# Automatic Data Layout for Distributed-Memory Machines

*Ulrich Kremer*

**CRPC-TR93299-S**
**February, 1993**

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

# Automatic Data Layout for Distributed-Memory Machines

Ulrich Kremer

**Thesis Proposal**

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251

February 17, 1993

**Abstract**

An approach to programming distributed memory-parallel machines that has recently become popular is one where the programmer explicitly specifies the layout of data in a global name space, relying on a compiler to generate a parallel program including all the communication. While this frees the programmer from the tedium of thinking about local name spaces and message-passing, no assistance is provided in determining an efficient data layout scheme on the target machine. We wish to fill this gap by providing automatic data alignment and distribution techniques for a large class of regular, scientific computations written in a *data-parallel programming style.*

We propose an interactive software tool that will first determine a set of efficient data decomposition schemes for the entire program. Subsequently, the user will be able to select a region of the input program and the system will respond with set of decomposition schemes and their performance characteristics for the selected region. For each scheme, the tool will provide information about the location and type of the communication operations generated by the compiler. This will enable the user to obtain insights into the characteristics of the program when executed on a distributed memory machine, and the behavior of the underlying compilation system.

The proposed tool will use static performance estimation based on *training sets.* An empirical study of application programs will show whether automatic techniques are able to generate data decomposition schemes that are close to optimal. If automatic techniques fail to do so, we want to answer the question of how user interaction can help overcome the deficiencies of automatic techniques.

## 1 Introduction

Although distributed-memory message-passing parallel computers are among the most cost-effective high performance machines available, scientists find them extremely difficult to program. The reason is that traditional programming languages support a global name space and hence, most programmers feel more comfortable working with a shared memory programming model. A number of researchers have proposed annotating a language based on a global name space with directives specifying how the data should be mapped onto the distributed memory machine [CK88, CCL88,

KMV90, PSvG91, RA90, RP89, RSW88, ZBG88, KZBG88, Ger89]. This approach was inspired by the observation that the most demanding intellectual step in writing a program for a distributed memory machine is the appropriate data layout - the rest is straightforward but tedious and error prone work. The Fortran D language and its compiler [FHK+90, HKT92b] support this programming style. Given a Fortran D program, the compiler uses data layout directives to automatically generates a single-program, multiple data (SPMD) node program for a given distributed-memory target machine.

Selecting a good data layout is important for a program to achieve high performance. Current tools provide little or no support. The choice of a good data decomposition scheme depends on the compiler, the target machine and its size, and the problem size. Depending on the specified data decomposition scheme and the structure of the source program, the compiler performs a variety of different optimization transformations. The machine size and problem size influence the balance between computation and communication in the compiler-generated node program. This balance is crucial for the performance of the node program. These factors make it difficult for a programmer to predict the behavior of a given data decomposition scheme without compiling and running the program on the specific target system.

My thesis is that *efficient data layouts can be generated automatically for many application programs that solve regular problems, if they are written in a data-parallel programming style.* A program is written in a data-parallel programming style if it allows advanced compilation systems to generate efficient code for most distributed-memory machines. In the context of vectorization, the existence of a usable programming style has been partially responsible for the success of automatic vectorization [CKK89, Wol89]. I do not believe that such fully automatic techniques will be successful for parallelizing 'dusty deck' programs.

To support my thesis I will build a prototype tool that takes a Fortran 77 program as input and generates a Fortran D program. This tool has to understand how the data layout scheme affects the execution time and memory requirements of the compiler generated code. This is a difficult task since advanced compilation systems perform extensive intra and inter-procedural analysis and optimizations. A large number of these optimizations depend on the specified data layout scheme. Once the compiler generated node programs are known the tool must be able to predict their performance on a given distributed-memory machine.

I will enhance existing techniques for automatic data layout and develop new techniques based on a new approach to static performance estimation that takes problem and machine sizes into account. In contrast to previous work, my techniques will handle the complexity of advanced compilation systems such as Fortran D and a rich set of data mappings. The techniques will consider the profitability of dynamic data mapping and data replication, and will be able to deal with control flow.

I will validate my thesis by applying my tool to a test suite of programs and program kernels written in a data-parallel programming style. I will use a benchmark suite developed by Geoffrey Fox at Syracuse [MFvL+92] and a set of real application programs. If automatic techniques will fail to generate data decomposition schemes that are close to optimal, I want to answer the question how user interaction can help to overcome the deficiencies of automatic techniques. In particular, I want to investigate language annotations that allow partial specifications of data layouts.

The remainder of this proposal is organized as follows. Section 2 provides a short introduction to the Fortran D language and compilation system, followed by the definition of a data-parallel programming style. Section 3 contains a discussion of related work. Section 4 lists an outline of my research plan and validation strategy. This proposal concludes with a discussion of possible spin-offs of automatic data decomposition and a summary of the proposed work and its contributions.

# 2 Fortran D

## 2.1 The Language

The task of distributing data across processors can be approached by considering the two levels of parallelism in data-parallel applications. First, there is the question of how arrays should be *aligned* with respect to one another, both within and across array dimensions. We call this the *problem mapping* induced by the structure of the underlying computation. It represents the minimal requirements for reducing data movement for the program, and is largely independent of any machine considerations. The alignment of arrays in the program depends on the natural fine-grain parallelism defined by individual members of data arrays.

Second, there is the question of how arrays should be *distributed* onto the actual parallel machine. We call this the *machine mapping* caused by translating the problem onto the finite resources of the machine. It is affected by the topology, communication mechanisms, size of local memory, and number of processors of the underlying machine. The distribution of arrays in the program depends on the coarse-grain parallelism defined by the physical parallel machine.

Fortran D is a version of Fortran that provides data decomposition specifications for these two levels of parallelism using `DECOMPOSITION`, `ALIGN`, and `DISTRIBUTE` statements. A decomposition is an abstract problem or index domain; it does not require any storage. Each element of a decomposition represents a unit of computation. The `DECOMPOSITION` statement declares the name, dimensionality, and size of a decomposition.

The `ALIGN` statement maps arrays onto decompositions. Arrays mapped to the same decomposition are automatically aligned with each other. Alignment can take place either within or across dimensions. The alignment of arrays to decompositions is specified by placeholders `I, J, K, ...` in the subscript expressions of both the array and decomposition. In the example below,

```
REAL X(N,N)
DECOMPOSITION A(N,N)
ALIGN X(I,J) with A(J-2,I+3)
```

$A$ is declared to be a two dimensional decomposition of size $N \times N$. Array $X$ is then aligned with respect to $A$ with the dimensions permuted and offsets within each dimension.

After arrays have been aligned with a decomposition, the `DISTRIBUTE` statement maps the decomposition to the finite resources of the physical machine. Distributions are specified by assigning an independent *attribute* to each dimension of a decomposition. Predefined attributes are `BLOCK`, `CYCLIC`, and `BLOCK_CYCLIC`. The symbol ":" marks dimensions that are not distributed. Choosing the distribution for a decomposition maps all arrays aligned with the decomposition to the machine. In the following example,

```
DECOMPOSITION A(N,N), B(N,N)
DISTRIBUTE A(:, BLOCK)
DISTRIBUTE B(CYCLIC,:)
```

distributing decomposition $A$ by `(:,BLOCK)` results in a column partition of arrays aligned with $A$. Distributing $B$ by `(CYCLIC,:)` partitions the rows of $B$ in a round-robin fashion among processors. These sample data alignment and distributions are shown in Figure 1.

We should note that the goal in designing Fortran D is not to support the most general data decompositions possible. Instead, the intent is to provide decompositions that are both powerful enough to express data parallelism in scientific programs, and simple enough to permit the compiler to produce efficient programs. Fortran D is a language with semantics very similar to sequential

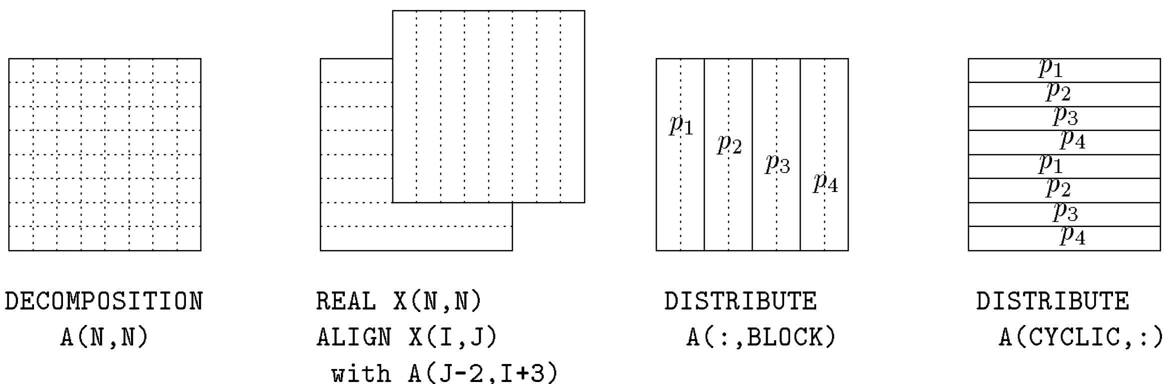


Figure 1: Fortran D Data Decomposition Specifications

Fortran. As a result, it should be easy to use by computational scientists. In addition, we believe that the two-phase strategy for specifying data decomposition is natural and conducive to writing modular and portable code. Fortran D bears similarities to both CM Fortran [TMC89] and Kali [KM91]. The complete language is described in detail elsewhere [FHK+90].

## 2.2 The Compilation System

A Fortran D compilation system translates a Fortran D program into a Fortran 77 SPMD node program that contains calls to library primitives for interprocessor communication. A vendor-supplied Fortran 77 node compiler is used to generate an executable that will run on each node of the distributed-memory target machine. A Fortran D compiler may support optimizations that reduce or hide communication overhead, exploit parallelism, or reduce memory requirements. Procedure cloning or inlining may be applied under certain conditions to improve context for optimization. A Fortran D compiler may relax the owner computes rule for reductions and parallel prefix operations, and for scalars or arrays that are recognized to be temporaries [HKT91, HKT92a, HHKT91, Tse93]. Node compilers may perform optimizations to exploit the memory hierarchy and instruction-level parallelism available on the target node processor [Car92, Wol92, Bri92].

Our tool for automatic data decomposition is based on a specific Fortran D compilation system representing state-of-the-art compiler technology. Note that the optimizations performed by the Fortran D compiler are not restricted to the ones implemented in the current prototype of the Fortran D compiler [Tse93]. Our compilation system will target a variety of distributed-memory multiprocessors such as Intel's iPSC/860 and Paragon, Ncube's Ncube-1 and Ncube-2, and Thinking Machine Corporation's CM-5.

## 2.3 The Programming Style

Our tool takes sequential Fortran 77 programs that solve regular problems as input. The programs are written in a style that enables a given compiler or class of compilers to generate efficient code for many different architectures. The burden for machine-dependent optimizations is shifted from the user to the compiler.

The following incomplete list contains guidelines to write programs in a data-parallel programming style:

- Do not write machine-dependent code. For instance, do not linearize for vector machines or block the code to improve locality of accesses (cache performance).
- Name data objects explicitly. Do not use work arrays. We assume the availability of dynamic memory allocation in our version of Fortran 77.
- Avoid indirect addressing where possible.

The program should be written in a way that takes full advantage of the abilities of advanced compilation systems. Machine dependent programming is substituted by compiler technology dependent programming. Although this might still not be the desired problem dependent programming style, we consider it a significant step into this direction. A discussion of a routine written for a vector CRAY and rewritten in data-parallel programming style is given in appendix A.

# 3 Related work

Most techniques for automatic data layout discussed in the literature address the problem of finding an optimal data layout scheme in terms of overall execution time rather than in terms of memory requirements. The techniques described below determine data layout schemes that minimize communication while maximizing parallelism.

## 3.1 MIMD Machines

Initial approaches to automatic data decomposition for MIMD machines concentrate on single loop nests. The iteration space is first partitioned into sets of iterations that can be executed independently. The data mapping is then determined by the iterations that are assigned to the different processors [RS89, Ram90, D'H89, KKBP91].

Other techniques for single loop nests are based on recognizing specific computation patterns in the loop, called *stencils* [SS90, HA90]. This more abstract representation is used to find a good data mapping. While these approaches will minimize the necessary communication in single loop nests, it is not clear whether a compiler can generate efficient node programs for the loop nests or between adjacent loop nests. In particular, the computation necessary to determine which processor owns which datum can be very complex.

In the remainder of this section we will only present related work that discusses automatic data decomposition for entire subroutines or whole programs.

### 3.1.1 Li, Chen, and Choo at Yale University

Li, Chen, and Choo investigated techniques for automatic data layout as part of the Crystal compiler and language project at Yale University [CCL89, LC90a, LC91b, LC91a, LC90b]. Crystal is a high-level, purely functional language. It does not contain statements that specify the data layout. The goal of the Crystal compiler is to generate efficient SPMD node programs with explicit communications or synchronization for a variety of massively parallel machines. In the following, we will only discuss the aspects of the project that relate to automatic data decomposition.

The automatic data layout algorithm used by Crystal works only on single procedures. A procedure is first partitioned into separate groups of computations, called *phases*, based on the flow of values in the subroutine. Alignment and distribution analysis is performed for each phase in isolation, resulting in a single data decomposition scheme for the phase. Finally, the data layout schemes of the different phases are merged.

Most of the published work concentrates on the problem of finding a static alignment for a single phase. A possible phase merging algorithm is only sketched. In the remainder of this section we will assume that we only deal with a single phase.

