April 1995

UILU-ENG-95-2210
CRHC-95-09

Center for Reliable and High Performance Computing

Automatic Selection of Dynamic Data Partitioning Schemes for Distributed Memory Multicomputers

Daniel J. Palermo & Prithviraj Banerjee

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Automatic Selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers

Daniel J. Palermo and Prithviraj Banerjee

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main Street
Urbana, IL 61801, U.S.A.
Tel: (217) 333-6564
Fax: (217) 333-1910
{palermo, banerjee}@crhc.uiuc.edu

Abstract

For distributed memory multicomputers such as the Intel Paragon, the IBM SP-2, the NCUBE/2, and the Thinking Machines CM-5, the quality of the data partitioning for a given application is crucial to obtaining high performance. This task has traditionally been the user's responsibility, but in recent years much effort has been directed to automating the selection of data partitioning schemes. Several researchers have proposed systems that are able to produce data distributions that remain in effect for the entire execution of an application. For complex programs, however, such static data distributions may be insufficient to obtain acceptable performance. The selection of distributions that dynamically change over the course of a program's execution adds another dimension to the data partitioning problem. In this paper, we present a technique that can be used to automatically determine which partitionings are most beneficial over specific sections of a program while taking into account the added overhead of performing redistribution. This system is being built as part of the PARADIGM (PARAllelizing compiler for DIstributed memory General-purpose Multicomputers) project at the University of Illinois. The complete system will provide a fully automated means to parallelize programs written in a serial programming model obtaining high performance on a wide range of distributed-memory multicomputers.

Keywords: distributed memory, multicomputers, data partitioning, dynamic redistribution, parallelizing compilers.

This research was supported in part by the National Aeronautics and Space Administration under Contract NASA NAG 1-613 and in part by the Advanced Research Projects Agency under contract DAA-H04-94-G-0273 administered by the Army Research office. We are also grateful to the National Center for Supercomputing Applications and the San Diego Supercomputing Center for providing access to their machines.

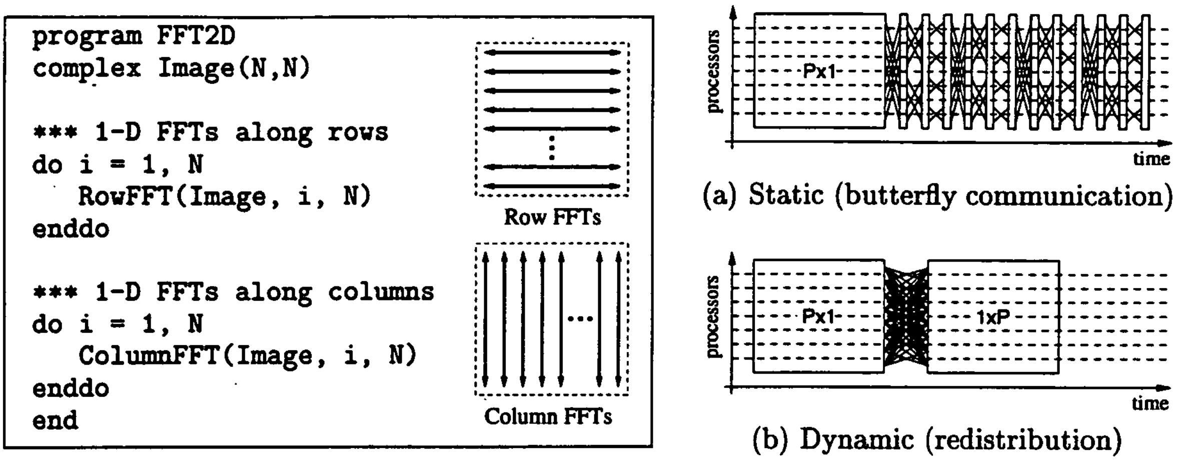
1 Introduction

Distributed-memory multicomputers such as the Intel Paragon, the IBM SP-2, the NCUBE/2, and the Thinking Machines CM-5 all offer significant advantages over shared-memory multiprocessors in terms of cost and scalability. However, lacking a global address space, they present a very difficult programming model in which the user must specify how data and computations are to be partitioned across processors and determine which sections of data need to be communicated among which processors. To overcome this difficulty, significant research effort has been aimed at source-to-source parallel compilers for multicomputers that relieve the programmer from the task of communication generation, while the task of data partitioning remains a responsibility of the programmer.

These compilers take a program written in a sequential or shared-memory parallel language and, based on user-specified partitioning of the data, generate code for a given multicomputer. Examples include Fortran D [13], Fortran 90D [5], the SUIF compiler [1], and the SUPERB compiler [7]. Many of these research efforts are also now looking into automated data partitioning techniques. Researchers in this area are also currently involved in defining High Performance Fortran (HPF) [17] to standardize parallel programming with data distribution directives.

As part of the PARADIGM (PARAllelizing compiler for DIstributed memory General-purpose Multicomputers) project [3] at the University of Illinois, automatic data partitioning techniques have been developed to relieve the programmer of the burden of selecting a good data distribution. The compiler can currently select the best static distribution of data using a constraint-based algorithm [12] which determines both the best configuration of an abstract multi-dimensional mesh topology along with how program data should be distributed on the mesh. In this paper, we present a technique that will be used to extend the static partitioning algorithm to select dynamic data distributions which can further improve the performance of the resulting parallel program.

The remainder of this paper is organized as follows: Section 2 presents a small example to illustrate the need for dynamic array redistribution; related work in automatic selection of static and dynamic data distribution schemes is discussed in Section 3; the methodology for selection of dynamic data distributions is presented in Section 4; code generation issues are described in Section 5; an experimental analysis of the presented techniques is performed in Section 6; and conclusions are presented in Section 7.



```
program FFT2D
complex Image(N,N)

*** 1-D FFTs along rows
do i = 1, N
   RowFFT(Image, i, N)
enddo

*** 1-D FFTs along columns
do i = 1, N
   ColumnFFT(Image, i, N)
enddo
end
```

(a) Static (butterfly communication)

(b) Dynamic (redistribution)

Figure 1: 2-D Fast Fourier Transform

2 Motivation

Figure 1 shows the basic computation performed in a two dimensional Fast Fourier Transform (FFT). To execute this program in parallel on a machine with distributed memory, the main data array, **Image**, is partitioned across the available processors. By examining the data accesses that will occur during execution it can be seen that, for the first half of the program, data is manipulated along the rows of the array. For the rest of the execution, data is manipulated along the columns. Depending on how data is distributed among the processors, several different patterns of communication could be generated. The goal of automatic data partitioning is to select the distribution which will result in the highest level of performance.

If the array were distributed by rows, every processor could independently compute the FFTs for each row that involved local data. After the rows had been processed, the processors would now have to communicate to perform the column FFTs since the columns have been partitioned across the processors. Conversely, if a column distribution were selected, communication would be required to compute the row FFTs while the column FFTs could be computed independently. Such static partitionings, as shown in Figure 1a, suffer in that they cannot reflect changes in a program's data access behavior. When conflicting data requirements are present, static partitionings tend to be compromises between a number of preferred distributions.

Instead of requiring a single data distribution for the entire execution, program data could also be redistributed dynamically for different *phases*[1] of the program. For this example, assume the

[1] A *phase* can be described simply as a sequence of statements in a program over which a given distribution is unchanged.

program is split into two separate phases; a row distribution is selected for the first phase and a column distribution for the second (as shown in Figure 1b). By redistributing the data between the two phases, none of the one-dimensional FFT operations would require communication. Such dynamic partitionings can yield higher performance than a static partitioning when the redistribution is more efficient than the communication pattern required by the statically partitioned computation.

