Section 4 we present an extended example to show how our strategy handles a difficult case. Experimental results are given in Section 5, and in Section 6 we compare this strategy with other known efforts.

2 Stencil Computations

In this section we introduce stencil computations and give an overview of their execution cost on distributed-memory machines. We also introduce the normalized intermediate form which our compiler uses for all stencils.

2.1 Stencils

A *stencil* is a stylized matrix computation in which a group of neighboring data elements are combined to calculate a new value. They are typically combined in the form of a sum of products. This type of computation is common in solving partial differential equations, image processing, and geometric modeling. The Fortran90 array assignment statement in Figure 1 is commonly referred to as a 5-point stencil. In this statement SRC and DST are arrays, and C1–C5 are either scalars or arrays. Each interior element of the result array DST is computed from the corresponding element of the source array SRC and the neighboring

```
 DST(2:N-1,2:N-1) = C1 * SRC(1:N-2,2:N-1)
&                 + C2 * SRC(2:N-1,1:N-2)
&                 + C3 * SRC(2:N-1,2:N-1)
&                 + C4 * SRC(3:N  ,2:N-1)
&                 + C5 * SRC(2:N-1,3:N )
```

Figure 1: 5-point stencil computation.

```
 DST = C1 * CSHIFT(CSHIFT(SRC,-1,1),-1,2)
&    + C2 * CSHIFT(SRC,-1,1)
&    + C3 * CSHIFT(CSHIFT(SRC,-1,1),+1,2)
&    + C4 * CSHIFT(SRC,-1,2)
&    + C5 * SRC
&    + C6 * CSHIFT(SRC,+1,2)
&    + C7 * CSHIFT(CSHIFT(SRC,+1,1),-1,2)
&    + C8 * CSHIFT(SRC,+1,1)
&    + C9 * CSHIFT(CSHIFT(SRC,+1,1),+1,2)
```

Figure 2: 9-point stencil computation.

```
RIP = CSHIFT(U,SHIFT=+1,DIM=1)
RIN = CSHIFT(U,SHIFT=-1,DIM=1)
T   = U + RIP + RIN
T   = T + CSHIFT(U,SHIFT=-1,DIM=2)
T   = T + CSHIFT(U,SHIFT=+1,DIM=2)
T   = T + CSHIFT(RIP,SHIFT=-1,DIM=2)
T   = T + CSHIFT(RIP,SHIFT=+1,DIM=2)
T   = T + CSHIFT(RIN,SHIFT=-1,DIM=2)
T   = T + CSHIFT(RIN,SHIFT=+1,DIM=2)
```

Figure 3: Problem 9 from the Purdue Set.

elements of SRC on the North, West, South, and East. A 9-point stencil that computes all grid elements by exploiting the CSHIFT intrinsic might be specified as shown in Figure 2.

In the previous two examples the stencils were specified as a single array assignment statement, but this need not always be the case. Consider again the 9-point stencil above. If the programmer attempted to optimize the program by hand, or if the stencil was preprocessed by other optimization phases of the compiler, we might be presented with the code shown in Figure 3[1].

A goal of our work is to generate the same, highly-optimized code for all stencil computations, regardless of how they have been written in HPF. For this reason, we have designed our optimizer to target the most general, normalized input form. All stencil and stencil-like computations can be translated into this normal form by factoring expressions and introducing temporary arrays. In fact, this is the intermediate form used by several distributed-memory compilers [18, 23, 3]. The normal form has several distinguishing characteristics:

- CSHIFT intrinsics and temporary arrays have been inserted to perform data movement needed for operations on array sections that have different processor mappings.

[1]This example was taken from Problem 9 of the Purdue Set [21] as adapted for Fortran D benchmarking by Thomas Haupt of NPAC.

```
      ALLOCATE TMP1, TMP2, TMP3, TMP4
      TMP1 = CSHIFT(SRC,SHIFT=-1,DIM=1)
      TMP2 = CSHIFT(SRC,SHIFT=-1,DIM=2)
      TMP3 = CSHIFT(SRC,SHIFT=+1,DIM=1)
      TMP4 = CSHIFT(SRC,SHIFT=+1,DIM=2)
      DST(2:N-1,2:N-1) = C1 * TMP1(2:N-1,2:N-1)
     &                 + C2 * TMP2(2:N-1,2:N-1)
     &                 + C3 * SRC (2:N-1,2:N-1)
     &                 + C4 * TMP3(2:N-1,2:N-1)
     &                 + C5 * TMP4(2:N-1,2:N-1)
      DEALLOCATE TMP1, TMP2, TMP3, TMP4
```

Figure 4: Intermediate form of 5-point stencil computation.

- Each CSHIFT intrinsic occurs as a singleton operation on the right-hand side of an array assignment statement and is only applied to whole arrays.
- The expression that actually computes the stencil operates on operands that are perfectly aligned, and thus no communication operations are required.

For example, given the 5-point stencil computation presented in Figure 1, the CM Fortran compiler would translate it into the sequence of statements shown in Figure 4.

For the rest of this paper we assume that all stencil computations have been normalized into this form, and that all arrays are distributed in a BLOCK fashion. And although we concentrate on stencils expressed using the CSHIFT intrinsic, the techniques presented can be generalized to handle the EOSHIFT intrinsic as well.

2.2 Stencil Execution

The execution of a stencil computation on a distributed-memory machine has two major components: the data movement associated with a set of CSHIFT operations and the calculation of the sum of products.

