in the Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing, Columbus, OH, Aug., 1995

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

Daniel J. Palermo and Prithviraj Banerjee

Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1308 West Main Street, Urbana, IL 61801
({palermo, banerjee}@crhc.uiuc.edu)

Abstract. For distributed-memory multicomputers such as the Intel Paragon, the IBM SP-1/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 (PARAlleliz-ing 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.

1 Introduction

Distributed-memory multicomputers such as the Intel Paragon, the IBM SP-1/SP-2, the NCUBE/2, and the Thinking Machines CM-5 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 effort has been aimed at source-to-source parallelizing compilers that relieve the programmer from the task of communication generation, while the task of data partitioning remains a responsibility of the programmer.

As part of the research performed in the PARADIGM (PARAllelizing compiler for DIstributed-memory General-purpose Multicomputers) project [2] at the University of Illinois, automatic data partitioning techniques have been developed to relieve the programmer of the burden of selecting good data distributions. Currently, the compiler can

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.

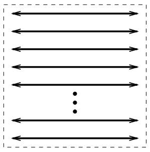
automatically select a static distribution of data (using a constraint-based algorithm [9]) specifying both the configuration of an abstract multi-dimensional mesh topology along with how program data should be distributed on the mesh. In this paper, we present a technique which extends the static partitioning 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; an experimental analysis of the presented techniques is performed in Section 5; and conclusions are presented in Section 6.

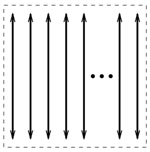
2 Motivation

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.

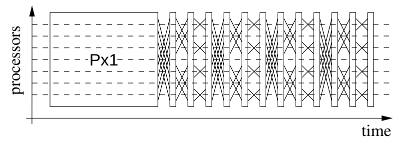
```
program FFT2D
complex Image(N,N)

*** 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
```



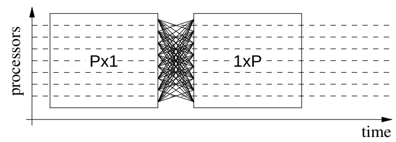
Row FFTs



Column FFTs



(a) Static (butterfly communication)



(b) Dynamic (redistribution)

Fig. 1. Two-dimensional Fast Fourier Transform

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 as 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 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

Static Partitioning Some of the ideas used in the static partitioning algorithm currently implemented in the PARADIGM compiler [9] were inspired by earlier work on multi-dimensional array alignment [14]. In addition to this work, in recent years much research has been focused on: performing multi-dimensional array alignment [5, 12, 14]; examining cases in which a communication-free partitioning exists [17]; showing how performance estimation is a key in selecting good data distributions [6, 22]; linearizing array accesses and analyzing the resulting one-dimensional accesses [20]; applying iterative techniques which minimize the amount of communication at each step [1]; and examining issues for special-purpose distributed architectures such as systolic arrays [21].

Dynamic Partitioning In addition to the work performed in static partitioning, a number of researchers have also been examining the problem of dynamic partitioning. Anderson and Lam [1] approach the dynamic partitioning problem using a heuristic which combines loop nests (with potentially different distributions) in such a way that the largest potential communication costs are eliminated first while still maintaining sufficient parallelism. Bixby, Kennedy and Kremer formulate the dynamic data partitioning problem in the form of a 0-1 integer programming problem by selecting a number of candidate distributions for each of a set of given phases and constructing constraints from the data relations [3]. Chapman, Fahringer, and Zima describe the design of a distribution tool that makes use of performance prediction methods when possible but also uses empirical performance data through a pattern matching process [4]. Hudak and Abraham have also proposed a method for selecting redistribution points based on locating significant control flow changes in a program [11]. More recently, Sheffler, Schreiber, Gilbert and Chatterjee have applied graph contraction methods to the dynamic alignment problem to reduce the size of the problem space that must be examined [19].

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 [3]. Even though this definition restricts phase boundaries to loop structures and does not allow overlapping 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.