The alignment algorithm performs inter-dimensional alignment, followed by intra-dimensional alignment. The index domain of arrays are mapped onto the single, common index domain of the phase based on four simple types of alignment functions, namely *permutation*, *embedding*, *shift*, and *reflection*. Permutation and embedding are inter-dimensional alignment functions that map dimensions of the array onto the dimensions of the common index domain. In a permutation, the array and the common index domain have the same number of dimensions. The alignment function is a permutation of the dimensions of the array. If the index domain has more dimensions than the array, the embedding maps each dimension of the array onto a distinct dimension of the common index domain and specifies the location of the induced subspace in the common index domain. In particular, diagonal embeddings are possible. Shift and reflection are intra-dimensional alignment functions of the form `g(i) = i - const` and `g(i) = - i`, respectively.

The inter-dimensional alignment problem is modeled as a graph problem. An undirected, weighted graph, called the *component affinity graph*, is constructed based on normalized reference patterns in the source program. Each dimension of an array that is referenced in the phase is represented by a node. There is an edge between two nodes if the subscript expressions of the corresponding dimensions are affine, i.e. have the form `i` and `i + const`, where const is a small constant. Edges that are generated by the same reference pattern and are incident to the same node are assigned a weight $\epsilon$, a small positive integer. All other edges have weights equal to 1. The alignment algorithm partitions the component affinity graph into `n` disjoint subsets of nodes, where `n` is the maximal number of dimensions of an array referenced in the phase. The goal is to find a partitioning that minimizes the overall sum of weights of edges between nodes in distinct partitions. Note that edges between partitions are alignment requests that cannot be satisfied. The solution of this alignment problem is shown to be NP-complete [LC90a]. The intra-dimensional alignment algorithm is based on the affine reference patterns and is straight forward.

In the next compiler step the functional program is transformed into an imperative program that allows multiple assignments into the same memory location in order to ensure efficient reuse of memory. Subsequently, calls to communication routines are inserted based on pattern matching using a parameterized layout scheme that distributes all dimensions of the common index domain. Hence the program is 'compiled' only once for a whole family of distribution schemes. For each communication routine a cost function is available that is parameterized with respect to the chosen distribution, problem size, and machine characteristics. The distribution strategy with the minimal cost is selected. Once a distribution strategy is chosen, redundant communication is eliminated.

A prototype of the compiler has been implemented as part of Li's Ph.D. thesis at Yale University. Experimental results are reported for a heuristic algorithm that performs inter-dimensional alignment on a set of randomly generated component affinity graphs [LC90a]. Distribution analysis has not been implemented. The distribution strategy is read in at runtime [Li92].

### 3.1.2 Banerjee and Gupta at the University of Illinois

Gupta and Banerjee at the University of Illinois at Urbana-Champaign developed techniques for automatic data layout as part of a compiler based on the Parafrase-2 program restructurer [GB90, GB91, GB92]. The compiler takes Fortran 77 as input and generates SPMD node programs with explicit communication. The compiler performs alignment and distribution analysis based on *constraints* for each single statement in the program. Constraints represent properties of the data

layout and are associated with a quality measure. Constraints that reflect the alignment of arrays in a statement are either satisfied or not. The quality measure is a penalty function representing the cost for the case that the arrays are not aligned. Constraints that reflect the distribution of aligned arrays have parameterized execution time cost functions as their quality measures. The parameters include the problem size, and number of processors and distribution schemes used in each dimension. The automatic techniques handle cyclic, block, and block-cyclic distributions. In addition, partial replication of arrays is considered. Scalars are assumed to be replicated. With the problem size and machine size known at compile time, the system selects a decomposition scheme that allows arrays to be distributed across a two-dimensional processor grid. The optimal number of processors in each distributed dimension is selected automatically. The compiler does not perform inter-procedural analysis. A single, static decomposition scheme is derived for the entire program, i.e. dynamic realignment or redistribution are not supported.

The compiler performs alignment analysis based on Li's and Chen' s approach [LC90a] (see Section 3.1.1). The communication cost of each statement with an array reference is expressed as a function of the machine size, number of processors in each dimension, and the method of partitioning, namely block or cyclic. The functions try to reflect the effects of loop transformations and communication optimizations, such as message vectorization and aggregation, on the communication costs of a single statements. Each function represents a constraint for the statement. In the next step, the best distribution scheme for each distributed dimension is determined for a default number of processors. The distribution schemes considered are block (continuous), cyclic, and block-cyclic with different block sizes. The best resulting scheme is parameterized with respect to the processor number in each dimension and the optimal number of processors in the two grid dimensions is computed. Finally, the compiler checks whether array replication is profitable.

The automatic techniques have been implemented as part of Parafrase-2. They have been applied to five Fortran programs, namely one routine from the Linpack library (*dgefa*), one Eispack routine (*tred2*), and three programs from the Perfect Club Benchmark Suite (*trfd, mdg, flo52*) [Clu89]. In the study, all the steps of the described automatic data layout techniques are simulated by hand. A distributed memory compiler is not part of Parafrase-2. Actual performance figures for the generated data layout schemes are only given for *tred2* on an iPSC/2 hypercube system. The automatic layout performs well compared to three other data mappings.

### 3.1.3 Wholey at Carnegie Mellon University

Wholey at Carnegie Mellon University [Who92a, Who91] developed a compiler for the high-level, block structured, non-recursive language ALEXI. Communication and parallelism is expressed explicitly by primitive operations that are similar to Fortran90 array constructs and intrinsic communication functions. The work concentrates on the problem of deriving a good data layout scheme automatically without the knowledge of the problem and machine size at compile time. Dynamic realignment or redistribution is not considered, but inter-procedural performance analysis is performed.

Alignment analysis is done at compile time based on the approach by Knobe, Lukas, and Steele [KLS90] (see Section 3.2.1). Distribution analysis is performed at run-time. Each primitive operation is associated with a cost function that computes the execution time of the operation under a given distribution, problem size, machine size, and machine topology. Given these parameters, the overall execution time of the program is determined by adding up the costs for each primitive operation. The performance estimation does not deal with the case where communication and computation overlaps. The search space is restricted to non-cyclic, block distributions. A hill

climbing search method generates the search space of the possible number of processors in each dimension of the virtual processor array. The algorithm returns the distribution, machine size, and topology with the minimal estimated execution time.

A prototype ALEXI compiler has been implemented. Based on simulations of some kernel routines on different distributed memory machines, the performance of the routines with the automatically generated data layouts is shown to be superior to the performance of the routines under some straight-forward data mappings. The performance of automatically determined data layouts has not been compared with the optimal possible [Who92b]. The precision of the performance estimation technique in terms of the relative performance of different data layouts is demonstrated by comparing the estimated costs with the actual execution times of a simplex program on a CM-2.

### 3.1.4 Sussman at Carnegie Mellon University

Sussman discusses static performance estimation to guide the mapping of data and computation onto distributed-memory machines in an automatic compiler [Sus92, Sus91]. His work focuses on determining efficient data and computation mappings for programs consisting of single loop nests. A program is classified as either a sequential loop, a sequentially iterated parallel loop, or a parallel loop. Given a loop nest, the compiler chooses a data and computation mapping scheme out of a set of schemes whose performance can be predicted efficiently at compile time. Each possible mapping scheme onto a target machine for a class of loops is associated with an execution model of that machine. Execution models are parameterized with respect to problem and machine characteristics, such as the number of iterations in the loop, the problem size, and the number of processors used.

The author implemented his execution model approach as part of a compiler for the applicative language SISAL for the Warp systolic array machine. For several benchmark programs and program kernels the compiler is able to predict the relative performance of different mapping schemes with high accuracy.

It is not clear whether the same results can be achieved for entire programs or on machines such as the iPSC/i860 or CM-5. The compiler does not perform any intra or inter-procedural optimizations. Data redistribution or replication is not considered.

### 3.1.5 Chapman, Fahringer, Blasko, Herbeck, Zima at the University of Vienna

Chapman, Fahringer, Blasko, Herbeck, and Zima at the University of Vienna propose automatic data decomposition as part of the interactive parallelization system SUPERB-2 [CHZ91, CH91, FBZ92]. SUPERB-2 takes Fortran 77 programs as input and generates SPMD node programs with explicit communication. Their approach is based on Gupta's and Banerjee's work at Illinois (see Section 3.1.2). In addition to the statement level pattern matching, high-level pattern matching is used to identify specific computations in the program such as stencil computations and matrix multiply. Information about the implementation of these computation patterns on the target machine is stored in a knowledge data base. A 'weight finder' locates the portions of the code that contribute the most to the overall execution time of the program. The effort to find a good data layout is concentrated on these crucial regions. Static performance estimation is used to evaluate the data mappings in a search space of reasonable data layouts. The tool performs inter-procedural analysis and determines the profitability of redistribution.

The proposed tool is currently being implemented at the University of Vienna. A prototype static performance estimator is in its testing phase [FBZ92, Fah92]. No experimental results have been published.

### 3.1.6 ASPAR and $P^3C$

ASPAR is a compiler for the C language developed by the ParaSoft corporation [IFKF90]. $P^3C$ is a research Pascal compiler designed and implemented at the Tel-Aviv University by Gabber, Averbuch, and Yehudai [GAY91]. Both systems generate SPMD node programs that contain calls to communication library routines. The compilers perform only a simple form of program analysis to generate the correct communications. The set of possible data decomposition schemes is small. Inter-procedural analysis is performed. The $P^3C$ has been tested on several programs with good results. Performance numbers of the ASPAR system are only reported for a conjugate gradient program.

## 3.2 SIMD Machines

### 3.2.1 Albert, Knobe, Lukas, Natarajan, Steele, and Weiss at Compass and Thinking Machines

Albert, Knobe, Lukas, Natarajan, Steele, and Weiss discuss automatic data layout as part of the design and implementation at Compass of SIMD compilers for Fortran 77 extended by Fortran 8x array features [AKLS88, KLS88, KLS90, KN90, Wei91]. The target machines are the Connection Machine CM-2 and the MasPar MP-1. Automatic data layout is an integral part of these compilers.

Arrays are aligned by mapping them onto *virtual processors* based on their usage as opposed to a their declared shape. The latter mapping is referred to as the *canonical mapping*. Each virtual processor holds at most a single element of each array. The alignment algorithm performs intra-dimensional alignment and inter-dimensional alignment using similar techniques as Li and Chen (see Section 3.1.1). However, inter-dimensional *permutations* are not supported. Arrays may be mapped differently in different sections of the program [KN90]. The described techniques work only on single procedures. However, they handle complex control flow.

Since the CM-2 supports the concept of virtual processors through its programming environment, data alignment is sufficient to specify the data layout. In contrast, the MasPar machine does not support virtual processors. The virtual processors have to be mapped onto the physical processors explicitly [Wei91].

The alignment algorithm is based on the usage patterns of arrays and Fortran 8x array sections in the source program. Each pattern generates allocation requests, called *preferences*, that indicate the optimal layout of the arrays relative to each other [KLS88, KLS90]. An *identity preference* exists between corresponding dimensions of a definition and a use of the *same* array. It describes a preference to allocate identical elements of the array on the same processors for the two textual occurrences. A *true dependence* exists between the definition and the use that generate the identity preference. A *conformance preference* is introduced between corresponding dimensions of textual occurrences of *different* arrays if they are operated on together. It indicates a preference to allocate corresponding elements of distinct arrays on the same processor. An *independence anti-preference* is associated with a *single* dimension of an array occurrence. It expresses the preference to allocate the array dimension across the processors in order to exploit the data parallelism in this dimension. More recently, Knobe, Lukas, and Dally introduced the concept of a *control preference*. A control preference exists between the corresponding dimensions of an array in a conditional expression and an array occurrence in an operation that is control dependent on this expression [KLD92].

The preferences of the program are represented by the undirected *preference graph* where the arcs correspond to the preferences and the nodes are dimensions of textual occurrences of arrays and array sections. Each edge is labeled with a cost that reflects the performance penalty that occurs if

the preference is not honored, i.e. not honoring identity and conformance preferences may lead to communication while unhonored independence anti-preferences potentially reduce the exploitable parallelism. The cost functions take the structure of the program into account. Conflicting alignment requirements can only occur in strongly connected components of the preference graph. To locate the cycles, a spanning tree is constructed, using a greedy algorithm that chooses the next arc to add by finding the highest cost arc that is not already processed. If a cycle-creating arc induces a conflict, the corresponding preference will not be honored [KLS90]. Knobe and Natarajan have extended this algorithm to optimize the communication resulting from unhonored identity and conformance preferences [KN90].

For the MasPar machine, the data allocation functions generated by the data optimization component have to be transformed from mappings based on virtual processors to mappings based on physical processors. Weiss discusses three distribution schemes, namely cyclic (*horizontal*), block (*vertical*), and block-cyclic distributions [Wei91].

Most of the described work has been implemented as part of the CM-2 and MP-1 Fortran compilers developed at Compass. The authors report significant performance improvements of up to a factor of 60 due to using the compiler generated data mapping instead of the naive, canonical mapping. The performance numbers are given for a few computational kernels that were hand-compiled and hand-simulated on the CM-2 and MP-1. Performance figures of actual runs are not reported.

#### 3.2.2 Chatterjee, Gilbert, Schreiber, and Teng at RIACS, Xerox PARC, and MIT

Chatterjee, Gilbert, Schreiber, and Teng discuss a framework for automatic alignment in an array-based, data-parallel language such as Fortran90 [CGST93, CGST92, GS91]. They provide algorithms for automatic alignment of arrays in a single basic block. Each intermediate result of a computations in a basic block is assigned to a temporary array. This allows intermediate results to be mapped explicitly. The basic block may contain explicit communication such as transposes, spreads, or reductions. Alignment functions for each of the array dimensions are restricted to linear functions of a single, distinct induction variable. Diagonal alignments are not possible.

