Global Optimizations for Parallelism and Locality on Scalable Parallel Machines

Jennifer M. Anderson and Monica S. Lam
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

Data locality is critical to achieving high performance on large-scale parallel machines. Non-local data accesses result in communication that can greatly impact performance. Thus the mapping, or decomposition, of the computation and data onto the processors of a scalable parallel machine is a key issue in compiling programs for these architectures.

This paper describes a compiler algorithm that automatically finds computation and data decompositions that optimize both parallelism and locality. This algorithm is designed for use with both distributed and shared address space machines. The scope of our algorithm is dense matrix computations where the array accesses are affine functions of the loop indices. Our algorithm can handle programs with general nestings of parallel and sequential loops.

We present a mathematical framework that enables us to systematically derive the decompositions. Our algorithm can exploit parallelism in both fully parallelizable loops as well as loops that require explicit synchronization. The algorithm will trade off extra degrees of parallelism to eliminate communication. If communication is needed, the algorithm will try to introduce the least expensive forms of communication into those parts of the program that are least frequently executed.

1 Introduction

Minimizing communication by increasing the locality of data references is an important optimization for achieving high performance on all large-scale parallel machines. The long message-passing overhead of multicomputer architectures, such as the Intel Touchstone[17], makes minimizing communication essential. Locality is also important to scalable machines that support a shared address space in hardware. For example, local cache accesses on the Stanford DASH shared-memory multiprocessor are two orders of magnitude faster than remote accesses[26]. Improving locality can greatly enhance the performance of such machines.

The mapping of computation onto the processors of a parallel machine is termed the *computation decomposition* of the program. Similarly, the placement of data into the processors' local memories is called the *data decomposition*. This paper describes a compiler algorithm that automatically finds the computation and data decompositions that optimize both the parallelism and locality of a program. This algorithm is designed for use with both distributed and shared address space machines. For machines with a distributed address space, the compiler must follow this phase with a pass that maps the decomposition to explicit communication code[2]. While it is not necessary to manage the memory directly for machines with a shared address space, many of the techniques used to manage data on distributed memory machines can be used to improve cache performance.

The choices of data and computation decomposition are interrelated; it is important to examine the opportunities for parallelism and the reuse of data to determine the decompositions. For example, if the only available parallelism in a computation lies in operating on different elements of an array simultaneously, then allocating those elements to the same processor renders the parallelism unusable. The data decomposition dictated by the available parallelism in one loop nest affects the decision of how to parallelize the next loop nest, and how to distribute the computation to minimize communication. It may be advantageous to abandon some parallelism to create larger granularity tasks if the communication cost overwhelms the benefit of parallelization.

A popular approach to this complex optimization problem is to solicit the programmer's help in determining the data decompositions. Projects using this approach include SUPERB[40], AL[34], ID Noveau[31], Kali[22], Vienna Fortran[7] and Fortran D[14, 33]. The current proposal for a High Performance Fortran extension to Fortran 90 also relies upon user-specified data decompositions[13]. While these languages provide significant benefit to the programmer by eliminating the tedious job of managing the distributed memory explicitly, the programmer is still faced with a very difficult programming problem. The tight coupling between the mapping of data and computation means that the programmer must, in effect, also analyze the parallelization of the program when specifying the data decompositions. As the best decomposition may change based on the architecture of the machine, the programmer must fully master the machine details. Furthermore, the data decompositions may need to be modified to make the program run efficiently on a different architecture.

The goal of this research is to automatically derive the data and computation decompositions for the domain of dense matrix code where the loop bounds and array subscripts are affine functions of the loop indices and symbolic constants. Our algorithm finds decompositions for loops in which the number of iterations is much larger than the number of processors. The emphasis of this paper is on finding the first-order, or "shape", of the decompositions. We do not address issues such as load balancing, choosing the block size for a block-cyclic decomposition, determining the number of physical processors to lay out in each dimension, and fitting the computation

This research was supported in part by DARPA contract N00039-91-C-0138, an NSF Young Investigator Award and a fellowship from Digital Equipment Corporation's Western Research Lab.
In *Proceedings of SIGPLAN '93 Conference on Programming Language Design and Implementation* (PLDI)
Albuquerque, New Mexico, June 23–25, 1993

and data to the exact number of physical processors. Even though these issues impact the performance of parallel machines, their effect is secondary and we do not address them in this paper.

We have developed a mathematical framework for expressing and calculating decompositions. This framework is general enough to handle a broad class of array access patterns, including array sections, and is also used to calculate the replication of read-only data. As there are many possible decompositions for a program, a systematic solution must successfully reduce this complex problem into a manageable one. Our model is based on the property that equivalent decompositions have the same data and computation allocated to a single processor. Once this aspect of the decomposition has been determined, we show that an assignment to specific processors can easily be calculated.

The cost of communication is determined by the data movement pattern. If the communication pattern is nearest-neighbor shifts of data, then the amount of data transferred can be significantly reduced by blocking. This form of communication is inexpensive compared to communication patterns that require general movement of the entire data structure (e.g. a transpose). We further differentiate between communication that occurs within a parallel loop with explicit synchronization, or across loops due to mismatches in decompositions. We call communication within a loop nest *pipelined* communication. Communication due to mismatches in decompositions, and that require moving the entire data structure, is called *data reorganization* communication. If a single data decomposition can be found for an array such that there is no reorganization communication in the program, then we consider that decomposition to be static (even though there may be some minor nearest-neighbor communication between parallel loop nests).

Section 2 briefly presents the background on optimizing parallelism and locality within a loop nest. We then introduce the issues involved in automatically calculating decompositions, and formulate the problem mathematically. Section 3 describes the components of a decomposition and gives an overview of our approach. To illustrate the basic ideas behind our decomposition model, we first discuss a simplified subproblem in Section 4. We present an algorithm that finds data and computation decompositions that have neither data reorganization nor pipelined communication. We then reapply the concepts to find decompositions with pipelined communication in Section 5. Section 6 uses the algorithms in Section 4 and 5 as building blocks to develop an algorithm that takes into account both pipelined and data reorganization communication. Section 7 presents additional techniques for handling replication, and for minimizing the number of idle processors and the amount of replication. We have implemented the algorithms described in this paper in the SUIF compiler at Stanford. Section 8 describes some experimental results using the compiler. Section 9 discusses related work, and we conclude in Section 10 with a summary of the contributions of this paper.

2 Problem Overview

This section briefly discusses optimizations for parallelism and locality within a single loop nest, and introduces the issues involved in finding decompositions by way of a simple example. After presenting a mathematical formulation of decompositions, we then formally state the problem.

2.1 Background

Techniques for maximizing parallelism and locality within a single loop nest have been presented in the literature[20, 25, 36]. A number of researchers have also looked at the specific problem of mapping a single loop nest onto parallel machines[15, 23, 24]. We refer to such loop-level techniques as *local* analysis. The *global* analysis is responsible for optimizing parallelism and locality across multiple loop nests.

First, our compiler normalizes the loops and performs loop distribution before executing the decomposition algorithms[1]. The compiler runs a loop fusion pass after decomposition to regroup compatible loop nests[5, 10].

Our compiler uses the algorithm developed by Wolf and Lam[25, 36] to apply unimodular transforms to find the coarsest granularity of parallelism within a loop nest. This pass leaves the loop nests in a canonical form consisting of a nest of *fully permutable* loop nests. The nests are as large as possible, starting from the outermost loops. A loop nest is fully permutable if any arbitrary permutation of the loops within the nest is legal. A fully permutable loop nest of depth j can be transformed to get $j-1$ degrees of parallelism. Our compiler positions the loops in each fully permutable nest such that any parallel loops are outermost. For example, given the following code:

(1) **for** i_1 := 0 **to** N **do**
    for i_2 := 0 **to** N **do**
        Y[i_1,$N-i_2$] += X[i_1,i_2];

(2) **for** i_2 := 1 **to** N **do**
    for i_1 := 1 **to** N **do**
        Z[i_1,i_2] := Z[i_1,i_2-1] + Y[i_2,i_1-1];

The local analysis would produce the code shown at the top of Figure 1. The keyword **forall** indicates that the iterations of the loop can be executed in parallel.

For a loop nest of depth l, let i_k be the outermost parallelizable loop (the first loop in the outermost fully permutable loop nest of size greater than 1). Loops $i_1 \ldots i_{k-1}$ are sequential (degenerate fully permutable nests of size 1), thus there must be dependences between iterations of these loops. The local phase is responsible for finding transformations that minimize communication of loops $i_k \ldots i_l$ with respect to the outer sequential loops. Parallelizable loops are allocated such that any neighboring loops in the iteration space are neighbors when mapped onto the processor space.

2.2 A Simple Example

Consider the code shown at the top of Figure 1. In the figure, the array elements at $(0,0)$ and $(0,N)$ are shaded light grey and dark grey, respectively, to identify the position of the arrays. The loop iterations are shaded similarly. The figure assumes that arrays are stored in row-major order. A naive approach that considers each loop nest individually would distribute the outermost loop i_1 in the first loop nest, to get the coarsest granularity of parallelism. Each processor then accesses rows of arrays X and Y. In the second loop nest, the parallel i_1 loop would be distributed, and each processor accesses columns of array Y and rows of array Z. Communication will occur, since a processor accesses a different section of array Y in each of the two loop nests. A solution that has no communication is to only parallelize the i_2 loop in the first loop nest. Each processor would then access columns of X, rows of Z, and columns of Y in both loop nests. The loop nests must also be analyzed to determine the relative positions of the arrays so that no communication is necessary. A complete communication-free decomposition is shown in Figure 1(c). The rest of the figure shows the mathematical representation of the decompositions and will be discussed in later sections.