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

```
program ADI2d
double precision u(N,N), uh(N,N), b(N,N), alpha
integer i, j, k
                                                        Phase
*** Initial value for u
do j = 1, N
  do i = 1, N
    u(i,j) = 0.0
  enddo                                                 I
  u(1,j) = 30.0
  u(n,j) = 30.0
enddo

*** Initialize uh
do j = 1, N
  do i = 1, N
    uh(i,j) = u(i,j)                                    II
  enddo
enddo

alpha = 4 * (2.0 / N)
do k = 1, maxiter
  *** Forward and backward sweeps along columns
  do j = 2, N - 1
    do i = 2, N - 1
      b(i,j) = (2 + alpha)
      uh(i,j) = (alpha - 2) * u(i,j) +                  III
&                           u(i,j + 1) + u(i,j - 1)
    enddo
  enddo
  do j = 2, N - 1
    uh(2,j) = uh(2,j) + u(1,j)                          IV
    uh(N - 1,j) = uh(N - 1,j) + u(N,j)
  enddo

  do j = 2, N - 1
    do i = 3, N - 1
      b(i,j) = b(i,j) - 1 / b(i - 1,j)
      uh(i,j) = uh(i,j) + uh(i - 1,j) / b(i - 1,j)      V
    enddo
  enddo
```

```
                                                             Phase
    do j = 2, N - 1
      uh(N - 1,j) = uh(N - 1,j) / b(N - 1,j)                 VI
    enddo
    do j = 2, N - 1
      do i = N - 2, 2, -1
        uh(i,j) = (uh(i,j) + uh(i + 1,j)) / b(i,j)           VII
      enddo
    enddo

    *** Forward and backward sweeps along rows
    do j = 2, N - 1
      do i = 2, N - 1
        b(i,j) = (2 + alpha)
        u(i,j) = (alpha - 2) * uh(i,j) +                     VIII
  &                           uh(i + 1,j) + uh(i - 1,j)
      enddo
    enddo
    do i = 2, N - 1
      u(i,2) = u(i,2) + uh(i,1)
      u(i,N - 1) = u(i,N - 1) + uh(i,N)                      IX
    enddo

    do j = 3, N - 1
      do i = 2, N - 1
        b(i,j) = b(i,j) - 1 / b(i,j - 1)                     X
        u(i,j) = u(i,j) + u(i,j - 1) / b(i,j - 1)
      enddo
    enddo
    do i = 2, N - 1
      u(i,N - 1) = u(i,N - 1) / b(i,N - 1)                   XI
    enddo
    do j = N - 2, 2, -1
      do i = 2, N - 1
        u(i,j) = (u(i,j) + u(i,j + 1)) / b(i,j)              XII
      enddo
    enddo
  enddo
  end
```

Fig. 2. 2-D Alternating Direction Implicit method (ADI) with operational phases shown

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 [9] previously developed in the PARADIGM project. Static cost estimation techniques [8] 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 iterative method[2] (ADI) is shown which computes the solution of an elliptic partial differential equation known as Poisson's equation [7]. 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

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

free-space potential created by a surface with an electrical charge.

For the program in Figure 2, a static data distribution will incur 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 such that the sum of the execution times of the resulting phases is less than the original (as illustrated in Figure 3). 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 (generated by Parafrase-2 [16]) 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 assigned to (*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.

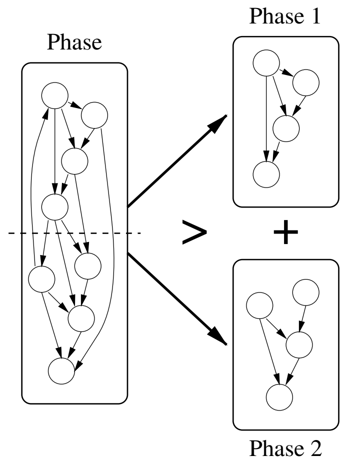
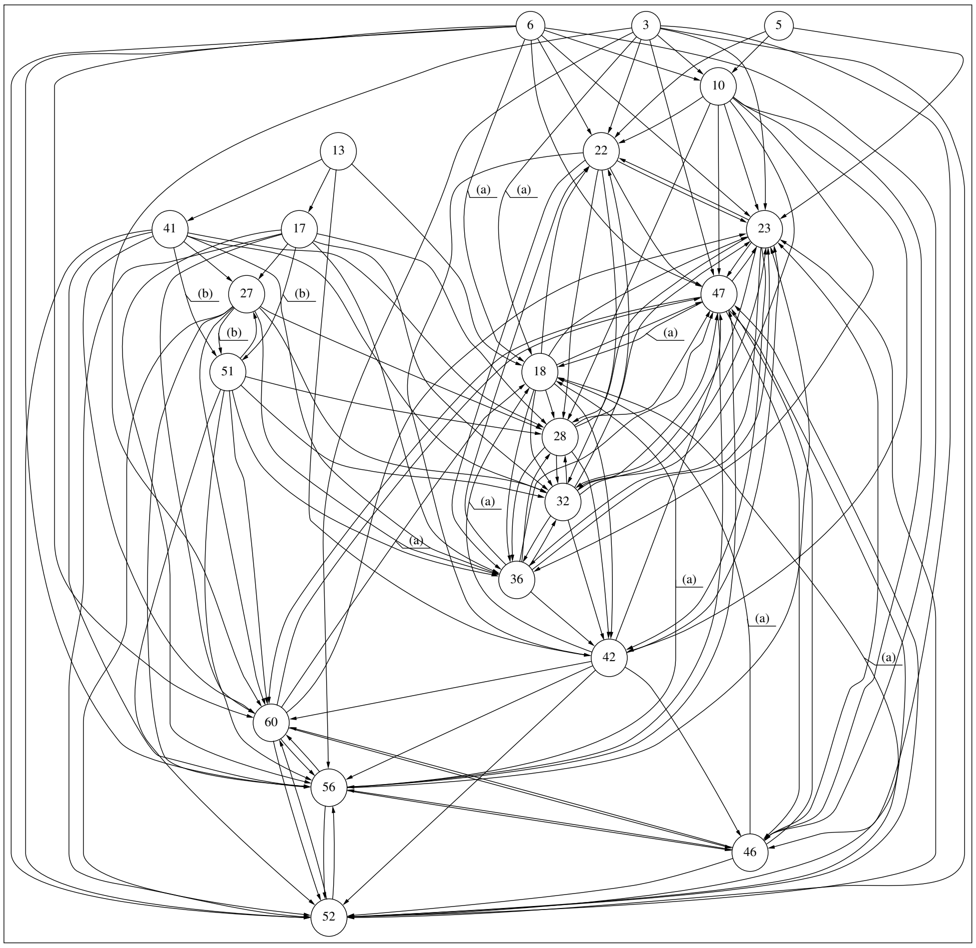


Fig. 3. Phase decomposition

In Figure 4, the communication graph is shown for ADI with some of 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). Conditionals appearing in the cost expressions represent costs that will be incurred based on specific distribution decisions (e.g. $P_2 > 1$ is true if the second mesh dimension is assigned more than one processor). For reference, the communication models for an Intel Paragon and a Thinking Machines CM-5, corresponding to the communication primitives used in the cost expressions, 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 to the same array, the edge weight is the sum of all communication for that array.



(a) $100 * (P_2 > 1) * \text{Shift}(510)$
(b) $3100 * \text{Transfer}(510)$

Fig. 4. Communication graph and example edge costs for ADI
(Statement numbers correspond to Figure 2)

	Intel Paragon	TMC CM-5
Transfer(m)	$50 + 0.018m$	$23 + 0.12m \;\; m \leq 16$ $86 + 0.12m \;\; m > 16$
Shift(m)	$2 * \text{Transfer}(m)$	

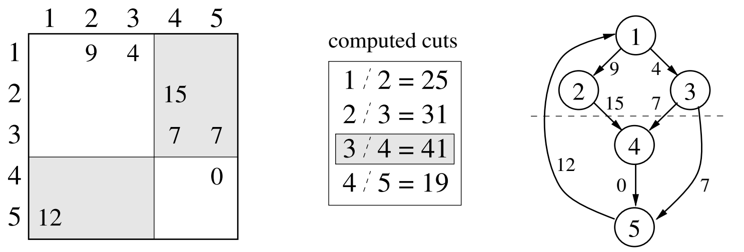
Table 1. Communication primitives

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.

