ACM International Conference on Supercomputing, Tokyo, Japan, July 1993

PARADIGM: A Compiler for Automatic Data Distribution on Multicomputers*

Manish Gupta
IBM T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598

Prithviraj Banerjee
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

One of the most challenging steps in developing a parallel program for a distributed memory machine is determining how data should be distributed across processors. Most of the compilers being developed to make it easier to program such machines still provide no assistance to the programmer in this difficult and machine-dependent task. We have developed PARADIGM, a compiler that makes data partitioning decisions for Fortran 77 procedures. A significant feature of the design of PARADIGM is the decomposition of the data partitioning problem into a number of sub-problems, each dealing with a different distribution parameter for all the arrays. This paper presents the algorithms that, in conjunction with the computational and the communication cost estimators developed by us, determine those distribution parameters. We also present results obtained on Fortran procedures taken from the Linpack and Eispack libraries, and the Perfect Benchmarks. We believe these are the first results demonstrating the success of automatic data partitioning on a significant class of Fortran procedures.

1 Introduction

Distributed-memory machines (multicomputers) have started playing an increasingly important role in delivering high levels of performance for numerous scientific applications. Unfortunately, such machines are not very easy to program. The programmer has to perform low-level tasks like distributing data across processors, and managing the communication among those processors explicitly. A number of research efforts seek to alleviate this problem by developing compilers that relieve the programmer of the burden of explicit message-passing. These compilers take a program written in a sequential or shared-memory parallel language, and based on user-specified partitioning of data, generate the target SPMD (Single Program Multiple Data) program for a multicomputer. Examples of such systems include the Fortran D [9], SUPERB [22], Oxygen [18], and the Dataparallel C [16] compilers.

The communication overheads and the extent of parallelism exploited in the resulting target program are determined largely by the manner in which data is partitioned across different processors of the machine. Most of the compilers provide no assistance to the programmer in the crucial task of determining a good data partitioning scheme. Automatic data partitioning offers the following advantages:

- *Reduced burden on the programmer*: The programmer is free to concentrate on the high-level design of the algorithm.
- *Machine-independence*: In general, the best data partitioning scheme depends not only on the program characteristics, but also on numerous machine-specific parameters. Therefore, true portability can only be achieved if the partitioning scheme is not specified as part of the program.
- *Relationship with compiler optimizations*: The programmer may not be familiar with the program transformations and the optimizations performed by a distributed-memory compiler. Hence, he or she may not have a good idea about the implications of certain data partitioning decisions on the performance of the compiled, message-passing program. A compiler would more easily be able to incorporate that information in the process of choosing the data partitioning scheme.

This paper describes PARADIGM, a compiler we have developed for automatic data partitioning on multicomputers. PARADIGM accepts Fortran 77 procedures, and outputs a description of the partitioning scheme for each array used in the procedure. It also accepts programs in which the parallelizable loops are explicitly marked as *doall* loops. The programmer interactively supplies any information missing at compile time about the array sizes and the loop bounds in the program. In conjunction with a small machine-dependent module that provides the cost model for the target machine, PARADIGM can be used on almost any distributed-memory machine. To allow validation of the underlying concepts, we have developed a version for the Intel iPSC/2 hypercube.

Some of the early ideas that led to the development of this compiler have been described in [7]. This paper

*This research was supported in part by the Office of Naval Research under Contract N00014-91J-1096, and in part by National Aeronautics and Space Administration under Contract NASA NAG 1-613. Have to leave some extra space. This research was supported in part by the Office of Naval Research under Contract N00014-91J-1096, and in part by National Aeronautics and Space Administration under Contract NASA NAG 1-613. This research was supported in part by the Office of Naval Research under Contract N00014-91J-1096. This research was supported in part by the Office of Naval Research under Contract N00014-91J-1096. This research was supported in part by the Office of Naval Research under Contract N00014-91J-1096.

describes the specific algorithms used in various passes of PARADIGM in which data distribution decisions are made. It also presents the experimental results obtained on real-life Fortran procedures taken from the Linpack and Eispack libraries, and the Perfect Benchmarks [4]. To the best of our knowledge, these are the first results demonstrating the success of automatic data partitioning on a significant class of Fortran procedures.

2 Overview of the System

The abstract target machine we assume is a D-dimensional mesh (D is the maximum dimensionality of any array used in the program) of N_1 x N_2 x ... x N_D processors. The distribution of an array is specified through separate distribution functions for each of its dimensions. Each array dimension A_k is mapped to a unique dimension $map(A_k), 1 \le map(A_k) \le D$ of the processor-mesh, over which it is partitioned or replicated. In case it is partitioned, the distribution function $f(A_k, i)$ specifies the position in the mesh dimension to which the ith element along A_k is mapped. It is of the form:

$$f(A_k, i) \quad = \quad \lfloor \frac{i - offset}{block} \rfloor [\mathrm{mod} N_{map(A_k)}]$$

The square parentheses surrounding mod $N_{map(A_k)}$ indicate that the appearance of this part in the expression is optional. The array distributions captured by this formulation are the same as those supported by primitives for data placement in languages like Fortran D [9] and High Performance Fortran [5]. Each distribution function is characterized by a number of parameters, such as the mesh dimension to which the array dimension is mapped, whether the method of partitioning is blocked or cyclic, and the number of processors in the mesh dimension. In the special case when the number of processors in a mesh dimension is one, the array dimensions mapped to it are said to be *collapsed.* Currently, PARADIGM sets the *offset* value to 1 for all the distribution functions, and replicates only "small" arrays with fewer elements than a certain threshold. For an array with fewer dimensions than the processor mesh, the distribution functions of the "missing" array dimensions are restricted to constant values.