A weighted directed acyclic graph (DAG) represents the computation in each basic block. Internal nodes represent operations and are labeled with names of temporary arrays. Edges are directed from the nodes representing the operands to the node representing the operator. Each edge is labeled with a nonnegative integer $w$ equal to the size of the data object at its source. A *position space* models all possible alignments of an array onto a decomposition (Fortran D terminology). A distance $d$(p,q) between two positions p and q is a nonnegative number describing the cost per unit data of changing positions from p to q. Different distance metrics $d$ are used to model communication characteristics of the target machine. Distance metrics cover machine topologies such as grids, rings, and fat-trees.

Alignment analysis is done in two separate steps. Inter-dimensional (axis) alignment and stride alignment is performed, followed by offset-alignment. Each step uses a different distance metric $d$. If the arrays at the sink and source of an edge in the DAG cannot be aligned, a communication cost of $w * d$(p,q) will occur, assuming the sink and source arrays are at position p and q, respectively, and $w$ is the edge label. A solution to the alignment problem minimizes the cost of all edges that are not aligned.

The authors discuss a variety of distance metrics and give asymptotically efficient solutions to the corresponding alignment problems. Algorithms are given for solving the alignment problem for a DAG where the alignment of the arrays at the leaf nodes is given as input (fixed-source variant) or

has to be chosen by the algorithm (free-source variant). The complexity of the algorithms depend on the characteristics of the metric used and the structure of the DAG, namely whether it is a forest or not. Some variations of the problem are shown to be NP-complete. The authors show how to extend their approach for single basic blocks across basic blocks. Their technique uses traces and a combination of free-source and fixed-source alignment algorithms.

The presented work is a big step towards the theoretical foundation of the alignment problem and discusses a variety of algorithms for its solution. The algorithms handle dynamic realignment. However, it is not clear, how these algorithms will work on real application programs. Experimental results on the applicability and efficiency of the algorithms, and efficiency of their produced alignments using real programs are not reported. Many of the listed examples are rather contrived.

Introducing temporary arrays has the advantage that intermediate values are named and therefore can be mapped explicitly to avoid an inefficient mapping due to the owner computes rule. This is often referred to as "relaxing the owner-computes rule". In the SIMD model of execution, array temporaries must be introduced by the compiler for intermediate values [CGST93]. The possibility of using these temporaries to relax the owner computes rule comes therefore 'for free'. This is not true for the compilation for MIMD machines with scalar node processors. The node compiler will generate the necessary temporaries. We are planning to introduce temporaries only if we expect a significant performance improvement by relaxing the owner computes rule.

### 3.3 Discussion

The presented works differ significantly in the assumptions that are made about the input language, the possible set of data decompositions, the compilation system, and the target distributed-memory machine.

The initial work in automatic data layout was done at Yale University and by Compass. Both groups concentrated on the problem of automatic data alignment. While Crystal uses an MIMD programming model and a purely functional input language, the Compass compilers are based on an SIMD programming model and Fortran8X. These differences lead to distinct alignment algorithms that stress inter-dimensional alignment in Crystal and intra-dimensional alignment in the Compass compilers. Crystal performs distribution analysis based on a simple cost model that takes machine characteristics and problem sizes into account. Only Compass' MasPar-1 compiler performs a simple form of distribution analysis. The machine model for the CM-2 using PARIS does not require data distribution.

Subsequent projects done by Gupta and Banerjee at the University of Illinois and by Wholey at Carnegie Mellon University base their alignment analysis on Crystal's and Compass' alignment analysis, respectively. In both cases, the distribution analysis models the costs for each statement under a set of possible data distributions. Machine characteristics and problem sizes are considered. Wholey performs distribution analysis at runtime allowing the machine size and problem size to be unknown at compile time. Chatterjee, Gilbert, Schreiber, and Teng provide a theoretical foundation for the alignment problem by providing a framework in which inter- and intra-dimensional alignment problems can be formulated and efficiently solved for basic blocks. Sussman demonstrates that static performance estimation can be used to efficiently predict the tradeoffs between different data mappings in the special case of a mapping compiler for a functional language targeted for a systolic array or processors.

Superb-2, a tool being developed at the University of Vienna, proposes automatic data decomposition as part of an interactive parallelization system. Their approach is based on the work done by Gupta and Banerjee. In addition, performance estimation uses pattern matching and a

knowledge data base. Inter-procedural data decomposition analysis is planned since the Superb interactive compilation system performs inter-procedural optimizations.

ParaSoft's ASPAR system and $P^3C$ at Tel-Aviv University do not give detailed descriptions of their approaches to automatic data decomposition. Both recognize and determine data mappings for stencil computations.

The published work on automatic data decomposition does not show that the presented algorithms and techniques will work well for real application programs. Only very few experiments were conducted to validate the different approaches. In many cases, no experimental results are given at all. The majority of the discussed techniques assume a simple compilation system that does not perform inter-procedural optimizations or optimizations that exploit existing parallelism by pipelining computations. In many cases, the set of possible data decomposition schemes is restricted. Dynamic data decomposition is not addressed in the context of an MIMD programming model. The relationship between data decompositions and memory requirements of the compiler generated node program is not considered.

The major difference between the previous work and my thesis work is that my approach to automatic data decomposition will assume an advanced compilation system that performs extensive intra and inter-procedural optimizations. My tool has to understand the implication of a data decomposition scheme on the optimizations performed by the compiler as well as on the performance of the compiler generated node program running on the target distributed-memory machine. I will develop new technique for performance estimation based on training sets that will be able to handle this complexity.

Previous work only deals with the problem of dynamic data alignment and only discusses a restricted form of data replication. The importance and complexity of dynamic data alignment and distribution is illustrated in Appendix B. My techniques will handle dynamic data decomposition and data replication in the presence of control flow.

Current supercomputers are used not only for their ability to perform millions of floating point operations per second, but also for the size of main memory that they provide. Dynamic data remapping can be used to reduce the memory requirements of a program. My tool may detect situations where dynamic data mapping is necessary due to the otherwise non-satisfiable memory requirements of the program.

In the context of the SIMD model of execution, using the owner computes rule can lead to inefficient code. I will investigate the benefits of relaxing the owner computes rule for the MIMD model of execution.

An essential contribution of my thesis will be the the validation of my automatic tool using a benchmark suite of real programs and program kernels written in a data-parallel programming style. It is important to note that I do not validate my techniques based for 'dusty deck' programs or programs written for a specific machine architecture. When automatic techniques will fail to find a good data layout, the user needs to be involved in the process of data mapping. I will investigate how the user can interact with the compilation system to find a good data layout.

# 4 Research Plan

In my thesis work I will design, implement and validate new techniques for automatic data decomposition. An initial assumption of this work is that for each program the problem size and the number of processors to be used are known at compile time. The following subsection contains a brief description of our overall research plan, the remainder discusses each research problem in more detail.

## 4.1 Overview

The proposed tool will focus on regular problems which are *loosely synchronous* or result in wavefront style computations [Lam74]. Loosely synchronous problems represent a large class of scientific computations [FJL+88]. They can be characterized by computation intensive regions that have substantial parallelism, with a synchronization point between the regions.

### 4.1.1 Automatic data decomposition

Our approach to data decomposition is based on the assumption that a good data decomposition for the entire program can be found by successively decomposing the data for smaller program segments, and realignment/redistribution when necessary between the segments. A program *phase* is the basic unit of this hierarchical approach. The definition of a phase is given in section 4.2. For each phase the set of locally efficient data decomposition schemes is determined. Subsequently, the local decompositions are merged in a hierarchical fashion until the global decomposition scheme for the entire program is found.

Under the above assumptions, we will develop algorithms and techniques that perform the following steps:

1. Define, identify, and represent program phases. The identification of a local phase may require inter-procedural analysis. Its definition will have a major impact on the quality of the generated decompositions and on the speed of the tool itself.

2. Build a search space of reasonable, local decompositions and search it efficiently. The search space is determined by intra and inter-dimensional alignment analysis, followed by distribution analysis. The search space contains schemes where data has been replicated. New source-to-source level transformations are considered that allow to relax the owner computes rule. The search algorithm employs a search space pruning heuristic.

3. Estimate the performance of a local decomposition at the Fortran D source level. The estimator has to evaluate the relative performance of local decomposition schemes for each phase or local sequence of phases. Performance is defined as execution time and memory requirements. The estimator must understand the performance implications of compiler transformations such as coarse-grain pipelining that overlap communication with computation. The control flow in a phase has to be considered.

4. Merge the local decompositions into a single decomposition scheme for the entire program. The profitability of realignment and redistribution has to be considered. The algorithm will require inter-procedural analysis. The control flow between phases has to be taken into account.

We will discuss each of the subproblems in more detail after this overview section.

### 4.1.2 Validation of Automatic Data Decomposition

The feasibility of the proposed automatic techniques depends on the 'quality' of the generated decomposition schemes compared to schemes that are considered optimal. For an unbiased discussion of results gained by an empirical study, we want to define our quality measure prior to an investigation into automatic techniques.

We will use a benchmark suite being developed by Geoffrey Fox at Syracuse that consists of a collection of Fortran programs and program kernels written in a data-parallel programming style

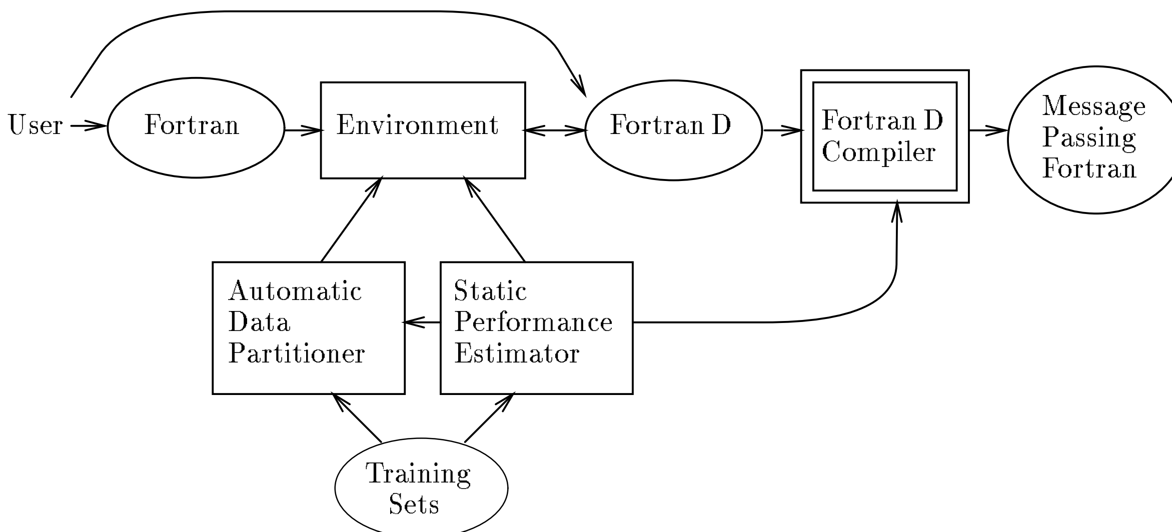


Figure 2: Fortran D Parallel Programming System

[MFvL+92]. In addition, we will examine parts of the version of UTCOMP that is written in a data-parallel programming style. UTCOMP is 33,000 line oil reservoir simulation code developed at the University of Austin, Texas.

#### 4.1.3 Interactive Programming Environment

For some programs we expect the automatic tool not to generate decomposition schemes that satisfy our quality measure. In these cases we want to understand why the techniques failed. We want to investigate how user interaction can help to overcome the deficiencies of our techniques. We will investigate possible extensions to Fortran D that allow the partial specification of a data layout. For instance, language annotations can be used to restrict the search space of possible decomposition schemes considered by the automatic tool.

The interactive system allows the user to browse through the search space of decomposition schemes considered by the automatic tool. For each decomposition scheme, the tool provides information such as the type and location of communication operations generated by the compiler and the overall performance estimates. The usefulness of such a tool has been recognized in the final report of the findings of the Pasadena Workshop on System Software and Tools for High-Performance Computing Environments [SMC+92].

#### 4.1.4 Implementation

The proposed automatic techniques will be implemented as part of the ParaScope parallel programming environment adapted to distributed memory multiprocessors [BKK+89, KMT91, BFKK90, HKK+91]. An overview of the system is shown in Figure 2. The system is part of the D Environment currently under development at Rice University [CCH+92]. A prototype of the machine module of the static performance estimator is available [BFKK91].

## 4.2 Program Phases

The analysis performed by the automatic data partitioner divides the program into separate *computation phases.* A phase is a syntactic entity. Phases try to identify program segments that perform operations on entire data objects. Dynamic realignment and redistribution is allowed only between phases. In the absence of procedure calls we define a phase as follows: A phase is a loop nest such that for each induction variable that occurs in a subscript position of an array reference in the loop body the phase contains the surrounding loop that defines the induction variable. The definition in the presence of procedure calls is given later. A phase is minimal in the sense that it does not include surrounding loops that do not define induction variables occurring in subscript positions.

For example, the red-black program in Figure 10 in appendix C has four phases (the inner loop nests) enclosed in the outer `k`-loop. Each phase has an associated Fortran D decomposition of dimensionality and size equal to the array `v`.

## 4.3 Static Performance Estimation

It is clearly impractical to use dynamic performance information to choose between data decompositions in our programming environment. Instead, a *static* performance estimator is needed that can predict the performance of a Fortran D program on the target machine as accurately as possible. This performance estimator is not based on a general theoretical model of distributed-memory computers. Instead, it employs the notion of a *training set* of kernel routines that measures the cost of various computation and communication patterns on the target machine. The results of executing the training set on a parallel machine are summarized and used to adjust the performance estimator for that machine. By utilizing training sets, the performance estimator achieves both accuracy and portability across different machine architectures.