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:

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).

1. 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).
2. While computing the cost of each cut also keep track of the current maximum cut.
3. If there is more than one cut with the same maximum value, choose the first.
4. Mark the arrays involved in the cut edges to redistribute and split the phase using the selected cut.



(a) Adjacency matrix (b) Actual representation

Fig. 5. Example graph illustrating the computation of a cut

In Figure 5, 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, 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 that if a cut occurs within a loop body, and loop distribution can be performed, the amount of redistribution can be greatly reduced by lifting it out of the distributed loop body and performing it in between the two sections of the loop. 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 while possibly reducing the amount of data to be redistributed. Both of these optimizations can be used to reduce the cost of redistribution but neither be examined in this paper.

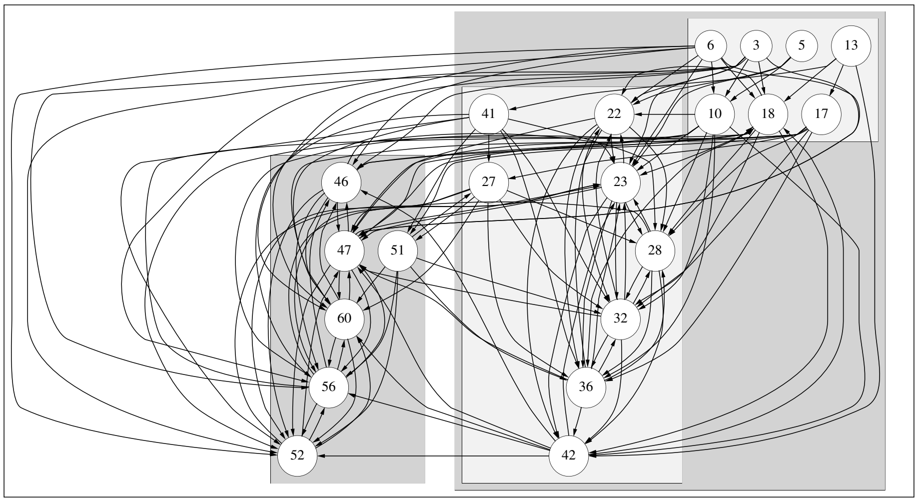


Fig. 6. Partitioned communication graph for ADI
(Statement numbers correspond to Figure 2)

A new distribution is selected for each of the resulting phases and the process is continued recursively. As shown in Figure 3, 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 6, 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 distributions, 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

[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.

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 [9].

4.2 Phase and Phase Transition Selection

After the program has been recursively decomposed into a hierarchy of phases, redistribution costs are estimated [18] and are weighted by their execution count. for each of the possible phase transitions. 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 is allowed at the granularity of the lowest level of the phase decomposition. Edges with the resulting costs are used to connect the phases in a *phase transition graph* (as in Figure 7) to help determine which phases and transitions are necessary to obtain the best performance.

In the presence of control flow which causes iteration in the program (i.e. loops and backward branches), additional redistribution may be induced. A redistribution edge that occurs within a loop or between the source and target of a backward branch will have its cost doubled to account for a potential reverse redistribution[5]. Once costs have been assigned to all redistribution edges, the best sequence of phases and phase transitions is selected by computing the shortest path on the phase transition graph.

Using the cost models for an Intel Paragon and a Thinking Machines CM-5, the distributions and estimated execution times reported by the static partitioner for the resulting phases (described as ranges of operational phases) is 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 static cost estimator, but they still exhibit similar enough performance trends to be used as estimates.

Op. Phases(s)	Distribution		Intel Paragon	TMC CM-5	
I-XII	`*,block`	1×32	20.588940	35.970293	Level 1
I-VIII	`*,block`	1×32	1.354800	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

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 7). 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.

[5] This is a conservative estimate of the actual cost since it assumes that reverse redistribution will always occur.

Pseudo-code for the dynamic partitioning algorithm is presented in Figure 8 to briefly summarize the entire procedure.

