Automatic Data Alignment and Distribution for Loosely Synchronous Problems in an Interactive Programming Environment

Ulrich Kremer
Ken Kennedy

CRPC-TR91205-S
April 1991

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

Revised: March, 1992.

Automatic Data Alignment and Distribution for Loosely Synchronous Problems in an Interactive Programming Environment

Ken Kennedy   Ulrich Kremer
ken@rice.edu   *kremer@rice.edu*

Department of Computer Science
Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, Texas 77251

Abstract

An approach to distributed memory parallel programming that has recently become popular is one where the programmer explicitly specifies the data layout using language extensions, and a compiler generates all the communication. While this frees the programmer from the tedium of thinking about message-passing, no assistance is provided in determining the data layout scheme that gives a satisfactory performance on the target machine. We wish to provide automatic data alignment and distribution techniques for a large class of scientific computations, known in the literature as *loosely synchronous problems*.

We propose an interactive software tool that allows the user to select regions of the sequential input program, then responds with a data decomposition scheme and diagnostic information for the selected region. The proposed tool allows the user to obtain insights into the characteristics of the program executing on a distributed memory machine and the behavior of the underlying compilation system without actually compiling and executing the program on the machine.

An empirical study of actual application programs will show whether automatic techniques are able to generate data decomposition schemes that are close to optimal. If automatic techniques will fail to do so, we want to answer the questions (1) how user interaction can help to overcome the deficiencies of automatic techniques, and (2) whether, in particular, there is a *data-parallel programming style* that allows automatic detection of efficient data alignment and distribution schemes.

1 Introduction

Although distributed memory parallel computers are among the most cost-effective machines available, most scientists find them difficult to program. The reason is that traditional programming languages support single name spaces and, as a result, most programmers feel more comfortable working with a shared memory programming model. To this end, a number of researchers [CK88, CCL88, KMV90, PSvG91, RA90, RP89, RSW88, ZBG88, KZBG88, Ger89] have proposed using a traditional sequential or parallel shared-memory language extended with annotations specifying how the data is to be mapped onto the distributed memory machine. This approach is inspired by the observation that the most demanding intellectual step in rewriting programs for distributed memory is the data layout — the rest is straightforward but tedious and error prone work. The

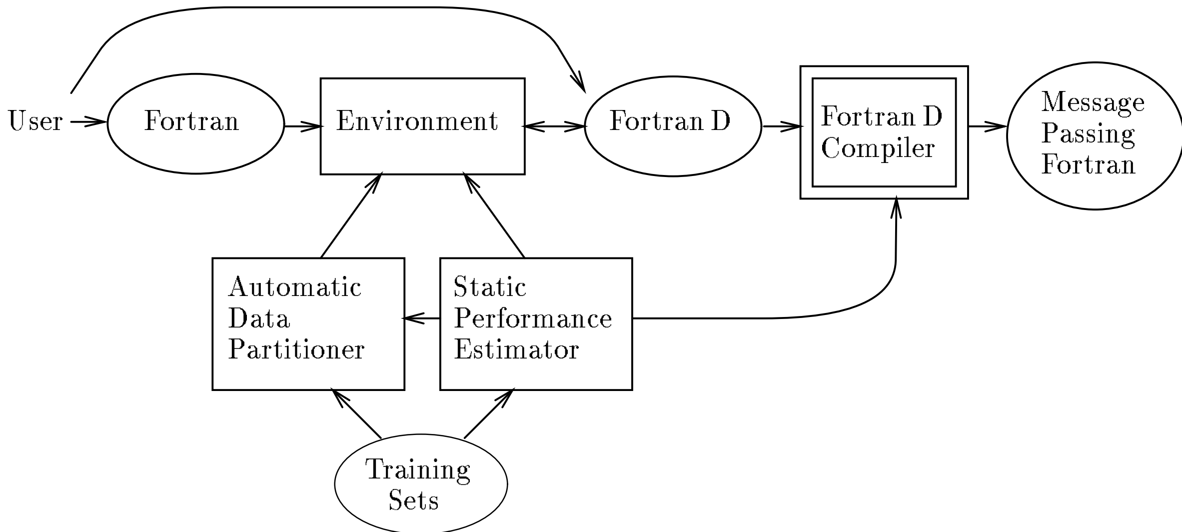

Figure 1: Fortran D Parallel Programming Environment

Fortran D language and its compiler [FHK+90, HKT92b] follow this approach. Given a Fortran D program the compiler mechanically generates the node program for a given distributed-memory target machine.

A problem with this approach is that it does not support the user in the decision process about a good data layout. We want to propose the development of automatic techniques for data alignment and distribution for regular *loosely synchronous* problems that deal with realignment/redistribution, data replication, procedure calls, and control flow. Loosely synchronous problems represent a large class of scientific computations [FJL+88]. They can be characterized by computation intensive regions that have substantial parallelism, with a synchronization point between the regions. We restrict our proposed system to regular loosely synchronous problems with arrays as their major data structures. In addition, we will allow regular problems that give rise to wavefront style computations [Lam74].We do not handle representations of data objects as they occur, for instance, in irregular problems like sparse matrix and unstructured mesh computations [SCMB90, WSHB91, KMV90].

We will investigate the feasibility of these new automatic techniques in the context of an interactive system. The proposed automatic data decomposition tool is part of the Fortran D programming environment as shown in Figure 1. The automatic data partitioner may be applied to an entire program or on specific program fragments. When invoked on an entire program, it automatically selects data decompositions without further user interaction.

The feasibility of the automatic system is determined by the 'quality' of the generated data decomposition schemes compared to schemes that are considered optimal. This quality measure has to be defined *a priori* to our investigation in order to guarantee an unbiased discussion of results gained by an empirical study of our techniques applied to *real* programs. If it will turn out that automatic data decomposition is not feasible for some applications we want to discuss why automatic techniques failed and how user interaction can help to overcome its deficiencies. In particular, we want to investigate whether there is a *data-parallel programming style* for sequential programs that facilitates automatic data alignment and distribution. The existence of a usable programming style in the context of vectorization has been observed by some researchers [CKK89, Wol89] and is partially responsible for the success of automatic vectorization. We believe that for

regular loosely synchronous problems written in a data-parallel programming style, the automatic data partitioner can determine an efficient decomposition scheme in most cases. However, we do not believe that fully automatic techniques will be successful for 'dusty deck' programs.

After a short introduction to Fortran D, we motivate the need for automatic data decomposition. A discussion of our approach to to automatic data alignment and distribution follows. The rest of the paper gives a description of the success/failure criterion for the proposed automatic techniques. An overview of related work together with our research goals and contributions conclude the paper.

2 Fortran D

The data decomposition problem can be approached by considering the two levels of parallelism in data-parallel applications. First, there is the question of how arrays should be *aligned* with respect to one another, both within and across array dimensions. We call this the *problem mapping* induced by the structure of the underlying computation. It represents the minimal requirements for reducing data movement for the program, and is largely independent of any machine considerations. The alignment of arrays in the program depends on the natural fine-grain parallelism defined by individual members of data arrays.

Second, there is the question of how arrays should be *distributed* onto the actual parallel machine. We call this the *machine mapping* caused by translating the problem onto the finite resources of the machine. It is affected by the topology, communication mechanisms, size of local memory, and number of processors in the underlying machine. The distribution of arrays in the program depends on the coarse-grain parallelism defined by the physical parallel machine.

Fortran D is a version of Fortran that provides data decomposition specifications for these two levels of parallelism using `DECOMPOSITION`, `ALIGN`, and `DISTRIBUTE` statements. A decomposition is an abstract problem or index domain; it does not require any storage. Each element of a decomposition represents a unit of computation. The `DECOMPOSITION` statement declares the name, dimensionality, and size of a decomposition for later use.