3 Related Work

3.1 Static Partitioning

Some of the ideas used in the static partitioning algorithm currently implemented in the PARADIGM compiler [12] were inspired by earlier work on multi-dimensional array alignment [19]. In addition to this work, in recent years much research has been performed in the area of multidimensional array alignment [8, 16, 19] as well as in a number of other areas which address various aspects of data partitioning. Others have approached the partitioning problem as follows: examining cases in which a communication-free partitioning exists [22]; showing how performance estimation is a key in selecting good data distributions [9, 26]; linearizing array accesses and analyzing the resulting one-dimensional accesses [24]; applying iterative techniques which minimize the amount of communication at each step [2]; and examining issues for special-purpose distributed architectures such as systolic arrays [25].

3.2 Dynamic Partitioning

Anderson and Lam [2] have also proven that the dynamic decomposition problem is NP-hard by transforming the colored multiway cut problem (which is known to be NP-hard) into a subproblem of dynamic decomposition. They address the dynamic distribution problem by using a communication graph in which nodes are loop nests and edges represent the time for communication if the two loop nests involved in an edge are given different distributions. A greedy heuristic is used to combine nodes in such a way that the largest potential communication costs are eliminated first while maintaining sufficient parallelism.

Work by Bixby, Kennedy and Kremer formulates the data partitioning problem in the form of a 0-1 integer programming problem [4]. For each phase, a number of candidate partial data layouts are enumerated along with the estimated execution costs. The costs of the possible transitions among the candidate layouts are then computed forming a data layout graph. A static performance estimator is used to obtain the node and edge weights of the graph. Since each phase only specifies a partial data distribution, redistribution constraints can cross over multiple phases, thereby requiring the use of 0-1 integer programming. The number of possible data layouts for a given phase is exponential in the number of partitioned array dimensions resulting in potentially large search spaces. In order to benefit from advanced techniques in the field of integer programming, the resulting formulation is processed by a commercial integer programming tool. In some previous work [18], they also examined a dynamic programming technique that could determine dynamic distributions in polynomial time if each candidate layout specified a mapping for every array in the program. The main drawback with both of these techniques is that the selection of phases must be made a priori to the formulation of the problem. This means that the size of the phases should be as small as possible so that the correct solution is found and that as many candidate layouts as possible should be specified in order to find the best dynamic data distribution. These factors contribute to the increase in the size of the search space.

Bixby, Kremer, and Kennedy have also described an *operational definition* of a phase which defines a phase as the outermost loop of a loop nest such that the corresponding iteration variable is used in a subscript expression of an array reference in the loop body [4]. Even though this definition restricts phase boundaries to loop structures and does not allow for overlapping or nesting of phases, it can be seen that for the example in Section 2 this definition is sufficient to describe the two distinct phases of the computation.

Chapman, Fahringer, and Zima have noted that there are many known good data distributions for important numerical problems that are frequently used [6]. They describe the design of a distribution tool that makes use of performance prediction methods when possible, but also heuristically uses empirical performance data when available. They have also developed cost estimates which model the work distribution, communication and data locality of a program while taking into account the features of the target architecture [9]. The combination of estimation techniques along with profile information will be used to guide their system in potentially selecting lists of distributions for different arrays that appear in the program. Currently, the performance prediction system has been integrated into a compiler based on Vienna Fortran [7], and research is under way on their partitioning system.

Hudak and Abraham have also proposed a method for selecting redistribution points for programs executed on machines with logically shared but physically distributed memory [14, 15]. Their technique is based on locating significant control flow changes in the program and inserting remapping points [15]. Remapping points are inserted between loop nests with different nesting levels, around control loops, and between a loop that requires nearest neighbor communication (which is assigned a block partitioning) and a loop that is linearly varying (which is given a cyclic partitioning). Once remapping points have been added to the program structure, a merge phase is performed to remove any unnecessary data redistribution. Remapping points are removed between two phases with identical distributions as well as when one phase has an *unspecified* distribution (in which case it is assigned a distribution from an adjacent phase). This heuristic was shown to improve the performance of several programs as compared to a strict data-parallel implementation. For a program consisting of a matrix multiplication followed by an LU factorization, they observed a 26% improvement by applying this technique.

4 Dynamic Distribution Selection

The technique we propose to automatically select redistribution points can be broken down into two main steps. First, the program is recursively decomposed into a hierarchy of candidate phases. Then, taking into account the cost of redistributing the data between the different phases, the most efficient sequence of phases and phase transitions is selected.

This approach allows us to build upon the static partitioning techniques [12] previously developed in the PARADIGM project. Static cost estimation techniques [11] are used to guide the selection of phases while static partitioning techniques are used to determine the best possible distribution for each phase. The cost models used to estimate communication and computation costs use parameters, empirically measured for each target machine, to separate the partitioning algorithm from a specific architecture.

To help illustrate the dynamic partitioning technique, an example program will be used. In Figure 2, a two-dimensional Alternating Direction Implicit (ADI) iterative method[2] is shown which computes the solution of an elliptic partial differential equation known as Poisson's equation [10]. Poisson's equation can be used to describe the dissipation of heat away from a surface with a fixed temperature as well as to compute the free-space potential created by a surface with an electrical

[2] To simplify later comparison and analysis of performance measurements, the program shown performs an arbitrary number of iterations as opposed to periodically checking for convergence of the solution.

```
program ADI2d
double precision u(N,N), uh(N,N), b(N,N), alpha
integer i, j, k

*** Initial value for u                               Phase
do j = 1, N                                            1  ┐
  do i = 1, N                                          2  │
    u(i,j) = 0.0                                       3  │
  enddo                                                4  │ I
  u(1,j) = 30.0                                        5  │
  u(n,j) = 30.0                                        6  │
enddo                                                  7  ┘

*** Initialize uh
do j = 1, N                                            8  ┐
  do i = 1, N                                          9  │
    uh(i,j) = u(i,j)                                  10  │ II
  enddo                                               11  │
enddo                                                 12  ┘

alpha = 4 * (2.0 / N)                                 13
do k = 1, maxiter                                     14
  *** Forward and backward sweeps along columns
  do j = 2, N - 1                                     15  ┐
    do i = 2, N - 1                                   16  │
      b(i,j) = (2 + alpha)                            17  │
      uh(i,j) = (alpha - 2) * u(i,j) +                    │ III
&                          u(i,j + 1) + u(i,j - 1)    18  │
    enddo                                             19  │
  enddo                                               20  ┘
  do j = 2, N - 1                                     21  ┐
    uh(2,j) = uh(2,j) + u(1,j)                        22  │ IV
    uh(N - 1,j) = uh(N - 1,j) + u(N,j)                23  │
  enddo                                               24  ┘

  do j = 2, N - 1                                     25  ┐
    do i = 3, N - 1                                   26  │
      b(i,j) = b(i,j) - 1 / b(i - 1,j)                27  │ V
      uh(i,j) = uh(i,j) + uh(i - 1,j) / b(i - 1,j)    28  │
    enddo                                             29  │
  enddo                                               30  ┘
```

```
                                                      Phase
  do j = 2, N - 1                                     31  ┐
    uh(N - 1,j) = uh(N - 1,j) / b(N - 1,j)            32  │ VI
  enddo                                               33  │
  do j = 2, N - 1                                     34  ┤
    do i = N - 2, 2, -1                               35  │
      uh(i,j) = (uh(i,j) + uh(i + 1,j)) / b(i,j)      36  │ VII
    enddo                                             37  │
  enddo                                               38  ┘

  *** Forward and backward sweeps along rows
  do j = 2, N - 1                                     39  ┐
    do i = 2, N - 1                                   40  │
      b(i,j) = (2 + alpha)                            41  │
      u(i,j) = (alpha - 2) * uh(i,j) +                    │ VIII
&                    uh(i + 1,j) + uh(i - 1,j)        42  │
    enddo                                             43  │
  enddo                                               44  │
  do i = 2, N - 1                                     45  ┤
    u(i,2) = u(i,2) + uh(i,1)                         46  │
    u(i,N - 1) = u(i,N - 1) + uh(i,N)                 47  │ IX
  enddo                                               48  ┘

  do j = 3, N - 1                                     49  ┐
    do i = 2, N - 1                                   50  │
      b(i,j) = b(i,j) - 1 / b(i,j - 1)                51  │ X
      u(i,j) = u(i,j) + u(i,j - 1) / b(i,j - 1)       52  │
    enddo                                             53  │
  enddo                                               54  ┤
  do i = 2, N - 1                                     55  │
    u(i,N - 1) = u(i,N - 1) / b(i,N - 1)              56  │ XI
  enddo                                               57  │
  do j = N - 2, 2, -1                                 58  ┤
    do i = 2, N - 1                                   59  │
      u(i,j) = (u(i,j) + u(i,j + 1)) / b(i,j)         60  │ XII
    enddo                                             61  │
  enddo                                               62  ┘
enddo                                                 63
end                                                   64
```

Figure 2: 2-D Alternating Direction Implicit method (with operational phases shown)

charge.