The overall structure of PARADIGM is shown in Figure 1. PARADIGM has been built on top of the Parafrase-2 compiler [15], which is used to provide information about the program, such as the parse tree structure, the data dependences, and the control-flow information. The internal program representation used by Parafrase-2 has been extended to incorporate some additional information, such as a detailed characterization of array subscripts, and a count of different operations in assignment statements [6].

After the relevant program information is gathered, PARADIGM makes data partitioning decisions in a number of distinct passes through the program. Each pass determines a single parameter of the distribution functions of all the arrays. The `align` pass maps each array dimension to a processor-mesh dimension, based on considerations of alignment between array dimensions. The `block-cyclic` pass determines for each array dimension, whether it should be distributed in a blocked or cyclic manner. The `block-size` pass determines the block size to be used for each dimension distributed in a cyclic manner. The `num-procs` pass determines the number of processors in each of the processor-mesh dimensions to which various array dimensions are mapped.

While a certain amount of coupling exists among the procedures for choosing each partitioning parameter, we have used the above ordering of passes based on the *extent* to which we believe decisions in one pass depend on those in the others. For instance, the `align` pass precedes the `block-cyclic` pass, because the decisions on alignment of array dimensions are quite independent of whether the dimensions are distributed in a blocked or cyclic manner. On the other hand, the decisions made in the `block-cyclic` pass are critically dependent on the alignment information. PARADIGM assigns the same method of partitioning, blocked or cyclic, to each aligned dimension, to ensure the appropriate alignment of distributed arrays.

In each pass, PARADIGM first identifies desirable requirements on the relevant distribution parameter of arrays referenced in each statement. These desirable requirements are referred to as *constraints* [7]. For each constraint, the compiler determines a *quality measure* that captures its importance with respect to the performance of the program. When different parts of a program give rise to conflicting requirements on data distribution, the compiler resolves those conflicts based on the corresponding quality measures.

Thus, each data distribution pass is logically composed of three modules. The `detector` module records constraints, and the `driver` module invokes the communication cost estimator and/or the computational cost estimator to obtain the quality measure of each constraint. Once all the constraints and their quality measures have been recorded, the `solver` determines the value of the corresponding data distribution parameter. Essentially, the `solver` obtains (an approximate) solution to an optimization problem, where the objective is to minimize the execution time of the target data-parallel program.

The computational and the communication cost estimators together determine the contribution of each statement to the expected program execution time, given the partitioning information about various arrays. These estimators are based on the *owner computes* rule [9], which makes the processor that owns a data item responsible for its computation. Since the complete data partitioning information is not known when these cost estimates are needed, the `driver` module of each pass specifies some appropriate, default values for the distribution parameters that are unknown. Further details on the cost estimators are given in [8, 6]. In this paper, we shall limit our discussion to the data distribution passes, and their interaction with the cost estimators.

3 Data Distribution Passes

3.1 Align Pass

The `align` pass identifies the constraints on alignment among various array dimensions, and groups them into classes that would be mapped to the same processor-mesh dimension. For this problem, we use the *Component Affinity Graph* (CAG) framework, introduced by Li and Chen [14].