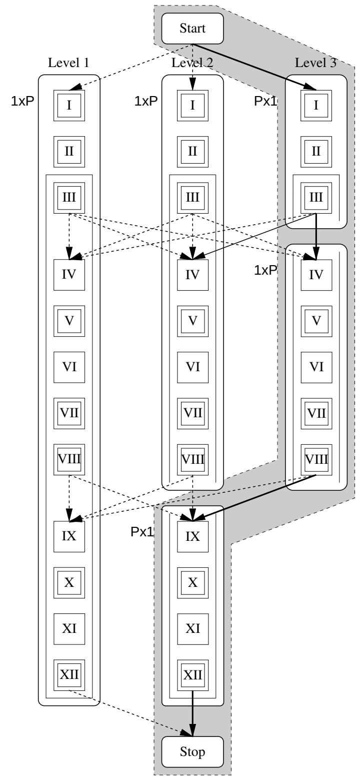


Fig. 7. Selected phases for ADI

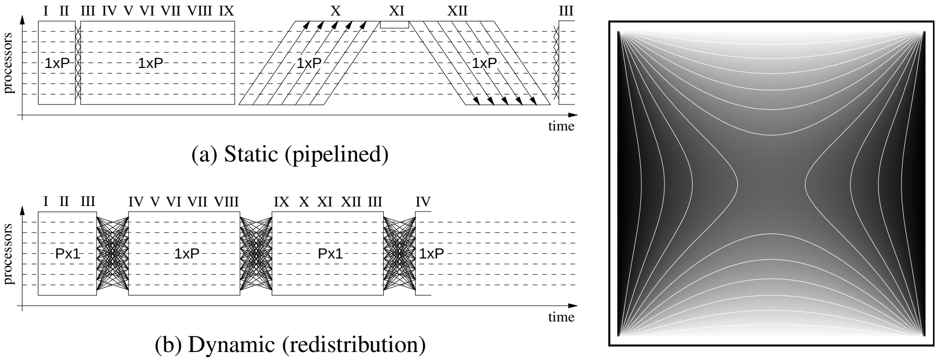
Construct the communication graph for the *program*
Perform an initial static partitioning on the *program*
phases = Decompose_Phase(*program*)
scheme = Select_Redistribution(*phases*)

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 (cost($phase_1$) + cost($phase_2$)) $<$ 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)

Fig. 8. Pseudo-code for the partitioning algorithm

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 [3] 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.



(a) Static (pipelined)

(b) Dynamic (redistribution)

Fig. 9. Modes of parallel execution for ADI and the solution for the test data

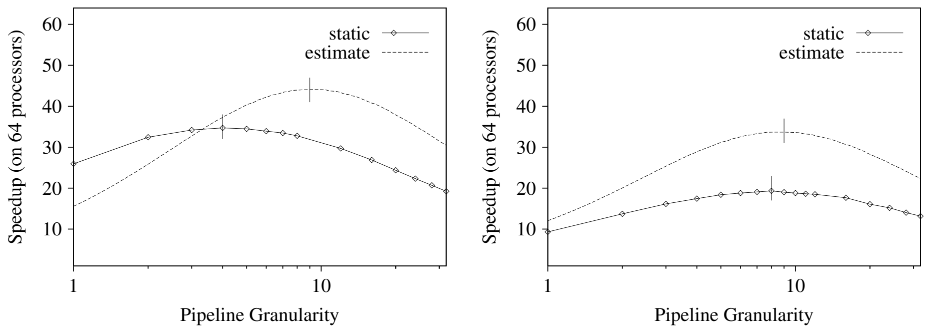
5 Evaluation

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 9a) as well as with the selected dynamic distribution (one iteration shown in Figure 9b). 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. With initial conditions of zero within the core of the matrix and upper and lower boundaries with a value of 30, both schemes obtain the solution shown in Figure 9 (overlaid with contours along constant potentials).

The static scheme illustrated in Figure 9a performs a shift operation to initially obtain some required data and then satisfies two recurrences in the program using software pipelining [10, 15]. 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.

