Automatic Data Layout for High Performance Fortran

Ken Kennedy *Ulrich Kremer*

CRPC-TR94498-S
December, 1994

Center for Research on Parallel Computation
Rice University
6100 South Main Street
Houston, TX 77005-1892

Revised April 1995, August 1995

Automatic Data Layout for High Performance Fortran*

Ken Kennedy Ulrich Kremer†

Department of Computer Science
Rice University

Abstract

High Performance Fortran (HPF) is rapidly gaining acceptance as a language for parallel programming. The goal of HPF is to provide a simple yet efficient machine independent parallel programming model. Besides the algorithm selection, the data layout choice is the key intellectual step in writing an efficient HPF program. The developers of HPF did not believe that data layouts can be determined automatically in all cases. Therefore HPF requires the user to specify the data layout. It is the task of the HPF compiler to generate efficient code for the user supplied data layout.

The choice of a good data layout depends on the HPF compiler used, the target architecture, the problem size, and the number of available processors. Allowing remapping of arrays at specific points in the program makes the selection of an efficient data layout even harder.

Although finding an efficient data layout fully automatically may not be possible in all cases, HPF users will need support during the data layout selection process. In particular, this support is necessary if the user is not familiar with the characteristics of the target HPF compiler and target architecture, or even with HPF itself. Therefore, tools for automatic data layout and performance estimation will be crucial if the HPF is to find general acceptance in the scientific community.

This paper discusses a framework for automatic data layout for use in a data layout assistant tool for a data-parallel language such as HPF. The envisioned tool can be used to generate a first data layout for a sequential Fortran program without data layout statements, or to extend a partially specified data layout in a HPF program to a totally specified data

*This research was supported by the Center for Research on Parallel Computation (CRPC), a Science and Technology Center funded by NSF through Cooperative Agreement Number CCR-9120008. This work was also sponsored by ARPA under contract #DABT63-92-C-0038. The content of this paper does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be inferred.

†**Corresponding author**; e-mail: kremer@cs.rice.edu; phone: (713) 527-6077; address: Department of Computer Science, Rice University, 6100 S. Main, Houston, Texas 77005–1892

layout. Since the data layout assistant is not embedded in a compiler and will run only a few times during the tuning process of an application program, the framework can use techniques that may be too computationally expensive to be included in a compiler.

A prototype data layout assistant tool based on our framework has been implemented as part of the D system currently under development at Rice University. The paper reports preliminary experimental results. The results indicate that the framework is efficient and generates data layouts of high quality.

1 Introduction

Compilers for data parallel languages such as High Performance Fortran (HPF) or Fortran D [FHK+90] typically perform many optimizations based on the specified data layout. The actual optimizations performed may vary between compilers. Different parallel machine architectures have different communication and computation costs, resulting in distinct optimal balances between communication and computation for each architecture. On some machines and for some compilers, remapping data between different parts of an application program may be profitable. All these factors make it extremely difficult for a user to predict the performance impact of a particular data layout for an application.

This paper discusses a framework for automatic data layout for regular problems as part of a data layout assistant tool in a data parallel programming environment such as the D system [ACG+94]. A data layout assistant tool can be used to determine an initial data layout for a sequential Fortran application. If the quality of the automatically generated layout is not satisfactory, static and dynamic performance analysis tools can help the user to understand the performance characteristics of the program. Once the user has chosen data layouts for program parts crucial for the overall performance, the layout assistant tool can be used to extend these data layouts to a data layout for the entire program. The application scenario for our proposed data layout assistant tool is shown in Figure 1. Typically, regular problems represent data objects as dense arrays as opposed to a sparse representation. Regular problems allow the compilation system to determine the communication requirements and to perform a variety of program optimizations at compile time. The automatically selected data layout may contain remappings if they are found profitable.

Our framework for automatic data layout has been designed to support experimentation with different data layout strategies. It uses explicit search spaces of candidate data layouts, allowing different techniques for search space construction, performance prediction, and final selection of candidates from each search space. In addition, explicit candidate layout search spaces provide a natural user interface. The user will be able to browse through the search spaces of candidate layouts with their predicted performances and insert new candidate layouts into or delete candidate layouts from the search spaces.

Some problems in finding efficient data layouts are known to be NP-complete. Rather than resorting to heuristics prematurely, the framework capitalizes on 0-1 integer programming technology to compute optimal solutions for two NP-complete problems. The framework is parameterized with respect to the HPF compiler, the machine architecture, the problem size, and the number of available processors. As a consequence, these entities have to be known at tool invocation time. Note that an automatically generated data layout can

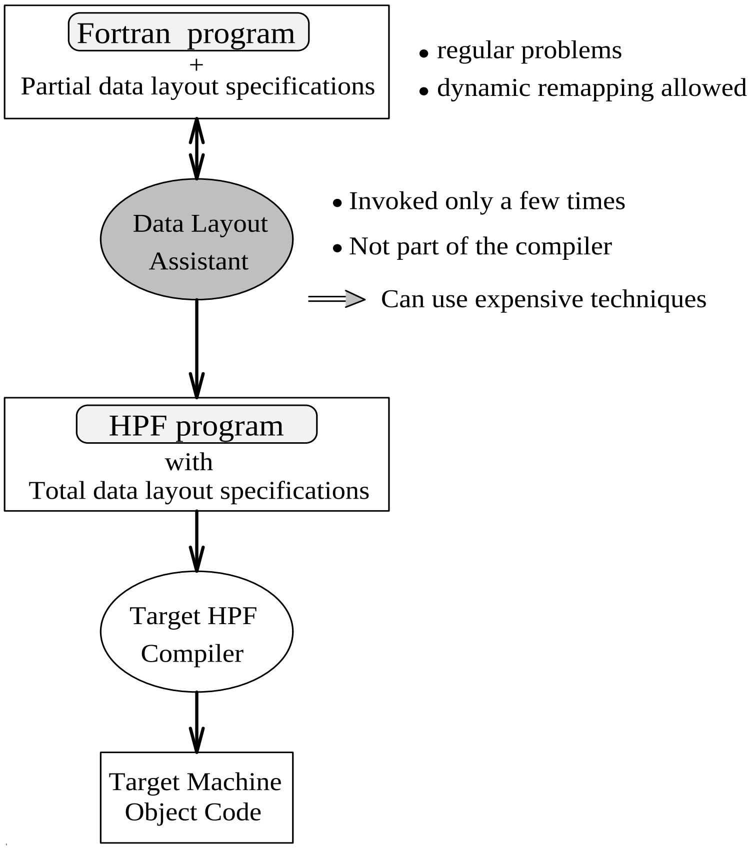

Figure 1: Automatic data layout as part of a programming environment

be used for different problem sizes and numbers of processors, although its quality may be suboptimal.

The framework consists of several steps. An overview of these steps is given in Section 2. A prototype implementation based on the framework is described in Section 3. The results of applying the prototype to four application programs are discussed in Section 4. The paper concludes with a discussion of related work, followed by a summary and discussion of future work in Section 5 and Section 6, respectively.

