Loop Fusion in High Performance Fortran *

Gerald Roth

Sun Microsystems
901 San Antonio Rd.
MS UMPK16-303
Palo Alto, CA 94303
jerry.roth@eng.sun.com

Ken Kennedy

Rice University
Computer Science Dept.
6100 Main Street
Houston, TX 77005
ken@cs.rice.edu

Abstract

In this paper we investigate a unique problem associated with fusing loops within a High Performance Fortran (HPF) program. In particular, we discuss the issue of performing loop fusion in an HPF compiler when compiling Fortran90 array assignment statements for execution on a distributed-memory machine. During compilation of an HPF program, Fortran90 array assignment statements must be scalarized into loop nests. We show how a certain class of these loop nests, when fused, can cause problems for the compiler's distributed-memory code generator. We then present an algorithm which not only prevents the fusion of these loops, but also increases the amount of useful fusion that can be performed.

1 Introduction

High-Performance Fortran (HPF)[12], 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.

One transformation an HPF compiler must address is the scalarization of the Fortran90 array operations into serial DO-loops. Scalarization is often followed by loop fusion in an attempt to improve a program's data locality and data reuse characteristics. Unfortunately, the fusion of some of these scalarized loops can produce loops for which it is difficult to generate an efficient SPMD program. However, loop fusion is too important of an optimization to simply disable. To solve this dilemma, we have developed an algorithm which not only prevents inappropriate fusion but also enhances the amount of useful fusion.

In the next section we briefly discuss the scalarization of Fortran90 array operations and the fusion of the resulting loops. In Section 3 we discuss some problems that may be encountered when generating SPMD code for some fused loops. We present our solution to these problems in Section 4. Experimental results are given in Section 5, and in Section 6 we discuss related works.

2 Scalarization and Loop Fusion

At some point during the compilation of an HPF program, Fortran90 array assignment statements must be translated into serial DO-loops. This process is known as *scalarization* [4, 24, 31]. The transformation replaces each array assignment statement with a loop nest containing a single assignment statement in which all array references contain only scalar subscripts.

As an example, the array assignment statement

```
X(1:256) = X(1:256) + 1.0
```

would be translated into the following loop

```
DO I = 1, 256
   X(I) = X(I) + 1.0
ENDDO
```

which iterates over the specified 256 elements of the array `X`.

Unfortunately, the naive translation of array statements into serial loops is not always safe. The Fortran90 semantics for an array assignment statement specify that all right-hand side array elements are read before any left-hand side array elements are stored. Thus a naive translation of

```
X(2:255) = X(1:254) + X(2:255)
```

into the following loop nest

```
DO I = 2, 255
   X(I) = X(I-1) + X(I)
ENDDO
```

is incorrect, since on the second and subsequent iterations of the `I` loop the reference `X(I-1)` accesses the new values of the array `X` assigned on the previous iteration.

Fortunately, data dependence information can tell us when the scalarized loop is correct. Allen and Kennedy [4] have shown that a scalarized loop is correct if and only if it does not carry a true dependence. Using this fact, most advanced compilers perform scalarization by computing data

*This work was performed while the first author was a graduate student at Rice University, and was supported in part by the IBM Corporation and by the Center for Research on Parallel Computation, an NSF Science and Technology Center.

From the *Proceedings of the ACM International Conference on Supercomputing* (ICS'98), July 1998, Melbourne, Australia.

dependences and then performing code transformations to either eliminate loop-carried true dependences or change them into antidependences.

The code transformations that can be applied to handle the loop carried true dependences include loop reversal, loop interchange, prefetching, and as a last resort the generation of array temporaries and copy loops. In the example above, loop reversal can be used to change the loop-carried true dependence into a loop-carried antidependence, thus creating a valid scalarization. The interested reader is referred to Allen and Kennedy [4] for a complete discussion.

After scalarization, a program will consist of many loop nests, each containing a single assignment statement. If the goal of a Fortran90/HPF compiler is to produce code for array expressions that is competitive with code produce for scalar programs by Fortran77 compilers, it is critical that the Fortran90/HPF compiler do a good job of fusing these loops when possible. *Loop fusion* [5, 32] not only reduces the total amount of loop overhead, but more importantly it can significantly increase the possibility of data reuse in a program. The importance of loop fusion in the compilation of Fortran90 array statements cannot be over emphasized [20].

In previous work on loop fusion for parallel machines [7, 30, 32] two adjacent loops are candidates for fusion if their headers are *conformable* and there do not exist any *fusion-preventing dependences*. Two loop headers are conformable if they specify the same number of iterations and both loops are either parallel or sequential. A data dependence between two loops is fusion-preventing if after fusion the dependence becomes loop-carried and its direction is reversed [1, 30].

