Context Optimization for SIMD Execution

Ken Kennedy
Gerald roth

CRPC-TR93306-S
April 1993

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houst on, TX 77251-1892

Revised October 1993, March 1994

Context Optimization for SIMD Execution

Ken Kennedy Gerald Roth [*]

Department of Computer Science
Rice University
Houston, Texas 77251

March 18, 1994

Abstract

SIMD architectures offer an alternative to MIMD architectures for obtaining high performance computation through parallelism. However, to obtain the best performance on such architectures, aggressive compilation techniques are required. One issue that SIMD compilers must address is generating code to change the machine context; *i.e.*, disabling processors not involved in the current computation. This paper addresses the cost of changing the machine context.

We present two compiler optimizations that reduce the cost of context changes. The first optimization, *context partitioning*, reorders the code so that as subgrid loops are generated, as many statements as possible that require the same context are placed in the same loop nest. The second optimization, *context splitting*, splits the iteration space of the subgrid loops into sets that have invariant contexts. This allows us to hoist the code that sets the machine context out of the subgrid loops.

1 Introduction

SIMD machines offer impressive cost/performance ratios, and they are very well suited for a large body of engineering and scientific applications. However, current compilers for SIMD machines do not come close enough to exploiting the full potential of these machines. Within the Fortran D project, we are developing a compiler to study advanced compilation techniques for SIMD machines. This paper describes one facet of the project.

With SIMD machines there is a need to explicitly turn processors on and off. This is due to the fact that there is only a single instruction stream and not all processors are to execute each instruction. Only processors containing data related to the current instruction should execute it. If a processor is not to execute a set of instructions, it must be explicitly "masked out". However, changing the machine state, or *context*, is an expensive operation. Setting the machine context is an overhead that one must pay to execute on a SIMD architecture. The work presented here addresses this overhead by reducing the number of times that the machine context must be set.

[*]This work supported in part by the IBM Corporation through the Graduate Resident Study Program.

The next section gives an overview of a SIMD architecture, the Fortran D language, and a general SIMD compilation framework. It introduces the concepts that motivate this work. Section 3 describes our strategy for reducing the cost of setting the machine context. We present some preliminary results in Section 4. Section 5 discusses related work by others. We conclude in Section 6 with a brief summary.

2 Machine, Language, and Compiler Overview

In this section we will give a brief overview of the target SIMD architecture, the Fortran D language, and our Fortran 90D SIMD compilation strategy. These descriptions will introduce the concepts necessary to understand how the machine works, why the compiler must generate certain code sequences, and why the optimizations described in later sections are beneficial.

2.1 A Distributed-Memory SIMD Architecture

SIMD is an acronym for *S*ingle *I*nstruction stream, *M*ultiple *D*ata streams. A SIMD computer contains many data processors operating synchronously, each executing the same instruction, using a common program counter. Each data processor is a fully functional ALU (Arithmetic Logical Unit). The SIMD architectures in which we are interested associate some local memory with each data processor. The data processor, along with its associated memory, is referred to as a *processing element* (PE). The collection of all PEs is called the *PE array.* The PE array may be treated as a linear array or as an array of higher dimensions; *e.g.*, a 16 PE array could be treated as a 16×1 grid, or an 8×2 grid, or a 4×4 grid, or even a $2 \times 2 \times 2 \times 2$ grid ($\equiv$ 4-d hypercube). For simplicity, we will limit our discussions to treating the PE array as a linear array or a two dimensional square grid.

Each PE has an execution flag which can be set on or off to indicate whether the PE should execute the current instruction. There are several reasons why a PE may be turned off during a computation. The two main reasons are that the current operation is being performed under a user specified mask (such as a Fortran 90 `WHERE` statement), or that the PE does not have any local data on which to operate. When taken as a whole, the execution flags of all the PEs are said to determine the *context* of the PE array. Some instructions are executed regardless of the execution mask, for example instructions that reset the execution flag.

There is also a serial *front end* processor or *control unit.* The front end (FE) processor has three responsibilities. The first is to drive the PE array by broadcasting instructions and related data to all PEs. The second is to perform all scalar computations and control flow operations. Third, the FE is the system interface to the external world.

Finally, the PEs are connected by an interprocessor communication network. This network allows PEs to access data that is stored within the memories of other PEs. The details of such a network, however, are not important to this work.

2.2 The Fortran D Language

Fortran D provides users with explicit control over data partitioning using data *alignment* and *distribution* specifications. The `DECOMPOSITION` statement specifies an abstract problem or index domain. The `ALIGN` statement specifies fine-grain parallelism, mapping each array element onto one or more elements of the decomposition. This provides the minimal requirement for reducing data movement for the program given an unlimited number of processors. The alignment of arrays to decompositions is determined by their subscript expressions in the statement; perfect alignment results if no subscripts are used.

The DISTRIBUTE statement specifies coarse-grain parallelism, grouping decomposition elements and mapping them and aligned array elements to the finite resources of the physical machine. Each dimension of the decomposition is distributed in a BLOCK, CYCLIC, or BLOCK_CYCLIC manner. Both irregular and dynamic data decomposition are supported. In this paper, however, we will only be concerned with regular data distributions. The complete language is described in detail elsewhere [12].

2.3 SIMD Compilation

This section describes our overall compilation strategy. It describes the steps necessary in translating a Fortran 90D program for execution on a SIMD architecture. The order that the steps are given reflects our compiler's structure, but this does not imply that all SIMD compilers are structured similarly. For a comparison, Albert *et al.* give an overview of compiling for the Connection Machine in Paris mode [2], and Sabot describes compiling for the Connection Machine in Slicewise mode [20].

2.3.1 Array Distribution

To exploit parallelism, the Fortran 90D SIMD compiler distributes the data arrays across the PE array so that each PE has some of the data to process. The manner in which arrays are distributed is very important for maximizing parallelism while minimizing expensive communication operations. When arrays are distributed across the PE array, each PE will locally allocate an equal-sized *subgrid* to hold its portion of the distributed array. The rank of the subgrid matches the rank of the distributed array. The extent of the i-th subgrid dimension is $Extent_i = \lceil N_i/P_i \rceil$, where N_i and P_i are the extents of the distributed array dimension and the PE array dimension, respectively. $P_i = 1$ for dimensions which are not distributed. The PE array itself will be considered has having a rank equal to the number of distributed dimensions of the distributed array. To simplify our discussion, we will limit the number of distributed dimensions to two.

The compiler uses a *distribution function* [14] to calculate the mapping of an array element to a subgrid location within a PE. Given an array A, the distribution function $\mu_A(\vec{i})$ maps an array index $\vec{i}$ into a pair consisting of a PE index $\vec{iproc}$ and a subgrid index $\vec{j}$. Inverse distribution functions, $\mu_A^{-1}(\vec{iproc}, \vec{j})$, give the reverse mapping. Figure 1 shows examples of distribution functions and their inverses for one-dimensional arrays with either BLOCK or CYCLIC distributions. In this paper all arrays, whether a user array or the PE array, use one-based indexing.

The Fortran D code in Figure 2 illustrates the concepts of data distribution. Given a SIMD machine with $P = 16$ PEs, the compiler would distribute array X as shown in Figure 3. Each PE would allocate a local subgrid $X'(16)$. The distribution function for X is:

$$\mu_X(i) \;=\; ((i-1) \bmod 16 + 1, \lceil i/16 \rceil)\,.$$

On the same machine, array Y would be distributed by the compiler as shown in Figure 4.

$$
\begin{aligned}
\mu_{block}(i) &= (\lceil i/Extent_1 \rceil, (i-1) \bmod Extent_1 + 1) \\
\mu_{block}^{-1}(iproc, j) &= (iproc - 1) * Extent_1 + j \\
\mu_{cyclic}(i) &= ((i-1) \bmod P_1 + 1, \lceil i/P_1 \rceil) \\
\mu_{cyclic}^{-1}(iproc, j) &= (j-1) * P_1 + iproc
\end{aligned}
$$

Figure 1 Distribution functions and their inverses.

```
REAL X(256), Y(20,20)
DECOMPOSITION A(256), B(20,20)
ALIGN X(I) WITH A(I)
ALIGN Y(I,J) WITH B(I,J)
DISTRIBUTE A(CYCLIC)
DISTRIBUTE B(BLOCK,BLOCK)
```

Figure 2 Fortran D code declaring two distributed arrays.

Notice how the PE array is now treated as a 4×4 matrix of PEs; *i.e.*, $P_1 = P_2 = 4$. Thus $Extent_1 = Extent_2 = 5$ and each PE would allocate $Y'(5,5)$ as the local subgrid. In this case the distribution function for Y is:

$$\mu_Y(i,j) = ((\lceil i/5 \rceil, \lceil j/5 \rceil), ((i-1) \bmod 5 + 1, (j-1) \bmod 5 + 1)).$$

2.3.2 Computation Partitioning

After the compiler maps distributed arrays onto the memory of the PE array, it must map parallel computations to the processors. Our philosophy is to use the "owner computes" rule, where every processor only performs computations that update data it owns [8, 26]. This rule could be replaced by a *data optimization* phase which may determine an alternate distribution for the computation and associated intermediate results in an attempt to reduce the amount of communication [9, 16].

2.3.3 Communication Generation

Once data and computation distributions are finalized, the compiler must insert any necessary communication operations to move data so that all operands of an expression reside on the PE which will perform the computation. These communication operations can often be optimized by exploiting efficient collective communication routines; *e.g.*, `EOSHIFT` or `TRANSPOSE`. The exact details of how the required communication operations are determined and generated are beyond the scope of this paper and are not necessary for understanding the optimizations discussed in later sections. Interested readers are referred to the work by Li and Chen [18].

After the communication operations have been inserted, all computational expressions reference data that are strictly local to the associated PEs. For example, the statement:

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

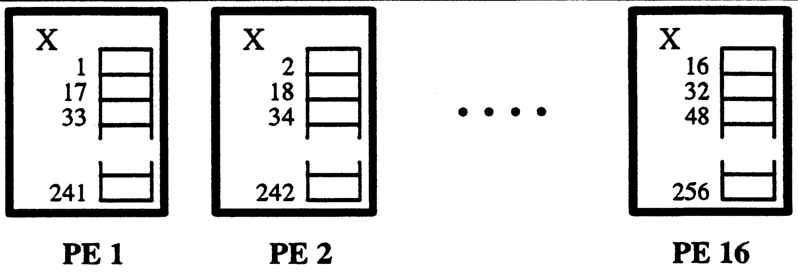


Figure 3 A one-dimensional array with 256 elements mapped in a CYCLIC manner onto a 16 PE machine.

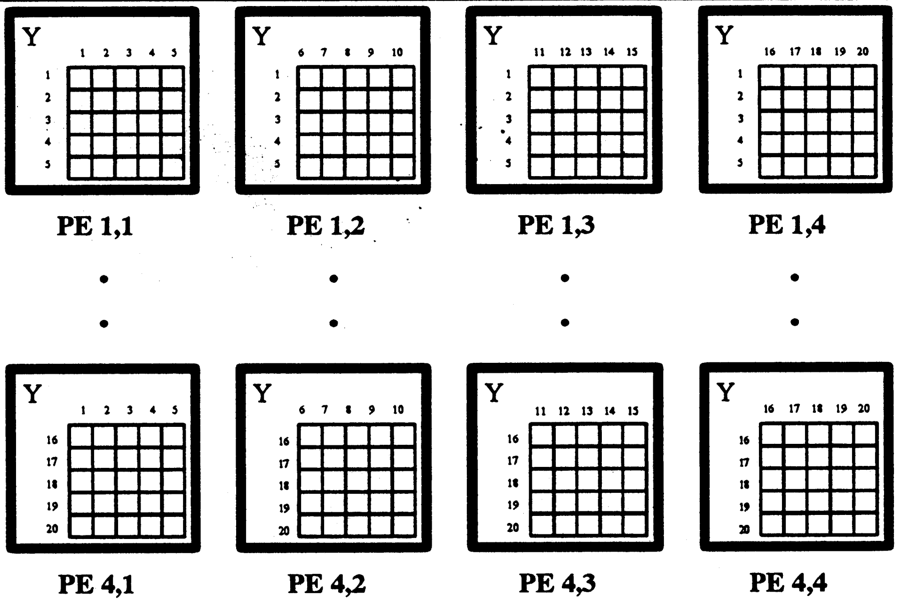


Figure 4 A 20 × 20 two-dimensional array mapped in a BLOCK fashion onto a 16 PE machine configured as a 4 × 4 matrix.

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

```
TMP1 = EOSHIFT(X,-1)
TMP2 = EOSHIFT(X,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 one another and that there is no further communication required to compute the expression or store the result.

2.3.4 Subgrid Looping

Finally, the compiler translates the parallelism that is explicit in the Fortran 90 array syntax into code that manipulates the arrays that have been distributed across the PEs. Since each PE is in fact a serial processor, the array expressions must be *scalarized*, *i.e.*, translated into serial code [4, 25]. The serial code operates on the data local to a PE. If an array is distributed such that several elements are allocated per PE, then the serial code is placed in a loop (or loop nest as required) that iterates over the subgrid. This is known as the *subgrid loop*. For a detailed description of the issues involved in generating correct subgrid loops for SIMD architectures, see the paper by Weiss [24].

As an example, if the array assignment statement:

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

is performed on array X in Figure 3, the following subgrid loop will be generated:

```
DO I = 1, Extent1          !  Extent1 = 16
   X'(I) = X'(I) + 1.0
ENDDO
```

Recall that X′ is the local subgrid instantiation of the distributed array X. $\texttt{Extent}_1$ is the size of the subgrid and is calculated as described in Section 2.3.1.

The execution of the subgrid loop is a cooperative effort between the FE and the PE array. The FE handles the control flow by executing the looping construct. For each iteration of the loop, the FE then broadcasts instructions to the PE array. The broadcast instructions will result in each PE adding 1.0 to X′(I), where the value of I is also broadcast from the FE.

Subgrid looping is very closely related to *sectioning* used for allocating vector registers [6]. In this case, the PE array can be thought of as a multidimensional vector register. Just as in vector register allocation, *loop fusion* [3] can be a powerful optimization. Loop fusion merges multiple loops covering the same iteration space into a single loop. Loop fusion not only decreases the total amount of loop overhead, but more importantly it creates opportunities for data to be kept in registers and reused, thus avoiding expensive loads from memory.

Unfortunately, loop fusion is not always safe. A data dependence between two adjacent loops is called *fusion-preventing* if after fusion the direction of the dependence is reversed [1, 23]. The existence of such a dependence means that fusion is not safe. In our case however, no such fusion-preventing dependences can exist between adjacent subgrid loops. This is due to the fact that the generation of communication, as described in the preceding section, causes all subgrid loops to operate on "perfectly aligned" data. For example, 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 Fortran 90 array expressions (and their corresponding subgrid loops) from becoming adjacent.

Due to this perfect alignment of data within array operations, our SIMD compiler can directly generate a single subgrid loop nest for adjacent Fortran 90 array statements if they have the same distribution and cover the same iteration space. We call such array statements *congruent*. Such subgrid loop generation precludes the need for loop fusion.

2.3.5 Context Switching

Up to this point all our examples used arrays that, when distributed, gave an equal number of elements per PE, and all expressions involving the arrays accessed the entire array. When either of these conditions is not met, the compiler may have to insert code into the subgrid loop to change the context of the PE array.

Given the Fortran D declarations in Figure 2, assume we now encounter the array assignment statement `X(2:242) = X(2:242) + 1.0`, which increments 241 elements of `X` starting with the second element. As illustrated in Figure 3, PE 1 holds 15 of these elements in `X'(2:16)`, PE 2's full subgrid is involved, and PEs 3 through 16 each have affected elements in `X'(1:15)`. The subgrid loop for this statement must enable and disable different sets of PEs depending upon which subgrid element is being processed. The inverse distribution functions described in Section 2.3.1 determine which sets of PEs to enable. The subgrid loop generated for this statement is:

```
DO I = 1, Extent1          !  Extent1 = 16
   Set_Context(((I-1)*P + iproc ≥ 2) .AND. ((I-1)*P + iproc ≤ 242))
   X'(I) = X'(I) + 1.0
ENDDO
```

`P` is the number of processors, while `iproc` is a unique number assigned to each PE and corresponds to its position in the PE array. The function `Set_Context` will cause each PE to evaluate the logical expression and enable its execution flag if the result is true, otherwise the execution flag is disabled.

In a similar manner, operations on arrays which do not "evenly" fill the machine require context switching code to be inserted into the subgrid loop. Compare the Fortran D declarations in Figure 5 to those in Figure 2. On the same 16 processor machine, array `Y2` would be distributed as seen in Figure 6. Looking at Figure 6, it can be seen that array `Y2` is distributed such that PEs (1:3,1:3) each contain a full subgrid of data, PEs (1:3,4) contain data only in the left portion of their subgrids, PEs (4,1:3) contain data only in the upper portion of their subgrids, PE (4,4) contains data only in the upper-left corner of its subgrid.

Since different PEs contain data in different subgrid locations, the subgrid loop must contain code to change the context depending upon which subgrid element is being processed. Again, the inverse distribution functions determine the active set of PEs. For example, the Fortran 90 statement `Y2 = ABS( Y2 )` would generate the following subgrid loop nest:

```
DO J = 1, Extent2                !  Extent2 = 5
   DO I = 1, Extent1             !  Extent1 = 5
     Set_Context (((iproc1-1) * Extent1 + I ≤ 17) .AND.
                  ((iproc2-1) * Extent2 + J ≤ 19))
     Y2'(I,J) = ABS ( Y2'(I,J) )
   ENDDO
ENDDO
```

`Extent`$_i$ and `iproc`$_i$ are the subgrid extent and PE index, respectively, along the i-th dimension.

It should be obvious by now that the code required for changing the context can be quite complex. If the previous example had only operated on a subrange of the array `Y2`, for example, then the call to `Set_Context` would also have required a lower bound check for each dimension. This would double the number of logical operations.

We do have one mitigating factor. When a single subgrid loop nest is generated for a set of adjacent congruent array statements, we do not need to change the context for each individual statement in the subgrid loop; *i.e.*, they all operate under the same context. This is true by our definition of congruent array statements: they have the same iteration space and operate on arrays that have identical distributions.

```
REAL Y2(17,19)
DECOMPOSITION B2(17,19)
ALIGN Y2(I,J) WITH B2(I,J)
DISTRIBUTE B2(BLOCK,BLOCK)
```

Figure 5 Fortran D code declaring an odd-shaped array.

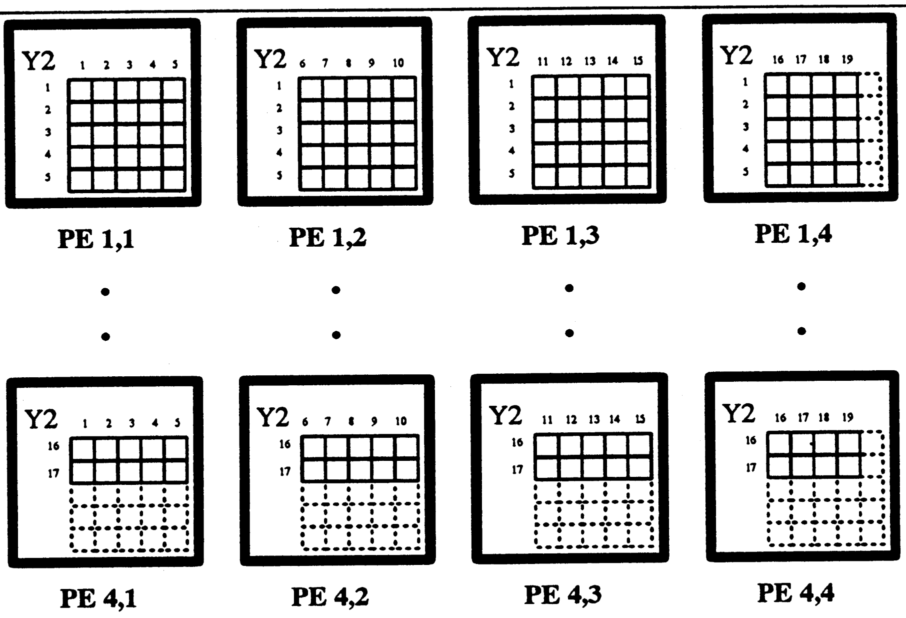


Figure 6 A 17 × 19 two-dimensional array mapped in a BLOCK fashion onto a 16 PE machine configured as a 4 × 4 matrix.

3 Context Optimization

As can be seen in the two simple examples given in Section 2.3.5, the code required to set the context of the PE array can include multiple logical and arithmetic operations. This code can be a significant portion of the work performed within a subgrid loop. It is our goal to reduce the impact of this overhead for programs compiled for SIMD machines.

Our Fortran 90D SIMD compiler strategy has a two-pronged approach to minimize the expense of context switching. First, we rearrange the program statements so that as subgrid loops are generated, as many statements as possible that execute under the same context are placed within the same subgrid loop. Second, we alter the order in which subgrid elements are processed by performing loop transformations on the subgrid loops. These loop transformations will allow us to hoist the calls to `Set_Context` out of the loops and thus reduce the number of context changes. These optimizations are described in detail in the following subsections.

3.1 Context Partitioning

As explained in Sections 2.3.4 and 2.3.5, a single subgrid loop nest is generated for adjacent Fortran 90 array statements that are congruent and these statements all execute within the same context. However, unless an effort is made to make congruent array statements adjacent, many small subgrid loops may still be generated. Sabot has recognized this problem, and recommends that users of the CM Fortran compiler rearrange program statements, when possible, to avoid the inefficiencies of such subgrid loops [21]. In order to alleviate this problem automatically, our compiler has an optimization phase that reorders the statements within a basic block. The reordering attempts to create separate partitions of scalar statements, communication statements, and congruent array statements. We call this optimization *context partitioning.* The partitioning could group scalar statements and communication statements together; however, we prefer to separate them so that the communication operations can be further optimized by subsequent phases.

To accomplish context partitioning, we use an algorithm proposed by Kennedy and McKinley [15]. Whereas they were concerned with partitioning parallel and serial loops, we are partitioning Fortran 90 statements. The algorithm works on the *data dependence graph* (DDG) [17] which must be acyclic. Since we are working with a basic block of statements, our dependence graph will contain only *loop-independent* dependences [5] and thus meets that criteria. 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 congruence classes for scalar statements, communication statements and each set of congruent array statements.

The priority ordering 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 from another class, but not both since that would introduce a cycle in the DDG and thus make it unschedulable. The following contrived code segment, whose DDG is shown in Figure 7, gives an example of a class conflict:

```
s1:  A(1:100) = A(1:100) + 1.0
s2:  B(2:99) = B(2:99) * C(2:99)
s3:  C(1:100) = B(1:100)
s4:  D(2:99) = A(2:99)
```

It is possible to merge nodes *s1* and *s3* or nodes *s2* and *s4*, but we cannot merge both pairs. The priority ordering is used to determine which pair should be merged. The algorithm will merge pairs with a higher priority before those with a lower priority. Kennedy and McKinley have shown that choosing an optimal ordering of classes is NP-hard in the number

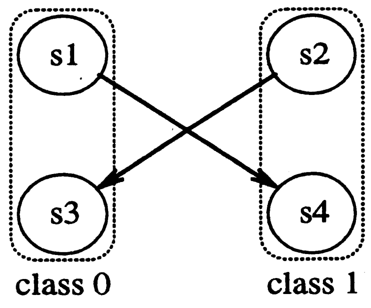


Figure 7 A context partitioning class conflict.

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.

We will now give a brief overview of the algorithm; we refer interested readers to their paper for the details. For each congruence class in priority order, the algorithm makes a single pass over the DDG. During the pass it greedily merges nodes for the given class. Two nodes from the same class may be merged if there does not exist a path between them that contains a node from another class. When two nodes are merged, the DDG is updated to reflect it.

Figure 8 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 two congruence classes: A and B. Statement 6 is a scalar statement. Figure 8(a) shows the original source code, and Figure 8(b) shows the data dependence graph. Naïve code generation would create six subgrid loops for these statements (only statements 4 and 5 would occur in the same subgrid loop). Figure 8(c) shows the data dependence graph after partitioning congruence class B, the larger of the two classes. Figure 8(d) displays the final result after partitioning class A. The final code is displayed in Figure 8(e). The modified code requires only three subgrid loops to be generated. (Note: Figures 8(b-d) show only true (flow) dependences; anti- and output-dependences have not been shown since they clutter the diagrams and add little benefit.)

Given the chosen priority ordering, the 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. The algorithm will partition the DDG 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.

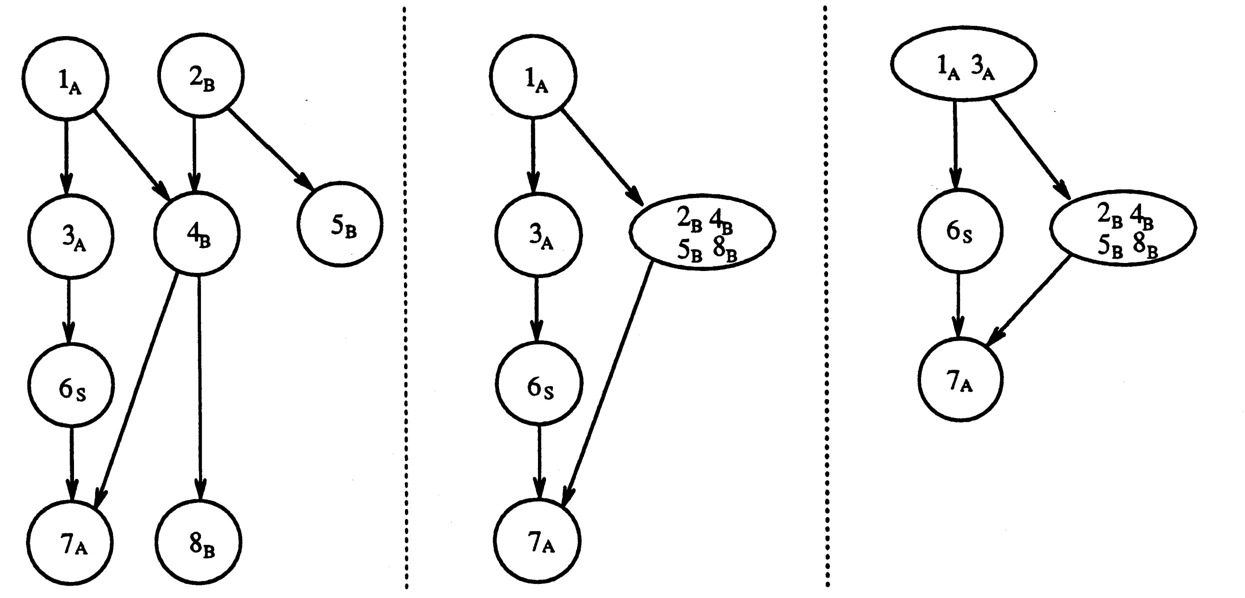
During subgrid loop generation, all statements in a partition will be placed in the same subgrid loop. The number of subgrid loops which operate over statements with the same context is thus minimal, given the chosen priority ordering.

It should be noted that context partitioning is not a SIMD-only optimization. It is useful for Fortran 90 compilers that target MIMD and scalar architectures as well. Even though such architectures do not require setting a machine context, context partitioning can reduce the overhead of loops generated during scalarization and can increase the possibilities for data reuse.

3.2 Context Splitting

In the first example in Section 2.3.5, which incremented `X(2:242)`, the PE array has the same context for iterations 2 through 15 of the subgrid loop; during these iterations all the PEs are active. To take advantage of this invariance, we will modify the subgrid loop by performing *loop splitting*, also called *index set splitting* [7, 25]. By splitting the iteration space into disjoint sets, each requiring a single context, we can safely hoist the context setting code out of the resulting loops. We call this optimization *context splitting.*

For the reference `X(2:242)`, the iteration space of the subgrid loop is split into 3 sets: {1}, {2:15}, and {16}. Applying context splitting to the subgrid loop produces the following code:



(b) dependence graph

(c) dependence graph after merging class B

(d) dependence graph after merging class A

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

(e) modified code

Figure 8 Partitioning congruent statements.

```
Set_Context(iproc ≥ 2)
X'(1) = X'(1) + 1.0
Set_Context(.TRUE.)
DO I = 2, 15
   X'(I) = X'(I) + 1.0
ENDDO
Set_Context(iproc ≤ 2)
X'(16) = X'(16) + 1.0
```

Compared to the original subgrid loop, this code has hoisted the call to **Set_Context** out of the loop, eliminated several arithmetic and logical operations, and the PE array context is now set only three times compared to the original 16 times. As we can see from this simple example, context splitting significantly reduces the number of times that the PE array context must be set for a subgrid loop. Whereas the original subgrid loop set the context once per iteration, it is now set only three times, no matter how large the subgrid is.

Unlike context partitioning, context splitting is a SIMD-only optimization. Compilers for MIMD machines can often side-step the issue that is addressed by context splitting. Since each PE in a MIMD machine also has control logic, a compiler can generate a program such that each PE determines the loop bounds for its own subgrid loop. By reducing the loop bounds, the compiler can often avoid iterations for which the PE has no work and thus does not need to introduce any guard statements into the subgrid loop body in those cases [13, 22].

We will now discuss the details of context splitting. To simplify the discussion, we will first discuss one-dimensional `CYCLIC`, `BLOCK`, and `BLOCK_CYCLIC` distributions, and then show how to combine one-dimensional splitting to handle multidimensional cases.

Our canonical example in the following presentation will be the statement `X(N:M) = X(N:M) + 1.0`. Context splitting of subgrid loops for full arrays that do not evenly fill the machine is treated simply as a special case, where `N = 1`.

3.2.1 Context Splitting a CYCLIC Distribution

With a standard `CYCLIC` distribution, element `X(N)` may reside on any processor relative to the processor holding element `X(M)`. However, the offset of `X(N)` within the subgrid `X'` must be less than or equal to the offset of `X(M)`. *I.e.*, given $\mu_X(N) = (iproc_N, j_N)$, $\mu_X(M) = (iproc_M, j_M)$, and $N \leq M$, then $j_N \leq j_M$ must hold. We cannot make any statement regarding the relationship of $iproc_N$ and $iproc_M$, except that $j_N = j_M$ implies $iproc_N \leq iproc_M$.

Using this information and our knowledge of `CYCLIC` distributions, we know that all PEs should be enabled for the subgrid iterations $j_N + 1$ to $j_M - 1$. This naturally divides the iteration space into three sets: $\{j_N\}$, $\{j_N + 1{:}j_M - 1\}$, and $\{j_M\}$. Figure 9 depicts this situation. When $j_N = j_M$, the second iteration set is empty and the first and third set are merged into a single set. The subgrid loop after context splitting now looks like this:

```
iproc_N = (N-1) mod P + 1
iproc_M = (M-1) mod P + 1
j_N = ⌈N/P⌉
j_M = ⌈M/P⌉
IF (j_N = j_M) THEN
   Set_Context(iproc ≥ iproc_N .AND. iproc ≤ iproc_M)
   X'(j_N) = X'(j_N) + 1.0
ELSE
   Set_Context(iproc ≥ iproc_N)
   X'(j_N) = X'(j_N) + 1.0
   Set_Context(.TRUE.)
   DO I = j_N+1, j_M-1
     X'(I) = X'(I) + 1.0
   ENDDO
   Set_Context(iproc ≤ iproc_M)
   X'(j_M) = X'(j_M) + 1.0
ENDIF
```

The code can be simplified if N and/or M are known constants. When N and M are both constants, the IF-test can be evaluated at compile-time and only the code for the appropriate branch needs to be generated. In the case where the operation is over the full array but the array does not evenly fill the machine, we know that N = 1. In this situation the IF-test is unnecessary and the pre-loop statements can be merged into the DO-loop. This is equivalent to peeling off the last iteration of the subgrid loop and hoisting the context setting code accordingly. The result is:

```
iproc_M = (M-1) mod P + 1
j_M = ⌈M/P⌉
Set_Context(.TRUE.)
DO I = 1, j_M-1
   X'(I) = X'(I) + 1.0
ENDDO
Set_Context(iproc ≤ iproc_M)
X'(j_M) = X'(j_M) + 1.0
```

3.2.2 Context Splitting a BLOCK Distribution

Given a BLOCK distribution, element X(N) will always reside on a processor that has a number less than or equal to the processor holding element X(M); *i.e.*, $iproc_N \leq iproc_M$. Figure 10 shows the affected elements of array X for the assignment X(N:M) = X(N:M) + 1.0 when X has a BLOCK distribution. As can be seen, all processors between $iproc_N$ and $iproc_M$ are enabled for all subgrid elements, whereas all processors outside that range are disabled for all subgrid elements. Processor $iproc_N$ is enabled at iteration j_N and subsequent iterations. Processor $iproc_M$ is enabled only for iterations up to and including j_M.

The difficulty in context splitting for a BLOCK distribution comes from distinguishing the case where $j_N \leq j_M$ from the case where $j_N > j_M$. If we let LO = $\min(j_N, j_M + 1)$ and

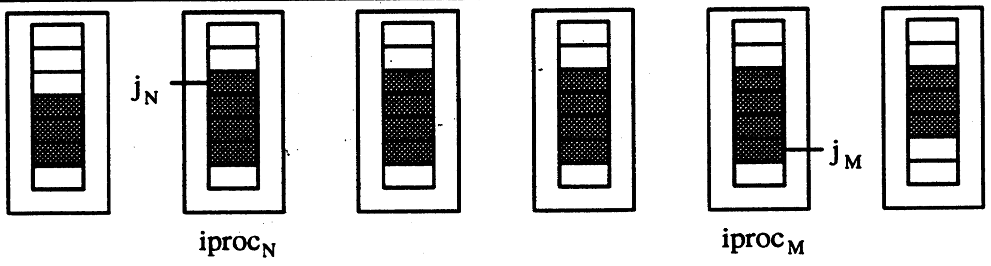


Figure 9 X(N:M) when X has a CYCLIC distribution.

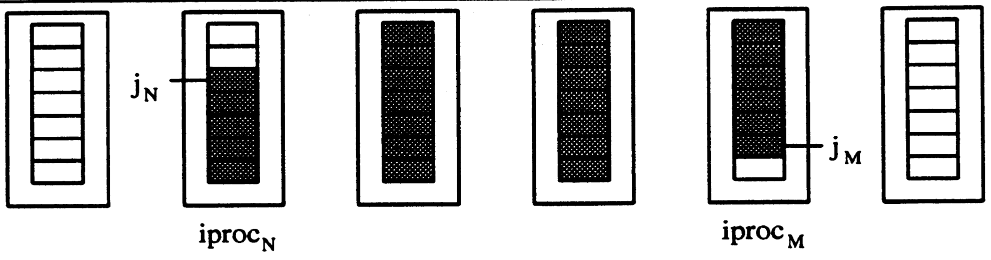


Figure 10 X(N:M) when X has a BLOCK distribution.

HI = max($j_N - 1, j_M$), then the iteration space is naturally split into the following three sets: {1:LO−1}, {LO:HI}, and {HI+1:Extent}. The context for the first iteration set will be the processor set $\{iproc_N + 1 : iproc_M\}$. The processor set for the second iteration set will include both $iproc_N$ and $iproc_M$ if $j_N \leq j_M$ holds, otherwise it will exclude both. The context for the third iteration set will be processors $\{iproc_N : iproc_M - 1\}$. The subgrid loop after context splitting is now:

```
iproc$_N$ = ⌈N/Extent⌉
iproc$_M$ = ⌈M/Extent⌉
j$_N$ = (N-1) mod Extent + 1
j$_M$ = (M-1) mod Extent + 1
IF (j$_N$ $\leq$ j$_M$) THEN
   LO = j$_N$
   HI = j$_M$
ELSE
   LO = j$_M$ + 1
   HI = j$_N$ - 1
ENDIF
Set_Context(iproc > iproc$_N$ .AND. iproc $\leq$ iproc$_M$)
DO I = 1, LO-1
   X'(I) = X'(I) + 1.0
ENDDO
IF (j$_N$ $\leq$ j$_M$) THEN
   Set_Context(iproc $\geq$ iproc$_N$ .AND. iproc $\leq$ iproc$_M$)
ELSE
   Set_Context(iproc > iproc$_N$ .AND. iproc < iproc$_M$)
ENDIF
DO I = LO, HI
   X'(I) = X'(I) + 1.0
ENDDO
Set_Context(iproc $\geq$ iproc$_N$ .AND. iproc < iproc$_M$)
DO I = HI+1, Extent
   X'(I) = X'(I) + 1.0
ENDDO
```

This code can be greatly simplified if both N and M are compile-time constants, in which case both IF expressions can be eliminated. In addition, if LO = 1 or HI = Extent then the body of the first or last DO-loop, respectively, will not be executed. That DO-loop and its associated call to Set_Context can then be safely eliminated.

In the case where the operation is over the entire array but the array does not evenly fill the machine, the above code can be simplified to:

```
iproc$_M$ = ⌈M/Extent⌉
j$_M$ = (M-1) mod Extent + 1
Set_Context(iproc $\leq$ iproc$_M$)
DO I = 1, j$_M$
   X'(I) = X'(I) + 1.0
ENDDO
Set_Context(iproc < iproc$_M$)
DO I = j$_M$+1, Extent
   X'(I) = X'(I) + 1.0
ENDDO
```

3.2.3 Context Splitting a `BLOCK_CYCLIC` Distribution

Up to this point we have not discussed using a `BLOCK_CYCLIC` distribution. This is mostly due to its complexity. We will introduce it now and so how context splitting can be used to optimize a subgrid loop operating over such a distribution.

`BLOCK_CYCLIC` is similar to `CYCLIC` but it takes a parameter b. It first divides the dimension into contiguous chunks of size b, then distributes these chunks in the same manner as `CYCLIC`. Figure 11 shows the distribution function used to map an array index to the PE index/subgrid index pair for a `BLOCK_CYCLIC` distribution. Also shown is the inverse function. It is interesting to compare these to the functions for the standard `BLOCK` and `CYCLIC` distributions, given in Figure 1. Recall that it is the inverse function that is used in the calls to `Set_Context` that occur in the naïve subgrid loops. In this case, the inverse function is so complex that it adds a significant amount to the overhead of the subgrid loop.

Let's consider once again our example `X(N:M) = X(N:M) + 1.0`. Figure 12 shows the affected elements given a block size $b = 3$ (dark lines indicate logical block boundaries). It can be easily seen that at any given subgrid index different sets of PEs contain elements from the range $N : M$. On closer inspection, it can be found that the sets of PEs that must be enabled form a logical progression. It is this progression that establishes the iteration sets for context splitting.

To ease the specification of the iteration sets we will not only need to know the values of j_N and j_M, the subgrid indices of $X(N)$ and $X(M)$ respectively, we will need to know the logical blocks in which they occur. We will label these blocks B_N and B_M. We use one-based indexing for the blocks just as we do for indexing subgrids and PEs.They can be calculated using the following formula:

$$B_k \;=\; \left\lfloor \frac{k-1}{b*P_1} \right\rfloor + 1 \;=\; \lceil j_k/b \rceil\,, \quad for\;\; k = N, M$$

Once we have determined the values of j_N, j_M, $iproc_N$, $iproc_M$, B_N, and B_M, we split the iteration space into the following five sections and determine the set of active PEs as shown:

$\{(B_N - 1) * b + 1 \;:\; j_N - 1\}$	Turn on PEs where $iproc > iproc_N$.
$\{j_N \;:\; B_N * b\}$	Turn on PEs where $iproc \geq iproc_N$.
$\{B_N * b + 1 \;:\; (B_M - 1) * b\}$	Turn on all PEs.
$\{(B_M - 1) * b + 1 \;:\; j_M\}$	Turn on PEs where $iproc \leq iproc_M$.
$\{j_M + 1 \;:\; B_M * b\}$	Turn on PEs where $iproc < iproc_M$.

Depending upon the values of N and M, one or more of the five iteration sets may be empty. In particular, when $N = 1$ the first set will be empty and the second and third

$$\mu_{block\_cyclic(b)}(i) \;=\; \left(\left\lceil \frac{(i-1) \bmod (b*P_1) + 1}{b} \right\rceil, \left\lfloor \frac{i-1}{b*P_1} \right\rfloor b + i \bmod b + 1 \right)$$

$$\mu^{-1}_{block\_cyclic(b)}(iproc, j) \;=\; \left\lfloor \frac{j-1}{b} \right\rfloor (P_1 * b) + (iproc - 1) * b + (j-1) \bmod b + 1$$

Figure 11 `BLOCK_CYCLIC` distribution function and its inverse.

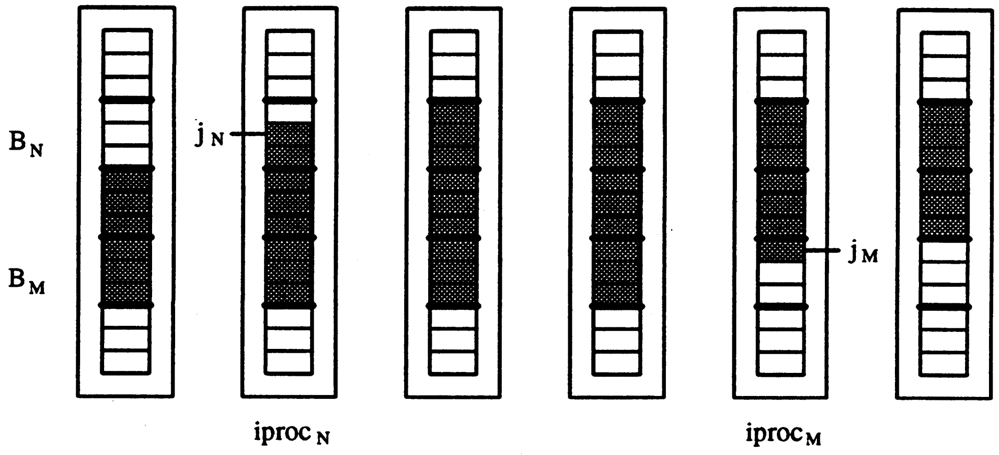


Figure 12 X(N:M) when X has a BLOCK_CYCLIC distribution.

set enable the same PEs and can thus be merged together. It is important to notice how context splitting has reduced the computations required for the calls to Set_Context to simple logical comparisons; the expensive operations of the inverse mapping function are no longer required.

With a BLOCK_CYCLIC distribution, there is one special case which we must handle when performing context splitting. It occurs when $B_N = B_M$. In this situation, all the affected elements of $X(N : M)$ are at the same logical block level. Furthermore, the elements are distributed across this block level in a normal BLOCK distribution. To properly handle this case we could apply the techniques presented in Section 3.2.2, with the modification that the subgrid loop ranges from $(B_N - 1) * b + 1$ to $B_N * b$ rather than from 1 to *Extent*. Alternatively, if the block size b is small, we could simply generate the naïve subgrid loop, since we know that the loop will perform at most b iterations.

The subgrid loop for the statement `X(N:M) = X(N:M) + 1.0` is shown below. For simplicity, we have used the naïve subgrid loop for the case where $B_N = B_M$. The variable GlobalIndex holds the value of the inverse distribution function, μ^{-1}, computed on each PE for each subgrid location.

```
iproc_N = ⌈((N-1) mod (b*P) + 1)/b⌉
iproc_M = ⌈((M-1) mod (b*P) + 1)/b⌉
j_N = ⌊(N-1)/(b*P)⌋ b + i mod b + 1
j_M = ⌊(M-1)/(b*P)⌋ b + i mod b + 1
B_N = ⌈j_N/b⌉
B_M = ⌈j_M/b⌉
IF (B_N = B_M) THEN    !! perform naïve subgrid looping
   DO i = (B_N - 1) * b + 1, B_N * b
     GlobalIndex = ⌊(i-1)/b⌋*(P*b) + (iproc-1)* b + (i-1) mod b + 1
     Set_Context(GlobalIndex ≥ N .AND. GlobalIndex ≤ M)
     X'(i) = X'(i) + 1.0
   ENDDO
ELSE
   Set_Context(iproc > iproc_N)
   DO I = (B_N-1)*b+1, j_N-1
```

```
      X'(I) = X'(I) + 1.0
    ENDDO
    Set_Context(iproc ≥ iproc_N)
    DO I = j_N, B_N*b
      X'(I) = X'(I) + 1.0
    ENDDO
    Set_Context(.TRUE.)
    DO I = B_N*b+1, (B_M-1)*b
      X'(I) = X'(I) + 1.0
    ENDDO
    Set_Context(iproc ≤ iproc_M)
    DO I = (B_M-1)*b+1, j_M
      X'(I) = X'(I) + 1.0
    ENDDO
    Set_Context(iproc < iproc_M)
    DO I = j_M+1, B_M*b
      X'(I) = X'(I) + 1.0
    ENDDO
ENDIF
```

3.2.4 Context Splitting a Multidimensional Distribution

To perform context splitting on a multidimensional distribution, we use loop splitting on each dimension separately. This produces a set of imperfectly nested DO-loops. We then use *loop distribution* [17, 19] to produce a set of perfectly nested DO-loops, each of which operates under a single context. The context for each loop nest is the intersection of the contexts produced for each dimension.

Let's consider again the array Y2 as declared and distributed in Figures 5 and 6. Performing context splitting on the statement Y2 = ABS(Y2), we first perform loop splitting on each dimension. For the first dimension, the iteration space is divided into the sets {1:2} and {3:5}, while the second dimension produces the sets {1:4} and {5}. After splitting the loops we use loop distribution, which produces these sets of two-dimensional iteration spaces: {1:2,1:4}, {3:5,1:4}, {1:2,5}, and {3:5,5}. The result is the following code, which sets the context four times compared to the 25 times of the naïve subgrid loop presented in Section 2.3.5:

```
Set_Context (.TRUE.)
DO J = 1, 4
   DO I = 1, 2
     Y2'(I,J) = ABS ( Y2'(I,J) )
   ENDDO
ENDDO
Set_Context (iproc_1 < 4)
DO J = 1, 4
   DO I = 3, 5
     Y2'(I,J) = ABS ( Y2'(I,J) )
   ENDDO
ENDDO
Set_Context (iproc_2 < 4)
DO I = 1, 2
   Y2'(I,5) = ABS ( Y2'(I,5) )
ENDDO
Set_Context (iproc_1 < 4 .AND. iproc_2 < 4)
DO I = 3, 5
   Y2'(I,5) = ABS ( Y2'(I,5) )
```

```
    ENDDO
```

3.2.5 Discussion

A close evaluation of the context splitting optimization reveals two possible concerns: loop overhead and code growth. Both of these occur because context splitting takes a single subgrid loop nest and generates multiple loop nests (with reduced loop bounds) each with a copy of the loop body (minus context setting code). We will address each of these concerns separately.

The additional loop overhead generated by context splitting is really not a concern at all. Recall that all the control flow operations related to the looping constructs are executed on the FE processor. Since the FE processor is usually much faster than the PE processors, and is executing asynchronously from them, it is able handle the extra loop overhead while still keeping the PE array busy. In essence, we have increased the workload executed on the FE, but this has allowed us to decrease the workload sent to the PE array.

Since context splitting produces several copies of the loop body for each loop level which is split, code growth is exponential in the number of nested subgrid loops which are split. If this growth is a concern, we have two alternatives that can be used to address it. First, the loop body could be encapsulated as an internal subroutine, which is branched to and returned from. Since the subroutine is internal, the interface simply requires that the return address be saved. Alternatively, by limiting context splitting to only the innermost one or two subgrid loop levels, one can keep code growth bounded by a linear amount. Our experiments have shown that this small limitation still retains most of the performance gains achieve when splitting all subgrid loop levels.

Throughout this presentation we have ignored the context setting required for masked assignment instructions, such as that generated by the `WHERE` statement. The semantics of masked instructions imply that each element has its own execution mask. This means that the iteration space cannot be divided into convenient sets each with a constant execution mask. For this reason context splitting cannot be used to hoist the calls to `Set_Context` out of such subgrid loops. Context partitioning is still applicable and can be used to generate a single subgrid loop for groups of congruent statements.

We have also not addressed strided references, such as `X(N:M:S)`. For strides known at compile-time we may be able to determine sets of PEs that have common strides through their subgrids. We would then be able to isolate these iterations into separate subgrid loops, allowing us to again hoist the context setting code from the loop. For example, for `S = 2` it may turn out that we can set the context to the even numbered PEs and process the even (odd) subgrid elements then switch to the odd numbered PEs and process the odd (even) elements. Additional splitting could be performed as required at the boundary iterations j_N and j_M. This is an item of current research.

4 Results

To verify the effectiveness of these optimizations, we performed them by hand on a section of code taken from a Fortran 90 version of the ARPS weather prediction code [11]. The section of code initializes 16 two-dimensional arrays. We chose this section of code since context partitioning would not benefit additionally from data reuse nor would it be penalized for generating excessive register pressure. Thus all performance improvements are directly attributable to the elimination of redundant context changes and the reduction of loop overhead.

We generated five versions of this code segment and timed each on a dedicated DECmpp 12000. The first version was simply the Fortran 90 segment as taken from the ARPS program. This was compiled with MP Fortran Version 2.1. The second version was a translation of the Fortran code into MPL. We optimized this version by performing the following optimizations by hand: common subexpression elimination, strength reduction, and loop-invariant code motion. We also used **register** declarations on array pointers. We then took this MPL version and generated three new versions by applying our context optimizations; one version for each of the optimizations, and one version which combined both optimizations. All the MPL routines were compiled by Version 3.0 of the MPL compiler. All five versions used a (CYCLIC,CYCLIC) distribution, the standard distribution of the MP Fortran compiler (the current version of the MP Fortran compiler does not support either BLOCK or BLOCK_CYCLIC distributions).

The version combining context partitioning and splitting reduced the execution time by 45% when compared to the original MPL code (which itself reduced the execution time by approximately 10% compared to the Fortran code). See Figure 13 for a comparison of execution time versus subgrid size for these three versions of the code. Individually, context partitioning and context splitting reduced the execution time by 35% and 45%, respectively. The reason that the combination of the two optimizations did not out-perform context splitting is that, once splitting eliminated the costly context setting code from the subgrid loops, the loops became memory bound. For subgrid loops that are more computationally intensive, we expect these two optimizations to have an additive effect, although the total effect may be less than the improvement experienced with this code.

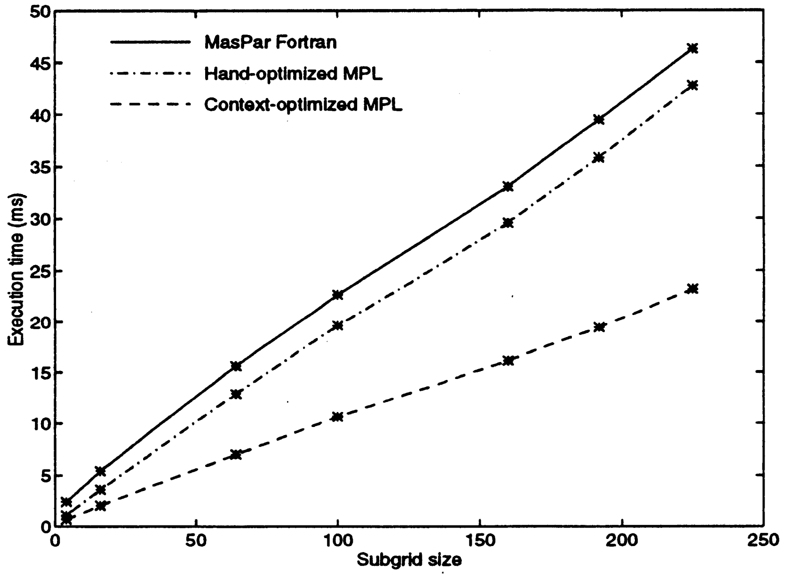


Figure 13 Time for ARPS weather code to initialize sixteen 2-D arrays.

To consider something more computationally interesting, we looked at a five-point difference computation:

```
RESULT = (A + CSHIFT(A,1,1) + CSHIFT(A,-1,1)
         + CSHIFT(A,1,2) + CSHIFT(A,-1,2))/5
```

RESULT and A were both two-dimensional arrays distributed in a (CYCLIC,CYCLIC) manner. We generated three versions of the code: a Fortran 90 version, a hand-optimized MPL version, and an MPL version which had context splitting applied (since there was only a single statement, context partitioning was not applicable). We then timed the subgrid loops. In all cases, the communication time to set up the computation was excluded from the measurements. The results are shown in Figure 14.

Since the time to compute and set the context is a smaller portion of the total work performed in this subgrid loop, the performance gain is not as impressive as that obtained on the array initialization code. But the 13% reduction in the execution time from the hand-optimized MPL version is still significant.

As a final point of interest, we took the above five-point difference computation and performed context splitting only on the innermost subgrid loop as discussed in Section 3.2.5. Code growth was minimal, adding only two statements to the MPL code: a call to Set_Context and a replication of the loop body (a single assignment statement). In comparison, the original split version, in which context splitting was applied to both loops in the subgrid loop nest, slightly more than doubled the amount of code. Additionally, the performance difference between the two split versions was minimal. The new split version reduced the execution time of the hand-optimized MPL version by 12%, compared to the 13% reduction of the original split version.

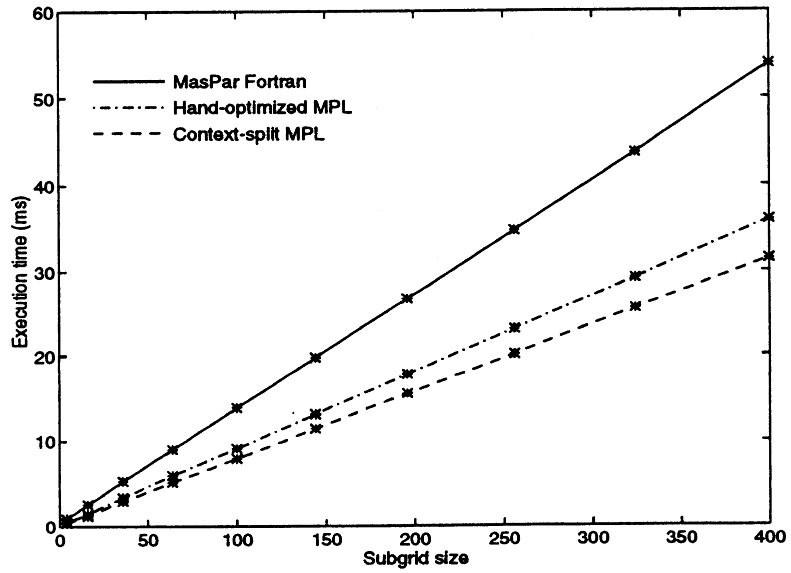


Figure 14 Time for 5-point difference computation.

5 Related Work

Work at Compass by Albert *et al.* describes the generation and optimization of context setting code [2]. They avoid redundant context computations when adjacent statements operate under the same context. They also perform classical optimizations on the context expressions, such as common subexpression elimination. They mention the possibility of reordering computations to minimize context changes, but they do not discuss such transformations.

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 code for which a single subgrid loop can be generated) [21]. 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 them as large as possible. In a later paper describing the internals of the compiler, he describes how it attempts to perform code motion so that subgrid loops may become adjacent and thus fused [20]. However, the code motion performed is limited to only moving scalar code from between subgrid loops, not in moving the loops themselves. Furthermore, due to the limited dependence analysis performed by the compiler, only compiler-generated scalar code will be moved. It was this work that motivated us to investigate the context partitioning problem.

Chen and Cowie also recognize the need to fuse parallel loops in their Fortran 90 compiler [10]. However, they only fuse adjacent loops and perform no code motion to increase the chances of fusion.

6 Summary

We have developed a double-edged sword to combat the cost of context switching in codes for SIMD machines. The first edge of the sword reduces the number of subgrid loops which operate over the same context to a minimum. The second edge reduces the number of context changes per subgrid loop from $O(N)$ to $O(1)$ for unmasked array assignment statements.

7 Acknowledgments

We would like to thank Chau-Wen Tseng for pointing out the similarities between context partitioning and loop fusion. We would also like to thank the members of the Fortran D group at Rice University who read early drafts of this paper. Finally, we thank NPAC, a CRPC facility, for providing the DECmpp used in the experiments.

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] E. Albert, K. Knobe, J. Lukas, and G. Steele, Jr. Compiling Fortran 8x array features for the Connection Machine computer system. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)*, New Haven, CT, July 1988.

[3] F. Allen and J. Cocke. A catalogue of optimizing transformations. In J. Rustin, editor, *Design and Optimization of Compilers.* Prentice-Hall, 1972.

[4] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Dept. of Computer Science, Rice University, April 1983.

[5] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

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

[7] U. Banerjee. *Speedup of ordinary programs.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1979. Report No. 79-989.

[8] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[9] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng. Optimal evaluation of array expressions on massively parallel machines. Technical Report CSL-92-11, Xerox Corporation, December 1992.

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

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

[12] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[13] M. Gerndt. Work distribution in parallel programs for distributed memory multiprocessors. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[14] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines.* North-Holland, Amsterdam, The Netherlands, 1992.

[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.

[16] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[17] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium*

on the Principles of Programming Languages, Williamsburg, VA, January 1981.

[18] 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.

[19] Y. Muraoka. *Parallelism Exposure and Exploitation in Programs.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, February 1971. Report No. 71-424.

[20] 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.

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

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

[23] 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.

[24] M. Weiss. Strip mining on SIMD architectures. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

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

[26] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.