The `ALIGN` statement is used to map arrays onto decompositions. Arrays mapped to the same decomposition are automatically aligned with each other. Alignment can take place either within or across dimensions. The alignment of arrays to decompositions is specified by placeholders in the subscript expressions of both the array and decomposition. In the example below,

```
REAL X(N,N)
DECOMPOSITION A(N,N)
ALIGN X(I,J) with A(J-2,I+3)
```

A is declared to be a two dimensional decomposition of size $N \times N$. Array X is then aligned with respect to A with the dimensions permuted and offsets within each dimension.

After arrays have been aligned with a decomposition, the `DISTRIBUTE` statement maps the decomposition to the finite resources of the physical machine. Distributions are specified by assigning an independent *attribute* to each dimension of a decomposition. Predefined attributes are `BLOCK`, `CYCLIC`, and `BLOCK_CYCLIC`. The symbol ":" marks dimensions that are not distributed. Choosing the distribution for a decomposition maps all arrays aligned with the decomposition to the machine. In the following example,

```
DECOMPOSITION A(N,N)
DISTRIBUTE A(:, BLOCK)
DISTRIBUTE A(CYCLIC,:)
```

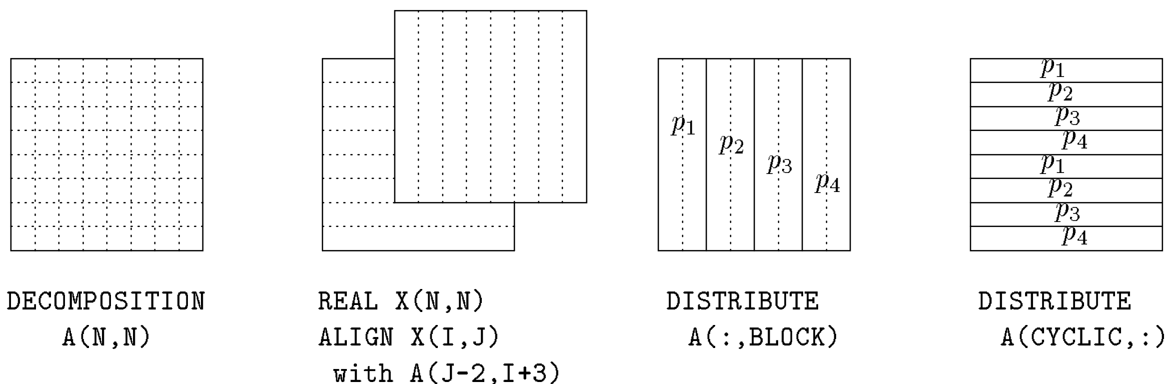

Figure 2: Fortran D Data Decomposition Specifications

distributing decomposition A by `(:,BLOCK)` results in a column partition of arrays aligned with A. Distributing A by `(CYCLIC,:)` partitions the rows of A in a round-robin fashion among processors. These sample data alignment and distributions are shown in Figure 2.

We should note that our goal in designing Fortran D is not to support the most general data decompositions possible. Instead, our intent is to provide decompositions that are both powerful enough to express data parallelism in scientific programs, and simple enough to permit the compiler to produce efficient programs. Fortran D is a language with semantics very similar to sequential Fortran. As a result, it should be quite usable by computational scientists. In addition, we believe that our two-phase strategy for specifying data decomposition is natural and conducive to writing modular and portable code. Fortran D bears similarities to both CM Fortran [TMC89] and Kali [KM91]. The complete language is described in detail elsewhere [FHK+90].

3 Why is Finding a Good Data Layout Hard?

The choice of a good data decomposition scheme depends on many factors. All these factors make it extremely difficult for a human to predict the behavior of a given data decomposition scheme without having to compile and run the program on the specific target system. The effect of a given data decomposition scheme and program structure on the efficiency of the *compiler-generated* code running on a given target machine depends on:

1. *compiler characteristics* such as the communication analysis algorithm and optimizing transformations used, and the set of communication primitives and routines that can be generated. For instance, fast collective communication routines as the Crystal router package developed at Caltech [FF88], are crucial for determining the profitability of realignment and redistribution. In addition, if the compiler generates a node program, the characteristics of the target machine node compiler have to be considered.

2. *machine characteristics*, such as communication and computation costs, and machine topology. For example, the specification of a three dimensional grid partition might be efficient on a machine with a three dimensional topology assuming the predominance of nearest neighbor communication. On a machine with a one dimensional ring topology the same specification leads to less efficient communication because messages must be sent to distant nodes on the ring, creating the potential for collisions.

```
1       DOUBLE PRECISION v(N,N), a, b
2       DECOMPOSITION d(N,N)
3       ALIGN v(I,J) WITH d(I,J)
4       DISTRIBUTE d(BLOCK,BLOCK)
5       DO k = 1, M
6           // Compute the red points
7           DO j = 1, N, 2
8               DO i = 1, N, 2
9                   v(i,j) = a*(v(i,j−1) + v(i−1,j) + v(i,j+1) + v(i+1,j)) + b * v(i,j)
11              ENDDO
12          ENDDO
13          DO j = 2, N, 2
14              DO i = 2, N, 2
15                  v(i,j) = a*(v(i,j−1) + v(i−1,j) + v(i,j+1) + v(i+1,j)) + b* v(i,j)
16              ENDDO
17          ENDDO
18          // Compute the black points
19          DO j = 1, N, 2
20              DO i = 2, N, 2
21                  v(i,j) = a*(v(i,j−1) + v(i−1,j) + v(i,j+1) + v(i+1,j)) + b* v(i,j)
22              ENDDO
23          ENDDO
24          DO j = 2, N, 2
24              DO i = 1, N, 2
26                  v(i,j) = a*(v(i,j−1) + v(i−1,j) + v(i,j+1) + v(i+1,j)) + b* v(i,j)
27              ENDDO
28          ENDDO
29      ENDDO
```

Figure 3: Fortran D code with `BLOCK` distribution, `DOUBLE PRECISION`

3. *problem characteristics*, such as actual problem size and number of processors to be used. The ratio between local computation and necessary communication can determine the efficiency of a data decomposition scheme. For some programs, problem characteristics are not needed to generate good data decomposition schemes. For such programs recompilation for different problem sizes and number of processors used can be avoided.

In the reminder of this section, we will examine the relationship between the listed factors using the characteristics of the Fortran D compiler [HKT92b, HKT91, HHKT91]. Figure 3 shows a Fortran D code for pointwise red-black relaxation using a block-partitioning scheme for the $N \times N$, double precision array `v`. Substituting line 4 by `DISTRIBUTE` d(:,`BLOCK`) specifies a column partitioning scheme for array `v`.

To execute a program on a distributed memory machine the program's data and code have to be mapped into the local memories of the machine's processors. The Fortran D compiler generates a node program according to the *single program, multiple data* (SPMD) programming model [Kar87], exploiting the data parallelism inherent in the program, as opposed to its functional parallelism.