By using loop fusion, in conjunction with other transformations such as statement substitution and array contraction, it is possible for a Fortran90/HPF compiler to generate code for a block of array assignments that is equivalent to the code produced by a Fortran77 compiler for a corresponding hand-coded loop nest.

3 Over Fusing Loops

While the criteria presented in the previous section determine the safety of loop fusion and attempt to address its profitability on parallel machines, they are insufficient when considering loop fusion on distributed-memory architectures. This is due to the fact that they ignore the distribution and alignment of the arrays accessed within the loops, as well as the exact iteration space of the loops.

When the distribution and alignment of the arrays and the iteration space of the loops is not considered, it is possible for loops to be *over fused*. Loops are over fused when the code produced for the resulting parallel loops exhibits worse performance than the code for the separate parallel loops. The poorer performance of the fused loops is not a result of register pressure or overflowing the instruction cache, which are common concerns when fusing loops. But rather the performance degradation is a result of the complications encountered when generating SPMD code for certain types of loops.

As an example, consider the simplified program fragment shown in Figure 1(a) from the SHALLOW weather prediction benchmark code[1]. In this example all the arrays are perfectly aligned and distributed. The scalarized loop nests generated for these two statements are shown in Figure 1(b). Figure 1(c) shows how the SPMD code generator could use

[1] This example taken from Tseng [29].

```
REAL, ARRAY(256,256) :: X,Y,Z
!HPF$ DISTRIBUTE(BLOCK,*)::X,Y,Z

X(2:256,1:255) = F1(Z(2:256,2:256),Z(2:256,1:255))
Y(1:255,2:256) = F2(Z(2:256,2:256),Z(1:255,2:256))
```

(a) SHALLOW weather prediction code

```
REAL, ARRAY(256,256) :: X,Y,Z
!HPF$ DISTRIBUTE(BLOCK,*)::X,Y,Z

DO j=1,255
 DO i=2,256
  X(i,j) = F1(Z(i,j+1),Z(i,j))
 ENDDO
ENDDO
DO j=2,256
 DO i=1,255
  Y(i,j) = F1(Z(i+1,j),Z(i,j))
 ENDDO
ENDDO
```

(b) After scalarization

```
REAL, ARRAY(64,256) :: X,Y,Z

lb = max(mypid*64,65)-(mypid*64)+1
DO j=1,255
 DO i=lb,64
  X(i,j) = F1(Z(i,j+1),Z(i,j))
 ENDDO
ENDDO
ub = min(mypid*64,255)-64*(mypid-1)
DO j=2,256
 DO i=1,ub
  Y(i,j) = F1(Z(i+1,j),Z(i,j))
 ENDDO
ENDDO
```

(c) SPMD code for separate loops

```
REAL, ARRAY(256,256) :: X,Y,Z
!HPF$ DISTRIBUTE(BLOCK,*)::X,Y,Z

DO j=1,255
 DO i=1,255
  X(i+1,j) = F1(Z(i+1,j+1),Z(i+1,j))
  Y(i,j+1) = F1(Z(i+1,j+1),Z(i,j+1))
 ENDDO
ENDDO
```

(d) After scalarization and loop fusion

```
REAL, ARRAY(64,256) :: X,Y,Z

lb = max(mypid*64,65)-(mypid*64)
ub = min(mypid*64,255)-64*(mypid-1)
DO j=1,255
 DO i=lb,ub
  IF (i.lt.64) THEN
   X(i+1,j) = F1(Z(i+1,j+1),Z(i+1,j))
  ENDIF
  IF (i.gt.0) THEN
   Y(i,j+1) = F1(Z(i+1,j+1),Z(i,j+1))
  ENDIF
 ENDDO
ENDDO
```

(e) SPMD code for fused loops

Figure 1: Example of over fusing loops due to different iteration spaces.

loop bounds reduction [29] to generate an efficient node program for a four processor machine. If on the other hand the compiler had fused the two loops prior to SPMD code generation, as shown in Figure 1(d), the compiler can no longer use loop bounds reduction to instantiate the computation partition. This is due to the fact that each processor does not process the same number of elements for each statement. Instead the compiler inserts guards around each statement, as is seen in Figure 1(e). This is less efficient since the guard statements must be evaluated at run-time, once per iteration of the loop.