The CAG of a program is a weighted graph that has

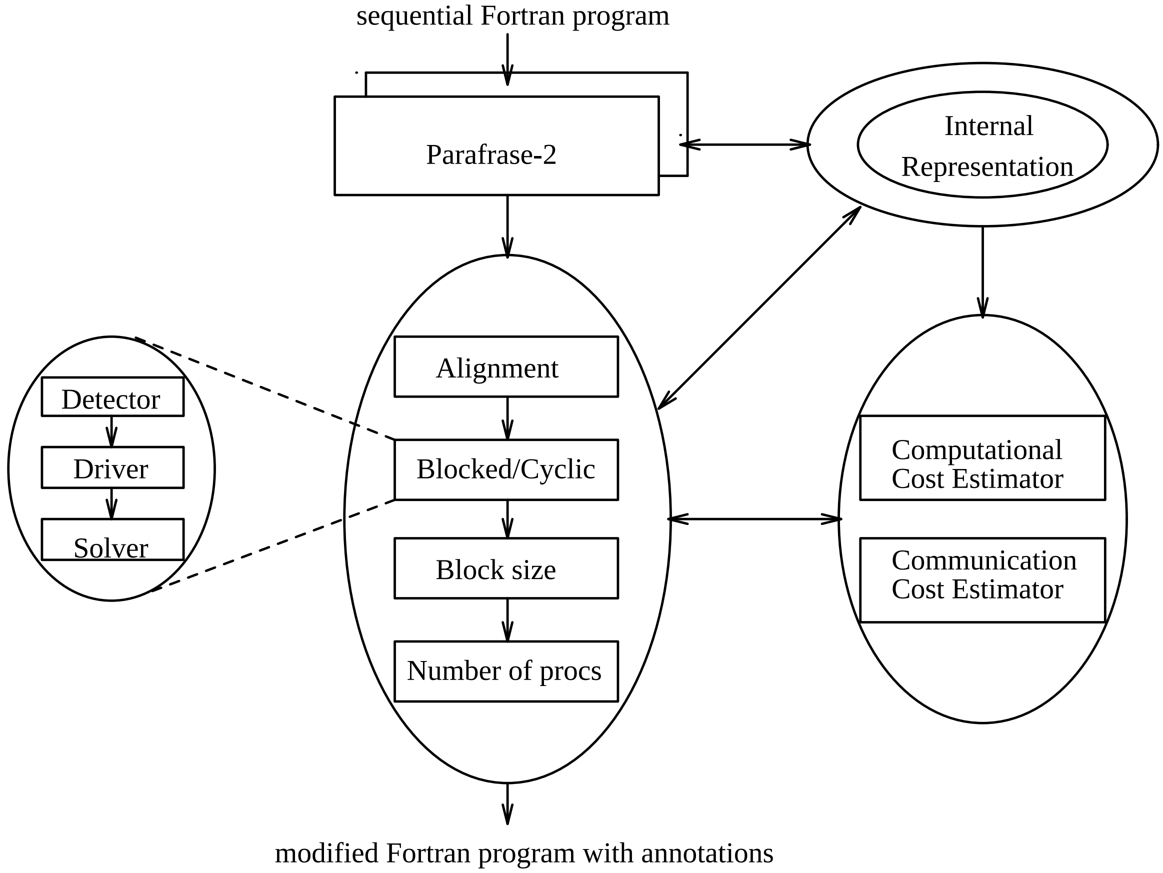

Figure 1: Overview of automatic data partitioning in PARADIGM

nodes representing array dimensions, and edges representing constraints on alignment between array dimensions, with weights representing the importance of honoring those alignment relationships. The component alignment problem involves partitioning the node set of CAG into disjoint subsets (that identify classes of mutually aligned array dimensions) such that the total weight of edges across nodes in different subsets is minimized. Our main contribution to this specific problem is in the method of assigning weights to the edges such that these weights reflect the communication costs saved by aligning the corresponding array dimensions.

Detection of Constraints The compiler performs a pairwise analysis of the `lhs` array reference with each `rhs` array reference (for a different array) in every assignment statement inside a loop. It looks for a pair of subscripts (one each from the `lhs` and `rhs` references) that are affine functions of the same loop index. The presence of such a pair suggests an alignment constraint on the corresponding array dimensions. For example, the program segment shown in Figure 2 leads to alignment constraints being recorded for A_1 and B_2, and for A_2 and B_1.

The significance of checking for the above condition is that it represents a case when the suggested alignment of array dimensions is needed to reduce, or completely eliminate interprocessor communication for the `rhs` reference, if those dimensions are distributed on more than one processor. In the above example, communication can be eliminated by further choosing identical distributions for A_2 and B_1, and distributions that give B_2 thrice the block size of A_1 but are identical in other regards (blocked/cyclic).

do $j = 1, n$
  do $i = 1, n$
    $A(i, j) = \mathcal{F}(B(j, 3 * i))$
  enddo
enddo

Figure 2: References suggesting alignment constraints

Determination of Quality Measure Consider an alignment constraint between a pair of array dimensions. Such dimensions are referred to as forming a *primary pair*. The weight on the corresponding edge in CAG should be set to the difference in communication costs for the `rhs` reference when the dimensions in that pair are aligned, and when they are not. However, if either of those arrays has three or more dimensions, there is no unique mapping (of the dimensions of the two arrays to mesh dimensions) corresponding to the alignment or mis-alignment of the primary pair. Thus, there is no definite basis for obtaining communication cost estimates in that case.

To overcome this problem, PARADIGM examines all alignment constraints for a given `lhs` and `rhs` array reference together. If there are an even number of primary pairs, it obtains edge weights for two pairs at a time. It first invokes the communication cost estimator to return the cost estimate, say, t_1, when the array dimensions are mapped so as to honor the alignment constraints. Next, it obtains the cost estimate, say, t_2, when the alignment of the two pairs of dimensions is swapped. The quality measure of each of those constraints is set to $(t_2 - t_1)/2$. This accounts for the fact that under the given assumption about each array dimension being distributed, if one alignment constraint is not satisfied, the other will also not be satisfied. Hence, their quality measures are assigned values that will add up to give the performance penalty of $t_2 - t_1$.

In case there are an odd number of primary pairs, the compiler first selects a dimension each from both the arrays to add another pair of dimensions, referred to as a *secondary pair*, and then follows the above procedure. The selection of such a pair is trivial if the arrays have two or fewer dimensions, as shown in Figure 3. The solid lines in the figure link primary pairs, and the dotted lines link secondary pairs. The solid circles denote actual array dimensions, while the dotted circles denote missing dimensions. For an array with more than one candidate for being part of a secondary pair, the compiler selects a dimension which "exhibits" parallelism somewhere in the program. An array dimension is said to exhibit parallelism if it is traversed in some parallelizable loop, indicating that it is a candidate for being distributed on more than one processor.

Clearly, besides alignment information, the communication cost estimator requires knowledge of the other data partitioning parameters, which have not yet been determined. The compiler obtains the cost estimates under the assumption that all array dimensions are distributed in a blocked manner. Furthermore, the estimates are obtained as functions of the number of processors in the two mesh dimensions, and the numerical values for the purpose of resolving conflicts are obtained assuming that both pairs of array dimensions are distributed on an equal number of processors. Our experimental results suggest that the resulting cost estimates, in spite of the possible inaccuracy due to these assumptions, usually guide the alignment decisions in the right direction.

Determination of Distribution Parameter The component alignment problem has been shown to be NP-complete, and a heuristic algorithm is given in [14]. PARADIGM uses a similar algorithm [6], based on weighted bipartite graph matching.

3.2 Block-Cyclic Pass

Constraints for Blocked Distribution The compiler analyzes the communication requirement of each `rhs` array reference, given the information about mapping of array dimensions to mesh dimensions. If the communication in any mesh dimension is recognized as a nearest-neighbor communication, it indicates the need for blocked distribution. As shown in Figure 4, the use of blocked distribution allows the communication to be restricted to the boundaries of regions assigned to processors. Let the subscript expressions corresponding to a pair of aligned dimensions in the `lhs` and the `rhs` array references be $\alpha_1 * i + \beta_1$, and $\alpha_2 * i + \beta_2$ respectively. The test for nearest-neighbor communication checks for the following conditions:

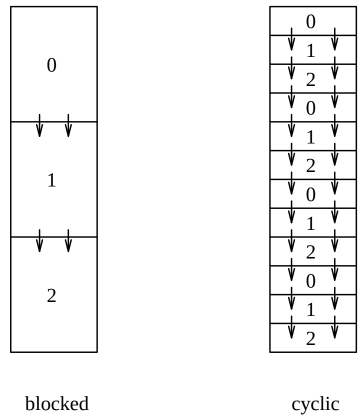

Figure 4: Need for blocked distribution

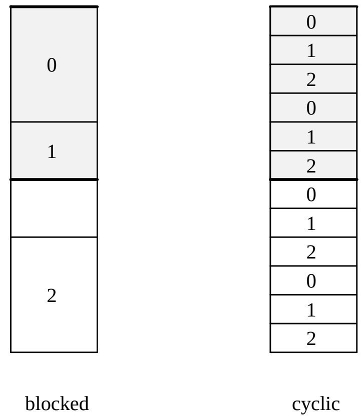

Figure 5: Need for cyclic distribution

1. $\alpha_1/b_1 = \alpha_2/b_2$.
2. $|(\beta_1 - o_1)/b_1 - (\beta_2 - o_2)/b_2| \leq 1$.

where b_1 and b_2 are the block sizes, and o_1 and o_2 are the offsets of distributions of the two dimensions.

Constraints for Cyclic Distribution The main motivation for distributing an array dimension in a cyclic manner is to get better load balance for parallelizable computations, under special conditions. Consider an assignment statement in which the `lhs` array is only partially traversed, as indicated by the filled region in Figure 5. Since the computation is partitioned according to the *owner computes* rule, using cyclic distribution with a small block size leads to a more even distribution of computation than using blocked distribution.

For every `lhs` array reference in an assignment statement, the compiler examines the variation of each subscript in loops surrounding the statement. If the ratio of the extent of traversal to the size of dimension is less than a certain threshold (the value of this threshold is set to 2/3 in the current implementation), *and* if the given statement is parallelizable with respect to the loop, the compiler records a constraint on cyclic distribution for the given array dimension.

Determination of Quality Measures The quality measure of a constraint for blocked (cyclic) distribution

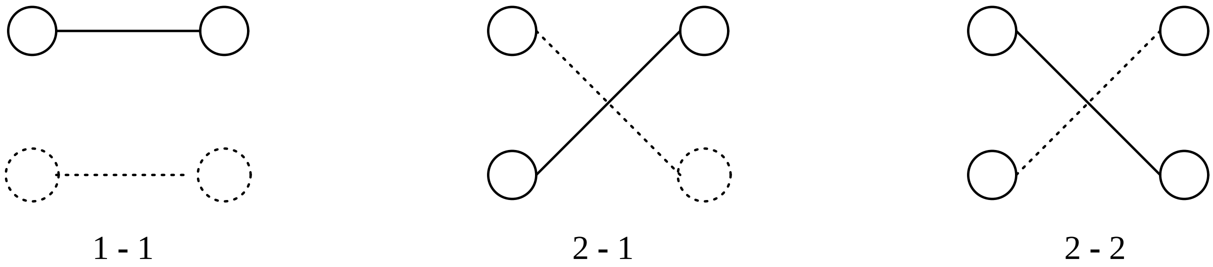

Figure 3: Identification of the secondary pair of dimensions

```
do i = 2, n
   D(i) = F(D(i − 1))
enddo
```

Figure 6: References suggesting constraint for blocked distribution

```
do i = 1, n
   do j = 1, i
      D(j) = ···
```

Figure 7: Reference suggesting constraint for cyclic distribution

is an estimate of the penalty incurred in execution time if the array dimension is instead given a cyclic (blocked) distribution. The compiler first obtains the expression for cost estimate, given a general, blocked-cyclic distribution (of which, blocked and cyclic distributions are special cases). Then it simply evaluates the expression for different values of block sizes corresponding to the two cases.

Blocked Distribution Consider the statement shown in Figure 6. The analysis of the given pair of references shows a need for boundary Transfers inside the i-loop, and hence suggests a constraint for blocked distribution of D_1. Given a blocked-cyclic distribution of D_1, the communication cost for the `rhs` reference is estimated by the compiler as:

$$\text{Cost} = (\lceil (n-1)/b_1 \rceil - 1) * \text{Transfer}(1),$$

where b_1 is the block size of distribution of D_1, and Transfer(1) represents the time taken to transfer a single word from one processor to another on the target machine. Let N_1 denote the number of processors on which D_1 is distributed. The quality measure for the constraint is obtained as the difference in the communication costs when the block size is 1 (corresponding to cyclic distribution), and when the block size is $\lceil n/N_1 \rceil$ (corresponding to blocked distribution). Thus, the compiler obtains the following value:

$$\begin{aligned}\text{Quality measure} \;&=\; (n-2) * \text{Transfer}(1) - \\ &\quad (\lceil (n-1)/(\lceil n/N_1 \rceil) \rceil - 1) * \text{Transfer}(1)\end{aligned}$$

Cyclic Distribution As an example of a constraint favoring cyclic distribution, consider the program segment shown in Figure 7. Let the array D consist of n elements. The compiler obtains the *expected value* of the range of the j-loop as $\lceil n/2 \rceil$. The extent of traversal of the subscript j over D_1 during the j-loop is accordingly determined as $\lceil n/2 \rceil$. Hence, the compiler recognizes the need for cyclic distribution of D_1 in order to obtain better load balance. Let t denote the estimated computational time for executing one instance of the given statement. The overall computational cost estimate for the statement is obtained as:

$$\begin{aligned}\text{Cost} \;&=\; t * n * (b_1 * \lfloor \lceil n/2 \rceil / (N_1 * b_1) \rfloor + \\ &\quad \min(b_1, \lceil n/2 \rceil \bmod (N_1 * b_1)))\end{aligned}$$

The quality measure of the constraint is given by the difference in computational cost when the block size is $\lceil n/N_1 \rceil$ (for blocked distribution), and when the block size is 1 (for cyclic distribution). The estimated cost amounts to $t * n * \lceil n/N_1 \rceil$ for blocked distribution, and to $t * n * \lceil n/(2 * N_1) \rceil$ for cyclic distribution. This corresponds to PARADIGM estimating the speedup roughly as $N_1/2$ for blocked distribution, and N_1 for cyclic distribution, and hence recording a constraint favoring cyclic distribution, with an appropriate quality measure.