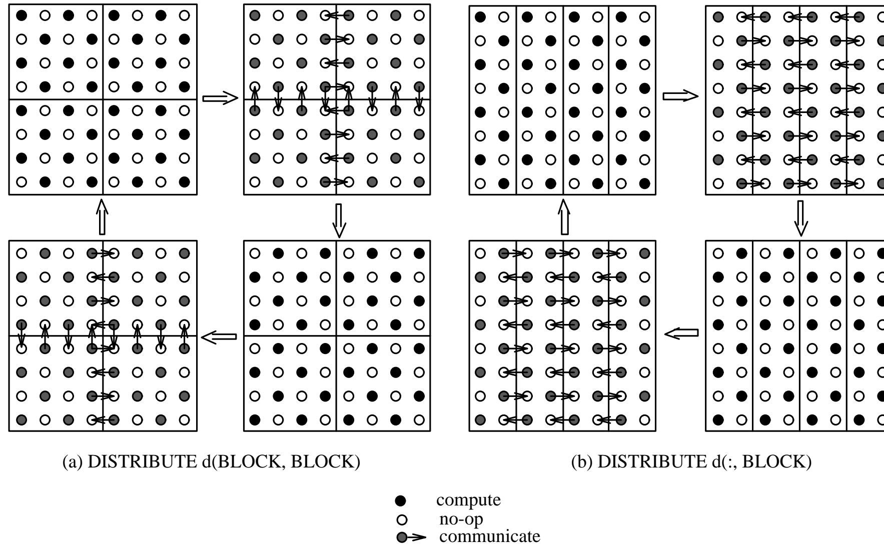

Figure 4: Computation and Communication cycles for block and column partitioning

Each processor executes only those program statement instances that define a value of a data item that has been mapped onto that processor by the decomposition, alignment and distribution specifications. Data items that are mapped onto a processor are said to be *owned* by that processor. Execution of such a statement may require non-local data, i.e., data that is owned by another processor. Such non-local data items must be obtained by communication.

For the given example we assume that the Fortran D compiler performs the following communication optimizations. *Message vectorization* uses the results of data dependence analysis [AK87, KKP+81] to combine element messages into vectors. The level of the deepest loop-carried true dependence or loop enclosing a loop-independent true dependence determines the outermost loop where element messages resulting from the same array reference may be legally combined [BFKK90, Ger90]. *Message coalescing* ensures that each data value is sent to a processor only once. A detailed description of different communication optimizations can be found in [HKT92a]. This paper also discusses an alternative approach to generating messages for the red-black example, called *vector message pipelining*.

Using message vectorization, the Fortran D compiler generates communication statements before the first loop nest at line 7 and after the second loop nest between line 17 and line 19. Our hypothetical, simplified version of the Fortran D compiler inserts code to communicate all boundary points although only black or red points need to be communicated before the first and third loop, respectively. Figure 4 illustrates the resulting compute-communicate sequence for the block-partitioning and column-partitioning schemes.

We assume in our example that the compiler generates an EXPRESS[1] node programs [EXP89] where communication is performed by calls to vector-send and vector-receive communication routines `KXVWRI` and `KXVREA`, respectively. Execution of `KXVWRI` is non-blocking in the sense that the sending processor does not wait until the message is received. Execution of `KXVREA` is blocking, i.e. the processor has to wait until it receives the message. Alternatively, the compiler could have chosen to insert calls to the communication routines `KXVCHA` that implement communication using

[1]EXPRESS a copyright of ParaSoft Corporation.

a handshaking protocol.

Figure 5 and Figure 6 show the actual execution times of one iteration of the outermost timestep loop (line 5 in Figure 4) for increasing sizes of array `v` using column-partitioning and block-partitioning schemes. Figure 5 shows the execution times on the Ncube-1 for 16 and 64 processors, where `v` is a single precision floating point array. Figure 6 contains the results on 16 processors on the iPSC/860 for a single precision and double precision array `v`. The figures show that the tradeoffs between a column-partitioning and block-partitioning schemes for a *compiler-generated* node program. The tradeoffs depend on the actual problem size, i.e. the the element size and overall size of the array `v`, the specific target machine, and the actual number of processors used on the target machine.

What we need is a programming environment that helps the user to understand the effect of a given data decomposition and program structure on the efficiency of the *compiler-generated* code at the *Fortran D language level.* The Fortran D programming system, shown in Figure 1, provides such an environment. The main components of the environment are a static performance estimator and an automatic data partitioner. In this paper we will mainly discuss the automatic data partitioner.

4 Automatic Data Decomposition

The key ideas and assumptions behind automatic data decomposition are

1. reasonably simple static models can be used to estimate the performance of a compiler-generated node program with explicit communication, and

2. the behavior of the Fortran D compiler on a specified program segment, such as a loop or the entire program, can be anticipated efficiently, i.e. without actually compiling the whole program, and

3. if we restrict ourselves to fairly simple decomposition schemes, then for a program segment there are only a small number of such decompositions suitable for each array and hence these decompositions can be exhaustively examined by an automatic system, and

4. the automatic data decomposition for the entire program can be done by successively decomposing the data for smaller program segments, and realignment/redistribution when necessary between the segments.

We believe that these assumptions are true for loosely synchronous problems written in a data parallel programming style. In the following we will discuss some of the issues involved in the above key ideas and assumptions.

4.1 Static Performance Estimation

It is clearly impractical to use dynamic performance information to choose between data decompositions in our programming environment. Instead, a *static* performance estimator is needed that can accurately predict the performance of a Fortran D program on the target machine.

The performance estimator is not based on a general theoretical model of distributed-memory computers. Instead, it employs the notion of a *training set* of kernel routines that measures the cost of various computation and communication patterns on the target machine. The results of executing the training set on a parallel machine are summarized and used to train the performance estimator for that machine. By utilizing training sets, the performance estimator achieves both

