DDT: A Research Tool for Automatic Data Distribution in High Performance Fortran

EDUARD AYGUADÉ, JORDI GARCIA, MERCÈ GIRONÈS, M. LUZ GRANDE, AND JESÚS LABARTA

Computer Architecture Department, Polytechnic University of Catalunya, cr. Gran Capità s/núm, Mòdul D6, 08034 - Barcelona, Spain

ABSTRACT

This article describes the main features and implementation of our automatic data distribution research tool. The tool (DDT) accepts programs written in Fortran 77 and generates High Performance Fortran (HPF) directives to map arrays onto the memories of the processors and parallelize loops, and executable statements to remap these arrays. DDT works by identifying a set of computational phases (procedures and loops). The algorithm builds a search space of candidate solutions for these phases which is explored looking for the combination that minimizes the overall cost; this cost includes data movement cost and computation cost. The movement cost reflects the cost of accessing remote data during the execution of a phase and the remapping costs that have to be paid in order to execute the phase with the selected mapping. The computation cost includes the cost of executing a phase in parallel according to the selected mapping and the owner computes rule. The tool supports interprocedural analysis and uses control flow information to identify how phases are sequenced during the execution of the application.

INTRODUCTION

Data distribution is one of the topics of current research in parallelizing environments for nonuniform memory access (NUMA) massive parallel processors (MPP). In these systems, each processor has direct access to its local (or close) memory and indirect access to the remote memories of other processors through the interconnection network. The cost of accessing a local memory location can be more than one order of magnitude faster than the cost of accessing a remote memory location. In these systems, the choice of a good data distribution can dramatically affect performance because of the nonuniformity of the memory system.

Received May 1995
Revised February 1996

Scientific Programming, Vol. 6, pp. 73–94 (1997)
CCC 1058-9244/97/010073-22

Several researchers have targeted their research efforts to this topic. For instance, the Crystal compiler and language project [1], the implementation of PARADIGM [2] on top of Parafrase-2 and its continuation on the PTRAN II compiler [3] at IBM, the framework for the automatic determination of array mappings presented in [4, 5], or the automatic data-mapping strategy [6] for use in the D-programming environment currently under development at Rice University are examples of projects in this area. Other groups have targeted their efforts to the effective compilation of programs containing the specification of the data mapping, such as the VFCS system [7] for the Vienna Fortran language [8], the Fortran-D compiler [9] and language [10], or the current commerical compilers (xlHPF [11], PGHPF [12]) for High Performance Fortran (HPF) [13].

Automatic data distribution maps arrays into the physically distributed memories of the processors according to the array access patterns and parallel execution of operations within computationally intensive

phases. This mapping can be either static or dynamic. In a static mapping, the layout of the arrays does not change during the execution of the program; in a dynamic mapping, remapping operations are performed in order to change the layout of arrays in different computational phases.

Most static data distribution methods [1, 2, 4, 14, 15] perform the job in two main independent steps: alignment and distribution. The alignment step tries to relate the dimensions of the arrays used in a block of code with the dimensions of another array called the template (interdimensional alignment), and for each aligned dimension, to find the appropriate shift between their elements (intradimensional alignment). A good alignment will minimize the overhead of interprocessor data movement.

The main differences between the previous methods is the kind of structure selected to represent the problem, and the way used to formulate and solve it. For the alignment step, Li and Chen [1] define and use the Component Affinity Graph (CAG) to represent alignment preferences, and it uses a heuristic algorithm to solve it. Gupta [2] also uses the CAG, but weighted with data movement costs. To do this, a default distribution has to be assumed. This proposal is more accurate than the previous one when weighting the edges of the CAG, but the solver is based on the same heuristic algorithm. Wholey [14] uses the preference graph defined by [16] in the framework of single-instruction multiple-data (SIMD) machines. This graph includes alignment preferences to preserve parallelism, but the resolution when the graph is in conflict is also based on heuristics. Scheffler et al. [4] define the alignment distribution graph where nodes represent program operations and edges connect definitions of array objects to their uses in these operations. Edges are weighted with the number of data items communicated along the edge. The alignment is found using a greedy algorithm as a heuristic to determine the minimum cost, and applying graph contraction operations to reduce the complexity of the problem. Kennedy and Kremer [15] design a framework to be used inside a data layout assistant tool for HPF. It is based on the CAG, and they solve the alignment problem using a 0-1 integer programming model, thus avoiding the use of heuristics.

Once the alignment has been decided, the distribution step decides which dimension(s) of the template are distributed and the number of processors assigned to each of them. A good distribution maximizes the potential parallelism of the code and offers the possibility of further reducing data movement by serializing. This goal could be trivially satisfied by assigning a datum to each processor, which maximizes parallelism. Li and Chen [17] match the aligned reference patterns with a predefined set of data movement routines. Each routine has an architecture-dependent cost parameterized in terms of the number of processors involved in the data motion and the amount of data being moved. The cost function for all the patterns is minimized by selecting the appropriate distribution strategy. Gupta [2] decides the dimensions to distribute (maximum two dimensions), assuming a default number of processors in each one and minimizing the total data movement plus computation time. When more than one dimension is distributed, he decides the number of processors to assign to each dimension by generating all possible combinations. Wholey [14] uses a hill climbing search method which initially assigns all array elements to one processor and estimates the cost. Then it double the number of processors and chooses the dimension to assign the new ones, until all available processors are utilized or the total cost is not further reduced.

In large problems where different computationally intensive phases occur, remapping actions between phases can increase the efficiency of the solution. In this case, a good solution is independently found for each phase, and realignment and/or redistribution statements are inserted where necessary. Data remapping is also one of the topics in this subject area of current research. Some of the proposals presented in the literature about array remapping [5, 18–20] are summarized in the rest of this section.

The D-System, currently under development at Rice University, considers the profitability of dynamic data remapping by exploring a search space of reasonable alignment and distribution spaces [18]. In their work, each phase has a set of candidate mapping schemes. Selecting a mapping scheme for each phase in the entire program is done by representing the problem with the data layout graph. Each possible mapping for a phase is represented with a node. Edges between two nodes in different phases represent the remapping that has to be carried out to execute each phase with the corresponding mapping. Nodes and edges have weights representing the overal cost of executing a phase with mapping and remapping costs, respectively, in terms of execution time. The problem is translated into a 0-1 integer programming problem suitable to be solved by a state-of-the-art general-purpose integer programming solver.

The FCS system [19] considers the problem in the framework of a data distribution tool for Fortran 90 source codes. In this scope, array-syntax assignment statements and WHERE masks are examined to determine candidate data mappings. A phase is basically a DO-loop containing array-syntax assignment state-

ments or WHERE masks in its body. Instead of looking for the optimal solution, it uses a tree-exhaustive algorithm with some heuristics to prune the search space. A conflict table storing the conflicts between the mappings of the arrays from one phase to the other is the basis of the remapping algorithm. This table determines which redistribution options are worth considering at each transition. From this information, a tree showing all the different alternatives of remapping is built. The aim is to determine the path in the tree with the lowest cost. The full remapping tree can easily grow to intractable proportions.

Chatterjee et al. [5] use a divide-and-conquer approach to the dynamic mapping problem. They initially assign a static mapping valid for all the nodes and then recursively divide them into regions which are assigned different mappings. Two regions are merged when the cost of the dynamic mapping is worse than the static mapping, taking computation, data movement, and remapping costs into account. Palermo and Banerjee [20] also use a divide-and-conquer approach in which the program is recursively decomposed into a hierarchy of candidate phases. Then, taking into account the cost of remapping between the different phases, the sequence of phases and phase transitions with the lowest cost is selected. They use [2] to assign mappings to the phases generated.

The Data Distribution Tool (DDT) is a research tool designed to generate both static and dynamic solutions. Since it is a research tool, it can use techniques that may be too computationally expensive to be included in a final compiler; however, this allows us to explore a rich set of solutions. The static module is based on the CAG but extended with some information regarding parallelism. We have also modified the original algorithms in [1, 17] to improve the quality of the mappings generated [21]. The current version of the static module generates both inter- and intradimensional alignments and *BLOCK* and *CYCLIC* distributions. The dynamic module explores a rich set of combinations; it is not exhaustive thanks to mechanisms included to cut down the search space [22]. The dynamic analysis is interprocedural and considers control flow to determine where the remapping actions have to be performed (between computational phases or across procedure boundaries).

The rest of the article is organized as follows. In the next section we give an overview of the whole data-mapping process in DDT. Section 3 details how the static solutions are found. Section 4 describes the algorithm which finds dynamic solutions and inserts remapping actions if they are found profitable. Sections 5 and 6 describe the extensions to the previous algorithm to handle control flow and interprocedural analyses, respectively. In Section 7 we present the main results from a set of experiments to test the validity and quality of the solutions generated by DDT. Finally, Section 8 gives some concluding remarks.

2 AN OVERVIEW OF THE DATA DISTRIBUTION PROCESS IN DDT

Our research tool (DDT) analyzes Fortran 77 code and annotates it with a set of HPF directives and executable statements that specify (1) how arrays are aligned to a set of template arrays and how the dimensions of these templates are distributed among processors; (2) the set of realignment and redistribution statements if the solution found is dynamic; and (3) the parallelization strategies for the loops that access distributed arrays. These decisions are done so that the amount of remote accesses is reduced as much as possible, while maximizing the parallelism achieved.

The external shell of DDT is the interprocedural analysis module; this module is based on the call graph for the entire program. In a bottom-up pass over the call graph, each procedure is analyzed when all the procedures called by it have already been processed. Notice that Fortran 77 does not allow recursion, so no cycles can be found in the call graph. From the analysis of a procedure, a set of candidate mappings are generated for it and stored in the DDT interprocedural database.

For each procedure, DDT can generate two different kinds of solutions: static and dynamic. Static solutions define an initial mapping for each array, and it does not change during the execution of the whole procedure. In a dynamic solution, the statements in the original source code are grouped into a collection of phases; each one may have different mappings for the arrays accessed so remapping operations might be necessary to execute each phase with its mapping. Notice that static solutions are a particular case of dynamic solutions where no remapping operations are needed.

A phase is either the outermost loop in a nest whose control variable is used to subscript an array or a call to a procedure. If the phase is a loop nest, the candidate mappings for it are obtained by performing an analysis of reference patterns within the nest. If the phase is a call, the candidate mappings are imported from the DDT interprocedural database. The control flow module guides the generation of all possible sequences of phases for each procedure.

DDT is targeted to generic NUMA architectures with local and remote accesses. Each processor has its own memory hierarchy and can access the memories in

other processors through the interconnection network. Data movement costs are estimated as the number of remote accesses multiplied by the remote access time. Given a parallelization strategy, computation costs are estimated from a profile of the sequential execution on a workstation based on the same processor and with the same memory hierarchy than the parallel machine.

Profiling the sequential execution of the original Fortran 77 program is required in order to obtain some problem-specific parameters, such as array sizes, the number of iterations for the loops and their execution time, and the probabilities of the different branches in conditional statements. There exists a configuration file that allows the user to specify some machine-specific parameters (number of processors, overhead of parallel thread creation, local and remote memory access costs, and so on) and restricts the kind of solutions explored by DDT (number of distributed dimensions, static or dynamic solutions, number of candidate mappings for the phases and procedures, and so on). All cost estimations in DDT are done numerically assuming the above-mentioned problem and machine-specific parameters.

```
      program adi
        double precision x(256,256)
        double precision a(256,256), b(256,256)

        do 1 i = 1,  256                                        Phase 1
           a(i, 1) = 0.0
           b(i, 1) = 3.0
           x(i, 1) = 4.0
1       continue
        do 2 j = 2, 255                                         Phase 2
          do 2 i = 1,  256
             a(i, j) = 1.0
             b(i, j) = 3.0
             x(i, j) = 5.0
2       continue
        do 3 i = 1,  256                                        Phase 3
           a(i, 256 ) = 1.0
           b(i, 256 ) = 3.0
           x(i, 256 ) = 4.0
3       continue
        do 10 iter = 1, 10
C ADI forward & backward sweeps along rows
        do 4 j = 2,  256                                        Phase 4
          do 4 i = 1,  256
             x(i, j) = x(i, j) - x(i, j - 1) * a(i, j) / b(i, j - 1)
             b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i, j - 1)
4       continue
        do 5 i = 1,  256                                        Phase 5
           x(i, 256 ) = x(i, 256 ) / b(i, 256 )
5       continue
        do 6 j = 255 , 1, -1                                    Phase 6
          do 6 i = 1,  256
             x(i, j) = (x(i, j) - a(i, j + 1) * x(i, j + 1)) / b(i, j)
6       continue
C ADI forward & backward sweeps along columns
        do 7 j = 1,  256                                        Phase 7
          do 7 i = 2,  256
             x(i, j) = x(i, j) - x(i - 1, j) * a(i, j) / b(i - 1, j)
             b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i - 1, j)
7       continue
        do 8 j = 1,  256                                        Phase 8
           x(256 , j) = x(256 , j) / b(256 , j)
8       continue
        do 9 j = 1,  256                                        Phase 9
          do 9 i = 255 , 1, -1
             x(i, j) = (x(i, j) - a(i + 1, j) * x(i + 1, j)) / b(i, j)
9       continue
10    continue
      end
```

FIGURE 1 Source code for ADI.

2.1 An Example: Alternate Direction Implicit

In this section we introduce the Alternating Direction Implicit (ADI) integration kernel to show the main features of the DDT intraprocedural data-remapping module. The source code of ADI defines a two-dimensional data space of size 256 in each dimension; it has a sequence of loops that initializes the data space followed by an iterative loop that performs the computations. In each iteration of this loop, forward and backward sweeps along rows and columns are done in sequence. In this example, and for simplicity, DDT only considers one-dimensional distributions; we have also set the configuration file so that the overhead due to parallel execution is zero, remote accesses take 1 μs per byte, and the parallel machine has 16 processors.

The source code is shown in Figure 1 for completeness. DDT identifies nine phases in this program. Each phase corresponds to one of the nested loops (labeled from 1 to 9) in Figure 1. For each phase, DDT estimates the data movement and the execution costs for different data-mapping and loop parallelization alternatives. For instance, while analyzing phase 4, the two possible mappings shown in Table 1 are taken into account. In this case, DDT suggests a perfect alignment of all the arrays used in the phase and two possible distributions: (*BLOCK*, *) and (*, *BLOCK*). For the first time, DDT also suggests to parallelize the outer *i* loop since there are no data dependencies preventing the loop from running in parallel. For the second alternative, the dependence in the second dimension of arrays b and x forces DDT to sequentialize the execution of the *j* loop.

For each alternative, an estimate of the data movement costs is performed by matching reference patterns within the phase with a predefined set of data movement patterns. The computation time of a phase with a parallelization strategy is estimated from the profile of a sequential execution. For instance, the estimation for phase 4 concludes that three shift-like data movement patterns appear due to the accesses to arrays x and b when the second dimension is distributed (each shift movement involves a column of an array, i.e., 256 elements or 2,048 byte). The profile for this phase reports a sequential execution time of 0.352925 s, which is the cost for Solution 2. However, the computation cost for this phase with Solution 1 is estimated as 1/16 of this sequential execution time plus the overhead due to parallel thread creation (zero in this example). The first row in Table 2 shows the data movement and computation times estimated for phase 4 for the two solutions evaluated by DDT.

From the analysis of the different possible mappings for a phase, a set of them are selected as candidate mappings (this selection is done based on cost

Table 1. Array Mapping Alternatives and Associated Loop Parallelization Strategies Analyzed by DDT for Phase 4 in ADI

	Array Mapping	Loop Parallelization
Solution 1	CHPF$ TEMPLATE target(256, 256) CHPF$ ALIGN WITH target::x, a, b CHPF$ DISTRIBUTE target(BLOCK, *)	CHPF$ INDEPENDENT DO 4 i = 1,256 DO 4 j = 1,256
Solution 2	CHPF$ TEMPLATE target(256, 256) CHPF$ ALIGN WITH target::x, a, b CHPF$ DISTRIBUTE target(*, BLOCK)	DO 4 i = 1,256 DO 4 j = 1,256

criteria). Once they are selected, an algorithm to check the compatibility of phases is used; we say that two phases are compatible when they have preferences for the same data mappings so no remapping is required when sequencing from one phase to the other. For instance, consider the sequence {4, 5, 6} of phases. From the source code in Figure 1, one can see that these three phases have the same preferred mapping (*BLOCK*, *) and loop parallelization strategy (execute the *i* loop in parallel). Similarly, one can conclude that the preferred mapping for each phase in the sequence {7, 8, 9} of phases is (*, *BLOCK*) and the parallelization of the *j* loop. Table 2 shows the costs of the candidate mappings for all the phases within the iterative loop *do iter*.

Table 2 shows that the favorite solution for phases {4, 5, 6} and {7, 8, 9} is not the same. Therefore, they are not compatible in their mapping and parallelization strategies. Three main alternatives are evaluated by the intraprocedural algorithm:

1. Assign Solution 1 to all the phases. In this case the cost per iteration is 0.547944 and the estimated cost for the outer iterative loop 5.47944.
2. Assign Solution 2 to all the phases. In this case the cost per iteration is 0.57559 and the estimated cost for the outer iterative loop 5.7559.
3. Assign the preferred solution to each phase. In this case the computation time is 0.064886 and we have to remap the arrays between incompatible phases. In particular, arrays a, b, and x will be remapped from row to column distribution before the execution of phase 7, which has an approximated cost of $(256 * 256)/16 = 4{,}096$ array elements each (or 32,768 byte each). This remapping action is performed 10 times during the execution of the iterative loop. Due to the same loop, phase 4 is executed again after executing phase 9. So we have to consider the compatibility between these two phases and the possible remapping costs if their mappings are not compatible. In particular, arrays a, b, and x have to be remapped from column to row distribution before the execution of phase 4 with the same estimated cost. This remapping action is performed nine times (since the last iteration of the iterative loop forces the execution to exit it). The total cost for the sequence of phases within the iterative loop is estimated as:

$$(10 * 0.064886) + (10 * 3 * 0.032768) + (9 * 3 * 0.032768) = 2.516636.$$

In this example, the cost of the dynamic alternative is lower than the costs of the two static alternatives. In Section 7, we analyze and evaluate the validity of the different solutions when changing architectural parameters such as data movement and the number of processors.

Table 2. Data Movement and Computation Costs (in seconds) for Phases 4 through 9 in ADI for Two Candidate Solutions

	Solution 1		Solution 2	
Phase	Movement	Computation	Movement	Computation
4	0	0.022058	0.006144	0.352925
5	0	0.000212	0	0.003391
6	0	0.011095	0.004096	0.177513
7	0.006144	0.324565	0	0.020285
8	0	0.002261	0	0.000141
9	0.004096	0.177513	0	0.011095

3 DATA DISTRIBUTION FOR A PHASE

The basic compilation steps for a computational phase are described next. First of all, a weighted graph called the Dimension Alignment Graph (DAG) is constructed from the analysis of the array references in the source program and it records preferences for alignment. The DAG is similar to the CAG but it includes preferences for alignment based on parallelization in addition to data movement. Then, an array alignment phase follows. In this step, all dimensions of the arrays in the program are related to each other by (1) mapping each array dimension into a dimension of a template array (interdimensional alignment) and (2) applying an offset between them (intradimensional alignment). Data movement requirements for those nonaligned references and loop parallelization strategies are analyzed in order to decide the dimensions of the template to distribute, the number of processors allocated, and the kind of distribution applied to them.

3.1 Reference Patterns and the DAG

The DAG is a weighted undirected graph built from the analysis of array reference patterns in loop statements. In this section we review how reference patterns are defined and analyzed to detect affinity, and how the DAG is built from this analysis.

Reference Pattern Analysis and DAG Building

The analysis of reference patterns is performed within the scope of nested loops. A reference pattern is defined by

$$A(i_1, \ldots, i_p, \ldots, i_m) \leftarrow B(j_1, \ldots, j_q, \ldots, j_n).$$

where A is an array that appears in the left-hand side (*lhs*) of an assignment statement located inside the loop and B is an array in the right-hand side (*rhs*) of the same assignment statement. If the assignment is under control of conditional statements, then all the arrays in the expressions that evaluate the conditions are considered as if they were in the *rhs* of the assignment statement.

An affinity relation can appear between two dimensions of the data arrays in a reference pattern. Dimension B_q is said to be affine with dimension A_p (denoted $\langle A_p, B_q \rangle$) if j_q and i_p are linear functions of the same loop control variable.

From the analysis of reference patterns, the DAG is built. Nodes of the DAG represent dimensions of data arrays and edges represent affinity relations between array dimensions obtained by examining cross-reference patterns (patterns in which the *rhs* and *lhs* arrays are different). Self-reference patterns are not considered in the DAG building step. Nodes in the DAG are grouped in columns; each column contains those nodes representing dimensions from the same data array. An edge $\langle A_p, B_q \rangle$ in the DAG shows a preference for alignment of dimensions A_p and B_q.

According to [1], edges in the DAG are weighted in two ways. On the one hand (and to solve the interdimensional alignment problem), each edge is weighted depending on whether it is competing or noncompeting with another edge (ε if it is competing and 1 if not). Two edges are said to be competing if they are generated by the same reference pattern and are incident on the same node. A DAG so defined may contain multiple edges between a pair of nodes since there might be several reference patterns involving two data arrays; each set of multiple edges can be replaced with a single edge whose weight is the sum of their weights. On the other hand (and to solve the intradimensional alignment problem), each edge is weighted with the offset between the two subscripts in the array dimensions involved. In this case, multiple edges are not merged into a single edge because each one may store information about a different shift preference. Preferences for stride alignment are not recorded in the DAG in the current version of the DDT.

DDT also performs a set of well-known optimizations such as expression substitution, subscript substitution, and induction variable detection. In [2] the authors evaluate the effectiveness of these optimizations in terms of amount of new reference patterns analyzed and affinity relations obtained. They also analyze the complexity of the DAG in real codes in terms of number of nodes, edges, and offsets.

3.2 Including Parallelization Constraints in the DAG

Loop parallelization is not independent of the way arrays in the phase are aligned and distributed. It would be interesting to have a clear relationship between loop levels in a phase that is parallelized and dimensions of arrays that are aligned and distributed; this can ease the application of the owner computes rule and of a more efficient generation of parallel code. For each loop in a phase eligible for parallel execution (according to dependence analysis), a set of edges linking dimensions of arrays (in the *lhs* of assignment statements) subscripted by its loop control variable are added in the DAG. Note that these edges are different than the independence antipreference edges defined in [16]. They characterize potentially paralleli-

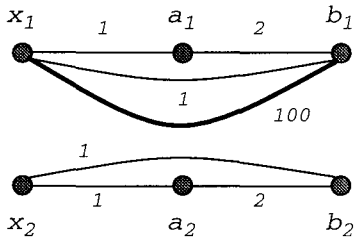


FIGURE 2 DAG for phase 4 in ADI including affinity and parallelization edges.

zable dimensions of arrays (in both sides of the assignment statements) if the subscript is not a scalar. In our DAG, the edges record preferences for alignment of the dimensions that are accessed by a loop eligible to be parallelized.

For instance, consider phase 4 in the ADI program in Figure 1. If the alignment and distribution for arrays `x` and `b` were the following:

```
!HPF$ TEMPLATE target(256, 256)
!HPF$ ALIGN x(i,j) WITH target(i,j)
!HPF$ ALIGN b(i,j) WITH target(j,i)
!HPF$ DISTRIBUTE target(BLOCK,*)
```

then there would not be an easy parallelization strategy for the loop nest. According to the owner computes rule, the *lhs* of the first assignment statement suggests a parallelization of the `i` loop while the second one suggests a parallelization of the `j` loop. If the rule is broken in one of the two statements, additional data movement arises.

Figure 2 shows the DAG for phase 4 in ADI. Notice that in addition to the edges that show affinity between dimensions in array references (thin edges $\langle x_1, a_1\rangle$, $\langle x_2, a_2\rangle$, $\langle a_1, b_1\rangle$, $\langle a_2, b_2\rangle$, $\langle x_1, b_1\rangle$, and $\langle x_2, b_2\rangle$), a new edge between $\langle x_1, b_1\rangle$ is added. This edge has a weight big enough to ensure that the DAG partitioning algorithm described in the next section will align all the nodes linked by it.

3.3 DAG Partitioning and Array Alignment

Two problems are faced when solving the array alignment step. First, the interdimensional alignment problem tries to decide how array dimensions are aligned into the dimensions of a common template array. This template has dimensionality equal to the largest dimensionality of all the arrays analyzed. Each dimension of each array is aligned with a dimension of the template. Second, the intradimensional alignment problem tries to decide how all the array dimensions aligned into a dimension of the template are shifted each other. Although this includes offset and stride alignments and reflections (stride -1), only offset alignments have been implemented in the current version of DDT.

Interdimensional Alignment

Given a DAG G, the interdimensional alignment problem can be stated as follows [1]:

> *Let n be the maximum number of nodes in a column of G. Partition the node set of G into n disjoint subsets $V_1, \ldots, V_n$, with the restriction that no two nodes belonging to the same data array are allowed to be in the same subset.*

Nodes in the same subset correspond to dimensions to be aligned. As a consequence, we want to partition the DAG so as to minimize the total weight of edges that are between nodes in different subsets.

The problem stated above is *NP-complete* and [1] propose a heuristic algorithm (greedy) to solve it. In this algorithm, a single data array is randomly chosen at each step for alignment with the template (which is chosen among the data arrays that have maximum dimensionality). The algorithm applied to a graph G is described below:

```
C_t = Choose_Template_Column(G);
while (not_empty(G)) {
      C_x = Pick_Up_Column(G);
      G_2 = Form_Bipartite_Graph(C_t, C_x, G);
      M = Optimal_Alignment(G_2);
      G = Reduce_Graph(M, C_t, C_x,G);
}
```

In each iteration of the above loop, the alignment between the data array corresponding to column C_x and the data array corresponding to the template column C_t is decided. The main steps of the heuristic are described below:

1. *Form_Bipartite_Graph*, a graph G_2 composed of the nodes in the two columns C_t and C_x is built. An edge is placed between two nodes in the bipartite graph G_2 if there is a path between the two original nodes in the DAG. The weight of the edge is the sum of all edges that compose the path. If several paths appear, then the weight is set to the sum of all the edges that compose the paths.
2. *Optimal_Alignment*. For each bipartite graph G_2, align dimensions of C_x with dimensions of C_t so that the total weight of edges not aligned is minimum.

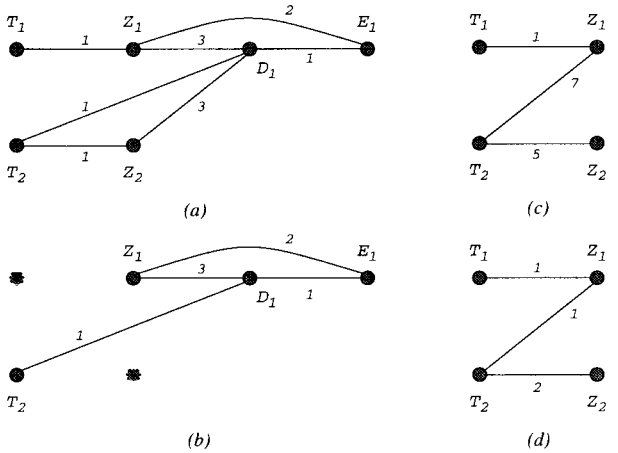


FIGURE 3 Min-cut vs. Sum in Form_Bipartite_Graph.

3. *Reduce_Graph.* Merges column C_x into C_t, replaces multiple edges between two nodes with a single edge whose weight is the sum of their weights, and deletes all self cycles.

In our implementation, several optimizations in the heuristic have been done in order to obtain better alignments. They are described below:

4. In *Form_Bipartite_Graph*, the weight of an edge between two nodes is set to the min-cut instead of the sum of the weight of all edges that compose the paths. With min-cut, the weight is set to the minimum sum of edge weights in G that we had to eliminate to isolate the two nodes. This represents the minimal cost of not aligning the two nodes. For instance, consider the DAG shown in Figure 3a (it corresponds to procedure TRED2 from the EISPACK library). Figure 3b shows a step of *Form_Bipartite_Graph* when a path between T_2 (of C_t) and Z_1 (of C_x in the step) is looked for. In this case, there are two paths: T_2, D_1, Z_1 and T_2, D_1, E_1, Z_1. The bipartite graphs obtained using the original [1] and the min-cut proposals are shown in Figures 3c and 3d, respectively. *Optimal_Alignment* would align $\langle T_1, Z_2\rangle$ and $\langle T_2, Z_1\rangle$ in Figure 3c (with six arcs to communicate) and $\langle T_1, Z_1\rangle$ and $\langle T_2, Z_2\rangle$ in Figure 3d (with one arc to communicate). The solution obtained with min-cut is better than the other solution and better reflects the actual data movement requirements.
5. *Pick_Up_Column* chooses a column C_x among all the columns in G as the column that is more critical in the alignment process instead of an arbitrary column. To decide how critical a column is, we inspect edges between the template and each column in G. For instance, consider the same example in Figure 3. Once *Optimal_Alignment* and *Reduce_Graph* have been done, the graph shown in Figure 4a is obtained. If we pick up column D, the difference between the two possible alignments ($\langle T_1, D_1\rangle$ or $\langle T_2, D_1\rangle$) in the number of communications is 1. On the contrary, if we pick up column E, the difference is 2. So in this case, it is more critical to first solve the alignment of array E rather than array D. Figures 4b and 4c show the difference. The algorithm used to decide that the next column is outlined below:

```
for each Cx in G {
      G2 = Form_Direct_Bipartite(Ct, Cx, G);
      dif(x) = Worst_Alignment(G2)
               - Best_Alignment(G2);
}
Cx = Find_Maximum(dif);
```

Function *Form_Direct_Bipartite* returns the bipartite graph between two columns in a graph that results from direct edges. Functions *Best_Alignment* and *Worst_Alignment* return for a bipartite graph the total weight of nonaligned edges with the best and worst possible alignments.

The use of min-cut increases the execution time of the algorithm with respect to the sum alternative. However, the use of a heuristic to choose the next column decreases the execution time of the min-cut solution because at each step, the complexity of the remaining graph is lower and the algorithm proceeds faster. In [21] the authors evaluate the usefulness of these optimizations oriented toward improving the output of the DAG partitioning algorithm.

Intradimensional Alignment

The algorithm we propose to find shifts among aligned dimensions is described next. For each dimension of the template, a directed graph G_x is created. Nodes in this graph correspond to array dimensions that are

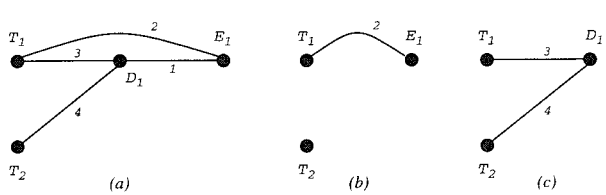


FIGURE 4 Heuristic to choose a new column in the graph for alignment. (a) Intermediate graph. (b) Bipartite graph with domain E. (c) Bipartite graph with domain D.

aligned with a dimension of the template. Edges in G_x are the subset of edges in the DAG between the nodes aligned. In this graph, edges are weighted with the offset between the two subscripts in the associated reference pattern. The algorithm implemented is:

```
for each dimension i of template {
     Gx = Obtain_Directed_Graph(G, i);
     Mark_Template_Node(Gx);
     while (not_all_marked(Gx)){
          N = Pick_Up_Node(Gx);
          S = Find_Shift(Gx, N);
          Gx = Apply_Retiming(Gx, N, S);
     }
     Mark_Node(Gx, N);
}
```

This algorithm is basically the same than the one proposed in [23] to solve the statement alignment problem in order to reduce synchronization costs in a shared memory execution model. The main steps of the algorithm are described below:

1. *Pick_Up_Node.* It returns an unmarked node of G_x connected with a marked node of G_x. If such a node is not found, then an unmarked node is randomly selected.
2. *Find_Shift.* This function returns the offset S (with respect to the template) that has to be applied to the node N currently analyzed. This value is obtained from the offset of all the edges between node N and any marked node of G_x. If several edges between node N and the template node appear, the one that is repeated more times is selected. If several edges are candidates, then the one with minimum value is chosen. We have observed that selecting a value different than zero when zero is one of the candidates leads to poor solutions.
3. *Apply_Retiming.* The idea of retiming as described in [24] is applied in this function. The offset S obtained in the previous step is subtracted to all incoming arcs into node N, and added to all outgoing arcs from node N.

At the end of the algorithm, each node in G_x has an associated shift with respect to the template node.

If G_x is acyclic, a perfect intradimensional alignment results. In this case, all the edges are aligned and no data movement is needed. If cycles are present in G_x, then some edges may not be aligned and therefore, data movement may be required for them.

Table 3. Communication Routines and Their Matching with Reference Patterns

Routine	Pattern
Local_Memory_Access	$i_p = j_p$
Copy	$const(i_p) \wedge const(j_p)$
Shift	$const(i_p - j_p)$
One_to_All	$const(j_p)$
All_to_One	$const(i_p)$
All_to_All	$i_p \neq j_p$

3.4 Communication Analysis

Once data arrays are aligned, each reference pattern that is not aligned (inter- or intracomponent alignment) after the previous phase represents data movement that has to be carried out. In this phase, a matching of reference patterns to a predefined set of data movement routines is done.

In the current implementation, DDT considers simple data movement routines (routines that perform data movement in a single dimension of the template). If the reference pattern requires data movement in more than one dimension, then the reference pattern is decomposed into subpatterns and each subpattern matched with a single data movement routine (each one performing data movement in a single dimension of the arrays).

For each reference pattern (or subpattern if decomposed), the data movement routine that performs the data movement with less cost is chosen. Table 3 shows the set of data movement routines considered by DDT and its matching with reference patterns. In this table, p is the dimension where data movement takes place and i_p and j_p are the subscripts in the dimension p of the reference pattern. Function *const(exp)* returns true if *exp* contains constants only.

The matching between data movement routines and reference patterns is performed in order to obtain an estimation of the overhead due to remote accesses. Each data movement routine has an estimated cost. This cost is dependent on the architecture of the system, the size of the block of data to be transfered, and the number of processors involved in the data movement. The size of the block B is estimated by DDT as follows:

$$B = S \times \prod_{i \neq p} \frac{dim_i}{N_i} \times pos(N_p - 1)$$

S represents the number of elements moved in the dimension p where the data movement takes place. N_i is the number of processors allocated to dimension i

Table 4. Values for S for Each Data Movement Routine

Routine	Value for S
Local_Memory_Access	0
Copy	1
Shift	$\begin{cases} i_p - j_p \text{ if } BLOCK \\ dim_i/N_p \text{ if } CYCLIC \end{cases}$
One_to_All	1
All_to_One	dim_i/N_p
All_to_All	dim_i/N_p

and dim_i is the size of the array in dimension i. Function *pos(const)* returns one if *const* is greater than zero; otherwise it returns zero. Table 4 shows the values of S for each data movement routine listed in Table 3.

If the reference pattern has *const*(f_i), then $dim_i/N_i = 1$, since only one element in that dimension has to be transferred.

Data movement routines involving pairs of processors (*Copy* and *Shift*) are treated like *Local_Memory_Access* when the processor source and target of the data movement are the same. This can be determined by analyzing the subscripts i_p and j_p in the reference pattern. For instance, for the *Copy* data movement and a *BLOCK* distribution, $S = 0$ when the following condition holds:

$$\left\lfloor \frac{i_p + s_p^i}{dim_i/N_p} \right\rfloor = \left\lfloor \frac{j_p + s_p^j}{dim_i/N_p} \right\rfloor$$

where s_p^i and s_p^j are the offsets of the arrays with respect to the template (obtained with the intradimensional alignment step) in dimension p. For a *CYCLIC* distribution, $S = 0$ when

$$(|(i_p + s_p^i) - (j_p + s_p^j)| \; mod \; N_p) = 0$$

3.5 Array Distribution and Loop Parallelization

In order to decide the dimensions of the template that are distributed and the loops that are parallelized, DDT generates a set of valid permutations with different number of processors assigned to each dimension (in powers of two steps). For each permutation, the number of remote accesses and the cost of executing loops in parallel are estimated. The permutation with the lowest overall cost is selected. The algorithm used to perform this is outlined below:

```
for each permutation π {
     if (valid(π)) {
          Decide_BLOCK_or_CYCLIC(π);
          cost_π = 0;
          for each reference pattern ρ {
              if (simple_data_movement(ρ))
                   cost_π += Movement_Cost(π, ρ);
              else
                   cost_π += Movement_Decomposed_Cost(π, ρ);
          }
          cost_π += Computation_Cost(π);
     }
}
Select_Candidate_Permutations(c_π);
```

Function *valid* filters those invalid permutations. Several aspects can be considered to invalidate a permutation, such as the amount of work to be performed by each processor, the amount of parallelism to be extracted according to the distributed dimensions, and load balancing. Function *Movement_Cost* estimates the overhead of performing the data movement specified by pattern ρ with the allocation of processors specified by permutation π. *Movement_Decomposed_Cost* estimates the cost of general patterns that do not match with a simple data movement routine.

Function *Computation_Cost* estimates the cost of executing the loops in the phase in parallel according to the permutation ρ, the data dependences, the owner computes rule (that specifies that loops are parallelized so that each processor executes those iterations that assign values to the array sections it owns), and the

execution time obtained from the sequential execution profile.

Function *Decide_BLOCK_or_CYCLIC* decides whether each distributed dimension in permutation ρ should be distributed in a *BLOCK* or *CYCLIC* manner. The conditions under which a *BLOCK* or *CYCLIC* distribution is preferred for a distributed dimension have been taken from [2]. If the distribution of an array dimension on more than one processor leads to data movement between nearest neighbors in that dimension, then a *BLOCK* distribution is preferred. In this case, a cyclic distribution leads to larger amounts of data being transferred than *BLOCK* distributions. *CYCLIC* distributions lead to a better load balance in some parallelizable computations. In case of triangular loops (i.e., loops whose lower or upper bounds are functions of outer loop control or induction variables), and according to the owner computes rule, it is better to assign the array dimensions accessed with triangular loop control variables in a *CYCLIC* manner rather than in a *BLOCK* manner. The kind of distribution selected in this function influences the movement cost computed in functions *Movement_Cost* and *Movement_Decomposed_Cost* (as shown in Table 4).

4 INTRAPROCEDURAL REDISTRIBUTION

In this section we describe the implementation of the algorithm that performs the intraprocedural data redistribution. For the sake of clarity, we consider the problem when the application is composed of a single module (main program). The presence of procedure calls is described in Section 6. In this section we also consider that a simple control flow between phases exists (phases are executed lexicographically). Section 5 describes the main control-flow structures considered and how they modify the functionality of the main algorithm.

The intraprocedural remapping algorithm implemented in DDT is shown in Figure 5. The main parts of this algorithm are described next.

4.1 Identification of Phases and Iterative Generation of Candidate Mappings

Function *Identify_Phases* tags each loop in the main data structure of DDT as phase or not according to the following definition of phase by [18]:

> *A phase is a loop nest such that for each induction variable occurring in a subscript position of an array reference in the loop body, the phase contains the surrounding loop that defines the induction variable.*

Once phases are defined, procedure *Generate_Candidate_Mappings* generates a set of candidate local mappings for each phase as explained in Section 3 and stores them in the DDT internal data structure. It is important to keep suboptimal solutions in the list of candidate mappings because sometimes it is better to execute a phase with one of them instead of the optimal one. In fact, a solution for a phase is better than another when not only its cost is smaller but also the remapping cost to execute it with the corresponding mapping. Each of the candidate mappings specifies only the relative alignment and distribution between the variables referenced within the phase, but not an absolute alignment over a global virtual template array for the application.