Determination of Distribution Parameter The aligned dimensions of arrays that cross-reference each other should be given the same kind of partitioning: blocked or cyclic, to ensure the intended alignment of array elements. Two arrays A and B are said to be CR-related if there is an assignment statement in the program that has A (B) as its `lhs` array, and B (A) as its `rhs` array. PARADIGM keeps track of the transitive closure of the CR-relation between arrays, using the well known *union* and *find* algorithms for sets. Given the above information, the `solver` module makes collective decisions on the method of partitioning for all the aligned dimensions that correspond to arrays in the same CR set. In the context of HPF [5], this corresponds to determining the method of partitioning for a template dimension, so that all the array dimensions aligned to it are distributed in a compatible manner. For each such group of array dimensions, the `solver` compares the sum of quality measures of constraints favoring blocked distribution and those favoring cyclic distribution, and chooses the one with the higher value. In case of a tie (for instance, if there are no constraints either way), it chooses blocked partitioning.

3.3 Block-Size Pass

While the compiler makes a choice between the extreme positions of blocked and cyclic distribution during the block-cyclic pass, further adjustments on the block size of dimensions given a cyclic distribution are made during the block-size pass.

Detection of Constraints The compiler examines the subscripts corresponding to aligned dimensions for each pair of lhs and rhs array references in an assignment statement. Block-size constraints are recorded if both the subscripts are affine functions of the same loop index. Let the subscripts be of the form $\alpha_1 * i + \beta_1$, and $\alpha_2 * i + \beta_2$, with $\omega_1 = |\alpha_1|$, and $\omega_2 = |\alpha_2|$. The constraint suggests two related requirements: (i) the block sizes of (distributions of) the two array dimensions should respectively be multiples of ω_1 and ω_2, and (ii) the ratio of block sizes of those dimensions should be ω_1/ω_2. The first requirement represents one of the conditions for regularity of data movement in case of interprocessor communication. The second requirement is essential for proper alignment of the elements of the two arrays. Both of these requirements are satisfied if the block sizes chosen for the two dimensions are $k * \omega_1$ and $k * \omega_2$, where k is an integer, preferably as small as possible.

The block-size constraints and their quality measures are represented by means of an undirected graph, the *Block-size Constraint Graph* (BCG). The BCG has nodes representing array dimensions, and edges representing the block-size constraints between aligned dimensions. Each edge is marked with two items of information: (i) the coefficients ω_1 and ω_2 associated with the given constraint, and (ii) the value of weight, that is equal to the quality measure of that constraint.

Determination of Quality Measures The quality measure of a block-size constraint between two array dimensions is an estimate of the extra communication cost incurred if the conditions associated with the given constraint are not met. The driver module first obtains cost estimates for the case when the block sizes of the dimensions are set to ω_1 and ω_2, and then for a default case when a different relationship holds between the two block sizes.

Determination of Distribution Parameter The objective of the solver module is to select the block sizes such that the sum of quality measures of constraints that are not satisfied is minimized. The following theorem indicates a sufficient condition for the absence of conflicts among constraints.

Theorem 1 *The conditions associated with all of the block-size constraints in a program can be satisfied if there are no cycles in the BCG.*

Proof: In the absence of cycles, each connected component of the BCG is a tree. For a tree, the following algorithm selects the block sizes corresponding to all its nodes such that every block-size constraint between them is satisfied.

1. Choose an arbitrary node r in the tree, and any edge incident on that node. Set the block size of the node r to the value of the coefficient ω_1 marked for r on the edge.

2. For each node v visited in a *preorder* traversal starting from node r do
   (a) Let b_1 be the block size of the previous node p (keep track of the last node visited). Let ω_1 and ω_2 be the values of coefficients associated with the nodes p and v respectively on the current edge.
   (b) If $b_1 * \omega_2$ is perfectly divisible by ω_1, set the block size b_2 of node v to $(b_1 * \omega_2)/\omega_1$, and go to the next step. Otherwise, set l to $lcm(b_1 * \omega_2, \omega_1)$; set the block size of v to l/ω_1; for each node visited so far (from r to p), multiply its value of block size by $l/(b_1 * \omega_2)$.

It can be seen that the assignment(s) made to block size(s) in Step 2 (b) at the time of traversal of each edge ensure that the following condition is satisfied for every edge traversed so far: the block sizes of the two nodes connected by the edge have values that can be expressed as $k * \omega_1$ and $k * \omega_2$ respectively, where k is an integer, and ω_1 and ω_2 are the values of the coefficients recorded for that edge. Thus, the application of the above algorithm to each component of the graph BCG results in an assignment of block sizes that satisfies all of the block-size constraints. □

In general, the BCG for a program may have cycles that introduce conflicts between block-size constraints, as shown in Figure 8. These conflicts may be (i) direct, as shown by the presence of multiple edges between a given pair of nodes, or (ii) indirect, as shown by cycles involving more than two nodes (the presence of such cycles is a necessary, and not a sufficient condition for an indirect conflict). The first problem is dealt with by retaining only one edge between any pair, the one with the highest weight, and deleting all other edges between that pair. The second problem is tackled by finding for each connected component of BCG, a *maximum cost spanning tree*, i.e., one for which the sum of weights on the edges is maximum. The algorithm shown in the proof of Theorem 1 is now used to determine the block size of each node in the given connected component of the BCG. The block size of each dimension with a cyclic distribution, not appearing in the BCG, is set to 1.