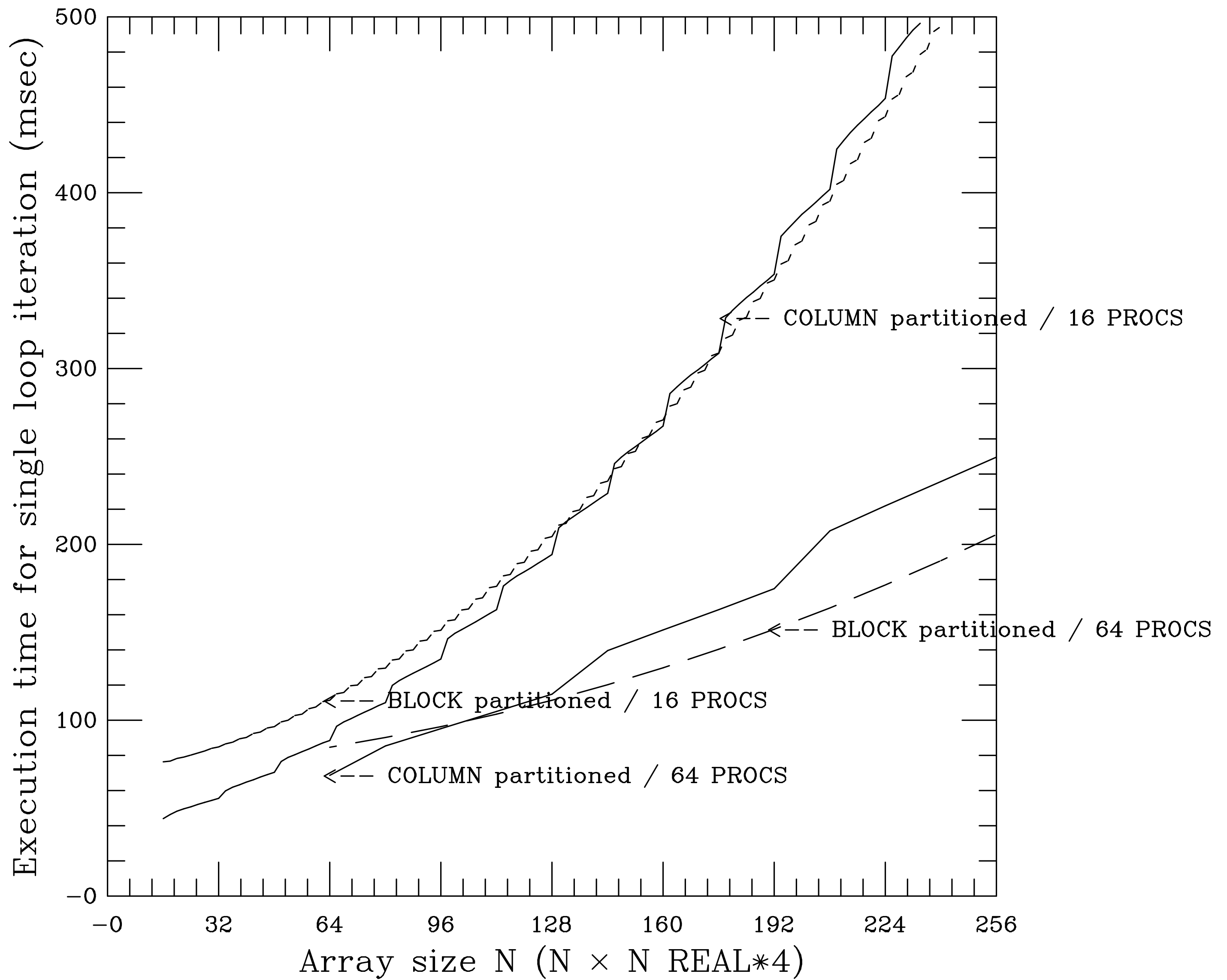

Figure 5: Measured times on Ncube-1: FLOAT operations on 16 and 64 processors

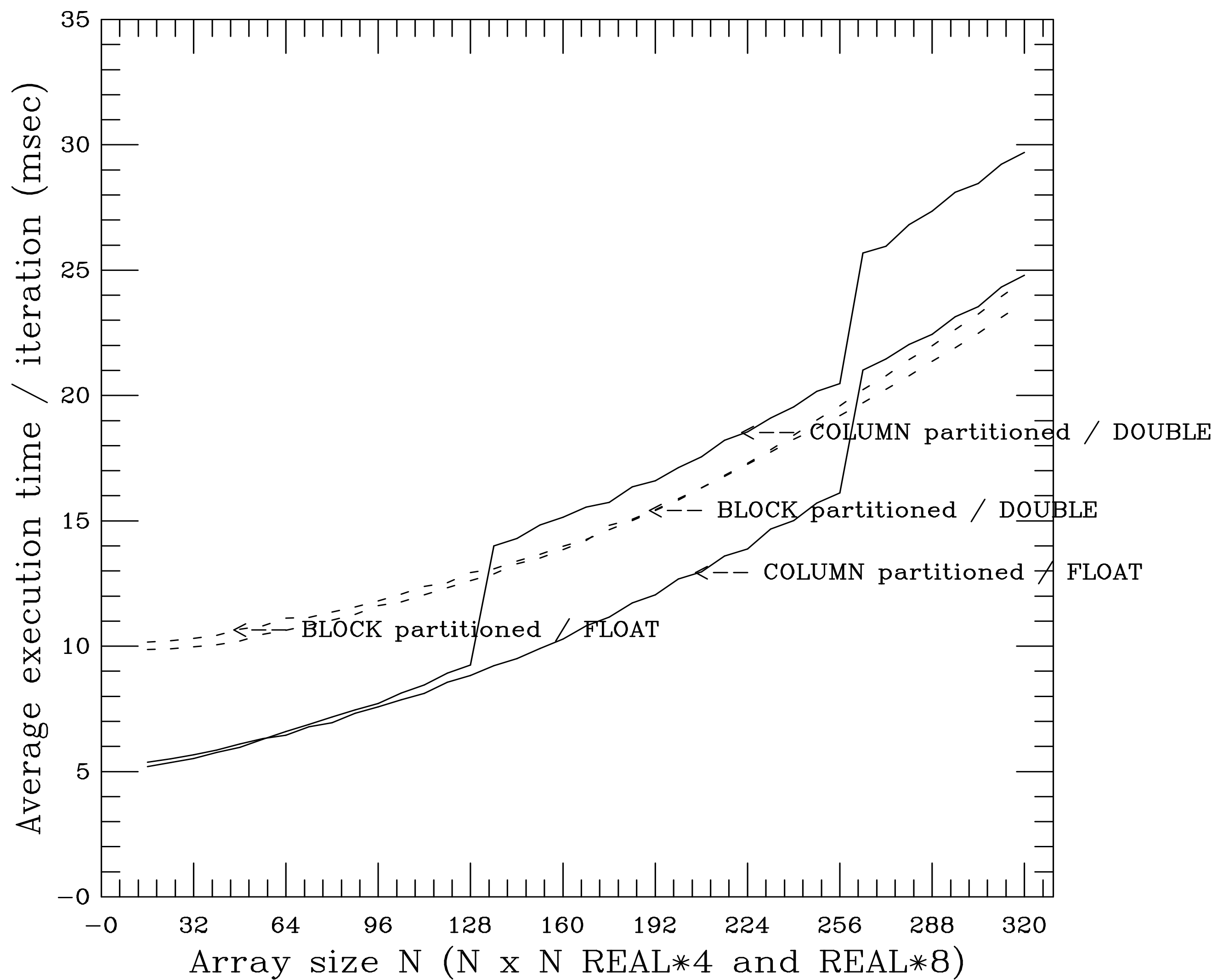

Figure 6: Measured times on iPSC/860: DOUBLE PRECISION and FLOAT operations on 16 processors

accuracy and portability across different machine architectures. The resulting information may also be used by the Fortran D compiler to guide communication optimizations.

The static performance estimator is divided into two parts, a machine module and a compiler module. The *compiler module* predicts the performance at the Fortran D language level while the *machine module* estimates the performance at a level where the decomposition scheme is already 'hardcoded' into the program, i.e. at the node program level containing explicit communications.

4.1.1 Machine Module

The *machine module* predicts the performance of a node programs with explicit communications. It uses a *machine-level* training set written in message-passing Fortran. The training set contains individual computation and communication patterns that are timed on the target machine for different numbers of processors and data sizes. To estimate the performance of a node program, the machine module can simply look up results for each computation and communication pattern encountered.

Note that even though it is desirable, to assist automatic data decomposition the static performance estimator does not need to predict the *absolute* performance of a given data decomposition. Instead, the it only needs to accurately predict the performance *relative* to other data decompositions. In many cases the accurate prediction of the *crossover point* at which one data decomposition scheme is preferable over another will be sufficient.

A prototype of the machine module has been implemented for a common class of *loosely synchronous* scientific problems [FJL+88]. It predicts the performance of a node program using EXPRESS communication routines for different numbers of processors and data sizes [EXP89]. The prototype performance estimator has proved quite precise, especially in predicting the relative performances of different data decompositions [BFKK91]. Figure 7 and Figure 8 show the estimated execution times for our red-black example using the training set approach. The crossover points and execution times induced by the different decomposition schemes are predicted with high accuracy.

4.1.2 Compiler Module

The *compiler module* forms the second part of the static performance estimator. It assists the user in selecting data decompositions by statically predicting the performance of a program for a set of data decompositions. The compiler module employs a *compiler-level* training set written in Fortran D that consists of program kernels such as stencil computations and matrix multiplication. The training set is converted into message-passing Fortran using the Fortran D compiler and executed on the target machine for different data decompositions, numbers of processors, and array sizes. Estimating the performance of a Fortran D program then requires matching computations in the program with kernels from the training set.

The compiler-level training set also provides a natural way to respond to changes in the Fortran D compiler as well as the machine. We simply recompile the training set with the new compiler and execute the resulting programs to reinitialize the compiler module for the performance estimator.

Since it is not possible to incorporate all possible computation patterns in the compiler-level training set, the performance estimator will encounter code fragments that cannot be matched with existing kernels. To estimate the performance of these codes, the compiler module must rely on the machine-level training set. We plan to incorporate elements of the Fortran D compiler in the performance estimator so that it can mimic the compilation process. The compiler module can

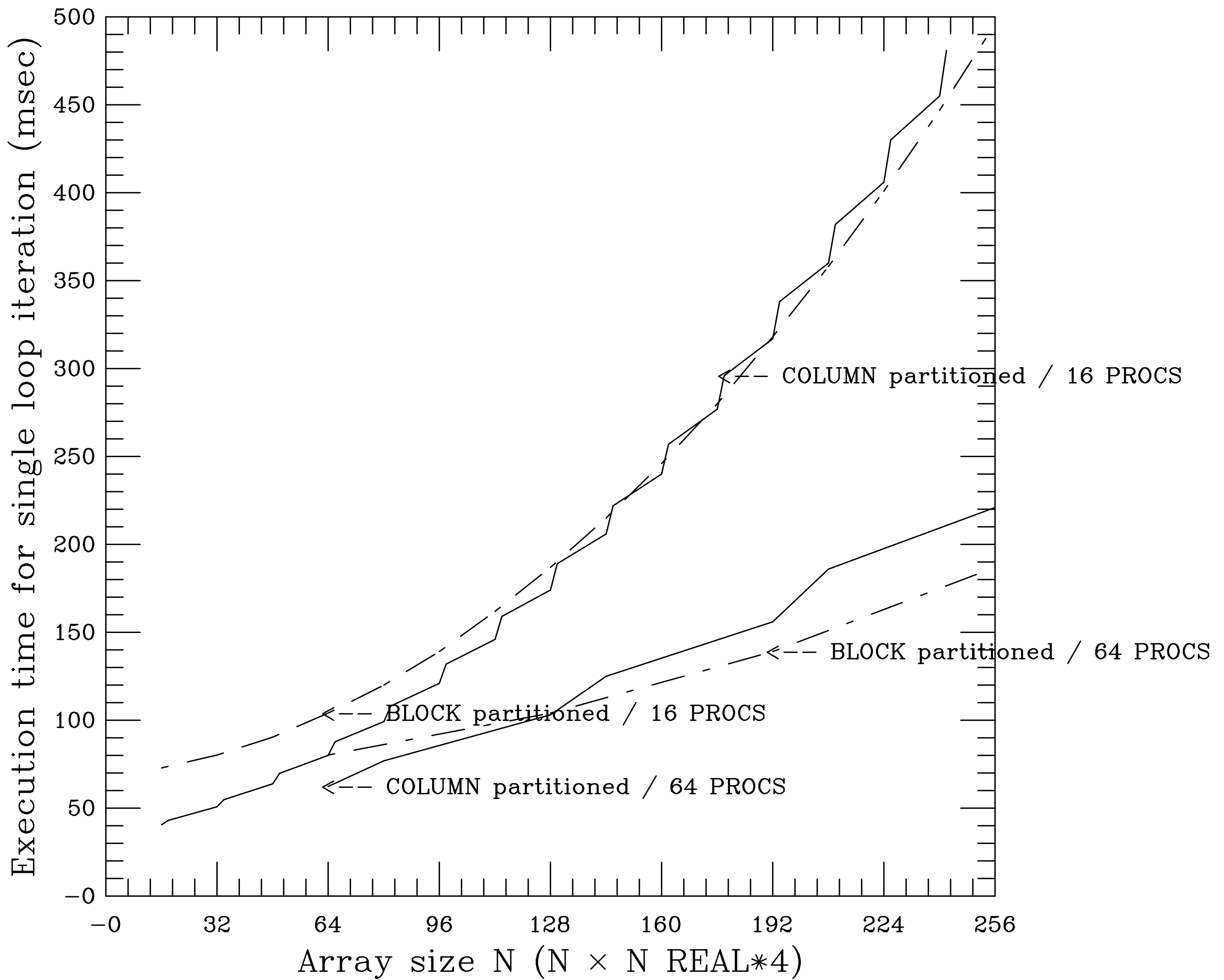

Figure 7: Estimated times on Ncube-1: FLOAT operations on 16 and 64 processors

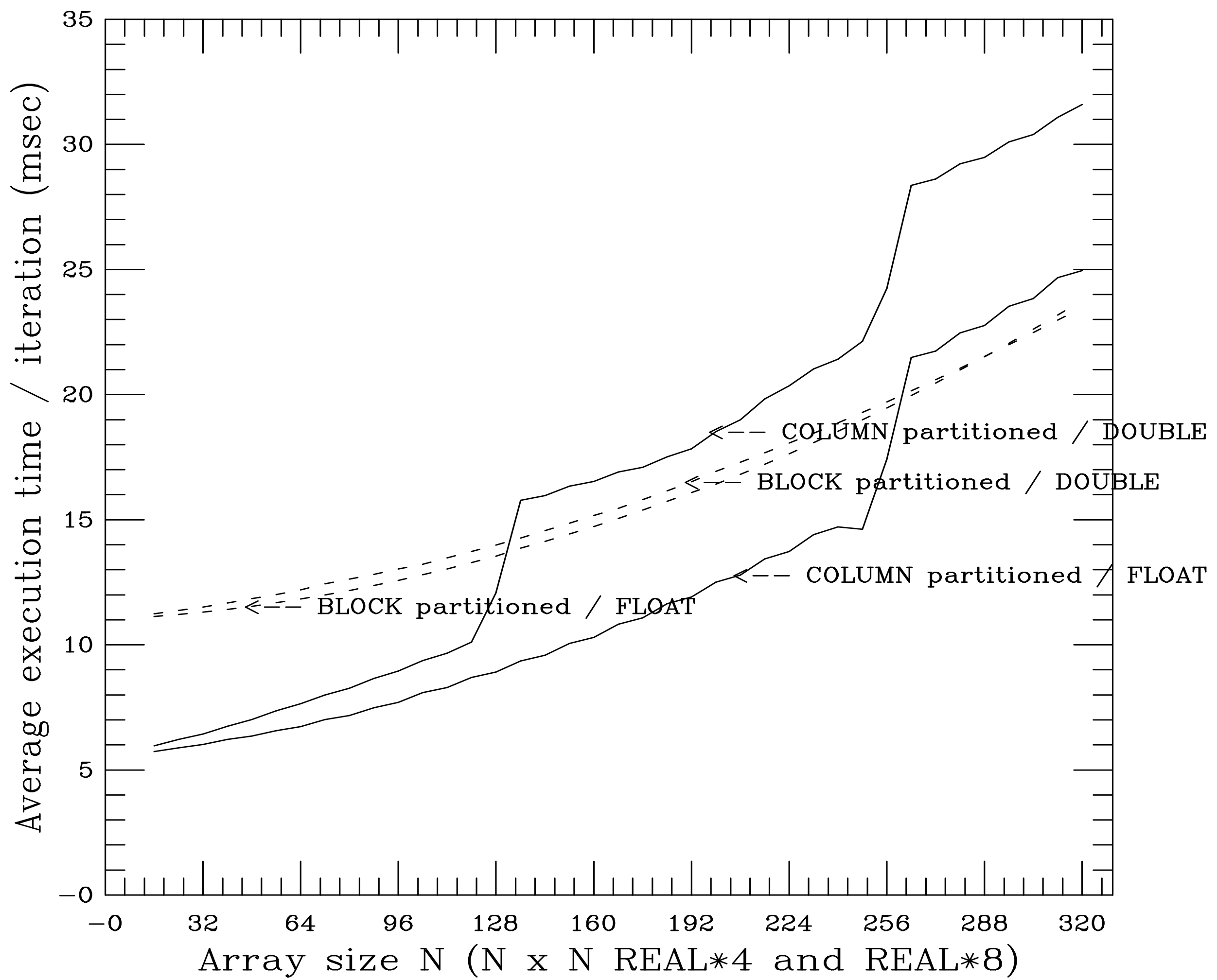

Figure 8: Estimated times on iPSC/860: DOUBLE PRECISION and FLOAT operations on 16 processors

thus convert any unrecognized Fortran D program fragment into an equivalent node program, and invoke the machine module to estimate its performance.

4.2 Data Alignment and Data Distribution

The analysis performed by the automatic data partitioner divides the program into separate *computation phases.* A phase consists of a group of statements that are mutually involved in a computation and therefore should be mapped onto the same Fortran D decomposition using `ALIGN` statements. In the absence of procedure calls we define a phase as follows: A phase is a loop nest such that for each induction variable that occurs in a subscript position of an array reference in the loop body the phase contains the surrounding loop that defines the induction variable. A phase is minimal in the sense that it does not include surrounding loops that do not define induction variables occurring in subscript positions. The maximal size and number of dimensions of arrays referenced in the phase defines the dimensionality and size of its associated Fortran D decomposition. For example, the red-black program in Figure 4 has four phases enclosed in the outer `k`-loop. Each phase has an associated Fortran D decomposition of dimensionality and size equal to the array `v`.

4.2.1 Intra-phase Alignment and Distribution

The *intra-phase* decomposition problem consists of determining a set of good data decompositions and their performance for each individual phase. The data partitioner first tries to match the phase or parts of the phase with computation patterns in the compiler training set. If a match is found, it returns the set of decompositions with the best measured performance as recorded in the compiler training set. If no match is found, the data partitioner must perform alignment and distribution analysis on the phase. The resulting solution may be less accurate since the effects of the Fortran D compiler and target machine can only be estimated.

Alignment analysis is used to prune the search space of possible arrays alignments by selecting only those alignments that minimize data movement. Alignment analysis is largely machine-independent; it is performed by analyzing the array access patterns of computations in the phase. We intend to build on the inter-dimensional and intra-dimensional alignment techniques of Li and Chen [LC90a] and Knobe *et al* [KLS90]. The alignment problems can be formulated as an optimization problem on an undirected, weighted graph. Some instances of the problem of alignment have been shown to be NP-complete [LC90a]. One major challenge in our proposed work will be to define the appropriate weights of the graph and to come up with a heuristic that solves the alignment problem in a way that is suitable for loosely synchronous problems.

Alignment analysis is compiler dependent. A compiler may recognize arrays that are used only as temporaries and therefore are allocated *locally* in each processor without inducing any communication. These *local* or *private* arrays are not considered during the alignment analysis.

Distribution analysis follows alignment analysis. It applies heuristics to prune unprofitable choices in the search space of possible distributions. Distribution analysis is compiler, machine, and problem dependent.

For instance, the compiler may not be able to generate efficient wavefront computations [Lam74] for a subset of distributions. Transformations like loop interchange and strip-mining can substantially improve the degree of parallelism induced by the wavefront [HKT91]. Distributions that sequentialize the computation are eliminated. Another consideration in our pruning heuristic are the sizes of the dimensions of the decomposition. If the size of a dimension is smaller than a machine dependent threshold, the dimension will always be localized. This eliminates all distributions

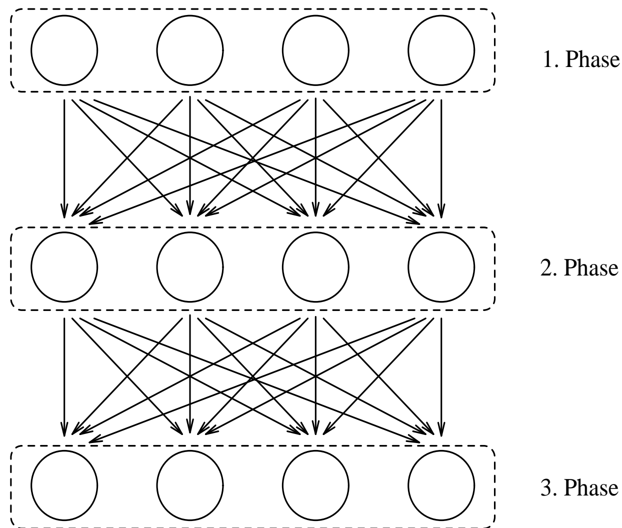

Figure 9: Interphase merge problem with realignment and redistribution

that map small dimensions to distinct processors. We will also restrict the possible block sizes in `BLOCK_CYCLIC` distributions to a reasonable subset of values.

After the automatic data partitioner has determined a set of reasonable data decomposition schemes, the static performance estimator is invoked to predict the performance of each reasonable scheme.

4.2.2 Inter-phase Alignment and Distribution

After computing data decomposition schemes for each phase, the automatic data partitioner must solve the *inter-phase* decomposition problem of merging individual data decompositions. It considers realigning and redistribution of arrays between computational phases to reduce communication costs or to improve the available parallelism.

The merging problem can be formulated as a single-source shortest paths problem over the *phase control flow graph*. The phase control flow graph is similar to the control flow graph [ASU86] where all nodes associated with a phase are substituted by nodes representing the set of reasonable data decomposition schemes for the phase. The static performance estimator is used to predict the costs for these reasonable decomposition schemes. The availability of fast collective communication routines will be crucial for the profitability of realignment and redistribution.

The merging problem for a linear phase control flow graph can be solved as a single-source shortest paths problem in a directed acyclic graph [CLR90]. For example, Figure 9 shows a three phase problem with four reasonable decompositions for each phase. Each decomposition scheme is represented by a node. The node is labeled with the predicted cost of the decomposition scheme for the phase. Edges between phases are labeled with the realignment and redistribution costs for the source and sink decomposition schemes. For each of the four decompositions of the first phase we will solve the single-source shortest paths problem. In general, let k denote the maximal number of decomposition schemes for each phase and p the number of phases. The resulting time complexity is $O(p \times k^3)$.

The merging of phases in a strongly connected component of the phase control flow graph should

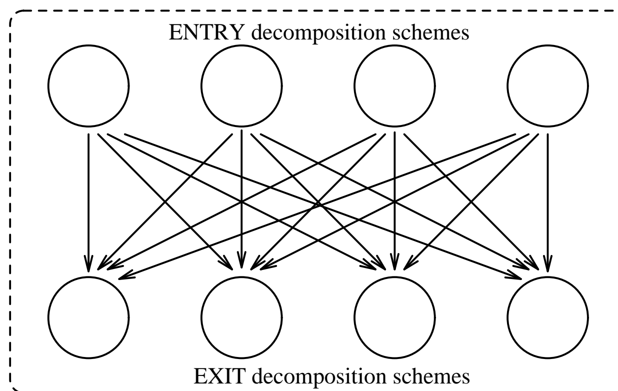

Figure 10: An example interval summary phase

be done before merging any of its phases with a phase outside of the strongly connected component. This suggests a hierarchical algorithm for merging phases based on, for example, Tarjan intervals [Tar74]. Assuming that the innermost loop bodies can be represented by a linear phase control flow subgraph, the merging problem is solved by adding a shadow copy of the first phase after the last phase in the linear subgraph keeping the subgraph acyclic. After solving the single-source shortest paths problem, the subgraph is collapsed into a single *interval summary phase* representing the different costs resulting from entering the interval with a decomposition scheme and exiting it with a possibly different scheme. An example interval summary phase is shown in Figure 10. In the resulting phase control flow graph we again identify the innermost loops and repeat the process of collapsing and summarizing until the phase control flow graph consists of a single node.

In fully automatic mode, the data partitioner selects the decomposition scheme that has the minimal cost for the shortest path from a decomposition in the first phase to a decomposition in the last phase of the selected program segment. Following the selected shortest path, `ALIGN`, and `DISTRIBUTE` statements are inserted if the decomposition at the source of an edge is different from the decomposition at the sink. `DECOMPOSITION` specifications are declared at the beginning of the subroutine containing the selected program segment.

4.2.3 The Algorithm

Figure 11 gives the basic algorithm for automatic data decomposition for a program segment without procedure calls. The algorithm does not handle control flow other than loops and does not consider data replication.

4.2.4 Automatic Decomposition in the Presence of Procedure Calls

One of the major challenges of the proposed research on automatic data decomposition is to devise techniques that can deal with procedure calls. Interprocedural analysis is used to allow the merging of computation phases across procedure boundaries.

Interprocedural phase merging is compiler dependent. In particular, the automatic data partitioner has to know whether and when the compiler performs interprocedural optimizations such as procedure cloning [CHK92, HHKT91] or procedure inlining [Hal91]. In the following we will assume that the compiler performs procedure cloning for every distinct pattern of entry and exit

Algorithm DECOMP

Input: program segment without procedure calls; problem sizes and number of processors to be used.
Output: data decomposition schemes for data objects referenced in the input program segment
with diagnostic information.

determine program phases of input program segment;
build phase control flow graph;
for each phase *do*
    perform alignment analysis;
    perform distribution analysis;
    generate diagnostic information, if available;
endfor
while phase control flow graph contains a loop *do*
    identify innermost loop (e.g. using Tarjan intervals);
    solve single-source shortest paths problem for this loop;
    identify realignment and redistribution points;
    generate diagnostic information, if available;
    substitute loop by its interval summary phase in the phase control flow graph;
endwhile
use the computed shortest path to generate `DECOMPOSITION`, `ALIGN`,
and `DISTRIBUTE` statements, if in fully automatic mode;

Figure 11: Basic Algorithm for Automatic Data Alignment and Distribution

decomposition schemes. We will only consider programs with acyclic call graphs.

The call graph is first traversed in topological order to propagate loop information into called procedures. This information is needed to identify single computation phases. Subsequently, the call graph is traversed in reverse topological order. For each procedure P the single-source shortest paths problem is solved on its phase control flow graph using the hierarchical approach of algorithm DECOMP in Figure 11. Each call site of P in procedure Q is represented by a copy of the interval summary phase of P in the phase control flow graph of Q. The information about the computed data decomposition schemes with their realignment and redistribution points are passed to the Fortran D compiler. The automatic data partitioner cannot insert the Fortran D data layout statements directly into the program since it does not actually perform the cloning of procedures.

If the compiler does not perform cloning a procedure can have only a single entry and exit decomposition scheme. The automatic data partitioner will use a heuristic to select the decomposition scheme. The heuristic takes the static execution count of call sites and the penalties due to mismatched decomposition schemes into account.

Using the outlined strategy the automatic system generates the same decomposition, alignment, and distribution for the programs of Figure 3 and Figure 12.

5 Success/Failure Criterion

The feasibility of the proposed automatic techniques depends on the 'quality' of the generated decomposition schemes compared to schemes that are considered optimal. We want to define our quality measure *prior* to an investigation into automatic techniques. This will guarantee an unbiased discussion of results gained by an empirical study.

We will use a benchmark suite being developed by Geoffrey Fox at Syracuse that consists of a collection of Fortran programs. Each program in the suite will have five versions:

- **(v1)** the original Fortran 77 program,
- **(v2)** the best hand-coded message-passing version of the Fortran program,
- **(v3)** a "nearby" Fortran 77 program,
- **(v4)** a Fortran D version of the nearby program, and
- **(v5)** a Fortran 90 version of the program.

The "nearby" version of the program will utilize the same basic algorithm as the message-passing program, except that all explicit message-passing and blocking of loops in the program are removed. The Fortran D version of the program consists of the nearby version plus appropriate data decomposition specifications.

To validate the automatic data partitioner, we will use it to generate a Fortran D program from the nearby Fortran program (v3). The result will be compiled by the Fortran D compiler and its running time compared with that of the compiled version of the hand-generated Fortran D program (v4). Our goal is that for 80% of the benchmark programs the nearby version with automatically generated data layout will run at most 20% slower than the hand-generated Fortran D program. We expect that such a performance degradation due to automatic data decomposition is still acceptable to the user.

We believe that in many cases it is easy for the user to specify the *problem mapping*, i.e. the data alignment onto a decomposition, since the problem mapping is determined by the structure

```
1      DOUBLE PRECISION v(N,N), a, b
2      DECOMPOSITION d(N,N)
3      ALIGN v(I,J) WITH d(I,J)
4      DISTRIBUTE d(BLOCK,BLOCK)
5      DO k = 1, M
6          // Compute the red points
7          CALL redpoints (v,a,b,N)
8          // Compute the black points
9          CALL blackpoints (v,a,b,N)
10     ENDDO
11     SUBROUTINE redpoints (v,a,b,n)
12     INTEGER n
13     DOUBLE PRECISION v(n,n), a, b
14         DO j = 1, N, 2
15             DO i = 1, N, 2
16                 v(i,j) = a*(v(i,j-1) + v(i-1,j) + v(i,j+1) + v(i+1,j)) + b* v(i,j)
17             ENDDO
18         ENDDO
19         DO j = 2, N, 2
20             DO i = 2, N, 2
21                 v(i,j) = a*(v(i,j-1) + v(i-1,j) + v(i,j+1) + v(i+1,j)) + b* v(i,j)
22             ENDDO
23         ENDDO
24     END
25     SUBROUTINE blackpoints (v,a,b,n)
26     INTEGER n
27     DOUBLE PRECISION v(n,n), a, b
28         DO j = 1, N, 2
29             DO i = 2, N, 2
30                 v(i,j) = a*(v(i,j-1) + v(i-1,j) + v(i,j+1) + v(i+1,j)) + b* v(i,j)
31             ENDDO
32         ENDDO
33         DO j = 2, N, 2
34             DO i = 1, N, 2
35                 v(i,j) = a*(v(i,j-1) + v(i-1,j) + v(i,j+1) + v(i+1,j)) + b* v(i,j)
36             ENDDO
37         ENDDO
38     END
```

Figure 12: Example loop nest with procedure calls.

of the underlying algorithm. However, the user will have difficulties to predict the combined effects of the compiler, problem, and machine characteristics on the performance of a specified data decomposition scheme. Therefore we expect the Fortran D program generated by the automatic data partitioner to do better than the hand-coded Fortran D version for some programs of the benchmark suite.

Initially, the automatic techniques will be validated by applying them to whole programs in the benchmark suite. If automatic techniques will fail for whole programs, we want to understand why this is the case and how user interaction can overcome their deficiencies.

6 Related Work and Research Goals

Our proposed techniques for automatic data decomposition are based on previous work done by

1. Knobe, Lukas at Compass, Steele at Thinking Machines Corporation, and Natarajan at Motorola [KLS90, KN90],
2. Li, Chen, and Choo for the Crystal project at Yale University [CCL89, LC90a, LC91b, LC91a, LC90b], and
3. Balasundaram, Fox, Kennedy, and Kremer at Caltech and Rice University in the context of the Fortran D distributed memory compiler and environment project [FHK+90, HKT92b, HKT91, HHKT91, BFKK90, BFKK91, HKK+91].