For the program in Figure 2, a static data distribution will result in a significant amount of communication for over half of the program's execution. For illustrative purposes only, the operational definition of phases previously described in Section 3 identifies twelve different "phases" in the program. These phases exposed by the operational definition need not be known for our technique (and, in general, are potentially too restrictive) but they will be used here for comparison as well as to facilitate the discussion.

4.1 Phase Decomposition

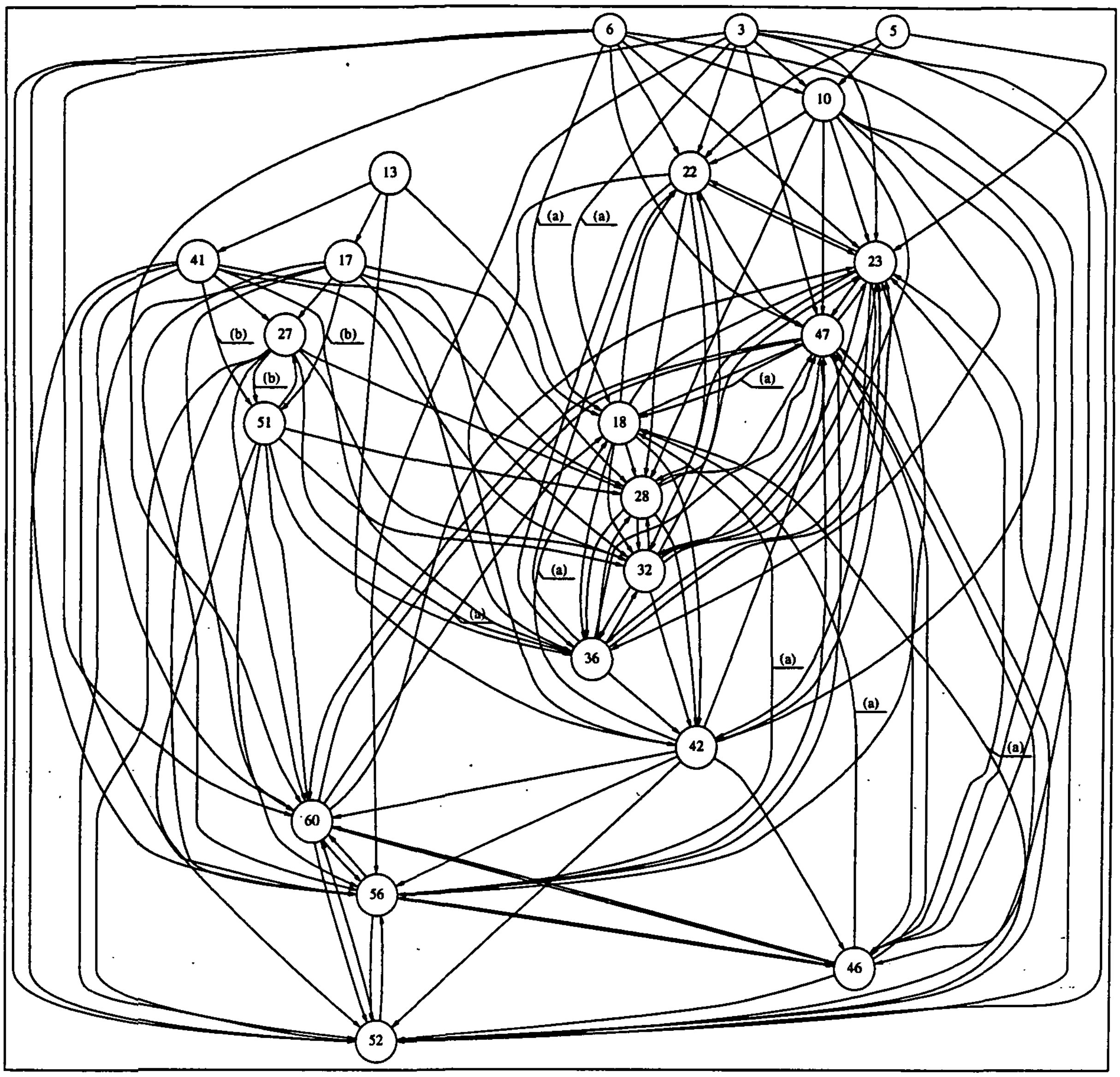
Initially, the entire program is viewed as a single phase for which a static distribution is determined. At this point, the immediate goal is to determine if and where it would be beneficial to split the program into two separate phases. Using the selected distribution, a *communication graph* is constructed to examine the cost of communication in relation to the flow of data within the program.

We define a *communication graph* as the flow information from the dependence graph (as generated by Parafrase-2 [21] which PARADIGM is built upon) weighted by the cost of communication. The nodes of the communication graph correspond to individual statements while the edges correspond to flow dependencies that exist between the statements. As a heuristic, the cost of communication performed for a given reference in a statement is reflected back along every incoming dependence edge corresponding to the reference involved[3]. Since flow information is used to construct the communication graph, the weights on the edges serve to expose communication costs that exist between producer/consumer relationships within a program. The granularity of phase partitioning is also restricted to the statement level, therefore, single node cycles in the flow dependence graph are not included in the communication graph.

In addition to the costs of communication generated by a statement, we also introduce the idea of *transparent* statements. These are statements for which the target of the assignment: (1) is also referenced in the assignment function with identical indexing, and (2) is not referenced again with a different indexing function. If any communication cost is reflected back to a transparent statement, it is also further reflected on any incoming dependence edges originating from statements prior to the current position. This has a net effect of encouraging redistribution as early as possible in the program text by allowing selected costs to be propagated toward the start of the program. For the ADI program, statements 22, 23, 46, and 47 can be considered transparent.

In Figure 3, the communication graph is shown for ADI with some of the costs on the edges labeled with the expressions automatically generated by the static cost estimator (using a problem size of 512 × 512 and `maxiter` set to 100). For reference, the functions for an Intel Paragon and a Thinking Machines CM-5, corresponding to the communication primitives used in the edge weights, are shown in Table 1.

[3] There is at most only one edge between two nodes for each array referenced in the statement. For multiple references using the same array, the edge weight is the sum of all communication for that array.