3.4 Num-Procs Pass

Recording of Constraints and Quality Measures The compiler analyzes each assignment inside a loop, to an array element, or to a scalar involved in a reduction operation. Both the computational and the communication cost estimators are invoked, and the sum of those estimate terms gives the contribution of the statement to the expected program execution time. Since all other data partitioning parameters are known at this point, these terms are functions only of the number of processors in different mesh dimensions. The sum of contributing terms from all statements examined by the compiler yields the relevant part of the expected program execution time, that drives the selection of the number of processors.

Determination of Distribution Parameter In the preceding passes, each dimension of an array is assumed to be distributed on a distinct mesh dimension, and hence, potentially on more than one processor. We believe that for most scientific application programs, restricting the number of distributed dimensions of a single array to two does not lead to any loss of effective parallelism. In a comprehensive study of Fortran programs [19], Shen et al. reported finding only 8% of the array

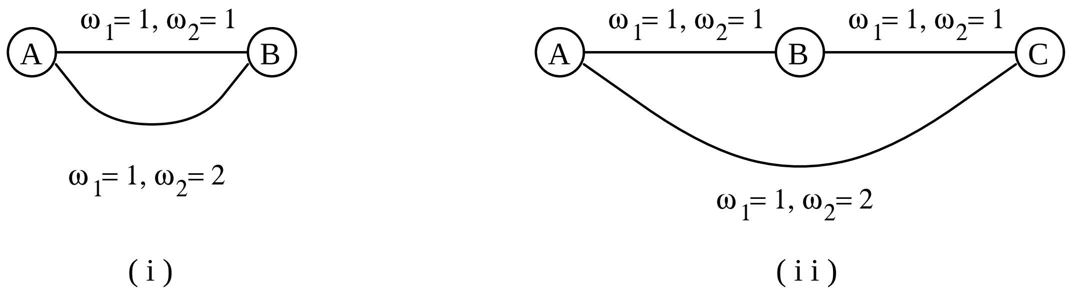

Figure 8: Conflicts between block-size constraints

references with more than two dimensions, and only 1% with more than three dimensions. Even when higher-dimensional arrays show parallelism in each dimension, restricting the number of distributed dimensions does not necessarily limit the extent to which parallelism can be exploited. Hence, in order to reduce the amount of search space, PARADIGM collapses all except for two (at most) mesh dimensions. This is done through the following heuristic steps:

1. For each set of aligned array dimensions, the compiler examines whether any dimension in that set "exhibits" parallelism (as discussed earlier) somewhere in the program. If none of the dimensions in the set exhibits parallelism, the corresponding mesh dimension is collapsed into a single processor.

2. Let D' be the number of dimensions not yet collapsed. If $D' > 2$, the compiler evaluates the expected execution times of the program for $C_2^{D'}$ cases ($C_2^{D'}$ is the number of ways of choosing 2 items from D' items), each case corresponding to exactly two variables representing the number of processors in different mesh dimensions set to $\sqrt{N}$, and the other $D' - 2$ variables set to 1. The case which yields the smallest value for expected execution time is chosen, and the corresponding $D' - 2$ dimensions are collapsed.

If at this point, there is only one dimension that has not been collapsed, the number of processors in that dimension is set to N, the total number of processors in the system. If there are two such dimensions, the only parameters left unknown in the partitioning scheme are N_1 and N_2, related by $N_1 * N_2 = N$. The compiler evaluates different mesh configurations corresponding to values of N_1 varying from 1 to N, and being doubled in each step (assuming N is a power of two). The expression for the expected execution time is evaluated for each configuration, and the one that leads to the smallest expected execution time is selected.

4 Experimental Results

The testbed used for the implementation and evaluation of our system is a 16-processor Intel iPSC/2 hypercube. Table 1 presents the data partitions obtained by PARADIGM for six Fortran programs of varying complexity. The source listing of each of these programs is given in [6]. The simplest program is `Jacobi`, a relaxation code that performs Jacobi iterations in a loop. `Tred2` is a routine from the Eispack library, that reduces a real symmetric matrix to a symmetric tridiagonal matrix. `Dgefa` (taken from Linpack library) factors a real matrix using gaussian elimination with partial pivoting. This program makes calls to other Linpack routines. Hence, the version we use is a transformed one where procedure in-lining has been done by hand. The remaining three programs are individual procedures taken from the Perfect Benchmarks [4]. `Olda` is the dominant procedure in the `trfd` program that simulates the computational aspects of a two-electron integral transformation. A profile of the sequential version showed the `trfd` program spending more than 98% of its time in the `olda` routine. The routines `dflux` and `eflux` are two of the three most important procedures (in terms of time spent) of the `flo52` program. `Flo52` is a two-dimensional code that analyzes the transonic inviscid flow past an airfoil by solving the unsteady Euler equations.

The final measure of success of any automatic data partitioning scheme is the performance of the resulting compiler-generated parallel program on the target multicomputer. However, most of the compilers that carry out the task of generating data-parallel programs, given the sequential program and data partitioning information, are still under development. Hence, in order to evaluate our results on data partitioning, we have manually developed (by simulating advanced compiler optimizations) different versions of parallel programs with message passing, corresponding to different data partitioning schemes.

In this paper, we describe results on the performance of different versions of two of the above programs on the iPSC/2. Further results and a more detailed evaluation of the data partitioning schemes automatically determined by PARADIGM are presented in [6].

JACOBI For the Jacobi program, PARADIGM selects column partitioning of the arrays A and B for smaller sizes, and a 2-D partitioning (where both rows and columns are distributed on the same number of processors) for larger data sizes. In each case, it chooses a blocked method of partitioning for all distributed array dimensions. The first two versions developed by us correspond to these partitioning schemes. The remaining two versions are based on variants of the above schemes, where the array dimensions are instead distributed in a cyclic manner.

The execution times obtained for each of those versions running on the 16-processor iPSC/2 are shown in Table 2. These results confirm that the 2-D partitioning starts performing better than the column partitioning for larger array sizes. The excessive communication re-

Program	*Arrays*	*Size*	*Distribution Functions*
tred2	A, Z	512 x 512	$(i-1) \bmod 16, 0$
	D, E	512	$(i-1) \bmod 16$
Jacobi	A, B	128 x 128	$0, \lfloor (j-1)/16 \rfloor$
		1024 x 1024	$\lfloor (i-1)/4 \rfloor, \lfloor (j-1)/4 \rfloor$
dgefa	A	128 x 128	$(i-1) \bmod 2, (j-1) \bmod 8$
		512 x 512	$(i-1) \bmod 4, (j-1) \bmod 4$
olda (trfd)	V	32 x 32	$0, (j-1) \bmod 16$
	$XRSIQ$	32 x 32	$(i-1) \bmod 16, 0$
	XIJ	32	$(i-1) \bmod 16$
	$XRSPQ$	278784	$(i-1) \bmod 16$
dflux (flo52)	FS, FW	193 x 33 x 4	$\lfloor (i-1)/16 \rfloor$, 0, 0
	W, DW	194 x 34 x 4	$\lfloor (i-1)/16 \rfloor$, 0, 0
	DP	195 x 35	$\lfloor (i-1)/16 \rfloor$, 0
	$RADJ, DTL, P, RADI, VOL$	194 x 34	$\lfloor (i-1)/16 \rfloor$, 0
	$EP, DIS4, DIS2$	193 x 33	$\lfloor (i-1)/16 \rfloor$, 0
eflux (flo52)	FS	193 x 33 x 4	$\lfloor (i-1)/16 \rfloor$, 0, 0
	W, DW	194 x 34 x 4	$\lfloor (i-1)/16 \rfloor$, 0, 0
	X	194 x 34 x 2	$\lfloor (i-1)/16 \rfloor$, 0, 0
	P	194 x 34	$\lfloor (i-1)/16 \rfloor$, 0

Table 1: Distribution functions obtained by PARADIGM

Data Size n	*Column Blocked Time (s)*	*2-D Blocked Time (s)*	*Column Cyclic Time (s)*	*2-D Cyclic Time (s)*
64	1.28	1.45	1.66	2.15
128	4.10	4.12	5.82	7.33
256	15.39	14.87	23.29	28.72
512	64.43	59.66	96.64	113.66
1024	257.84	243.24	386.94	

Table 2: Performance of different versions of Jacobi on iPSC/2

quirements resulting from the naive decision to partition the array dimensions in a cyclic manner are reflected in the poor performance of the last two versions. Those versions also require much more space to hold the non-local data received from other processors through collective communication. In fact, the program with 2-D cyclic partitioning could not be run for the data size $n = 1024$ due to memory limitations on each processor.

While these results confirm the compiler's prediction regarding the suitability of the 2-D partitioning at larger data sizes, the actual data size at which that scheme starts performing better than column partitioning is not predicted very accurately. We believe that this difference between the predicted value ($n = 1024$) and the observed value ($n = 256$) is due to the cost function used for the Shift operation being slightly inaccurate. The primary focus of our work in estimating communication costs has been to obtain the estimates in terms of cost functions of various communication primitives (which is performed satisfactorily in this case). Given more accurate performance characteristics of such primitives on the target machine, obtained by approaches proposed in the literature, such as the "training set" method [2], we believe our compiler would do an even better job of selecting good data partitioning schemes.

OLDA As shown in Table 1, PARADIGM distributes the array $XRSIQ$ by rows in a cyclic manner, V by columns in a cyclic manner, and the arrays XIJ and $XRSPQ$ also in a cyclic manner. The first parallel program version we developed corresponds to this partitioning scheme, referred to as *1-D cyclic*. The other version is based on a variant of the first scheme, where all the distributed array dimensions are given a blocked distribution instead. This method of partitioning is referred to as *1-D blocked*.

The performance of the two parallel program versions is shown in Figure 9. The method of partitioning chosen by PARADIGM, 1-D cyclic, leads to a significantly better performance than the other method. This confirms the desirability of the decision made by PARADIGM to distribute all dimensions in a cyclic manner. There are numerous other methods of data partitioning possible for the `olda` program, such as those in which both the dimensions of $XRSIQ$ and V are distributed on more than one processor, and those in which the alignment between the dimensions of $XRSIQ$ and V is reversed ($XRSIQ_1$ being aligned with V_1, and $XRSIQ_2$ with V_2). A manual inspection of the program shows that each of those schemes would lead to much higher communication overheads, and worse performance. For instance, if both the dimensions of $XRSIQ$ and V are