In the first phase of a stencil computation, all data movement associated with CSHIFT operations is performed. We illustrate the data movement for a single CSHIFT using an example. Figure 5 shows the effects of a CSHIFT by -1 along the second dimension of a two-dimensional BLOCK-distributed array. When a CSHIFT operation is performed on a distributed array, two major actions take place:

1. Data elements that must be shifted across processing element (*PE*) boundaries are sent to the appropriate neighboring PE. This is the interprocessor component of the shift. In Figure 5, the dashed lines represent this type of data movement, in this case the transfer of a column of data between neighboring processors.
2. Data elements shifted within a PE are copied to the appropriate locations in the destination array. This is the intraprocessor component of the shift. The solid lines in Figure 5 represent this data movement.

Following data movement, the second phase of a stencil computation is the execution of a loop nest to calculate a sum of products. The loop nest for a stencil computation is

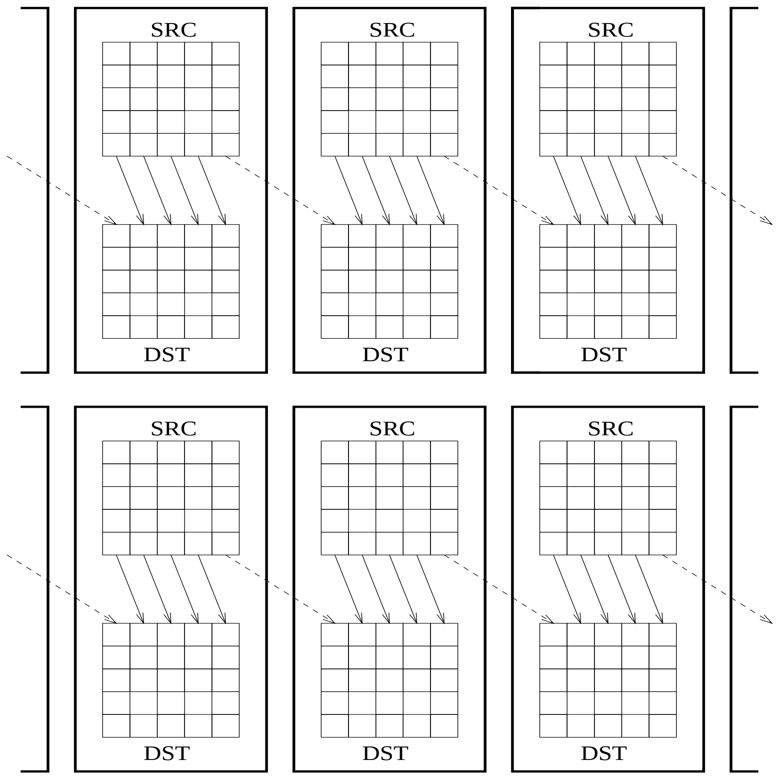

Figure 5: `DST = CSHIFT(SRC,SHIFT=-1,DIM=2)`

constructed during compilation in two steps. First the compiler applies scalarization [24] to replace Fortran 90 array operations with a serial loop nest that operates on individual data elements. Next, the compiler transforms this loop nest into SPMD code [8]. The SPMD code is synthesized by reducing the loop bounds so that each PE computes values only for the data it owns. A copy of this transformed loop nest, known as the *subgrid loop nest*, executes on each PE of the parallel machine.

Due to the nature of stencils which make many distinct array references, these subgrid loops can easily become *memory bound*. In such loops, the CPU must often sit idle while it waits for the array elements to be fetched from memory.

3 Compilation Strategy

In this section we start with an overview of our compilation strategy, and then present the individual component optimizations.

Given a stencil computation in normal form (as described in Section 2.1), we optimize it by applying a sequence of four optimizations. The first addresses the intraprocessor data movement associated with the CSHIFT operations, eliminating it when possible. The second rearranges the statements into separate blocks of computation operations and communication operations. This optimizes the stencil by promoting loop fusion for the computation operations and it prepares the communication operations for further optimization by the following phase. Next, the interprocessor data movement of the CSHIFT operations is optimized by eliminating redundant and partially-redundant communication. Finally, loop-level transformations are applied to optimize the computation.

3.1 Optimizing Intraprocessor Data Movement

Intraprocessor data movement associated with shift intrinsics is completely eliminated when possible. This is accomplished by an optimization we call *offset arrays* [15]. This optimization determines when the source array (SRC) and the destination array (DST) of the CSHIFT can share the same memory locations. If this is the case only the interprocessor data movement needs to occur. We exploit *overlap areas* [11] to receive the data that is copied between processors. After this has been accomplished, appropriate references to the destination array can be rewritten to refer to the source array with indices offset by the shift amount.

The principal challenge then is to determine when the source and destination arrays can share storage. We have established a set of criteria to determine when it is safe and profitable to create an offset array. These criteria, and an algorithm used to verify them are described in detail elsewhere [15, 22]. In general, our approach allows the source and destination arrays of a shift operation to share storage between destructive updates to either array when the shift offset is a small constant.

Once we have determined that the destination array of an assignment statement `DST = CSHIFT(SRC,SHIFT,DIM)` may be an offset array, we perform the following transformations on the code. These transformations take advantage of the data that may be shared between the source array SRC and destination array DST and move only the required data between the PEs.

First we replace the shift operation with a call to a routine that moves the off-processor data of `SRC` into an overlap area: `CALL OVERLAP_SHIFT(SRC,SHIFT,DIM)`. We then replace all uses of the array DST, that are reached from this definition, with a use of the array SRC. The newly created references to SRC carry along special annotations representing the values of SHIFT and DIM. Finally, when creating subgrid loops during the scalarization phase, we alter the subscript indices used for the offset arrays. The array subscript used for the offset reference to SRC is identical to the subscript that would have been generated for DST with the exception that the DIM-th dimension has been incremented by the SHIFT amount.