Knobe, Lukas, Natarajan and Steele deal with the problem of automatic data layout for SIMD architectures such as the Connection machine. They discuss an algorithm that uses heuristics to solve conflicting requirements between minimizing communication and maximizing parallelism. They also observed the problem of data replication of arrays that are used solely as temporaries.

Li, Chen, and Choo discuss automatic data decomposition in the context of a functional language. They introduce the notion of index domains for phases of the computation. After alignment, i.e. mapping data objects or functions into the common index domain of the phase, the functional program is transformed into an imperative, intermediate form. Data partitioning is defined on the data objects of the intermediate program and is specified by the user at execution time of the program. Li, Chen, and Choo also briefly discuss ideas for automatic data partitioning. Given a data partitioning strategy, a performance model is used to determine optimal communication, selecting efficient collective communication routines whenever possible. In contrast to the Crysal project, the input to our system is an imperative language, namely Fortran. Since Crystal starts out with a functional program many functional characteristics are still present in the imperative, intermediate code.

Balasundaram, Fox, Kennedy, and Kremer proposed an interactive performance estimation tool that helps the user to understand the effects of a user specified decomposition with respect to communication time and overall execution time [BFKK90] without actually compiling and executing the program. The described techniques are based on the knowledge about the compiler, the actual problem size, and the number of processors to be used. In [BFKK91], techniques are discussed

that allow static performance estimation with high accuracy for a large class of loosely synchronous problems.

Some approaches to automatic data decomposition concentrate on single loop nests. The iteration space is first partitioned into sets of iterations that can be executed independently. The data mapping is determined by the iterations that are assigned to the different processors [RS89, Ram90, D'H89, KKBP91]. Other techniques for single loop nests are based on recognizing specific computation patterns in the loop, called *stencils* [SS90, HA90, IFKF90, GAY91]. This more abstract representation is used to find a good data mapping.

Our approach to automatic data alignment and distribution is closely related to the work done by Gupta and Banerjee at the University of Illinois at Urbana-Champaign [GB90, GB91, GB92], Wholey at Carnegie Mellon University [Who91], and Chapman, Herbeck, and Zima at the University of Vienna [CHZ91]. The described techniques for automatic data decomposition work on whole programs taking machine characteristics and problem characteristics into account. The automatic data partitioner is an integral part of the compiler.

The major contributions of our research are

1. the development of new techniques for automatic data alignment and distribution. These techniques consider realignment, redistribution, and data replication to minimize communication or enhance the available parallelism in the program. The techniques have to consider procedure calls and control flow.

2. the design of a portable automatic data decomposition tool that achieves compiler and machine independence by using compiler and machine level training sets.

3. the validation of the new techniques using a suite of benchmark programs.

4. the answer to the question whether there is a data parallel programming style that allows the automatic data decomposition of regular loosely synchronous programs across a variety of distributed memory machines.

The proposed automatic techniques will be implemented as part of the ParaScope parallel programming environment adapted to distributed memory multiprocessors [BKK+89, KMT91, BFKK90, HKK+91]. A prototype of the machine module of the static performance estimator is already available [BFKK91].

7 Acknowledgement

We wish to thank Seema Hiranandani and Chau-Wen Tseng for many discussions and helpful comments on the content of this paper. Many thanks also to Joe Warren for his comments and ideas, and to the ParaScope research group for providing the underlying software infrastructure for the Fortran D programming system.

References

[AK87] 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.

[ASU86] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, second edition, 1986.

[BFKK90] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[BFKK91] 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*, Williamsburg, VA, April 1991.

[BKK+89] V. Balasundaram, K. Kennedy, U. Kremer, K. S. McKinley, and J. Subhlok. The ParaScope Editor: An interactive parallel programming tool. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[CCL88] M. Chen, Y. Choo, and J. Li. Compiling parallel programs by optimizing performance. *Journal of Parallel and Distributed Computing*, 1(2):171–207, July 1988.

[CCL89] M. Chen, Y. Choo, and J. Li. Theory and pragmatics of compiling efficient parallel code. Technical Report YALEU/DCS/TR-760, Dept. of Computer Science, Yale University, New Haven, CT, December 1989.

[CHK92] K. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the 1992 IEEE International Conference on Computer Language*, Oakland, CA, April 1992.

[CHZ91] B. Chapman, H. Herbeck, and H. Zima. Automatic support for data distribution. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[CKK89] D. Callahan, K. Kennedy, and U. Kremer. A dynamic study of vectorization in PFC. Technical Report TR89-97, Dept. of Computer Science, Rice University, July 1989.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[D'H89] E. D'Hollander. Partitioning and labeling of index sets in do loops with constant dependence. In *Proceedings of the 1989 International Conference on Parallel Processing*,

St. Charles, IL, August 1989.

[EXP89] Parasoft Corporation. *Express User's Manual*, 1989.

[FF88] W. Furmanski and G. Fox. Optimal communication algorithms for regular decompositions on the hypercube. Technical Report C3P-314B, California Institute of Technology, March 1988.

[FHK+90] 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.

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[GAY91] E. Gabber, A. Averbuch, and A. Yehudai. Experience with a portable parallelizing Pascal compiler. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[GB90] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. Technical Report CRHC-90-14, Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, October 1990.

[GB91] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[GB92] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, April 1992.

[Ger89] M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, University of Bonn, December 1989.

[Ger90] M. Gerndt. Updating distributed variables in local computations. *Concurrency—Practice & Experience*, 2(3):171–193, September 1990.

[HA90] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[Hal91] M. W. Hall. *Managing Interprocedural Optimization.* PhD thesis, Rice University, April 1991.

[HHKT91] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. Technical Report TR91-169, Dept. of Computer Science, Rice University, November 1991.

[HKK+91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[HKT91] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[HKT92a] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[HKT92b] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors.* Elsevier, Amsterdam, The Netherlands, to appear 1992.

[IFKF90] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[Kar87] A. Karp. Programming for parallelism. *IEEE Computer*, 20(5):43–57, May 1987.

[KKBP91] D. Kulkarni, K. 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*, Cologne, Germany, June 1991.

[KKP+81] 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.

[KLS90] 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.

[KM91] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[KMT91] K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[KMV90] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[KN90] K. Knobe and V. Natarajan. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[KZBG88] U. Kremer, H. Zima, H.-J. Bast, and M. Gerndt. Advanced tools and techniques for automatic parallelization. *Parallel Computing*, 7:387–393, 1988.

[Lam74] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[LC90a] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[LC90b] J. Li and M. Chen. Synthesis of explicit communication from shared-memory program references. Technical Report YALEU/DCS/TR-755, Dept. of Computer Science, Yale University, New Haven, CT, May 1990.

[LC91a] 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.

[LC91b] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(4):213–221, August 1991.

[PSvG91] E. Paalvast, H. Sips, and A. van Gemund. Automatic parallel program generation and optimization from data decompositions. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[RA90] R. Ruhl and M. Annaratone. Parallelization of FORTRAN code on distributed-memory parallel processors. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[Ram90] J. Ramanujam. *Compile-time Techniques for Parallel Execution of Loops on Distributed Memory Multiprocessors.* PhD thesis, Department of Computer and Information Science, Ohio State University, Columbus, OH, 1990.

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[RS89] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[RSW88] M. Rosing, R. Schnabel, and R. Weaver. Dino: Summary and examples. Technical

Report CU-CS-386-88, Dept. of Computer Science, University of Colorado, March 1988.

[SCMB90] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, April 1990.

[SS90] L. Snyder and D. Socha. An algorithm producing balanced partitionings of data arrays. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[Tar74] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[TMC89] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, version 5.2-0.6 edition, September 1989.

[Who91] S. Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991.

[Wol89] M. J. Wolfe. Semi-automatic domain decomposition. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[WSHB91] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[ZBG88] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.