Tools and Techniques for Automatic Data Layout: A Case Study

Eduard Ayguadé Jordi Garcia

Computer Architecture Department
Universitat Politècnica de Catalunya

Ulrich Kremer

Department of Computer Science
Rutgers University

Abstract

Parallel architectures with physically distributed memory providing computing cycles and large amounts of memory are becoming more and more common. To make such architectures truly usable, programming models and support tools are needed to ease the programming effort for these parallel systems. Automatic data distribution tools and techniques play an important role in achieving that goal. This paper discusses state–of–the–art approaches to fully automatic data and computation partitioning. A kernel application is used as a case study to illustrate the main differences of four representative approaches. The paper concludes with a discussion of promising future research directions for automatic data layout.

1 Introduction

Shared and distributed–memory multiprocessors, networks of workstations, and machines connected through the Internet provide a variety of computational and other resources such as a large amounts of main memory or fast I/O capabilities. Using a parallel platform instead of a uniprocessor is often the only choice for running computation or resource intensive applications. In fact, some researchers argue that machines with a large main memory should always have multiple processors in order to make cost–effective use of the memory's capacity and bandwidth [WH95]. If the current trend can be used as an indication for future developments, parallel platforms will soon become ubiquitous.

A significant step towards ease of use of parallel platforms and program portability across platforms has been the development of shared name space programming models with or without explicit parallelism. The advantage of a single name space is that the user does not have to explicitly manage processor local name spaces, local computations, and necessary communication between processors. Shared name space models can be supported in hardware (shared memory [Ost89, FFG+95]), through a combination of hardware and operating system layers (distributed shared memory [LLG+90, KCDZ94, IDFL96, KHS+97]), or mostly through compiler technology (distributed memory [KLS+94, Tse93, CGC96]). All shared name space implementations have in common that in order to achieve good and scalable performance most computations have to be local, i.e., each processor accesses data already available in its local memory or local cache. An efficient data and computation mapping ensures locality of computations.

This paper focuses on state–of–the–art compile time approaches to fully automatic data and computation mappings for distributed memory and distributed shared memory architectures. The main application domain of the discussed approaches are regular, scientific programs. Nearly all scientific codes use arrays as their main data structures with most computations performed in loop nests iterating over the arrays. Regular problems use dense arrays allowing the compiler to determine data access and computation patterns at compile time. Based on these patterns, the compiler chooses a data mapping that assigns array elements to local memories. The determined computation mappings consist of assigning loop iterations to different processors. For physically distributed memory machines, a subsequent compilation stage has to translate global data references into local and non–local references based on the determined mappings. All non–local references must be satisfied through communication.

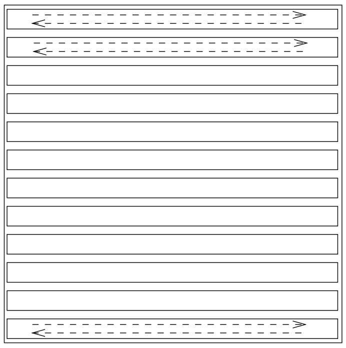
The choice of a good data and computation mapping is important as it determines the amount of remote data accesses and the potential parallelism in the resulting program. Due to its influence on the amount of inter–processor communication, the selection of a data and computation mapping has a significant impact on the performance of the parallel program. An efficient mapping depends on many complex factors such as the program structure, the compiler capabilities, the characteristics of the target machine, and the program's data sizes. In addition, there is often a trade–off between minimizing interprocessor data movement and load balancing on processors. Crucial aspects such as data movement, parallelism, and load balance have to be taken into consideration in a unified way to efficiently solve the data and computation mapping problem.

Four approaches have been selected to show different strategies for automatic data and computation mappings as they have been discussed in the literature. The goal of the four strategies is to determine mappings that minimize inter–processor communication while maximizing parallelism. Approaches that in addition may try to reduce cache traffic are beyond the scope of this paper. The purpose of this paper is not to give an exhaustive overview of all strategies, but to discuss a few representative strategies in more detail using a small benchmark program as an example application. The program used as a case study is described in Section 2. A summary of the most relevant contributions in the field is presented in Section 3. The four approaches are outlined in Section 4. Finally, Section 5 analyzes the approaches presented in Section 4, and describes some of the key issues that still require further research effort.

2 Alternating Direction Implicit (ADI) Integration Kernel

The following example program kernel will be used to illustrate different approaches to automatic data and computation mapping discussed in the literature. Figure 1 shows an Alternating Direction Implicit (ADI) integration kernel. ADI integration is a technique frequently used to solve partial differential equations (PDEs).

```
REAL c(N, N), a(N, N), b(N, N)

// READ (c, a, b)

DO   iter = 1, max    //  << loop 1 >>
    // Forward and backward sweeps along rows

    DO j = 2, N    //  << loop 2 >>
        DO i = 1, N
            c(i, j) = c(i, j) - c(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)
        ENDDO
    ENDDO

    DO i = 1, N    //  << loop 3 >>
        c(i, N) = c(i, N) / b(i, N)
    ENDDO

    DO j = N - 1, 1, -1    //  << loop 4 >>
        DO i = 2, N
            c(i, j) = ( c(i, j) - a(i, j + 1) * c(i, j + 1) ) / b(i, j)
        ENDDO
    ENDDO

    // Downward and upward sweeps along columns

    DO j = 1, N    //  << loop 5 >>
        DO i = 2, N
            c(i, j) = c(i, j) - c(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)
        ENDDO
    ENDDO

    DO j = 1, N    //  << loop 6 >>
        c(N, j) = c(N, j) / b(N, j)
    ENDDO

    DO j = 1, N    //  << loop 7 >>
        DO i = N - 1, 1, -1
            c(i, j) = ( c(i, j) - a(i + 1, j) * c(i + 1, j) ) / b(i, j)
        ENDDO
    ENDDO

ENDDO

// WRITE (c, b)
```



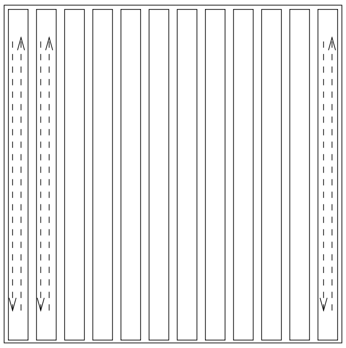


Figure 1: ADI integration kernel with computation illustration

The execution of the ADI integration kernel consists of a repeated sequence of forward and backward sweeps along rows, followed by downward and upward sweeps along columns. For the sweeps along the rows, a row layout has the best performance. The same holds for a column layout for the column sweeps. Transposing dynamically the arrays between the all row and all column sweeps eliminates communication within the sweeps, but requires data reorganization between the sweeps. In contrast, choosing the same data layout for both, row and column sweeps, will avoid communication between the sweeps but will make communication necessary either in the row or column sweep. Finally, the static distribution of rows and columns (two–dimensional distribution) over a squared processors grid, communicates along both the row and column sweeps but the number of elements to communicate is lower.