The algorithm that we have devised for verifying the criteria and for performing the above transformations is based upon the *static single assignment* (SSA) intermediate representation [9]. The algorithm, after validating the use of an offset array at a shift operation, transforms the program and propagates that information in an optimistic manner. The propagation continues until there are no more references to transform or one of the criteria has been violated. When a criterion has been violated, it may be necessary to insert an array copy statement into the program to maintain its original semantics. The inserted copy statement performs the intraprocessor data movement that was avoided with the OVERLAP_SHIFT.

Due to the offset array algorithm's optimistic nature, it is able to eliminate intraprocessor data movement associated with shift operations in many difficult situations. In particular, it can determine when offset arrays can be exploited even when their definition and uses are separated by program control flow. This allows our stencil compilation strategy to eliminate intraprocessor data movement in situations where other strategies fail.

3.2 Statement Reordering

We follow the offset array optimization with our *context partitioning* optimization [17]. This optimization partitions a set of Fortran90 statements into groups of congruent array statements[2], scalar expressions, and communication operations. This assists the compilation of stencils in the following two ways:

1. First, by grouping congruent array statements together, we ensure that as subgrid loops are generated, via scalarization and loop fusion, as much computation as possible is placed within each loop without causing the loops to be *over-fused* [22]. Loops are over-fused when the code produced for the resulting parallel loops exhibits worse performance than the code for the separate parallel loops. Also, the structure of the subgrid loops produced is very regular. These characteristics increase the chances that loop transformations performed later are successful in exploiting data reuse and data locality.

2. Second, by grouping together communication operations, we simplify the task of reducing the amount of interprocessor data movement, which we discuss in the next subsection.

To accomplish context partitioning, we use an algorithm proposed by Kennedy and McKinley [16]. While this algorithm was developed to partition parallel and serial loops into fusible groups, we use it to partition Fortran90 statements into congruence classes. The algorithm works on the *data dependence graph* (DDG)which must be acyclic. Since we apply it to a set of statements within a basic block, our dependence graph contains only *loop-independent* dependences and thus is acyclic. A complete description of our context partitioning algorithm is available elsewhere [17, 22], along with a discussion of its advantages for both SIMD and MIMD machines.

Context partitioning is key to our ability to optimize multi-statement stencils as fully as single-statement stencils. No other stencil compilation strategy has this capability.

3.3 Minimizing Interprocessor Data Movement

Once intraprocessor data movement has been eliminated and we have partitioned the statements into groups of congruent operations, we focus our attention on the interprocessor data movement that occurs during the calls to CSHIFT. Due to the nature of offset arrays, we are presented with many opportunities to eliminate redundant and partially redundant data movement. We call this optimization *communication unioning* [22], since it combines a set of communication operations to produce a smaller set of operations.

There are two key observations that allow us to find and eliminate redundant interprocessor data movement. First, shift operations, including OVERLAP_SHIFT, are commutative:

```
CSHIFT(CSHIFT(SRC,+1,1),-1,2) ≡ CSHIFT(CSHIFT(SRC,-1,2),+1,1)
```

[2]Array statements are congruent if they operate on arrays with identical distributions and cover the same iteration space.

Thus, for arrays that are shifted more than once, we can order the shift operations in any manner we like without affecting the result. Second, since all OVERLAP_SHIFTs move data into the overlap areas of the subgrids, a shift of a large amount in a given direction and dimension may subsume all shifts of smaller amounts in the same direction and dimension. More formally, an OVERLAP_SHIFT of amount i in dimension k is redundant if there exists an OVERLAP_SHIFT of amount j in dimension k such that $|j| \geq |i|$ and $sign(j) = sign(i)$. Since we have already applied our context partitioning optimization to the program, we can restrict our focus to the individual groups of calls to OVERLAP_SHIFT.

To eliminate redundant data movement using communication unioning, we first use the commutative property to rewrite all the shifts for multi-offset arrays such that the OVERLAP_SHIFTs for the lower dimensions occur first and are used as input to the OVERLAP_SHIFTs for higher dimensions. We then reorder all the calls to OVERLAP_SHIFT, sorting them by the shifted dimension, lowest to highest. We now scan the OVERLAP_SHIFTs for the lowest dimension and keep only the largest shift amount in each direction. All others can be eliminated as redundant.

Communication unioning then proceeds to process the OVERLAP_SHIFTs for each higher dimension in ascending order by performing the following three actions:

1. We scan the OVERLAP_SHIFTs for the given dimension to determine the largest shift amount in each direction.

2. We look for source arrays that are already offset arrays, indicating a multi-offset array. For these, we use the annotations associated with the source array to create an RSD to be used as an optional fourth argument in the call to OVERLAP_SHIFT. The argument indicates those data elements from the adjacent overlap areas that should also be moved during the shift operation. Mapping the annotations to the RSD is simply a matter of adding the annotations to the corresponding RSD dimension; the annotation is added to the lower bound of the RSD if the shift amount is negative, otherwise it is added to the upper bound. As with shift amounts, larger RSDs subsume smaller RSDs.

3. We generate a single OVERLAP_SHIFT in each direction, using the largest shift amount and including the RSD as needed – all other OVERLAP_SHIFTs for that dimension can be eliminated.

This procedure eliminates all redundant offset-shift communication, including partially redundant data movement associated with accessing "corner elements" of stencils.

This algorithm is unique in that it is based upon the understanding and analysis of the shift intrinsics, rather than being based upon pattern-matching as is done in many stencil compilers. This optimization eliminates all communication for a shifted array, except for a single message in each direction of each dimension. The number of messages for the stencil is thus minimized.