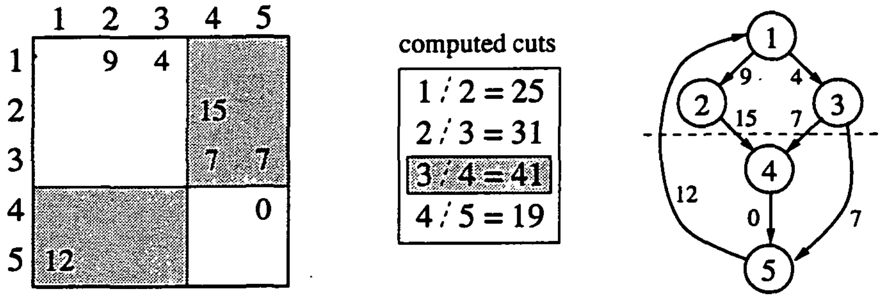


(a) $100.000000 * (P_2 > 1) * \text{Shift}(510.000000)$
(b) $3100.000000 * \text{Transfer}(510.000000)$

Figure 3: Communication graph and example edge costs for ADI
(Statement numbers correspond to Figure 2)

	Intel Paragon	TMC CM-5
Transfer(m)	$95 + 0.038m$	$23 + 0.12m \quad m \leq 16$ $86 + 0.12m \quad m > 16$
Shift(m)	$2 * \text{Transfer}(m)$	

Table 1: Communication primitives



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & & 9 & 4 & & \\ 2 & & & & 15 & \\ 3 & & & & 7 & 7 \\ 4 & & & & & 0 \\ 5 & 12 & & & & \end{array}$$

computed cuts

1 / 2 = 25

2 / 3 = 31

3 / 4 = 41

4 / 5 = 19

(a) Adjacency matrix

(b) Actual representation

Figure 4: Example graph illustrating the computation of a cut

Once the communication graph has been constructed, a split point is determined by computing a maximal cut of the communication graph. The maximal cut removes the largest communication constraints from a given phase to potentially allow better individual distributions to be selected for the two resulting split phases. Since the communication graph can potentially contain edges with a zero communication cost, it is also possible to find several cuts which all have the same cost. The following algorithm is used to determine which cut to use to split a given phase:

1. To better describe the algorithm, view the communication graph $G = (V, E)$ in the form of an adjacency matrix (with source vertices on rows and destination vertices on columns).

2. For each statement S_i $\{i \in [1, (|V| - 1)]\}$ compute the cut of the graph between statements S_i and S_{i+1} by summing all the edges in the sub-matrices specified by $[S_1, S_i] \times [S_{i+1}, S_{|V|}]$ and $[S_{i+1}, S_{|V|}] \times [S_1, S_i]$ (an efficient implementation, which only adds and subtracts the differences between two successive cuts, takes $\mathcal{O}(E)$ time on the actual representation)

3. While computing the cost of each cut also keep track of the current maximum.

4. The maximum cut is used to select the phase split; if there is more than one cut with the same maximum value, choose the first.

5. Mark the arrays involved in the cut edges to redistribute and split the phase using the selected cut.

In Figure 4, the computation of the maximal cut on a smaller example graph with arbitrary weights is shown. The maximal cut is found to be between vertices 3 and 4 with a cost of 41. This is shown both in the form of the sum of the two adjacency submatrices, specified by the algorithm,

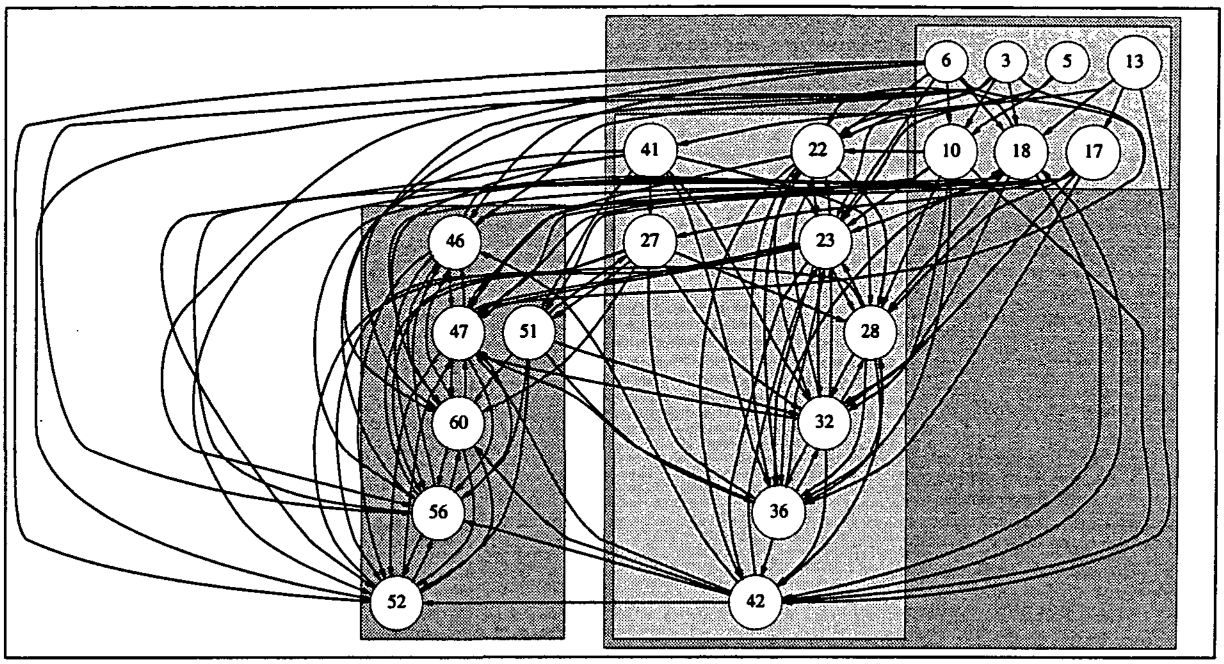


Figure 5: Partitioned communication graph for ADI
(Statement numbers correspond to Figure 2)

and graphically as a cut on the actual representation. Since the ordering of the nodes is related to the linear ordering of statements in a program, the algorithm also guarantees that the nodes on one side of the cut will always all precede or all follow the node most closely involved in the cut. This is necessary to ensure that the cut divides the program at exactly one point.

It is interesting to note (but will not be examined here) that in some cases a cut can actually require performing loop distribution (when allowed). Also if dependencies allow statements to be reordered, statements may be able to move across a cut boundary without affecting the cost of the cut or possibly even reduce the amount of data to be redistributed. Neither of these optimizations will be examined in this paper.

A new distribution is selected for each of the resulting phases and the process is continued recursively. Each level of the recursion is carried out in branch and bound fashion such that a phase is split only if the sum of the estimated execution times of the two resulting phases shows an improvement over the original[4]. In Figure 5, the partitioned communication graph is shown for ADI after the phase decomposition is completed.

To be able to bound the depth of the recursion without ignoring important phases and distri-

[4] A further optimization can also be applied to bound the size of the smallest phase that can be split by requiring its estimated execution time to be greater than a "minimum cost" of redistribution.

Op. Phases(s)	Distribution	Intel Paragon	TMC CM-5	
I-XII	*,block 1 × 32	22.239688	35.970293	Level 1
I-VIII	*,block 1 × 32	1.454816	2.347115	Level 2
IX-XII	block,* 32 × 1	0.603424	0.942850	
I-III	block,* 32 × 1	0.376035	0.590306	Level 3
IV-VIII	*,block 1 × 32	0.978784	1.529350	

Table 2: Detected phases and estimated execution times (sec) for ADI

butions, the static partitioner must also obey the following property. A partitioning technique is said to be *monotonic* if it selects the best available partition for a segment of code such that (aside from the cost of redistribution) the time to execute a code segment with a selected distribution is less than or equal to the time to execute the same segment with a distribution that is selected after another code segment is appended to the first. In practice, this condition is satisfied by the static partitioning algorithm that we are using. This can be attributed to the fact that conflicts between distribution preferences are not broken arbitrarily, but are resolved based on the costs imposed by the target architecture [12].

4.2 Phase and Phase Transition Selection

After the program has been recursively decomposed by the phase selection and initial partition selection, redistribution costs are estimated for the possible phase transitions [23]. Edges with the resulting costs are used to connect the phases in a *phase transition graph* (as in Figure 6) to determine which phases and transitions are necessary to obtain the best performance.