The best data layout choice depends on several characteristics of the target distributed–memory machine, as well as on the ability of the compiler to exploit parallelism efficiently. The actual size N of the arrays may also influence the optimal data layout choice. In addition, the performance characteristics of the underlying I/O system has to be considered since the program performs read and write disk accesses.

Figure 2.a shows the estimated speed–up for the ADI integration kernel assuming different bandwidths for a target architecture with 16 processors. This has been computed for a static column layout, a dynamic layout, and a two–dimensional layout. Note that when the the bandwidth is higher than 2Mbytes/second the dynamic layout becomes more efficient than the other.

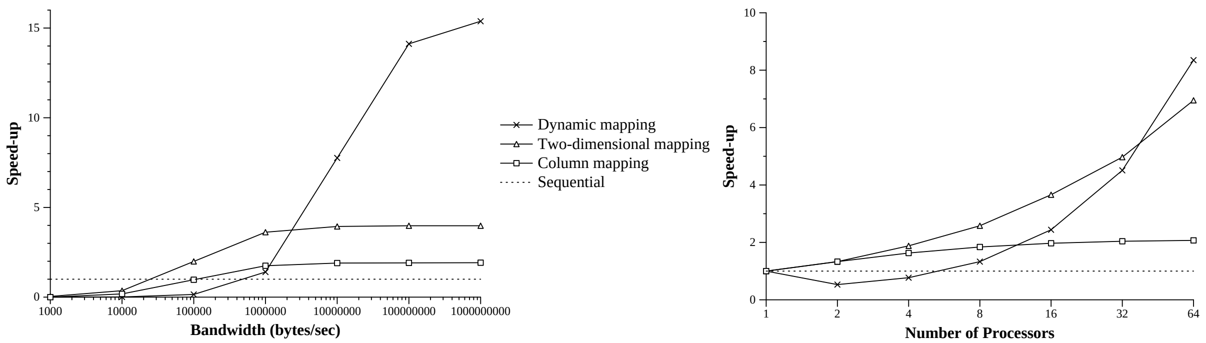


Figure 2: Estimated speed–up for ADI with different (a) bandwidth, and (b) number of processors.

Similarly, Figure 2.b shows the estimated speed–up for ADI assuming a target system with a bandwidth of 1.5 Mbytes/second and changing the number of available processors. Note that the dynamic and two–dimensional layouts are more sensitive to the number of processors than the column layout. In addition, the optimal solution with 32 processors is different than the one with 64 processors.

3 Approaches to Automatic Data Mapping

Data mapping allows the user to specify the compiler how to allocate data objects over the memory of the logical arrangement of processors in physically distributed memory architectures. When there is a single mapping for the whole program, then the solution is said to be *static*. However, in large programs where several computationally intensive code blocks occur, remapping actions between them may improve the efficiency of the solution. In this case the solution is said to be *dynamic*.

Automatically deciding the data decomposition for a program is usually performed in two main steps: *alignment* and *distribution*. The alignment step tries to relate the dimensions of different arrays according to the access patterns between them. A good alignment will minimize the overall overhead of inter–processor data movement. Once the alignment has been performed, the distribution step decides which of the aligned dimensions are distributed, the number of processors assigned to each distributed dimension, and the distribution fashion, i.e., `BLOCK` or `CYCLIC`. A good distribution maximizes the potential parallelism of the application, balance the computational load, and offers the possibility of further reducing data movement by serializing.

There is a large number of researchers that have addressed the problem of automatic data distribution in the context of regular applications. The main differences between the proposed methods is the kind of structure selected to represent the problem, the cost model adopted, and the set of techniques used to find a solution. This section gives an overview of the major contributions related to the automatic generation of data mappings.

3.1 Alignment Problem

An array alignment specifies relationships among array dimensions. The problem of deciding the relative orientation of arrays in a static data mapping has two components: inter–dimensional and intra–dimensional. Inter–dimensional alignment is concerned with permuting the array dimensions with respect to each other. Intra–dimensional alignment allows, for each aligned dimension, shifting, striding, and reversing the ordering. Array alignment requirements are induced by the characteristics of the computations in which the arrays are referenced. These requirements can be expressed mathematically through a set of equations or algebraic formulation, or can be represented as a weighted graph problem.

Ramanujam and Sadayappan [RS89] use matrix notation to represent a communication–free static alignment, and solve the problem with linear algebra. However in real codes, a communication–free alignment can be rarely found. Bau et al. propose different problem formulations to minimize communication costs when communication–free partitioning is not possible [BKK$^+$94a]. Anderson and Lam [AL93, And97a] combine mathematical and graph based problem representations to find a minimal communication cost alignment.

However, the majority of the approaches use a weighted graph based formulation. The advantage of a graph based formulation is that it allows an easier representation of tradeoff decisions between communication and exploitable parallelism. Typically, the alignment problem is solved as a minimal cost graph partitioning problem. Most of the graph based approaches have been inspired by the original proposals of Li and Chen [LC90], and Knobe, Lucas, and Steele [KLS90].

Li and Chen [LC90] present the *Component Alignment Graph* (CAG), a graph where nodes represent array dimensions and edges reflect preferences for alignment. Edges in the graph are weighted with 1 or ε to reflect either a strong or a competing preference respectively. Finding the optimal alignment is reduced to a minimal graph partitioning problem over the CAG. They prove the NP–completeness of the problem and propose a heuristic algorithm. Knobe, Lucas, and Steele [KLS90] use a similar graph formulation: the *preference graph*. In addition to communication related preferences, preferences that reflect possible parallelism are also represented in the graph (anti–preferences). Ayguadé et al. [AGG+97] extend the CAG representation in the framework of their DDT (Data Distribution Tool) to include alignment preferences that would lead to an efficient parallelization of the loops. Tandri and Abdelrahman use a variant of the CAG, called the NUMA–CAG, that represents the affinity between parallel loop iterations and array dimensions [TA96].

The selection of a good alignment requires a communication cost model. Most authors that have adopted the CAG as their basic infrastructure to solve the alignment problem, have targetted their efforts to accurately weighting the edges in the CAG. For instance, Gupta in the PARADIGM system, weights edges representing communication costs assuming a default distribution for the aligned arrays [Gup92]. Laure and Chapman weight CAG edges with an estimate of the data movement cost assuming the worst case [LC96]. Similarly, Kremer determines the edge weights as the maximum data volume of communication that may occur if the corresponding alignment preference is not satisfied [Kre95].

Schreiber et al. [CGS93] propose an alternative representation to solve the alignment problem: the *alignment distribution graph* (ADG). Computations are represented by nodes with edges modeling the flow of data. Edges are weighted by the communication costs that will occur if the source and sink nodes do not have compatible alignments.

The original heuristic solution algorithm proposed by Li and Chen has been improved by several groups in order to generate better alignments with only a low increase in algorithm complexity [AGG+94, LC96]. Alternatively, Kennedy and Kremer [KK95] propose the use of 0–1 integer programming techniques to find an optimal solution to the minimum cost partitioning of the CAG.