As an example, consider again the 9-point stencil computation that we presented in Figure 2. The original stencil specification required twelve CSHIFT intrinsics. After applying communication unioning, only the four calls to OVERLAP_SHIFT shown in Figure 6 are required.

Figures 7–10 display the data movement that results from these calls. The figures contain a 5×5 subgrid (solid lines) surrounded by its overlap area (dashed lines). Portions of the

```
CALL OVERLAP_SHIFT(SRC,-1,1)
CALL OVERLAP_SHIFT(SRC,+1,1)
CALL OVERLAP_SHIFT(SRC,-1,2,[0:N+1,*])
CALL OVERLAP_SHIFT(SRC,+1,2,[0:N+1,*])
```

Figure 6: Result of communication unioning for 9-point stencil.

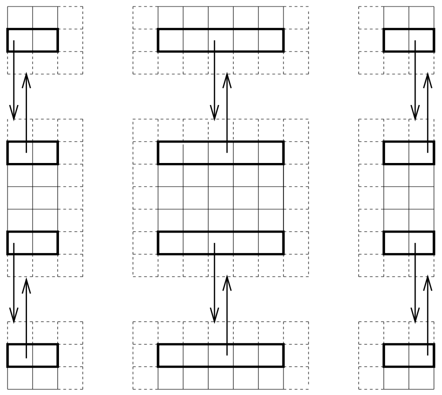

Figure 7: First half of 9-point stencil communication

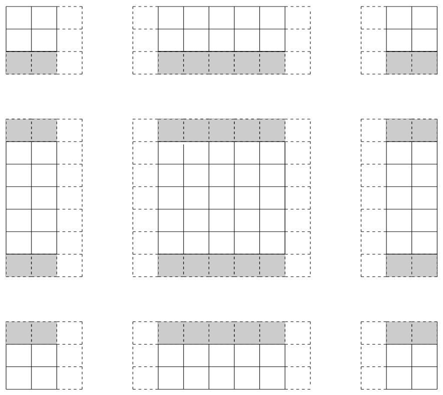

Figure 8: Result of communication operations

adjacent subgrids are also shown. Figure 7 depicts the data movement specified by the first two calls. The result of that data movement is shown in Figure 8, where the overlap areas have been properly filled in. The data movement of the last two calls is shown in Figure 9. Notice how the last two calls pick up data from the overlap areas that were filled in by the first two calls, and thus they populate all overlap area elements needed for the subsequent computation, as shown in Figure 10.

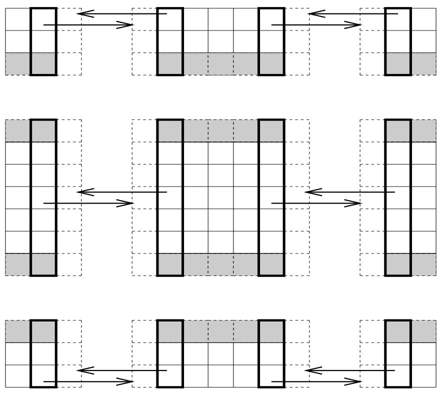

Figure 9: Second half of 9-point stencil communication

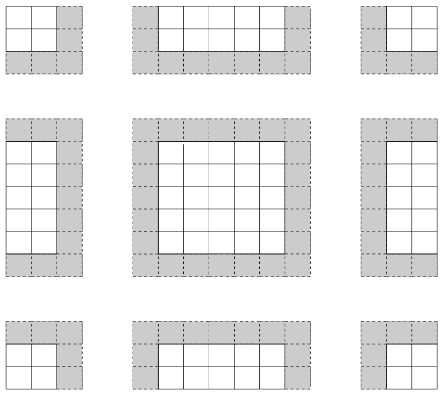

Figure 10: Result of communication operations

3.4 Optimizing the Computation

Finally, after scalarization has produced a subgrid loop nest, we can optimize it by applying a set of loop-level transformations designed to improve the performance of memory-bound programs. These transformations include unroll-and-jam, which addresses memory references, and loop permutation, which addresses cache references. Each of these optimize the program by exploiting reuse of data values. These optimizations are described in detail elsewhere [7, 19] and are not addressed in this paper.

4 An Extended Example

In this section, we trace our compilation strategy through an extended example. This detailed examination shows how our strategy is able to produce code that matches or beats hand-optimized code. It also demonstrates how we are able to handle stencil computations that cause other methods to fail.

For this exercise, we have chosen to use Problem 9 of the Purdue Set [21], as adapted for Fortran D benchmarking by Thomas Haupt of NPAC [20, 13]. The program kernel is shown in Figure 3. The arrays `T`, `U`, `RIP`, and `RIN` are all two-dimensional and have been distributed in a (BLOCK,BLOCK) fashion. This kernel computes a standard 9-point stencil, identical to that computed by the single-statement stencil shown in Figure 2. The reason it has been written in this fashion is to reduce memory requirements. Given the single-statement 9-point stencil, most Fortran90 compilers will generate 12 temporary arrays, one for each CSHIFT. This greatly restricts the size of the problem that can be solved on a given machine. In contrast, the Problem 9 specification can be computed with only 3 temporary arrays since the live-ranges of the last 6 CSHIFTs do not overlap. This reduces the temporary storage requirements by a factor of four! Additionally, the assignments of the CSHIFTs into `RIP` and `RIN` perform a common subexpression elimination, removing four duplicate CSHIFTs from the original specification of the stencil.

Figure 11 shows a comparison of execution times for the single-statement CSHIFT stencil in Figure 2 and the multi-statement Problem 9 stencil in Figure 3. The programs were compiled with IBM's xlhpf compiler and executed on a 4-processor SP-2 for varying problem sizes. As can be seen, the single-statement stencil specification exhausted the available memory for the larger problem sizes, even though each PE had 256Mbytes of real RAM.

4.1 Program Normalization

We now step through the compilation of the stencil code in Figure 3 using the strategy presented in this paper. Figure 12 shows the stencil code after normalization. The six CSHIFTs that are subexpressions in the assignment statements to array `T` are hoisted from the statements and assigned to compiler-generated temporary arrays. Since the live ranges of the temporary arrays do not overlap, a single temporary can be shared among all the statements. Alternatively, each CSHIFT could receive its own temporary array – that would not affect the results of our stencil compilation strategy.

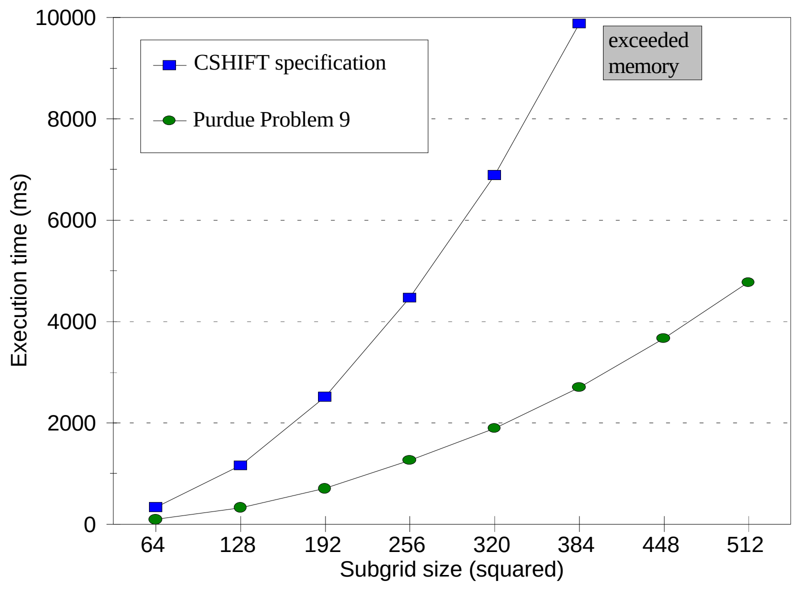

Figure 11: Comparison of two 9-point stencil specifications.

```
RIP = CSHIFT(U,SHIFT=+1,DIM=1)
RIN = CSHIFT(U,SHIFT=-1,DIM=1)
T   = U + RIP + RIN
ALLOCATE TMP
TMP = CSHIFT(U,SHIFT=-1,DIM=2)
T   = T + TMP
TMP = CSHIFT(U,SHIFT=+1,DIM=2)
T   = T + TMP
TMP = CSHIFT(RIP,SHIFT=-1,DIM=2)
T   = T + TMP
TMP = CSHIFT(RIP,SHIFT=+1,DIM=2)
T   = T + TMP
TMP = CSHIFT(RIN,SHIFT=-1,DIM=2)
T   = T + TMP
TMP = CSHIFT(RIN,SHIFT=+1,DIM=2)
T   = T + TMP
DEALLOCATE TMP
```

Figure 12: Problem 9 after normalization.

4.2 Offset Array Optimization

Once all shift operations have been identified and hoisted into their own assignment statements, we apply our offset array optimization. For this example, our algorithm determines that all the shifted arrays can be made into offset arrays. As can be seen in Figure 13, all the CSHIFT operations have been changed into OVERLAP_SHIFT operations, and references to the assigned arrays have been replaced with offset references to the source array `U`. All intraprocessor data movement has thus been eliminated.

In addition, notice how the temporary arrays, both the compiler-generated `TMP` array

```
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=1)
T = U + U^{<+1,0>} + U^{<-1,0>}
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=2)
T = T + U^{<0,-1>}
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=2)
T = T + U^{<0,+1>}
CALL OVERLAP_CSHIFT(U^{<+1,0>},SHIFT=-1,DIM=2)
T = T + U^{<+1,-1>}
CALL OVERLAP_CSHIFT(U^{<+1,0>},SHIFT=+1,DIM=2)
T = T + U^{<+1,+1>}
CALL OVERLAP_CSHIFT(U^{<-1,0>},SHIFT=-1,DIM=2)
T = T + U^{<-1,-1>}
CALL OVERLAP_CSHIFT(U^{<-1,0>},SHIFT=+1,DIM=2)
T = T + U^{<-1,+1>}
```

Figure 13: Problem 9 after offset array optimization.

```
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=2)
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=2)
CALL OVERLAP_CSHIFT(U^{<+1,0>},SHIFT=-1,DIM=2)
CALL OVERLAP_CSHIFT(U^{<+1,0>},SHIFT=+1,DIM=2)
CALL OVERLAP_CSHIFT(U^{<-1,0>},SHIFT=-1,DIM=2)
CALL OVERLAP_CSHIFT(U^{<-1,0>},SHIFT=+1,DIM=2)
T = U + U^{<+1,0>} + U^{<-1,0>}
T = T + U^{<0,-1>}
T = T + U^{<0,+1>}
T = T + U^{<+1,-1>}
T = T + U^{<+1,+1>}
T = T + U^{<-1,-1>}
T = T + U^{<-1,+1>}
```

Figure 14: Problem 9 after context partitioning optimization.

and the user-defined RIP and RIN, are no longer needed to compute the stencil. If there are no other uses of these arrays in the routine, they need not be allocated. This reduction in storage requirements allows for larger problems to be solved on a given machine.