Finally, the static solution cost is computed in function *Static_Solution_Cost*, which is used as the initial lower bound for the remapping process in procedure *Analyze_Compatibility*. The static solution is obtained by applying the algorithm described in Section 3 assuming all loops in the procedure to be a single phase.

4.2 Compatibility between Phases

Procedure *Analyze_Compatibility* builds a search tree composed of the candidate mappings for the different phases in the procedure under analysis. In progressing from one phase to another, we are faced with the problem of deciding which arrays are remapped and which ones are kept with the same mapping. We say that two phases are compatible when they have preferences for the same data mapping. Assume a sequence of phases $\{p_0, p_1, \ldots, p_i, \ldots, p_{n-1}\}$. Each phase p_i has an associated set of n_i candidate local mappings $LM_i^{1..n_i}$. In addition to the local mappings, we have a global mapping GM_i specifying the reaching mapping of all the arrays that have been used until phase p_{i-1} with respect to a global virtual template array. The procedure analyzes the effects of assigning to phase p_i each of its n_i candidate local mappings in the outer *while* statement. Three different alternatives could be considered when analyzing phase p_i with a given local mapping LM_i^k with respect to the global mapping GM_i:

1. Remap all the arrays for which LM_i^k conflicts with GM_i.
2. Remap some of the arrays for which LM_i^k conflicts with GM_i.
3. Do not remap any of the arrays in phase p_i.

```
void Intra_Procedural(procedure_id)
{
      phases_list = Identify_Phases(procedure_id);
      for each phase_id in phases_list {
            Generate_Candidate_Mappings(phase_id);
      }
      phase_id = First_Phase(phases_list);
      initial_cost = Static_Solution_Cost(procedure_id);
      Analyze_Compatibility(phase_id, 0, initial_cost);
}
void Analyze_Compatibility(phase_id, cost, max_cost)
{
      if (phase_id ≠ NIL) {
            for each local_M in phase_id {
                  combinations_list = Analyze_Combinations(global_M, local_M);
                  for each combination in combinations_list {
                        new_cost = cost + Compute_Cost(global_M, local_M, combination);
                        if (new_cost < max_cost) {
                              phase_id = Next_Phase(phases_list, phase_id);
                              Analyze_Compatibility(phase_id, new_cost, max_cost);
                        }
                        if (new_cost < max_cost) {
                              max_cost = new_cost;
                        }
                  }
            }
      }
}
```

FIGURE 5 Intraprocedural remapping algorithm.

The philosophy behind the first alternative is that each phase should be executed with its preferred local mapping and data should be redistributed if necessary before executing the phase. The cost is estimated as the cost of executing phase p_i with LM_i^k plus the cost of remapping. In the second alternative, some of the arrays are not remapped and as a consequence the phase is not executed with the preferred mapping. In addition to the cost of remapping some arrays, the cost of executing phase p_i with a noncandidate local mapping $LM_i^{k'}$ should be evaluated. In the third alternative, the assumption is that it is not worth trying to adapt to the preferred distribution of a phase, and thus the cost of executing phase p_i with mapping GM_i has to be evaluated. The second alternative has not been considered in the implementation since the search space can easily grow to intractable proportions. This alternative is considered by [19], and they realize this problem. One heuristic they propose is to cut the search by limiting the number of arrays that can be simultaneously remapped between two phases.

When trying to adapt the actual global mapping GM_i to a solution LM_i^k in p_i, one of the following actions will take place for each array used in p_i:

1. If the array is not included in GM_i, this means that it has not yet been used in the previous phases and it was not included in the initial global mapping GM_0. As a consequence, this array can be included in GM_i with any desirable mapping, as it will be assumed the initial one.
2. If the array is included in GM_i, and the mapping for it in GM_i and LM_i^k differs either in the number of distributed dimensions or in the dimensions actually distributed, then the array should be redistributed.
3. If the array is already included in GM_i, and its mapping in GM_i has the same distributed dimensions than in LM_i^k (no matter if they are transposed or not), then the array is eligible to be realigned. If only one array fits in this case, then no realignment is necessary. Realignment

is only necessary when two or more arrays in the LM_i^k need a different permutation of their distributed dimensions to fit into the GM_i. In this case we propose to keep one of them by turn in the GM_i, and realign the rest in order to maintain the relative alignment that is specified in the LM_i^k. All these alternatives are stored in *combinations_list*. With each one of these combinations, the algorithm proceeds with the next phase calling recursively to procedure *Analyze_Compatibility*.

Example: Consider the following reaching global mapping GM_i to phase p_i:

GM_i:

A	$1_{B(*)}$	$2_{B(*)}$
B	$1_{B(*)}$	$2_{B(*)}$
C	$2_{B(*)}$	$1_{B(*)}$
D	$1_{B(16)}$	$2*$

The first row reflects that the first and second dimensions of array A are *BLOCK* distributed with four processors allocated to each one. In addition, there is a perfect alignment between the dimensions of array A and the dimensions of the virtual global template (each column of the table represents a dimension of the template). Array B is mapped with the same alignment and distribution as is array A, and array C is transposed with respect to the dimensions of the template. Array D has the first dimension distributed among the 16 processors and the second dimension internalized.

Assume that the following candidate local mapping LM_i^k is suggested for this phase:

LM_i:

A	$1_{B(*)}$	$2_{B(*)}$
B	$2_{B(*)}$	$1_{B(*)}$
C	$1_{B(*)}$	$2_{B(*)}$
D	$1_{B(*)}$	$2_{B(*)}$
E	$1_{B(*)}$	$2_{B(*)}$

If we compare this local mapping with the reaching global mapping, we can see that array D has to be redistributed since the number of distributed dimensions is different. Array E is not yet included in GM_i, therefore it can directly be added. Arrays A, B, and C are candidates to be realigned since the number of dimensions and the dimensions actually distributed are the same. Since arrays B and C have the same relative alignment in both GM_i and LM_i, two different alternatives can be considered:

1. To keep array A as it is in GM_i and to transpose arrays B and C according to LM_i. In this case we would obtain the following global mapping GM_{i+1}:

GM_{i+1}:

A	$1_{B(*)}$	$2_{B(*)}$
B	$2_{B(*)}$	$1_{B(*)}$
C	$1_{B(*)}$	$2_{B(*)}$
D	$1_{B(*)}$	$2_{B(*)}$
E	$1_{B(*)}$	$2_{B(*)}$

2. To keep arrays B and C as they are in GM_i and then transpose array A according to LM_i. In this case all arrays must keep their relative alignment with respect to each other as specified in LM_i. The resulting global mapping GM_{i+1} would be the following:

GM_{i+1}:

A	$2_{B(*)}$	$1_{B(*)}$
B	$1_{B(*)}$	$2_{B(*)}$
C	$2_{B(*)}$	$1_{B(*)}$
D	$2_{B(*)}$	$1_{B(*)}$
E	$2_{B(*)}$	$1_{B(*)}$

Notice that although the second alternative is locally worse (it requires more data movement), it is possible that it leads to a lower overall cost for the whole sequence of phases. So the algorithm must keep it as valid and go deeper in the search tree in order to see which one is the best for the whole sequence of phases.

In the current implementation, the initial global mapping GM_0 is considered empty; however, it is possible to initialize it with either a mapping specified by the user in the source code or a mapping inherited from a caller procedure.

The algorithm in Figure 5 performs a recursive exploration of all different alternatives of candidate solutions $LM_i^{1..n_i}$ for each phase $p_{0..n-1}$ and remapping alternatives between each LM_i^k and GM_i. The actual algorithm reduces the search space based on the cost of the different combinations of all these alternatives. Initially we compute the cost of the static solution. This cost is used to leave the exploration of a (complete or incomplete) sequence of phases if we detect that its current cost is worse than the cost of the static solution. Every time we know the cost of a complete sequence of phases (better than the static solution), its cost is used to update the cost bound for the process.

4.3 Realignment and Redistribution Costs

Remapping costs are estimated by DDT from the specification of the global GM_i and local mapping LM_i^k. Function *Compute_Cost* in Figure 5 creates a dummy self-reference pattern for each array whose mappings GM_i and LM_i^k differ. Then the cost of this reference pattern is estimated by matching it with one of the data movement routines described in Section 3.4.

5 CONTROL FLOW

In this section we describe the aspects that have to be considered when control statements (like conditional or iterative statements) appear in the source code. These statements provoke a sequencing of the phases in the program different than the lexicographical order. In this section we present how the algorithm in Figure 5 is used when these statements appear.

The Phase Control Flow Graph (*PCFG*) is built for each procedure and main program analyzed. In this graph, nodes are phases and edges link nodes when there is a flow of control between the associated phases. There are other nodes in the *PCFG* that represent statements in the source code that provoke changes in the flow of phases. From the information in the control flow graph, the different sequences of phases that might appear during the execution of a procedure are generated. For each sequence, the same algorithm described in Figure 5 is applied.

In the rest of this section we detail how iterative loops and conditional statements are handled by DDT and how they influence the generation of sequences of phases. Other control flow structures (such as entry points and multiple exits) and nesting of all of them are also handled by DDT but are not explained in this article.

5.1 Iterative Loops

Phases might be included within loops whose loop control variables or induction variables generated by them are not used to subscript arrays. In this case, control flow indicates that after executing the last phase inside the loop, the first phase inside it will be executed again.

For instance, Figure 6a shows the control flow graph between the phases that appear in the ADI program shown in Figure 1. Notice that compatibility has to be analyzed between phases {9, 4} since there is a flow of control due to the outer *do iter* loop.

When an outer iterative loop is found in the source code, DDT generates a sequence of phases that try to represent what happens during the actual execution. As shown in Figure 6c, DDT repeats twice the phases in the body of the outer loop. Phases {4, 5, 6, 7, 8, 9} are assumed to be executed once but phases {4′, 5′, 6′, 7′, 8′, 9′} are assumed to be executed N-1 times, where N is the number of times the outer loop is executed. Notice that now possible remapping between phases {3, 4} is accounted for once, remapping between any pair of phases within the loop body is accounted for N times, and remapping between phases {9, 4′} is accounted for N-1 times. The algorithm ensures that the same solution is selected for each pair of phases p_i and p_i'.

If the loop only contains a phase, then it is not necessary to duplicate the phase, since remapping between a phase and itself will never occur.

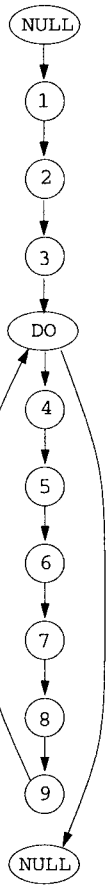
In our running example (ADI), DDT would choose a dynamic data layout that changes twice at every iteration of the iterative loop. An outline of the solution generated by DDT is shown in Figure 6b.

5.2 Conditional Statements

Conditional statements generate alternative phase sequences that are executed depending on the condition evaluated in the statement. The probability of taking one of the alternative branches his initially been obtained by profiling, and it is used to compute the probability of each phase sequence.

The different sequences are analyzed iteratively, starting from the most probable one. Since sequences may have phases in common, different solutions may be suggested for a given phase in different sequences. The algorithm we propose ensures that each phase is always executed with the same solution. To ensure that, the solution for a phase is chosen in the first sequence it appears (i.e., the most probable sequence where the phase is used). Other less probable sequences where the same phase appears have the solution for that phase fixed.

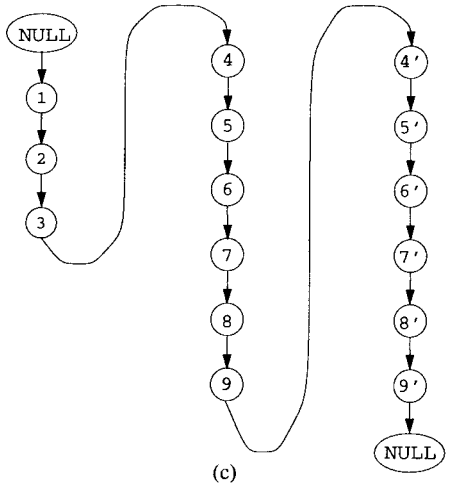
To illustrate this aspect, we analyze the main program in the SPEC swm256 benchmark. As shown in Figure 7a, there is a conditional statement in the main program that selects either the execution of one phase (call to *calc*3) or the execution of another one (call to *calc*3*z*). The control flow (Fig. 7b) for the main program generates two sequences of phases. In this case, if the *then* path has less probability than the *else* path, then compatibility will be first analyzed for phases {1, 2, 3, 5}. From this analysis, a solution among the possible candidate ones is selected for each phase in the sequence. Once selected, compatibilty between phases on the other sequence is analyzed: phases whose mapping has been selected in the previous step remain



(a)

```
   program adi
     double precision x(256,256)
     double precision a(256,256), b(256,256)
CHPF$      TEMPLATE TARGET(256,256)
CHPF$      ALIGN a(i,j) WITH TARGET(i,j)
CHPF$      ALIGN b(i,j) WITH TARGET(i,j)
CHPF$      ALIGN x(i,j) WITH TARGET(i,j)
CHPF$      DYNAMIC, DISTRIBUTE TARGET1(BLOCK,*)
      do 1 i = 1,  256                        Phase 1
                    ...
1     continue
      do 2 j = 2, 255                         Phase 2
        do 2 i = 1,  256
                    ...
2     continue
      do 3 i = 1,  256                        Phase 3
                    ...
3     continue
      do 10 iter = 1, 10
C ADI forward & backward sweeps along rows
CHPF$      REDISTRIBUTE TARGET(BLOCK,*)
         do 4 j = 2,  256                     Phase 4
           do 4 i = 1,  256
                    ...
4        continue
         do 5 i = 1,  256                     Phase 5
                    ...
5        continue
         do 6 j = 255 , 1, -1                 Phase 6
           do 6 i = 1,  256
                    ...
6        continue
C ADI forward & backward sweeps along columns
CHPF$      REDISTRIBUTE TARGET(*,BLOCK)
         do 7 j = 1,  256                     Phase 7
           do 7 i = 2,  256
                    ...
7        continue
         do 8 j = 1,  256                     Phase 8
                    ...
8        continue
         do 9 j = 1,  256                     Phase 9
           do 9 i = 255 , 1, -1
                    ...
9        continue
10    continue
   end
```

(b)

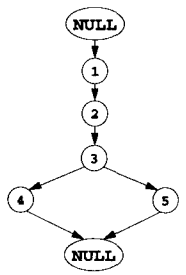


(c)

FIGURE 6 (a) Control flow graph for ADI. (b) Source code for ADI with the directives specifying mapping and remapping of arrays generated by DDT. (c) Sequence of phases analyzed by DDT.

```
program shallow
  ...
  call inital              Phase 1
  ...
  call calc1               Phase 2
  ...
  call calc2               Phase 3
  ...
  if (ncycle .le. 1) then
     call calc3z           Phase 4
  else
     call calc3            Phase 5
  endif
  ...
end
```

(a)



(b)

FIGURE 7 (a) Outline of the main program in the SPEC swm256 benchmark. (b) Control flow graph.

invariant. As a result, a single mapping for phase 4 is selected according to the previously fixed mappings of phases 1, 2, and 3.

6 INTERPROCEDURAL DATA DISTRIBUTION

The main aspects considered in the interprocedural data distribution analysis performed by DDT are described in this section.

The algorithms used both for static data distribution or dynamic redistribution in the case of interprocedural analysis are basically the same as those used for the intraprocedural case. The definition of phase is extended to consider some procedure calls as phases, besides the loops as described in Section 4. If the procedure call is outside a loop which is considered a phase itself, then the procedure is considered a phase. Otherwise, some information obtained from the previous analysis of the called procedure is used to estimate the effects of the mapping of this procedure while deciding the mapping for the phase.

The analysis is based on the call graph in which nodes represent procedures and edges represent call sites. This graph contains representations of the formal and actual parameters and their dimensions associated with each procedure and call site. It is traversed by DDT to decide the order in which procedures will be analyzed. The approach that has been considered is a bottom-up traversal of the call graph: Those procedures that are deeper in the call graph are analyzed first. This bottom-up traversal ensures that when a call to a procedure is found then it has already been analyzed, and therefore the information for that procedure is already in the interprocedural DDT database.

When a mapping specified for a dummy argument or global variable differs from its actual argument or global variable, HPF requires implicit data redistribution and/or realignment. Additionally, upon return to the caller program, the original mapping must be reestablished. So two possible remappings for each argument and global variable should be considered. Other programming models based on HPF, such as the one offered by FORGE [11], allow you to leave an output mapping different than the input one. In this case, the information stored in the internal DDT database after deciding the mapping for the procedure will be its initial and final mapping (in addition to the realignment actions performed inside it and its execution cost with the selected strategy). Notice that this later model is a generalization of the definition of HPF. DDT actually supports both alternatives.

A procedure may have several candidate mappings stored in the interprocedural database. If different mappings are preferred in different invocations of the same procedure, then procedure cloning is applied in order to generate versions of the same procedure with different mapping and parallelization alternatives.

6.1 The Procedure Call is a Phase

A procedure call is considered a phase when it is not placed inside a loop tagged as a phase. The remapping algorithm used in this case is mainly the one described in Section 4.1. However, function *Generate_Local_Mappings* reads the internal DDT database in order to obtain the different candidate mappings for the called procedure, instead of computing them from scratch.

Phases due to procedure calls have candidate mappings composed of an initial and a final mapping. In this case, the cost of remapping will be determined by the mapping differences between the actual global mapping and the corresponding initial mapping for the phase. However, after the execution of the phase, the global mapping will be updated with the final mapping. This means that when generating different permutations of the local mapping in order to estimate different realignment options, both the initial and the final mappings should be permuted.

This analysis could be extrapolated and used to analyze any kind of phase (loop and call), assuming that the local mapping for a loop has the same initial and final mappings, whereas the call has its corresponding initial and final mappings.

The next example is used to illustrate the aspects described above. Assume that some phases of a procedure have been analyzed, and that at this point the global mapping GM_i contains the following information:

GM_i:

A	$1_{B(4)}$	$2_{B(4)}$
B	$1_{B(4)}$	$2_{B(4)}$
C	$1_{B(4)}$	$2_{B(4)}$
D	$1_{B(4)}$	$2_{B(4)}$

This means that in the global mapping GM_i, all arrays used before that phase are perfectly aligned, and their first and second dimensions are distributed with four processors assigned to each one. Assume also that the next phase to be analyzed is a procedure call whose local mappings in LM_{i+1} (obtained from the interprocedural database) are the following:

Initial LM_{i+1}:

A	$1_{B(4)}$	$2_{B(4)}$
B	$2_{B(4)}$	$1_{B(4)}$
C	$2_{B(4)}$	$1_{B(4)}$
E	$2_{B(4)}$	$1_{B(4)}$

Final LM_{i+1}:

A	$1_{B(4)}$	$2_{B(4)}$
B	$1_{B(4)}$	$2_{B(4)}$
C	$1_{B(4)}$	$2_{B(4)}$
E	$1_{B(4)}$	$2_{B(4)}$

When analyzing this phase, arrays used in this phase must be tagged according to the distribution differences between the global mapping and the initial local mapping. We can see that array E is new, so it will be included in the global mapping with its desired distribution (note that it will also be included in the initial mapping for this procedure). Arrays A, B, and C are candidates to be realigned. Two different alternatives should be considered: to keep array A as it is and realign arrays B and C; or to keep arrays B and C as they are in the global mapping and realign array A.

In order to update the global mapping GM_{i+1}, the final local mapping is the one that must be taken into consideration. This means that if the first alternative is selected, array A is kept as it is in the global mapping and neither the initial local mapping nor the final one is transposed. So the global mapping will remain:

GM_{i+1}:

A	$1_{B(4)}$	$2_{B(4)}$
B	$1_{B(4)}$	$2_{B(4)}$
C	$1_{B(4)}$	$2_{B(4)}$
D	$1_{B(4)}$	$2_{B(4)}$
E	$1_{B(4)}$	$2_{B(4)}$

But if the second alternative is selected, then arrays B and C are kept as they are in the global mapping so both the initial and the final local mappings are transposed. In this case the global mapping will be:

GM_{i+1}:

A	$2_{B(4)}$	$1_{B(4)}$
B	$2_{B(4)}$	$1_{B(4)}$
C	$2_{B(4)}$	$1_{B(4)}$
D	$1_{B(4)}$	$2_{B(4)}$
E	$2_{B(4)}$	$1_{B(4)}$

Note that the mapping for all arrays (except for D) in GM_{i+1} in this last alternative is different (transposed) from their mapping in GM_i, and apparently only array A has been realigned. Attention must be paid to the local mappings of phase p_{i+1}. The difference between the initial local mapping and the final one means that, at least, arrays B, C, and D have been realigned inside the procedure, and thus their cost has already been assumed within the cost of executing the procedure. If the first alternative is selected, then arrays B and C are realigned before the procedure call and inside it as well, so the GM_{i+1} remains unchanged with respect to GM_i. At this point it is not possible to say which alternative is the best because it depends on the phases not yet analyzed, so the analysis must continue with the two alternatives.

6.2 The Procedure Call is Inside a Phase

When the procedure call is placed inside a loop which is tagged as a phase, then the call is not considered a phase. In this case, the initial and final mappings assigned to the procedure may affect the choice of the candidate mappings for the phase. When a call statement is found, DDT imports from the interprocedural DDT database all the information associated to each possible candidate mapping for the called procedure.

During the alignment step, when a call statement is found, DDT imports from the corresponding file in the interprocedural DDT database the information regarding ALIGN directives for the global variables and the actual parameters. This information is included in the DAG of the phase as additional edges. In fact, this is an approximation to the problem. A more accurate model should have to weight these new edges with its corresponding realignment cost.

During the distribution step, and for each call to a procedure, the global variables and the actual parameters must be remapped (if necessary) before and after

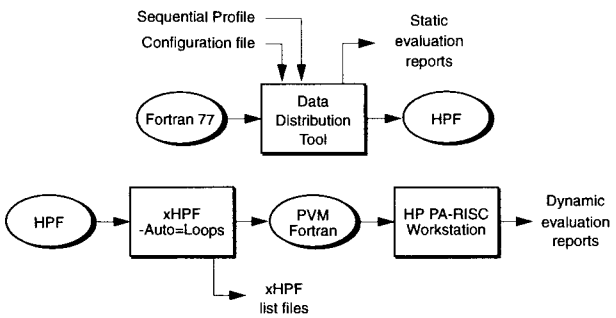


FIGURE 8 Main components of our automatic data distribution platform: DDT, xHPF compiler, and simulator from APR, Inc.

the call. The mapping that minimizes the overall cost including remapping is selected among the candidate ones.

7 EXPERIMENTAL RESULTS

The main components of the data distribution environment we are using and developing are shown in Figure 8. Our research tool (DDT) is implemented on top of ParaScope [25] and assumes sequential programs written in Fortran 77 as input. DDT parses the input code and annotates it with a set of HPF directives and executable statements.

The xHPF compiler from Applied Parallel Research [11] is used to compile the program generated by DDT and to generate a single-program multiple-data (SPMD) node program using PVM3 communication primitives [26]. The xHPF execution model allows us to simulate the execution of the instrumented code generated by xHPF on a single workstation. This simulated execution is used to perform comparisons of the performance of different data distribution strategies and to validate our proposals.

We have analyzed three programs: ADI (shown in Fig. 1 and analyzed in Section 2), routine RHS from the APPBT, APPLU, and APPSP NAS benchmarks, and swm256 from the xHPF benchmarks set.*

We will see for the examples how changes in some architectural parameters, such as number of processors and remote access time, lead to changes in the solution generated by DDT. The tool is useful for the characterization of programs as well as the study of the effects of these architectural parameters.

* Available by anonymous ftp at ftp.informall.org in directory tenants/apri/Bench.

7.1 Alternate Direction Implicit ADI

In this section we further analyze ADI and compare the performance predicted by DDT against the performance obtained when simulating the execution of the message-passing code generated by xHPF. In addition, we also show the usefulness of the tool to predict the performance of different mapping strategies when changing architectural parameters.

Figure 9 shows the predicted and the measured speedups for the program for different numbers of processors (ranging from 1 to 32) for two possible solutions: The static solution where all arrays are column distributed (adi2 in all plot labels) and the dynamic solution (adiD in all plot labels) as shown in Section 2. For this plot we considered a remote access time of 1 μs. We can draw the following conclusions:

1. The prediction performed by DDT (solid lines) is very close to the actual speedup (dashed lines). In the dynamic solution we have noticed a small difference due to the estimation of redistribution costs. The model that we consider to estimate these costs (see Section 4.3) is not very accurate and overestimates the number of data elements moved.
2. The speedup of the static solution grows from one (for one processor) to two (for machine configurations with a large number of processors). This is due to the fact that about one half of the program is executed in a synchronized way with an execution time close to the sequential execution time.
3. The speedup of the dynamic solution is lower than one for configurations with less than four processors but then grows with an efficiency close to four. This is due to the fact that remap-

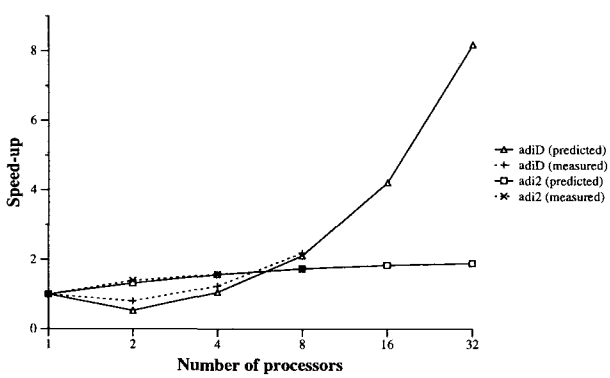


FIGURE 9 ADI—speedup vs. number of processors for the static and dynamic solutions. Comparison of the predicted and measured speedup (remote access time = 1 μs).

Table 5. Breakdown of the Total Execution Time for ADI (Remote Access Time = 1 μs)

	Static Solution			Dynamic Solution		
Num_Procs	Movement	Computation	Total	Movement	Computation	Total
1	0	10.515	10.514	0	10.680	10.680
2	0.1024	7.909	8.011	14.942	5.340	20.282
4	0.1024	6.606	6.709	7.471	2.670	10.141
8	0.1024	5.955	6.058	3.735	1.335	5.070
16	0.1024	5.629	5.732	1.867	0.667	2.535
32	0.1024	5.467	5.569	0.933	0.372	1.372

ping costs are very large when a small number of processors are available and that all phases in the program are executed in parallel.

4. With this remote access time, DDT chooses the static solution for less than eight processors and the dynamic solution when eight or more processors are available.

To further compare the dynamic and static solutions, Table 5 shows the breakdown of the predicted execution time in computation and data movement times. Notice that for the static solution, the data movement overhead is constant (it is due to shifts where the number of elements moved is independent of the number of processors). However, in the dynamic solution the redistribution overheads decrease with the number of processors involved in the data movement.

Figure 10 shows the predicted speedup when the remote access time changes for the static and dynamic solutions. The aim of this graph is to show the influence of remote access latencies in these solutions. In this plot we assume that the number of processors is 16. The following conclusions are drawn:

1. For very low access latencies, the speedup tends to be in 16 in the dynamic solution and 2 in the static solution.
2. The static solution is less sensitive to the memory latency than the dynamic solution. This is due to the fact that the volume of data transferred in the static solution is small while in the dynamic solution it is large. For large latencies, any gain due to parallel execution is offset by the data movement overhead.
3. For this number of processors, DDT chooses the dynamic solution when the remote access time is less than 5 μs and the static solution otherwise.

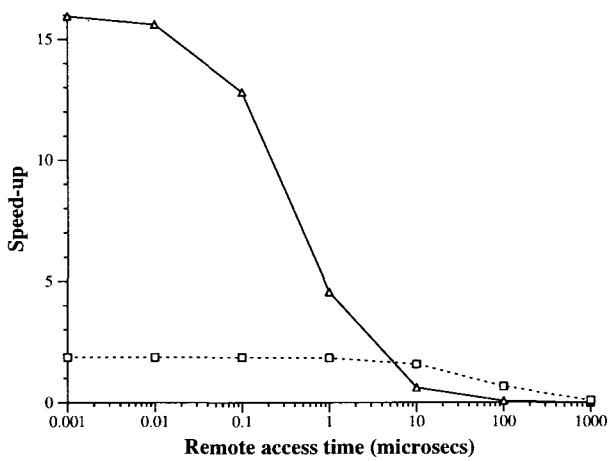


FIGURE 10 ADI—predicted speedup vs. remote access time for the static and dynamic solutions. Δ, adiD; □, adiZ.

7.2 rhs Routine from NAS and swm256 Benchmark

In this section we analyze the behavior of the solution suggested by DDT for the other two benchmarks: the rhs routine from NAS and the swm256 program. For each of them we compare the performance predicted by DDT against the performance obtained in the simulated execution. We assume that the remote access time is 1 μs and that the system has from one to eight processors.

Figure 11a shows the behavior for rhs. In this case DDT suggests a dynamic solution where three arrays have to be remapped. The dynamic solution implies that the outer loop in each phase runs in parallel. In this case the prediction is close to the actual performance because DDT performs an accurate estimation of both data movement and parallel computation times.