The distributions and estimated execution times reported by the static partitioner for the decomposed phases (described as ranges of operational phases) are shown in Table 2. The performance parameters of the two machines are similar enough that the static partitioning actually selects the same distribution at each phase for each machine. The times estimated for the static partition are a bit higher than those actually observed, resulting from a conservative assumption made by the cost estimator, but they still exhibit similar enough performance trends to be used as estimates.

Since it is possible that using lower level phases may require transitioning through distributions found at higher levels to keep the overall redistribution costs to a minimum, redistribution stages are allowed at the granularity of the lowest level of the phase decomposition. To take into account redistribution induced by iteration in the program, any phase which is split within a loop body will have its redistribution costs doubled to account for a potential reverse redistribution. Once this is completed, the shortest path in the phase transition graph is computed to select the appropriate phases and transitions.

If a path is selected which has a change in distribution at some point after immediately entering a loop, the entry distribution is compared to the exit distribution of the loop body. If they are different, but there is a entry phase which does have the same distribution as the exit, the redistribution costs in the loop are reduced by half and a new shortest sub-path is computed from the matching entry point to the exit. If the cost of the newly computed sub-path is close enough to that originally selected, it is used in place of the original. Since the redistribution costs were already conservative estimates (by a factor of two) to represent possible redistribution induced by the iteration in these types of loops, this substitution is legal. The number of valid paths through the program with different active distributions can be greatly reduced by ensuring that the entry and exit phase distributions match (for loops without redistribution upon entry) thereby simplifying code generation (as will be discussed in Section 5).

On an Intel Paragon the cost of performing redistribution is low enough that a dynamic distribution scheme is selected (shown by the shaded area in Figure 6). The phase matching procedure described above can also be seen in Figure 6 as the rightmost path is selected instead of the center path to maintain matching entry and exit distributions. For a Thinking Machines CM-5, however, the cost of redistribution is more expensive than the gains that can be made using a dynamic distribution; therefore, a static distribution is selected for this machine. To briefly recap the entire procedure, pseudo-code for the dynamic partitioning algorithm is presented in Figure 7.

Since the selection of the split point during decomposition implicitly maintains the coupling between individual array distributions, redistribution at any stage will only affect the next stage. This can be contrasted to the technique proposed by Bixby, Kremer, and Kennedy [4] which first selects a number of partial candidate distributions for each phase specified by the operational definition. Since their phase boundaries are chosen in the absence of flow information, redistribution can affect stages at any distance from the current stage. This causes the redistribution costs to become binary functions depending on whether or not a specific path is taken, therefore, necessitating the need for 0-1 integer programming. If distributions are exhaustively enumerated for every operational phase, the integer programming technique will obtain an optimal solution. Since the choice of candidate distributions can be considered somewhat of a heuristic in itself, it would be of interest to compare the quality and performance of these two techniques as more results are obtained.

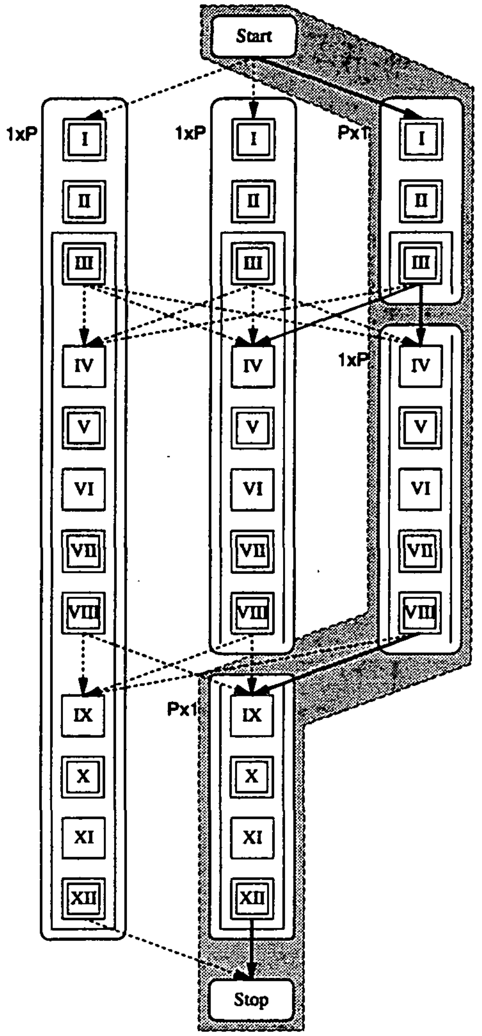


Figure 6: Selected phases for ADI

```
Construct the communication graph for the program
Perform an initial static partitioning on the program
phases = Decompose_Phase(program)
scheme = Select_Redistribution(phases)

Decompose_Phase(phase)
    Add phase to list of recognized phases
    Assign new costs to the communication graph
    Compute the maximal cut
                    $phase \rightarrow phase_1, phase_2$
    Perform static partitioning on $phase_1$
    Perform static partitioning on $phase_2$
    Mark the arrays to be redistributed
    if $(\text{cost}(phase_1) + \text{cost}(phase_2)) < \text{cost}(phase)$
        phase→left   = Decompose_Phase($phase_1$)
        phase→right  = Decompose_Phase($phase_2$)
    else
        phase→left   = null
        phase→right  = null
    return(phase)

Select_Redistribution(phases)
    Construct the phase transistion graph
    Estimate the interphase redistribution costs
    Compute the shortest phase transition path
    return(selected phase transition path)
```

Figure 7: Pseudo-code for the partitioning algorithm

5 Code Generation Issues

In order to utilize the available memory on a given parallel machine as efficiently as possible, only the distributions which are active at any given point in the program should actually be allocated space. It is interesting to note that as long as a given array is distributed among the same total number of processors, the actual space required to store one section of the partitioned array is the same no matter how many array dimensions are distributed[5]. By using this observation, it is possible to statically allocate the minimum amount of memory required by simply renaming array references for each distribution used and declaring the newly generated names to all be "equivalent" for a given array. The `equivalence` statement in Fortran allows this to be performed at the source level as long as redistribution operations are guaranteed to read all source data before writing to the target. Compared to C, this is somewhat similar to assigning two array base pointers to point at the same memory location. The use of the `equivalence` statement also avoids the need to linearize array accesses at this level since the target Fortran compiler will generate the appropriate indexing functions for multidimensional accesses.

This also implies that the send and receive operations used to implement the communication are buffering the data. In the worst case, an entire copy of a partitioned array can be buffered by such "synchronous" communication before it is received and moved into the destination array. As soon as more than two different distributions are present for a given array, the `equivalence` begins to pay off, even in the worst case, in terms of the amount of memory overhead that is eliminated. If the performance of buffered synchronous communication is insufficient for a given machine, non-buffered asynchronous communication could be used instead, thereby precluding the use of `equivalence` (unless explicit buffering is performed by the redistribution operation itself).