4.3 Context Partitioning Optimization

After offset array optimization, we apply our context partitioning algorithm. This algorithm begins by determining the congruence classes present in the section of code. In this example there are only two congruence classes: the array statements, which are all congruent, and the communication statements. The dependence graph is computed next. There are only two types of dependences that exist in the code: true dependences from the OVERLAP_SHIFT operations to the expressions that use the offset arrays, and the true and anti-dependences that exist between the multiple occurrences of the array T. Since all the

```
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=2,[0:N+1,*])
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=2,[0:N+1,*])
T = U + U<+1,0> + U<-1,0>
T = T + U<0,-1>
T = T + U<0,+1>
T = T + U<+1,-1>
T = T + U<+1,+1>
T = T + U<-1,-1>
T = T + U<-1,+1>
```

Figure 15: Problem 9 after communication unioning optimization.

dependences between the two classes are from statements in the communication class to statements in the congruent array class, the context partitioning algorithm is able to partition the statements perfectly into two groups. The result is shown in Figure 14. Since the array statements are now adjacent, scalarization will be able to fuse them into a single loop nest. Similarly, the communication statements are adjacent and communication unioning will be successful at its task.

4.4 Communication Unioning Optimization

We now turn our attention to the interprocessor data movement specified in the OVERLAP_SHIFT operations. As described in Section 3.3, we first exploit the commutativity of OVERLAP_SHIFT operations and rewrite multi-dimensional OVERLAP_SHIFTs so that the lower dimensions are shifted first. No rewriting is necessary for this example since all the dimension 1 shifts occur first, as can be seen in Figure 14.

Next we look at the shifts across the first dimension. Since there is only a single shift of distance one in each direction, there is no redundant communication to eliminate. In the second dimension we again find only shifts of distance one. However, we discover four multi-offset arrays. Examining the annotations of the offset arrays, we create RSD's that summarize the overlap areas that are necessary. We generate the two calls to OVERLAP_SHIFT that include the RSD's and then eliminate all other OVERLAP_SHIFT calls for the second dimension. The resulting code is shown in Figure 15. Communication unioning has reduced the amount of communication to a minimum: a single communication operation for each dimension in each direction.

4.5 Scalarization and Memory Optimizations

Figure 16 shows the code after scalarization. The code now contains only 4 interprocessor communication operations, and no intraproceesor data movement is performed. Final transformations refine the loop bounds to generate a node program that only accesses the subgrids local to each PE. Our strategy has generated a single loop nest which, due to the nature of stencil computations, is ripe with opportunities for memory hierarchy optimization. We hand the final code to an optimizing node compiler that performs loop-level transformations such as scalar replacement and unroll-and-jam.

```
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=1)
CALL OVERLAP_CSHIFT(U,SHIFT=-1,DIM=2,[0:N+1,*])
CALL OVERLAP_CSHIFT(U,SHIFT=+1,DIM=2,[0:N+1,*])
DO i=1,N
  DO j=1,N
    T(i,j) = U(i,j) + U(i+1,j) + U(i-1,j)
    T(i,j) = T(i,j) + U(i,j-1)
    T(i,j) = T(i,j) + U(i,j+1)
    T(i,j) = T(i,j) + U(i+1,j-1)
    T(i,j) = T(i,j) + U(i+1,j+1)
    T(i,j) = T(i,j) + U(i-1,j-1)
    T(i,j) = T(i,j) + U(i-1,j+1)
  ENDDO
ENDDO
```

Figure 16: Problem 9 after scalarization.

It is important to note that our strategy also produces the exact same code when given the single-statement 9-point stencil from Figure 2. This example shows how our stencil compilation algorithm is capable of fully optimizing stencils, no matter how they are instantiated by the programmer.

5 Experimental Results

To measure the performance boost supplied by each step of our stencil compilation strategy, we ran a set of tests on a 4-processor IBM SP-2. We started by generating a naive translation of the Problem 9 test case into Fortran77+MPI. This is considered our *"original"* version. We then successively applied the transformations as outlined in the preceding section and measured the execution time. The results are shown in Figure 17.

Before analyzing the results in Figure 17, it is worthwhile to compare them to the results shown in Figure 11 for the Problem 9 code. The performance of our "original" MPI version of the code for this example is already an order of magnitude faster than the code produced by IBM's xlhpf compiler: 0.475 seconds versus 4.77 seconds for the largest problem size.

After applying our offset array optimization to the Fortran77+MPI test case as shown in Figure 13, execution time improves by 45%, equivalent to a speedup of 1.80. Next, after applying context partitioning, as shown in Figure 14, scalarization was able to merge all of the computation into a single loop nest, improving execution time an additional 31%. At this point, we have reduced the execution time of the original program by 62%, a speedup of 2.64.

As shown in Figure 15, our communication unioning optimization eliminates four communication operations, which reduces the execution time by 41% when compared to the context-optimized version. Applying memory optimizations such as scalar replacement and unroll-and-jam further reduce the execution time another 14%. The execution time of the original program has been trimmed by 81%, equivalent to a speedup of 5.19. Comparing our code to the code produced by IBM's xlhpf compiler shows a speedup by a factor of 52!