Figure 11b shows the behavior for swm256. In this case DDT suggests a static solution where all the arrays are distributed by columns. This static solution implies that almost all the loops run in parallel. The main

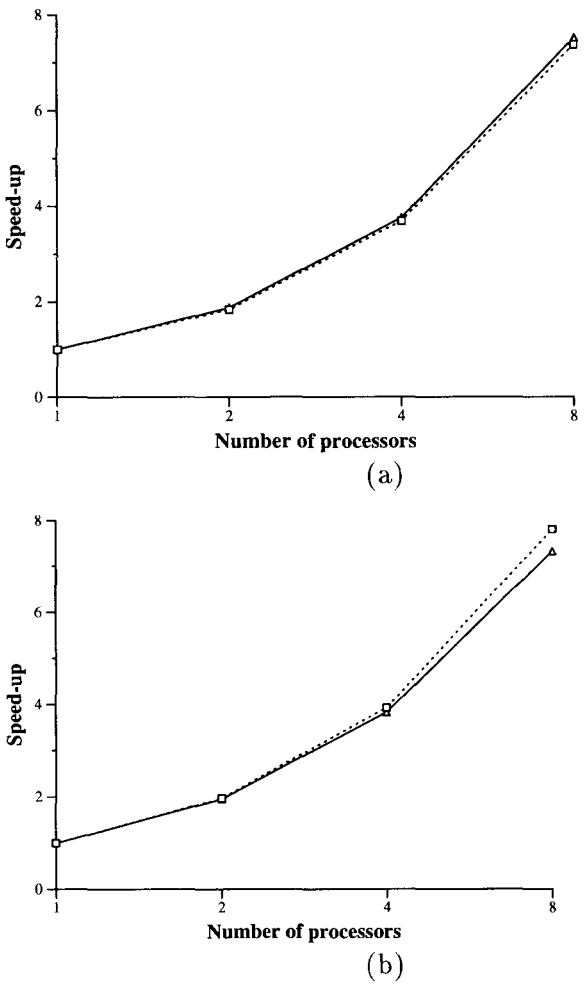


FIGURE 11 Predicted and measured speedup vs. number of processors for (a) rhs—dynamic solution. Δ. rhs (predicted); □. rhs (measured). (b) swm256—static solution, assuming remote access time = 1 ms. Δ. shallow (predicted); □. shallow (measured).

program in swm256 includes an iterative loop and conditional statements that validate the correct behavior and estimations of the control flow module in DDT. Notice a small difference between the predicted and measured speedups. The difference is due to an overestimation of the data movement overhead: to get a better estimation, we have to improve the module that detects redundant data motion either within a phase or between phases.

8 CONCLUSIONS AND REMARKS

In this article we have presented the key modules in our automatic DDT. DDT generates both static and dynamic HPF data distributions for a given Fortran 77 routine and for a whole application with interprocedural analysis. In the static solutions, the mapping (alignment and distribution) of each array in the program does not change during the execution. The static module is based on the CAG but is extended with some information regarding parallelism. We have also modified the original algorithms in [1, 17] to improve the quality of the mappings generated [21].

Dynamic solutions include executable statements in the source code that change the mapping of specific arrays when necessary between computational phases. DDT performs a cost analysis of profitability in order to include them. This analysis of profitability is based on the following steps:

1. Detection of phases or computationally intensive portions of code, which mainly correspond to nested loops and calls to procedures. Remapping is only allowed between phases.
2. Generation of candidate mappings for the previously detected phases and estimation of their cost (including data movement and execution time costs).
3. Analysis of compatibility among phases, selection of mappings for them, and remapping actions to be performed between consecutive phases. This selection is done by analyzing the cost in terms of data movement due to redistribution and its benefits in the cost of successive phases.

Control flow information is used to identify sequencing of phases. The algorithm explores a rich set of combinations although it is not exhaustive. It includes mechanisms to cut down the search space.

DDT is a research tool which is currently used in our group to support different research aspects. Since it is a research tool, it can use techniques that may be too computationally expensive to be included in a final compiler; however, this allows us to explore a rich set of solutions.

We have evaluated the quality of the solutions generated by DDT by comparing predicted performance against the actual performance when the parallel program is executed. We have also shown the usefulness of the tool for the characterization of the programs as well as the study of the effects of architectural parameters. We have shown how the predicted speedups are close to the actual ones obtained when the program is executed. DDT also accepts HPF directives in the source Fortran 77 program; in this case DDT is useful as a support tool for the developer of HPF codes in estimating the effect of user-selected data

mappings and parallelization strategies in the final performance of the parallel program.

We are currently porting this technology to generate efficient code for hierarchical global shared memory architectures. In these architectures a number of central processing units can simultaneously access data anywhere in the system. However, the nonuniformity of the memory accesses is still an important issue to consider and may require a higher programming effort in order to achieve performance; trying to access those levels in the hierarchy closer to the processor will increase execution efficiency. The technology developed to study the profitability of dynamic data remapping can be used to track the movement of data during program execution and thus parallelize loops accordingly, so that the access to data is done locally as much as possible.

ACKNOWLEDGMENTS

This work has been partially supported by CONVEX Computer Corporation, CONVEX Supercomputers S.A.E, CEPBA (European Center for Parallelism of Barcelona), and by the Ministry of Education of Spain under contract TIC-429/95. We thank Miquel Huguet from CONVEX Supercomputer S.A.E, Robert Metzger from CONVEX Computer Corporation, and the anonymous reviewers for their constructive comments. We also thank Jordi Torres for his help in the implementation of the interprocedural driver.

REFERENCES

[1] J. Li and M. Chen, "Index Domain alignment: Minimizing cost of cross-referencing between distributed arrays," presented at Frontiers90: 3rd Symp. on the Frontiers of Massively Parallel Computation, College Park, MD, 1990.

[2] M. Gupta, "Automatic data partitioning on distributed memory multicomputers," PhD thesis, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1992.

[3] M. Gupta, S. Midkiff, E. Schonberg, P. Sweeney, K. Y. Wang, and K. Burke, "*PTRAN* II - A compiler for high performance fortran," in H. J. Sips, Ed., *Proceedings of the 4th Workshop on Computers for Parallel Computers*. The Netherlands: Delft University of Technology, pp. 479–493, 1993.

[4] T. J. Scheffler, R. Schreiber, J. R. Gilbert, and S. Chatterjee, "Aligning parallel arrays to reduce communication," presented at the Frontiers95: The 5th Symp. on the Frontiers of Massively Parallel Computation, McLean, VA, 1995.

[5] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler, "Array distribution in data-parallel programs," in K. Pingali et al., Eds., *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 892. New York: Springer-Verlag, pp. 76–91, 1994.

[6] U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle, "Automatic data layout for distributed-memory machines in the D programming environment," presented at the 1st Int. Workshop on Automatic Distributed Memory Parallelization. Automatic Data Distribution and Automatic Parallel Performance Prediction, Saarbruecken, Germany, 1993.

[7] B. Chapman, T. Fahringer, and H. Zima, "Automatic support for data distribution on distributed memory multiprocessor systems, in U. Banerjee et al., Eds., *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 768. New York: Springer-Verlag, pp. 184–199, 1993.

[8] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran," *Sci. Prog.* vol. 1, pp. 31–50, 1992.

[9] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling Fortran-D for MIMD distributed-memory machines," *Commun. ACM*, vol. 35, pp. 66–80, Aug. 1992.

[10] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, "Fortran D language specification," Department of Computer Science, Rice University, Houston, TX, Tech. Rep. CRPC TR 90-141, Dec. 1990.

[11] Applied Parallel Research, *xhpf Version 2.0, User's Guide*. Placerville, CA: APR, 1995.

[12] The Portland Group, *PGHPF - Reference Manual*. Portland, OR: Portland Group, 1994.

[13] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, and M. E. Zosel, *The high Performance Fortran Handbook*. Number 1-2 in Scientific and Engineering Computation Series. Cambridge, MA: MIT Press, 1994.

[14] S. Wholey, "Automatic data mapping for distributed-memory parallel computers, in *Proc. of the ACM Int. Conf. on Supercomputing*, pp. 25–33, 1992.

[15] K. Kennedy and U. Kremer, "Automatic data layout for high performance Fortran. Center for Research on Parallel Computation, Rice University, Houston, TX, Tech. Rep. CRPC-TR94498-S, Dec. 1994.

[16] K. Knobe, J. D. Lukas, and G. L. Steele, "Data optimization: Allocation of arrays to reduce communication on SIMD machines," *J. Parallel Distrib. Comput.*, vol. 8, pp. 102–118, Feb. 1990.

[17] J. Li and M. Chen, "Compiling communication-efficient programs for massively parallel machines," *IEEE Trans. Parallel Distrib. Systems*, vol. 2, pp. 361–375, July 1991.

[18] R. Bixby, K. Kennedy, and U. Kremer, "Automatic data layout using 0-1 integer programming," in *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 111–122, 1994.

[19] P. Crooks and R. H. Perrott, An automatic data distribution generator for distributed memory MIMD machines," in H. J. Sips, Ed., *Proc. of the 4th Int. Workshop on Compilers for Parallel Computers*. The Netherlands: Delft University of Technology, pp. 33–44, 1993.

[20] D. J. Palermo and P. Banerjee, "Automatic selection of dynamic partitioning schemes for distributed-memory multicomputers," in C.-H. Huang et al., Eds., *Proc. of the 8th Annual Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 1033. New York: Springer-Verlag, pp. 392–406, 1995.

[21] E. Ayguadé, J. Garcia, M. Gironès, J. Labarta, J. Torres, and M. Valero, "Detecting and using affinity in an automatic data distribution tool," in *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computing*, K. Pingali et al., Eds. Lecture Notes in Computer Science vol. 892. New York: Springer-Verlag, 1994, pp. 61–75.

[22] E. Ayguadé, J. Garcia, M. Gironès, M. L. Grande, and J. Labarta, "Data redistribution in an automatic data distribution tool, in *Proceedings of the 8th Annual Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 1033. New York: Springer-Verlag, pp. 407–421, 1995.

[23] J. Peir, "Program partitioning and synchronization on multiprocessor systems," PhD thesis, University of Illinois at Urbana-Champaign, 1986.

[24] C. Leiserson, F. Rose, and J. Saxe, "Optimizing synchronous circuitry by retiming," presented at the 3rd Caltech Conference on VLSI, CA, 1983.

[25] K. Kennedy, K. McKinley, and C-W. Tseng, "Interactive parallel programming using the ParaScope editor," Center for Research on Parallel Computation, Rice University, Houston, TX, Tech. Rep. CRPC-TR90096, Oct. 1990.

[26] A. Gueist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM3 user's guide and reference manual," Oak Ridge National Laboratory, Tech. Rep. ORNL/TM-12187, May 1993.