In Figure 8, the ADI program is shown with the three phases selected in Figure 6 along with the use of renaming and `equivalence` statments to separate the distributions of the different phases. Assignments shown in R1 and R2 indicate the redistribution to be performed between the phases. This approach also fits nicely with the compilation techniques being developed for static data distributions as the renaming describes the code specialization that is required. The guarantee that only one array in an equivalence set will be in use at any given time also allows the code to

[5] Taking into account distributions in which the number of processors allocated to a given array dimension does not evenly divide the size of the dimension, it can be equivalently said that there is a given amount of memory which can store all possible distributions with very little excess.

```
program ADI2d
double precision u_1(N,N), uh_1(N,N), b_1(N,N), alpha
double precision u_2(N,N), uh_2(N,N), b_2(N,N)
distribute (block, *) :: u_1, uh_1, b_1
distribute (*, block) :: u_2, uh_2, b_2
equivalence u_1, u_2
equivalence uh_1, uh_2
equivalence b_1, b_2
integer i, j, k

                                                          Phase
*** Initial value for u
do j = 1, N                                            1  ┐
  do i = 1, N                                          2  │
    u_1(i,j) = 0.0                                     3  │
  enddo                                                4  │
  u_1(1,j) = 30.0                                      5  │
  u_1(n,j) = 30.0                                      6  │
enddo                                                  7  │
                                                          │
*** Initialize uh                                         │
do j = 1, N                                            8  │
  do i = 1, N                                          9  │
    uh_1(i,j) = u_1(i,j)                              10  │
  enddo                                               11  │ A
enddo                                                 12  │
                                                          │
alpha = 4 * (2.0 / N)                                 13  │
do k = 1, maxiter                                     14  │
  *** Forward and backward sweeps along columns           │
  do j = 2, N - 1                                     15  │
    do i = 2, N - 1                                   16  │
      b_1(i,j) = (2 + alpha)                          17  │
      uh_1(i,j) = (alpha - 2) * u_1(i,j) +                │
&                        u_1(i,j + 1) + u_1(i,j - 1)  18  │
    enddo                                             19  │
  enddo                                               20  ┘

  *** Redistribute data
  u_2  = u_1                                              ┐
  uh_2 = uh_1                                             │ R1
  b_2  = b_1                                              ┘

  do j = 2, N - 1                                     21  ┐
    uh_2(2,j) = uh_2(2,j) + u_2(1,j)                  22  │
    uh_2(N - 1,j) = uh_2(N - 1,j) + u_2(N,j)          23  │
  enddo                                               24  │
                                                          │
  do j = 2, N - 1                                     25  │
    do i = 3, N - 1                                   26  │ B
      b_2(i,j) = b_2(i,j) - 1 / b_2(i - 1,j)          27  │
      uh_2(i,j) = uh_2(i,j) +                             │
&                      uh_2(i - 1,j) / b_2(i - 1,j)   28  │
    enddo                                             29  │
  enddo                                               30  │

                                                           Phase
  do j = 2, N - 1                                     31  │
    uh_2(N - 1,j) = uh_2(N - 1,j) / b_2(N - 1,j)      32  │
  enddo                                               33  │
  do j = 2, N - 1                                     34  │
    do i = N - 2, 2, -1                               35  │
      uh_2(i,j) = (uh_2(i,j) +                            │
&                      uh_2(i + 1,j)) / b_2(i,j)      36  │
    enddo                                             37  │ B
  enddo                                               38  │
                                                          │
  *** Forward and backward sweeps along rows              │
  do j = 2, N - 1                                     39  │
    do i = 2, N - 1                                   40  │
      b_2(i,j) = (2 + alpha)                          41  │
      u_2(i,j) = (alpha - 2) * uh_2(i,j) +                │
&                  uh_2(i + 1,j) + uh_2(i - 1,j)      42  │
    enddo                                             43  │
  enddo                                               44  ┘

  *** Redistribute data
  u_1  = u_2                                              ┐
  uh_1 = uh_2                                             │ R2
  b_1  = b_2                                              ┘

  do i = 2, N - 1                                     45  ┐
    u_1(i,2) = u_1(i,2) + uh_1(i,1)                   46  │
    u_1(i,N - 1) = u_1(i,N - 1) + uh_1(i,N)           47  │
  enddo                                               48  │
                                                          │
  do j = 3, N - 1                                     49  │
    do i = 2, N - 1                                   50  │
      b_1(i,j) = b_1(i,j) - 1 / b_1(i,j - 1)          51  │
      u_1(i,j) = u_1(i,j) +                               │
&                     u_1(i,j - 1) / b_1(i,j - 1)     52  │
    enddo                                             53  │ C
  enddo                                               54  │
  do i = 2, N - 1                                     55  │
    u_1(i,N - 1) = u_1(i,N - 1) / b_1(i,N - 1)        56  │
  enddo                                               57  │
  do j = N - 2, 2, -1                                 58  │
    do i = 2, N - 1                                   59  │
      u_1(i,j) = (u_1(i,j) +                              │
&                     u_1(i,j + 1)) / b_1(i,j)        60  │
    enddo                                             61  │
  enddo                                               62  ┘
enddo                                                 63
end                                                   64
```

Figure 8: 2-D Alternating Direction Implicit method (with selected phases and transitions)

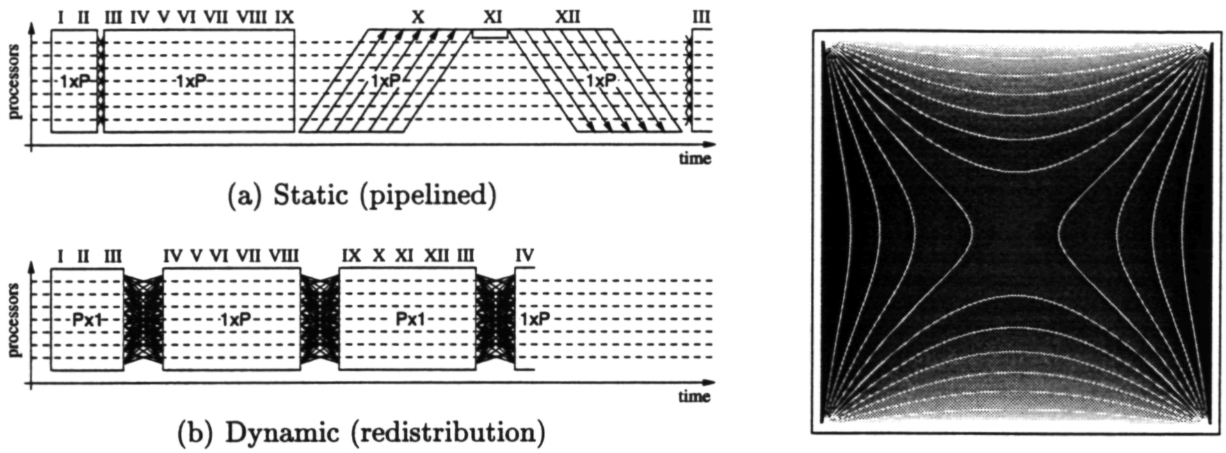
```
transpose(p, numproc, ID, direction, byte,
          R, rm, rn, mb, n,
          C, cm, cn, m, nb, buf)

transpose2(p, numproc, ID, direction, byte,
          R1, R2, rm, rn, mb, n,
          C1, C2, cm, cn, m, nb, buf)

transpose3(p, numproc, ID, direction, byte,
          R1, R2, R3, rm, rn, mb, n,
          C1, C2, C3, cm, cn, m, nb, buf)
```

Figure 9: Interfaces for library-based transpose operations

be treated in the same manner as statically partitioned code. Note that general redistribute and realign HPF directives [17] can also be handled in a similar manner. Some care must be taken during the renaming process when control flow (loops and conditionals) is present in order to ensure that all paths are always operating on active distributions. In general, this may require some amount of code duplication and specialization (potentially causing exponential code growth if some care is not taken), although this topic is beyond the scope of this paper.

The communication required to perform the individual redistribution operations will eventually be generated automatically by the compiler [23]. Since these techniques have not yet been integrated into PARADIGM, the redistribution required in ADI will be performed using library support. In Figure 9, the interfaces are shown for performing a transpose on a two-dimensional array between a row-wise and a column-wise block distribution. The transpose itself depends on the processor location (p), the total number of processors involved (numproc), and the transpose direction (forward or reverse). For both the row-wise (R) and the column-wise (C) distributions of the array, the function also requires: the size in bytes of an individual array element, the declared size of the dimensions (rm × rn, cm × cn), as well as size of the section to transpose (mb × n, m × nb). Each transpose is also assigned a unique ID as well as a small buffer large enough to pack or unpack one section of the array (mb × nb). To support the simultaneous transpose of multiple arrays, separate functions are also provided for a fixed number of identically sized arrays. This could instead have been accomplished with more generality by requiring pack and unpack operations to be provided to handle the individual sections of outgoing and incoming data.