This example shows why it is important to consider the exact iteration space of the loops, rather than just the iteration count, when fusing for distributed-memory architectures. We note that in this simple example, it is true that *loop peeling* [5] could have been used during SPMD code generation to avoid generating the guard statements in the fused loops. However, more complicated examples exists in which avoiding loop fusion is preferred due to the multiple levels of peeling which are required.

As another example, consider the sample code in Figure 2. In Figure 2(a) we see two simple Fortran90 array assignment statements which operate on arrays with identical iteration spaces, but have different distributions. In Figure 2(b) we see the loops that would result from scalarization. The Rice dHPF compiler [2] generates the code shown in Figure 2(c) for the two loops from Figure 2(b) when targeting a four processor machine. Since the scalarized loops have the same number of iterations and do not have any fusion-preventing dependences, many Fortran90 compilers that support loop fusion would fuse the two loops into a single loop, such as seen in Figure 2(d). But even though the loops have the same iteration space, the arrays upon which they operate have different distributions. This causes problems for the SPMD code generator of an HPF compiler. As of March 1998, the Rice dHPF compiler, which uses a sophisticated code generation algorithm based on the manipulation of integer sets [3], produces the code seen in Figure 2(e) when given the fused loops as input. Notice that three loops were generated, and that the second loop contains a conditional and an expensive `IMOD` function. This does not demonstrate a short-coming of the dHPF code generator, but is instead indicative of the challenge of generating efficient code for loop nests which reference arrays with different distributions.

4 Context Partitioning

To handle the fused loops presented in the previous section, Tseng [29] proposes using loop distribution to avoid the complications of code generation. But this is actually doing double work: one phase of the HPF compiler fuses loops only to have a later phase distribute them. The proper resolution is to avoid such loop fusion in the first place. However, one cannot just simply disable loop fusion, since it is critical for obtaining good performance for Fortran90 array statements [20], even when targeting a distributed-memory architecture.

Others may claim that this problem can be avoided by delaying loop fusion until after loop bounds have been reduced and SPMD code has been produced. This is dubious at best. The generation of SPMD code often requires the creation of symbolic loop bounds which are defined by calls to run-time library routines. This obfuscation of the code makes it extremely difficult to detect conformable loop headers at compile-time.

```
REAL, ARRAY(100) :: A,B
!HPF$ DISTRIBUTE(BLOCK)::A
!HPF$ DISTRIBUTE(CYCLIC)::B

A(:) = A(:) + 1
B(:) = B(:) + 1
```

(a) Original program

```
REAL, ARRAY(100) :: A,B
!HPF$ DISTRIBUTE(BLOCK)::A
!HPF$ DISTRIBUTE(CYCLIC)::B

DO i=1,100
  A(i) = A(i) + 1
ENDDO
DO i=1,100
  B(i) = B(i) + 1
ENDDO
```

(b) After scalarization

```
REAL, ARRAY(25) :: A,B

DO i=25*mypid-24,25*mypid
  A(i-25*(mypid-1)) = A(i-25*(mypid-1)) + 1
ENDDO
DO i=mypid,mypid+96,4
  B((i+3)/4) = B((i+3)/4) + 1
ENDDO
```

(c) SPMD code generated from separate loops

```
REAL, ARRAY(100) :: A,B
!HPF$ DISTRIBUTE(BLOCK)::A
!HPF$ DISTRIBUTE(CYCLIC)::B

DO i=1,100
  A(i) = A(i) + 1
  B(i) = B(i) + 1
ENDDO
```

(d) After scalarization and loop fusion

```
REAL, ARRAY(25) :: A,B

DO i=mypid,25*mypid-28,4
  B((i+3)/4) = B((i+3)/4) + 1
ENDDO
DO i=25*mypid-24,25*mypid
  A(i-25*(mypid-1)) = A(i-25*(mypid-1)) + 1
  IF (imod(i-mypid,4).eq.0) THEN
    B((i+3)/4) = B((i+3)/4) + 1
  ENDIF
ENDDO
DO i=25*mypid+4,mypid+96,4
  B((i+3)/4) = B((i+3)/4) + 1
ENDDO
```

(e) SPMD code generated from fused loops

Figure 2: Example of over fusing loops due to different array distributions.

For portions of HPF programs written with Fortran77 syntax, loop fusion is not an important optimization since Fortran programmers typically put as much computation into each loop nest as possible. But as noted earlier, loop fusion is extremely important when compiling Fortran90 array statements. Thus this paper concentrates on loop fusion of scalarized loop nests.

To summarize, an HPF compiler needs to fuse scalarized loop nests when possible, and it can perform that fusion either before SPMD code generation or afterward. If performed prior to SPMD code generation it is possible for the loops to be over fused. If performed after SPMD code generation then it is likely to have limited success.