The static performance estimator is divided into two parts, a compiler module and a machine module. The *compiler module* predicts the performance at the source language level while the *machine module* estimates the performance at a level where the decomposition scheme is already 'hard coded' into the program, i.e. at the node program level containing explicit communications.

### 4.3.1 Machine Module

The *machine module* predicts the performance of a node programs with explicit communications. It uses a *machine-level* training set written in message-passing Fortran. The training set contains individual computation and communication patterns that are timed on the target machine for different numbers of processors and data sizes. To estimate the performance of a node program, the machine module can simply look up results for each computation and communication pattern encountered.

Note that the static performance estimator does not need to predict the *absolute* performance of a given data decomposition to assist automatic data decomposition. Instead, it only needs to accurately predict the performance *relative* to other data decompositions. In many cases the accurate prediction of the *crossover point* at which one data decomposition scheme is preferable over another will be sufficient.

We implemented a prototype of the machine module for the large class of loosely synchronous scientific problems. Techniques to estimate the performance of programs where communication and computation overlap, for instance pipelined computations, have to be developed as part of this thesis.

The prototype predicts the performance of a node program using Express communication routines for different numbers of processors and data sizes [EXP89]. The prototype performance estimator has proved quite precise, especially in predicting the relative performances of different data decompositions [BFKK91]. Our experience with applying the prototype estimator to a point-wise red-black relaxation routine is given in Appendix C. In the context of shared-memory programming, Kennedy, McIntosh, and McKinley have used our training set approach to estimate the performance of entire programs with do-loop parallelism [KMM91].

#### 4.3.2 Compiler Module

The *compiler module* forms the second part of the static performance estimator. It predicts the performance of a program at the source level for a set of data decomposition schemes. The compiler module employs a *compiler-level* training set written in the source language that consists of common computation patterns and program kernels such as stencil computations and matrix multiplication. Note that in contrast to pattern matching algorithms as they are used in compilers for optimization or code generation [ASU86, LC90b, PP91], our pattern matcher does not need to preserve the semantics of the program. Program segments are considered 'equivalent' if their corresponding compiler generated code exhibit a similar performance behavior. The training set itself as well as the pattern matching algorithm has to be developed and validated as part of this thesis.

The training set is converted into message-passing Fortran using the Fortran D compiler and executed on the target machine for different data decompositions, numbers of processors, and array sizes. Estimating the performance of a Fortran D program then requires matching computations in the program with computation patterns from the training set.

Since it is not possible to incorporate all possible computation patterns in the compiler-level training set, the performance estimator will encounter code fragments that cannot be matched with existing kernels. To estimate the performance of these codes, the compiler module must rely on the machine-level training set. We plan to incorporate elements of the Fortran D compiler in the performance estimator so that it can mimic the compilation process. The compiler module can thus convert any unrecognized Fortran D program fragment into an equivalent node program and invoke the machine module to estimate its performance.

#### 4.3.3 Intra-Phase Alignment and Distribution

The *intra-phase* decomposition problem consists of determining a set of good data decompositions and their performance for each individual phase. The data partitioner first performs alignment analysis to determine a set of reasonable alignment schemes for the entire program. Subsequently, the automatic tool tries to match phases or a sequence of phases with computation patterns in the compiler training set. If a match is found, it returns the set of distributions with the best measured performance as recorded in the compiler training set. If no match is found, the data partitioner must perform distribution analysis on the phase. The resulting solution may be less accurate since the effects of the Fortran D compiler and target machine can only be estimated.

*Alignment analysis* is used to prune the search space of possible array alignments by selecting only those alignments that minimize data movement. Alignment analysis is largely machine-independent; it is performed by analyzing the array access patterns of computations in the entire program. Interprocedural techniques will be developed that identify good alignment schemes.

In a source-to-source transformation step, temporary arrays are introduced to facilitate the relaxation of the owner computes rule by naming intermediate results of computations explicitly. This process is selective and not as general as described in [CGST93] where all intermediate results

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

DECOMPOSITION d1(N)
ALIGN a, b, c WITH d1
DISTRIBUTE d1(BLOCK)

. . .

DO i = 1, N-1
      a(i) = F ( b(i+1), c(i+1) )
ENDDO
```

(A)

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

DECOMPOSITION d1(N)
ALIGN a, b, c, temp WITH d1
DISTRIBUTE d1(BLOCK)

. . .

DO i = 1, N-1
      temp(i+1) = F ( b(i+1), c(i+1) )
      a(i) = temp(i+1)
ENDDO
```

(B)

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

DECOMPOSITION d2(N, N)
ALIGN a, b, c WITH d2
DISTRIBUTE d2(BLOCK,:)

. . .

DO j = 1, N
      DO i = 1, N
            a(i, j) = F ( b(j, i), c(j, i) )
      ENDDO
ENDDO
```

(C)

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

DECOMPOSITION d2(N, N)
ALIGN a, b, c, temp WITH d2
DISTRIBUTE d2(BLOCK,:)

. . .

DO j = 1, N
      DO i = 1, N
            temp(j, i) = F ( b(j, i), c(j, i) )
            a(i, j) = temp(i, j)
      ENDDO
ENDDO
```

(D)

Figure 3: Opportunities to relax owner computes rule by insertion of temporary variables

of a computation are assigned to newly introduced temporary data objects. In contrast to SIMD architectures, such as the CM-2 or MP-1, relaxing the owner computes rule for computations that require nearest-neighbor communication is not profitable on most MIMD machines. An example of such a situation is given in Figure 4.3.3. By introducing a temporary array in (B), one `cshift` operations can be saved as compared to the two `cshift` operations necessary in (A). On an MIMD machine, in terms of execution time, it does not make a significant difference whether 4 or 8 bytes are communicated between neighboring processors in case (A) or (B), respectively. A contrived example where relaxing the owner computes rule is profitable on an MIMD machine is given in Figure 4.3.3. The example assumes that due to the surrounding context of the loop, the arrays `a`, `b`, and `c` have to be aligned. In case (C), arrays `b` and `c` have to be transposed. In contrast, only the transpose of `temp` is necessary in case (D).

We intend to build on the inter-dimensional and intra-dimensional alignment techniques of Li and Chen [LC90a], Knobe *et al.* [KLS90], and Chatterjee, Gilbert, Schreiber, and Teng [CGST93]. The alignment problems can be formulated as an optimization problem on an undirected, weighted

graph. Some instances of the problem of alignment have been shown to be NP-complete [LC90a, CGST93]. One major challenge in our proposed work will be to define the appropriate weights of the graph and to develop an interprocedural algorithm that solves the alignment problem in a way that is suitable for loosely synchronous problems. Note that our alignment algorithm will not necessarily determine a single alignment scheme but may generate a set of possible alignment schemes in the case of alignment conflicts.

*Intra-Phase distribution analysis* follows alignment analysis. It applies heuristics to prune unprofitable choices in the search space of possible distributions for each single phase. Distribution analysis is compiler, machine, and problem dependent. For instance, the compiler may not be able to generate efficient wavefront computations [Lam74] for a subset of distributions. Transformations such as loop interchange and strip-mining can substantially improve the degree of parallelism induced by the wavefront [HKT91]. A consideration in our pruning heuristic are the sizes of the dimensions of the decomposition. If the size of a dimension is smaller than a machine dependent threshold, the dimension will always be localized. This eliminates all distributions that map small dimensions to distinct processors. We will also restrict the possible block sizes in `BLOCK_CYCLIC` distributions to a reasonable subset of values.

After the automatic data partitioner has determined a set of reasonable data decomposition schemes, the static performance estimator is invoked to predict the performance of each reasonable scheme.

### 4.4 Inter-Phase Distribution

After computing data decomposition schemes for each phase, the automatic data partitioner must solve the *inter-phase* decomposition problem of merging individual data decompositions. It considers realigning and redistribution of arrays between computational phases to reduce communication costs or to improve the available parallelism.

The merging problem can be formulated as a single-source shortest paths problem over the *phase control flow graph*. The phase control flow graph is similar to the control flow graph [ASU86] where all nodes associated with a phase are substituted by nodes representing the set of reasonable data decomposition schemes for the phase. The static performance estimator is used to predict the costs for these reasonable decomposition schemes. The availability of fast collective communication routines will be crucial for the profitability of realignment and redistribution.

The merging problem for a linear phase control flow graph can be solved as a single-source shortest paths problem in a directed acyclic graph [CLR90]. For example, Figure 4 shows a three phase problem with four reasonable decompositions for each phase. Each decomposition scheme is represented by a node. The node is labeled with the predicted cost of the decomposition scheme for the phase. Edges between phases are labeled with the realignment and redistribution costs for the source and sink decomposition schemes. For each of the four decompositions of the first phase we will solve the single-source shortest paths problem. In general, let $k$ denote the maximal number of decomposition schemes for each phase and $p$ the number of phases. The resulting time complexity is $O(p \times k^3)$.

The merging of phases in a strongly connected component of the phase control flow graph should be done before merging any of its phases with a phase outside of the strongly connected component. This suggests a hierarchical algorithm for merging phases based on, for example, Tarjan intervals [Tar74]. Assuming that the innermost loop bodies can be represented by a linear phase control flow subgraph, the merging problem is solved by adding a shadow copy of the first phase after the last phase in the linear subgraph keeping the subgraph acyclic. After solving the single-source shortest

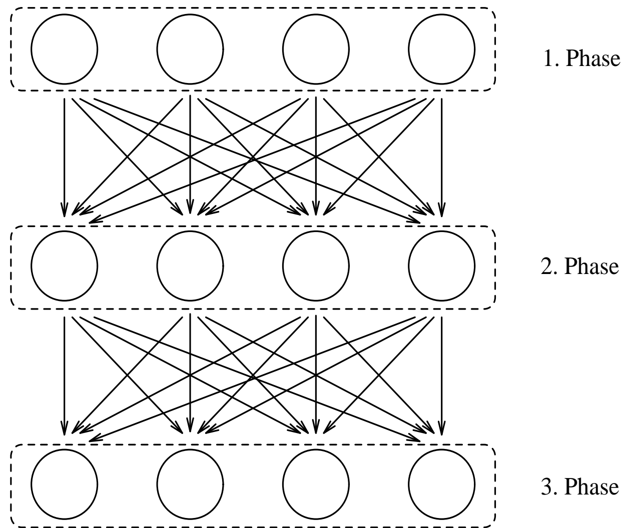


Figure 4: Interphase merge problem with realignment and redistribution

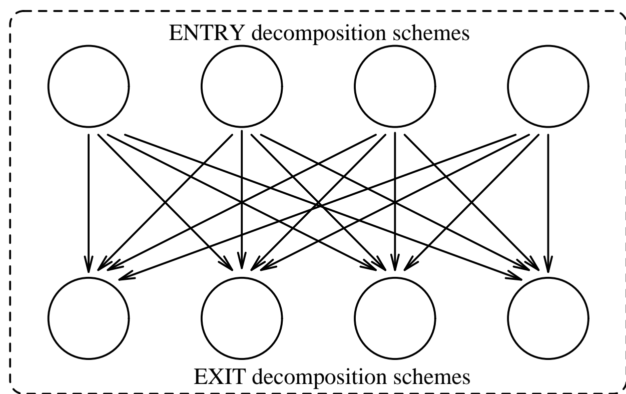


Figure 5: An example interval summary phase

paths problem, the subgraph is collapsed into a single *interval summary phase* representing the different costs resulting from entering the interval with a decomposition scheme and exiting it with a possibly different scheme. An example interval summary phase is shown in Figure 5. In the resulting phase control flow graph we again identify the innermost loops and repeat the process of collapsing and summarizing until the phase control flow graph consists of a single node.

In fully automatic mode, the data partitioner selects the decomposition scheme that has the minimal cost for the shortest path from a decomposition in the first phase to a decomposition in the last phase of the selected program segment. Following the selected shortest path, `ALIGN`, and `DISTRIBUTE` statements are inserted if the decomposition at the source of an edge is different from the decomposition at the sink. `DECOMPOSITION` specifications are declared at the beginning of the subroutine containing the selected program segment.

### 4.4.1 The Algorithm

Figure 6 gives the basic algorithm for automatic data decomposition for a program segment without procedure calls. The algorithm does not handle control flow other than loops and does not consider data replication.

---

**Algorithm DECOMP**

*Input*: program segment without procedure calls; problem sizes and number of processors to be used.
*Output*: data decomposition schemes for data objects referenced in the input program segment with diagnostic information.

perform alignment analysis for input program;
determine program phases of input program;
build phase control flow graph;
*for each* phase *do*
    perform distribution analysis;
    generate diagnostic information, if available;
*endfor*
*while* phase control flow graph contains a loop *do*
    identify innermost loop (e.g. using Tarjan intervals);
    solve single-source shortest paths problem for this loop;
    identify realignment and redistribution points;
    generate diagnostic information, if available;
    substitute loop by its interval summary phase in the phase control flow graph;
*endwhile*
use the computed shortest path to generate `DECOMPOSITION`, `ALIGN`,
and `DISTRIBUTE` statements, if in fully automatic mode;

Figure 6: Basic Algorithm for Automatic Data Alignment and Distribution

---

### 4.4.2 Automatic Decomposition in the Presence of Procedure Calls

One of the major challenges of the proposed research on automatic data decomposition is to devise techniques that can deal with procedure calls. Inter-procedural analysis is used to allow the merging of computation phases across procedure boundaries.

Inter-procedural phase merging is compiler dependent. In particular, the automatic data partitioner has to know whether and when the compiler performs inter-procedural optimizations such as procedure cloning [CHK92, HHKT91] or procedure inlining [Hal91]. In the following we will assume that the compiler performs procedure cloning for every distinct pattern of entry and exit decomposition schemes. To simplify our discussion we will assume that programs have only acyclic call graphs.