2 Framework for Automatic Data Layout

The framework for automatic data layout consists of four steps. In the first step the input program is partitioned into program segments. For each such program segment, the second step constructs a search space of promising candidate layouts. A candidate layout for a program segment is a mapping of every array referenced in the segment onto the target architecture. Heuristics are used to generate the candidate layout search spaces. In the third step each candidate layout is evaluated in terms of its estimated execution time. In addition, costs of possible remappings between candidate layouts are determined. The performance estimation is based on a compiler model, execution model, and machine model. Based on the estimated candidate layout costs and costs of possible remappings between candidate layouts, a single candidate layout from each search space has to be selected such that the overall cost is minimal. This selection process is performed in the fourth and last step of our framework. In the remainder of this section each of the four steps is discussed in more detail.

2.1 Program Partitioning

The first step partitions the program into code segments, called program *phases*. In our current framework, data remapping is allowed only between phases. A *phase* is the outermost loop in a loop nest such that the loop defines an induction variable that occurs in a subscript expression of an array reference in the loop body. This operational definition does not allow the overlapping or nesting of phases. Other strategies for identifying program phases are a topic of current research. For instance, two adjacent phases can be merged into a single phase if remapping can never be profitable between them. Sheffler et al. describe techniques to perform such phase merges [SSP+95]. Transformations to improve phase recognition are beyond the scope of this paper.

The phase structure of the program is represented in the *phase control flow graph* (PCFG), an augmented control flow graph [ASU86] where each phase is represented by a single node. The graph is annotated with branch probabilities and loop control information. Branch probabilities can either be supplied by the user or are determined based on a guessing heuristic.

2.2 Layout Search Spaces Construction

The second step of the framework for automatic data layout constructs explicit search spaces of candidate data layouts for each phase. Promising candidate layouts for a phase are generated based on their expected performance as part of an efficient data layout for the entire program.

A data layout in HPF is defined in two stages, referred to as *alignment* and *distribution*. Arrays are aligned relative to each other by specifying a mapping of their elements to the same array of virtual processors, called a *template*. Every array element aligned with a template is mapped to a real processor by distributing the template onto the processors of the target architecture.

The use of explicit search spaces is an important design decision in our framework for automatic data layout. Explicit alignment search spaces allow the framework to postpone the evaluation of an alignment candidate until all distribution candidates are known. Therefore, promising alignment candidates are not eliminated prematurely, but are evaluated in combination with the selected candidate distributions.

The current framework determines a single template for the entire program based on the maximal dimensionalities and maximal dimensional extents of the arrays in the program. All alignments and distributions are specified relative to this *program template*. Corresponding to the two stage mapping, the framework first builds search spaces of promising candidate alignments for each phase. If arrays have fewer dimensions than the program template, alignment analysis may generate different embeddings for the arrays. Then, distribution analysis uses the alignment search spaces to build candidate data layout search spaces of reasonable alignments and distributions for each phase. Alignment and distribution analysis is discussed in the next two sections.

2.2.1 Alignment Analysis

Alignment analysis takes the phase control flow graph as input and generates explicit alignment search spaces for each phase. Heuristics have to be used to determine a reasonably sized set of alignment candidates that will guarantee a good overall performance for most applications.

Alignment analysis is done in two stages. First, only alignment preferences between arrays are considered. In the second stage, each array is mapped onto the unique program template such that the relative alignment preferences are respected. The second stage of the alignment mapping is called *orientation* [AL93].

This section discusses basic operations that are needed to identify and represent relative alignment preferences, to detect and resolve conflicting relative alignment preferences, and to compare relative candidate alignments. The comparison of alignment candidates is important in order to avoid redundant alignment information in the alignment search spaces. In addition, some methods for orientation selection are discussed.

The basic operations and methods form the building blocks for implementing different heuristics and strategies for the alignment search space construction. The heuristic implemented in our prototype tool is discussed in Section 3.2.

There are two types of alignment preferences, namely *inter–dimensional* and *intra–dimensional* alignment [LC90, KLS90, CGST93]. The current framework does not perform intra-dimensional alignment analysis, i.e., assumes canonical offset and stride alignments.

Identification and Representation of Relative Alignment Preferences

A central representation for the relative, inter-dimensional alignment problem is the weighted, undirected *component affinity graph* (CAG) introduced by Li and Chen at Yale University [LC90]. It represents the alignment preferences of arrays that are coupled in a computation. A d-dimensional array is represented in the CAG by d nodes, one node for each dimension. Alignment preferences between dimensions of distinct arrays are represented as edges between the corresponding nodes. The weights of the edges reflect the relative importance of alignment preferences. Typically, an edge weight represents the expected performance penalty if the corresponding alignment preferences is not satisfied. Therefore, methods to determine edge weights are based on some performance model. Our framework is independent of the actual performance model used. The alignment analysis performance model in our prototype implementation is discussed in Section 3.1.

Detection of Relative Alignment Conflicts

Assume that d is the dimensionality of the program template. A solution to the inter-dimensional alignment problem is a partitioning of the nodes in the CAG into d partitions such that no two nodes representing dimensions of the same array are in the same partition. A CAG contains a *conflict* if there is a path between two nodes that represent distinct dimensions of the same array. The definition of a conflict does not allow diagonal alignments such as aligning a one-dimensional array with the main diagonal of a two-dimensional array. The test for alignment conflicts is linear in the size of the CAG, since it involves solutions of reachability problems between nodes in the CAG.

Relative Alignment Conflict Resolution

A conflict implies that every solution of the corresponding inter-dimensional alignment problem will have alignment preferences that cannot be satisfied. A good solution tries to minimize the weights of the edges that cross partitions and therefore cannot be satisfied. Li and Chen showed that finding the optimal solution for the inter-dimensional alignment problem is NP-complete [LC90]. Instead of using a heuristic, the current framework formulates the inter-dimensional alignment problem as an efficient 0–1 integer programming problem. A detailed description of our 0–1 formulation is given in the appendix.

Comparison of Relative Alignment Preferences

The inter-dimensional alignment information of a conflict-free CAG can be represented as a partitioning of its nodes. Each partition in the partitioning of a conflict-free CAG is a connected component in the CAG. The set of all possible conflict-free, inter-dimensional alignments of a set of arrays forms a semi-lattice [Hec77]. The bottom element of the lattice is the CAG that contains no alignment information, i.e., the graph contains no edges and therefore its partitioning consists of partitions that contain only single nodes. Figure 2 shows an example lattice for two two-dimensional arrays.

The partial order $\sqsubseteq$ defined over the set of conflict-free CAGs is that of partitioning refinement. Assume that CAG_1 and CAG_2 are two conflict-free CAGs, then $CAG_1 \sqsubseteq CAG_2$ if and only if the node partitioning of CAG_1 is a refinement of the node partitioning of CAG_2. A partitioning X is a refinement of a partitioning Y, if for each partition $x \in X$ there is a partition $y \in Y$, such that $x \subseteq y$. Assuming that partitions are implemented using hashing, and the elements in X and Y are tagged with their partition membership, then the test "X is refinement of Y" is linear in practice in the number of elements in all partitions of X. In other words, the complexity of computing $CAG_1 \sqsubseteq CAG_2$ is linear in practice in the number of nodes in CAG_1.

The test for $CAG_1 \sqsubseteq CAG_2$ allows the direct comparison of the alignment information in two CAGs. Other operations on the semi-lattice are the *meet* operation $CAG_1 \sqcap CAG_2$ and the *join* operation $CAG_1 \sqcup CAG_2$. Both operations can be implemented efficiently.

Orientation Selection

A conflict-free CAG represents relative inter-dimensional alignment preferences of arrays. A final stage is needed to determine the orientation of the CAG relative to the unique program template. An orientation of a conflict-free CAG maps the CAG's connected components, i.e., the sets of aligned array dimensions to the dimensions of the program template. For a d-dimensional program template and a conflict-free CAG with d connected components,

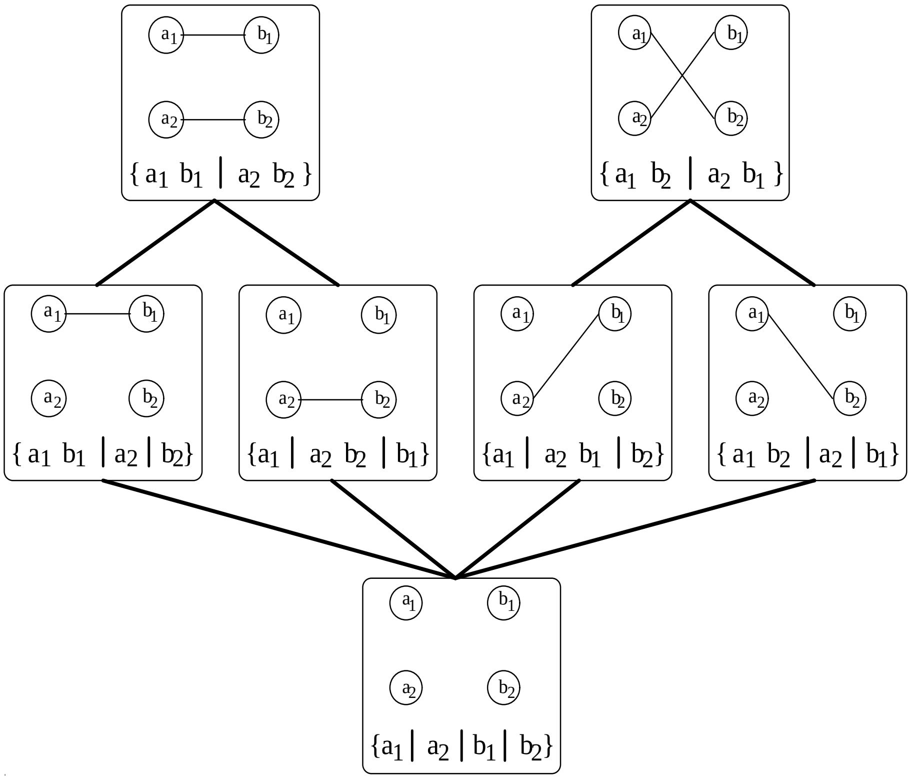

Figure 2: Inter-dimensional alignment information lattice for two arrays a and b. Both arrays have two dimensions. Each conflict-free CAG is shown with its node partitioning. The bottom element of the lattice is the CAG without edges, i.e., the partitioning $\{a_1 \,|a_2 \,|b_1 \,|b_2\}$.

there are $d!$ possible orientations.

Any orientation of a conflict-free CAG satisfies the alignment preferences represented in the CAG. However, in the presence of dynamic realignment, the orientation of two distinct CAGs may influence the potential remapping costs between the two alignments. Therefore an algorithm is needed that matches the orientations of the CAGs in the alignment search spaces as closely as possible. Anderson and Lam propose a greedy strategy to determine orientations [AL93]. We discuss a similar strategy based on meet operations over the lattice of conflict-free CAGs in [Kre95].

2.2.2 Distribution Analysis

Distribution analysis is performed after alignment analysis. A candidate distribution can map single template dimensions either by `block`, `cyclic`, or `block-cyclic` onto the target architecture, or replicate dimensions on each processor. In addition, a candidate distribution specifies the number of processors in each distributed dimension. Different heuristics can be used to determine a suitable set of promising distribution candidates. Once the distribution candidates have been determined, the cross product of alignment candidates and distribution candidates defines the candidate data layout search spaces for each phase.

The heuristics for the construction of candidate distribution search spaces can be roughly divided into two classes, *exhaustive* and *constructive*. The sizes of some exhaustive search spaces are discussed in [Kre95]. Exhaustive heuristics approximate the exhaustive set of all possible distributions of the program template. An approximation is a subset of candidate distributions that can be considered a sparse representation of the exhaustive set. Constructive heuristics choose distribution candidates based on the alignments in the alignment search spaces.

The quality of the performance estimator in the framework determines a bound on the granularity of the distribution search space. If the estimator is not able to distinguish two similar distributions, then only a single representative distribution should be added to the search space.

2.3 Performance Estimation

After the generation of the search spaces each candidate data layout is evaluated in terms of its expected execution time for its phase. In addition, execution time estimates are needed for possible remappings between candidate data layouts.

The performance estimator uses a compiler model to determine where and what kind of communication will be generated for a given candidate data layout and its phase. The compiler model is parameterized with respect to the transformations and communication optimizations that may be performed by the target compiler.

The compilation process needs to be simulated for performance purposes only. Special cases that have a small impact on the the overall performance, but must be handled by a real compiler in order to generate correct code, may be ignored in the performance estimation compiler model. For instance, the compiler model implemented as part of our prototype ignores code that is generated for boundary processors in loops. The implemented compiler model is discussed in Section 3.

Once locations and types of compiler generated communications are known for a candidate layout and its phase, an execution model is used to estimate the performance effects of synchronizations induced by the communications. Communication inside a phase may lead to a reduction or pipelined execution of the loop. In contrast, communication outside of the phase may result in a loosely synchronous execution scheme [FJL+88].

2.4 Layout Selection

As the result of the performance estimation step, performance numbers in terms of relative execution times are available for all candidate data layouts and possible remappings between layouts. In the last step of our framework for automatic data layout, a single candidate layout has to be selected from each search space of each phase such that the resulting set of candidate layouts has minimal overall cost. The overall cost is determined by the costs of each selected candidate layout and the required remapping costs between selected layouts. Note that the optimal data layout for a program may consist of candidate data layouts that are each suboptimal for their phases.

In our framework, the final layout selection problem is formulated as a graph problem. The *data layout graph* has one node for each candidate layout. Edges represent possible remappings between layouts. Nodes and edges are weighted by their relative execution times. The optimal selection problem has been shown to be NP-complete [Kre93]. We discuss a translation of the graph problem into a 0–1 integer programming formulation elsewhere [BKK94b]. We found our 0–1 formulation to be efficient in practice.

3 Prototype Implementation

A prototype data layout assistant tool has been implemented as part of the D system [ACG+94]. The prototype tool performs only intra-procedural analysis. Non-linear control flow in input Fortran programs is restricted to `Do` loops and `If` statements.

The alignment analysis performance model and the heuristic for alignment search space construction are discussed in Section 3.1 and Section 3.2, respectively. Distribution analysis generates exhaustive search spaces for only one-dimensional, `block` distributions. This restriction is due to the fact that the compiler model implementation mimics the program analysis steps in the Fortran D prototype compiler which does not support multi-dimensional distributions [Tse93]. Fortran D [FHK+90] shares many features with HPF since it was one of the main contributors to the HPF language proposal. The execution model uses data dependence information to detect processor synchronization. Phases are classified as either pipelined, loosely synchronous, or reductions. Performance estimates for basic computations and communication patterns are based on machine level training sets for Intel's iPSC/860 or Paragon [BFKK91]. The training set node programs were compiled using the highest level of optimization (if77 –O4).

The prototype uses over 100 training sets that measure basic computations such as `real` and `double` floating point operations, and basic communication patterns such as nearest neighbor communication, single send/receive pairs, broadcasts, reductions, and transpose operations. For each communication pattern there are training sets for different numbers of processors, different memory access patterns, and different observable message latencies. The current implementation distinguishes between a unit or non-unit stride memory access pattern, and high or low latency messages. A non-unit memory access pattern requires message buffering. Low latency message costs are used to estimate the communication costs in pipelined phases where computation and communication can be overlapped. In contrast, message costs for loosely synchronous phases are based on high latency training sets.

The prototype tool solves NP-complete problems during alignment analysis and the final data layout selection step. Instances of these problems are translated into 0–1 integer programming problems suitable to be solved by *CPLEX*[1], a linear integer programming tool and library, partly developed by Robert Bixby at Rice University [Bix92]. The prototype tool builds the required constraint matrices internally, and directly calls the CPLEX routines without creating any intermediate files.

[1] *CPLEX* is a trademark of CPLEX Optimization, Inc.

3.1 CAG Edge Weights

Our algorithm to determine CAG edge weights assumes an advanced compilation system that caches communicated values and uses the owner-computes rule for computation mapping. The target machine is assumed to be a MIMD architecture. Our alignment analysis performance model is pessimistic since it assumes that unsatisfied alignment preferences will always lead to communication.

During the process of determining edge weights, our CAG representation is a directed graph. The edge directions keep track of the flow of values due to the owner-computes rule. If the same alignment preference is encountered, the algorithm checks whether the preference has the same direction as the one already represented in the CAG. If the directions are not identical, the edge weight is increased by the estimated communication cost and the edge direction is reversed. If the directions are the same, the CAG remains unchanged. The estimated communication cost models the volume of the communication, i.e., corresponds to the size of the array that has to be communicated. Due to the owner-computes rule, a communicated array is at the source of an edge in the CAG. Once the edge weights of a CAG have been determined, all edge directions are removed.

3.2 Heuristic for Construction of Alignment Search Spaces

The alignment search spaces are initialized with the undirected, weighted CAGs of their phases. If a CAG has inter-dimensional alignment conflicts, the conflict is resolved and the resulting CAG is used for the initialization. After initialization, the phases are partitioned into classes such that their merged CAGs are conflict-free. Each class of phases is represented by its joined, conflict-free CAG. The current prototype uses a greedy strategy to determine the phase CAGs to be merged next. The implemented algorithm visits the phases, i.e., the nodes in the PCFG in reverse postorder [Hec77], and joins their CAGs as long as no conflict is detected. Once a conflict is encountered, a new class is created and initialized with the CAG of the single phase that led to the conflict. The partitioning algorithm terminates after all phases have been visited.

The main step of our heuristic to construct alignment search spaces for each class consists of exchanging alignment information between different phase classes by inserting corresponding alignment candidates into their alignment search spaces. An *imported* alignment candidate is the result of the optimal embedding of the source CAG into the sink CAG of the import operation. The import process merges the CAGs of the source and sink class of the import operation after increasing the edges weights of the source CAG by some constant factor. The edge increase guarantees that the alignment preferences of the source CAG will dominate the alignment preferences of the sink CAG. This is important if the merged CAG has an alignment conflict. The imported alignment candidate is the alignment scheme resulting from solving possible conflicts in the merged CAG, and restricting the resulting alignment information to those arrays that are referenced in the sink class of the import operation.

The current prototype imports each optimal alignment candidate of another phase class into the search space of a phase class. If the phase partitioning has p classes, then each final class alignment search space can have at most p candidates. In order to avoid duplication

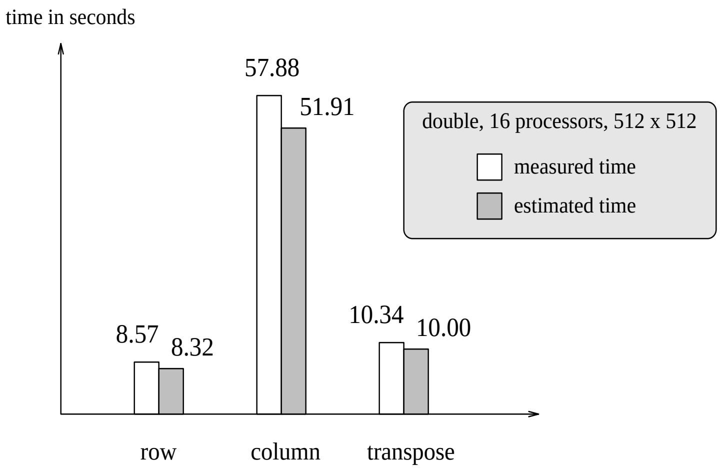

Figure 3: Example test case for ADI with three possible data layouts

of alignment information, the imported alignment candidate is only inserted into the class search space if its alignment information is not weaker or equal to any alignment information already in the search space. A more detailed discussion of the alignment heuristic can be found in [Kre95]. Finally, candidate alignment schemes for phase classes are translated into candidate alignments for each individual phase in the class.

Since the current prototype generates exhaustive search spaces of only one-dimensional `block` distributions, the orientation selection is trivial due to the symmetry between orientations and distribution candidates. For instance, in the two-dimensional case, the candidate layout resulting from a transposed orientation and distribution by row is the same as from a canonical orientation and distribution by column.

4 Experimental Results

The experiments were based on a target compiler that performs message coalescing and message vectorization, but does not perform coarse grain pipelining, loop interchange, or loop distribution. The parameters in the compiler model were set to simulate such a target compiler. The target architecture for our experiments was Intel's iPSC/860.

It is important to note that it is not the goal of our experiments to evaluate the quality of any target compiler, but to show the ability of the data layout assistant tool to simulate the target compiler and to correctly estimate the relative performance of the candidate layouts in its generated search spaces. The quality of a data layout for a program is always relative to the HPF compiler that is used to compile the program.

We used four programs for our experiments, an alternating direction implicit integration kernel (Adi), a 3D tridiagonal solver based on ADI integration and developed by Thomas Eidson at ICASE (Erlebacher), a grid generation program, adapted from the SPEC benchmark suite by Applied Parallel Research (Tomcatv), and a weather prediction program based on shallow-water equations (Shallow). Shallow was written by Paul Swarztrauber from the National Center for Atmospheric Research (NCAR).

The automatic data layout tool was applied to each program for different test cases. A test case consists of a data type for the arrays in the program, a problem size, and a given number of processors used. Figure 3 shows a single test case for the Adi kernel and its results. The test case is for double precision arrays, 16 processors, and a problem size of 512×512. For each test case, the overall execution times of promising data layouts for the entire program were measured and compared to execution times predicted by the prototype data layout assistant tool. For the Adi kernel test case shown in Figure 3 the prototype tool picked the best data layout, namely a static row-wise data layout, and also ranked the data layout alternatives correctly.

For all four programs, the prototype tool did not miss any promising data layouts. The remaining questions are whether the tool ranked the data layout alternatives correctly, and whether the best predicted data layout alternative was also the best measured alternative.

To perform the comparison, each program was compiled for each data layout in its set of promising data layouts using the Fortran D compiler prototype [Tse93] with loop interchange and coarse-grain pipelining disabled. When necessary, the output of the Fortran D compiler was modified by hand to ensure correct code. The resulting SPMD node programs were compiled using the highest optimization level (if77 –O4), and executed and timed on the iPSC/860. In the remainder of this section, we will discuss the results for each program in more detail.

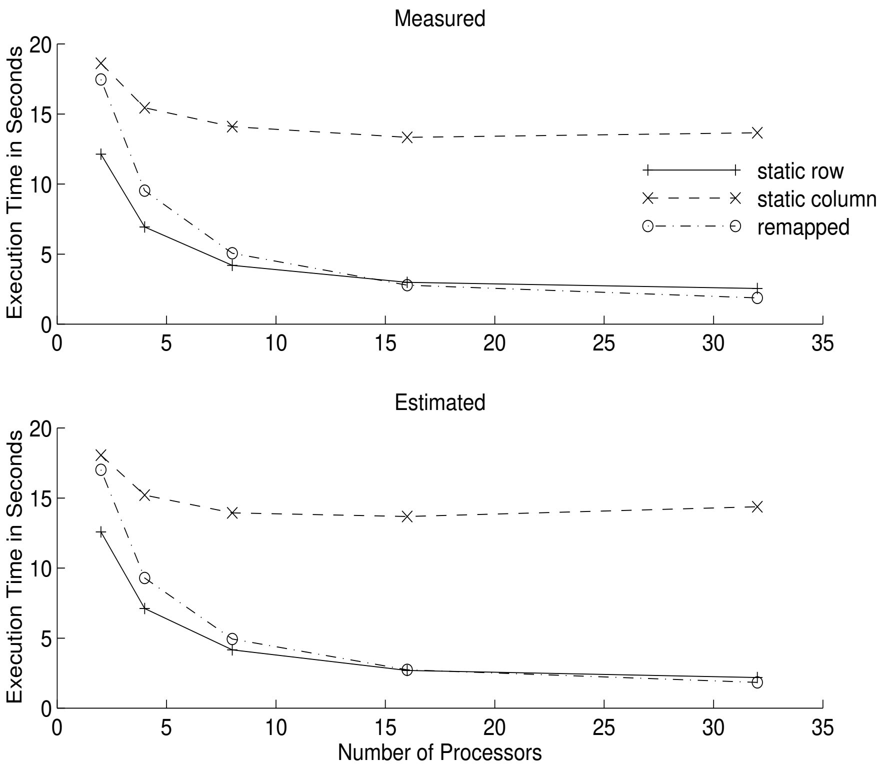

Figure 4: Measured and estimated execution times for Adi kernel with problem size 256×256, double precision

Adi: Adi solves a two-dimensional problem. The program has 9 phases. There are no inter-dimensional alignment conflicts. The solution of the 0–1 data layout selection problem took CPLEX 60 milliseconds on average on a SPARC-10. The problem had 61 variables and 53 constraints.

We measured 40 test cases, one of which is shown in Figure 3. The results of all five test cases for problem size 256×256 and double precision arrays can be found in Figure 4. Distributing the second dimension (column layout) resulted in the sequential execution of two phases. This was always the worst choice. Distributing the first dimension (row layout) introduced a fine-grain pipeline in two phases and resulted in the best possible data layout in 24 cases. In the remaining 16 cases, a dynamic layout that remaps the arrays between row and column sweeps (transpose or remapped layout) was the best data layout choice.

The prototype tool selected the best data layout in 36 cases. In all these cases the relative rankings of data layout alternatives were correct. In 4 cases the automatically chosen layout was suboptimal. The worst case performance loss due to the suboptimal selection was 9.3% as compared to the best possible choice.

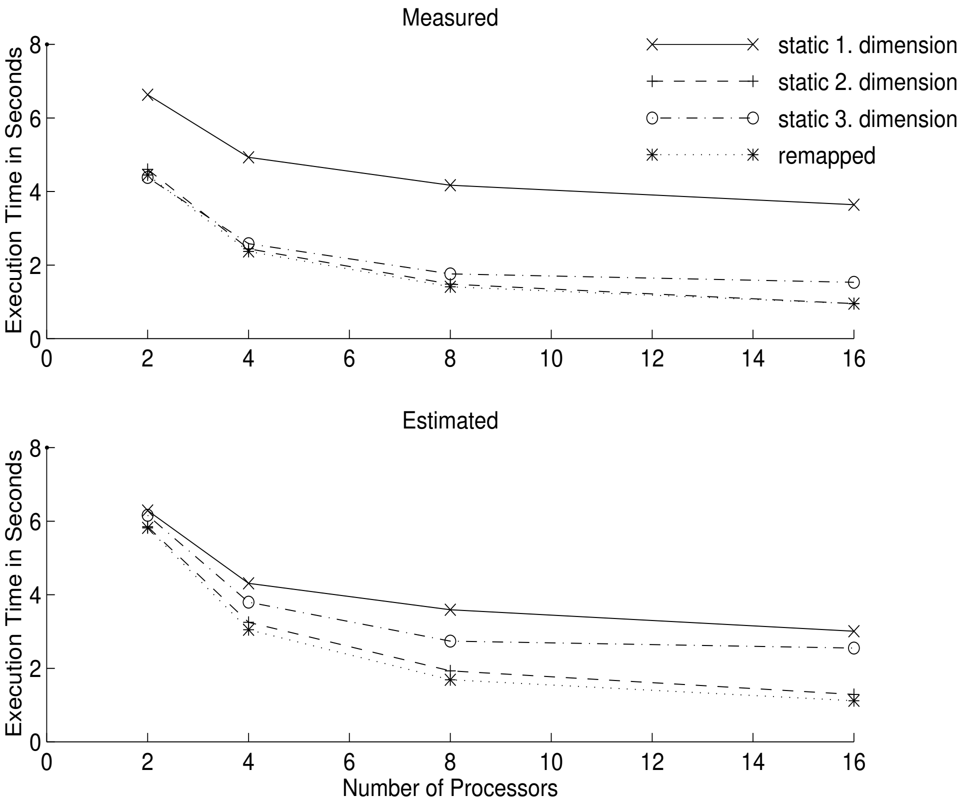

Figure 5: Measured and estimated execution times for Erlebacher with problem size $64 \times 64 \times 64$, double precision

Erlebacher: We used an inlined version of Erlebacher for the experiments, since the prototype implementation of the data layout assistant does not perform inter-procedural analysis. Erlebacher has 40 phases. There are no inter-dimensional alignment conflicts. The data layout selection step generated a 0–1 problem with 327 variables and 190 constraints. CPLEX solved the problem in 120 milliseconds on average on a SPARC-10.

The program consists of a three symmetric computations, each along one of the dimensions of the problem. The computations share access to a 3-dimensional, read-only array. All four 3-dimensional arrays are aligned canonically, i.e., there is no inter-dimensional alignment conflict. The choice of a static data layout leads to cross-processor dependences in exactly one of the three symmetric computations. Since the target compiler performs message vectorization but no coarse-grain pipelining or loop interchange, the particular loop order in the partitioned loops determines the granularity of the resulting pipelined execution.

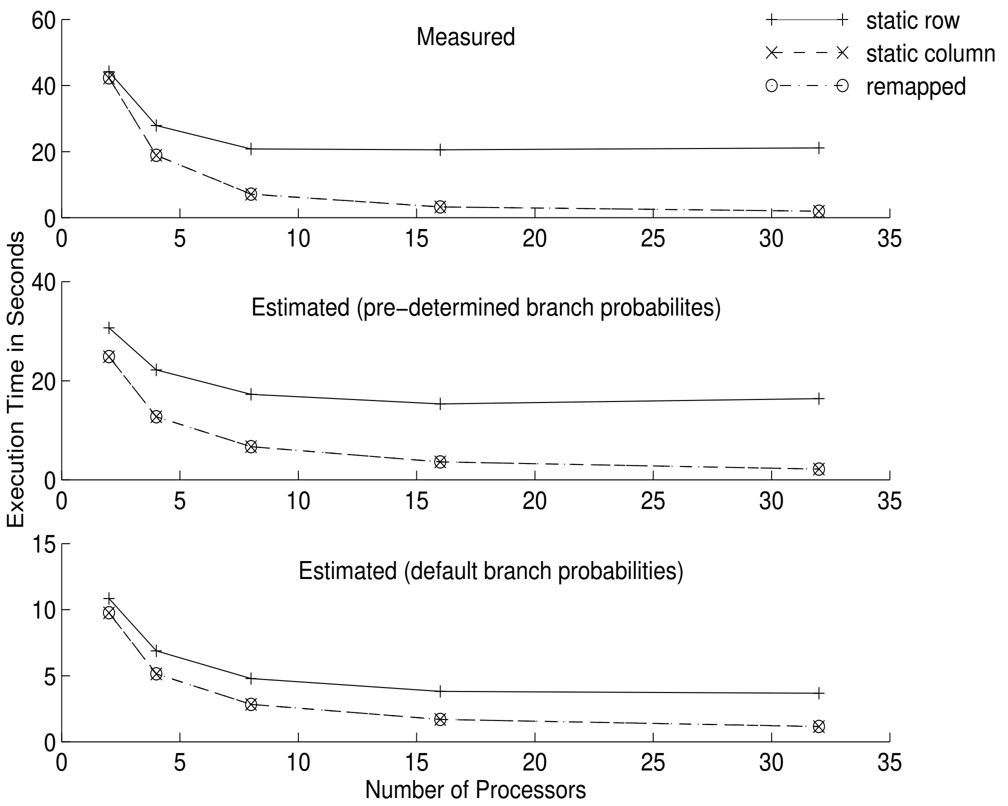

Figure 6: Measured and estimated execution times for Tomcatv with problem size 128×128, double precision (Note the different time scales)

We measured 21 test cases. Distributing the first dimension resulted in introducing a fine-grain pipeline which was never profitable. Introducing a coarse-grain pipeline by distributing the second dimension was the best choice in 9 cases. The last possible static data layout, namely distributing the third dimension, resulted in the sequential execution of one of the three symmetric computations. This choice was the best in 2 cases. Finally, using a dynamic data layout by remapping the read-only array once between a pair of symmetric computations was the best choice in 10 cases.

The prototype tool determined the best layout in 13 cases. Since the performance of the dynamic data layout and the static layout that distributes the second dimension were very close, the tool failed to rank them correctly in some cases. However, the incorrect ranking would have only resulted in a maximum performance loss of 8.6% as compared to the best possible data layout choice.

Figure 5 shows the measured and estimated execution times for the test cases of problem size $64 \times 64 \times 64$ and double precision. Although the different candidate layouts are ranked correctly, the layout that distributes the third dimension is overestimated by up to 60 %.

Tomcatv: In contrast to Erlebacher and Adi, Tomcatv has inter-dimensional alignment conflicts for two of its 2-dimensional arrays. The assistant tool partitioned the 17 phases into two classes and exchanged their inter-dimensional alignment information. The resulting alignment search spaces for each phase had two entries. Together with the two possible single dimension distributions, the final data layout search space contained four candidate layouts for most phases. Some phases had search spaces with only two entries, since the projection of phase partition layouts onto single phase layouts resulted in identical candidate data layouts.

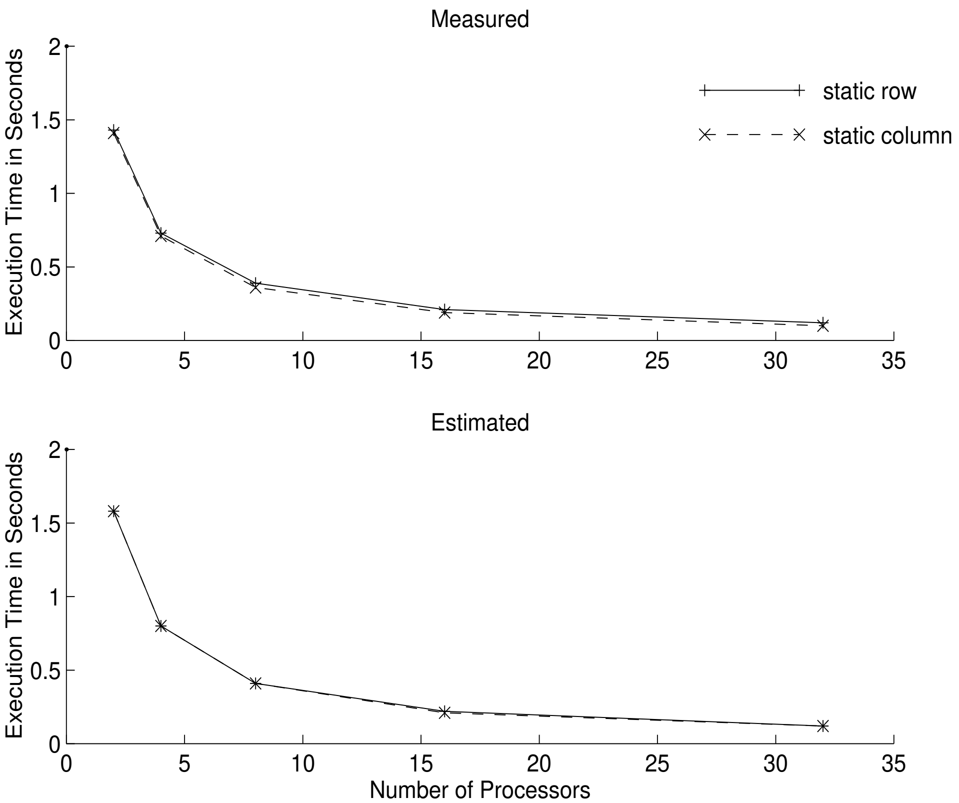

Figure 7: Measured and estimated execution times for Shallow with problem size 384×384, real

The two inter-dimensional alignment conflicts were translated into 0–1 problems with 312 variables and 530 constraints. Although the sizes of the two problems are the same, their objective functions are different, since the edge weights in the two merged CAGs are not identical. The sizes of the 0–1 problems are quite large since we scalar expanded all scalar temporaries. On a SPARC-10, CPLEX solved the two problems in 480 and 1030 milliseconds on average. The 0–1 formulation of the data layout selection problem had 336 variables and 203 constraints. CPLEX determined the optimal solution in 160 milliseconds on average on a SPARC-10.

We measured 19 test cases. In all but two cases, distributing the second dimension was the best choice. In all cases the prototype tool selected the column-wise data layout. The single wrong choice resulted in a performance degradation of 1.0% as compared to the best choice.

Figure 6 shows the measured and estimated execution times for the test cases of problem size 128×128 and double precision. Tomcatv has control flow inside its main iterative loop. The prototype implementation guesses a 50% branch probability. The bottom graph in Figure 6 shows the resulting estimates. However, if the actual branch probabilities are used, the performance prediction is more precise although still lower than the actually measured timings.

Shallow: The program is a 200 line benchmark weather prediction program developed by Paul Swarztrauber at the National Center for Atmospheric Research (NCAR) in Boulder, Colorado. It uses a two-dimensional, finite-difference model of the shallow-water equations. The main computations consist of two-dimensional stencils that can be parallelized in either dimension. However, a row distribution requires messages to be buffered. Therefore, the column distribution should perform slightly better than the row distribution.

Shallow has 28 phases. There are no interdimensional alignment conflicts. Each candidate layout search space has two layouts. The data layout selection problem is solved by CPLEX in 150 milliseconds on average. The 0–1 formulation has 228 variables and 200 constraints.

We ran 19 test cases. Column distribution was the best choice in all but one case. Our automatic data layout tool always picked the column distribution. The potential performance loss due to the single suboptimal selection was 1.8% as compared to the optimal choice. Figure 7 shows the 5 test cases for data type *real* and a problem size of 384×384. The static performance estimates slightly overestimate the measured timings. However, the relative performance is predicted with high accuracy.

5 Related Work

The problem of automatic data layout has been addressed by many researchers [AL93, CGS93, CGST93, Gup92, HA90, Keß93, KLS90, KLD92, LT93, LC90, RS89, BKK+94a, Who91]. The presented solutions differ significantly in the assumptions that are made about the input language, the possible set of data layouts, the compilation system, and the target machine architecture.

Our work is similar in nature to the recent work done by Anderson and Lam at Stanford University [AL93], Chatterjee, Gilbert, Schreiber, Sheffler, and Pugh at RIACS, Xerox Parc, and the University of Maryland [CGSS94, SSP+95], Ayguadé, Garcia, Girones, Labarta, Torres and Valero at the University of Catalunya in Barcelona, [AGG+94, GAL95], and Ning, Van Dongen, and Gao at CRIM and McGill University [NDG95].

In contrast to previous work, our framework is designed to be used inside a data layout assistance tool. To support user interaction, the framework builds and examines explicit search spaces of possible candidate layouts. Many previously published heuristics for choosing data layout candidates can be implemented in our framework.

Our framework can use techniques that may be too expensive to be included in a compiler, such as 0–1 integer programming. More recently, other researchers have started to investigate the feasibility of 0–1 integer programming techniques in the context of automatic data layout [GAL95, Phi95].

6 Summary and Future Work

The paper discussed a new framework for automatic data layout designed to be used in a data layout assistant tool. Since the tool is not part of a compiler, automatic data layout techniques that may be too expensive to be included in a compiler are used. The paper presented an efficient 0–1 formulation of the NP-complete inter-dimensional alignment problem. A prototype data layout assistant tool has been implemented based on our framework.

The prototype implementation uses the latest and most powerful general purpose techniques for linear and integer programming to solve two NP-complete problems optimally, namely the inter-dimensional alignment problem and the data layout selection problem.

A total of 99 experiments based on four scientific programs and program kernels were conducted. For three out of the four programs, the data layout choice is non-trivial, i.e., choosing the wrong layout results in substantial performance loss. Choosing the best layout is difficult since it involves complex trade-off decisions between minimizing communication overhead and maximizing usable parallelism. Making such trade-off decisions requires knowledge about the target compilation system, the performance characteristics of the target machine, the problem size, and the number of processors used.

In 84 cases, the tool selected the optimal data layout. In the cases where the tool selected a suboptimal layout, the performance loss incurred was within 9.3% of the optimal layout. All encountered instances of the inter-dimensional alignment problem and the data layout selection problem were solved in less than 1.1 seconds. This result shows that our framework is efficient and generates good data layouts.

We are currently extending our distribution analysis, compiler model, and execution model to handle multi-dimensional distributions. In addition, the framework will be extended to deal with programs that consist of multiple procedures in order to allow experiments on larger programs.

References

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

[AGG+94] E. Ayguadé, J. Garcia, M. Girones, J. Labarta, J. Torres, and M. Valero. Detecting and using affinity in an automatic data distribution tool. In *Proceedings of the Seventh*

Workshop on Languages and Compilers for Parallel Computing, Ithaca, New York, August 1994.

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

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

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

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

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

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

[CGS93] S. Chatterjee, J.R. Gilbert, and R. Schreiber. The alignment-distribution graph. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[CGSS94] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. Sheffler. Array distribution in data-parallel programs. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.

[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.

[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.

[GAL95] J. Garcia, E. Ayguadé, and J. Labarta. A novel approach towards automatic data distribution. In *Proceedings of the Workshop on Automatic Data Layout and Performance Prediction (AP'95)*, Houston, TX, April 1995.

[Gup92] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, September 1992.

[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.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs.* North Holland, New York, NY, 1977.

[Keß93] C.W. Keßler. Knowledge–based automatic parallelization by pattern recognition. In C.W. Keßler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 110–135. Verlag Vieweg, Wiesbaden, Germany, 1993.

[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.

[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.

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

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

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

[LT93] P. Lee and T-B. Tsai. Compiling efficient programs for tightly-coupled distributed memory computers. In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[NDG95] Q. Ning, V. V. Dongen, and G. R. Gao. Automatic data and computation decomposition for distributed memory machines. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, January 1995.

[NW88] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, 1988.

[Phi95] M. Philippsen. Automatic alignment of array data and processes to reduce communication time on DMPPs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

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

[SSP+95] T. J. Sheffler, R. Schreiber, W. Pugh, J. R. Gilbert, and S. Chatterjee. Efficient distribution analysis via graph contraction. In *Proceedings of the Workshop on Automatic Data Layout and Performance Prediction (AP'95)*, Houston, TX, April 1995.

[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.

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

Appendix

Alignment Conflict Resolution as a 0–1 Problem

This section describes the translation of an instance of the inter-dimensional alignment problem into an instance of a 0–1 integer programming problem with linear constraints. Note that there are many possible translations. Experiments showed that our formulation is very promising. A proof of the correctness of our formulation is given in [Kre95]. An introduction to integer programming can be found in [NW88].

Definition An instance of the inter-dimensional alignment problem with d dimensions consists of finding a d-partitioning of an undirected, weighted component affinity graph (CAG) such that the sum of the weights of the edges that cross distinct partitions is minimized.

Definition An instance of the 0–1 problem consists of a set of variables X, a set of linear constraints over the variables in X, and a linear objective function with domain X. A solution to an instance of the 0–1 problem is a function $s_{01} : X \rightarrow \{0,1\}$ that minimizes (or maximizes) the objective function while respecting the constraints.

In the following, we will discuss the translation of a d-dimensional alignment problem into a 0–1 problem. We will assume that all arrays represented in the CAG have d or less dimensions. Let a_i be the node in the CAG that represents the i-th dimension of array a, $1 \leq i \leq dim(a)$, where $dim(a)$ denotes the number of dimensions of array a. Each such node is represented by d variables or switches in X, $a_{ik} \in X$, $1 \leq k \leq d$. The switch a_{ik} will be *on* if and only if the node a_i belongs to the k-th partition in the final solution. Let $e = (a_i, b_j)$ be an edge in the CAG. Each edge e is represented by d variables or switches in X, $a\$b_{jk}^{ik} \in X$, $1 \leq k \leq d$. In the final solution, the switch $a\$b_{jk}^{ik}$ is *on*, if and only if the sink and the source of the edge e belong to the same partition.

There are two types of constraints. *Node constraints* ensure that any solution is a d-partitioning of the CAG, and *edge constraints* identify edges with source and sink nodes in the same partition.

Node constraints: To ensure that an array dimension is in exactly one partition, constraints of the form $\sum_{k=1}^{d} a_{ik} = 1 \;\; (type1)$ are introduced for each node a_i. Two dimensions of the same array must not be in the same partition. This property is enforced by the following constraints, one constraint for each pair of an array a and a partition k, $1 \leq k \leq d$: $\sum_{i=1}^{dim(a)} a_{ik} \leq 1 \;\; (type2)$. Note that in the case of an embedding of array a, $dim(a) < d$, some partition k will not contain any CAG node associated with a. There are $|N|$ node constraints of $(type1)$ and $d|Arrays|$ node constraints of $(type2)$, where N is the set of nodes in the CAG, and $Arrays$ is the set of arrays represented in the CAG.

Edge constraints: The formulation of the inter-dimensional alignment problem uses counting arguments on the number of incoming and outgoing edges of nodes in the CAG. For each node a_i, constraints are introduced for incoming and outgoing edges. The translation requires a directed graph. The particular direction of edges in the CAG is irrelevant for the correctness of the formulation. However, the direction influences the form and number of edge constraints, and therefore has an impact on the performance of the generated 0-1 problem instance. An edge direction normalization step ensures that for any pair of arrays (a, b), all edges between nodes that represent a dimension of a and a dimension of b have the same direction, e.g., are all oriented "from a to b".

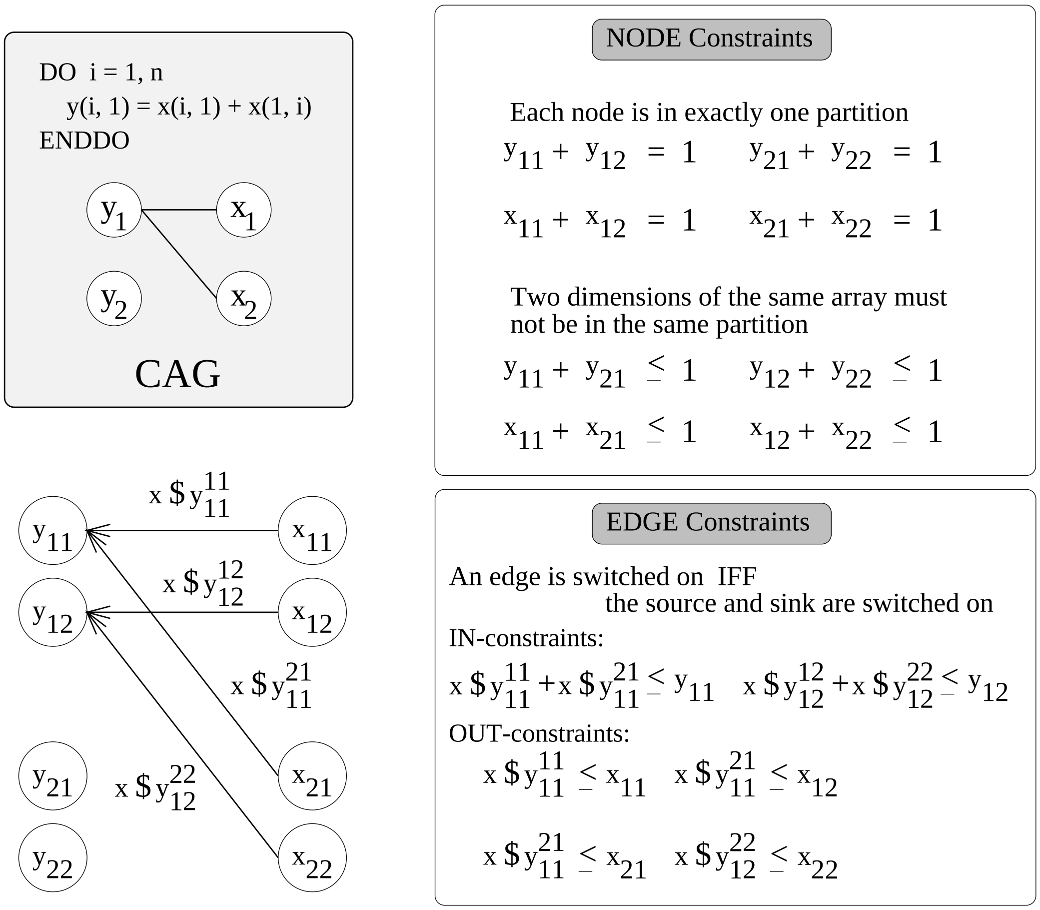

Figure 8: Alignment conflict resolution of an example CAG as a 0–1 integer programming problem

Let $SRC(b, a_i)$ denote the set of all nodes b_j that represent a dimension of array b and there is an edge from b_j to a_i. For each k, $1 \leq k \leq d$, and each non-empty set $SRC(b, a_i)$, IN-constraint of the form $\sum_{b_j \in SRC(b,a_i)} \; b\$a_{ik}^{jk} \leq a_{ik}$ are introduced. Let $SINK(a_i, c)$ denote the set of all nodes c_j that represent a dimension of array c and there is an edge from a_i to c_j. For each k, $1 \leq k \leq d$, and each non-empty set $SINK(a_i, c)$, OUT-constraint of the form $\sum_{c_j \in SINK(a_i,c)} \; a\$c_{jk}^{ik} \leq a_{ik}$ are introduced.

The total number of edge constraints is $\mathcal{O}(2d|E|)$, where E is the set of edges in the CAG. Each edge occurs in exactly two constraints, a IN-constraints of its sink node and a OUT-constraints of its source node. To be more precise, the number of edge constraints is the number of nonempty SRC and $SINK$ sets, multiplied by d. In the worst case, each SRC and $SINK$ set contains only one edge, resulting in $2d|E|$ edge constraints.

Figure 8 shows the constraints resulting from the translation of an example CAG with an alignment conflict into a 0–1 problem. The CAG was generated for a loop with a two-dimensional program template. Therefore, conflict resolution requires a minimal cost two-partitioning of the example CAG. Edge weights are not shown here since they are only relevant for the objective function and not for the constraints. The graph below the CAG in Figure 8 is not actually generated during the translation process, but illustrates our 0–1 formulation.

Objective function: A solution of the 0–1 problem formulation of the inter-dimensional alignment problem *maximizes* the following objective function under the given constraints:

$$\sum_{(a_i,b_j)\in E} \sum_{k=1}^{d} a\$b_{jk}^{ik} \;\; weight\ (a_i, b_j)\ .$$

The switch $a\$b_{jk}^{ik}$ is *on*, if and only if the corresponding edge is inside a partition. A solution maximizes the edge weights inside a partition and thereby minimizes the edge weights across different partitions.