The challenge then is to perform useful loop fusion of scalarized loop nests, while at the same time preventing loops from being over fused. We present an algorithm, called *context partitioning*, that solves this exact problem. Not only does it solve this problem, but it is also capable of enhancing the amount of useful loop fusion performed by rearranging program statements. The algorithm makes fusion decisions prior to SPMD code generation, but does so while considering the distribution and alignment of the arrays being operated upon. Before describing the algorithm, we give a brief description of our HPF compilation model.

4.1 Compilation Model

During the compilation of an HPF program, the compiler is responsible for inserting any necessary communication operations. These are required to move data so that all operands of an expression reside on the processor which performs the computation. For scalar array references, the compiler generates individual SEND and RECEIVE pairs for non-local data accesses, and then depends upon later compilation phases to optimize them [29].

However, Fortran90 array constructs supply additional information which can allow a compiler to directly recognize and exploit collective communication routines. Examples of collective communication routines include CSHIFT and TRANSPOSE. These routines have several features which provide compelling reasons for an HPF compiler to exploit them; in particular they provide high performance from specially tuned library routines, they allow the composition of operations to handle complex communication patterns, and they simplify the complexity of the compiler.

For these reasons an HPF compiler should exploit collective communication routines whenever possible. This requires the compiler to recognize applicable patterns in the array syntax used in assignment statements. Our compiler uses a variant of the pattern matching techniques proposed by Li and Chen [19]. This requires an analysis of the array subscripts that are used, in conjunction with information about the array's alignment and distribution.

After the communication operations have been inserted, all computations reference data that are strictly local to the associated processors. For example, the array assignment statement:

```
X(2:255) = X(1:254) + X(2:255) + X(3:256)
```

would be changed into the following three statements, where TMP1 and TMP2 are arrays that match the size and distribution of X:

```
TMP1 = CSHIFT(X,SHIFT=-1,DIM=1)
TMP2 = CSHIFT(X,SHIFT=+1,DIM=1)
X(2:255) = TMP1(2:255) + X(2:255) + TMP2(2:255)
```

Notice that in the third statement all the operands are "perfectly aligned" with each other and that there is no further communication required to compute the expression or store the result. This code is equivalent to the code produced by several other commercial and research compilers [6, 17, 26].

Such perfectly aligned array statements have two distinct advantageous characteristics. First, this property ensures that a naive scalarization of array assignment statements is always correct. Any possible loop-carried true dependences now exist only in the temporary arrays used by the communication operations.

Second, no fusion-preventing dependences can exist between adjacent scalarized loops. Any fusion-preventing dependence that existed prior to communication generation is now carried through a communication operation and its compiler temporary. This communication operation prevents the Fortran90 array expressions (and their corresponding scalarized loops) from becoming adjacent and are thus not considered for loop fusion.

For example, given the following two array assignments

```
X(2:255) = X(2:255) + A(2:255)
B(2:255) = X(1:254) + B(2:255) + X(3:256)
```

communication generation would result in

```
X(2:255) = X(2:255) + A(2:255)
TMP1 = CSHIFT(X,SHIFT=-1,DIM=1)
TMP2 = CSHIFT(X,SHIFT=+1,DIM=1)
B(2:255) = TMP1(2:255) + B(2:255) + TMP2(2:255)
```

in which the fusion-preventing dependences are now carried by the temporary arrays. The definitions of the temporary arrays prevent the scalarized loops for the array assignments from becoming adjacent.

Due to these two characteristics of perfectly aligned data within array operations, our compiler can directly generate a single loop nest for adjacent Fortran90 array statements if they have identical distributions and cover the same iteration space. We call such array statements *congruent* [2]. This form of scalarization precludes the need for loop fusion. For example, when presented with the following array statements

```
X(1:256) = X(1:256) + 1.0
A(1:256) = X(1:256) ** 2.0
B(1:256) = X(1:256) + A(1:256) + B(1:256)
```

our compiler would directly generate the following loop during scalarization.

```
DO I = 1, 256
   X(I) = X(I) + 1.0
   A(I) = X(I) ** 2.0
   B(I) = X(I) + A(I) + B(I)
ENDDO
```

The compiler is able to determine if array statements are congruent by analyzing the array subscripts along with the declared distribution and alignment of the arrays. This information is all available at compile time. It is not necessary for the compiler to know the exact computation partitioning that will be used at run time. It is sufficient to only know that the statements will receive the same computation partitioning.