The augmented call graph is used to identify phases across procedure boundaries [HKM91]. Subsequently, the call graph is traversed in reverse topological order. For each procedure $P$ the single-source shortest paths problem is solved on its phase control flow graph using the hierarchical approach of algorithm DECOMP in Figure 6. Each call site of $P$ in procedure $Q$ is represented by a copy of the interval summary phase of $P$ in the phase control flow graph of $Q$.

If the compiler does not perform cloning a procedure can have only a single entry and exit decomposition scheme. The automatic data partitioner will use a heuristic to select the decomposition scheme. The heuristic takes the static execution count of call sites and the penalties due to mismatched decomposition schemes into account.

## 4.5 Validation

Our validation is based on a benchmark suite being developed by Geoffrey Fox at Syracuse and parts of an oil reservoir simulation code, called UTCOMP, developed at the University of Austin, Texas. Fox's program suite consists of a collection of Fortran programs and program kernels and is described in detail in [MFvL+92]. Each program in the suite will have five versions:

- **(v1)** the original Fortran 77 program,
- **(v2)** the best hand-coded message-passing version of the Fortran program,
- **(v3)** a "nearby" Fortran 77 program,
- **(v4)** a Fortran D version of the nearby program, and
- **(v5)** a Fortran 90 version of the program.

The "nearby" version of the program will utilize the same basic algorithm as the message-passing program, except that all explicit message-passing and blocking of loops in the program are removed. The Fortran D version of the program consists of the nearby version plus appropriate data decomposition specifications.

To validate the automatic data partitioner, we will use it to generate a Fortran D program from the nearby Fortran program (v3). The result will be compiled by the Fortran D compiler and its running time compared with that of the compiled version of the hand-generated Fortran D program (v4). Our goal is that for 80% of the benchmark programs the nearby version with automatically generated data layout will run at most 20% slower than the hand-generated Fortran D program. We expect that such a performance degradation due to automatic data decomposition is still acceptable to the user.

We believe that in many cases it is easy for the user to specify the *problem mapping*, i.e. the data alignment onto a decomposition, since the problem mapping is determined by the structure

of the underlying algorithm. However, the user will have difficulties to predict the combined effects of the compiler, problem, and machine characteristics on the performance of a specified data decomposition scheme. Therefore we expect the Fortran D program generated by the automatic data partitioner to do better than the hand-coded Fortran D version for some programs of the benchmark suite.

Initially, the automatic techniques will be validated by applying them to whole programs in the benchmark suite. Where automatic techniques will fail for whole programs, we want to understand why this is the case and how user interaction can overcome their deficiencies.

# 5 Spin-offs and Future Research

An obvious extension of our proposed work is to relax the restrictions that (1) programs cannot contain recursion and that (2) the problem size and number of processors used have to be known at compile time.

We believe that automatic data decomposition generates information that is very useful in the context of virtual shared memory systems. The decomposition scheme together with its induced communication can support the decision of how to efficiently place the data in the shared memory to improve the locality of data accesses and avoid *false sharing*.

An important future research topic is how to make our automatic data decomposition tool more compiler independent. It is desirable to have a tool that is able to adapt easily to changes in the underlying compilation system. One possible solution might be to include a parameterized compiler model in the tool. A training set approach to gather information about the compiler itself might help to solve this problem.

Since irregular problems represent a large class of real applications the applicability of our techniques to irregular problems should be investigated. It might be possible to use some of our algorithms at runtime to solve the automatic data mapping problem.

# 6 Summary

We believe that an efficient data layout can be automatically generated for many application programs that solve regular problems, if they are written in a data-parallel programming style. Previous research has been limited in many respects: the underlying compilation system does not perform extensive intra and inter-procedural analysis, the choice of possible decomposition schemes is restricted, or only a single machine architecture is targeted. Most techniques have not been sufficiently validated using real programs.

We will enhance the existing techniques for automatic data layout and develop techniques based on a new approach to static performance estimation. These new techniques will deal with dynamic data mapping and data replication, the complexity of advanced compilation systems such as Fortran D, and the possibility of a rich set of data mappings. A source-to-source transformation will allow the relaxation of the owner computes rule. We will validate our thesis by applying our tool to a test suite of programs and program kernels written in a data-parallel programming

style. We will use a benchmark suite developed by Geoffrey Fox at Syracuse and parts of a 33,000 line oil reservoir simulation code developed at the University of Austin, Texas. If automatic techniques fail to generate good data decomposition schemes, we want to answer the question how user interaction can help to overcome the deficiencies of our techniques. We propose an interactive system that allows the user to browse through the set of data decomposition schemes selected by the automatic system. For each scheme, the location and type of the compiler generated communication is presented to the user. We will consider language extensions to Fortran D as one form of user input to the compiler.

# References


[AKLS88] E. Albert, K. Knobe, J. Lukas, and G. Steele, Jr. Compiling Fortran 8x array features for the Connection Machine computer system. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)*, New Haven, CT, July 1988.

[ASU86] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, second edition, 1986.

[BFKK90] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[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.

[BGMZ92] P. Brezany, M. Gerndt, P. Mehrotra, and H. Zima. Concurrent file operations in a high performance FORTRAN. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[BKK+89] V. Balasundaram, K. Kennedy, U. Kremer, K. S. McKinley, and J. Subhlok. The ParaScope Editor: An interactive parallel programming tool. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[Bri92] P. Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.

[Car92] S. Carr. *Memory-Hierarchy Management.* PhD thesis, Rice University, September 1992.

[CCH+92] A. Carle, K. Cooper, M. Hall, K. Kennedy, , C. Koelbel, J. Mellor-Crummey, L. Torczon, and S. Warren. A software platform for parallel scientific programming. *Internal Report, Rice University*, 1992.

[CCL88] M. Chen, Y. Choo, and J. Li. Compiling parallel programs by optimizing performance. *Journal of Parallel and Distributed Computing*, 1(2):171–207, July 1988.

[CCL89] M. Chen, Y. Choo, and J. Li. Theory and pragmatics of compiling efficient parallel code. Technical Report YALEU/DCS/TR-760, Dept. of Computer Science, Yale University, New Haven, CT, December 1989.

[CGST92] S. Chatterjee, J.R. Gilbert, R. Schreiber, and S-H. Teng. Optimal evaluation of array expres-

sions on massively parallel machines. In *Proceedings of the Second Workshop on Languages, Compilers, and Runtime Environments for Distributed Memory Multiprocessors*, Bolder, CO, October 1992.

[CGST93] S. Chatterjee, J.R. Gilbert, R. Schreiber, and S-H. Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, Albuquerque, NM, January 1993.

[CH91] B.M. Chapman and H.M. Herbeck. Knowledge-based parallelization for distributed memory systems. In *First International Conference of the Austrian Center for Parallel Computation*, Salzburg, Austria, September 1991.

[CHK92] K. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the 1992 IEEE International Conference on Computer Language*, Oakland, CA, April 1992.

[CHZ91] B. Chapman, H. Herbeck, and H. Zima. Automatic support for data distribution. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[CKK89] D. Callahan, K. Kennedy, and U. Kremer. A dynamic study of vectorization in PFC. Technical Report TR89-97, Dept. of Computer Science, Rice University, July 1989.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[Clu89] The Perfect Club. The Perfect Club benchmarks: efficient performance evaluation of supercomputers. *Int. J. Supercomp. Appl.*, 3(3):5–40, 1989.

[D'H89] E. D'Hollander. Partitioning and labeling of index sets in do loops with constant dependence. In *Proceedings of the 1989 International Conference on Parallel Processing*, St. Charles, IL, August 1989.

[EXP89] Parasoft Corporation. *Express User's Manual*, 1989.

[Fah92] T. Fahringer. Private communication. 1992.

[FBZ92] T. Fahringer, R. Blasko, and H.P. Zima. Automatic performance prediction to support parallelization of Fortran programs for massively parallel systems. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[FHK+90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[GAY91] E. Gabber, A. Averbuch, and A. Yehudai. Experience with a portable parallelizing Pascal compiler. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[GB90] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. Technical Report CRHC-90-14, Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, October 1990.

[GB91] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[GB92] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, April 1992.

[Ger89] M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, December 1989.

[GS91] J.R. Gilbert and R. Schreiber. Optimal expression evaluation for data parallel architectures. *Journal of Parallel and Distributed Computing*, 13(1):58–64, September 1991.

[HA90] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[Hal91] M. W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, April 1991.

[HHKT91] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. Technical Report TR91-169, Dept. of Computer Science, Rice University, November 1991.

[HKK+91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[HKM91] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[HKT91] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[HKT92a] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[HKT92b] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier, Amsterdam, The Netherlands, to appear 1992.

[IFKF90] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[KKBP91] D. Kulkarni, K. Kumar, A. Basu, and A. Paulraj. Loop partitioning for distributed memory multiprocessors as unimodular transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[KLD92] K. Knobe, J.D. Lukas, and W.J. Dally. Dynamic alignment on distributed memory systems. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 1992.

[KLS88] K. Knobe, J. Lukas, and G. Steele, Jr. Massively parallel data optimization. In *Frontiers88: The 2nd Symposium on the Frontiers of Massively Parallel Computation*, Fairfax, VA, October 1988.

[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.

[KM91] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[KMM91] K. Kennedy, N. McIntosh, and K. S. McKinley. Static performance estimation. Technical Report TR91-174, Dept. of Computer Science, Rice University, December 1991.

[KMT91] K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[KMV90] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[KN90] K. Knobe and V. Natarajan. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[KZBG88] U. Kremer, H. Zima, H.-J. Bast, and M. Gerndt. Advanced tools and techniques for automatic parallelization. *Parallel Computing*, 7:387–393, 1988.

[Lam74] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[LC90a] 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.

[LC90b] J. Li and M. Chen. Synthesis of explicit communication from shared-memory program references. Technical Report YALEU/DCS/TR-755, Dept. of Computer Science, Yale University, New Haven, CT, May 1990.

[LC91a] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[LC91b] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(4):213–221, August 1991.

[Li92] J. Li. Private communication. 1992.

[MFvL+92] A. Mohamed, G. Fox, G. von Laszewski, M. Parashar, T. Haupt, K. Mills, Y-H. Lu, N-T. Lin, and N-K. Yeh. Applications benchmark set for Fortran D and High Performance Fortran.

Technical Report SCCS-327, NPAC, Syracuse University, October 1992.

[PP91] S. Pinter and R. Pinter. Program optimization and parallelization using idioms. In *Proceedings of the Eighteenth Annual ACM Symposium on the Principles of Programming Languages*, Orlando, FL, January 1991.

[PSvG91] E. Paalvast, H. Sips, and A. van Gemund. Automatic parallel program generation and optimization from data decompositions. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[RA90] R. Ruhl and M. Annaratone. Parallelization of FORTRAN code on distributed-memory parallel processors. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[Ram90] J. Ramanujam. *Compile-time Techniques for Parallel Execution of Loops on Distributed Memory Multiprocessors*. PhD thesis, Department of Computer and Information Science, Ohio State University, Columbus, OH, 1990.

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[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.

[RSW88] M. Rosing, R. Schnabel, and R. Weaver. Dino: Summary and examples. Technical Report CU-CS-386-88, Dept. of Computer Science, University of Colorado, March 1988.

[SMC+92] T. Sterling, P. Messina, M. Chen, F. Darema, G. Fox, M. Heath, K. Kennedy, R. Knighten, R. Moore, S. Ranka, J. Saltz, L. Tucker, and P. Woodward. Workshop on system software and tools for high performance computing environments. In *Final Report on the Findings of the Workshop*, Pasadena, CA, April 1992.

[SS90] L. Snyder and D. Socha. An algorithm producing balanced partitionings of data arrays. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[Sus91] A. Sussman. *Model-Driven Mapping onto Distributed Memory Parallel Computers*. PhD thesis, School of Computer Science, Carnegie Mellon University, September 1991.

[Sus92] A. Sussman. Model-driven mapping onto distributed memory parallel computers. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[Tar74] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[TMC89] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, version 5.2-0.6 edition, September 1989.

[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.

[Wei91] M. Weiss. Strip mining on SIMD architectures. In *Proceedings of the 1991 ACM International*

*Conference on Supercomputing*, Cologne, Germany, June 1991.

[Who91] S. Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991.

[Who92a] S. Wholey. Automatic data mapping for distributed-memory parallel computers. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[Who92b] S. Wholey. Private communication. 1992.

[Wol89] M. J. Wolfe. Semi-automatic domain decomposition. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[Wol92] M.E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, August 1992.

[ZBG88] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

# A Data-Parallel Programming Style

UTCOMP is a fully 3-D compositional reservoir simulation package developed at the Petroleum Engineering Department at UT-Austin for vector processors. DISPER is a 500 non-comment lines routine in UTCOMP which computes dispersion tensor terms. DISPER contains a variety of stencils that lead to different communication patterns.

The original version of the routine has been written for the CRAY. A code fragment is shown in Figure 7. The code contains indirections due to the linear representation of a 3-dimensional oil reservoir and the fact that depending on runtime values different stencils are used in the computation. These indirections prohibit the detection and optimization of communication patterns at compile time. To convert the routine into data-parallel style, we delinearized arrays and removed indirect addressing by using code replication and other techniques[1]. The resulting program is 1500 lines long (non-comment lines). We added Fortran D data layout specifications and compiled the program using the current implementation of the Fortran D compiler. Figure 8 shows the data-parallel version of the code fragment of Figure 7 with Fortran D data layout specifications.