A small experiment is first performed to determine the best pipeline *granularity* for the static partitioning. A granularity of one (fine-grain) causes values to be 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. In addition to the experimental data, compile-time estimates of the pipeline execution [15] are shown in Figure 10. For the two machines, it can be seen that by selecting the appropriate granularity; the performance of the static partitioning can be improved. Even though the current computational model is only based on high-level instruction counts [8] and ignores

[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

Fig. 10. Coarse grain pipelining for ADI

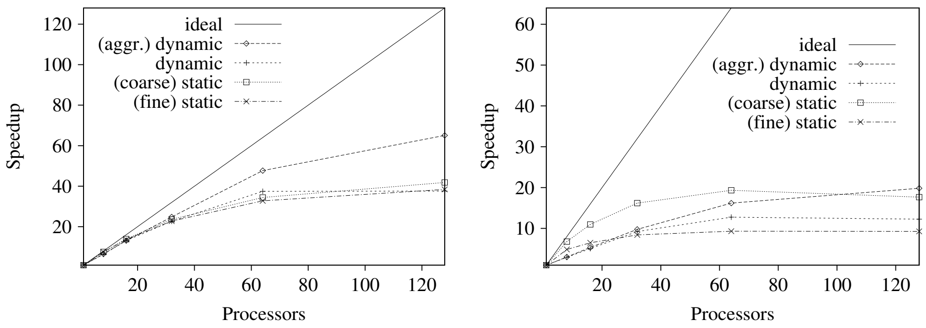
processor pipeline and cache effects, the trend is still modeled well enough to select a granularity at compile-time that closely approximates the optimal. Both a fine-grain and the optimal coarse-grain static partitioning will be compared with the dynamic partitioning.

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 7). 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 [15], so we will also examine aggregating the individual transpose operations here.

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 the transpose redistribution is currently part of the run-time library, these modifications will also be incorporated into the code generation techniques for automated redistribution [18] when it is integrated with the rest of the compiler.

In Figure 11, the performance of both static and dynamic partitionings for ADI is shown for an Intel Paragon and a Thinking Machines CM-5. 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. For the dynamic partitioning, both aggregated and non-aggregated transpose operations were compared. For both machines, it is apparent that

[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.



(a) Intel Paragon (b) TMC CM-5

Fig. 11. Performance of ADI

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 can be several 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 granularity 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 over 70% compared to the fine-grain and 60% compared to the coarse-grain static distribution). 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, in Table 3 the cost of performing a single transpose in either direction ($P\times 1 \rightleftarrows 1\times P$) 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 trans-

	Intel Paragon		TMC CM-5	
processors	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) to transpose a 1-D partitioned 512×512 double precision matrix

posed 200 times, the resulting overhead divided by 600 yields a rough estimate of how much time is required to redistribute a single array.

From Table 3 it can be seen that as more processors are involved in the operation, the time taken to perform one transpose levels off until a certain number of processors is reached. After this point, the amount of data being handled by each individual processor is small enough that the start-up overhead of the communication has become the controlling factor. Aggregating the redistribution operations minimizes this effect thereby achieving higher levels of performance than would be possible otherwise.

6 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. A key requirement in automating this selection process is to be able to obtain estimates of communication and computation costs which accurately model the behavior of the program under a given distribution. Furthermore, by building upon existing static partitioning techniques the number of phases examined as well as the amount of redistribution considered is kept to a minimum.

Further investigation into the application of statement reordering and loop distribution to reduce the amount of required redistribution is currently under way. We are also in the process of applying interprocedural analysis along with the techniques presented in this paper to investigate possible redistribution at procedure boundaries.

Acknowledgements: We would like to thank the reviewers for their comments, Amber Roy Chowdhury for his assistance with the coding of the serial ADI algorithm, 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” developed by Eleftherios Koutsofios and Steven North with the Software and Systems Research Center, AT&T Bell Laboratories [13].