Even though these functions only serve the purpose of redistributing two-dimensional arrays between one-dimensional block-wise distributions, it should be possible to implement a (more complex) operation that could exchange any two block-distributed dimensions of a multi-dimensional



(a) Static (pipelined)

(b) Dynamic (redistribution)

Figure 10: Modes of parallel execution for ADI and the solution for the test data

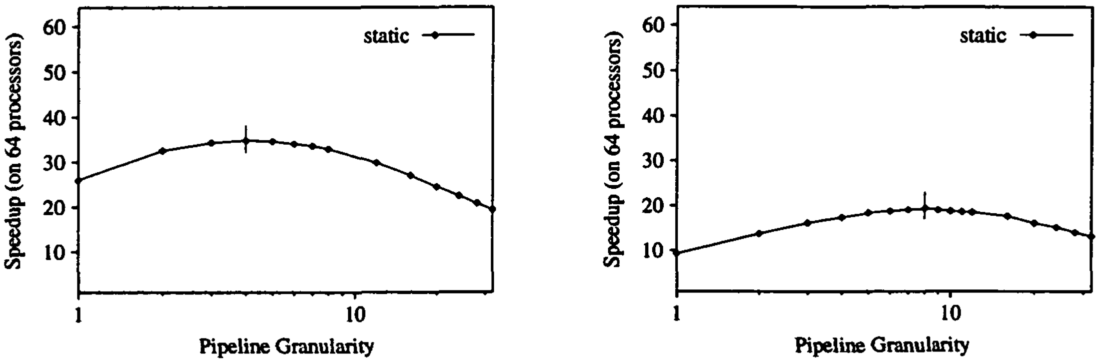
array. By considering every possible pairing of distributions (block, cyclic, and block-cyclic) over any number of partitioned dimensions, the number of functions required to fully implement a general redistribution library would be enormous. For this reason, redistribution operations should be automatically generated for specific cases, as they are needed, in order to support general redistribution [23].

Also, to improve the efficiency of memory management when moving large regions of data (such as required during redistribution), the array section pack and unpack routines in the PARADIGM run-time library are optimized to use block memory operations (`memcpy`) when possible. Furthermore, to avoid unnecessary congestion in the communication network, the order in which communication takes place during the redistribution is scheduled such that every processor sends data to a different processor at each step in the redistribution (as opposed to each processor communicating with an identical sequence of destinations). Even though these optimizations are currently implemented as part of the run-time library, they will also be incorporated into the code generation techniques for automated redistribution when it is integrated with the rest of the compiler.

6 Evaluation

In order to evaluate the effectiveness of dynamic distribution, the ADI program with a problem size of 512 × 512[6] is compiled with both a fully static distribution (one iteration shown in Figure 10a)

[6] In order to prevent poor serial performance from cache-line aliasing due to the power of two problem size, the arrays were also padded with an extra element at the end of each column. This optimization, although here performed by hand, is automated by aggressive serial optimizing compilers such as the KAP preprocessor from KAI.



(a) Intel Paragon

(b) TMC CM-5

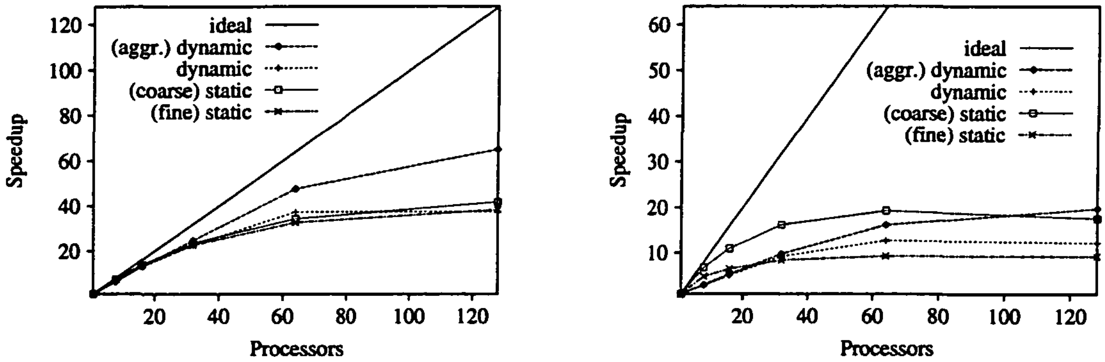
Figure 11: Coarse grain pipelining for ADI

well as with the best dynamic distribution (one iteration shown in Figure 10b). These two parallel versions of the code were run on an Intel Paragon and a Thinking Machine's CM-5 to examine the performance of each on the different architectures. Recall that the time for redistribution on the CM-5 was high enough that a static partitioning was predicted to perform better. On the Paragon, the cost of redistribution was low enough that a dynamic partitioning was selected.

With initial conditions of zero within the core of the matrix and upper and lower boundaries with a value of 30, the two schemes return the resulting solution shown in Figure 10 (overlaid with contours along constant potentials) which corresponds to that computed by the serial code.

The static scheme illustrated in Figure 10a performs a shift operation to communicate some required values and then satisfies two recurrences in the program through a technique known as software pipelining [13, 20]. Since values are being propagated through the array during the pipelined computation, processors must wait for results to be computed before continuing with their own part of the computation. Depending on the ratio of communication and computation performance for a given machine, exactly how much data is computed before communicating to the next processor will have a great effect on the performance of pipelined computations.

In Figure 11, a small experiment is performed to determine the *granularity* of the pipelines for the static partitioning. A granularity of one (fine-grain) indicates that values are communicated to waiting processors as soon as they are produced. By increasing the granularity, more values are computed before communicating, thereby amortizing the cost of establishing communication in exchange for some reduction in parallelism. For the two machines, it can be seen that by selecting



(a) Intel Paragon

(b) TMC CM-5

Figure 12: Performance of ADI

the appropriate granularity; the performance of the static partitioning can be improved. Both a fine-grain and an optimal coarse-grain static partitioning will be compared with the dynamic partitioning.

The redistribution present in the dynamic scheme appears as 3 transposes[7] performed at two points within an outer loop (the exact points in the program can be seen in Figure 6). Since the sets of transposes occur at the same point in the program, the data to be communicated for each transpose can be aggregated into a single message during the actual transpose. It has been previously shown that aggregating communication improves performance by reducing the overhead of communication [20], so we will also examine aggregating the individual transpose operations here.

In Figure 12, the performance of both static and dynamic partitionings for ADI is shown for an Intel Paragon and a Thinking Machines CM-5. For the dynamic partitioning, both aggregated and non-aggregated transpose operations were compared. For both machines, it is apparent that aggregating the transpose communication is very effective, especially as the program is executed on larger numbers of processors. This can be attributed to the fact that the start-up cost of communication (which is usually a couple orders of magnitude greater than the per byte transmission cost) is being amortized over multiple messages with the same source and destination.

For the static partitioning, fine grain pipelining was compared to coarse-grain using the gran-

[7] This could have been reduced to 2 transposes at each point if we allowed the cuts to reorder statements and perform loop distribution on the innermost loops (between statements 17, 18 and 41, 42), but these optimizations are not examined here.

	Intel Paragon		TMC CM-5	
procs	individual	aggregated	individual	aggregated
8	36.7	32.0	138.9	134.7
16	15.7	15.6	86.8	80.5
32	14.8	10.5	49.6	45.8
64	12.7	6.2	40.4	29.7
128	21.6	8.7	47.5	27.4

Table 3: Empirically estimated time (ms) for a 512 × 512 transpose

ularity selected earlier. The coarse-grain optimization yielded the greatest benefit on the CM-5 while still improving the performance (to a lesser degree) on the Paragon. For the Paragon, the dynamic partitioning with aggregation clearly improved performance (by a factor of 1.7 over fine-grain static, 1.6 over coarse-grain). On the CM-5 the dynamic partitioning with aggregation showed performance gains of over a factor of two compared to the fine-grain static partitioning but only outperformed the coarse-grain version for extremely large numbers of processors. For this reason, it would appear that the limiting factor on the CM-5 is the performance of the communication.