The performance of the compiler-generated code on a problem of size $(8 \times 256 \times 8)$ is given in Table 1. For this problem, only the logically second dimension was distributed across processors. The second dimension of the oil reservoir is represented as the first dimension in the associated data structures.

| machine | memory/proc | compiler | #procs | exec time |
|---|---|---|---|---|
| Sparc2 | 64Mbytes | f77 -O4 | | 30.30 secs |
| RS6000 | 192Mbytes | xlf -O | | 17.74 secs |
| iPSC/860 | 16Mybtes | if77 -O4 | 2 nodes | 19.67 secs |
| iPSC/860 | 8Mybtes | if77 -O4 | 4 nodes | 9.96 secs |
| | | | 8 nodes | 5.03 secs |
| | | | 16 nodes | 2.44 secs |
| | | | 32 nodes | 1.29 secs |

Table 1: Performance of data-parallel version of UTCOMP dispersion tensor routine

The numbers show a nearly linear speed-up. As a comparison, the table also contains the performance numbers of the data-parallel style program on the Sparc2 and RS6000.

[1] joined work with Marcelo Ramé at Rice University

```
c
      subroutine disper
cccccc.........................................................ccc
c     purpose: computes the dispersion term.
c     calls:   none
cccccc.........................................................ccc
      implicit real*8 ( a-h, o-z )
c
      parameter ( npm = 4 )
      parameter ( ncm = 5 )
      parameter ( nbm = 8 * 256 * 8 )
      parameter ( ncmp1 = ncm + 1 )
c
      common /com100/ nb, nx, ny, nz, nw, np, nc, ncp1, nbwp
      common /com210/ nblk(nbm,6), nblkup(nbm,npm,3)
      common /com440/ ddx(nbm), ddy(nbm), ddz(nbm)
      common /com470/ porstd(nbm), por(nbm), vb(nbm),
      common /com480/ diffun(npm,ncmp1), tau, alphal(npm), alphat(npm)
      common /com510/ sat(nbm,npm), denml(nbm,npm), denms(nbm,npm),
      logical lsat, lomfr, lpmfr
      common /com515/ lsat(nbm,npm), lomfr(nbm,ncm), lpmfr(nbm,npm,ncm)
      common /com540/ omfr(nbm,ncm), pmfr(nbm,npm,ncm),
      common /com580/ trxcof(nbm), trycof(nbm), trzcof(nbm)
      common /com595/ velx(nbm,npm), vely(nbm,npm), velz(nbm,npm)
      common /com850/ disp(nbm,ncmp1), dispx(nbm), dispy(nbm),
     &                                dispz(nbm)
      common /com870/ coexx(nbm,npm), coexy(nbm,npm), coexz(nbm,npm),
     &                coeyx(nbm,npm), coeyy(nbm,npm), coeyz(nbm,npm),
     &                coezx(nbm,npm), coezy(nbm,npm), coezz(nbm,npm)
c
c
          .
          .
          .

      do 900 k = 1, nc

        do 850 j = 2, np
          do 840 i = 1, nb

            ilyp1 = nblk(i,2)

            iupy = nblkup(i,j,1)

c
c           ------
c           y-direction
c           ------
c
            if ( ( ilyp1.gt.0 ).and.( lsat(i,j) )
     &                        .and.( lsat(ilyp1,j) ) ) then
              grady = 2. * ( pmfr(ilyp1,j,k) - pmfr(i,j,k) )
     &                   / ( ddy(i) + ddy(ilyp1) )
              dispy(i) = dispy(i)
     &                 + denml(iupy,j) * ( por(iupy) * sat(iupy,j)
     &                                 * diffun(j,k) + coeyy(i,j) )
     &                                 * grady
            endif
            .
            .
            .

 840      continue
 840    continue
            .
            .
            .

 900  continue
```

Figure 7: Cray code fragment

```
      double precision  ddx(256,8,8),ddy(256,8,8),ddz(256,8,8)
      double precision  por(256,8,8),pmfr(256,8,8,4,5),diffun(4,6)
      double precision  alphal(4),alphat(4),sat(256,8,8,4),denml(256,8,8,4)
       logical          lsat(256,8,8,4)
      double precision  trxcof(256,8,8),trycof(256,8,8),trzcof(256,8,8)
      double precision  potx(256,8,8,4),poty(256,8,8,4),potz(256,8,8,4)
      double precision  velx(256,8,8,4),vely(256,8,8,4),velz(256,8,8,4)
      double precision  disp(256,8,8,6),dispx(256,8,8),dispy(256,8,8),
     &                  dispz(256,8,8)
      double precision  coexx(256,8,8,4),coexy(256,8,8,4),coexz(256,8,8,4),
     &                  coeyx(256,8,8,4),coeyy(256,8,8,4),coeyz(256,8,8,4),
     &                   coezx(256,8,8,4),coezy(256,8,8,4),coezz(256,8,8,4)

c ———- FORTRAN D ————-
      integer    n$proc
      parameter (n$proc = 8)
c     decomposition dd(256)
c     align ltemp(i),grady(i)                      with dd(i)
c     align ddy(i,j,k),r(i,j,k),dispy(i,j,k)       with dd(i)
c     align poty(i,j,k,l),coeyy(i,j,k,l)           with dd(i)
c     align sat(i,j,k,l),lsat(i,j,k,l),denml(i,j,k,l)  with dd(i)
c     align pmfr(i,j,k,l,m)                        with dd(i)
c     distribute dd(block)
c ——————————————-
         . . .

      do 900 k = 1, 5

        do 850 j = 2, 4
          do 840 i3 = 1, 8
            do 840 i2 = 1, 8
              do 840 i1 = 1, 256

                if(poty(i1,i2,i3,j).lt.0) then
c                 ————–
c                 y-direction
c                 ————–
                  if ( i1.ne. 256 ) then
                    ltemp(i1) = lsat(i1,i2,i3,j).and.lsat(i1+1,i2,i3,j)
                    if ( ltemp(i1) ) then
                      grady(i1) = 2.*(pmfr(i1+1,i2,i3,j,k)
     &                           -pmfr(i1,i2,i3,j,k))
     &                          / ( ddy(i1,i2,i3) + ddy(i1+1,i2,i3) )
                      dispy(i1,i2,i3) = dispy(i1,i2,i3) + denml(i1,i2,i3,j)
     &                     * ( por(i1,i2,i3) * sat(i1,i2,i3,j)
     &                     * diffun(j,k) + coeyy(i1,i2,i3,j) )
     &                     * grady(i1)
                    endif
                  endif
                  . . .

                else
                  . . .

                  if ( i1.ne. 256 ) then
                    ltemp(i1) = lsat(i1,i2,i3,j).and.lsat(i1+1,i2,i3,j)
                    if ( ltemp(i1) ) then
                      grady(i1) =2.*(pmfr(i1+1,i2,i3,j,k)
     &                          - pmfr(i1,i2,i3,j,k))
     &                          / ( ddy(i1,i2,i3) + ddy(i1+1,i2,i3) )
                      dispy(i1,i2,i3) = dispy(i1,i2,i3)
     &                     + denml(i1+1,i2,i3,j) * ( por(i1+1,i2,i3)
     &                     * sat(i1+1,i2,i3,j)
     &                     * diffun(j,k) + coeyy(i1,i2,i3,j) )
     &                     * grady(i1)
                    endif
                  endif
                  . . .

                endif
                . . .

840           continue
850         continue

900       continue
```

Figure 8: Code fragment in data-parallel style with Fortran D data layout specifications

| #procs | problem size | butterfly | | | transpose | | | relative |
|---|---|---|---|---|---|---|---|---|
| | | total | communication only | (% of total) | total | communication only | (% of total) | speed-up |
| | 128 × 128 | | | | | | | |
| 2 | | 0.423 | 0.016 | 3.8% | 0.432 | 0.019 | 4.4% | **0.98** |
| 4 | | 0.272 | 0.061 | 22.4% | 0.217 | 0.015 | 6.9% | **1.25** |
| 8 | | 0.207 | 0.092 | 44.4% | 0.113 | 0.012 | 10.6% | **1.83** |
| 16 | | 0.187 | 0.119 | 63.6% | 0.062 | 0.011 | 17.7% | **3.02** |
| 32 | | 0.193 | 0.147 | 76.1% | 0.042 | 0.017 | 40.5% | **4.60** |
| 64 | | 0.160 | 0.124 | 77.5% | 0.035 | 0.022 | 62.8% | **4.57** |
| | 256 × 256 | | | | | | | |
| 2 | | 1.731 | 0.036 | 2.0% | 1.819 | 0.070 | 3.8% | **0.95** |
| 4 | | 0.979 | 0.119 | 12.1% | 0.903 | 0.050 | 5.5% | **1.08** |
| 8 | | 0.630 | 0.181 | 28.7% | 0.459 | 0.031 | 6.7% | **1.37** |
| 16 | | 0.485 | 0.238 | 49.0% | 0.237 | 0.023 | 9.7% | **2.05** |
| 32 | | 0.444 | 0.296 | 66.6% | 0.130 | 0.023 | 17.7% | **3.42** |
| 64 | | 0.352 | 0.250 | 71.0% | 0.081 | 0.026 | 32.0% | **4.34** |
| | 512 × 512 | | | | | | | |
| 2 | | 7.199 | 0.057 | 0.8% | 7.822 | 0.299 | 3.8% | **0.92** |
| 4 | | 3.812 | 0.235 | 6.1% | 3.814 | 0.194 | 5.0% | **1.00** |
| 8 | | 2.178 | 0.360 | 16.5% | 1.919 | 0.108 | 5.6% | **1.13** |
| 16 | | 1.428 | 0.474 | 33.2% | 0.969 | 0.064 | 6.6% | **1.47** |
| 32 | | 1.127 | 0.597 | 53.0% | 0.498 | 0.046 | 9.2% | **2.26** |
| 64 | | 0.826 | 0.503 | 60.9% | 0.270 | 0.040 | 14.8% | **3.06** |
| | 1024 × 1024 | | | | | | | |
| 4 | | 15.640 | 0.444 | 2.8% | 16.561 | 0.836 | 5.0% | **0.94** |
| 8 | | 8.332 | 0.718 | 8.6% | 8.274 | 0.432 | 5.2% | **1.01** |
| 16 | | 4.827 | 0.939 | 19.4% | 4.156 | 0.238 | 5.7% | **1.16** |
| 32 | | 3.324 | 1.274 | 48.8% | 2.097 | 0.137 | 6.5% | **1.59** |
| 64 | | 2.152 | 1.005 | 46.7% | 1.083 | 0.090 | 8.3% | **1.99** |
| | 2048 × 2048 | | | | | | | |
| 16 | | 18.323 | 1.895 | 10.3% | 18.360 | 0.893 | 4.9% | **0.99** |
| 32 | | 10.764 | 2.356 | 21.9% | 9.215 | 0.487 | 5.3% | **1.17** |
| 64 | | 6.456 | 2.007 | 31.1% | 4.687 | 0.277 | 5.9% | **1.38** |

Table 2: Performance of two versions of 2D-FFT on the iPSC/860

# B Redistribution

It is very difficult for a programmer to decide when dynamic data remapping is profitable of not. The availability of fast collective communication routines is crucial for the profitability of realignment and redistribution. In our experiments we used a transpose routine distributed by Intel as part of a set of example programs for the iPSC/860. This sections gives examples where the transposition of a matrix is profitable (2D-FFT) and where it is not (ADI). The latter case is interesting, since a programmer might think that transposing the matrix is profitable.

Dynamic data remapping can be profitable before or after I/O operations if the underlying I/O systems prefers specific data layout schemes as described in [BGMZ92].

### Two-dimensional Fast Fourier Transform (2D-FFT)

The computation performed by a two-dimensional FFT can be described as a sequence of one-dimensional FFTs (1D-FFTs) along each row of the input array, followed by one-dimensional FFTs along each column. The input array in our example is of type complex. The butterfly version of the 2D-FFT distributes the first dimension of the two-dimensional array. This leads to communication during the computation of the 1D-FFTs along each column. This communication can be avoided if the array is transposed after all 1D-FFTs along each row have been performed. The transpose version uses a row-distribution for the row-wise 1D-FFTs and a column-distribution for the column-

wise 1D-FFTs. Both versions were compiled using if77 under -O4 option and executed on the iPSC/860 at Rice (32 processors) and Caltech (64 processors). Both machines have a two-way set associative from instruction cache (4Kbytes) and data cache (8Kbytes). The cache lines are 32bytes long. Table 2 lists execution times in seconds for the butterfly and transpose implementation alternatives over a variety of data sizes and processor configurations. For each implementation alternative, the table lists the total execution time and the fraction of the time spent communicating. The last column lists the relative speed-ups of the transpose version over the butterfly version for different problem size and processor configurations.

This program example assumes that the compiler is able to detect the FFT (butterfly) communication pattern. If this is not the case, we expect the compiler-generated program for the static row partitioning to run slower than the butterfly version, increasing the benefits of the transpose version even more.

### Alternating-Direction-Implicit Integration (ADI)

The sequential code is shown in Figure 9. The single iterations of the DO-loop in line 2 consists of a forward and backward sweep along the rows of arrays `x` and `b`, followed by a forward and backward sweep along the columns. The *pipeline* version of the code uses a static column-wise partitioning of the perfectly aligned arrays `x`, `a`, and `b`. We specified this data layout using Fortran D language annotations and compiled the program using the current implementation of the Fortran D compiler. The compiler generated a coarse-grain pipelined loop for the forward and backward sweeps along the rows of arrays `x` and `b`. The sweeps along columns do not require communication under this data layout. The *transpose* version transposes arrays `x` and `b` between the row and column sweeps, i.e. twice per iteration of the outermost DO-loop (line 2). No communication is needed during each sweep.

The execution times for 10 iterations (MAXITER = 10) for the iPSC/860 is shown in Table 3. The timings are given in seconds. Since the selected problem sizes are powers of two, cache conflicts lead to a significant increase of the total execution time for the transpose version. To alleviate this problem, we added a single column or row to each local segment of the arrays in the node SPMD program. We expect a sophisticated node compiler to perform such an optimization. The performance of the modified node programs are listed under the problem sizes marked with asterisks in Table 3.

### Summary

The 2D-FFT example shows that dynamic remapping can result in significant performance improvements over a static data layout scheme. A programmer might have expected a similar performance improvement for the ADI example program. However, due to the coarse grain pipelining optimization performed by the Fortran D compiler dynamic data remapping is not profitable even if we ignore cache conflicts.

```
1    REAL x(N, N), a(N, N), b(N, N)
2    DO iter = 1, MAXITER
3        // ADI forward & backward sweeps along rows
4        DO j = 2, N
5            DO i = 1, N
6                x(i, j) = x(i, j) − x(i, j−1) ∗ a(i, j) / b(i, j−1)
7                b(i, j) = b(i, j) − a(i, j) ∗ a(i, j) / b(i, j−1)
8            ENDDO
9        ENDDO
10       DO i = 1, N
11           x(i, N) = x(i, N) / b(i, N)
12       ENDDO
13       DO j = N−1, 1, −1
14           DO i = 1, N
15               x(i, j) = ( x(i, j) − a(i, j+1) ∗ x(i, j+1) ) / b(i, j)
16           ENDDO
17       ENDDO
18       // ADI forward & backward sweeps along columns
19       DO j = 1, N
20           DO i = 2, N
21               x(i, j) = x(i, j) − x(i−1, j) ∗ a(i, j) / b(i−1, j)
22               b(i, j) = b(i, j) − a(i, j) ∗ a(i, j) / b(i−1, j)
23           ENDDO
24       ENDDO
25       DO j = 1, N
26           x(N, j) = x(N, j) / b(N, j)
27       ENDDO
28       DO j = 1, N
29           DO i = N−1, 1, −1
30               x(i, j) = ( x(i, j) − a(i+1, j) ∗ x(i+1, j) ) / b(i, j)
31           ENDDO
32       ENDDO
33   ENDDO
```

Figure 9: Sequential ADI code, `REAL`

| #procs | problem size | pipeline | | | transpose | | | relative |
|---|---|---|---|---|---|---|---|---|
| | | total | communication only | (% of total) | total | communication only | (% of total) | speed-up |
| | 128 × 128 | | | | | | | |
| 2 | | 2.305 | 0.021 | 0.9% | 3.894 | 0.371 | 9.5% | **0.59** |
| 4 | | 1.265 | 0.053 | 4.2% | 1.547 | 0.298 | 19.3% | **0.82** |
| 8 | | 0.720 | 0.077 | 10.7% | 1.371 | 0.246 | 17.9% | **0.52** |
| 16 | | 0.485 | 0.103 | 21.2% | 0.617 | 0.280 | 45.4% | **0.78** |
| 32 | | 0.404 | 0.141 | 34.9% | 1.137 | 0.444 | 39.0% | **0.35** |
| 64 | | 0.431 | 0.235 | 54.5% | 1.130 | 0.821 | 72.6% | **0.38** |
| | 128 × 128* | | | | | | | |
| 2 | | 2.283 | 0.020 | 0.9% | 2.486 | 0.365 | 14.7% | **0.92** |
| 4 | | 1.281 | 0.053 | 4.1% | 1.381 | 0.308 | 22.3% | **0.93** |
| 8 | | 0.715 | 0.080 | 11.1% | 0.835 | 0.255 | 30.5% | **0.85** |
| 16 | | 0.505 | 0.116 | 23.0% | 0.614 | 0.348 | 56.7% | **0.82** |
| 32 | | 0.402 | 0.143 | 35.6% | 0.596 | 0.454 | 76.2% | **0.67** |
| 64 | | 0.430 | 0.236 | 54.9% | 0.921 | 0.845 | 91.7% | **0.47** |
| | 256 × 256 | | | | | | | |
| 2 | | 9.142 | 0.041 | 0.4% | 21.585 | 1.351 | 6.2% | **0.42** |
| 4 | | 4.781 | 0.106 | 2.2% | 10.261 | 1.009 | 9.8% | **0.46** |
| 8 | | 2.598 | 0.162 | 6.2% | 4.250 | 0.677 | 15.9% | **0.61** |
| 16 | | 1.531 | 0.180 | 11.7% | 3.192 | 0.532 | 16.6% | **0.48** |
| 32 | | 1.064 | 0.215 | 20.2% | 2.050 | 0.607 | 29.6% | **0.52** |
| 64 | | 0.838 | 0.307 | 36.6% | 3.046 | 0.921 | 30.2% | **0.27** |
| | 256 × 256* | | | | | | | |
| 2 | | 9.045 | 0.042 | 0.5% | 10.116 | 1.368 | 13.5% | **0.89** |
| 4 | | 4.758 | 0.106 | 2.2% | 5.296 | 1.009 | 19.0% | **0.90** |
| 8 | | 2.566 | 0.167 | 6.5% | 2.844 | 0.678 | 23.8% | **0.90** |
| 16 | | 1.566 | 0.204 | 13.0% | 1.709 | 0.596 | 34.9% | **0.92** |
| 32 | | 0.986 | 0.220 | 22.3% | 1.235 | 0.623 | 50.4% | **0.80** |
| 64 | | 0.828 | 0.320 | 38.6% | 1.255 | 0.958 | 76.3% | **0.66** |
| | 512 × 512 | | | | | | | |
| 2 | | 39.553 | 0.084 | 0.2% | 161.896 | 5.504 | 3.4% | **0.24** |
| 4 | | 20.270 | 0.209 | 1.0% | 81.868 | 3.717 | 4.5% | **0.25** |
| 8 | | 10.612 | 0.308 | 2.9% | 39.072 | 2.289 | 5.8% | **0.27** |
| 16 | | 5.780 | 0.352 | 6.0% | 10.980 | 1.439 | 13.1% | **0.53** |
| 32 | | 3.406 | 0.379 | 11.1% | 6.576 | 1.110 | 16.9% | **0.52** |
| 64 | | 2.289 | 0.468 | 20.4% | 5.704 | 1.257 | 22.0% | **0.40** |
| | 512 × 512* | | | | | | | |
| 2 | | 35.913 | 0.083 | 0.2% | 144.500 | 5.561 | 3.8% | **0.25** |
| 4 | | 18.434 | 0.207 | 1.1% | 21.365 | 3.768 | 17.6% | **0.86** |
| 8 | | 9.573 | 0.313 | 3.3% | 10.795 | 2.270 | 21.0% | **0.89** |
| 16 | | 5.302 | 0.372 | 7.0% | 5.799 | 1.497 | 25.8% | **0.91** |
| 32 | | 3.060 | 0.376 | 12.2% | 3.329 | 1.132 | 34.0% | **0.92** |
| 64 | | 2.083 | 0.467 | 22.4% | 2.402 | 1.296 | 54.0% | **0.87** |
| | 1024 × 1024 | | | | | | | |
| 2 | | 168.055 | 0.175 | 0.1% | 949.241 | 27.171 | 2.9% | **0.18** |
| 4 | | 85.106 | 0.411 | 0.5% | 388.358 | 15.016 | 3.9% | **0.22** |
| 8 | | 43.605 | 0.602 | 1.4% | 237.368 | 8.449 | 3.5% | **0.18** |
| 16 | | 22.860 | 0.678 | 3.0% | 98.511 | 4.854 | 4.9% | **0.23** |
| 32 | | 12.509 | 0.686 | 5.5% | 52.708 | 2.969 | 5.6% | **0.24** |
| 64 | | 7.351 | 0.788 | 10.7% | 13.927 | 2.283 | 16.4% | **0.53** |
| | 1024 × 1024* | | | | | | | |
| 2 | | 147.682 | 0.177 | 0.1% | 752.135 | 27.400 | 3.6% | **0.20** |
| 4 | | 73.467 | 0.410 | 0.5% | 352.282 | 15.162 | 4.3% | **0.21** |
| 8 | | 37.484 | 0.608 | 1.6% | 43.869 | 8.534 | 19.4% | **0.85** |
| 16 | | 19.860 | 0.714 | 3.6% | 22.066 | 4.866 | 22.0% | **0.90** |
| 32 | | 10.820 | 0.683 | 6.3% | 11.629 | 2.980 | 25.6% | **0.93** |
| 64 | | 6.466 | 0.786 | 12.1% | 6.727 | 2.325 | 34.6% | **0.96** |
| | 2048 × 2048 | | | | | | | |
| 4 | | 337.115 | 0.909 | 0.3% | *memory* | *memory* | | |
| 8 | | 170.599 | 1.219 | 0.7% | 815.495 | 34.042 | 4.2% | **0.21** |
| 16 | | 87.407 | 1.365 | 1.6% | 602.598 | 17.979 | 3.0% | **0.14** |
| 32 | | 45.911 | 1.353 | 2.9% | 193.612 | 10.004 | 5.2% | **0.24** |
| 64 | | 25.098 | 1.452 | 5.8% | 131.027 | 6.052 | 4.6% | **0.19** |
| | 2048 × 2048* | | | | | | | |
| 4 | | *memory* | *memory* | | *memory* | *memory* | | |
| 8 | | 146.694 | 1.241 | 0.8% | 671.508 | 34.311 | 5.1% | **0.22** |
| 16 | | 75.563 | 1.414 | 1.9% | 89.174 | 18.161 | 20.4% | **0.85** |
| 32 | | 39.464 | 1.353 | 3.4% | 44.293 | 9.913 | 22.4% | **0.89** |
| 64 | | 21.691 | 1.450 | 6.7% | 23.345 | 6.078 | 26.0% | **0.93** |

Table 3: Performance of pipeline and transpose versions of ADI on the iPSC/860

# C Red-Black Relaxation

Figure 11 and Figure 13 show the actual execution times of one iteration of the outermost time step loop of the red-black relaxation program (shown in Figure 10) for increasing sizes of array `v` using column-partitioning and block-partitioning schemes. Figure 11 shows the execution times on the Ncube-1 for 16 and 64 processors, where `v` is a single precision floating point array. Figure 13 contains the results on 16 processors on the iPSC/860 for a single precision and double precision array `v`.

Figure 12 and Figure 14 show the estimated execution times for the red-black program using the training set approach. The crossover points and execution times induced by the different decomposition schemes are predicted with high accuracy.

```
1    DOUBLE PRECISION v(N, N), a, b
2    DECOMPOSITION d(N, N)
3    ALIGN v(I, J) WITH d(I, J)
4    DISTRIBUTE d(BLOCK, BLOCK)
5    DO k = 1, M
6        // Compute the red points
7        DO j = 1, N, 2
8            DO i = 1, N, 2
9                v(i, j) = a*(v(i, j−1) + v(i−1, j) + v(i, j+1) + v(i+1, j)) + b * v(i, j)
11           ENDDO
12       ENDDO
13       DO j = 2, N, 2
14           DO i = 2, N, 2
15               v(i, j) = a*(v(i, j−1) + v(i−1, j) + v(i, j+1) + v(i+1, j)) + b* v(i, j)
16           ENDDO
17       ENDDO
18       // Compute the black points
19       DO j = 1, N, 2
20           DO i = 2, N, 2
21               v(i, j) = a*(v(i, j−1) + v(i−1, j) + v(i, j+1) + v(i+1, j)) + b* v(i, j)
22           ENDDO
23       ENDDO
24       DO j = 2, N, 2
24           DO i = 1, N, 2
26               v(i, j) = a*(v(i, j−1) + v(i−1, j) + v(i, j+1) + v(i+1, j)) + b* v(i, j)
27           ENDDO
28       ENDDO
29   ENDDO
```