3.2 Distribution Problem

The arrays distribution strategy specifies a partition of the group of aligned arrays onto an abstract processors grid, according to a given distribution fashion. In order to decide the number of distributed dimensions and the number of processors assigned to each distributed dimension, most proposals have to estimate the execution time (including data movement and computation) for the program with different distribution strategies.

In order to decide the distribution strategy, some groups have proposed iterative algorithms that evaluate the behavior of a set of distributions [LC91, Gup92, AGG+97], most of them restricting to one or two dimensional distributions, either `BLOCK` or `CYCLIC(n)`. Wholey uses a hill climbing approach to find multidimensional distributions [Who92]. The proposal starts assigning two processors to a single dimension. Then iteratively doubles the number of processors and decides the dimension where these should have to be assigned.

The cost model presented by Li and Chen is based on a pattern matching between the reference pattern of an assignment statement within a loop and a predefined set of data movement routines [LC91]. Each routine has an associated cost which is a function of the block size, communication parameters of the target architecture, and number of processors. This strategy has been adopted by most proposals as a way to estimate the data movement cost. Kennedy and Kremer estimate performance using training sets [BFKK91], a collection of kernel routines that measure run times of several communication and computation patterns in a given architecture.

The benefits of parallel loop execution are usually computed from the profiled sequential execution time and considering the number of processors that will execute that loop in parallel. In addition, several groups have extended the parallel loop model to consider pipelined computations. Synchronization is used to ensure the correct overlap of chunks of dependent iterations [AL93, Kre95, KP96].

3.3 Remapping Problem

For complex problems, static data alignments and distributions may be insufficient to obtain acceptable performance. In this case remapping actions between code blocks (usualy named *phases*) can improve the efficiency of the solution, but communication is required to reorganize the data. The optimal solution can be one with suboptimal mappings in each phase. In addition, the selection of remapping points in the program adds another dimension to the data partitioning problem.

Bixby, Kennedy, and Kremer formulate the dynamic data partitioning problem as a 0–1 integer programming problem that selects a single candidate data layout from predetermined candidate layout search spaces for each phase [BKK94b]. The NP–completeness of the dynamic problem is proven in [Kre93]. Anderson and Lam use a heuristic which combines loop nests (with potentially different distributions) in such a way that the largest potential communication costs are eliminated while still maintaining sufficient parallelism [AL93]. Other approaches are based on the exploration of a search space of candidate solutions for a given set of phases and using a cut function to prune the exhaustive search; Kelly and Pugh consider the problem at the intraprocedural level [KP96] and Ayguadé et al. at the interprocedural level [AGG+95].

Alternatively, Palermo and Banerjee use a different approach to the problem that starts from a static solution for the whole program [PB95]. Then, it is recursively decomposed into a hierarchy of candidate phases using a cost function. Taking into account the remapping costs between these phases, the most efficient sequence of phases is selected. A similar divide–and–conquer approach is used by Chatterjee et al. to solve the dynamic alignment problem[CGSS94].

Finally, Garcia, Ayguadé and Labarta define a single data structure, named the *Communication Parallelism Graph* (CPG), to represent the whole data mapping problem. There is a trade–off between minimizing data movement and maximizing load balance on processors. Therefore they propose to solve the alignment, distribution, and remapping problems in a single step [GAL95, GAL96]. The CPG includes information related to both data movement and parallelization costs. The authors use general purpose 0–1 integer programming techniques to find an optimal solution to the whole problem according to the assumed cost model.

4 A Case Study: Four Implementations

Four different implementations that solve the whole dynamic data layout problem are further analyzed and discussed in this section. They have been selected as representative of other proposals that use a similar approach, as described in Section 3. First we start by describing a proposal by Anderson and Lam at Stanford University, based on combining different phases trying to find a good combination for them. Second we discuss a different approach by Palermo and Banerjee at the University of Illinois at Urbana–Champaign, that starts from a single phase and performs a recursive splitting of phases while profitable. Third we describe an alternative by Kennedy and Kremer at Rice University, based on finding a global solution starting from an initial enumeration of phases and candidate solutions for them. All previous approaches solve the problem in several steps (alignment, distribution and remapping). Finally we describe an approach by Garcia, Ayguadé and Labarta at the Universitat Politècnica de Catalunya, that solves the whole problem in a single step, starting from an enumeration of phases.

4.1 Anderson and Lam

Anderson and Lam in the framework of the SUIF compiler at Stanford University, discuss a compiler algorithm for automatic data and computation mapping for dense matrix computations [AL93, And97a]. The input programs consist of generally nested loops with or without explicit parallelism. Each loop can be either a *forall*, *doacross*, or sequential loop. Automatic parallelization is performed for sequential loops. Loop bounds and array subscripts are assumed to be affine functions of the loop indices and symbolic constants.

The focus of their work is on finding an efficient data and computation mapping onto a virtual processor array. Such a mapping can be referred to as a data and computation alignment. They also address issues related to distribution by calculating a mapping from the virtual processor array onto the physical processor space, using either a block, cyclic, or block–cyclic folding function [And97a]. Their work handles data and computation alignments that may change dynamically, i.e., are not the same for the entire program.

A central tool in their approach is a mathematical formulation of the problem of finding communication-free, static data and computation alignments for a single loop nest or groups of loop nests. Their formulation of the alignment problem considers dimensional and stride alignments, including embeddings. Offset alignments are done in a separate step. Constraints for the data and computation alignments are expressed through requirements for the kernel of the linear functions that represent the data and computation mappings. The kernel computation is based on an iterative method that successively matches the mutual constraints of the data and computation mappings. The result of the kernel computation is the set of all communication-free alignments.

If communication-free alignments can only be found if all data and computations are mapped to the same processor, i.e., if only a trivial communication-free alignment can be found, constraints that enforce the sequential execution of loops that carry cross-iteration dependences (*doacross* loops) are removed. The resulting relaxed system is solved for communication-free alignments. Any non-trivial communication-free solution of the relaxed system corresponds to an alignment that will require nearest-neighbor communication and will result in a pipelined execution.

Anderson's and Lam's algorithm for automatic data layout and computation partitioning considers dynamic data remapping. The program is represented as a weighted, undirected graph where the nodes correspond to loop nests and edges represent possible data remapping between loop nests. The node weights are estimated loop execution times and the edge weights are the combination of worst-case approximations of the actual communication costs and the probabilities that the data remapping will actually occur. Anderson and Lam show that their problem representation of dynamic data layout and computation partitioning is NP-hard.

The algorithm for dynamic data layout and computation partitioning uses a heuristic. The greedy algorithm tries to join loop nodes that are connected by a remapping edge in order to eliminate possible remapping costs. Edges are visited in the order of decreasing weights. The algorithm works in a bottom-up fashion, starting with innermost loops first. Two candidate loop nodes at the current level are merged into a single component if the performance of the joined nodes is higher than the performance of the individual nodes including the remapping costs. Once two nodes have been joined at a level, they are considered a single component for all subsequent levels.

After the greedy algorithm terminates, the entire graph is partitioned into components, and data and computation mappings for each component have been determined. Since the data and computation mappings only specify which array elements and iterations are local to a single processor, a last step is needed to determine the actual virtual processor on which the data and computation are mapped. This last step is referred to as orientation and displacement computation [AL93]. A greedy algorithm is proposed to match as closely as possible the orientations between components.

ADI example

The ADI example fits the author's base case for dynamic data layout where the program consists of one outer sequential loop containing a number of perfectly nested loops.

In the initialization step, the algorithm constructs fully permutable loop nests for all six loop nests inside the sequential outer loop. The loops in each fully permutable nest are ordered such that any parallel loops are outermost. As a result, the compiler performs loop interchange for loops 2 and 4, and transforms all six inner loop nests into a parallel outer loop. For the doubly nested loops the inner loops are sequential.

The communication graph for our ADI example is shown in Figure 3.a. The probability of each remapping edge is 100%. The edge weights represent worst case approximations of necessary communication. In the worst case, an array has to be transposed between loop nests. For illustration purposes, we assume that the cost for a transpose operation is 1.

Based on the communication graph, the algorithm for dynamic data and computation mapping merges nodes in the graph if the resulting combined node reduces the overall cost. Merging avoids remapping, but may introduce restrictions on exploitable parallelism. The merging process is done in a greedy fashion, trying to merge nodes in decreasing order of their weights.

In order to merge two nodes, their interference graph is built. Figure 3.b shows the interference graph for nodes 4 and 5. First, the algorithm tries to find a communication–free data and computation mapping for the interference graph based on the assumptions that all iterations of

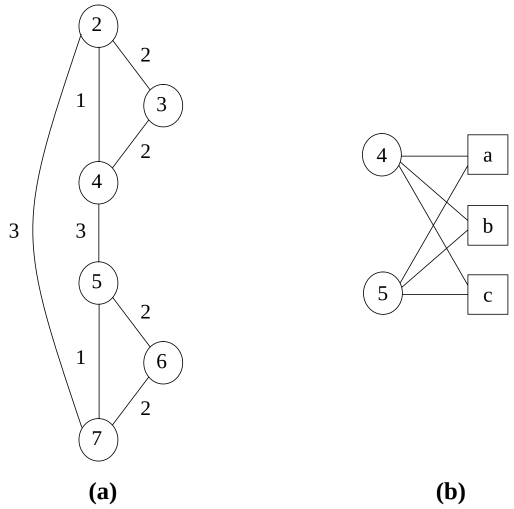


Figure 3: (a) Communication graph for inner loops of ADI; (b) Interference graph for merged nodes for loops 4 and 5.

a sequential loop and all loop iterations that reference the same data have to be mapped to the same processor. A communication–free mapping is not possible for nodes 4 and 5. Since the first step failed, the algorithm checks in a second step whether a wavefront style computation is possible to exploit partial parallelism. The check is based on the form of the array subscripts in the loops. For nodes 4 and 5, a *blocked* distribution that will result in a wavefront execution may be profitable.

The performance model determines whether a wavefront execution of the two nodes is better than dynamic remapping between the nodes. Depending on the result, the nodes 4 and 5 may be merged under a blocked distribution or kept as individual nodes. The interference graph is rebuilt, now containing a single node representing both merged nodes with a blocked distribution if merging showed profitable. The greedy merging process is continued until every edge has been visited.

Finally, an orientation and displacement has to be computed to determine the final data and computation mapping onto the virtual processor space. Displacement tries to eliminate remaining nearest neighbor communication.

4.2 Palermo and Banerjee

PARADIGM (PARAlelizing compiler for DIstributed-memory General-purpose Multicomputers) is a research tool developed at the University of Illinois at Urbana–Champaign, that can automatically select dynamic data distributions starting from static distributions generated using a constraint-based algorithm [GB93] and compile-time cost estimations [GB92] based on empirically measured parameters.

The technique proposed by Palermo and Banerjee for the automatic selection of dynamic data mappings [PB95] 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 different phases, the most efficient sequence of phases and transitions is selected. The authors extend their proposal for the interprocedural case in [PB96].

Initially, the whole program is viewed as a single phase for which a static distribution is determined. The phase decomposition step is based on the *Communication Graph*. Using the selected distribution, the graph is constructed as follows: nodes correspond to to individual statements in the program while edges correspond to flow dependencies that exist between statements weighted by the cost of communication (producer/consumer relationship). The graph records the cost of communication in relation to the flow of data within the program. Once the graph is generated, a single split point is determined by computing the maximal cut of the graph, which removes the largest communication constraints from the phase to potentially allow better individual distributions to be selected for the two resulting split phases. The authors propose an algorithm that takes O(E) time on an adjacency matrix representation of the graph G(V,E). A new distribution is selected for each of the resulting phases and the process is continued recursively in a branch and bound fashion; a phase is split only if the sum of the estimated execution times for the two new phases shows an improvement over the original (i.e., $cost(phase_1) + cost(phase_2) < cost(phase)$).

Once the program has been decomposed into a hierarchy of phases, redistribution costs are estimated using the model presented in [RB95]. With these costs, the *Phase Transition Graph* is built, where nodes are phases and edges are used to connect transitions between them, taking into account control flow information (loops and backward branches). A redistribution edge that occurs within a loop or between the source and the target of a backward branch will have its cost doubled to account for a potential reverse redistribution. A shortest path algorithm is used to find the sequence of phases and phase transitions that minimizes the overall cost.

ADI example

The authors show the behavior of the proposed algorithm on a slightly different version of our ADI running example when targeting an Intel Paragon and a Thinking Machines CM-5. The initial static partitioner selects a column BLOCK distribution for the entire program. For this distribution, the Communication Graph is built with nodes corresponding to the statements in the loop bodies and edges representing the flow of data between them weighted according to the costs of the communication primitives for these machines. The graph is shown in Figure 4.a. For this code, the first iteration of the recursive algorithm divides the program (*Phase 0*) into two phases: *Phase 1* including loops 2 to 4, and *Phase 2* including loops 5 to 7. This is the splitting point that represents the maximal cut of the graph. For *Phase 2*, the same static solution is valid, but for *Phase 1* a row BLOCK distribution is selected. The splitting is valid because the cost is reduced by a factor of 10 for both machines. The recursive algorithm proceeds but does not find other interesting split points inside each phase.

After the phase decomposition step, the Phase Transition Graph is built with three nodes and a set of edges reflecting redistribution costs, as shown in Figure 4.b. On the Intel Paragon the cost of performing redistribution is low enough that a dynamic distribution scheme is selected by the shortest path algorithm. For the CM-5, however, the cost of redistribution is more expensive and therefore a static solution is selected.