4.2 Context Partitioning Algorithm

As explained above, our compilation model generates a single loop nest for adjacent Fortran90 array statements which

[2] Congruence is a stronger restriction than conformance, which just considers shape (rank) and size (extent).

are congruent. This eliminates the need for a separate loop fusion optimization and prevents the problems of over fusing loops. However, unless an effort is made to make congruent array statements adjacent, many small loops may still be generated. In order to alleviate this problem, our compiler uses the following context partitioning algorithm to reorder statements within a basic block. The reordering attempts to create separate partitions of congruent array statements, scalar statements, and communication statements.

To accomplish context partitioning, we use an algorithm proposed by Kennedy and McKinley [15]. While they were concerned with partitioning parallel and serial loops into fusible groups, we are partitioning Fortran90 statements into congruence classes. The algorithm works on the *data dependence graph* (DDG) which must be acyclic. Since we apply the algorithm to a set of statements within a basic block, our dependence graph contains only loop-independent dependences and thus meets that criterion. Besides the DDG, the algorithm takes two other arguments: the set of congruence classes contained in the DDG, and a priority ordering of the congruence classes. We create a congruence class for each set of congruent array statements and then add separate classes for scalar statements and communication statements.

The context partitioning algorithm is shown in Figure 3. For each congruence class in priority order, the algorithm makes a pass over the DDG. During the pass it greedily merges statements for the given class. Two statements from the same class may be merged into the same partition if there does not exist a bad path between them. A *bad path* for a congruence class c is defined to be a path that begins with a statement from class c and either contains a fusion-preventing edge between two statements from class c, or a statement from a class different than c. Due to our compilation model presented in the preceding subsection, all fusion-preventing dependences are carried by temporary arrays defined by communication operations. Since communication statements are in their own congruence class, it is sufficient to define a bad path as a path containing a statement from a different class. When two statements are merged into a partition, the DDG is updated to reflect it.

The strength of the algorithm comes in its ability to choose in constant time the correct partition x with which to merge the given statement n of the selected congruency class. Intuitively, n can merge with any of its ancestors from which there does not exist a bad path, but it cannot bypass any predecessor to merge with an ancestor. It can also merge with partitions which are neither ancestors nor descendants. As statements are processed for a given class, the algorithm determines the highest numbered partition m of class c from which there exists a bad path to n. Statement n cannot merge with m or any partition with a lower number, where partitions are numbered breadth-first in the congruency class.

To compute the partition with which n is to be merged, the algorithm takes the number of the next partition of class c that occurs after partition m, the highest numbered partition for which a bad path exists. If this number is 0, then n is placed in a new partition and is given the next unassigned number. Alternatively, n is merged with the lowest-numbered partition of class c from which there is no bad path to n.

Assigning a priority ordering to the congruence classes is required to handle class conflicts. A *class conflict* occurs when there exist dependences such that a pair of statements from one class may be merged during partitioning or a pair

---

Procedure *Context_Partitioning*

Input:
Stmts, the set of statements to be partitioned.
Classes, the set of congruence classes.
Priority, the priority ordering for *Classes*.
DDG, the data dependence graph for *Stmts*.

Output:
A linear list of *Stmts* that has been partitioned.

Intermediate:
$num(n)$, the number of the first visit to node n of class c.
lastnum, the most recently assigned number.
$maxBadPrev(n)$ is $max(num(x))$|class of x is c and $\exists$ a bad path for class c from x to n).
$node(i)$, an array that maps numbers to nodes such that $node(num(x)) = x$.
visited, the number of the first node of class c in the *DDG*.
$next(i)$, maps the i^{th} node to the number of the next node of the same class.

Algorithm:

```
for each c ∈ Classes in decreasing priority do
  /* Initialization */
  lastnum = 0; count(*) = 0; visited = 0; node(*) = 0;
  for each edge e = (m, n) ∈ DDG do count(n) = count(n) + 1;
  for each node n ∈ DDG do
    maxBadPrev(n) = 0; num(n) = 0; next(n) = 0;
    if count(n) is 0 then add n to Worklist
  endfor
  /* Iterate over Worklist, visiting nodes and */
  /* fusing nodes from class c */
  while Worklist ≠ ∅ do
    remove arbitrary node n from Worklist
    if class(n) = c then
      /* Compute node to fuse with. If none, */
      /* assign a new number and add to visited. */
      if maxBadPrev(n) = 0 then
        p = visited
      else
        p = next(maxBadPrev(n))
      endif
      if p ≠ 0 then
        x = node(p)
        num(n) = num(x)
        maxBadPrev(n) = max(maxBadPrev(n), maxBadPrev(x))
        fuse x and n, and call the result n
      else /* a new node */
        lastnum = lastnum + 1
        num(n) = lastnum
        node(num(n)) = n
        /* append node n to end of visited */
        if lastnum = 1 then visited = lastnum
        else next(lastnum − 1) = num(n) endif
      endif
    endif
    /* Update maxBadPrev for successors of n and add to */
    /* Worklist as appropriate */
    for each edge (n,m) ∈ DDG do
      count(m) = count(m) − 1
      if count(m) = 0 then add m to Worklist endif
      if class(n) ≠ c then
        maxBadPrev(m) = max(maxBadPrev(m), maxBadPrev(n))
      else /* class(n) = c */
        if class(m) = c then
          maxBadPrev(m) = max(maxBadPrev(m), maxBadPrev(n))
        else /* different class */
          maxBadPrev(m) = max(maxBadPrev(m), num(n))
        endif
      endif
    endfor
  endwhile
endfor
topologically sort DDG to produce the final order of Stmts
end Context_Partitioning
```

Figure 3: The Context Partitioning algorithm.

```
1_A :  X(1:256) = X(1:256) + 1.0
2_B :  Z(1:100) = Z(1:100) + W(1:100)
3_A :  V(1:256) = X(1:256) ** 2
4_B :  W(1:100) = X(1:100) + Z(1:100)
5_B :  U(1:100) = Z(1:100) + SCALAR1
6_S :  SCALAR1  = SCALAR1 * V(I)
7_A :  X(1:256) = W(1:256) * SCALAR1
8_B :  Z(1:100) = SQRT(W(1:100))
```

(a) Source code

```
1_A :  X(1:256) = X(1:256) + 1.0
3_A :  V(1:256) = X(1:256) ** 2
2_B :  Z(1:100) = Z(1:100) + W(1:100)
4_B :  W(1:100) = X(1:100) + Z(1:100)
5_B :  U(1:100) = Z(1:100) + SCALAR1
8_B :  Z(1:100) = SQRT(W(1:100))
6_S :  SCALAR1  = SCALAR1 * V(I)
7_A :  X(1:256) = W(1:256) * SCALAR1
```

(b) Modified code

Figure 4: Context partitioning example.

from another class, but not both since that would introduce a cycle in the DDG and thus make it unschedulable. The priority ordering is used to determine which pair should be merged. The algorithm merges pairs with a higher priority before those with a lower priority. Choosing an optimal ordering of classes is NP-hard in the number of classes. However, since class conflicts are considered rare, a good heuristic for choosing an order should be effective. The heuristic that we have chosen is to order the array statement congruence classes by their size, largest to smallest for the given basic block, and to give the scalar and communication classes the lowest priority.

Given the chosen priority ordering, the context partitioning algorithm is incrementally optimal; *i.e.*, for each class c, given a partitioning of classes with higher priority, the partitioning of c results in a minimal number of partitions. In the absence of class conflicts, maximal fusion is obtained, sans fusion that leads to SPMD code generation problems. The algorithm makes two passes over the DDG for each class, and thus partitions the statements in $O((N+E)C)$ time, where N is the number of statements, E is the number of dependence edges, and C is the number of congruence classes.

During subgrid loop generation, all array statements in a partition are placed in the same subgrid loop. The number of subgrid loops which operate on congruent array statements is thus minimal, given the chosen priority ordering. In addition, the problem of over fusing loops has been completely avoided.

4.3 Context Partitioning Examples

Figure 4 displays how context partitioning would handle a block of eight statements. The statements are numbered to represent their textual order. The subscripts represent their congruence class. In this example, there are three congruence classes: class A with iteration space 1:256, class B with iteration space 1:100, and a separate class S for the scalar statement appearing in statement 6. In this example, all arrays have identical distributions. Figure 4(a) shows the original source code. Naive code generation would create six loops for these statements (only statements 4 and 5 would be fused into the same loop). Figure 4(b) shows the code after context partitioning. The modified code requires only

```
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 5: Problem 9 from the Purdue Set.

```
RIP  = CSHIFT(U,SHIFT=+1,DIM=1)
RIN  = CSHIFT(U,SHIFT=-1,DIM=1)
T    = U + RIP + RIN
TMP1 = CSHIFT(U,SHIFT=-1,DIM=2)
T    = T + TMP1
TMP2 = CSHIFT(U,SHIFT=+1,DIM=2)
T    = T + TMP2
TMP3 = CSHIFT(RIP,SHIFT=-1,DIM=2)
T    = T + TMP3
TMP4 = CSHIFT(RIP,SHIFT=+1,DIM=2)
T    = T + TMP4
TMP5 = CSHIFT(RIN,SHIFT=-1,DIM=2)
T    = T + TMP5
TMP6 = CSHIFT(RIN,SHIFT=+1,DIM=2)
T    = T + TMP6
```

Figure 6: Problem 9 after communication generation.

```
RIP  = CSHIFT(U,SHIFT=+1,DIM=1)
RIN  = CSHIFT(U,SHIFT=-1,DIM=1)
TMP1 = CSHIFT(U,SHIFT=-1,DIM=2)
TMP2 = CSHIFT(U,SHIFT=+1,DIM=2)
TMP3 = CSHIFT(RIP,SHIFT=-1,DIM=2)
TMP4 = CSHIFT(RIP,SHIFT=+1,DIM=2)
TMP5 = CSHIFT(RIN,SHIFT=-1,DIM=2)
TMP6 = CSHIFT(RIN,SHIFT=+1,DIM=2)
T    = U + RIP + RIN
T    = T + TMP1
T    = T + TMP2
T    = T + TMP3
T    = T + TMP4
T    = T + TMP5
T    = T + TMP6
```

Figure 7: Problem 9 after context partitioning.

three loops to be generated, and these loops exhibit significantly improved data locality and reuse compared to the loops generated for the original code.

As a second example, consider the code shown in Figure 5. This example, which computes a standard 9-point stencil, was taken from Problem 9 of the Purdue Set [22] as adapted for Fortran D benchmarking by Thomas Haupt of NPAC [21]. The arrays `T`, `U`, `RIP`, and `RIN` are all two-dimensional and have been distributed in a (BLOCK,BLOCK) fashion.

The compilation model described in Section 4.1 will hoist the explicit calls to the CSHIFT intrinsics out of the array statements and assign them to temporary arrays. This results in the code shown in Figure 6. If no effort is made by the compiler to rearrange these statements, the final code will contain seven loop nests each separated from the others by communication operations.

In this example there are only two congruence classes: the array statements, which are all congruent, and the communication statements. There are also only two types of dependences that exist in the code: true dependences from the CSHIFT operations to the expressions that use the `TMP` arrays, and the true and anti-dependences that exist between the multiple occurrences of the array `T`. Since all the

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 7. Since the array statements are now adjacent, scalarization will be able to fuse them into a single loop nest which exhibits a tremendous amount of data reuse. Similarly, the communication statements are now adjacent and can be easily optimized [14, 23].

5 Experimental Results

Context partitioning enhances the performance of HPF programs in two distinct ways. First, by blocking the fusion of noncongruent array statements, it prevents the performance degradation caused by over fusing loops. Second, by rearranging array statements into larger blocks of congruent statements, it increases the amount of useful fusion thus resulting in better data locality, reduced loop overhead, and improved performance. We have gathered experimental results to illustrate both of these aspects.

The second example in Section 3 presented code produced by Rice's dHPF compiler. When executed on an IBM SP-2, the performance of the code generated for the fused loops is 22% worse than for the code where the loops were not fused. This is due to the existence of the conditional and `IMOD` function within the second loop. This demonstrates that the performance degradation caused by over fusing loops is real and can be quite significant.

We have also included context partitioning as a component of our stencil compilation strategy [25]. In that work, context partitioning was responsible for a 31% reduction in execution time for the Problem 9 test case described in the example of the preceding section when executed on a four processor SP-2. Additionally, in early experiments on SIMD machines [16], context partitioning reduced the execution time of a section of the ARPS weather prediction code [9] by 35% when executed on a MasPar MP-2. In that section of code, it was able to reduce the number of loop nests from eight down to two. These demonstrate the utility of context partitioning for improving data locality and reducing loop overhead by enhancing the amount of fusion performed.

6 Related Work

Many people have recognized the need to fuse loops generated by the scalarization of Fortran90 array statements [4, 8, 28, 32]. However, in all of these cases, only adjacent scalarized loops are fused and no code motion is performed to increase the chances of fusion.

The ZPL compiler [18] is an exception in that it aggressively rearranges array statements to promote loop fusion for the purpose of array contraction. The compiler performs dependence analysis at the array statement level and includes an advanced scalarization algorithm, similar to the analysis and transformations we presented in an earlier paper [24]. They limit fusion to array statements that operate under the same ZPL *region*, which is similar to our congruence classes, and thus they avoid the problem of over fusing loops. Their algorithm, however, performs no fusion for statements not containing contractible arrays, thus losing out on the data locality and reuse benefits of fusion. Conversely, since our algorithm strives for maximum beneficial fusion, array contraction analysis could subsequently be performed on each context partition. Such a strategy would likely be just as successful in contracting arrays.

IBM's pHPF compiler [11] will avoid the problem of over fusing loops since it attempts loop fusion after SPMD loop generation. However, this requires that their compiler perform sophisticated symbolic analysis of the parameterized SPMD loop bounds to identify conformable loops. This is likely to limit their success of fusing loops to only the simplest cases.

Hwang *et al.* [13] propose optimizing array operations by synthesizing consecutive array operations or expressions into a single composite mathematical function. Such a synthesis is able to fuse loops and eliminate array temporaries, but is limited to arrays whose live range does not extend beyond the basic block.

Kennedy and McKinley [15], as well as Gao *et al.* [10], investigate rearranging Fortran77 loop nests to increase fusion so as to improve parallelism and data locality, or to promote array contraction. However, their work targets shared-memory parallel processors, and thus they do not have to consider the problem of over fusing loops as addressed in this paper.

While giving some optimization hints for the Slicewise CM Fortran compiler, Sabot describes the need for code motion to increase the size of elemental code blocks (blocks of Fortran90 array statements for which a single loop can be generated) [28]. He goes on to state that the compiler does not perform this code motion on user code, and thus it is up to the programmer to make the code blocks as large as possible. In a later paper describing the internals of the compiler, he describes how it attempts to perform limited code motion so that loops may become adjacent and thus fused [27]. However, the code motion performed is limited to only moving compiler-generated scalar code from between loops, not in moving the loops themselves.

7 Conclusion

In this paper, we introduced a problem with loop fusion which is unique to distributed-memory architectures. Loops become over fused if the arrays they operate on have different distributions or iteration spaces. The SPMD code generated for such loops often executes slower than the code generated for the separate loops. To address this problem, we presented our context partitioning algorithm, which not only prevents loops from being over fused, but also increases useful loop fusion. Our experimental results have shown that this optimization can have a significant impact on the performance of HPF programs.

References

[1] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1979.

[2] Vikram Adve and John Mellor-Crummey. Advanced code generation for High Performance Fortran. In *Languages, Compilation Techniques and Run Time Systems for Scalable Parallel Systems*, Lecture Notes in Computer Science Series. Springer-Verlag, 1997.

[3] Vikram Adve and John Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[4] J. R. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, October 1992.

[5] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[6] 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.

[7] D. Callahan. *A Global Approach to Detection of Parallelism*. PhD thesis, Dept. of Computer Science, Rice University, March 1987.

[8] M. Chen and J. Cowie. Prototyping Fortran-90 compilers for massively parallel machines. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.

[9] K. Droegemeier, M. Xue, P. Reid, J. Bradley, and R. Lindsay. Development of the CAPS advanced regional prediction system (ARPS): An adaptive, massively parallel, multi-scale prediction model. In *Proceedings of the 9th Conference on Numerical Weather Prediction*, American Meteorological Society, October 1991.

[10] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[11] 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.

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

[13] G.-H. Hwang, J. Lee, and D.-C. Ju. Array operation synthesis to optimize HPF programs. In *Proceedings of the 25th International Conference on Parallel Processing (ICPP'96)*, Bloomingdale, IL, August 1996.

[14] 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.

[15] 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. (also available as CRPC-TR94370).

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

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

[18] E Lewis, C. Lin, and L. Snyder. The implementation and evaluation of fusion and contraction in array languages. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[19] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[20] John D. McCalpin. A case study of some issues in the optimization of Fortran 90 array notation. *Scientific Programming*, 5:219–237, 1996.

[21] 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.

[22] 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.

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

[24] G. Roth and K. Kennedy. Dependence analysis of Fortran90 array syntax. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, Sunnyvale, CA, August 1996.

[25] G. Roth, J. Mellor-Crummey, K. Kennedy, and R. G. Brickner. Compiling stencils in High Performance Fortran. In *Proceedings of SC'97: High Performance Networking and Computing*, San Jose, CA, November 1997.

[26] 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.

[27] G. Sabot. Optimized CM Fortran compiler for the Connection Machine computer. In *Proceedings of the 25th Annual Hawaii International Conference on System Sciences*, Kauai, HI, January 1992.

[28] G. Sabot, (with D. Gingold, and J. Marantz). CM Fortran optimization notes: Slicewise model. Technical Report TMC-184, Thinking Machines Corporation, March 1991.

[29] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, January 1993.

[30] J. Warren. A hierachical basis for reordering transformations. In *Conference Record of the Eleventh Annual ACM Symposium on the Principles of Programming Languages*, Salt Lake City, UT, January 1984.

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

[32] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.