Figure 10: Fortran D code with `BLOCK` distribution, `DOUBLE PRECISION`

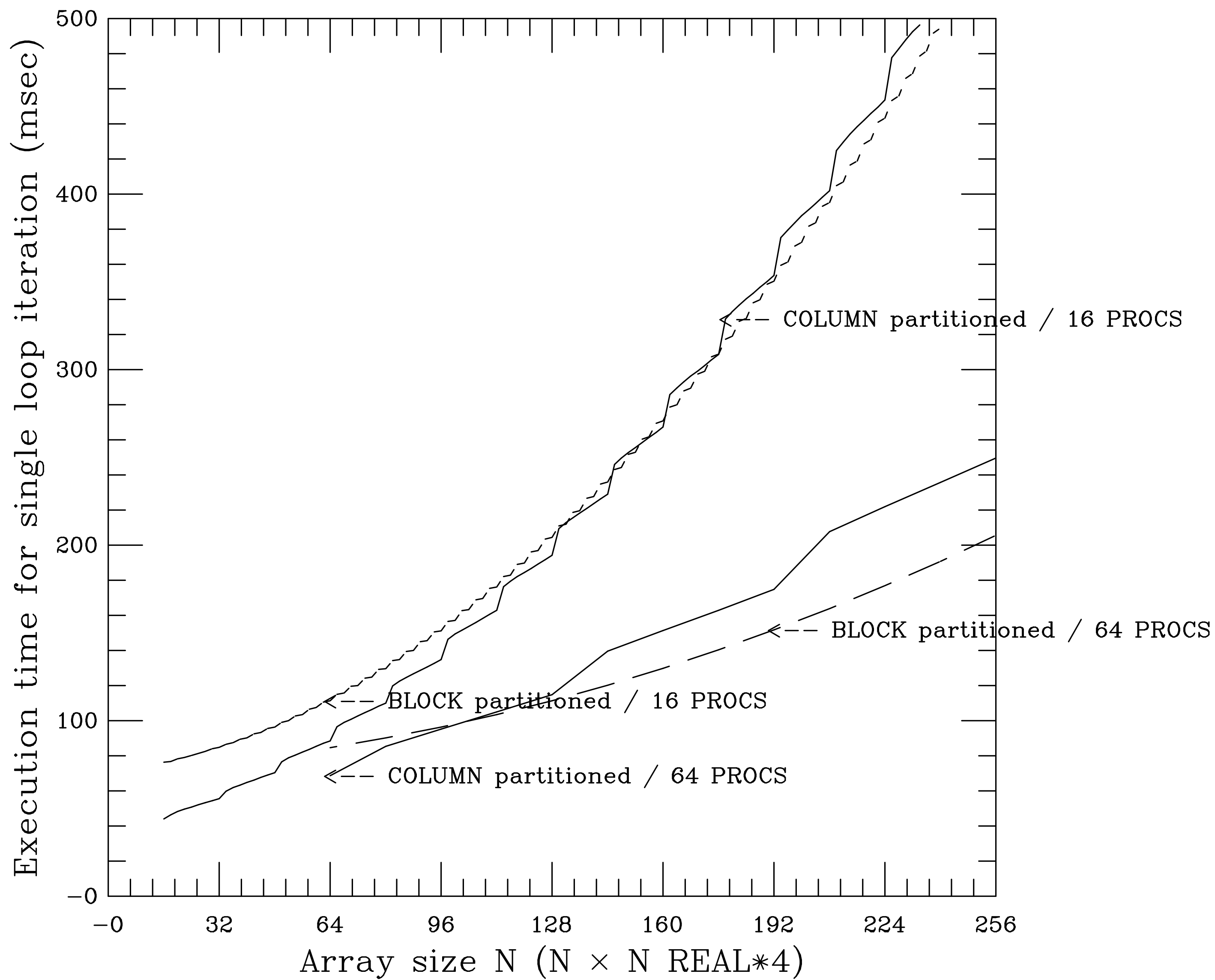


Figure 11: Measured times on Ncube-1: FLOAT operations on 16 and 64 processors

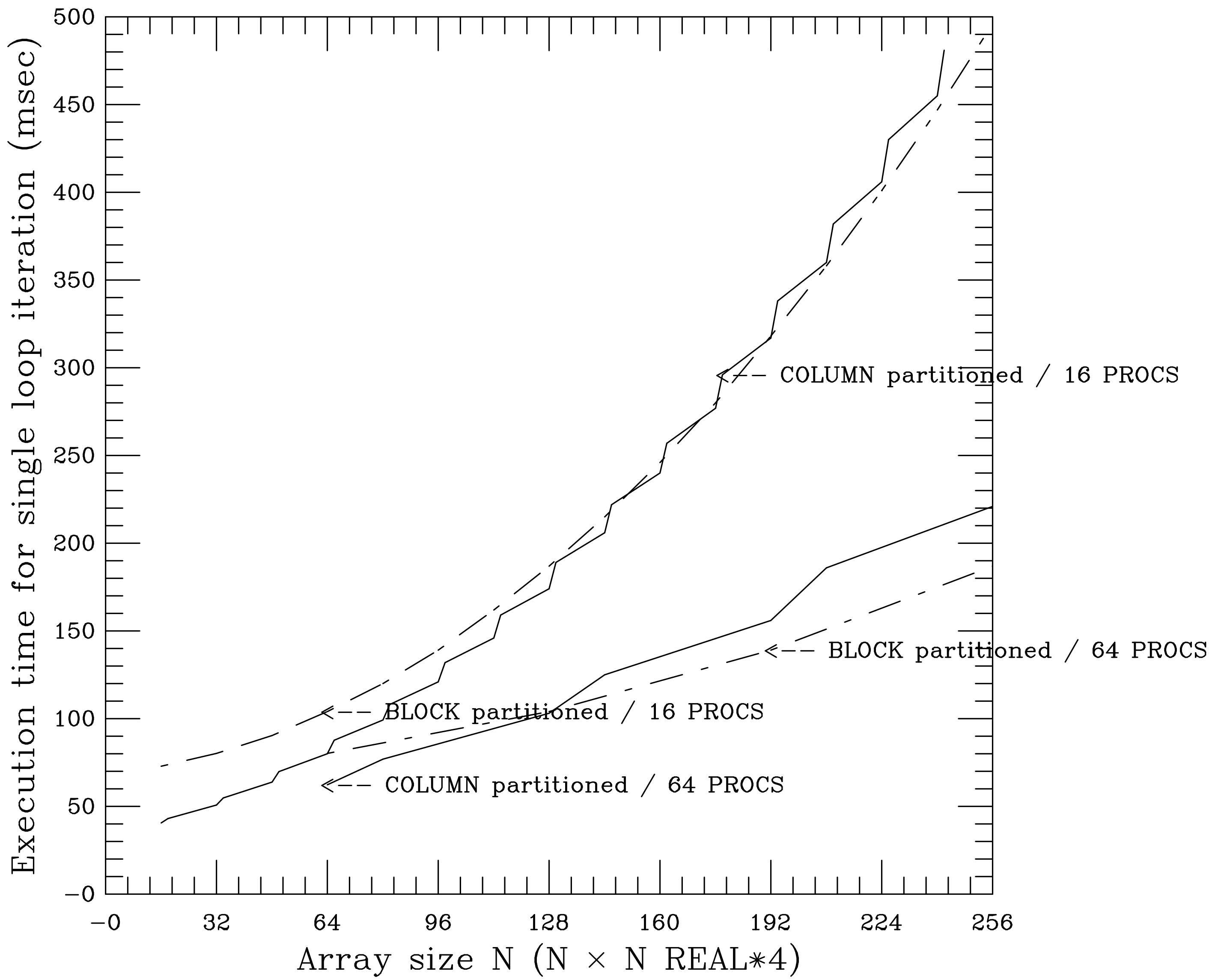


Figure 12: Estimated times on Ncube-1: FLOAT operations on 16 and 64 processors

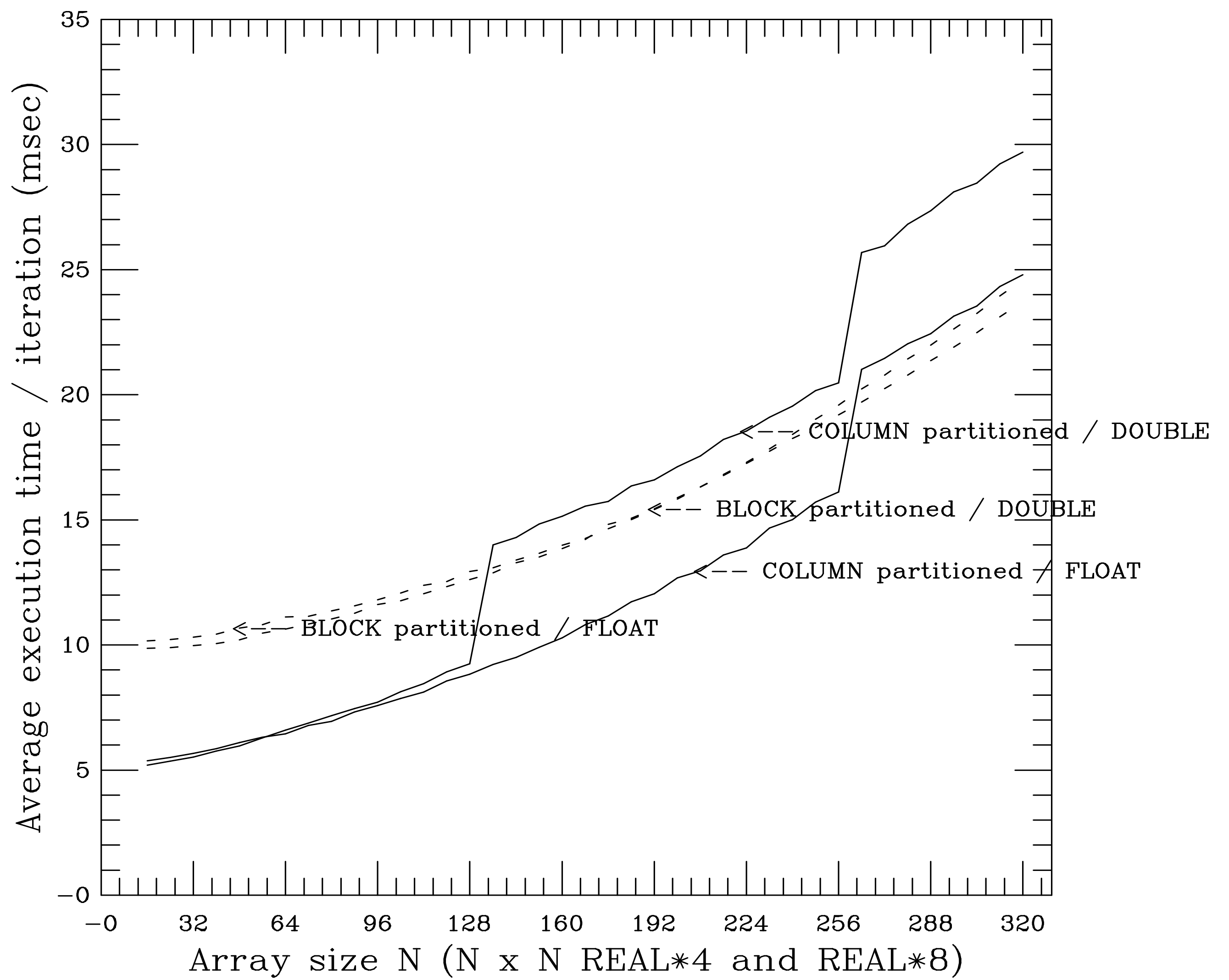


Figure 13: Measured times on iPSC/860: DOUBLE PRECISION and FLOAT operations on 16 processors

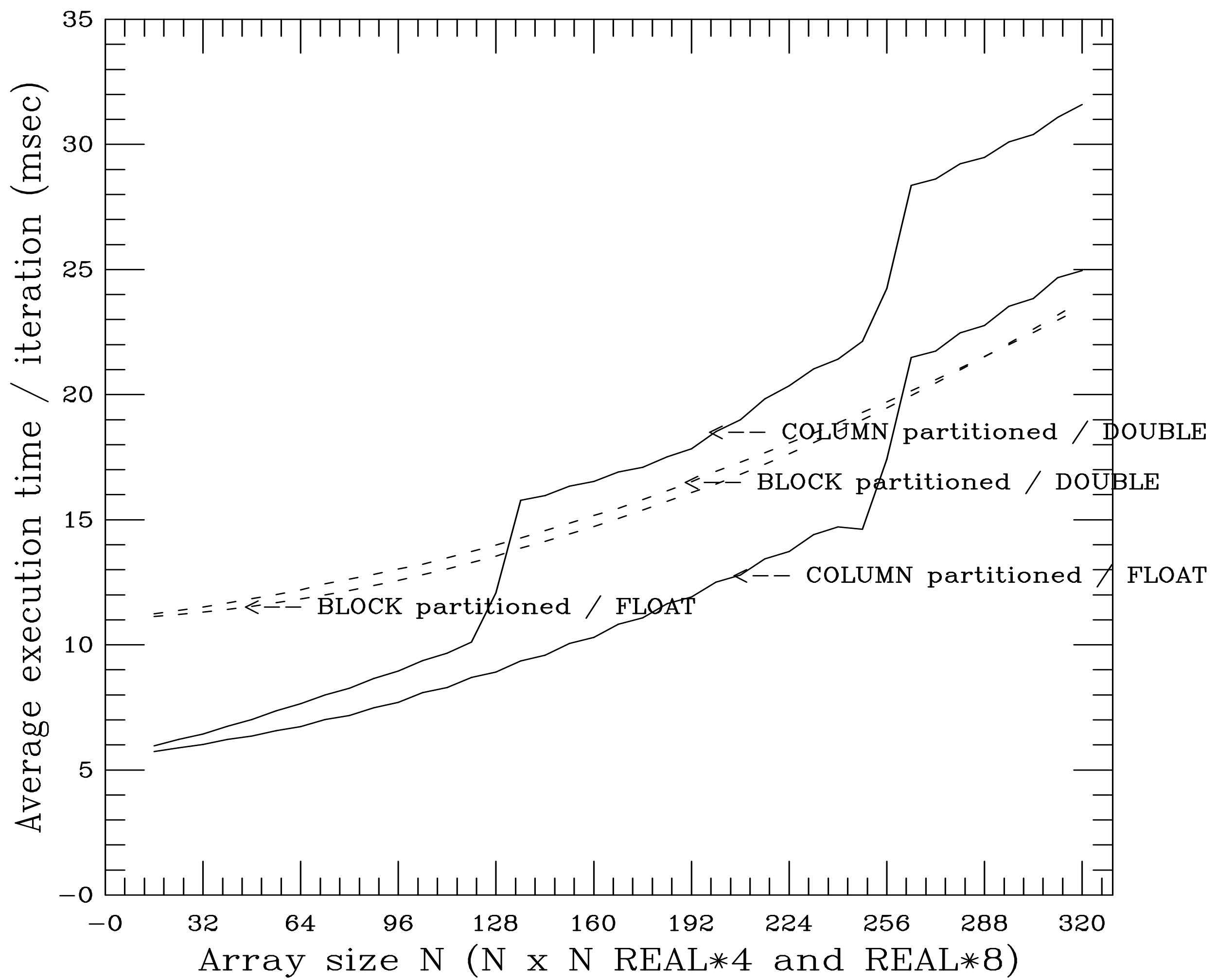


Figure 14: Estimated times on iPSC/860: DOUBLE PRECISION and FLOAT operations on 16 processors