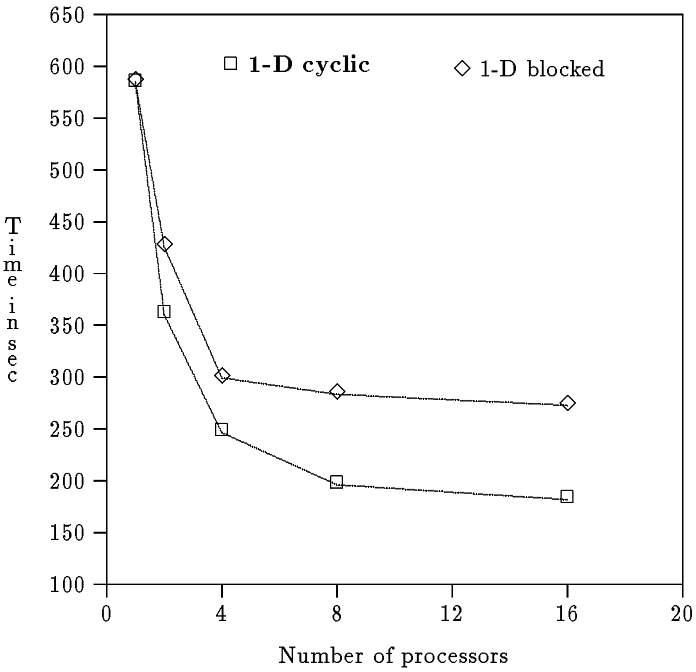

Figure 9: Performance of `olda` on Intel iPSC/2

distributed, many statements inside loops which execute under the 1-D scheme without any communication would now require a significant amount of communication. A reversal of alignment between the dimensions of $XRSIQ$ and V would lead to even greater communication costs due to expensive, transpose operations. These results show the success of PARADIGM in obtaining a good data partitioning scheme for the `olda` program too.

5 Related Work

Many researchers have worked on deriving data partitions for restricted classes of programs [17, 10, 20]. These approaches are mainly directed towards individual loops, and do not deal with conflicting requirements that might be imposed by different parts of the program on the data partitioning scheme.

Knobe, Lukas, and Steele have developed techniques for automatic data layout on SIMD machines [12]. They use the concept of *preferences* between data references to guide the layout process, which is similar in spirit to our use of constraints to guide the choice of data distribution parameters. A significant feature unique to our approach is the analysis carried out to record the quality measure with each constraint, which leads to a much more precise characterization of the "weight" to be attached to each constraint.

Balasundaram, Fox, Kennedy, and Kremer discuss an interactive tool that provides assistance to the user in determining the data distribution [1, 11]. They use the method of "training sets" to help estimate the performance of a program with message passing [2]. Those techniques need to be extended to allow performance estimation, given just the source program and data partitioning information, without actually translating it to one with explicit message passing.

Our research is quite closely related to the work of Li and Chen on the Crystal compiler [14, 13]. We have used the CAG framework that they introduced for the component alignment problem, and have extended it by developing a better method of assigning weights to the edges. However, for decisions after the alignment phase, Li and Chen propose comparing all possible data partitioning schemes [13]. While this may allow the performance estimation to be done in a more accurate manner, the cost of examining all possibilities becomes prohibitively high as the problem size is increased.

Chatterjee, Gilbert, Schreiber, and Teng [3] describe a framework for automatic alignment of arrays in data-parallel languages. In the context of the distribution functions used by us, that corresponds to determining the alignment of array dimensions, the offsets, and the relationship between block sizes of various array dimensions. They do not deal with the problem of determining the method of partitioning and the number of processors in different mesh dimensions.

Wholey describes an automatic data mapping system for the ALEXI language [21]. The problem of performance estimation is simpler compared to that for Fortran 77 programs, since the ALEXI programs already have the calls to primitives to carry out high-level communication and parallel array operations.

6 Conclusions

We have described the design of a compiler called PARADIGM, that makes data partitioning decisions on Fortran 77 programs. Our main focus in this paper has been on presenting the algorithms used in different passes of PARADIGM, that determine various distribution parameters of the arrays. We have shown the effectiveness of our compiler on regular, numeric computations through results on real-life Fortran codes.

We are currently exploring several directions in which this work can be extended. An important problem being addressed is the handling of procedure calls. We are developing techniques that will enable the compiler to determine constraints and their quality measures across procedure boundaries. We also have some preliminary ideas on how the compiler could make decisions on repartitioning of data in different parts of the program.

In the future, we also plan to examine other strategies for searching through the space of possible data partitions. Currently, PARADIGM does not backtrack on any decision made in a previous pass, even though the resolution of conflicts among constraints in the earlier passes is based on less precise cost estimates, due to many data partitioning parameters being unknown. So far, our results do not indicate a need for backtracking. However, further research is needed to determine the best search strategy.

References

[1] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proc. Fifth Distributed Memory Computing Conference*, April 1990.

[2] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proc. Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Williamsburg, VA, April 1991.

[3] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Automatic array alignment in data-parallel programs. In *Proc. Twentieth Annual ACM Symposium on Principles of Programming Languages*, Charleston, SC, January 1993.

[4] The Perfect Club. The perfect club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, 3(3):5–40, Fall 1989.

[5] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University, January 1993.

[6] M. Gupta. *Automatic data partitioning on distributed memory multicomputers.* PhD thesis, University of Illinois, September 1992.

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

[8] M. Gupta and P. Banerjee. A methodology for high-level synthesis of communication on multicomputers. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

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

[10] D. E. Hudak and S. G. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proc. 1990 International Conference on Supercomputing*, pages 187–200, Amsterdam, The Netherlands, June 1990.

[11] K. Kennedy and U. Kremer. Automatic data alignment and distribution for loosely synchronous problems in an interactive programming environment. Technical Report TR91-155, Rice University, April 1991.

[12] 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*, 8:102–118, 1990.

[13] J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Proc. Supercomputing '90*, New York, NY, November 1990.

[14] 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.

[15] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors. In *Proc. 1989 International Conference on Parallel Processing*, August 1989.

[16] M.J. Quinn and P. J. Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, 7:69–76, September 1990.

[17] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–481, October 1991.

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

[19] Z. Shen, Z. Li, and P.-C. Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[20] D. G. Socha. An approach to compiling single-point iterative programs for distributed memory computers. In *Proc. Fifth Distributed Memory Computing Conference*, Charleston, S. Carolina, April 1990.

[21] S. Wholey. Automatic data mapping for distributed-memory parallel computers. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

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