References

1. J. M. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proc. of the ACM SIGPLAN ’93 Conf. on Prog. Lang. Design and Implementation*, pages 112–125, Albuquerque, NM, June 1993.
2. P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. An Overview of the PARADIGM Compiler for Distributed-Memory Multicomputers. to appear in *IEEE Computer*, 1995.
3. R. Bixby, K. Kennedy, and U. Kremer. Automatic Data Layout Using 0-1 Integer Programming. In *Proc. of the 1994 Int’l Conf. on Parallel Archs. and Compilation Techniques*, pages 111–122, Montréal, Canada, Aug. 1994.
4. B. Chapman, T. Fahringer, and H. Zima. Automatic support for data distribution on distributed memory multiprocessor systems. In *Proc. of the 6th Work. on Langs. and Compilers for Parallel Computing*, pages 184–199, Portland, OR, Aug. 1993. Springer-Verlag.

5. S. Chatterjee, J. R. Gilbert, R. Schreiber, and S. H. Teng. Automatic Array Alignment in Data-Parallel Programs. In *Proc. of the 20th ACM SIGPLAN Symp. on Principles of Prog. Langs.*, pages 16–28, Charleston, SC, Jan. 1993.
6. T. Fahringer. *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers*. PhD thesis, Univ. of Vienna, Vienna, Austria, Sept. 1993. TR93-3.
7. G. Golub and J. M. Ortega. *Scientific Computing: An Introduction with Parallel Computing*. Academic Press, San Diego, CA, 1993.
8. M. Gupta and P. Banerjee. Compile-Time Estimation of Communication Costs on Multicomputers. In *Proc. of the 6th Int'l Parallel Processing Symp.*, pages 470–475, Beverly Hills, CA, Mar. 1992.
9. M. Gupta and P. Banerjee. PARADIGM: A Compiler for Automated Data Partitioning on Multicomputers. In *Proc. of the 7th ACM Int'l Conf. on Supercomputing*, Tokyo, Japan, July 1993.
10. 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.
11. D. E. Hudak and S. G. Abraham. *Compiling Parallel Loops for High Performance Computers – Partitioning, Data Assignment and Remapping*. Kluwer Academic Pub., Boston, MA, 1993.
12. K. Knobe, J. Lukas, and G. Steele Jr. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *J. of Parallel and Distributed Computing*, 8(2):102–118, Feb. 1990.
13. B. Krishnamurthy, editor. *Practical Reusable UNIX Software*. John Wiley and Sons Inc., New York, NY, 1995.
14. J. Li and M. Chen. The Data Alignment Phase in Compiling Programs for Distributed-Memory Machines. *J. of Parallel and Distributed Computing*, 13(2):213–221, Oct. 1991.
15. D. J. Palermo, E. Su, J. A. Chandy, and P. Banerjee. Compiler Optimizations for Distributed Memory Multicomputers used in the PARADIGM Compiler. In *Proc. of the 23rd Int'l Conf. on Parallel Processing*, pages II:1–10, St. Charles, IL, Aug. 1994.
16. 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 *Proc. of the 18th Int'l Conf. on Parallel Processing*, pages II:39–48, St. Charles, IL, Aug. 1989.
17. J. Ramanujam and P. Sadayappan. Compile-time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):472–481, Oct. 1991.
18. S. Ramaswamy and P. Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. In *Frontiers '95: The 5th Symp. on the Frontiers of Massively Parallel Computation*, pages 342–349, McLean, VA, Feb. 1995.
19. T. J. Sheffler, J. R. Gilbert, R. Schreiber, and S. Chatterjee. Aligning Parallel Arrays to Reduce Communication. In *Frontiers '95: The 5th Symp. on the Frontiers of Massively Parallel Computation*, pages 324–331, McLean, VA, 1995.
20. 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 Univ., Pullman, WA, 1994.
21. P. S. Tseng. Compiling Programs for a Linear Systolic Array. In *Proc. of the ACM SIGPLAN '90 Conf. on Prog. Lang. Design and Implementation*, pages 311–321, White Plains, NY, June 1990.
22. S. Wholey. Automatic Data Mapping for Distributed-Memory Parallel Computers. In *Proc. of the 6th ACM Int'l Conf. on Supercomputing*, pages 25–34, Washington D.C., July 1992.

This article was processed using the LaTeX macro package with LLNCS style