(1) **forall** $i_1 := 0$ **to** N **do**
    forall $i_2 := 0$ **to** N **do**
        Y[i_1,$N - i_2$] += X[i_1,i_2];

(2) **forall** $i_1 := 1$ **to** N **do**
    for $i_2 := 1$ **to** N **do**
        Z[i_1,i_2] := Z[i_1,$i_2 - 1$] + Y[i_2,$i_1 - 1$];

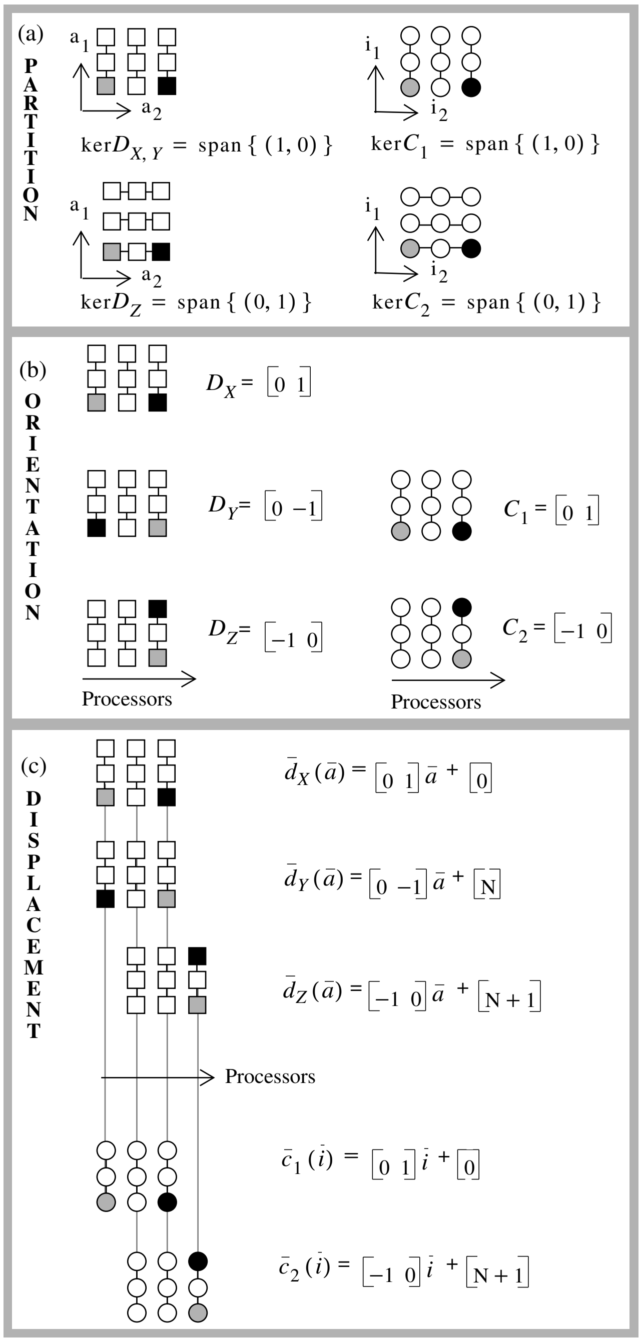

Figure 1: A simple decomposition example. Squares represent array elements and circles represent iterations. Lines connect the array elements and iterations that are allocated to the same processor.

2.3 Problem Formulation

This section presents a mathematical model of the decomposition problem. We represent data and computation decompositions as affine transformations. In this discussion, all loops are normalized to have a unit step size, and all arrays subscripts are adjusted to start at 0. A loop nest of depth l, with loop bounds that are affine functions of the loop indices, defines an iteration space $\mathcal{I}$, a polytope in l-dimensional space. Each iteration of the loop nest corresponds to an integer point in the polytope and is identified by its index vector $\vec{\imath} = (i_1, i_2, \ldots, i_l)$. An array of dimension m defines an array space $\mathcal{A}$, an m-dimensional rectangle. Each element in the array is accessed by an integer vector $\vec{a} = (a_1, a_2, \ldots, a_m)$. Similarly, an n-dimensional processor array defines a processor space $\mathcal{P}$, an n-dimensional rectangle. We write an affine array index function $\vec{f} : \mathcal{I} \to \mathcal{A}$ as $\vec{f}(\vec{\imath}) = F\vec{\imath} + \vec{k}$, where F is a linear transformation and $\vec{k}$ is a constant vector.

Definition 2.1 *For each index $\vec{a}$ of an m-dimensional array, the* data decomposition *of the array onto an n-dimensional processor array is a function $\vec{d}(\vec{a}) : \mathcal{A} \to \mathcal{P}$, where*

$$\vec{d}(\vec{a}) = D\vec{a} + \vec{\delta}$$

D is an $n \times m$ linear transformation matrix and $\vec{\delta}$ is a constant vector.

Definition 2.2 *For each iteration $\vec{\imath}$ of a loop nest of depth l, the* computation decomposition *of the loop nest onto an n-dimensional processor array is a function $\vec{c}(\vec{\imath}) : \mathcal{I} \to \mathcal{P}$, where*

$$\vec{c}(\vec{\imath}) = C\vec{\imath} + \vec{\gamma}$$

C is an $n \times l$ linear transformation matrix and $\vec{\gamma}$ is a constant vector.

The problem can now be stated formally as follows. We want to find the computation decomposition $\vec{c}(\vec{\imath})$ for each loop nest, and the data decomposition $\vec{d}(\vec{a})$ for each array in each loop nest, such that parallelism is maximized and communication is minimized. The formal decompositions for the simple example from the previous section are shown in Figure 1(c).

3 Basic Concepts

The problem of finding data decompositions $\vec{d}(\vec{a}) = D\vec{a} + \vec{\delta}$ and computation decompositions $\vec{c}(\vec{\imath}) = C\vec{\imath} + \vec{\gamma}$ can be broken down into three distinct components using the affine model. The *partition* determines the computation and data that are allocated to the same processor. Mathematically, data and computation partitions are described by the nullspace of the matrices D and C from Definitions 2.1 and 2.2. The *orientation*, represented by the matrices D and C, describes the mapping between the axes of the array elements and loop iterations, and the processors. Lastly, the *displacement* gives the offset of the starting position of the data and computation, and corresponds to the constant vectors $\vec{\delta}$ and $\vec{\gamma}$. We illustrate the partition, orientation and displacement by developing a communication-free decomposition for the sample program in Figure 1.

The Partition. There is a data dependence of (0,1) in loop nest 2 which serializes the i_2 loop. No communication is necessary when all the elements in each column of array Y and each row of array Z are assigned to the same processor. Since the elements in each column of array Y are on the same processor, iterations of i_1 in loop nest 1 are also assigned to the same processor and execute sequentially (even though there are no dependences in loop nest 1). In turn, the columns of array X are allocated to the same processor as well. The partitions for this example are shown in Figure 1(a). Informally, the data and computation partitions specify which array elements and iterations, respectively, are assigned to the same processor, but *not* which processor.

Formally, the subspace of the array space accessed by an array referenced in a loop nest is denoted by S and is the range of the array index matrix F:

$$S = \text{range}(F) \tag{1}$$

For an array of dimension m, whenever $\text{rank}(F) < m$, then $S \subset \mathcal{A}$.

Let D be the data decomposition matrix from Def. 2.1. Two array elements $\vec{a}_1, \vec{a}_2 \in S$ are allocated to the same processor if and only if

$$D\vec{a}_1 = D\vec{a}_2,$$

that is,

$$D(\vec{a}_1 - \vec{a}_2) = 0, \text{ or } \vec{a}_1 - \vec{a}_2 \in \ker D.$$

Conversely, any two array elements such that $(\vec{a}_1, \vec{a}_2 \in S) \wedge (\vec{a}_1 - \vec{a}_2 \notin \ker D)$ may be assigned to different processors and are considered distributed.

Let C be the computation decomposition matrix from Def. 2.2. Two iterations $\vec{\imath}_1, \vec{\imath}_2 \in \mathcal{I}$ are executed on the same processor if and only if

$$C\vec{\imath}_1 = C\vec{\imath}_2$$

that is,

$$C(\vec{\imath}_1 - \vec{\imath}_2) = 0, \text{ or } \vec{\imath}_1 - \vec{\imath}_2 \in \ker C.$$

Any two iterations $\vec{\imath}_1, \vec{\imath}_2 \in \mathcal{I}$ such that $\vec{\imath}_1 - \vec{\imath}_2 \notin \ker C$ are said to be distributed and may run on different processors. The mathematical representation of the partitions for the example is also shown in Figure 1(a). The data partitions for X and Y indicate that all array elements along the direction $(1, 0)$ (i.e. each column) are assigned to the same processor. Similarly, the data partition for Z means that all elements along the direction $(0, 1)$ are assigned to the same processor. The computation partitions indicate that all iterations of the i_1 loop in the first loop nest, and all iterations of the i_2 loop in the second loop nest, are executed on the same processor.

The Orientation. The partition determines which array elements and iterations are local to a single processor. The orientation together with the displacement can now specify the processor on which the data and computation are allocated. In particular, the orientation gives the correspondence between the data and computation dimensions and the processor dimensions. In loop nest 1, the columns of array Y are accessed in the reverse order from the columns of X. In loop nest 2, the columns of array Y are accessed in the same order as the rows of array Z. One solution that satisfies all these requirements is to allocate the columns of X in forward order, and the columns of Y and the rows of Z in reverse order. The iterations of the i_1 loop in loop nest 2 must now be reversed as well. The orientation is illustrated in Figure 1(b).

Formally, the matrix D from Def. 2.1 defines the data orientation and the matrix C from Def. 2.2 is the computation orientation. The matrices for the example are also shown in Figure 1(b). Note that there exist many different communication-free orientations, all with the same partition. For this example, we could just have easily chosen to allocate the columns of X in reverse order, and the columns of Y and the rows of Z in forward order. This alternative orientation would result in $D_X = [0\ -1]$, $D_Y = [0\ 1]$ and $D_Z = [1\ 0]$, with $C_1 = [0\ -1]$ and $C_2 = [1\ 0]$.

The Displacement. The displacement specifies the offsets of the array elements and iterations with respect to the processors. In loop nest 2, accesses by the i_1 loop to the columns of array Y are offset by one from the rows of array Z. In loop nest 1, accesses to arrays X and Y have no offset. Assigning columns $0..N$ of array X on processors $0..N$, the columns of Y on processors $N..0$ and the rows Z on processors $N+1..1$ satisfies this requirement. Iterations $1..N$ of loop i_1 in the second loop nest are then assigned to processors $N..1$. The complete decompositions with displacements are illustrated in Figure 1(c).

Formally, the displacements $\vec{\delta}$ and $\vec{\gamma}$ are the constant vectors from Definitions 2.1 and 2.2, respectively. The orientation matrix derived from the partition, plus the displacement forms the complete decomposition. Figure 1(c) also shows the data and computation displacements and the final decompositions for the example. As was the case with orientations, there are also many possible displacements that lead to communication-free decompositions.

We can now summarize the basis of our approach. There are many different, yet equivalent, decompositions with the same partition. We reduce the complexity of finding the decomposition functions $\vec{d}(\vec{a})$ for each array and $\vec{c}(\vec{\imath})$ for each loop nest by first finding a partition that is *guaranteed* to lead to the desired decomposition. Then a simple calculation can be used to find the appropriate orientations and displacements that completely specify the decompositions.

4 Static Decompositions

In this section, we present an algorithm to find data and computation decompositions that have neither pipelined communication nor data reorganization communication. This simplified problem illustrates the basic ideas of our decomposition model. The algorithm finds a single, static decomposition for each array and each loop nest, and only considers the parallelism available in **forall** loops.

4.1 Relationship Between Data and Computation

No communication will occur when the data is local to the processor that references that data. This relationship between data and computation is expressed by the following theorem.

Theorem 4.1 *Let the computation decomposition for loop nest j be $\vec{c}_j$ and the data decomposition for array x be $\vec{d}_x$. Let $\vec{f}_{xj}$ be an array index function for array x in loop nest j. For all iterations $\vec{\imath}$, the elements of the array will be local to the processor that references those elements if and only if*

$$D_x(\vec{f}_{xj}(\vec{\imath})) + \vec{\delta}_x = C_j(\vec{\imath}) + \vec{\gamma}_j \tag{2}$$

Communication at the displacement level is inexpensive since the amount of data transferred can be significantly reduced by blocking. Thus the priority is in finding the best partitions and orientations, and we first focus on the version of Eqn. 2 that omits displacements. Letting $\vec{f}_{xj}(\vec{\imath}) = F_{xj}(\vec{\imath}) + \vec{k}_{xj}$,

$$D_x F_{xj}(\vec{\imath}) = C_j(\vec{\imath}) \tag{3}$$

Given the array index function matrices F_{xj} for each array x in each loop nest j, a decomposition is free of reorganization communication if the data decomposition matrix D_x and the computation decomposition C_j are such that Eqn. 3 is true. A trivial solution that guarantees no communication is to execute everything sequentially by setting all computation decomposition matrices $C = 0$ and all data decomposition matrices $D = 0$. Therefore $\ker C$ would span the entire iteration space $\mathcal{I}$ and $\ker D$ would span the entire array space $\mathcal{A}$. However, maximizing parallelism means finding data and computation decompositions such that the partition $\ker C$, the nullspace of C, for all loop nests is as small as possible.

4.2 Partition Constraints

We first consider the base case of at most one outer sequential loop containing a number of perfectly nested loops. We assume that the local phase has analyzed and transformed the perfectly nested loops individually into canonical form. In section 6.4 we will discuss the general case of multiple nesting levels.

A collection of loops nests and arrays is represented as a bipartite *interference graph*, $G = (V_c, V_d, E)$. The loop nests form one set of vertices V_c, and the arrays form the other set of vertices V_d. There is an undirected edge $e \in E$ between an array and a loop nest if the array is referenced in the loop nest. Each edge contains one or more array index functions for all accesses of the array in the loop nest.

Each connected component of the interference graph corresponds to a set of arrays and loop nests that have inter-related decompositions. The algorithms presented later in this section operate on a single connected component at a time.

To find a static decomposition, the following constraints are placed on the data and computation partitions.

(1) Single Loop Nest. The constraints on a single loop nest characterize the loops that are assigned to the same processor. These constraints are used to initialize the computation partition for each loop nest. As this algorithm considers only **forall** loops, the initial computation partition for a loop nest of depth l is the span of a set of l-dimensional elementary basis vectors[1] representing the sequential loops in the loop nest. If a loop at nesting level k is sequential, then e_k is included in the initial computation partition.

(2) Multiple Arrays. The role of the constraints due to multiple arrays is to ensure that there exists a single data decomposition matrix D for each array. These constraints are used to initialize the data partitions. Such constraints are necessary if there are two distinct paths in the interference graph from an array x to another array y. For example, consider the following code fragment:

```
(1) forall i1 := 0 to N do
      forall i2 := 0 to N do
        X[i1,i2] += Y[i1,i2];

(2) forall i1 := 0 to N do
      forall i2 := 0 to N do
        Y[i2,i1] := X[i1,i2];
```

[1] The kth elementary vector, written e_k, has a 1 in the kth position and zero in all other positions.

The array index functions for X and Y in the first loop nest are $F_{X1} = F_{Y1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. The array index functions for X and Y in the second loop nest are $F_{X2} = F_{X1}$ and $F_{Y2} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, respectively.

A decomposition that is free of reorganization communication will have decomposition matrices D_X and D_Y (for arrays X and Y, respectively) such that Eqn. 3 holds for some C_1 and C_2 (for loop nests 1 and 2, respectively). For iterations $\vec{\imath}_1$ in loop nest 1 and $\vec{\imath}_2$ in loop nest 2, we have the following:

$$D_X F_{X1}(\vec{\imath}_1) = D_Y F_{Y1}(\vec{\imath}_1) = C_1(\vec{\imath}_1)$$
$$D_X F_{X2}(\vec{\imath}_2) = D_Y F_{Y2}(\vec{\imath}_2) = C_2(\vec{\imath}_2)$$

Since the array index functions are invertible, these equations produce the following equations for D_Y: $D_Y = D_X F_{X1} {F_{Y1}}^{-1}$ and $D_Y = D_X F_{X2} {F_{Y2}}^{-1}$. Thus,

$$D_X(F_{X1}{F_{Y1}}^{-1} - F_{X2}{F_{Y2}}^{-1}) = \vec{0}. \tag{4}$$

and $(F_{X1}{F_{Y1}}^{-1} - F_{X2}{F_{Y2}}^{-1}) \in \ker D_X$.

If all the array access functions for each array are equal then the above equation will yield $D_X\vec{0} = \vec{0}$ and $\vec{0} \in \ker D_X$, and no additional constraints are placed on the partitions. In the example, Eqn. 4 is $D_X\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\right) = \vec{0}$. Thus, $D_X\left(\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\right) = \vec{0}$. Simplifying the equation gives a constraint on the partition of array X: $\ker D_X \supseteq \text{span}\{(1,-1)\}$. Similar analysis yields the same constraint on the partition of array Y: $\ker D_Y \supseteq \text{span}\{(1,-1)\}$. This partition means that elements along the diagonal are allocated to the same processor. In general, when the array index functions are not invertible, we must introduce auxiliary variables and use a pseudo-inverse function. The techniques we use are similar to those presented in other literature [3, 29].

This analysis is run on all pairs of arrays involved in a cycle in the interference graph (including the degenerate case of multiple access functions for one array in a loop nest). If an array is involved in multiple cycles and multiple constraints are found, then the constraints are summed. In general, when computing the data constraints on an array used in multiple loop nests, it is possible that the loop nests access different subsections of the array. If this is the case, each loop nest only contributes to the constraints for the section of the array that it references.

(3) Data-Computation Relation. This constraint ensures that the relationship between data and computation from Eqn. 3 holds. If two iterations $\vec{\imath}_1$ and $\vec{\imath}_2$ in loop nest j are mapped to the same processor, then the data of array x they access must also be mapped to the same processor. For $\vec{t} = \vec{\imath}_1 - \vec{\imath}_2$, then from section 3, $\vec{t} \in \ker C_j$. Using Eqn. 3: $D_x F_{xj}\vec{t} = C_j\vec{t} = \vec{0}$ and thus $F_{xj}\vec{t} \in \ker D_x$. Formally,

$$\ker D_x \supseteq \text{span}\{\vec{s} \mid \vec{s} = F_{xj}\vec{t}, \vec{t} \in \ker C_j\} \tag{5}$$

Similarly, two iterations $\vec{\imath}_1$ and $\vec{\imath}_2$ in loop nest j must be mapped to the same processor if the data of array x they access are mapped to the same processor. Again, let $\vec{t} = \vec{\imath}_1 - \vec{\imath}_2$. If $\vec{t} \in \ker(D_x F_{xj})$ then $\vec{t} \in \ker C_j$. Since $\ker(D_x F_{xj}) \supseteq \ker F_{xj}$, if $\vec{t} \in \ker F_{xj}$ then the two iterations reference the same array location and must be mapped to the same processor. Let $S_{xj} = \text{range}(F_{xj})$. In general,

$$\ker C_j \supseteq \text{span}\{\vec{t} \mid (\vec{t} \in \ker F_{xj}) \vee (F_{xj}\vec{t} \in (\ker D_x \cap S_{xj}))\} \tag{6}$$

The sequential loops in each loop nest cause elements of the array referenced in that loop nest to be allocated local to the same processor. The local array elements cause iterations of the loop nests that access those elements to be executed sequentially.

4.3 Calculating Partitions

To find partitions that maximize parallelism and have neither pipelined nor data reorganization communication, we find the minimum partitions that satisfy constraints $(1)-(3)$. Constraint 1 (single loop) is used to initialize the computation partitions and constraint 2 (multiple arrays) is used to initialize the data partitions. An iterative algorithm is used to satisfy constraint 3 (data-computation relationship). An overview of this algorithm is shown in Figure 2.

---

algorithm Update_Arrays
　　(j : Loop_Nest;
　　IG : Interference_Graph; /* *IG* = (V_c, V_d, E) */
　　DP_Set : **set of** vector_space)

　foreach $x \in$ referenced_in(j) **do**
　　$\ker D_x := \ker D_x + \text{span}\{\vec{s} \mid \vec{s} = F_{xj}\vec{t}, \vec{t} \in \ker C_j\}$;
　end foreach;
end algorithm;

algorithm Update_Loops
　　(x : Array;
　　IG : Interference_Graph; /* *IG* = (V_c, V_d, E) */
　　CP_Set : **set of** vector_space)

　foreach $j \in$ loops_using(x) **do**
　　$\ker C_j := \ker C_j +$
　　　$\text{span}\{\vec{t} \mid (\vec{t} \in \ker F_{xj}) \vee (F_{xj}\vec{t} \in (\ker D_x \cap S_{xj}))\}$;
　end foreach;
end algorithm;

algorithm Calc_Relation
　　(*IG* : Interference_Graph; /* *IG* = (V_c, V_d, E) */
　　CP_Set : **set of** vector_space;
　　DP_Set : **set of** vector_space)

　while changes **do**
　　if changed($x \in V_d$) **then** Update_Loops(x,*IG,CP_Set*);
　　if changed($j \in V_c$) **then** Update_Arrays(j,*IG,DP_Set*);
　end while;
end algorithm;

algorithm Partition
　　(*IG* : Interference_Graph; /* *IG* = (V_c, V_d, E) */
　　　/* Computation and data partitions */
　　CP_Set : **set of** vector_space;
　　DP_Set : **set of** vector_space)

　　/* Satisfy constraints */
　　foreach $j \in V_c$ **do** $\ker C_j$:= single_loop_constraint(j);
　　multiple_loop_constraint(*IG, DP_Set*);
　　Calc_Relation(*IG,CP_Set,DP_Set*);
end algorithm;

Figure 2: Algorithm for calculating partitions.

The iterative algorithm calculates the effects of the loop nests on the arrays using Eqn. 5 and of the arrays on the loop nests using Eqn. 6. This continues until a stable partition is found. Informally, the partition algorithm trades off extra degrees of parallelism to eliminate communication. Going back to the simple example in Figure 1, the array index functions for arrays X and Z are $F_{X1} = F_{Z2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. The index functions for array Y in the first and second loop nests are $F_{Y1} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ and $F_{Y2} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, respectively. $\ker C_1$ is initialized to $\emptyset$ and $\ker C_2$ is initialized to $\text{span}\{(0,1)\}$. The data partitions $\ker D_{X,Y,Z}$ are initialized to $\emptyset$. The routine *Update_Arrays* is called with loop nest 2. Eqn. 5 is applied to arrays Y and Z, resulting in $\ker D_Y = \text{span}\{(1,0)\}$ and $\ker D_Z = \text{span}\{(0,1)\}$. Next, routine *Update_Loops* is called with arrays Y and Z. Because of array Y, Eqn. 6 is applied to loop nest 1, resulting in $\ker C_1 = \text{span}\{(1,0)\}$. Finally, *Update_Arrays* is called again with loop nest 1 and $\ker D_X = \text{span}\{(1,0)\}$.

Lemma 4.2 *The partition algorithm finds the maximum parallelism (minimum partitions) that satisfy constraints 1–3, and is guaranteed to terminate.*

Proof: The partition algorithm satisfies constraints 1 and 2 because the data and computation partitions are initialized with these constraints. The algorithm finds the minimum partition that satisfies the data-computation relation constraint 3 because the algorithm only ever increases the partitions in order to ensure that the constraint is satisfied. To prove termination, we use the fact that the spaces $\ker D$ (data partitions) and $\ker C$ (computation partitions) increase in size monotonically as the algorithm progresses. In the worst case, the partitions will span the entire space and the algorithm will terminate. □

After a data partition has been found for each array and a computation partition for each loop nest, the next step is to determine the number of virtual processor dimensions. The number of virtual processor dimensions n is

$$n = \max_{x \in \text{Arrays}} (\dim(S_x) - \dim(\ker D_x))$$

Here $S_x = \sum_{\forall j \in \text{loops\_using}(x)} \text{range}(F_{xj})$ is the total array space accessed, typically the entire array. This equation will yield a value of n such that all the parallelism found in the partition algorithm is exploited. In the example from Figure 1, $n = 1$.

4.4 Calculating Orientations

Once the partition and number of virtual processor dimensions have been found, the algorithm finds the orientations. The partitions determine the kernels of each of the decomposition matrices. Since the orientations in a connected component of the interference graph are all relative to one another, we can choose *one* arbitrary decomposition matrix and derive the rest of the decomposition matrices in the component. The algorithm starts by choosing an $n \times m$ data decomposition matrix D_x for an array x of dimension m such that the nullspace of D_x is the data partition $\ker D_x$. According to Eqn. 3, the computation decomposition matrix for a loop nest j that references the array is $C_j = D_x F_{xj}$. Again, for simplicity of presentation we assume that the array index functions are invertible. The data decomposition matrix, D_y, for another array y accessed in the same loop nest is calculated using $D_y = C_j F_{yj}^{-1} = D_x F_{xj} F_{yj}^{-1}$. The remaining decompositions in the connected component are calculated in a similar fashion. When an array index function only accesses a subsection of the array (i.e. $S_{yj} \subset \mathcal{A}_y$), auxiliary variables are used temporarily in the unspecified dimensions of the data decomposition matrix. Note that when calculating the orientations, non-integer entries in the decomposition matrices can result. Because orientations are relative, the matrices can be multiplied by the least common multiple to eliminate the fractions.

Lemma 4.3 *The orientation algorithm finds decomposition matrices for all arrays x and all loop nests j that have exactly the nullspace found by the partition algorithm, and such that $D_x F_{xj} = C_j$.*

Proof: We only outline the proof here because of space considerations. We prove this lemma by induction. The base case is the array x for which we chose an arbitrary decomposition matrix that has the specified kernel. Using partition constraints 2 and 3, we then show that as each decomposition matrix is calculated, it has the correct nullspace and $D_x F_{xj} = C_j$ holds. □

Theorem 4.4 *The partition and orientation algorithms together find decomposition matrices for all arrays and all loop nests that maximize parallelism when there is no communication within loops and no reorganization communication across loops.*

Proof: This theorem follows directly from Lemmas 4.2 and 4.3. □

4.5 Calculating Displacements

As we expect communication at the displacement level to be relatively inexpensive nearest-neighbor communication, we do not consider sacrificing parallelism to avoid communication due to displacements. However, the algorithm minimizes any communication caused by conflicting displacements whenever possible. The displacements are calculated after the partitions and orientations have already been determined. Our compiler uses a simple greedy strategy that takes into account branch predictions and the offset sizes to find displacements that minimize communication along the most frequently executed paths. Eqn. 2 says that given the full data decomposition, $D_x + \vec{\delta}_x$, for array x referenced in a loop nest j (with the array index function $F_{xj}(\vec{\imath}) + \vec{k}_{xj}$) the computation displacement $\vec{\gamma}_j = D_x \vec{k}_{xj} + \vec{\delta}_x$. The data displacement, $\vec{\delta}_y$, for another array y accessed in the same loop nest can be calculated using $\vec{\delta}_y = \vec{\gamma}_j - D_y \vec{k}_{yj}$.

5 Blocked Decompositions

In this section we discuss the problem of finding data and computation decompositions that have pipelined communication, but no data reorganization communication.

The previous section only considered the parallelism available in **forall** loops. However, it may be the case that it is not possible to legally transform the iteration space so that there are outermost **forall** loops. For example, consider the four point difference operation:

for i_1 := 1 **to** $N-1$ **do**
  for i_2 := 1 **to** $N-1$ **do**
    X[i_1,i_2] := f(X[i_1,i_2], X[i_1-1,i_2] + X[i_1+1,i_2] +
        X[i_1,i_2-1] + X[i_1,i_2+1]);

Here the parallelism is only available along a *wavefront*, or diagonal, of the original loop nest. Tiling[37, 39] (also known as blocking, unroll-and-jam and stripmine-and-interchange) is a well-known transformation that allows both parallelism and locality to be exploited within a loop nest.

The original iteration space is shown in Figure 3(a) and Figure 3(b) demonstrates how this loop can be executed in parallel using **doacross** parallelism. The iterations in each shaded block are assigned to different processors. The computation proceeds

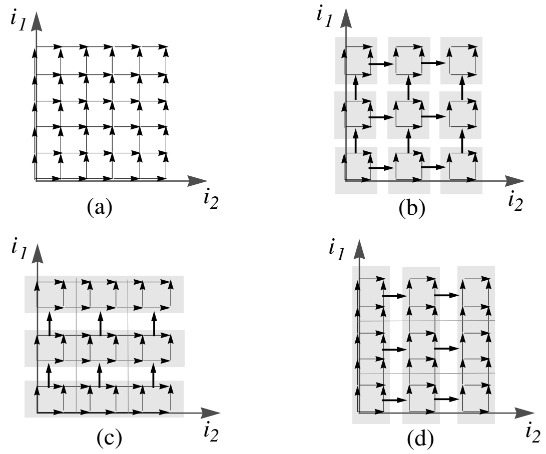

Figure 3: (a) Original iteration space. (b)–(d) Iteration spaces showing the parallel execution of tiled loops. The arrows represent data dependences.

along the wavefront dynamically, by using explicit synchronization to enforce the dependences between the blocks.

When all dimensions of the iteration space are blocked, there will be idle processors as only blocks along the diagonal can execute in parallel. We can gain the advantages of tiling without idle processors by assigning entire rows (Figure 3(c)) or columns (Figure 3(d)) to different processors. In these two cases, each processor is assigned a *strip* of the iteration space, and all processors can start executing in parallel. For example, the tiled code that corresponds to Figure 3(d) is as follows:

for ii_2' := 1 **to** $N-1$ **by** B **do**
  for i_1 := 1 **to** $N-1$ **do**
    for i_2' := ii_2' **to** min($N-1$,$ii_2'+B-1$) **do**
    X[i_1,i_2'] := f(X[i_1,i_2'], X[i_1-1,i_2'] + X[i_1+1,i_2'] +
        X[i_1,$i_2'-1$] + X[i_1,$i_2'+1$]);

When this loop nest is tiled, the original i_2 loop is split into two dimensions: the outer ii_2' loop and the inner i_2' loop. Allocating each shaded strip from Figure 3(d) to a different processor spreads iterations of the ii_2' loop across processors, while iterations of the i_2' loop reside on the same processor.

Tiling can also be used to reduce communication across loop nests, even when **forall** parallelism is available in both nests. Consider the following example of an ADI (Alternating Direction Implicit) integration:

(1) **forall** i_1 := 0 **to** N **do**
    for i_2 := 1 **to** N **do**
      X[i_1,i_2] := f_1(X[i_1,i_2], X[i_1,i_2-1]);

(2) **for** i_1 := 1 **to** N **do**
    forall i_2 := 0 **to** N **do**
      X[i_1,i_2] := f_2(X[i_1,i_2], X[i_1-1,i_2]);

In the first loop nest, the sequential loop accesses columns of X. In the second loop nest, the sequential loop accesses rows of X. For there to be no communication, then the data partition for X must be $\ker D_X = \text{span}\{(0,1),(1,0)\}$, and the computation partition for both loops must be $\ker C_{1,2} = \text{span}\{(0,1),(1,0)\}$. This partition specifies that the entire array X is allocated on the same processor, and that both loops run sequentially. Only considering the parallelism available in the **forall** loops provides only two options: either run the loops sequentially, or incur data reorganization communication between the two loops.

However, if the compiler tiles both loops to extract wavefront parallelism, then the reorganization communication is reduced to inexpensive pipelined communication. A tiled version of the ADI code is shown below.

(1) **for** ii'_2 := 1 **to** N **by** B **do**
  for i_1 := 0 **to** N **do**
    for i'_2 := ii'_2 **to** min(N,$ii'_2 + B - 1$) **do**
      X[i_1,i'_2] := f_1(X[i_1,i'_2], X[i_1,$i'_2 - 1$]);

(2) **for** ii'_2 := 0 **to** N **by** B **do**
  for i_1 := 1 **to** N **do**
    for i'_2 := ii'_2 **to** min(N,$ii'_2 + B - 1$) **do**
      X[i_1,i'_2] := f_2(X[i_1,i'_2], X[$i_1 - 1$,i'_2]);

In both loop nests, the outer ii'_2 is distributed across processors, and the inner i_1 and i'_2 loops are executed on the same processor. Therefore, each processor is assigned a block of columns of the array. In the first loop nest, there are dependences across the blocks and there is pipelined communication within the loop nest. In the second loop nest, the data dependences are within the block so no communication is necessary.

5.1 Blocked Decomposition Model

Our decomposition model is easily extended to incorporate the concept of tiling. In general, tiling creates two sets of loops: the inner loops iterate within the block and the outer loops iterate across the blocks. The inner loops are allocated to the same processor, while the outer loops are distributed across the processors. In this way, we achieve locality within the block, and parallelism across the blocks.

Mathematically we have represented the computation that is allocated to the same processor as a vector space $\ker C$. Focusing now on the loops within an inner block, the iterations that are allocated to same processor also form a vector space, L_c. The vector space L_c is called the *localized vector space* in [37], where L_c is used to represent tile iterations that have cache locality. In our model the localized vector space L_c contains all dimensions of the iteration space that are local to a processor, be they completely local or blocked. Thus $\ker C \subseteq L_c$. Any dimension of the iteration space that is in $L_c - \ker C$ is blocked. Only the iterations within a finite block are allocated to the same processor, not the entire dimension. The blocks themselves are then distributed across the processors.

Similarly, we define a vector space L_d to characterize the array dimensions within a block that are allocated to the same processor. The relationship between the data partition $\ker D$ and space L_d is $\ker D \subseteq L_d$.

In the ADI example, the blocked computation partitions are $\ker C_{1,2} = \emptyset$ and $L_{c_{1,2}} = \text{span}\{(0,1),(1,0)\}$. Similarly, the blocked data partition is $\ker D_X = \emptyset$ and $L_{d_X} = \text{span}\{(0,1),(1,0)\}$.

Our algorithm finds the computation and data partitions $\ker C$ and $\ker D$; these spaces correspond to those dimensions that must be entirely mapped onto the same processor. If blocking is desired, the algorithm also finds L_c and L_d; the iterations in $L_c - \ker C$, and the data in $L_d - \ker D$ are distributed, but must be blocked.

5.2 Calculating Blocked Decompositions

We now present an algorithm to find data and computation partitions that may have pipelined communication. Our algorithm first tries to apply the partition algorithm as specified in section 4.3, considering only the parallelism available in the outermost **forall** loops. This will try to find a solution such that every parallelizable loop has parallelism, and there is neither reorganization nor pipelined communication. If such a solution cannot be found, the compiler then tries to exploit **doacross** parallelism.

Recall that the local phase of our compiler transforms each loop nest such that the largest possible fully permutable loop nests are outermost. Also within each fully permutable nest, any **forall** loops are positioned outermost. A loop nest that is fully permutable can also be fully tiled[18, 38]. If the dependence vectors in the fully permutable loop nest are all distance vectors, then the pipelined communication is inexpensive because only the data elements at the block boundaries need to move. Otherwise, the cost of the communication within the loop must be weighed against the cost of reorganization communication between the loops.

algorithm Partition_with_Blocks
  (*IG* : Interference_Graph; /* *IG* = (V_c, V_d, E) */
    /* Computation and data partitions */
   CP_Set : **set of** vector_space;
   DP_Set : **set of** vector_space;
    /* Computation and data localized spaces */
   CL_Set : **set of** vector_space;
   DL_Set : **set of** vector_space)

  /* Try to find solution with no communication */
 Partition(*IG,CP_Set,DP_Set*);

 if no parallelism **then**
    /* Record localized spaces */
  foreach $\ker C_j \in$ *CP_Set* **do** $L_{c_j} := \ker C_j$;
  foreach $\ker D_x \in$ *DP_Set* **do** $L_{d_x} := \ker D_x$;

    /* Find blocked iterations and data */
  foreach $j \in V_c$ **do** $\ker C_j$:= single_blocked_loop_constraint(j);
  multiple_loop_constraint(*IG, DP_Set*);
  Calc_Relation(*IG,CP_Set,DP_Set*);
 end if;
end algorithm;

Figure 4: Algorithm for calculating partitions with blocks.

An overview of the algorithm is shown in Figure 4. Once the algorithm determines that a solution with neither reorganization nor pipelined communication (and with at least one degree of parallelism) cannot be found, it recalculates $\ker C$ and $\ker D$. The partition algorithm in Figure 2 is reapplied – the only change is in the single loop constraint used to initialize the computation partitions. Any dimensions that can be tiled are not considered in the initial computation partitions. Thus the initial computation partition for a loop nest of depth l is again the span of a set of l-dimensional elementary basis vectors. If a loop at nesting level k is sequential *and* cannot be tiled, then e_k is included in the initial computation partition. The multiple array constraints are used as before to initialize the data partition $\ker D$. The iterative partition algorithm is then run to find the data and computation partitions. The final $\ker C$

and $\ker D$ represent the computation and data that must be allocated to the same processor.

We now need to find L_c and L_d to determine the iterations and array elements that are either completely local to a processor or blocked. Note that these vector spaces have already been calculated. When the partition algorithm was called to find a solution with no pipelined communication, the resulting $\ker C$ is exactly the vector space L_c for each loop nest. Similarly, the resulting $\ker D$ is exactly the vector space L_d for each array. Once the partitions have been found, then an orientation and displacement are calculated as discussed in sections 4.4 and 4.5. When the iteration and data spaces are the blocked, the orientations and displacements are found for entire blocks.

In the ADI example above, both loop nests are fully permutable and can be completely tiled. When the compiler discovers that the **forall** parallelism cannot be exploited without communication, it tries to exploit the **doacross** parallelism in these loops. The initial computation partitions are $\ker C_{1,2} = \emptyset$, and the initial data partition is $\ker D_X = \emptyset$. Running the iterative partition algorithm does not change the partitions. Since $L_{c_{1,2}} = \text{span}\{(0,1),(1,0)\}$ and $L_{d_X} = \text{span}\{(0,1),(1,0)\}$, the spaces are completely tiled.

Note that the algorithm yields a solution that allows the entire iteration and data space to be tiled, and does not over-constrain the partitions unnecessarily. This solution can have idle processors, as was shown in Figure 3(b) above. The optimizations our compiler uses to reduce idle processors are described in Section 7.1.

When we exploit **doacross** parallelism, it is possible that different processors will write to the same array location within a loop nest. Thus, there is no single static data decomposition for the array at that loop nest. In these cases, however, the amount of data that must be communicated is small with respect to the amount of computation. A code generator for a shared address space machine does not need to know exactly which processor has the current value of the data. Our code generator for distributed address space machines uses data-flow analysis on individual array accesses to find efficient communication when the data moves within a loop nest[2].

6 Dynamic Decompositions

In this section we solve the problem of finding data and computation decompositions that maximize parallelism when both data reorganization and pipeline communication are allowed. Data reorganizations occur when the decomposition for an array in one loop nest differs from the decomposition of the same array in another loop nest. We find a data decomposition for each array at each loop nest, and a computation decomposition for each loop nest.

6.1 The Communication Graph

To model decompositions that change dynamically, we use a *communication graph* $G = (V, E)$. The nodes in the graph correspond to the loop nests in the program. The edges in the graph represent places in the program where data reorganization communication can occur.

The edges in the graph are calculated using information that is similar to the *reaching decompositions*[14, 33] used in the Fortran D compiler. In Fortran D, the reaching decompositions are defined to be the set of decomposition statements that may reach an array reference that uses the decomposition. In our case, all loop nests may define a decomposition. Thus, the decomposition for an array in one loop nest reaches another loop nest if it is possible for the values of the array in the two loop nests to be the same. This problem can be calculated in a manner similar to the standard reaching definitions data flow problem. The edges in the communication graph are the chains formed by the reaching decompositions, and are not directed. For simplicity of presentation, this discussion assumes each array is both read and written in the loop nests that access the array.

Associated with each edge $e \in E$ is the probability that the decomposition in one loop nest will reach the other loop nest. Each loop node has a weight that is a function of the number of instructions in the loop and an estimate of the number of times the loop executes. Our implementation currently uses profile information to calculate the probabilities for the edge weights as well as the loop execution counts for the loop node weights.

6.2 Problem Formulation

We can now formally state the dynamic decomposition problem. For a given communication graph $G = (V, E)$ we want to find the data decomposition of each array at each loop node, and the corresponding computation decomposition of the loops.

The computation decomposition determines the degree of parallelism in a loop nest. For each loop node, we use the computation decomposition and the loop node weight to estimate the benefit to the execution time as a result of the parallelism in the loop nest. Note that if there is tiling, the parallelism benefit of the loop takes into account the cost of pipeline communication within the loop. Data reorganization can occur if the decomposition for an array in one loop differs from the decomposition of the same array in another loop. Thus the data decompositions, together with the probabilities on the edges, are used to estimate the communication time. The value of the graph is the sum of the parallelism benefits of all the loop nodes minus the total communication cost. The goal is to label the arrays and loops with decompositions such that the overall value of the graph is maximized. For example, consider the following program fragment.

(1) **forall** i_1 := 0 **to** N **do**
    forall i_2 := 0 **to** N **do**
        X[i_1,i_2] := f_1(X[i_1,i_2], Y[i_1,i_2]);
        Y[i_1,i_2] := f_2(X[i_1,i_2], Y[i_1,i_2]);

if (expr) **then**
(2) **forall** i_1 := 0 **to** N **do**
    for i_2 := 0 **to** N **do**
        X[i_1,i_2] := f_3(X[i_1,$g_1(i_2)$]);

else
(3) **forall** i_1 := 0 **to** N **do**
    for i_2 := 0 **to** N **do**
        Y[i_2,i_1] := f_4(Y[$g_2(i_2)$,i_1]);
end if;

(4) **forall** i_1 := 0 **to** N **do**
    forall i_2 := 0 **to** N **do**
        X[i_1,i_2] := f_5(X[i_1,i_2], Y[i_1,i_2]);
        Y[i_1,i_2] := f_6(X[i_1,i_2], Y[i_1,i_2]);

Figure 5(a) shows the communication graph, assuming the expression is true 75% of the time, and that both arrays are of size 10×10. The edges are labeled with the estimated communication time assuming that none of the decompositions match, and all the data must be reorganized between each loop nest. The value on the edge between nodes 1 and 4 is the sum of the communication estimates for arrays X and Y. From loop nest 1, the decomposition

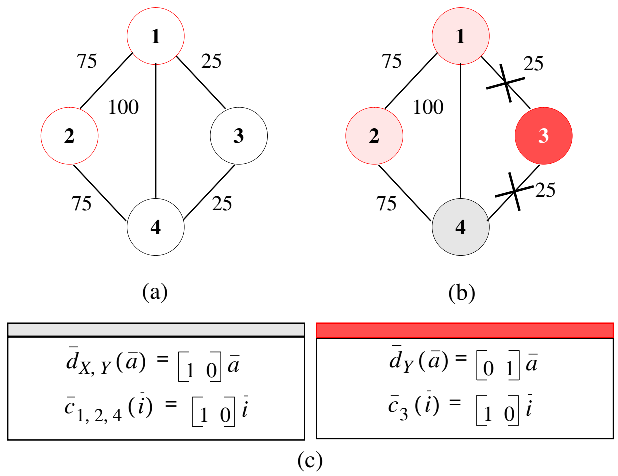

Figure 5: (a) A communication graph. (b) The components resulting from the dynamic decomposition. (c) The final decompositions.

of X has a 25% probability of reaching loop nest 4, and the decomposition of Y has a 75% probablility. Figures 5(b) and (c) illustrate the final decompositions for this example, and are discussed in the next section.

Theorem 6.1 *The dynamic decomposition problem is NP-hard.*

Proof: We prove the dynamic decomposition problem NP-hard by transforming the known NP-hard problem, Colored Multiway Cut[9], into a subproblem of this problem. The Colored Multiway Cut problem is given a graph $G = (V, E)$ with weighted edges, and a partial k-coloring of the vertices, i.e., a subset $V' \subseteq V$ and a function $f : V' \rightarrow 1, 2, \ldots, k$. Can f be extended to a total function such that the total weight of edges that have different colored endpoints is minimized? Consider the subproblem of the dynamic decomposition problem in which the program accesses only a single array X. If parallelized, each loop node has a value that is greater than the sum of the weights of all the edges; otherwise it has a value of 0.

We can reduce an instance of Colored Multiway Cut into an instance of our subproblem in polynomial time. We only give a brief overview of the reduction here. Each node in the original problem becomes a loop nest of depth k in our subproblem's input program, and the edge weights in G become branch probabilities. The array X is k-dimensional, where each dimension represents a color in G. We write the input program such that for each node $v \in V'$ of color i, only the ith loop is parallel in the loop nest representing v. For each node $v \in V - V'$ all loops in the corresponding loop nest are parallel. $\square$

After finding the dynamic decompositions, the edges that have communication correspond to the cutset of edges in the Colored Multiway Cut problem. If the edges are removed, then an array will have the same decomposition across all loop nests in a connected component of the communication graph.

6.3 Dynamic Decomposition Algorithm

Given that the dynamic decomposition problem is NP-hard, our compiler algorithm uses heuristics to find dynamic decompositions. Our dynamic algorithm uses a greedy approach that eliminates the largest amounts of communication first. The algorithm joins the loop nodes that have the greatest edge costs into the same component, thus eliminating the possibility of data reorganization between those two loop nodes. We only consider loop nests that have some degree of parallelism when joining components. Purely sequential loops are treated as being in a component by themselves. An overview of the algorithm is shown in Figure 6. The routine *Single_Level* describes the algorithm for the base case of a single nesting level. The rest of the algorithm deals with the multiple level case and is described in the next section.

At each nesting level, the algorithm operates on a communication graph. The algorithm initializes the components such that each node in the communication graph $G = (V, E)$ is its own component, and then calculates the edge weights. The edge weights are a worst-case approximation of the actual communication cost. The worst case occurs when none of the decompositions match and all the data must be reorganized between each loop nest.

The edges in E are examined in decreasing order of their weights. For each edge $(u, v) \in E$, the algorithm tries to join the current component of u and the current component of v into a single component. An interference graph is created from the loop nodes (and arrays referenced in the loops) in the new, joined component. The partition algorithm from section 5 is called with the interference graph to find the new partitions.

In forming the new component, the algorithm eliminates the data reorganization cost of the edge. However, the union operation may cause some (or all) of the loop nodes to execute sequentially, or it may generate pipeline communication within loop nodes (as a result of tiling). The algorithm finds the value of the graph before and after the new partitions have been calculated. If the value of the graph is greater after the join, then the new component is saved. The algorithm then records the new partitions of all loops and arrays within the new component. Otherwise, there is communication along the edge (u, v), and the new component is discarded.

Consider the communication graph of Figure 5(a). For this example, we assume that the loop node weights are very large. As none of dependences in the code are distance vectors, we assume that tiling is not practical. The edge between nodes 1 and 4 is examined first. The partition algorithm determines that there is at least one degree of parallelism without data reorganization communication between the two loops, so the nodes are joined. Next the algorithm examines either the edge between nodes 1 and 2 or the edge between 2 and 4. In this case the partition algorithm can still find parallelism among the nodes 1, 2 and 4. Next, the algorithm tries to add node 3 into the component. This time the partition algorithm finds that the only way to eliminate reorganization communication is to run all four loops sequentially and the algorithm decides not to add node 3. Thus, nodes 1, 2 and 4 form one component and node 3 is in another component. Figure 5(b) illustrates the resulting components and Figure 5(c) shows the final decompositions within each component.

6.4 Putting It All Together

So far we have only considered the base case of at most one outer sequential loop containing a number of perfectly nested loops. The *Dynamic_Decomposition* routine shown at the bottom of Figure 6 gives an overview of the decomposition algorithm in the general case.

Each nesting level is examined in a bottom-up order. This has the effect of pushing communication into the outermost loops as much as possible. The components are re-initialized at each level so that each loop nest is considered in the context of its sequential outer nest at the current level. The partitions found at each level are used to initialize the partitions for the next level. All references to

```
algorithm Single_Level
    (CG : Communication_Graph;  /* CG = (V, E) */
    /* Computation and data partitions */
    CP_Set : set of vector_space;
    DP_Set : set of vector_space;
        /* Computation and data localized spaces */
    CL_Set : set of vector_space;
    DL_Set : set of vector_space)

  joined_comp, comp1, comp2 : Component;
  IG : Interference_Graph;
  val : integer;

  initialize components;
  foreach (u, v) ∈ E do calculate w(u, v);
  val := value(CG);
  foreach (u, v) ∈ E in decreasing order of weight do
    comp1 := find_component(u);
    comp2 := find_component(v);
    joined_comp := union_components(comp1,comp2);
    IG := create_interference_graph(joined_comp);
    Partition_with_Blocks(IG,CP_Set,DP_Set,CL_Set,DL_Set);
    if value(CG) > val then
      val := value(CG);
      install joined_comp;
    else
       discard joined_comp;
       record data reorganization between u and v;
    end if;
  end foreach;
end algorithm;

algorithm Dynamic_Decomposition
  CG : Communication_Graph;
    /* Computation and data partitions */
  CP_Set : set of vector_space;
  DP_Set : set of vector_space;
    /* Computation and data localized spaces */
  CL_Set : set of vector_space;
  DL_Set : set of vector_space;

  foreach nesting level i in bottom-up order do
    CG :=
      create_comm_graph(i,CP_Set,DP_Set,CL_Set,DL_Set);
    Single_Level(CG, CP_Set,DP_Set,CL_Set,DL_Set);
  end foreach;

  calculate orientations;
  calculate displacements;
end algorithm;
```

Figure 6: Algorithm for calculating dynamic decompositions.

loop indices that are outside the current nesting level are treated as symbolic constants when finding the partition constraints. In this manner, only the constraints for the current level are considered.

In the bipartite interference graph, there is an edge between each array node and each loop node that accesses the array. Each array node in the bipartite interference graph only has a single decomposition. Thus, if the dynamic algorithm discovers that an array's decomposition changes, the node corresponding to that array in the interference graph is split at all subsequent levels. The edges are adjusted so that loops using the reorganized decomposition now point to the proper array node.

Once the components have been formed, the algorithm finds the orientations and displacements. The orientations of the arrays and loops within a component are relative to one another; for example, no additional communication would result from transposing all the decompositions within a component. We use this observation to reduce the amount of communication when finding the orientations across components. Our compiler chooses an orientation for each component that matches as closely as possible to the other components that are connected by edges. Again, the compiler uses a greedy strategy based on the edge weights to decide which components to orient first. The orientations and displacements within a component are found using the algorithms described in Section 4.4 and Section 4.5, respectively.

The dynamic decomposition algorithm shown in Figure 6 is the driver algorithm for finding decompositions in the general case. It finds data and computation decompositions that maximize parallelism and minimize data reorganization and pipelined communication. The partitioning algorithms from the previous two sections are used as subroutines to the dynamic algorithm. In particular, if a static decomposition exists, then the dynamic algorithm will be able to successfully join all the loop nodes into a single component. In general, the algorithm reports a data decomposition for each array at each loop nest, and a computation decomposition for each loop nest.

7 Optimizations

There are several ways to improve upon the program decompositions found in the previous section. This section briefly summarizes how to minimize the number of idle processors and how to find and minimize replication of read-only data.

7.1 Idle Processors

When a loop nest accesses only a subsection of an array, the number of virtual processor dimensions may be larger than the nesting depth of a loop nest. As a result, only a fraction of the processors will be busy during the execution of the loop nest. To avoid idle processors, we use the computation decomposition to find those processor dimensions that have parallelism for all loops. The equation for the number of virtual processor dimensions n is modified so that n is limited to the minimum distributed iteration space:

$$n' = \min\Big(\max_{x \in \text{Arrays}} (\dim(S_x) - \dim(\ker D_x)), \\ \min_{j \in \text{Loops}} (l - \dim(\ker C_j))\Big)$$

Here $S_x = \sum_{\forall j \in \text{loops\_using}(x)} \text{range}(F_{xj})$ is the array space accessed.

We then reduce the number of virtual processor dimensions by projecting the n-dimensional virtual processor space onto an n'-dimensional processor space. In choosing the dimensions in the virtual processor space to project onto, n' vectors are selected $(\vec{t}_1, \vec{t}_2, \ldots, \vec{t}_{n'})$ such that $\forall i, \vec{t}_i \notin \ker C^T$ for all computation decomposition matrices C. This means that there are no projections onto a processor dimension that is idle during the execution of any loop nest.

7.2 Replication

Replication of read-only data is a common technique used to improve the performance of parallel machines. Our algorithms find the amount of read-only data replication needed to maintain the degree of parallelism inherent in the read-write data without introducing additional communication.

We consider two types of replication: *constant replication* and *dimension replication*. Constant replication occurs when there are multiple data decompositions for an array. Dimension replication means that all processors along a given dimension have a copy of the same data. The increase in the space requirements to accommodate constant replication is a linear function of the array size, whereas dimension replication can cause the space needed to grow by the number of processors. Thus we focus on dimension replication.

To allow the necessary replication, we first run the decomposition algorithm *without* taking into account the read data in the program. From the resulting computation partition, we can then find the data partitions of the read-only arrays using Eqn. 5. We must now find the processor dimensions which contain replicated data.

Dimension replication is modeled using a reduced processor space, which is then expanded into the full n-dimensional processor space. All processors in the expanded dimensions have copies of the data that are on the corresponding processor in the reduced space. Let x be a replicated m-dimensional array, with data partition $\ker D_x$. The dimensionality of the reduced processor space, n_r, is $n_r = \dim(S_x) - \dim(\ker D_x)$, where $S_x = \sum_{\forall j \in \text{loops\_using}(x)} \text{range}(F_{xj})$ is the array space accessed. The *degree* of replication for an array (the number of processor dimensions along which the data is copied) is $n - n_r$. The data decomposition matrix for array x is an $n_r \times m$ decomposition matrix D_x, which maps array elements onto the reduced processor space.

Let C_j be the computation decomposition matrix for a loop nest j that accesses array x. C_j maps iterations onto the full processor space. To relate the full processor space to the reduced processor space, we use an $n_r \times n$ matrix R_{xj}. Eqn 3 from section 4.1 is modified to express the relationship between computation and data with replication:

$$D_x F_{xj}(\vec{\imath}) = R_{xj} C_j(\vec{\imath}) \qquad (7)$$

The matrix R_{xj} maps C_j onto the reduced space, and the nullspace of R_{xj}, $\ker R_{xj}$, corresponds to dimensions along which there is replication.

The algorithm uses the data partition $\ker D_x$ to find $\ker R_{xj}$. In section 4.4, we used Eqn. 3 to find the data and computation decomposition matrices. Similarly, we use Eqn. 7 to find the decomposition matrices with replication.

The computation and data decompositions are initially derived without any consideration for the amount of replication needed. As a result, the amount of replication called for could be much greater than is practical on the target machine. Thus, we also use the techniques in section 7.1 to limit the degree of replication by projecting the virtual processor space onto a smaller processor space.

8 Experimental Results

We have implemented the algorithms described in this paper in the SUIF compiler at Stanford. The experiments described in this section were performed on the Stanford DASH shared-memory multiprocessor[26]. Since we do not have a code generator for DASH at this point, we implemented by hand parallel SPMD programs with the decompositions generated by our compiler. All programs were compiled with the SGI *f77* compiler at the *-O2* optimization level.

The DASH multiprocessor is made up of a number of physically distributed clusters. Each cluster is based on Silicon Graphics POWER Station 4D/340, consisting of 4 MIPS R3000/R3010 processors. A directory-based protocol is used to maintain cache coherence across clusters. It takes a processor 1 cycle to retrieve data from its cache, 29 cycles from its local memory and 100-130 cycles from a remote memory. The DASH operating system allocates memory to clusters at the page level; if a page is not assigned to a specific cluster then it is allocated to the first cluster that touches the page.

We compare the decomposition our algorithm finds with the decomposition the SGI Power Fortran Accelerator (version 4.0.5) parallelizing compiler used. We also compare our results with other possible decompositions. We ran our programs on an 8-cluster DASH multiprocessor, with 28MB of main memory per cluster.

We looked at the heat conduction phase of the application SIMPLE, a two-dimensional Lagrangian hydrodynamics code from Lawrence Livermore National Lab. The heat conduction routine *conduct* is 165 lines long and has about 20 loop nests. Within this routine is a set of loops that performs an ADI integration where the parallelism is first across the rows of the arrays and then across the columns of the arrays. In all cases, we used a blocked distribution scheme.

Figure 7 shows the speedups (over the best sequential version) of four different decompositions of this routine, for a problem size of 1K $\times$ 1K using double precision.

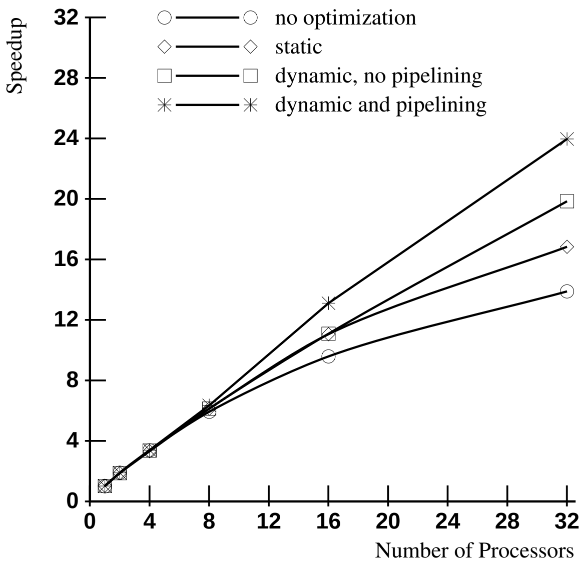

Figure 7: Speedup over sequential execution time for *conduct*. The problem size is 1K $\times$ 1K, double precision.

The total amount of data used by this routine is on the order of

128MB. When the amount of memory needed by a cluster exceeds the memory available on that cluster, the DASH operating system allocates the memory on the next available cluster. Thus when executing on four or fewer clusters, the data used in the application may actually be allocated to another cluster.

The first curve labeled *no optimization* shows the results of the SGI Power Fortran compiler. We allowed the DASH operating system to allocate the pages to the first cluster that accessed the data. Since Fortran arrays are allocated column major, this resulted in blocks of columns being allocated to the clusters. When the only available parallelism is by row, the processors perform remote reads and writes. The second curve labeled *static* shows the performance if a single data decomposition is used for each array. In this case blocks of rows were made contiguous in the shared address space. This represents the best possible static decomposition if only **forall** parallelism is exploited. The third curve labeled *dynamic, no pipelining* reallocates the data when the dimension of the parallelism changes. However, in this case the program incurs the cost of reorganization communication when the data is reallocated. This curve represents the best possible overall decomposition with only **forall** parallelism. The fourth curve labeled *dynamic and pipelining* shows the results of allocating blocks of rows contiguously and using explicit synchronization between processors when the parallelism is by column. In this version the processors only synchronize between blocks of columns (we used a block size of 4). This is the decomposition our compiler finds when considering both pipeline and reorganization communication.

9 Related Work

A number of researchers have addressed problems that are related to the decomposition problem. Sarkar and Gao have developed an algorithm that uses collective loop transformations to perform array contraction[32]. They use loop interchange and reversal transformations to orient the computation. Ju and Dietz use a search-based algorithm to find data layout and loop restructuring combinations that reduce cache coherence overhead on shared memory machines[19]. Hwang and Hu describe a method for finding the computation mapping of two systolic array stages that share a single array[16]. Their algorithm works by first calculating the projection vector, which is similar to what we call the partition, of the computation mapping.

Many projects have examined the problem of finding array alignments (what we call data orientations and displacements) for data parallel programs[8, 11, 21, 30, 35]. These approaches focus on element-wise array operations, and try to eliminate the communication between consecutive loops.

Li and Chen prove the problem of finding optimal orientations NP-complete[28], and have developed a heuristic solution which is used to implement their functional language Crystal on message-passing machines[27]. In contrast to these approaches, our model supports loop nests containing both parallel and sequential loops and general affine array index functions. These approaches all optimize for a fixed degree of parallelism, whereas we make explicit decisions about which loops are run in parallel.

Several researchers have developed data decomposition algorithms based on searching through a fixed set of possible decompositions. Gupta and Banerjee have developed an algorithm for automatically finding a static data decomposition[12]. Their approach is based on an exhaustive search through various possible decompositions using a system of cost estimates. Carle et. al. have developed an interactive tool, as part of the Fortran D project, that finds data decompositions within and across phases of a procedure[6]. Data can be remapped dynamically between phases. Their approach uses a static performance estimator[4] to select the best decompositions among a fixed set of choices. In comparison, our algorithm avoids expensive searches by systematically calculating the decompositions. As a result of our mathematical model, we are able to derive decompositions that take into account pipeline communication within loop nests and data reorganization communication across loop nests.

10 Summary and Conclusions

The decomposition problem is very complex, as there are many inter-related issues that must be addressed. This paper addresses the full problem of automatically calculating data and computation decompositions for programs in a systematic way.

Our algorithms are based on the mathematical model of decompositions as affine functions. This framework is general enough to handle a broad class of array access patterns. Using the affine model we structure decompositions into three components: partition, orientation and displacement. Since equivalent decompositions have the same partition, we solve for the partition first and can therefore evaluate many possible decomposition designs simultaneously.

To maximize parallelism, our algorithm exploits **forall** parallelism, as well as **doacross** parallelism using tiling. To minimize communication, the algorithm tries to find a static decomposition that exploits the maximum degree of parallelism available in the program such that there is no reorganization nor pipeline communication. The algorithm will trade off extra degrees of parallelism to eliminate communication. If communication is needed, the algorithm will try to reduce expensive reorganization communication to inexpensive pipelined communication by tiling. Finally, any necessary data reorganization communication is inserted into the least frequently executed parts of the program.

References

[1] 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.

[2] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, June 1993.

[3] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the Third ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[4] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 213–222, April 1991.

[5] D. Callahan. *A Global Approach to Detection of Parallelism*. PhD thesis, Rice University, April 1987. Published as COMP-TR-87-50.

[6] A. Carle, K. Kennedy, U. Kremer, and J. Mellor-Crummey. Automatic data layout for distributed-memory machines in the D programming environment. Technical Report CRPC-TR93-298, Rice University, February 1993.

[7] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[8] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Automatic array alignment in data-parallel programs. In *Proceedings, 20th Annual ACM Symposium on Principles of Programming Languages*, pages 16–28, January 1993.

[9] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiway cuts. In *Proceedings of the 24th ACM Symposium on the Theory of Computing*, pages 241–251, May 1992.

[10] G. R. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Programming Languages and Compilers for Parallel Computing*, pages 171–181, August 1992.

[11] J. R. Gilbert and R. Schreiber. Optimal expression evaluation for data parallel architectures. *Journal of Parallel and Distributed Computing*, 13(1):58–64, September 1991.

[12] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[13] High Performance Fortran Forum. *High Performance Fortran Language Specification*, November 1992. Version 0.4.

[14] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[15] C. H. Huang and P. Sadayappan. Communication-free hyperplane positioning of nested loops. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 186–200. Springer-Verlag, Berlin, Germany, 1992.

[16] Y.-T. Hwang and Y. H. Hu. On systolic mapping of multistage algorithms. In *Proceedings of the IEEE International Conference on Application Specific Array Processors*, pages 47–61, August 1992.

[17] Intel Corporation, Santa Clara, CA. *iPSC/2 and iPSC/860 User's Guide*, June 1990.

[18] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 319–329, January 1988.

[19] Y. Ju and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 344–358. Springer-Verlag, Berlin, Germany, 1992.

[20] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 323–334, July 1992.

[21] K. Knobe, J. D. Lukas, and G. L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[22] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. In *Proceedings of the Second ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 177–186, March 1990.

[23] D. Kulkarni, K. G. Kumar, A. Basu, and A. Paulraj. Loop partitioning for distributed memory multiprocessors as unimodular transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, pages 206–215, June 1991.

[24] K. G. Kumar, D. Kulkarni, and A. Basu. Deriving good transformations for mapping nested loops on hierarchical parallel machines in polynomial time. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 82–91, July 1992.

[25] M. S. Lam and M. E. Wolf. Compilation techniques to achieve parallelism and locality. In *Proceedings of the DARPA Software Technology Conference*, pages 150–158, April 1992.

[26] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[27] J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Supercomputing 1990*, pages 865–876. IEEE, May 1990.

[28] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Proceedings of Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, pages 424–432. IEEE, October 1990.

[29] D. E. Maydan. *Accurate Analysis of Array References*. PhD thesis, Stanford University, September 1992. Published as CSL-TR-92-547.

[30] J. F. Prins. A framework for efficient execution of array-based languages on SIMD computers. In *Proceedings of Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, pages 462–470. IEEE, October 1990.

[31] A. Rogers and K. Pingali. Compiling for locality. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 142–146, June 1990.

[32] V. Sarkar and G. R. Gao. Optimization of array accesses by collective loop transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, pages 194–204, June 1991.

[33] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993. Published as Rice COMP TR93-199.

[34] P.-S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, May 1989. Published as CMU-CS-89-148.

[35] S. Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, May 1991. Published as CMU-CS-91-121.

[36] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, August 1992. Published as CSL-TR-92-538.

[37] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[38] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *Transactions on Parallel and Distributed Systems*, 2(4):452–470, October 1991.

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

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