Lest someone think that we have chosen IBM's xlhpf compiler as a straw man, we have collected some additional performance numbers. We generated a third version of a 9-point

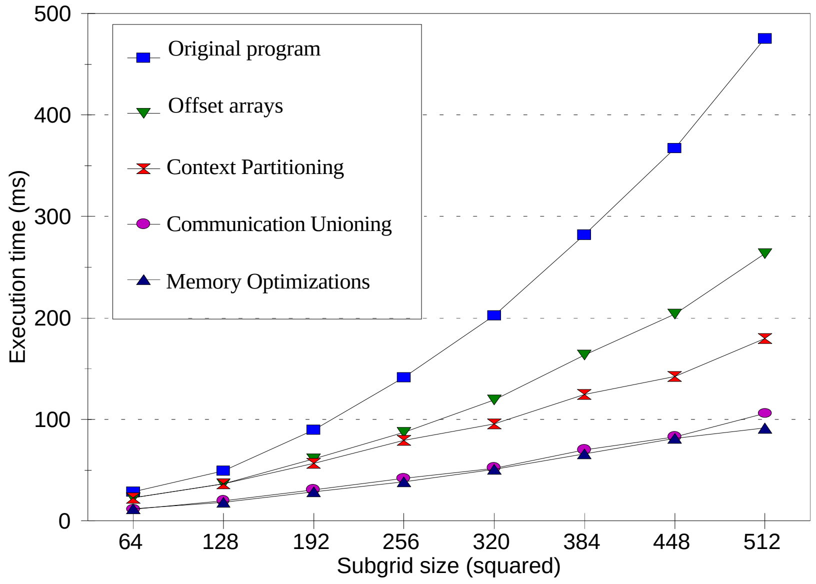

Figure 17: Step-wise results from stencil compilation strategy on Problem 9 when executed on an SP-2.

stencil computation, this one using array syntax similar to the 5-point stencil shown in Figure 1. This 9-point stencil computation only computes the interior elements of the matrix; that is, elements 2:N-1 in each dimension. A graph comparing its execution time to the other two 9-point stencil specifications is given in Figure 18. The IBM xlhpf compiler was used in all cases. It is interesting to note that for the array syntax stencil the xlhpf compiler produced performance numbers that tracked our best performance numbers for all problem sizes except the largest, where we had a 10% advantage.

It is important to note that the stencil compilation strategy that we have presented handles all three specifications of the 9-point stencil equally well. That is because our algorithm is based upon the analysis and optimization of the base constructs upon which stencils are built. Our algorithm is designed to handle the lowest common denominator – a form into which our compiler can transform all stencil computations.

6 Related Work

One of the first major efforts to specifically address the compilation of stencil computations for a distributed-memory machine was the stencil compiler for the CM-2, also known as the convolution compiler [4, 5, 6]. The compiler eliminated intraprocessor data movement and optimized the interprocessor data movement by exploiting the CM-2's polyshift communication [10]. The final computation was performed by hand-optimized library microcode that took advantage of several loop transformations and a specialized register allocation scheme.

Our general compilation methodology produces the same style code as this specialized compiler. We both eliminate intraprocessor data movement and minimize interprocessor

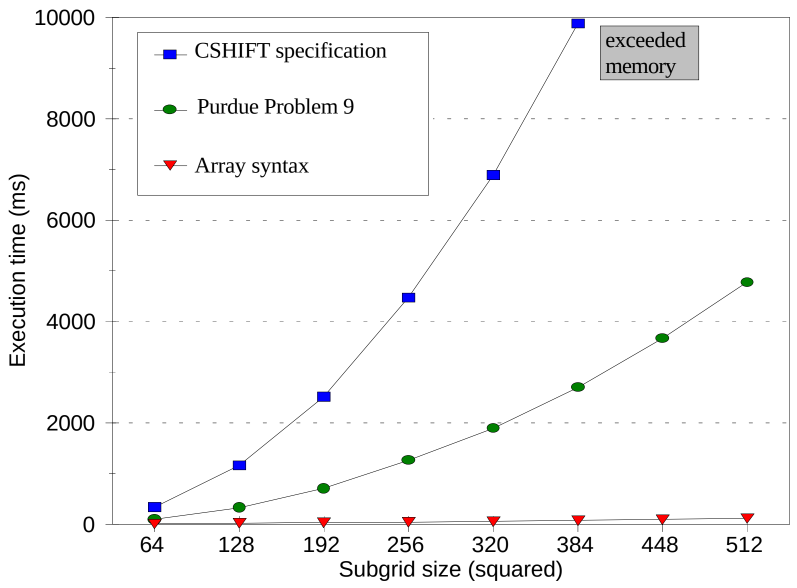

Figure 18: Comparison of three 9-point stencil specifications.

data movement. Finally, our use of a loop-level optimizer to perform the unroll-and-jam optimization accomplish the same data reuse as the stencil compiler's "*multi-stencil swath*".

The CM-2 stencil compiler had many limitations however. It could only handle single-statement stencils. The stencil had to be specified using the CSHIFT intrinsic; no array-syntax stencils would be accepted. Since the compiler relied upon pattern matching, the stencil had to be in a very specific form: a sum of terms, each of which is a coefficient multiplying a shift expression. No variations were possible. And finally, the programmer had to recognize the stencil computation, extract it from the program and place it in its own subroutine to be compiled by the stencil compiler.

Our compilation scheme handles a strict superset of patterns handled by the CM-2 stencil compiler. In their own words, they *"avoid the general problem by restricting the domain of applicability."* [6] We have placed no such restrictions upon our work. Our strategy optimizes single-statement stencils, multi-statement stencils, CSHIFT intrinsic stencils, and array-syntax stencils all equally well. And since our optimizations were designed to be incorporated into an HPF compiler, they benefit those computations that only slightly resemble stencils.

There are also some other commercially available compilers that can handle certain stylized, single-statement stencils. The MasPar Fortran compiler avoids intraprocessor data movement for single-statement stencils written using array notation. This is accomplished by scalarizing the Fortran90 expression (avoiding the generation of CSHIFTs) and then using dependence analysis to find loop-carried dependences that indicate interprocessor data movement. Only the interprocessor data is moved, and no local copying is required. However, the compiler still performs all the data movement for single-statement stencils written using SHIFT intrinsics. This strategy is shared by many Fortran90/HPF compilers that focus on handling scalarized code. As with the CM-2 stencil compiler, our methodology is a strict superset of this strategy.

Gupta, *et al.* [12], in describing IBM's xlhpf compiler, state that they are able to reduce the number of messages for multi-dimensional shifts by exploiting methods similar to ours. However, they do not describe their algorithm for accomplishing this, and it is unknown whether they would be able to eliminate the redundant communication that arises from shifts over the same dimension and direction but of different distances.

The Portland Group's pghpf compiler, as described by Bozkus, *et al.* [1, 2], performs stencil recognition and optimizes the computation by using OVERLAP_SHIFT communication. They also perform a subset of our communication unioning optimization. However, they are limited to single-statement expressions in both cases.

In general, there have been several different methods for handling specific forms of stencil computations. Our strategy handles a more general form of stencil computations than these earlier methods.

7 Conclusion

In this paper, we presented a general compilation scheme for compiling HPF stencil computations for distributed-memory architectures. The strategy optimizes such computations by orchestrating a unique set of optimizations. These optimizations eliminate unnecessary intraprocessor data movement resulting from CSHIFT intrinsics, rearrange the array statements to promote profitable loop-fusion, eliminate redundant interprocessor data movement, and optimize memory accesses via loop-level transformations. The optimizations are general enough to be included in a general-purpose HPF/Fortran90 compiler as they will benefit many computations, not just those that fit a stencil pattern.

The strength of these optimizations is that they operate on a normal form into which all stencil computations can readily be translated. This enables us to optimize all stencil computations regardless of whether they are written using array syntax or explicit shift intrinsics, or whether the stencil is computed by a single statement or multiple statements. This approach is significantly more general than stencil compilation approaches in previous compilers. Even though we focused on the compilation of stencils for distributed-memory machines in this paper, the techniques presented are equally applicable to optimizing stencil computations on shared-memory and scalar machines (with the exception of reducing interprocessor movement).

Acknowledgments

This work has been supported in part by the IBM Corporation, the Center for Research on Parallel Computation (an NSF Science and Technology Center), and DARPA Contract DABT63-92-C-0038. This work was also supported in part by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0159. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency and Rome Laboratory or the U.S. Government.

References

[1] Z. Bozkus, L. Meadows, D. Miles, S. Nakamoto, V. Schuster, and M. Young. Techniques for compiling and executing HPF programs on shared-memory and distributed-memory parallel systems. In *Proceedings of the First International Workshop on Parallel Processing*, Bangalore, India, December 1994.

[2] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. PGHPF – an optimizing High Performance Fortran compiler for distributed memory machines. *Scientific Programming*, 6(1):29–40, 1997.

[3] T. Brandes. Compiling data parallel programs to message passing programs for massively parallel MIMD systems. In *Working Conference on Massively Parallel Programming Models*, Berlin, 1993.

[4] R. G. Brickner, W. George, S. L. Johnsson, and A. Ruttenberg. A stencil compiler for the Connection Machine models CM-2/200. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[5] R. G. Brickner, K. Holian, B. Thiagarajan, and S. L. Johnsson. A stencil compiler for the Connection Machine model CM-5. Technical Report CRPC-TR94457, Center for Research on Parallel Computation, Rice University, June 1994.

[6] M. Bromley, S. Heller, T. McNerney, and G. Steele, Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[7] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, October 1994.

[8] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and C.-W. Tseng. Compiling Fortran 77D and 90D for MIMD distributed-memory machines. In *Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[9] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[10] W. George, R. Brickner, and S. L. Johnsson. Polyshift communications software for the Connection Machine systems CM-2 and CM-200. *Scientific Programming*, 3(1):83, Spring 1994.

[11] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.

[12] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[13] T. Haupt, S. Reddy, and G. Vengurlekar. Low level HPF compiler benchmark suite. Technical Report SCCS-735, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY, August 1995.

[14] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[15] K. Kennedy, J. Mellor-Crummey, and G. Roth. Optimizing Fortran 90 shift operations on distributed-memory multicomputers. In *Languages and Compilers for Parallel Computing, Eighth International Workshop*, Columbus, OH, August 1995. Springer-Verlag.

[16] K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, August 1993.

[17] K. Kennedy and G. Roth. Context optimization for SIMD execution. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, Knoxville, TN, May 1994.

[18] K. Knobe, J. Lukas, and M. Weiss. Optimization techniques for SIMD Fortran compilers. *Concurrency: Practice and Experience*, 5(7):527–552, October 1993.

[19] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[20] A. Mohamed, G. Fox, G. v. Laszewski, M. Parashar, T. Haupt, K. Mills, Y. Lu, N. Lin, and N. Yeh. Applications benchmark set for Fortran-D and High Performance Fortran. Technical Report SCCS-327, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY, June 1992.

[21] J. R. Rice and J. Jing. Problems to test parallel and vector languages. Technical Report CSD-TR-1016, Dept. of Computer Science, Purdue University, 1990.

[22] G. Roth. *Optimizing Fortran90D/HPF for Distributed-Memory Computers*. PhD thesis, Dept. of Computer Science, Rice University, April 1997.

[23] G. Sabot. A compiler for a massively parallel distributed memory MIMD computer. In *Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[24] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.