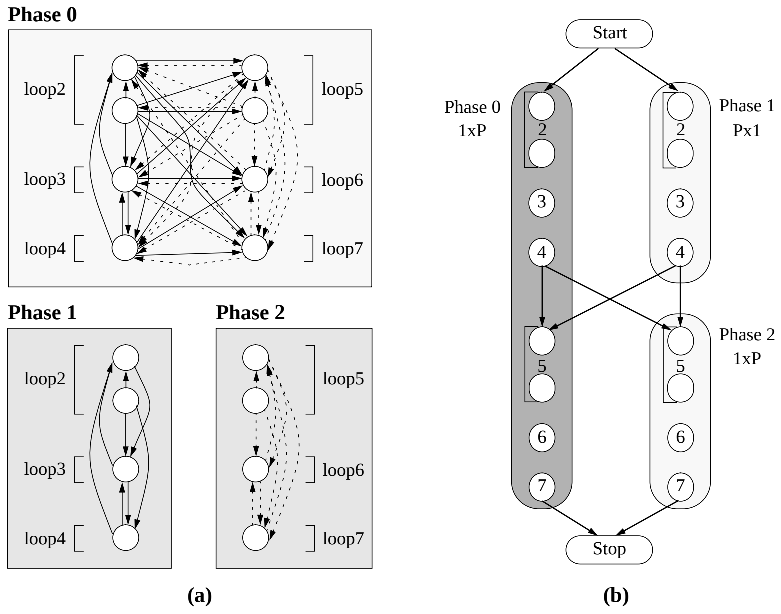


Figure 4: (a) Communication graph for the program after and before phase decomposition; (b) Phase Transition Graph for ADI.

4.3 Kennedy and Kremer

Kennedy and Kremer at Rice University, developed a framework for automatic data layout that builds and examines explicit search spaces of candidate layouts. A candidate layout is an efficient layout for some part of the input program. The relevant program parts are called *phases*. A phase identifies operations on arrays between which remapping may be profitable.

After the generation of candidate layout search spaces for each phase, a single candidate layout is selected from each search space, i.e. for each phase, resulting in a data layout for the entire program. The framework considers dynamic remapping only between phases.

The *phase control flow graph* (PCFG) and the *data layout graph* (DLG) are the main program representation and data structure of Kennedy's and Kremer's approach. A PCFG is a compacted version of a control flow graph where all CFG nodes associated with a phase are represented by a single node in the PCFG, and edges are annotated with control flow information such as branch probabilities or frequency of execution [BKK94b, Kre95].

Candidate layout search spaces are generated for each phase in the PCFG based on heuristics. Heuristics are needed to restrict the search space sizes, but should avoid the elimination of promising candidates. First, a suitable sets of candidate alignment are determined, followed by a corresponding set of candidate distributions. The final data layout search space for each phase consists of the cross product of the alignment and distribution candidates of the search spaces in each phase. Kremer discusses different heuristics for alignment and distribution analysis in [Kre95]. Heuristics for alignment analysis use operations over a lattice of conflict–free CAGs. For instance, the alignment heuristic in the author's prototype implementation computes the optimal, conflict–free CAG for each phase, and then exchanges alignment information among different phases through mutual optimal graph embeddings of the conflict–free, optimal CAGs of different phases. The resulting CAGs are included in the individual phase search spaces.

A node in the DLG represents a candidate data layout in the data layout search space of a phase. The edges represent possible remapping between candidate layouts. The DLG construction as described in [Kre95] uses the assumption that only a single copy of an array can exist at any time during program execution. Phases from which data layouts can be inherited are determined by solving the least recently referenced data flow problem (LR-Refs). More recent work extends the framework to allow read–only replication where profitable [Kre96].

DLG nodes and edges are weighted with their estimated execution times. A powerful performance estimator based on a compiler model, an execution model, and a machine model is used to predict the execution times of each candidate layout and the costs of possible remapping between candidate layouts. For each candidate layout, the compiler model determines the computation and communication that would be generated by the target compiler, if the phase were compiled using the candidate layout. The compiler model supports advanced communication optimizations such as message vectorization and aggregation. The model ignores computations and communications that are not crucial for the overall performance of the compiler generated code. Based on the resulting information, the execution model and target architecture machine model computes the overall execution time. The execution model can predict the performance of pipelined program execution with different granularities. The machine model uses a training set approach [BFKK91].

Finally, a solution to the global data layout selection problem chooses exactly one candidate data layout for each phase such that the overall cost of the resulting path is minimal. This problem has been shown to be NP–complete [Kre93]. Rather than resorting to heuristic prematurely, the framework capitalizes on state–of–the–art 0–1 integer programming technology to compute optimal solutions to two NP–complete problems, namely the data layout selection problem and the interdimensional alignment problem.

An intraprocedural prototype of a data layout tool based on the discussed framework has been implemented as part of the D System [ACG+94]. Experiments indicate that good data layouts can be determined efficiently.

ADI example

Figure 5 shows the partitioning of iterative loop of our ADI kernel example into phases, resulting in the PCFG shown on the left. To simplify the DLG, node and edge weights are not shown. Phases may have candidate layout search spaces of different sizes. The search space construction process is based on a heuristic that considers constraints with respect to a maximal search space size or expected accuracy of the underlying performance estimator. Control flow information presented in the PCFG such as the "max" number of iterations of the outermost loop are considered in the final node and edge weights. Edges may represent possible remappings for more than one array. In the example DLG, there are three classes of edges, namely edges that describe the possible remapping of array a, arrays b, c, and arrays a, b, c.

A solution to the data layout selection problem consists of a set of DLG nodes and the edges between them. The cost of a solution is the sum of its node and edge weights. Figure 5

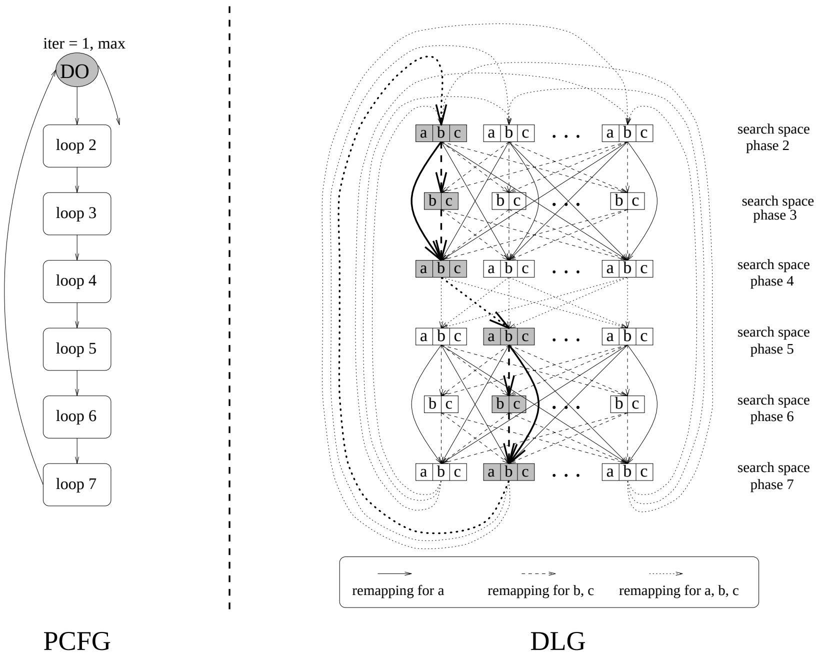


Figure 5: PCFG and DLG for ADI example

highlights a sample solution to the data layout selection problem. In the authors' current prototype implementation, only one–dimensional distributions are considered, resulting in only two layout candidates for each search space, namely a row–wise and column–wise BLOCK distribution. Therefore, the prototype would generate a DLG that corresponds to the DLG shown in Figure 5 restricted to the first two columns of nodes. If we further assume that the first column represents a row–wise and the second a column–wise distribution, the highlighted solution is the dynamic data layout that remaps arrays between the row and column sweeps in the ADI computation.

The framework for automatic data layout translates the NP–complete data layout selection problem into an efficient 0–1 integer programming problem. In cases where the optimal solution cannot be determined within an acceptable time bound, 0–1 technology can be used to produce efficient approximate solutions. However, recent results indicate that optimal solutions can be computed efficiently in many cases. Currently, the largest instance of the data layout selection problem that was tested (approx. instance size: 200 phases, 1400 nodes, and 13,000 edges) was solved optimally in less than 3.4 seconds on a SUN UltraSparc using the general–purpose mixed integer programming tool *CPLEX* with dual simplex as the initial node relaxation.

4.4 Garcia, Ayguadé and Labarta

Garcia et al. [GAL95, GAL96] at the Universitat Politècnica de Catalunya in Spain, propose a framework to automatically determine the data mapping and parallelization strategy for a Fortran 77 program, in the context of a parallelizing environment for massive parallel processor systems. Their main interest has been to develop an approach to find an optimal solution for the data mapping problem, given some characteristics of the target architecture, and according to a predetermined compilation model. Such a mapping can be one-dimensional static or dynamic, `BLOCK` or `CYCLIC`, and takes into account the effects of control flow statements between phases.

This is achieved by defining a single data structure, named the Communication-Parallelism Graph (CPG), which contains information about possible data movement and parallelism within phases, and remapping between phases. All this information is weighted in seconds, according to an assumed cost model for the target system. Once the CPG has been built, the data mapping problem is modeled as a minimal path problem with a set of additional constraints that ensure the correctness of the solution. General purpose linear 0-1 integer programming techniques, which guarantee the optimality of the solution provided, are used to solve the minimal path problem. An appropriate formulation of the problem has been shown to be effective for these kind of problems.

The first step in the method is to decompose the code into phases, according to the definition of phase made in [KK95]. Each phase in the program is represented by a subgraph in the CPG. Subgraphs are structured in columns of nodes: one column is an array, and each node in a column is a distributable dimension of the array.

Data movement information in a phase is obtained from the analysis of the reference patterns within this phase. For each reference pattern between arrays `a` and `b`, a set of data movement edges connect all nodes of array `a` to all nodes of array `b`, denoting all alignment choices between these two arrays. The weight of each data movement edge represents the estimated data movement time required if the corresponding array dimensions are aligned and distributed.

Parallelism information is obtained from the data dependence analysis, and respecting the owner computes rule. Each candidate parallel loop has a hyperedge in the CPG, that links all the array dimensions that have to be aligned and distributed for the loop to be parallelized. The weight associated to a hyperedge is the computation time saved if the loop is finally parallelized.

Remapping information s added in terms of data movement edges and connects subgraphs in the CPG. It is obtained from a data-flow analysis, guided by a phase control flow graph that stores information about probabilities of conditionals and loop iteration counters. If an array `a` at `phase i` is used in a later phase, say `phase j`, then all nodes of array `a` at `phase i` are connected to all nodes of the same array `a` at `phase j` by data movement edges. The weight of each data movement edge represents, as usual, the estimated data movement time required if the corresponding array dimensions are aligned and distributed.

A valid data mapping in the CPG is a path that contains one node from each column. The weight of all edges and hyperedges contained in the path have to be taken into consideration in order to estimate the effects of the corresponding data mapping strategy. Therefore the optimal solution is a valid data mapping such that the predicted execution time is minimized. This is found by modeling a minimal path problem with a set of additional constraints that guarantees that all edges and hyperedges contained in the path are taken into account. This problem is solved optimally using general purpose linear 0-1 integer programming techniques. The data mapping strategy provided includes array alignment and distribution, selection of loops to parallelize, and remapping actions between phases.

ADI example

According to the definition of phase, the approach detects six phases within the iterative `do iter` loop. Each phase corresponds to a loop nest. Data movement and parallelism information is obtained from the analysis of each phase, and remapping information is obtained from the analysis between phases.

The data movement analysis for the first phase detects reference patterns between arrays. The effects of a reference pattern in terms of data movement are inserted in the CPG with edges connecting the array dimensions involved in the reference pattern. For instance, the following reference pattern from the first phase:

$$c(i,j) \leftarrow b(i,j-1)$$

connects all nodes from array `b` to array `c`, as shown in Figure 6.a. The weight of each edge reflects the amount of data movement to be carried out if the two dimensions connected by the edge are aligned and distributed. For instance, according to the reference pattern, if the first dimension of array `b` and the second dimension of array `c` are aligned and distributed, the expected data movement to be carried out is a $ManyToMany$ primitive. Similarly, if the second dimension of arrays `b` and `c` are aligned and distributed, then the expected data movement to be carried out is a $OneToOne$ primitive (a shift).

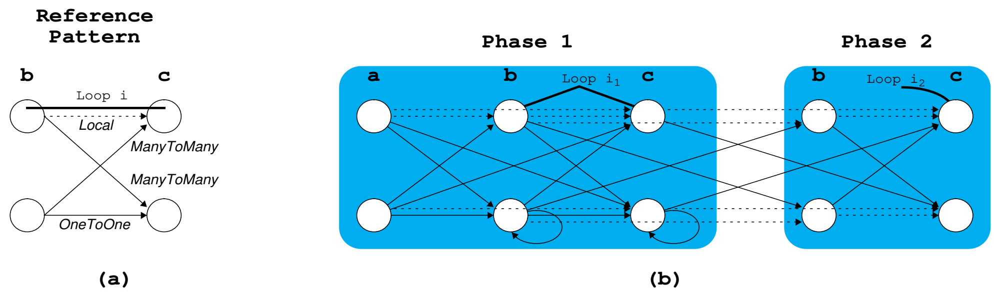


Figure 6: (a) Reference pattern in the CPG, and (b) information in phases 1 and 2 in the CPG.

In terms of parallelism and in the same phase, only the `do i` loop can be parallelized. According to the owner computes rule, for this loop to be parallelized the first dimension of arrays `b` and `c` have to be distributed. Therefore a hyperedge associated to this loop is added connecting the corresponding nodes in the CPG. This hyperedge is shown with bold line in the same Figure 6.a.

In terms of remapping information, the phase control flow analysis generates the PCFG similar to the one generated by [BKK94b], and detects that the six phases are contained within an iterative loop of `max` iterations. Therefore forward remapping between phases has to be accounted for `max` times, and backward remapping between the last and the first phases within the loop has to be accounted for `max-1` times. Remapping information is inserted in the CPG in terms of data movement edges that represent the data movement cost incurred the distribution of one array changes between two phases. In Figure 6.b the CPG for phases 1 and 2 is shown, plus the corresponding remapping edges between arrays `b` and `c` in both phases. Dotted edges represent local memory accesses.

The final CPG is composed of edges and hyperedges weighted in time units. The optimal solution has to select a node from each column such that the summation of weights associated to edges minus the summation of weights associated to hyperedges is minimized. Experiments performed in a 32 processors Silicon Graphics ORIGIN2000 system produce a dynamic solution. Figure 7 shows the set of nodes selected for the optimal solution, and the corresponding set of edges and hyperedges contained within the selected set of nodes.

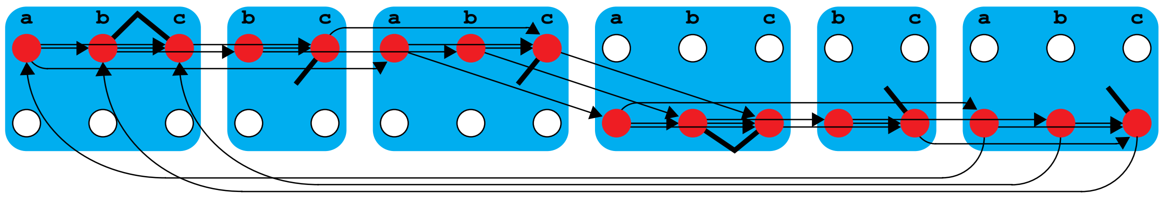


Figure 7: Valid solution in the CPG.

5 Discussion

This paper discussed the different steps and analysis performed by four selected alternative approaches to the data and computation mapping problem. The ADI kernel was used to illustrate the main features of each alternative. Although the different approaches are quite similar, there are several conceptual differences among them, in particular with respect to the heuristics and performance estimates used to determine a final solution.

Those approaches using a top-down phase decomposition step (such as Palermo and Banerjee) result in a reduced number of phases examined as well as the number of redistribution points considered. The top-down approach uses a heuristic to determine a single split point in a static solution. The optimal solution is not guaranteed because finding an optimal solution would require to find multiple split points simultaneously. The other two approaches start by having a set of phases initially defined (either loops by Anderson and Lam, or using an operational definition by Kremer and Kennedy) with a set of candidate distributions for each one. Therefore they start with a search space of maximum size that needs to be explored. The success of the heuristic depends on the set of (possibly suboptimal) partial solutions initially chosen for each phase. Partial solutions not considered at the initial point will never appear in the final solution. Similarly, Garcia et al. also start by defining phases and the kind of solutions allowed for each of them. Allowing different kind of solutions (such as `BLOCK` or `CYCLIC`, or multidimensional distributions) for each phase results in CPG replication and therefore in an increase of the problem complexity [Gar97].

Anderson and Lam, and Palermo and Bannerjee report execution times of their implementations in the order of seconds for programs in their validation program test suites [And97b, Pal97]. However it is important to consider the expected execution times of the approaches by Kremer and Kennedy, and Garcia et al. since they use 0–1 integer programming techniques to find optimal solutions. Table 1 shows the execution times of prototype implementations of their approaches in seconds for three benchmark programs. The timings were taken on single processor SUN workstations. The 0–1 problem instances were solved by general purpose integer programming tools CPLEX v4.0 [Bix92] (Kremer and Kennedy) and LINGO v3.1 [LIN94] (Garcia et al.).

	Adi	**Erlebacher**	**Tomcatv**
Execution time for 1–dim block distributions in seconds			
Kremer and Kennedy†	**0.35** (0.01)	**2.4** (0.03)	**2.62** (0.44‡)
Garcia, Ayguadé and Labarta	**0.5**	**4.9**	**0.8**
Execution time for 2–dim block distributions in seconds			
Kremer and Kennedy†	**1.36** (0.02)	**13.4** (0.86)	**6.7** (0.58‡)
Garcia, Ayguadé and Labarta	**1.6**	**1542**	**2.5**

Table 1: Comparison of execution times of the two data layout approaches that provide optimal solutions based on 0–1 integer programming techniques. † Excludes the initialization time for the performance estimation machine model (training sets). ‡ Time includes solution of two 0–1 integer programming formulations of the inter–procedural alignment problem.

The numbers in paranthesis are the overall solution times for CPLEXv4.0 for the generated 0–1 problems. For the one dimensional case, the majority of local, i.e., phase data layout search spaces in Kremer and Kennedy's approach had 2 (Adi), 3 (Erlebacher), and 4 (Tomcatv) data layout candidates. The corresponding numbers for the two dimensional case were 6 (Adi), 15 (Erlebacher), and 12 (Tomcatv). However in Garcia et al. the data layout search spaces at each phase are tipically much larger than Kennedy and Kremer's approach. The reason is that they decide alignment and distribution for all phases within the same problem. For instance, in the one dimensional case, the number of data layout candidates considered can become 8 (Adi), 1.5 millions (Erlebacher), and 128 (Tomcatv). The numbers in the two dimensional case were 64 (Adi), 13,000 millions (Erlebacher), and 16,384 (Tomcatv).

All approaches need performance estimates to make decisions regarding the quality of the considered data and computation mappings. Performance estimates have to be precise enough to be able to distinguish the considered search space of possible data and computation mappings. Due to the lack of reported experimental data, there is not yet an understanding of the precise relation between data layout search spaces and their corresponding matching performance models. However, approaches that use 0–1 integer programming will need performance estimates that are integral values rather than symbolic performance expressions. For cache-coherent NUMA systems, the estimation of data movement has to consider the physical allocation of data across the memory modules; data elements are moved either in pages or cache lines. Invalidations and false sharing are additional aspects to consider when considering alternatives for data mapping [AGGL96, TA96]. For cahe-coherent distributed memory NUMA architectures, the performance model may have to consider data locality across different levels of the memory hierarchy, i.e., local memory and caches. Recently, researchers have started to develop automatic data and computation mapping techniques that consider memory hierarchies of multiple levels. The techniques are able to deal with the tradeoffs between parallelism, inter–processor communcication, and cache traffic [BKA96].

More experiments are needed to validate the different approaches to automatic data and computation mapping. A crucial mile stone in allowing comprehensive experimentation is interprocedural analysis since large real application programs consist of many procedures. The PARADIGM and SUIF projects have already interprocedural prototype implementations [PB96, And97a]. An interprocedural implementation based on the Kennedy and Kremer framework is currently under development at Rutgers University.

Besides the need for more experimental validation, future challenges include data and computation mappings under resource constraints and mappings for objective functions different from merely minimal execution times. Different data mappings may induce different memory requirements of the generated node programs. Read–only replication can be used to reduce communication costs under the constraint that the additional memory requirements can be satisfied [Kre96].

New objective functions for data and computation mappings will be needed in programming environments that provide computation and memory resources in a time–shared fashion. In such systems, users will be charged for the actual amount of computing resources used by their program execution. Data and computation mappings are needed that keep the program execution efficiency above a given threshold, trading off execution times vs. execution costs in terms of dollar charges.

6 Acknowledgements

We would like to thank Jennifer Anderson for her helpful comments and suggestions. This work has been partially supported by DARPA contract DABT 63-93-C-0064, by the Ministry of Education of Spain under contract TIC429/95, and the CEPBA (European Center for Parallelism of Barcelona).

References

[ACG+94] V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crummey, C-W. Tseng, and S. Warren. Requirements for data-parallel programming environments. *IEEE Parallel and Distributed Technology*, 2(3):48–58, 1994.

[AGG+94] E. Ayguadé, J. Garcia, M. Gironès, J. Labarta, J. Torres, and M. Valero. Detecting and using affinity in an automatic data distribution tool. In K. Pingali et al., editors, *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computing*, pages 61–75, Ithaca, NY, August 1994. Springer-Verlag.

[AGG+95] E. Ayguadé, J. Garcia, M. Gironès, M.L. Grande, and J. Labarta. Data redistribution in an automatic data distribution tool. In C.-H. Huang et al., editors, *Proceedings of the 8th Annual Workshop on Languages and Compilers for Parallel Computing*, pages 407–421, Columbus, OH, August 1995. Springer-Verlag.

[AGG+97] E. Ayguadé, J. Garcia, M. Gironès, M.L. Grande, and J. Labarta. DDT: A research tool for automatic data distribution. *Scientific Programming*, Vol 6(1):73–94, Spring 1997.

[AGGL96] E. Ayguadé, J. Garcia, M.L. Grande, and J. Labarta. Data distribution and loop parallelization for shared-memory multiprocessors. In *Proceedings of the 9th Annual Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996. Sprinver-Verlag.

[AL93] J. Anderson and M.S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN'93 Conference on Program Language Design and Implementation*, Albuquerque, NM, June 1993.

[And97a] J. Anderson. *Automatic Computation and Data Decomposition for Multiprocessors*. PhD thesis, Stanford University, March 1997.

[And97b] J. Anderson. Private communication, August 1997.

[BFKK91] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[Bix92] R. Bixby. Implementing the Simplex method: The initial basis. *ORSA Journal on Computing*, 4(3), 1992.

[BKA96] R. Barua, D. Kranz, and A. Agarwal. Communication-minimal partitioning of parallel loops and data arrays for cache-coherent distributed-memory multiprocessors. In *Proceedings of the 9th Annual Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996. Springer-Verlag.

[BKK+94a] D. Bau, I. Kodukula, V. Kotlyar, K. Pingali, and P. Stodghill. Solving alignment using elementary linear algebra. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.

[BKK94b] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0–1 integer programming. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT94)*, pages 111–122, Montreal, Canada, August 1994.

[CGC96] S. Chakrabarti, M. Gupta, and J-D. Choi. Global communication analysis and optimization. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 68–78, Philadelphia, PA, May 1996.

[CGS93] S. Chatterjee, J.R. Gilbert, and R. Schreiber. The alignment-distribution graph. In U. Banerjee et al., editors, *Proceedings of the 6th Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993. Springer-Verlag.

[CGSS94] S. Chatterjee, J.R. Gilbert, R. Schreiber, and T.J. Sheffler. Array distribution in data-parallel programs. In K. Pingali et al., editors, *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[FFG+95] D. Fenwick, D. Foley, W. Gist, S. VanDoren, and D. Wissell. The Alphaserver 8000 series: High-end server platform development. *Digital Technical Journal*, 7(1), 1995.

[GAL95] J. Garcia, E. Ayguadé, and J. Labarta. A novel approach towards automatic data distribution. In *Proceedings of Supercomputing'95*, San Diego, CA, December 1995.

[GAL96] J. Garcia, E. Ayguadé, and J. Labarta. Dynamic data distribution with control flow analysis. In *Proceedings of Supercomputing'96*, Pittsburgh, PA, November 1996.

[Gar97] J. Garcia. *Automatic Data Distribution for Massively Parallel Processors*. PhD thesis, Universitat Politècnica de Catalunya, Barcelona, Spain, April 1997. Available as UPC-DAC-97-27 or UPC-CEPBA-97-08.

[GB92] M. Gupta and P. Banerjee. Compile-time estimation of communication costs on multicomputers. In *Proceedings of the 6th International Parallel Processing Symposium*, Beverly Hills, CA, March 1992.

[GB93] M. Gupta and P. Banerjee. PARADIGM: A compiler for automatic data distribution on multicomputers. In *Proceedings of the 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[Gup92] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, September 1992.

[IDFL96] L. Iftode, C. Dubnicki, E.W. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *Proceedings of the 2nd International Symposium on High Performance Computer Architecture*, February 1996.

[KCDZ94] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter USENIX Conference*, January 1994.

[KHS+97] L Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. Scott. VM-based shared memory on low-latency, remote-memory-access networks. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[KK95] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proceedings of Supercomputing'95*, San Diego, CA, December 1995.

[KLS90] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[KLS+94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, MA, 1994.

[KP96] W. Kelly and W. Pugh. Minimizing communication while preserving parallelism. In *Proceedings of the 10th ACM International Conference on Supercomputing*, Philadelphia, PE, May 1996.

[Kre93] U. Kremer. NP-completeness of dynamic remapping. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993. Also available as technical report CRPC-TR93-330-S (D Newsletter #8), Rice University.

[Kre95] U. Kremer. *Automatic Data Layout for Distributed Memory Machines.* PhD thesis, Rice University, October 1995. Available as CRPC-TR95-559-S.

[Kre96] U. Kremer. Automatic data layout with read–only replication and memory constraints. Technical Report LCSR-TR93-283, Department of Computer Science, Rutgers University, December 1996.

[LC90] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[LC91] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, Vol 2(3), July 1991.

[LC96] E. Laure and B. Chapman. A refined method for alignment analysis combining inter- and intradimensional alignment preferences. In M. Gerndt, editor, *Proceedings of the 6th International Workshop on Compilers for Parallel Computers*, pages 169–180, December 1996.

[LIN94] LINDO Systems Inc. *LINGO Optimization Modeling Language*, April 1994.

[LLG+90] D. Lenoski, J. Laudon, K Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990.

[Ost89] A. Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems.* Sequent Technical Publications, San Diego, CA, 1989.

[Pal97] D. Palermo. Private communication, August 1997.

[PB95] D.J. Palermo and P. Banerjee. Automatic selection of dynamic partitioning schemes for distributed-memory multicomputers. In C.-H. Huang et al., editors, *Proceedings of the 8th Annual Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.

[PB96] D.J. Palermo and P. Banerjee. Interprocedural array redistribution with data-flow analysis. In *Proceedings of the 9th Annual Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996. Sprinver-Verlag.

[RB95] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *Frontiers95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 1995.

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

[TA96] S. Tandri and T.S. Abdelrahman. Automatic data and computation partitioning on scalable shared–memory multiprocessors. In *Proceedings of the Nineth Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996.

[Tse93] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, Houston, TX, January 1993. Rice COMP TR93-199.

[WH95] D. Wood and M. Hill. Cost–effective parallel computing. *IEEE Computer*, Feb 1995.

[Who92] S. Wholey. Automatic data mapping for distributed-memory parallel computers. In *Proceedings of the 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.