As a final check, the cost of performing a single transpose is estimated from the communication overhead present in the dynamic runs. Ignoring any performance gains from cache effects, the communication overhead can be computed by subtracting the ideal run time (serial time divided by the selected number of processors) from the measured run time. Given that 3 arrays are transposed 200 times, the resulting overhead divided by 600 yields a rough estimate of how much time is required to redistribute a single array. The results of this exercise are summarized in Table 3.

7 Conclusions

Dynamic data partitionings can provide higher performance from programs containing competing data access patterns. The distribution selection technique presented in this paper provides a means of automatically determining the best distribution scheme to use for a particular machine in an efficient manner. By utilizing an existing static partitioning algorithm and cost estimation framework, the number of phases examined as well as the amount of redistribution considered is kept to a minimum. Further investigation into the techniques for improving the heuristic used to obtain the maximal cut during phase selection is currently under way. We are also in the process of applying interprocedural analysis to investigate possible redistribution at procedure boundaries.

Acknowledgements: We would like to thank Amber Roy Chowdhury for his assistance with the coding of the serial ADI algorithm used as an example in this paper, John Chandy, Amber Roy Chowdhury, and Eugene Hodges for discussions on algorithm complexity, Steven Parkes for his suggestions on improving the efficiency of the memory management routines in the PARADIGM run-time library, as well as Christy Palermo for her suggestions and comments on this paper.

The figures used in this paper for the communication graphs were also generated using a software package known as "Dot" written by Eleftherios Koutsofios and Steven North with the Software and Systems Research Center, AT&T Bell Laboratories (contact: north@research.att.com).

References

[1] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 126–138, Albuquerque, NM, June 1993.

[2] J. M. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112–125, Albuquerque, NM, June 1993.

[3] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers. In *Proceedings of the First International Workshop on Parallel Processing*, pages 322–330, Bangalore, India, Dec. 1994.

[4] R. Bixby, K. Kennedy, and U. Kremer. Automatic Data Layout Using 0-1 Integer Programming. In *Proceedings of the 1994 International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, Montréal, Canada, Aug. 1994.

[5] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers, Design, Implementation, and Performance Results. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 351–360, Tokyo, Japan, July 1993.

[6] B. Chapman, T. Fahringer, and H. Zima. Automatic support for data distribution on distributed memory multiprocessor systems. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, 1993. Springer-Verlag.

[7] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, pages 145–164, 1992.

[8] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S. H. Teng. Automatic Array Alignment in Data-Parallel Programs. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles of Programming Languages*, Charleston, SC, Jan. 1993.

[9] T. Fahringer. *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers.* PhD thesis, University of Vienna, Vienna, Austria, Sept. 1993. TR93-3.

[10] G. Golub and J. M. Ortega. *Scientific Computing: An Introduction with Parallel Computing.* Academic Press, San Diego, CA, 1993.

[11] M. Gupta and P. Banerjee. Compile-Time Estimation of Communication Costs on Multicomputers. In *Proceedings of the Sixth International Parallel Processing Symposium*, Beverly Hills, CA, Mar. 1992.

[12] M. Gupta and P. Banerjee. PARADIGM: A Compiler for Automated Data Partitioning on Multicomputers. In *Proceedings of the 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[13] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD Distributed Memory Machines. *Communications of the ACM*, 35(8):66–80, Aug. 1992.

[14] D. E. Hudak and S. G. Abraham. Compiler Techniques for Data Partitioning of Sequentially Iterated Parallel Loops. In *Proceedings of the Fourth ACM International Conference on Supercomputing*, pages 187–200, Amsterdam, The Netherlands, June 1990.

[15] D. E. Hudak and S. G. Abraham. *Compiling Parallel Loops for High Performance Computers – Partitioning, Data Assignment and Remapping.* Kluwer Academic Publishers, Boston, MA, 1993.

[16] K. Knobe and V. Natarajan. Data Optimization: Minimizing Residual Interprocessor Data Motion on SIMD Machines. In *Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, Oct. 1990.

[17] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, MA, 1994.

[18] U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle. Automatic Data Layout for Distributed-Memory Machines in the D Programming Environment. In C. Kessler, editor, *Automatic Parallelization – New Approaches to Code Generation, Data Distribution, and Performance Predication*, pages 136–152, 1993.

[19] J. Li and M. Chen. Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays. In *Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, Oct. 1990.

[20] D. J. Palermo, E. Su, J. A. Chandy, and P. Banerjee. Compiler Optimizations for Distributed Memory Multicomputers used in the PARADIGM Compiler. In *Proceedings of the 23rd International Conference on Parallel Processing*, pages II:1–10, St. Charles, IL, Aug. 1994.

[21] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten. Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors. In *Proceedings of the 18th International Conference on Parallel Processing*, pages II:39–48, St. Charles, IL, Aug. 1989.

[22] 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, Oct. 1991.

[23] S. Ramaswamy and P. Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. In *Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, Feb. 1995.

[24] H. Sivaraman and C. S. Raghavendra. Compiling for MIMD Distributed Memory Machines. Tech. Report EECS-94-021, School of Electrical Enginnering and Computer Science, Washington State University, Pullman, WA, 1994.

[25] P. S. Tseng. A Parallelizing Compiler for Distributed-Memory Parallel Computers. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[26] S. Wholey. Automatic Data Mapping for Distributed-Memory Parallel Computers. In *Proceedings of the Sixth ACM International Conference on Supercomputing*, Washington D.C., July 1992.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
distribution unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
Coordinated Science Lab
University of Illinois

6b. OFFICE SYMBOL *(If applicable)*
N/A

7a. NAME OF MONITORING ORGANIZATION
NASA and ARPA

6c. ADDRESS *(City, State, and ZIP Code)*
1308 W. Main St.
Urbana, IL 61801

7b. ADDRESS *(City, State, and ZIP Code)*
600 Independence, SW, Washington, DC 20546
3701 North Fairfax Dr., Arlington, VA 22203

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
7a.

8b. OFFICE SYMBOL *(If applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS *(City, State, and ZIP Code)*
7b.

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Automatic Selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers

12. PERSONAL AUTHOR(S)
Daniel J. Palermo and Prithviraj Banerjee

13a. TYPE OF REPORT
Technical

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT *(Year, Month, Day)*
April 1995

15. PAGE COUNT
24

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
distributed memory, multicomputers, data partitioning, dynamic redistribution, parallelizing compilers

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

For distributed memory multicomputers such as the Intel Paragon, the IBM SP-2, the NCUBE/2, and the Thinking Machines CM-5, the quality of the data partitioning for a given application is crucial to obtaining high performance. This task has traditionally been the user's responsibility, but in recent years much effort has been directed to automating the selection of data partitioning schemes. Several researchers have proposed systems that are able to produce data distributions that remain in effect for the entire execution of an application. For complex programs, however, such static data distributions may be insufficient to obtain acceptable performance. The selection of distributions that dynamically change over the course of a program's execution adds another dimension to the data partitioning problem. In this paper, we present a technique that can be used to automatically determine which partitionings are most beneficial over specific sections of a program while taking into account the added overhead of performing redistribution. This system is being built as part of the PARADIGM (PARAllelizing compiler for DIstributed memory General-purpose Multicomputers) project at the University of Illinois. The complete system will provide a fully automated means to parallelize programs written in a serial programming model obtaining high performance on a wide range of distributed-memory multicomputers.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL

22b. TELEPHONE *(Include Area Code)*

22c. OFFICE SYMBOL

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED