Automatic Data Layout for Distributed Memory Machines

Ulrich Kremer

COMP TR96-261

October, 1995

Department of Computer Science
Rice University
6100 South Main Street
Houston, TX 77005-1892

RICE UNIVERSITY

Automatic Data Layout for Distributed Memory Machines

by

Ulrich Kremer

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Ken Kennedy, Noah Harding Professor
Computer Science
Rice University

John Mellor-Crummey, Faculty Fellow
Computer Science
Rice University

Keith Cooper, Associate Professor
Computer Science
Rice University

Robert Bixby, Professor
Computational and Applied Mathematics
Rice University

Dan Sorensen, Professor
Computational and Applied Mathematics
Rice University

Houston, Texas
October, 1995

Automatic Data Layout for Distributed Memory Machines

Ulrich Kremer

Abstract

The goal of languages like Fortran D or High Performance Fortran (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 challenge in writing an efficient program in such languages. The performance of a data layout depends on the target compilation system, the target machine, the problem size, and the number of available processors. This makes the choice of a good layout extremely difficult for most users of such languages.

This thesis discusses the design and implementation of a data layout selection tool that generates Fortran D or HPF style data layout specifications automatically. Because the tool is not embedded in the target compiler and will be run only a few times during the tuning phase of an application, it can use techniques that may be considered too computationally expensive for inclusion in today's compilers.

The proposed framework for automatic data layout selection builds and examines explicit search spaces of candidate data layouts. A candidate layout is an efficient layout for some part of the program. After the generation of search spaces, a single candidate layout is selected for each program part, resulting in a data layout for the entire program. A good overall data layout may require the remapping of arrays between program parts. A performance estimator based on a compiler model, an execution model, and a machine model is used to predict the execution time of each candidate layout and the costs of possible remappings between candidate data layouts. The machine model uses the novel training set approach which determines the costs of arithmetic operations and simple communication patterns.

In the proposed framework, instances of NP-complete problems are solved during the construction of candidate layout search spaces and the final selection of candidate layouts from each search space. Rather than resorting to heuristics prematurely,

the framework capitalizes on state-of-the-art 0–1 integer programming technology to compute optimal solutions of these NP-complete problems.

A prototype of the data layout assistant tool has been implemented. Experiments indicate that good data layouts can be determined efficiently.

Acknowledgments

I am very grateful to my thesis advisor Ken Kennedy for his encouragement and advice all through my graduate career. I feel privileged that he gave me the opportunity to work with him in an excellent research environment. I wish to thank my other thesis committee members Robert Bixby, Keith Cooper, John Mellor-Crummey, and Dan Sorensen for their time and support. In particular, John was always willing to listen to ideas and gave me important guidance, and Bob provided me with insights into the world of 0–1 integer programming. I also would like to thank Geoffrey Fox who showed me what scientific computing is really all about.

Throughout graduate school I have been fortunate to meet and enjoy the company of some excellent researchers and good friends. Vasanth Bala, Jaspal Subhlok, Marina Kalem, Willy Zwaenepoel, Kathryn McKinley, Mary Hall, Ervan Darnell, Paul Havlak, Alan Carle, and Laura Arbilla gave me much needed advice particularly during my first years at Rice. I will always remember the basketball games in the unairconditioned Rice gymnasium (you had to keep running if you wanted to stay cool) or the late night programming sessions just before an upcoming site visit (this was the time when I started to appreciate the caffeine content of a Dr.Pepper soda).

Chau-Wen Tseng, Seema Hiranandani, Ravi Mirchandaney, Jerry Roth, René Rodriguez, Mike Paleczny, Kathi Fletcher, Reinhard von Hanxleden, Nat McIntosh, Lisa Thomas, Vikram Adve, Sandhya Dwarkadas, Alan Cox, Dejan Mircevski, Nenad Nedeljkovic, Ajay Sethi, Debbie Campbell, Kevin Curetonk, Chuck Koelbel, (Mootaz) Elmootazbellah Elnozahy, Pete Keleher, Karim Esseghir, John Carter, Joe Warren, and Marcelo Ramé became fellow graduate students or colleagues a few years later. In particular I would like to thank Chau-Wen and Seema for all the compiler discussions and suggestions regarding places to go for lunch, Reinhard and Nat for being great office mates, the '59–er club (Jerry, René, and Mike) for sharing my age bracket, Debbie for proof reading drafts of papers written in nearly perfect Änglisch, Kevin for managing the D System and helping me out so many times during my implementation work, Lisa for not contributing any research ideas to my thesis and keeping my spirits up, and Joe for many competitive tennis matches. All these people made my life so much more enjoyable. Thanks a lot, guys!

I also would like to mention a very important institution, the graduate student deli and bar at Rice commonly referred to as Valhalla. It provided me with my staple food, the Valhallawich, a sandwich immune to changes in quality and consistency over the years (a Valhallawich is arguably the bottom element in the lattice of sandwiches).

My deepest gratitude, however, is reserved for those to whom this thesis is dedicated: my parents and my wife Devyani. Devyani had to endure many of my thesis induced mood swings, ranging from workaholic to high-performance couch potato. Thank you very much for sticking with me and helping me through all this.

For Mom, Dad, and Devyani

Contents

Illustrations

Chapter 1

Introduction

1.1 Background

Distributed-memory multiprocessors provide the necessary scalability to solve many large scientific problems in physics, chemistry, and biology. These problems are usually so computational or memory intensive that they cannot be solved using today's single-processor architectures. Typically, distributed-memory multiprocessors provide a message passing programming model which requires explicit management of parallelism and local name spaces. As a consequence, most programmers find these machines extremely hard to use. In addition, once a program has been written in this model, porting it to another architecture is a tedious task at best.

The goal of High Performance Fortran (HPF) is to provide a machine-independent parallel programming model for scientific problems. HPF augments Fortran with statements that allow the programmer to specify the layout of the program's data. A data layout is a mapping of array elements onto the local memories of a distributed-memory machine. The compiler uses this mapping to partition the arrays across the local memories of the distributed-memory machine and to generate message-passing code based on the array partitioning. Porting a HPF program from one target machine to another is accomplished by a single recompile. However, to ensure good performance of the ported code on the new architecture, the user may have to modify the data layout specifications since many compiler decisions are driven by the data layout specified in the program.

While HPF supports a single name space programming model, the choice of an efficient data layout is still left to the user. The quality of a data layout depends not only on the target distributed-memory architecture, but also on the target HPF compiler, the problem size, and the number of processors available. The option of remapping arrays at specific points in the program makes the choice of an efficient data layout even harder.

Besides the algorithm selection, the data layout choice is the key intellectual step in writing an efficient HPF program. Although finding an efficient data layout fully automatically may not be possible in all cases, HPF users will need support during the difficult 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.

1.2 Thesis

This thesis discusses techniques for automatic data layout for regular problems in an interactive data layout assistant tool. The envisioned tool will analyze Fortran programs and will generate efficient HPF data layout specifications. The application scenario for the data layout assistant tool is shown in Figure 1.1. 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. Our automatic techniques assume that different data layouts can be specified for different program sections. We envision the data layout assistant tool to 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 completely specified data layout. The data layout is optimized for a target compiler, a target distributed-memory machine, a problem size, and a number of available processors. This implies that these entities have to be known at tool invocation time.

Thesis *Data layouts of high quality can be generated efficiently for regular problems in the context of an automatic data layout assistant tool.*

Since the data layout assistant is not part of a compiler, and will run only a few times during the tuning process of a program, it can use techniques that may be too computationally expensive to be included in a compiler. Note that an automatically generated data layout can be used for different problem sizes and numbers of processors, although its quality may be suboptimal.

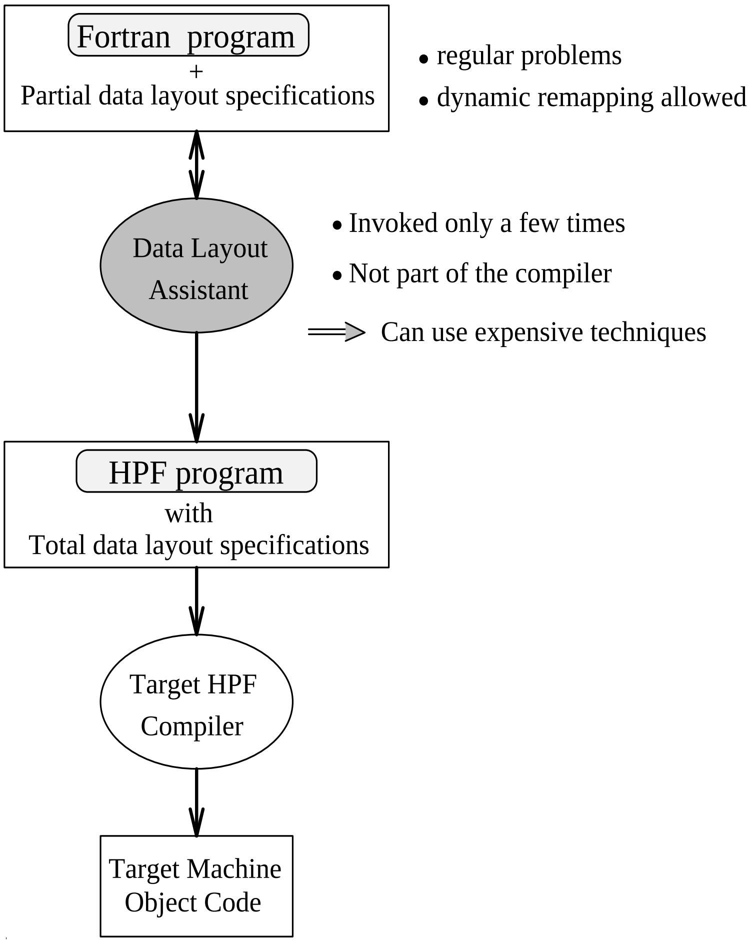


Figure 1.1 Automatic data layout as part of a data layout assistant tool

1.3 Contributions

Our new framework for automatic data layout consists of two phases. The first phase gathers information about the program and its data usage, resulting in partitioning of the program into program segments and sets of explicit search spaces of candidate data layouts for each such segment. A data layout for a program segment is a mapping of every array referenced in the segment onto the target architecture. The mapping is defined in two stages, typically referred to as "alignment" and "distribution". A candidate data layout is an efficient data layout for some part of the program. The goal of the first phase is to identify potential points of data remapping and to restrict the sizes of the candidate data layout search spaces without missing any promising candidates. The second phase predicts the performance of each candidate layout and selects the combination of data layouts that will result in the best overall performance for the entire program.

The explicit construction of candidate data layout search spaces for each program segments and the static evaluation of each candidate in the search spaces are the basis for user interaction with the envisioned data layout assistant tool. 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.

Rather than resorting to heuristics prematurely, our work capitalizes on 0-1 integer programming technology to compute optimal solutions for two NP-complete problems in our framework [LC90a, Kre93b]. Experiments indicate that through the use of the latest and most powerful general purpose techniques for linear and integer programming, computing the optimal solution of the two NP-complete problems is efficient in practice in the context of a programming environment. All encountered instances of the two NP-complete problems were solved in a matter of seconds [BKK94, KK95]. Contrary to the common belief in the compiler and programming environment community, this new result suggests that not all NP-complete problems encountered in compilers or programming environments may have to be approximated using heuristics. In cases where optimal solutions cannot be determined within an acceptable time bound, 0–1 integer programming technology can be used to produce efficient approximate solutions.

Another unique aspect of the framework is its new approach to performance estimation. To predict the performance of each candidate data layout in the search space of a program segment, a compiler model is used to determine the computation and communication that would be generated by the target compiler, if the program segment were compiled under the candidate layout. The compiler model ignores computations and communications that are not crucial for the overall performance of the compiler generated code. Based on the resulting information, an execution model and target architecture machine model computes the overall execution time. The machine model uses our training set approach [BFKK91, HKK+91].

A prototype based on our framework has been implemented as part of the D system [ACG+94]. Experiments showed that through the use of 0-1 integer programming and our new approach to performance prediction, good data layouts can be determined efficiently. Our automatic data layout framework is a big step towards making distributed-memory multiprocessors truly usable.

1.4 Overview

A short introduction to High Performance Fortran and 0–1 integer programming can be found in Chapter 2. Techniques for automatic data layout have been investigated for nearly a decade. Chapter 3 contains a discussion of different formulations of the data layout problem and their proposed solutions.

Our framework for automatic data layout consists of four steps. Each of these steps is discussed in detail in Chapter 4. In the first step the input program is partitioned into program segments (Section 4.2). For each such program segment, the second step constructs a search space of promising candidate layouts (Section 4.3). 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 (Section 4.4). 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 (Section 4.5).

In order to be able to evaluate the efficiency and quality of our framework for automatic data layout, a prototype data layout assistant tool has been implemented. Chapter 5 describes the prototype implementation and shows that the prototype tool is efficient and generates data layouts of high quality. A summary of the contributions of this thesis and a discussion of future work can be found in Chapter 6.

Chapter 2

Background

The goal of languages such as High Performance Fortran (HPF) and Fortran D is to provide a machine-independent programming model that supports fine-grain data parallelism found in scientific codes. Section 2.1 reviews the data layout directives of HPF relevant to our work on automatic data layout. The HPF language specification document contains a full language description [Hig93].

Some problems encountered in our framework for automatic data layout have been shown to be NP-complete. As part of this thesis we have translated instances of these NP-complete problems to instances of 0–1 integer programming problems. This translation allows us to solve our NP-complete problems optimally or compute approximations efficiently by taking advantage of the latest technology in 0–1 integer programming.

An introduction to 0–1 integer programming and a discussion of the state-of-the-art integer programming technology is presented in Section 2.2.

2.1 HPF Data Layout Directives

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 target parallel machine.

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

The `ALIGN` statement maps arrays onto templates. Arrays mapped to the same template are automatically aligned with each other. Alignment can take place either within a dimension (*intra-dimensional alignment*) or across dimensions (*inter-dimensional alignment*). The alignment of arrays to templates is specified by placeholders `I, J, K, ...` in the subscript expressions of both the array and template. In the example below,

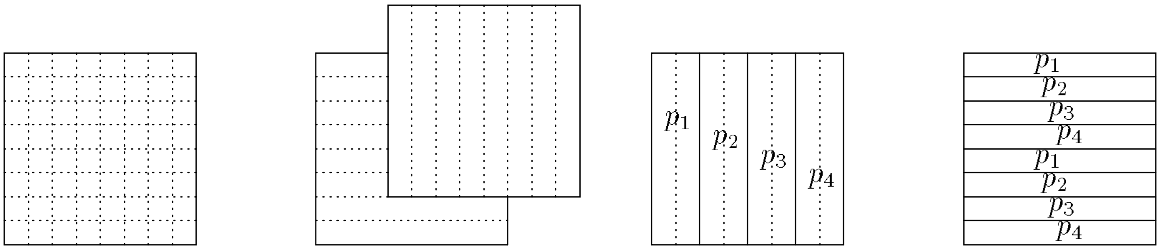
```
        REAL X(N,N)
!HPF$   TEMPLATE A(N,N)
!HPF$   ALIGN X(I,J) WITH A(J-2,I+3)
```

A is declared to be a two dimensional template 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 template, the `DISTRIBUTE` statement maps the template to the finite resources of an abstract machine. An abstract machine is a rectilinear processor arrangement with a name, rank, and number of processors in each dimension. A `PROCESSORS` statement defines an abstract machine. Distributions are specified by assigning an independent *attribute* to each dimension of a template. Predefined attributes are `BLOCK` and `CYCLIC(i)`, where `i` is the block size of a block-cyclic distribution. The symbol "$*$" marks dimensions that are not distributed. Choosing the distribution for a template maps all arrays aligned with the template to an abstract machine. In the following example,

```
!HPF$   TEMPLATE A(N,N), B(N,N)
!HPF$   PROCESSORS PROCS(4)
!HPF$   DISTRIBUTE A(*,BLOCK) ONTO PROCS
!HPF$   DISTRIBUTE B(CYCLIC(1),*) ONTO PROCS
```

distributing template A by `(*,BLOCK)` results in a column partition of arrays aligned with A. Distributing B by `(CYCLIC(1),*)` partitions the rows of B in a round-robin



```
TEMPLATE        REAL X(8,8)          DISTRIBUTE       DISTRIBUTE
   A(8,8)       ALIGN X(I,J)           A(*,BLOCK)       A(CYCLIC(1),*)
                 WITH A(J-2,I+3)        ONTO PROCS        ONTO PROCS
```

Figure 2.1 HPF Data Layout Specifications

fashion among the processors. These sample data alignment and distributions for the case `N=8` are shown in Figure 2.1.

HPF supports dynamic remapping, i.e., the data layout of arrays can change during the execution of the program. `REALIGN` and `REDISTRIBUTE` statements are used to specify remappings. In contrast to their "static" versions, realignment and redistribution statements are executable statements and not declarative.

We should note that the goal in designing HPF 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. The design of HPF has been influenced by languages such as FORTRAN D [FHK+90], CM FORTRAN [TMC89], KALI [KM91], and Vienna Fortran [CMZ92].

2.2 0–1 Integer Programming

Integer programming can be used to solve many real world problems that require the management and efficient use of scarce resources to improve productivity. Examples of such problems are VLSI circuit design, airline crew scheduling, and communication and transportation network design. An instance of an integer programming problem consists of a set of variables, a set of inequality and equality constraints, and an objective function. A solution of the integer programming instance assigns integral values to all variables such that the objective function is maximized or minimized while all constraints are respected. If the integrality restriction is relaxed for some variables, the problem is called a *mixed integer programming problem.*

A *0–1 linear integer programming problem* is a special case of an integer programming problem where variables can only be assigned the integral values 0 or 1, and all constraints are linear functions of the variables. Solving a 0–1 integer programming problem has been shown to be NP-complete. In this thesis we will refer to a 0–1 linear integer programming problem as a *0–1 problem.* An in-depth discussion of integer programming can be found in [NW88].

For decades, the integer and combinatorical optimization community has been working on methods to solve integer programming problems fast in practice. The ability to solve integer programming problems has been remarkably improved over the last five to ten years, particularly 0–1 problems such as those being generated by our framework for automatic data layout. The basic technique for solving integer programming problems is to apply intelligent branch-and-bound using linear programming at the nodes. Important improvements have occurred in three areas. First, linear programming codes are on average approximately two orders of magnitude faster than they were five years ago, particularly for larger problems [Bix94]. Combined with the improvements in computing speed over that same period these codes represent an approximate four orders of magnitude improvement in our ability to solve linear programming problems.

The second major development is in so-called cutting-plane technology. Motivated by work of Dantzig, Johnson and Fulkerson in the 50's [DFJ54], Padberg, Groetschel and others have shown how cutting-plane techniques could be used to strengthen the linear programming relaxations of many 0–1 integer programming problems [PR91]. The strengthening is effected by studying the facets of the underlying polytope generated by the convex hull of 0–1 solutions. Knowledge of these facets leads to subroutines for recognizing inequalities violated by the current fractional solution. These violated inequalities can then be added to the linear programming formulation in lieu of branching.

The third major area of improvement has come in the application of parallel processing to handle the branching when cutting planes do not succeed in sufficiently strengthening the linear programming formulation. Parallelism is particularly appropriate for current cutting-plane methods because cuts are computed not only at the root node but at all nodes in the branching tree. The extra computation at the nodes has the effect of making the computations sufficiently coarse grained that communication costs need not be significant. The most striking example of an integer programming success story exploiting all of the above advances is the recent work of Applegate, Bixby, Cook and Chvatal in which a 4461 city traveling salesman problem was solved to exact optimality using a complex branch-and-cut code running on a network of up to 60 loosely connected workstations [ABCC93].

Chapter 3

Related Work

Compiling a single name space program for a physically distributed-memory architecture requires the mapping of the program's data and computation onto the processors of the target machine. If there is a mismatch between the data mapping and the computation mapping, i.e., the data needed for a computation is not assigned to the same processor as the computation, communication will be necessary. A good data and computation mapping minimizes the performance loss due to communication and maximizes load balancing.

The problem of automatic data and computation mappings for scientific applications have been discussed extensively in the literature. We will focus our discussion on compile-time approaches that map elements of dense arrays to processors, and that use single loop iterations as the basic units of computation. Other approaches may map parts of high-level data structures such as trees or lists, or low-level data structures such as memory pages or cache lines to processors. Computation mappings in other approaches may be based on fine-grain computations such as single floating point operations, or coarse-grain tasks consisting of entire subroutines.

The proposed solutions for automatic data layout and computation mapping for dense arrays differ significantly in the assumptions that are made about the input language, the possible sets of data and computation mappings, the compilation system, and the target machine architecture. For instance, the input language may be a functional or imperative language, the data and computation mappings may be chosen from a restricted set or be rather general, the compilation system may perform a variety of different optimization transformations or only a few basic optimizations, or the target machine may support a SIMD or MIMD parallel programming model, or a shared-memory parallel programming model. Corresponding to its set of assumptions, each approach uses a different representation and formulation of the data and computation mapping problems.

Since performance tradeoff decisions are crucial in the selection process, performance models to estimate the qualities of the considered data and computation mappings are a central component of every automatic data layout and computation partitioning approach. The performance models used have to be precise enough to distinguish all data and computation mapping alternatives of the particular approach.

Most previous approaches to automatic data layout and computation partitioning have been developed as part of an optimizing compiler, i.e., have been designed as an optimization pass in the compiler. As a consequence, the design of the automatic partitioning pass has to guarantee an acceptable response time since the pass is invoked during every compile. Typically, a response time may be considered acceptable if it is in the order of seconds or a few minutes, but not in the order of hours. In contrast, automatic data layout techniques outside of a compiler, i.e., as part of a data layout tool may not be subject to the same time constraints.

In this thesis we have developed a framework for automatic data layout for Fortran programs designed to be used in an interactive data layout assistant tool. Instead of assuming a fixed compilation system, the framework can be retargeted for different compilers. This is done by changing parameters of the compiler model in the framework's data layout performance estimator. The framework generates HPF or Fortran D style data layout directives. Data layouts are optimized for a target compiler, target machine, problem size, and number of available processors. The target machine is assumed to support a MIMD programming model.

A detailed discussion of some of the related work on automatic data and computation mapping, and source-level performance estimation is given below. Since this thesis deals mainly with the data mapping problem, i.e., assumes that the computation mapping is induced by the data mapping, we will emphasize the data mapping aspects of the related approaches discussed in the literature.

3.1 Automatic Data Layout

The division of the related work into static and dynamic mappings is more a historic one. In general, finding a single data layout for the entire program is considered a simpler problem than allowing dynamic changes of the data mapping.

3.1.1 Static Data Mappings

Some early work on static data and computation mapping used data dependence information to determine whether a communication-free partitioning of the iteration space of a single loop nests was possible [RS89, D'H89, HA90]. The communication-free computation mapping induces the corresponding data mapping.

The majority of approaches emphasize the data mapping view of the mapping problem. The decomposition of the data mapping into alignment and distribution lead to a two step strategy, where alignment analysis is followed by distribution analysis [AKLS88, Wei91, LC90a, LC91b, GB91, GB92b, Who92a, CGST93]. Since alignment and distribution decisions may be mutually dependent, performing distribution after alignment may lead to inferior results. To avoid this problem, some approaches consider alignment and distribution at the same time [Keß93, GAL95].

Crystal

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 ϵ, 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].

Parafrase-2

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, GB92b]. 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]. 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 were 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.

ALEXI

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]. 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 best data layout 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.

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. Interprocedural 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.

3.1.2 Static and Dynamic Data Mappings

Dynamic data mappings allow the remapping of data at specific points in the program. The identification of program segments in which data can be statically mapped, and the accurate modeling of the potential remapping costs make the dynamic data mapping problem harder than the static problem. The smallest possible statically mapped program regions may be single statements [KN90, CGST93, Phi95], loop nests [AL93, LT93, AGG+94, NDG95, PB95], or groups of statements or loop nests for which it can be shown that remapping between them can never be profitable [SSP+95]. The approaches for dynamic data mappings differ in the set of static mappings considered for each program region and in the techniques used to perform the final selection among the considered candidate mappings.

A unique aspect of this thesis is its efficient formulation of two NP-complete automatic data layout subproblems as 0–1 integer programming problems. The choice of the particular integer programming formulation is the crucial step for achieving good performance. Although using exact solutions for NP-complete problems is a rather new idea, a few researchers have already recognized the potential benefits of using 0–1 integer programming or general integer programming as part of a compiler or programming environment. Pugh developed a dependence analysis test, called the Omega Test based on an integer programming algorithm [Pug91]. Using integer programming for instruction scheduling under resource constraints for super-scalar machines has been discussed by Feautrier [Fea94] and Ning, Govindarajan, Altman and Gao [NG93, AG94, AGG95]. Integer programming techniques in the context of a distributed-memory compiler have been discussed by Phillipsen [Phi95] and Garcia, Ayguadé and Labarta [GAL95]. The latter two works have been based on our experience with 0–1 integer programming for efficient solutions of NP-complete problems in an automatic data layout tool [BKK94, KK95].

Compass

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. 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.

Projects 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 template (HPF 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.

SUIF

Anderson and Lam discuss a compiler algorithm for automatic data and computation mapping for dense matrix computations [AL93]. The input programs consist of generally nested loops with explicit parallelism. Each loop can be either a *forall*, *doacross*, or sequential loop. Loop bounds and array subscripts are assumed to be affine functions of the loop indices and symbolic constants.

The focus of their work is on finding an efficient data and computation mapping onto a virtual processor array. Such a mapping can be referred to as a data and computation alignment. They do not address issues related to distribution, such as load balancing, finding the right block size for block-cyclic distributions, or determining the number of physical processors for each distributed dimension. However, their work handles data and computation alignments that may change dynamically., i.e., are not the same for the entire program.

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

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

Anderson's and Lam's algorithm for automatic data layout and computation partitioning considers dynamic data remapping. The program is represented as a weighted, undirected graph where the nodes correspond to loop nests and edges represent possible data remappings between loop nests. The node weights are estimated loop execution counts and the edge weights are the combination of worst-case approximations of the actual communication costs and the probabilities that the data remappings will actually occur. The performance model for remappings does not take different distributions into account. Anderson and Lam show that their problem representation of dynamic data layout and computation partitioning is NP-hard.

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

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

Paradigm

Palermo and Banerjee extend previous work on automatic data layout by Gupta and Banerjee to handle dynamic remapping (see Section 3.1.1). Their approach is based on a hierarchical decomposition of the program into components such that remapping is profitable only between these components [PB95]. Each decomposition step consists of splitting a single component into two subcomponents if the best static data layout for the entire component has a higher cost than the sum of the costs for the best data layouts for each subcomponent. The splitting step does not consider the remapping costs between components. However, the splitting process is stopped once the cost of a component is below a predefined threshold.

The program decomposition algorithm uses a directed graph that is built based on the best static data mapping for the entire program. Nodes in the *communication graph* represent single program statements and edges correspond to data flow dependences between statements. Edges are weighted by the communication costs induced by the initial static data mapping. A maximal cut algorithm determines the split point in a communication graph. This process is repeated until splitting is no longer profitable.

After the program has been recursively decomposed, redistribution costs are estimated and all encountered components and subcomponents are represented in a single, acyclic graph, the *phase transition graph*. The result of the solution of a single shortest path problem over the phase transition graph yields the final data layout for the entire program.

The described algorithms does not perform interprocedural analysis. It is not clear how their work can be extended to handle control flow such as branches.

3.2 Performance Estimation

Performance is the most important issue for programs running on parallel platforms. Some applications are executed on a parallel machine mainly because of the amount of main memory that such machines provide. However, the majority of applications try to achieve a significant reduction in execution time as compared to a sequential execution.

Parallel machines are complex and expensive computing resources. The ability to predict the performance of an application without actually executing it on the parallel machine is crucial to make efficient use of the expensive resource. Performance models are used to predict the benefits of parallel execution for an application, to tune the performance of a parallel application, or to allow compilers to choose between different code improvement transformations. These different application scenarios impose different requirements on the performance models, in particular with respect to the precision of the generated estimates.

Performance models that support benefit analysis can be used to "recruit" new users for a specific parallel architecture [PIJJ88] or to predict the scalability of a parallel application under varying machine and problem parameters [Tol95]. Decisions to improve program performance require knowledge about the performance characteristics of the program. Interactive programming tools may use dynamic profiling and/or static performance estimation to identify code segments critical for the overall program performance [AMCA+95, MCAK94, HKTW94, Ree94]. This information can be used to guide the user's program tuning efforts or to direct more aggressive compile-time analysis and code improvement techniques to the identified performance "hot spots". Compilers use static performance models to predict the performance benefits of program transformations including automatic data layout [GB92a, WL91, CMT94, CGST93, Wan94]. Different transformations require different performance estimation accuracy.

In this section we will focus on a discussion of performance models that can be used to support automatic data layout at the program source code level. Static performance estimation models are crucial components in every automatic data layout approach. Performance models are needed at different levels of program and system abstractions. The resulting performance estimates have to be precise enough to distinguish the data layouts considered by the corresponding automatic data layout approach.

IBM's Static Performance Estimation for High-Level Languages

Wang has developed a framework for static performance prediction for high-level languages such as HPF running on superscalar-based, parallel machines [Wan93, Wan94].

The goal of the performance estimation is to support the selection of optimizing transformations for different target machines.

In a first step, the framework maps high-level language statements into low-level machine instructions. This mapping takes into account common compiler back-end optimizations such as sum-reductions, common subexpression elimination, invariant code motion, loop unrolling, and branch optimization. In a second step, a machine dependent cost model determines a symbolic cost function for the low-level instruction stream. The cost model uses a machine model and data dependence information to simulate instruction scheduling algorithms common for superscalar architectures. If communication instructions are present, communication and synchronization costs are represented in the computed performance expression. There is no compiler model to simulate high-level optimization transformations in response to a particular data layout selection. Wang assumes that this information is input to his framework.

The overall cost function for a program is determined as follows. First cost functions for straight line code, i.e., code without branches or loop structures are computed. Then, based on performance expressions for straight line code, performance estimates for loop structures and branches are computed. The resulting performance expressions may contain symbolic variables representing control information or other unknowns.

Wang's approach tries to eliminate the propagation of errors due to insufficient control flow information and allows the symbolic comparison of performance expressions. In addition, "sensitivity analysis" can be applied to performance expressions to identify variables that have the greatest impact on the overall performance. Non-sensitive variables may be eliminated from a performance expression without significant loss of performance prediction accuracy.

The framework has been implemented for straight-line code. Unfortunately, experimental data to verify the accuracy and efficiency has not been provided.

Superb-2

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 3.1.1). 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 proposed tool is currently being implemented at the University of Vienna. A prototype performance estimator has been implemented. [FBZ92, Fah92]. No experimental results have been published.

Syracuse's Performance Interpretation Engine

Parashar, Hariri, Haupt, and Fox present an interpretive approach for performance prediction in a high performance computing environment [PHHF94]. An "interpretation engine" uses information about the target machine and the program characteristics to determine different performance metrics such as estimated execution time, estimated communication and computation times. The authors introduce a system abstraction methodology to characterize the behavior of the target system in a hierarchical fashion. The program is represented in a way that allows performance prediction based on the system abstraction.

Their framework for performance prediction is designed to support (1) application design such as choosing a good parallel algorithm, (2) compiler directive selections such as automatic data mapping, (3) general performance debugging, and (4) experimentation with different systems and run-time parameters. The framework has been implemented in the context of the HPF/Fortran90D compiler developed at Syracuse. Experiments show that the implementation was able to predict the execution times of a set of Fortran90D programs and program kernels on the iPSC/860 within 20% of their measured values, assuming that the array sizes, loop bounds, and branch conditions are known or provided at compile time.

The precision of the HPF/Fortran90D performance framework is very impressive. One main reason for this is that the the HPF/Fortran90D compiler only exploits explicit FORALL parallelism and generates loosely synchronous SPMD node programs for such programs. The execution of a loosely synchronous program can be described as a sequence of computations and communication phases. Although this fact made the estimation process much easier than for programs that can only be represented by a more general task graph, estimating the cache performance of the iPSC/860 remains a difficult problem. Unfortunately, the authors do not give a detailed description of their approach to modeling the memory hierarchy.

Using off-line benchmarking routines to estimate the system performance in terms of basic computation and communication operations is not a new idea [BFKK91]. What is new is the representation of the system characteristics in an abstract hierarchy. This representation should allow the performance framework to pick the right level of system and application abstraction for different performance prediction goals. For instance, choosing a good parallel algorithm or choosing a good data mapping may have different performance estimation precision requirements. The performance framework may be able to support both, each on its most efficient level of abstraction. The paper does not contain a validation of the different abstraction levels.

Chapter 4

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 uses 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.

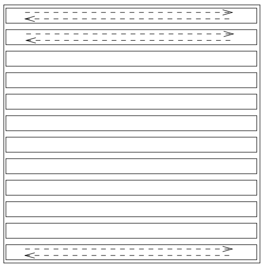
In the remainder of this section each of the four steps is discussed in more detail. A small program kernel will serve as an example to illustrate each of the four steps.

4.1 ADI Example Program

The following example illustrates the framework for automatic data layout. Figure 4.1 shows an Alternating Direction Implicit (ADI) integration kernel. ADI integration is a technique frequently used to solve partial differential equations (PDEs).

The execution of the ADI integration kernel consists of a repeated sequence of forward and backward sweeps along rows, followed by downward and upward sweeps along columns. For the sweeps along the rows, a row layout has the best performance. The same holds for a column layout for the column sweeps. Transposing the arrays between the all row and all column sweeps eliminates communication within the sweeps. In contrast, choosing the same data layout for both, row and column sweeps will avoid communication between the sweeps but will make communication necessary either in the row or column sweeps. The best data layout choice will depend on the

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

// READ (c, a, b)

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

   DO j = 2, N
      DO i = 1, N
         c(i, j) = c(i, j) - c(i, j - 1) * a(i, j) / b(i, j - 1)
         b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i, j - 1)
      ENDDO
   ENDDO

   DO i = 1, N
      c(i, N) = c(i, N) / b(i, N)
   ENDDO

   DO j = N - 1, 1, -1
      DO i = 2, N
         c(i, j) = ( c(i, j) - a(i, j + 1) * c(i, j + 1) ) / b(i, j)
      ENDDO
   ENDDO
```



```
   // Downward and upward sweeps along columns

   DO j = 1, N
      DO i = 2, N
         c(i, j) = c(i, j) - c(i - 1, j) * a(i, j) / b(i - 1, j)
         b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i - 1, j)
      ENDDO
   ENDDO

   DO j = 1, N
      c(N, j) = c(N, j) / b(N, j)
   ENDDO

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

ENDDO

// WRITE (c, b)
```

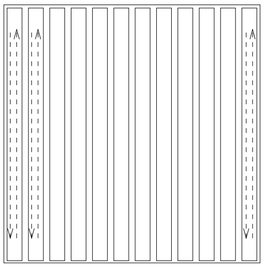


Figure 4.1 ADI integration kernel with computation illustration

speed of the communication hardware and software of the target distributed-memory machine, and the ability of the compiler to exploit pipelined parallelism efficiently. In addition, the performance characteristics of the underlying I/O system has to be considered since the program performs read and write disk accesses. Finally, the actual size N of the arrays and the number of available processors may influence the data layout choice. Figure 4.2 shows two different data layout specifications that may be generated by our automatic data tool assistant tool. The left hand side shows a static, column-wise data layout. The right hand side depicts a dynamic data layout where transpose operations will be performed between the row and column sweeps.

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

// Static column-wise layout
!HPF$ TEMPLATE X(N, N)
!HPF$ ALIGN c(i, j), a(i, j), b(i, j) WITH X(i, j)
!HPF$ DISTRIBUTE X(*, BLOCK)

      . . .
DO iter = 1, max
   // Forward and backward sweeps along rows

         . . .

   // Downward and upward sweeps along columns

         . . .

ENDDO
      . . .
```

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

// Dynamic row and column-wise layout
!HPF$ TEMPLATE X(N, N)
!HPF$ DYNAMIC X
!HPF$ ALIGN c(i, j), a(i, j), b(i, j) WITH X(i, j)
!HPF$ DISTRIBUTE X(*, BLOCK)
      . . .
DO iter = 1, max
   // Forward and backward sweeps along rows
   !HPF$ REDISTRIBUTE X(BLOCK, *)
         . . .
   // Downward and upward sweeps along columns
   !HPF$ REDISTRIBUTE X(*, BLOCK)
         . . .
ENDDO
      . . .
```

Figure 4.2 Example output of the proposed framework for automatic data layout for the ADI integration kernel. The left hand side shows a static column-wise data layout, and the right hand side shows a dynamic layout that performs transposes between the sweeps along rows and columns.

4.2 Program Partitioning

The first step partitions the program into code segments, called program *phases*. In our framework, data remapping is allowed only between phases. A good phase definition tries to identify operations on arrays between which remappings may be profitable. In languages such as Fortran 90, array statements, *FORALL* statements, and *DO* loops can be used as a basis for a phase definition. The following phase definition worked well on all programs in our test suite of Fortran 77 programs.

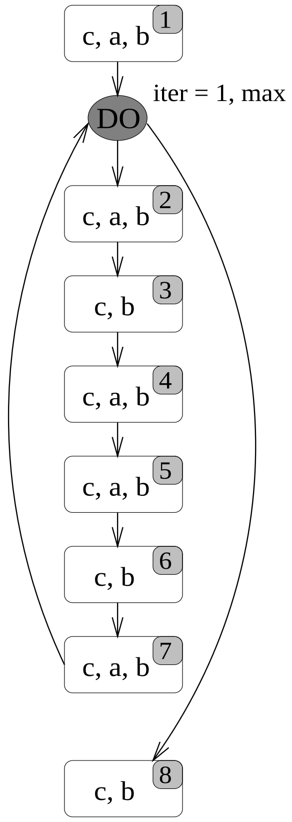
> **Definition 4.1** A *phase* is the outermost loop in a loop nest such that the loop defines an induction variable that occurs in a subscript expression of an array reference in the loop body.

This operational definition does not allow the overlapping or nesting of phases. Other strategies for identifying program phases are a topic of future 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 thesis.

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, called a *phase node*. Structured control flow between phases such as loops and branches are represented by special nodes in the PCFG. 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.

Figure 4.3 shows the partition of our ADI kernel example into phases. The resulting PCFG has eight phase nodes. Each phase node is labeled with the set of arrays referenced in the corresponding phase.

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

// READ (c, a, b)                                                    1

DO   iter = 1, max
    // Forward and backward sweeps along rows
    DO j = 2, N                                                      2
        DO i = 1, N
            c(i, j) = c(i, j) - c(i, j - 1) * a(i, j) / b(i, j - 1)
            b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i, j - 1)
        ENDDO
    ENDDO

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

    DO j = N - 1, 1, -1                                              4
        DO i = 2, N
            c(i, j) = ( c(i, j) - a(i, j + 1) * c(i, j + 1) ) / b(i, j)
        ENDDO
    ENDDO
    // Downward and upward sweeps along columns
    DO j = 1, N                                                      5
        DO i = 2, N
            c(i, j) = c(i, j) - c(i - 1, j) * a(i, j) / b(i - 1, j)
            b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i - 1, j)
        ENDDO
    ENDDO

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

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

ENDDO

// WRITE (c, b)                                                      8
```



PCFG

Figure 4.3 Result of the program partitioning step

4.3 Layout Search Spaces Construction

The second step of our 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, typically 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. In addition, explicit search spaces provide a natural basis for user interaction. The user can browse through the search spaces and influence the selection process by adding or deleting candidate data layouts.

Our 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 based on 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.

4.3.1 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 4.3.2.

There are two types of alignment preferences, namely *inter–dimensional* and *intra–dimensional* alignment [LC90a, KLS90, CGST93]. The current framework does not perform intra-dimensional alignment analysis, i.e., assumes canonical offset and stride alignments. The discussion of the basic operations will be restricted to inter-dimensional alignment preferences.

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 [LC90a]. 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.

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 [LC90a]. Instead of using a heuristic, the current framework formulates the inter-dimensional alignment problem as an efficient 0–1 integer programming problem. Section 2.2 constrains an introduction to integer programming and 0–1 integer programming.

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 provided in the appendix.

> **Definition 4.2** 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 4.3** 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 $s_{01} : X \to \{0, 1\}$ 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, $a\$b^{ik}_{jk} \in X$, $1 \leq k \leq d$. In the final solution, the switch $a\$b^{ik}_{jk}$ 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 $\sum_{k=1}^{d} a_{ik} = 1 \quad (type1)$ 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, $1 \leq k \leq d$: $\sum_{i=1}^{dim(a)} a_{ik} \leq 1 \quad (type2)$. 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".

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 $\sum_{b_j \in SRC(b,a_i)} \quad b\$a^{jk}_{ik} \leq a_{ik}$ 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 $\sum_{c_j \in SINK(a_i,c)} \; a\$c^{ik}_{jk} \leq a_{ik}$ 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 4.4 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 4.4 is not actually generated during the translation process, but illustrates our 0–1 formulation.

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

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

The switch $a\$b^{ik}_{jk}$ 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.

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 represents 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 4.5 shows an example lattice for two two-dimensional arrays.

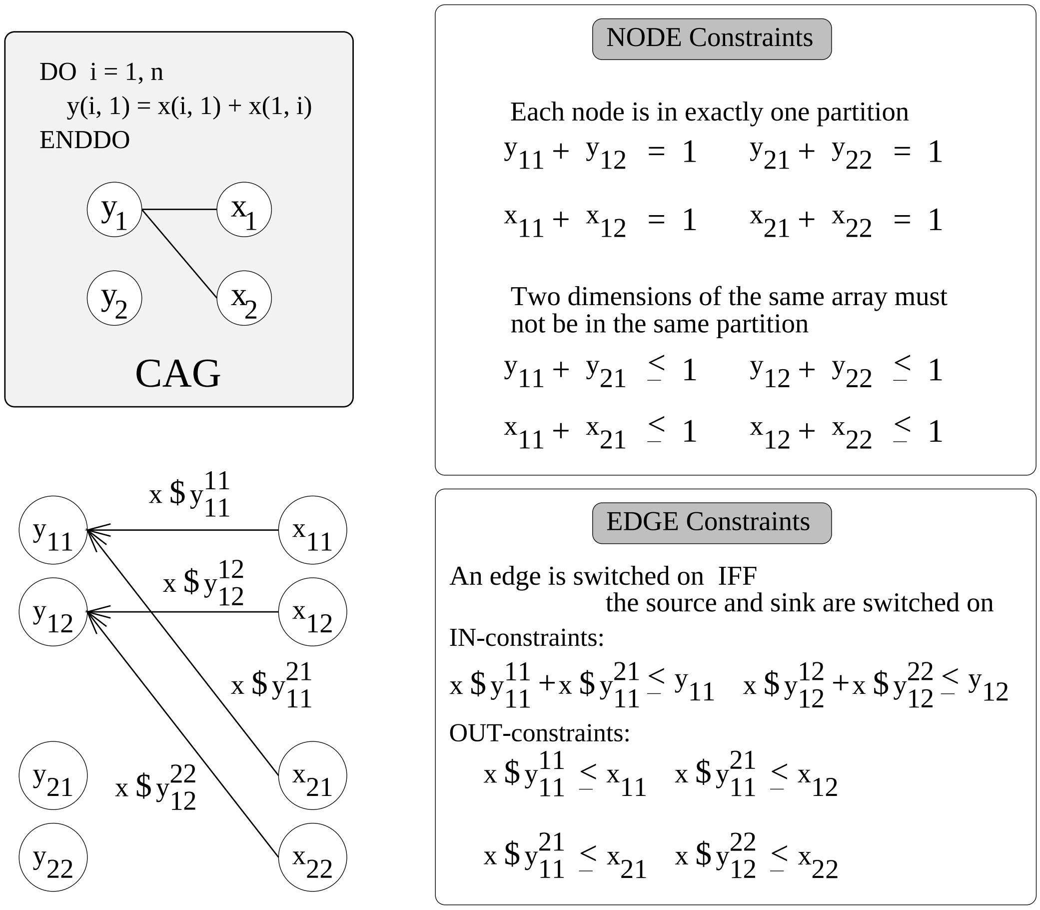


Figure 4.4 Alignment conflict resolution of an example CAG as a 0–1 integer programming problem

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.

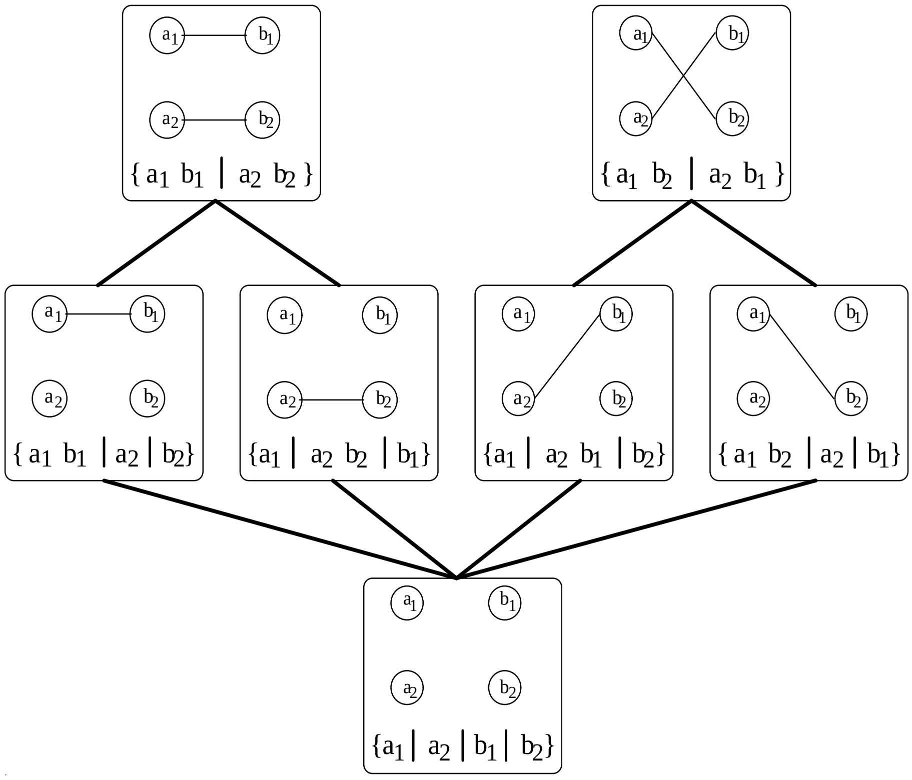


Figure 4.5 Inter-dimensional alignment information lattice for two arrays a and b. Both arrays have two dimensions. Each conflict-free CAG is shown with its corresponding node partitioning. The bottom element of the lattice is the CAG without edges, i.e., the partitioning $\{a_1 \; |a_2 \; |b_1 \; |b_2\}$.

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.

A meet operation of two CAGs determines the CAG that represents the weakest alignment information that contains the alignment information common to both operand CAGs. In other words, $CAG_1 \sqcap CAG_2$ determines the intersection of the alignment information in the two operand CAGs. Since intersecting two conflict-free alignments cannot introduce a conflict, the result of the meet operation is always a unique, conflict-free CAG.

A join operation of two CAGs determines the CAG that represents the weakest alignment information that contains the alignment information in both operand CAGs. Note that such a CAG may not exist since the two operand CAGs may have conflicting alignment information. In other words, $CAG_1 \sqcup CAG_2$ computes the union of the alignment information in CAG_1 and CAG_2 if the union does not have conflicts. If a conflict-free union exists, then the resulting CAG is unique.

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 a ligned array dimensions to the dimensions of the program template. For a d-dimensional program template and a conflict-free CAG with d connected components, 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.

4.3.2 Heuristics for Alignment Analysis

The goal of alignment analysis is to determine a promising set of candidate alignments for each phase. In most cases, an exhaustive alignment search space will not be possible. Heuristics are needed that restrict the search space sizes and at the same time avoid the elimination of promising alignment candidates. In this section, we will give examples of such heuristics. The heuristics use the basic operations of our framework.

CAG Construction

The edge weights in the CAG reflect the relative importance of alignment preferences. The basic operation of alignment conflict resolution assumes that edge weights reflect the cost that will occur if the associated alignment preference cannot be satisfied. Therefore, conflict resolution tries to minimize the sum of edge weights that cannot be satisfied (see Section 4.3.1). Typically, edge weights represent communication costs in some performance model. The choice of a performance model is compiler and machine dependent.

Performance models at this level typically do not consider array distributions. If a distribution maps misaligned dimensions to the same processors, no communication is needed.

An edge in the CAG may represent multiple occurrences of the same alignment preference in the program. If the compiler allows the caching of communicated values, only a single communication may be needed to satisfy all occurrences of the same alignment preference. In this case, the edge weight will be the same for one or more occurrences of the alignment preference. If the caching of the communicated value is not possible, or the compiler does not support value caching, each communication has to be accounted for separately, resulting in an edge weight that is the sum of all individual communication costs of each occurrence.

Our sample heuristic to determine the CAG edge weights assumes an advanced compilation system that performs communication value caching and uses the owner-computes rule for computation mapping [Tse93]. The target machine is assumed to be a MIMD target architecture. Our sample performance model is pessimistic since it assumes that unsatisfied alignment preferences will always lead to communication.

In contrast to the original definition of a CAG by Li and Chen [LC90a], the heuristic uses a directed CAG while computing the edge weights. The edge directions keep track of the flow of values due to the owner-computes rule. If the same alignment preference is encountered, the heuristic 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 edges in the CAG. Once edge weights have been determined, edge directions in the resulting CAG are removed.

Alignment Search Space Construction

There are many possible heuristics to construct the alignment search spaces for each phase. In this section, we will discuss one heuristic in more detail. The overall outline of our heuristic is shown in Figure 4.6.

The alignment search spaces are initialized with the undirected, weighted CAGs of their phases. If a CAG has inter-dimensional alignment conflicts, the conflicts are 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 merged, conflict-free CAG. The current prototype uses a greedy strategy to determine the single phase CAG to be merged next. The algorithm visits the phases, i.e., the nodes in the PCFG in reverse postorder (rPOSTORDER) [Hec77], and merges 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 greedy merging procedure is resumed based on this new class. The partitioning algorithm terminates after all phases have been visited. The algorithm is shown in Figure 4.7.

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. The heuristic is shown in Figure 4.8. 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.

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 of alignment information, the imported alignment candidate is only inserted into the search space if its alignment information is not weaker or equal to any alignment information already in the search space. The sizes of the final search spaces can be further reduced if alignment candidates are joined that do not contain conflicting alignment information.

Algorithm **ALIGN**

Input: PCFG G=(N, E), with phase nodes, DO nodes, and IF nodes
Output: Alignment candidate search spaces for each phase node in the PCFG

```
for each phase node P_i in the PCFG do
    local_i = create_CAGs (P_i) // builds the set of CAGs for phase P_i
    if has_conflict? (local_i) then
        local_i = resolve_conflicts (local_i) // conflict-free CAGs for single phase
    endif
endforeach

// Heuristic to determine node partitioning of phase nodes in conflict-free
// CAG (See Figure 4.7).
call PART_PCFG (node partitioning)

// Heuristic to generate candidate alignment search spaces for each phase.
// Phases in the same partition are treated as a unit since they do not
// contain conflicts (See Figure 4.8).
call IMPORT_ALIGNMENTS (node partitioning)

// Heuristic to generate orientations of relative candidate alignments
// (See Figure 4.9).
call ORIENTATION_SELECTION (node partitioning)

// Convert each candidate alignment of a partition into a candidate
// alignment of each phase in the partition.
call GENERATE_PHASE_ALIGNMENTS (node partitioning)
```

Figure 4.6 Heuristic (driver routine) for inter-dimensional alignment analysis

Algorithm **PART_PCFG (node partitioning)**

Input: PCFG G=(N, E), and for each phase the set of conflict-free CAGs that represent the inter-dimensional alignment requirements of the phase

Output: Partitioning of the phase nodes in the PCFG such that there are no interdimensional alignment conflicts between CAGs of phases in the same partition.

```
// Determine conflict-free partitions of phase nodes in a greedy fashion.
// The minimal number of conflict-free partitions is not guaranteed.

// Determine an rPOSTORDER of the phase nodes in the PCFG such that
// indices of nodes inside a loop are smaller than the indices of the
// nodes following that loop [KZBG88]

node partitioning = ∅;  CURRENT_PART = ∅
i = 1
while i ≤ |N| do
    CURRENT_PART = {i};  current_part_CAGs = local_i;  conflict-free = true
    while conflict-free and i + 1 ≤ |N| do
        current_part_CAGs = merge (current_part_CAGs, local_{i+1})
        if has_conflicts? (current_part_CAGs) then
            conflict-free = false
        else
            CURRENT_PART = CURRENT_PART ∪ {i + 1}
        endif
        i = i + 1
    endwhile
    node partitioning = node partitioning ∪ { CURRENT_PART }
endwhile
```

Figure 4.7 Heuristic for partitioning of phases into conflict-free sets.

Algorithm IMPORT_ALIGNMENTS (node partitioning)

Input: Partitioning of phase nodes into sets of conflict-free phases. Phase nodes in the same partition are treated as a unit, i.e., represented by the same CAG.

Output: Candidate alignment search spaces for each partition in the input node partitioning.

```
increment = 100 // weight increment for imported CAG
// Initialization of search spaces
foreach partition in node partitioning do
  build the conflict-free CAG representing the partition
  initialize the alignment search space of the partition with its CAG
endforeach
foreach pair (part_i, part_j) of distinct partitions in node partitioning do
  //    import alignment scheme of partition i into partition j
  weighted_CAG =
          merge ( increase_weights (CAG (part_i), increment), CAG (part_j))
  if has_conflicts? (weighted_CAG) then
    weighted_CAG = resolve_conflicts (weighted_CAG)
  endif
  foreach candidate_alignment_CAG in alignment search space of part_j do
    compare_result =
          compare_alignments(weighted_CAG, candidate_alignment_CAG)
    case  compare_result of
      WEAKER or EQUAL: exit innermost foreach
      STRONGER:
        replace candidate_alignment_CAG by weighted_CAG in search space
        exit innermost foreach
      NEITHER:
        if comparison with last candidate_alignment_CAG in search space then
          add weighted_CAG to search space
        endif
    endcase
  endforeach
  //    import alignment scheme of partition j into partition i
  weighted_CAG =
          merge ( CAG (part_i), increase_weights (CAG (part_j), increment))
  ...analogue to above import process ...
endforeach
```

Figure 4.8 Heuristic for importing candidate alignments.

<u>Algorithm</u> ORIENTATION_SELECTION (node partitioning)

Input: Node partitioning with relative alignment search spaces
Output: Set of orientation for each candidate alignment in
the relative alignment search spaces of phase partitions

```
// Determine common relative alignment information in each search space
foreach partition part_i do
  meet_i = ⊓ candidate alignments in search space of part_i
endforeach

foreach pair (part_i, part_j) of distinct partitions in node partitioning do
  // Determine partial orientations for each pair of partitions
  meet = meet_i ⊓ meet_j
  if is_bottom? (meet) then
    skip // no common relative alignment preferences
  else
    determine orientation for meet
    extend orientation of meet to all candidate alignments in the
      search spaces of part_i and part_j
    insert resulting orientation into the set of orientations for
      each candidate alignment in part_i and part_j
  endif
  foreach candidate alignment do
    if set of orientations is empty then
      determine canonical orientation for the candidate alignment
    endif
  endforeach
endforeach
```

Figure 4.9 Heuristic to determine sets of orientations

Orientation Selection

In the presence of alignment conflicts, the relative alignment heuristic will construct multiple candidate alignment search spaces, one for each phase partition. Figure 4.9 shows a sample heuristic that generates a set of orientations for each candidate alignment in the alignment search spaces.

The sample heuristic identifies common alignment preferences in each search space and across search spaces. Array dimensions that are part of the common alignment preferences are mapped to the same template dimensions. If the partial orientations do not induce unique total orientations for each candidate alignment, a canonical orientation scheme is used to map the remaining array dimensions to template dimensions.

As the result of the sample heuristic, the final candidate alignment search spaces have an entry for each relative alignment and each orientation in its computed set of orientations. Other heuristics may use a greedy algorithm, for instance, to propagate orientations from the most frequently executed phases to less frequently executed phases [AL93].

4.3.3 Distribution Analysis

Distribution analysis is performed after alignment analysis. A candidate distribution maps each template dimension either by `BLOCK` or `CYCLIC(i)` onto the target architecture, or replicates or localizes the template dimension. If a dimension is distributed across processors, 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.

Exhaustive Search Space for `BLOCK` Distributions

Although the search space of possible distribution candidates may be large in general, there are cases where using an exhaustive distribution search space is feasible. In this section we will discuss the sizes of exhaustive distribution search spaces for `BLOCK` distributions.

Lemma 4.1 Let d denote the number of dimensions of the program template. Assuming that at least one dimension has to be partitioned, the number of possible dimensional partitionings, *part*, is

$$part = \sum_{i=1}^{d} \binom{d}{i} = 2^d - 1$$

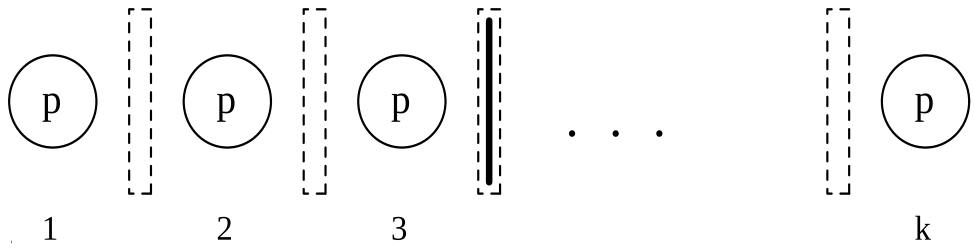


Figure 4.10 Possible partitions of k objects representing $p * p * p * ... * p = p^k$ processors

Proof The index i keeps track of the number of dimensions to be partitioned. For any given i, $1 \leq i \leq d$, there are

$$\binom{d}{i}$$

choices of i dimensions out of d dimensions. □

Lemma 4.2 Let *procs* denote the number of processors used. Assume that the number of processors is a power of some prime number p, i.e., $procs = p^k$. Assuming that at least one dimension is partitioned, the number of distribution schemes for a d-dimensional program template, *size*, is

$$size = \sum_{i=1}^{d} \binom{d}{i} * \binom{k-1}{i-1} = \binom{k+d-1}{d-1}$$

Proof The index i keeps track of the number of dimensions to be partitioned. There are $k-1$ possible choices to place a partition for p^k processors to create two groups of processors as shown in Figure 4.10. For any given i, $1 \leq i \leq d$, there are

$$\binom{k-1}{i-1}$$

choices of $i-1$ slots out of $k-1$ possible slots to divide the number of processors into i distinct, non-empty groups, each assigned to one distributed dimension. Note that the number of choices is 0 if $i > k$. The resulting values for *size* form a Pascal's triangle. □

The following table gives the values of *size* for different numbers of processors used and program templates of up to seven dimensions. The number of processors is assumed to be a power of two, i.e., $procs = 2^k$.

#procs	#dimensions						
	1	2	3	4	5	6	7
4	1	3	6	10	15	21	28
8	1	4	10	20	35	56	84
16	1	5	15	35	70	126	210
32	1	6	21	56	126	252	462
64	1	7	28	84	210	462	924
128	1	8	36	120	330	792	1716
256	1	9	45	165	495	1287	3003
512	1	10	55	220	715	2002	5005
1024	1	11	66	286	1001	3003	8008

For example, if the program template has three dimensions and there are 64 available processors, the exhaustive distribution search space has 28 distribution candidates. Depending on the number of alignment candidates in the alignment search spaces, an exhaustive distribution search space is feasible. For a fixed number of template dimensions, the size of the exhaustive candidate distribution search space is polynomial as shown in Lemma 4.3.

Lemma 4.3 Let d be constant. Then $size = \Theta(k^{d-1})$ *

Proof Define constant c as $d-1$.

$$\binom{k+c}{c} = \frac{(k+c)!}{c!\ k!} = \frac{1}{c!} * \prod_{i=1}^{c}(k+i) =: T$$

$$T \leq \frac{1}{c!} * (k+c)^c = O(k^c)$$

$$T \geq \frac{1}{c!} * k^c = \Omega(k^c)$$

Substituting $d-1$ for c in $\Omega(k^c) \leq T \leq O(k^c)$ yields the claimed result. □

The presented results are also applicable for the case where the number of processors is a power of some integer value i, and i is considered a unit of processors that must not be further partitioned. In other words, the number of processors can be a power of some integer value that must not be factorized itself.

Similar results can be obtained if the number of processors is a power of some prime number multiplied by another, distinct prime number. The analysis for this case can be found in the appendix in Section A.3.

Exhaustive Search Space for `BLOCK` and `CYCLIC(1)` Distributions

In this section we will discuss the sizes of exhaustive distribution search spaces for `BLOCK` and `CYCLIC(1)` distributions.

Lemma 4.4 Let *procs* denote the number of processors used. Assume that the number of processors is a power of some prime number p, i.e., $procs = p^k$. Further assume that at least one dimension is partitioned and each partitioned dimension can be either distributed by `BLOCK` or `CYCLIC(1)`. Then the number of distribution schemes for a d-dimensional program template, *size*, is

$$size = \sum_{i=1}^{d} 2^i * \binom{d}{i} * \binom{k-1}{i-1}$$

*Θ, O, and Ω denote asymptotically tight bounds, asymptotically upper bound, and asymptotically lower bound, respectively [CLR90].

Proof The index i keeps track of the number of dimensions to be partitioned. For any given i, $1 \leq i \leq d$, there are

$$\begin{pmatrix} k-1 \\ i-1 \end{pmatrix}$$

choices to divide the number of processors into i distinct, non-empty groups, each assigned to one distributed dimension. Each group can be either partitioned using a `BLOCK` or `CYCLIC(1)` distribution. There are 2^i possible combinations of `BLOCK` and `CYCLIC(1)` distributions for i groups. □

The following table gives the values of *size* for different numbers of processors used and program templates of up to seven dimensions. The number of processors is assumed to be a power of two, i.e., $procs = 2^k$.

#procs	#dimensions						
	1	2	3	4	5	6	7
4	2	8	18	32	50	72	98
8	2	12	38	88	170	292	462
16	2	16	66	192	450	912	1666
32	2	20	102	360	1002	2364	4942
64	2	24	146	608	1970	5336	12642
128	2	28	198	952	3530	10836	28814
256	2	32	258	1408	5890	20256	59906
512	2	36	326	1992	9290	35436	115598
1024	2	40	402	2720	14002	58728	209762

4.3.4 Heuristics for Distribution Analysis

A dimension of the program template can either be distributed by `BLOCK` or `CYCLIC(i)`, or can be localized. `CYCLIC(i)` distributions are mainly used to improve load balancing. A cyclic distribution may require more communication than a corresponding block distribution due to the difference in their induced surface to volume ratios. However, in the cases where load balancing is an issue, the improved load balancing due to a cyclic distribution typically justifies the additional communication costs.

The blocking size `i` in the `CYCLIC(i)` distribution allows the specification of a cyclic distribution that represents a tradeoff between communication and load balancing.

The heuristics for the construction of candidate distribution search spaces can be roughly divided into two classes, *exhaustive* and *constructive*. Exhaustive heuristics try to 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. A performance model is needed to distinguish the qualities of possible candidate layouts. Each layout consists of a candidate alignment and a possible candidate distribution.

In this section we will discuss an exhaustive sample heuristic. The heuristic does not consider replication or cyclic distributions with a blocking size other than 1. We will abbreviate `CYCLIC(1)` as `CYCLIC`. Heuristics to determine blocking sizes for block-cyclic distributions are beyond the scope of this thesis.

Our sample heuristic classifies each dimension of the program template as *cyclic* or *blocked*. A dimension of the program template is *cyclic* if there is a phase in the program that performs a triangular computation along the dimension. If such a phase does not exist, the template dimension is considered *blocked*.

In order to classify the dimensions of the program template, the alignment candidates in the alignment search spaces of every phase have to be visited. In the example shown in Figure 4.11, the alignment search space is assumed to have only a single entry, namely the canonical alignment.

In the example in Figure 4.11, the first and second dimensions of the program template are classified as *cyclic*, and the third dimension is considered *blocked* due to the shape of array A's region accessed in the triangular loops i and j.

Depending on the system or user specified maximal number of candidate layouts for any search space, and the precision of the performance estimator to evaluate candidate data layouts, the heuristic selects uniformly spread candidate distributions in the exhaustive distribution search space.

If a dimension has been classified as *cyclic*, `CYCLIC` and `BLOCK` distributions for the dimension are inserted into the search spaces. The size of the exhaustive distribution search space for the case where all dimensions are classified as *cyclic* is discussed in Section 4.3.3. The performance estimator of the system determines an upper bound on the granularity of the search space. For instance, if the estimator is not able

```
TEMPLATE  PROG_TEMPLATE(N, N, N)
ALIGN  A(I, J, K)  WITH  PROG_TEMPLATE(I, J, K)

      do  10  k = 1, N
        do  10  j = 1, N
          do  10  i = j, N
10            A(i, j, k) = . . .
```

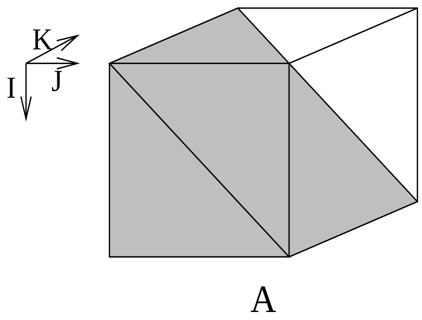


Figure 4.11 Example computation phase with triangular iteration space and canonical alignment

to distinguish two similar candidate distributions, then only a single representative distribution should be added to the search space.

4.3.5 Example

Figure 4.12 shows the final candidate data layout search spaces for our example ADI kernel introduced in Section 4.1. The example kernel has a two-dimensional program template of size N in each dimension. Since there are no inter-dimensional alignment conflicts between relative alignment preferences of the phases, all phases are merged into the same phase partition. As a result, the same canonical orientation is chosen for all alignment candidates and each alignment search space contains only a single candidate.

The heuristic for distribution analysis described in Section 4.3.4 classifies both dimensions of the program template as *blocked* since the program does not contain any triangular loops. Figure 4.12 depicts the exhaustive candidate layout search spaces for a target machine with 8 processors. Every search space has four candidate layouts. Each candidate layout only specifies the data mapping of arrays referenced in its phase. For example, the candidate layout search spaces of phases 3, 6, and 8 do not contain mapping information for array a.

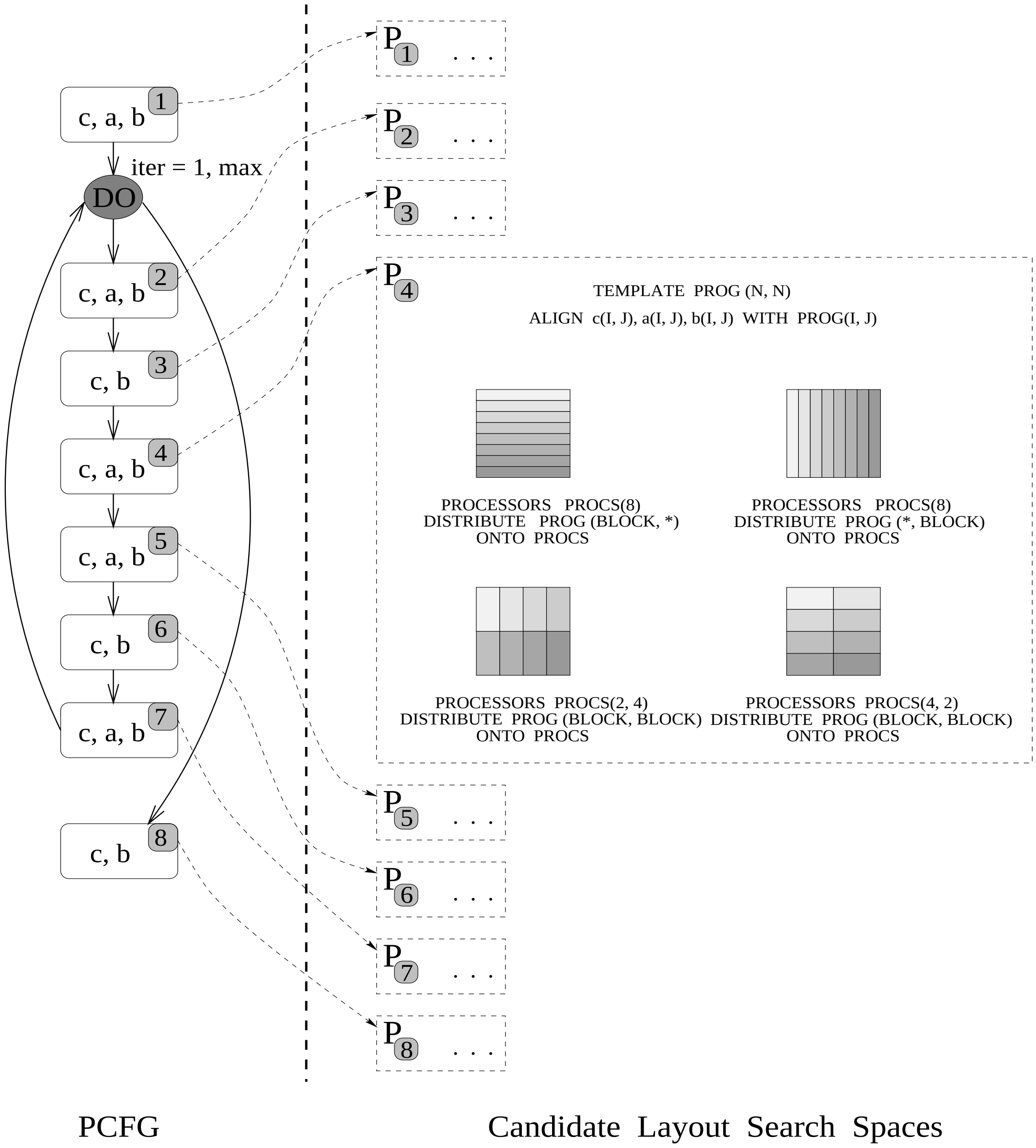


Figure 4.12 Exhaustive candidate data layout search spaces for 8 processors

4.4 Performance Estimation

After the generation of the data layout search spaces each candidate data layout is evaluated in terms of its expected execution time for its phase. In addition to the phase execution times, execution time estimates are needed for possible remappings between candidate data layouts. The static performance estimator does not need to predict the *absolute* performance of a given data layout to assist automatic data layout, even though it is desirable. Instead, the estimator should accurately predict the performance *relative* to other data layouts.

The performance of a candidate layout for a phase depends on the compiler that is used to generate the executable for the target architecture, and on the performance characteristics of the target architecture itself. Therefore, a source-level performance estimator is needed that can target different compilers and architectures. One of the main contributions of this thesis is a new approach to source-level performance estimation. The approach is based on so-called *training sets* [BFKK91, HKK+91] and efficient compiler, execution, and machine models.

4.4.1 Training Sets

A training set is a collection of kernel routines that measure the cost of various communication and computation patterns. The results of executing the training sets on a parallel machine are summarized and processed for use by the performance estimator. By utilizing training sets, the performance estimator achieves both accuracy and portability across different target compilers and machine architectures.

There are two types of training sets, *compiler-level* and *machine-level* training sets. A compiler-level training set consists of program kernels such as stencil computations and matrix multiplication written in Fortran. The training set is converted into message-passing Fortran using the target compiler and executed on the target machine for different data layouts, numbers of processors, and array sizes. Estimating the performance of a phase under a given data layout requires matching the computation in the phase with kernels from the training set. In contrast, a machine-level training set is written in message-passing Fortran. A machine-level 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 message-passing program, the performance estimator can simply look up results for each computation and communication pattern encountered.

In summary, instead of using a general theoretical performance model, training sets probe the underlying system for its performance characteristics. The training set method provides a natural way to respond to changes in the target compiler as well as the target machine — simply rerun the training sets with the new compiler on the new target machine to initialize a new performance estimator. In contrast, if a general theoretical model is used, changes to the target hardware/software system will nearly always require changes to the performance model itself, a tedious task at best.

Although the use of training sets simplifies the task of performance prediction significantly, its complexity now lies in the design of the training sets themselves. Training sets should cover all computation and data movement patterns that will allow the performance estimator to extract the performance characteristics of the target hardware/software system.

4.4.2 Performance Estimation Framework

If a phase matches a program kernel in a compiler-level training set, a single table lookup provides the estimated performance for the candidate data layout. Since it is not possible to incorporate all possible computation patterns in a 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 performance estimator relies on its *compiler model*, *execution model*, and *machine model*.

The compiler model and machine model generate information that is used by the execution model to determine the overall performance of a code fragment at the Fortran source level. The models have to be mutually calibrated in order to allow efficient and accurate performance estimation. An overview of the compiler model, execution model, and machine model is given below. A detailed discussion of the models implemented in our prototype layout assistant tool can be found in Section 5.2.1.

Compiler Model

The compiler model determines 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

Message Vectorization

Input: A candidate data layout for a phase, the phase, and the dependence graph of the phase

Output: A set of pairs (q,λ), indicating that processor q must send the section λ of data that it owns to processor k, and the loop level at which the communication occurs

for each statement **do**
  Let def(X) be the section of the LHS array X that is written and owned by the kth processor
  for each RHS array reference Y **do**
    Let use(Y) be the section of the RHS array Y that is needed to compute the elements of def(X)
    Define `commlevel` of a true dependence to be:
    $$\begin{cases} \text{level of dependence,} & \text{if loop-carried} \\ \text{common nesting level of src and sink of dependence,} & \text{if loop-independent} \end{cases}$$
    Let `lmax` = max(`commlevels` of all true dependences with Y as sink reference)
    Let $\Uparrow$use(Y) be the section use(Y) "translated" to the level `lmax`
    Let Y\$(q) denote the section of array Y owned by processor q
    Then, the set of all (q,λ) pairs is given by
      $\{(q,\lambda) \mid q \neq k \text{ , } \lambda = Y\$(q) \cap \Uparrow\text{use}(Y) \text{ , } \lambda \neq \phi\}$,
     with the communications occurring at loop level `lmax`
  endfor
endfor

Figure 4.13 Algorithm to determine the communication for processor k for a candidate data layout and its phase that may be performed by the target compiler. The compilation process needs to be simulated for performance purposes only. Special cases that have only 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 may ignore code that is generated for boundary processors in parallel loops.

Since candidate data layout search spaces may be large, the efficient simulation of the compilation process is crucial for the overall efficiency of our framework for automatic data layout. A central problem during compilation is the placement and optimization of communication. Early compilation approaches first inserted so-called *run-time resolution* code for each array reference into the program. When executed, this code performs the necessary communication for a single array reference. Based on the run-time resolution program, the compiler tried to optimize the communication through source-level transformations [ZBG88, CK88, RP89]. This early compilation method is hard to simulate efficiently.

We have developed a new compilation approach that allows the analysis and optimization of communication at an abstract level without source-level transformations. The approach is based on data access summary information such as data access descriptors (DAD [Bal90]) or regular section descriptors (RSD [Cal87]) to determine *what* data has to be communicated, and on the dependence graph to determine *when* the data can be communicated [AK87, Wol89]. Figure 4.13 shows the message vectorization algorithm that moves communication out of loop nests as far as possible [BFKK90]. A similar algorithm has been proposed by Gerndt [Ger90].

Tseng extended our basic compilation approach and showed that it is flexible enough to accommodate other communication optimizations such as message aggregation, message coalescing, and coarse-grain pipelining [Tse93]. He also showed that in practice message vectorization is a highly effective communication optimization.

Execution Model

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 pipelined execution of the loop. Communication outside of the phase may result in a loosely synchronous execution scheme [FJL+88]. In addition, special communication patterns may be recognized that represent global operations such as reductions. Based on the synchronization schemes and the costs for simple communication patterns and basic computations, the execution model determines the overall cost estimate for a candidate layout and its phase.

In a loosely synchronous program, all processors operate in a loose lockstep, consisting of alternating phases of parallel asynchronous computation and synchronous communication. The performance of the overall program can be estimated by predicting the cost of the node program that operates on the largest local data segment. This performance model assumes that the largest segment has the highest computation and communication costs and therefore represents the "critical path" in a loosely synchronous execution of the entire program. The computation and communication costs for the largest local data segment are just added up to determine the overall phase performance.

Performance prediction of a pipelined phase execution is more complicated than in the loosely synchronous case. Execution models that can estimate pipelines of different granularity have been discussed in the literature, for instance in [MCAK94, PSCB94]. The actual choice of a particular pipeline model will depend on the desired accuracy.

A detailed discussion of the execution model used in the prototype implementation of our layout assistant tool can be found in Section 5.2.1.

Machine Model

The actual costs of communication operations and basic computations for the target machine architecture are determined by a *machine model*. The machine model may be based on machine-level training sets or on other performance models, such as the models discussed by Gupta and Banerjee [GB92a] or Fahringer and Zima [FZ93]. Training sets of global communication operations can be used to estimate the cost of remappings between candidate data layouts.

A computation training set measures the execution times for arithmetic operations and intrinsic functions for different data types and memory access patterns. Memory access patterns try to capture the impact of the memory hierarchy on the execution time of the operation. For instance, an arithmetic operand may be in a register, in cache, or out of cache.

A communication training set for a basic communication pattern *comm* such as `shift`, `broadcast`, or `send/receive` typically generates a cost table that represents a cost function of the form

$$Cost_{comm} : (\#processors \times \#transmitted\_bytes) \rightarrow execution\_time$$

To improve the overall accuracy of performance prediction, additional parameters may be considered in the cost functions. For example, a cost function may also be parameterized with respect to the memory stride of the transmitted data.

A communication training set for a global data remapping operation such as `transpose` may be parameterized with respect to the source and destination data layout of the remapping operation in addition to the array size, the array data type, and the number of available processors.

Executing the machine-level training sets generates cost tables in so-called *raw data* files. These files have to be read and processed by the data layout assistant tool at tool invocation time. Since the amount of data in the raw data files may be large, a compact internal representation is desirable or even necessary. We have experimented with the chi-square fit method [PFTV88] to approximate cost functions represented by raw data files. If the communication cost function can be decomposed into piece-wise linear functions, the chi-square fit method can be used to approximate each of the linear functions [BFKK91]. Other approximation methods may represent only a subset of the data points from a raw data file in an internal cost table. Linear interpolation can be used for data points between defined entries in this cost table.

4.4.3 Experiments

To evaluate the effectiveness of our training set approach, we have implemented a performance estimator for loosely synchronous SPMD node programs. The performance estimator was implemented as part of the ParaScope interactive parallel programming environment [BKK+89].

Since the estimator takes programs with explicit communication as input, a compiler model is not necessary. Given a node program, the estimator's execution model computes the largest local segment for the specified problem size and number of available processors. The execution model performs a tree walk over the abstract syntax tree (AST [ASU86]) of the program. The program may only contain structured control flow. If a control-flow branch is visited, the user is queried for the

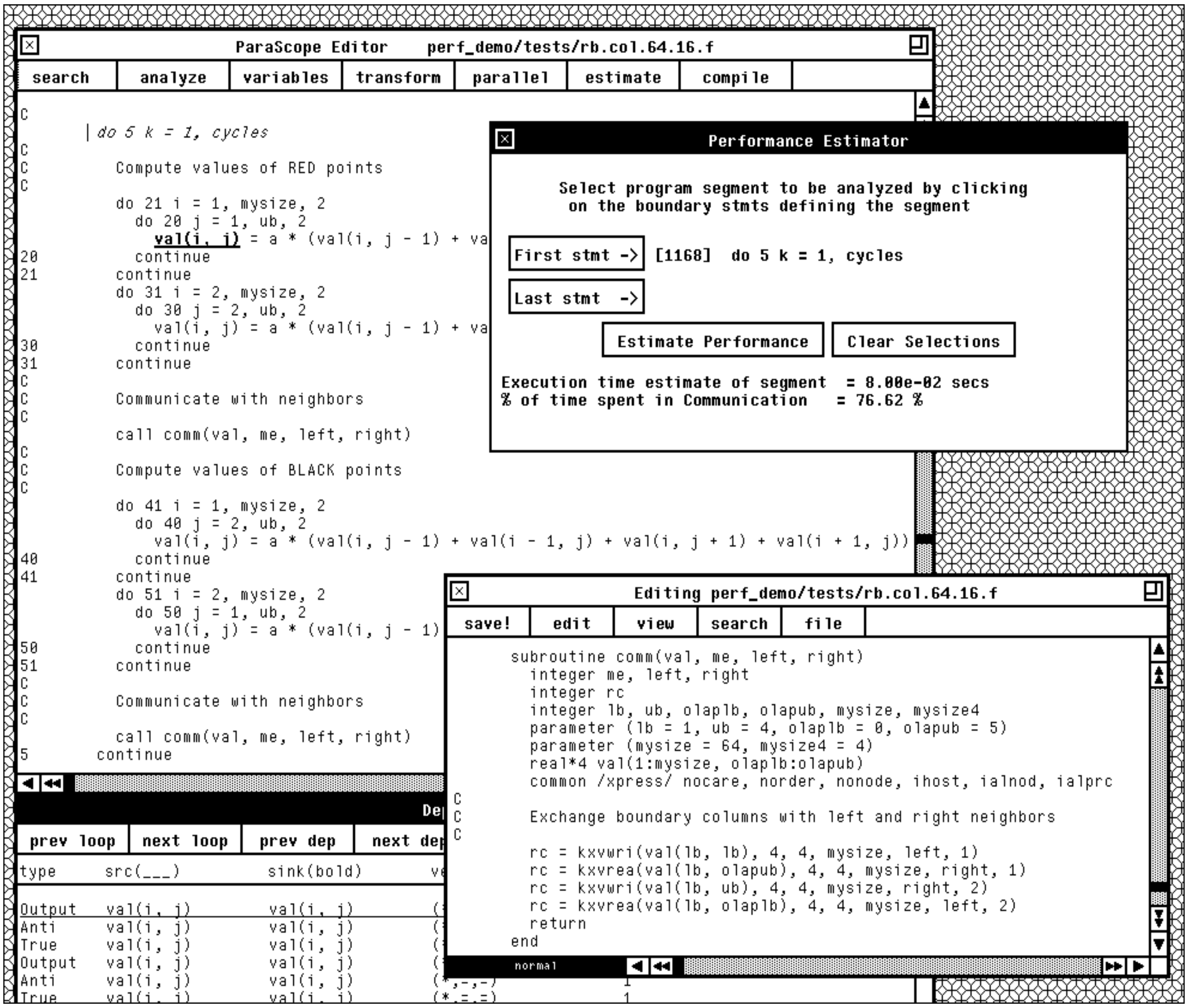


Figure 4.14 Screen snapshot of performance estimator applied to a point-wise red-black relaxation node program with column-wise data layout

branch probabilities. If a computation or communication pattern is encountered, the cost of the pattern is determined using the machine-level training set, and the result is added to the overall estimated execution time of the node program [BFKK91].

The estimator's machine-level training sets contain communication patterns that call EXPRESS routines [EXP89]. EXPRESS is a portable communication library similar to PVM [GBD+94] and MPI [GLS94]. The estimator predicts the performance of node programs with calls to EXPRESS communication routines. A snapshot of the estimator applied to a point-wise red-black relaxation code is shown in Figure 4.14.

The machine-level training set has proved quite precise for the NCUBE-1 and the iPSC/860, especially in predicting the relative performance of different data layouts. For the red-black relaxation code shown in Figure 4.14, the predicted execution times for a static column-wise data layout and a static two-dimensional block-wise data layout were within 10% of the actual execution times. For the example code, the best data layout choice depends on the problem size, data type, and number of processors used. For smaller problem sizes, the one-dimensional column-wise layout is more efficient due to the smaller number of messages. However, for bigger problem sizes, the two-dimensional block-wise layout is superior since less data has to be communicated and load balancing is better. For varying problem sizes, the "crossover points" of the two data layouts were determined with high accuracy [BFKK91, Kre93a].

It is important to note that the estimator was developed for training-set evaluation purposes only. The machine model in our prototype data layout assistant tool described in Section 5.2.1 uses different machine-level training sets and does not share any code with the estimator discussed in this section.

4.5 Data 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.

We will show in Section 4.5.3 that the data layout selection problem as formulated in the Section 4.5.1 is NP-complete. In Section 4.5.4 we will discuss a special case where the problem can be solved in polynomial time in the sizes of the data layout search spaces. Instead of relying on heuristics to approximate the optimal solution in the general case, the data layout selection problem is translated into a linear 0–1 integer programming problem and solved optimally. Different 0–1 integer programming formulations are discussed in Section 4.5.5.

4.5.1 Problem Formulation

The data layout selection problem is modeled as an optimization problem over the *data layout graph* (DLG). The DLG is a weighted, directed graph. The graph has one node for each candidate data layout. Edges represent possible remappings between candidate data layouts. Nodes and edges have weights representing the overall cost of each layout and remapping, respectively, in terms of execution time. The costs reflect the frequencies or probabilities of phase execution. The phase execution frequencies are provided in the PCFG.

The DLG construction assumes that only a single copy of an array can exist at any time during program execution, unless the array is replicated due to multiple ownership. This restriction can be relaxed by adding additional remapping edges. The placement of these remapping edges is a topic of future research. The discussed formulation of the data layout selection problem as an optimization problem over the DLG does not model the possible overlap of communication and computation between phases.

A data flow problem over the PCFG is solved to determine the data layouts that can reach each single phase. The proposed data flow problem is similar to the reaching definitions [ASU86] and reaching decomposition problems [Tse93]. If a data layout of $phase_1$ reaches $phase_2$, then the DLG contains edges between every node representing a candidate data layout in the search space of $phase_1$ to every node representing a candidate data layout in the search space of $phase_2$.

4.5.2 Example

The example ADI program kernel introduced in Section 4.1 has a two-dimensional program template of size `N` in each dimension. The alignment analysis builds alignment search spaces that map the three arrays `a`, `b`, and `c` canonically onto the program template as described in Section 4.3.2. To simplify the example, we assume that distribution analysis generates only one-dimensional `BLOCK` distributions, i.e., there are only two candidate data layouts for each phase, namely a row layout `(BLOCK,*)` and a column layout `(*,BLOCK)`. The resulting data layout graph is shown in Figure 4.15. The edges represent possible remappings of arrays between candidate data layouts. To model the effects of the iterative loop, the data layout graph contains edges between the layouts of the seventh phase and the second phase. The node and edge weights are the estimated costs for phase execution and remapping, respectively, in terms of execution time, multiplied by their predicted execution frequencies. The cost for transposing a single array is denoted by `T`. `max` is the number of iterations in the iterative loop.

To solve the data layout selection problem, a single candidate data layout must be chosen for each phase such that the overall cost of the selected layouts is minimal. The overall cost of a set of selected layouts is the sum of the weights of their representing nodes and the weights of all edges between these nodes. The solution requires that the value of `T` and `max` are known. As shown in the next section, the

4.5.3 NP-completeness Proof

For the purposes of this discussion, programs are assumed to have no loops or branches. Therefore, the PCFG consists of a linear sequence of phase nodes, $P_1, \dots P_n$, with no control flow between phases. Showing that the restricted problem is NP-complete implies that the general problem is NP-complete as well.

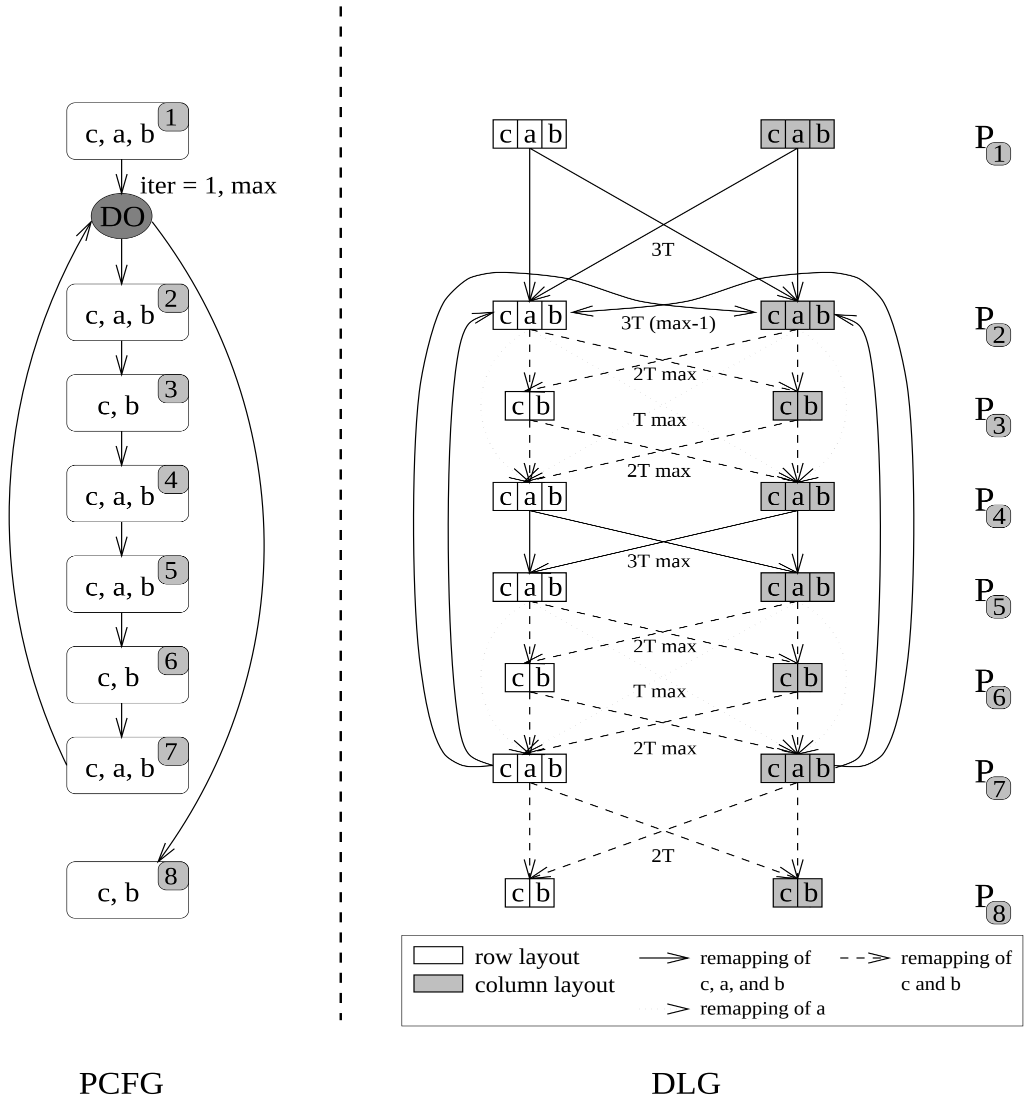


Figure 4.15 PCFG and DLG for the ADI kernel example. To simplify the example, we assume that there are only two candidate data layouts in each search space. Weights in the DLG represent static performance estimates of overall execution times. Node weights are not shown. Unlabeled edges have zero weight. `T` is the cost of performing a single array transpose, and `max` is the number of iterations of the outermost loop of the ADI integration kernel.

Let V denote the set of variables in the program, $V = \{v_1, ...v_r\}$. Let p_i denote the variables referenced in the i-th phase P_i, i.e. $p_i \subseteq 2^V, 1 \leq i \leq n$. For each P_i there is a set of candidate data layouts $D_i = \{d_i^1, ... \ d_i^{m_i}\}$. Note that two phases may have a different number of candidate data layouts. A single candidate data layout $d_i^k = \{d_{ij_1}^k, ...d_{ij_{q_i}}^k\}, 1 \leq k \leq m_i$, is a set of layouts, one layout for each variable $v \in p_i = \{v_{j_1}, ...v_{j_{q_i}}\}$.

The cost of executing phase P_i under the data layout $d_i^k \in D_i$ is denoted by $c(P_i, d_i^k)$. The remapping cost from one data layout scheme to another can be defined based on the remapping costs of each individual variable common to both schemes. Let d_α^s and d_β^t be two candidate data layouts for phase P_α and phase P_β, respectively. The remapping cost is given below:

$$c(d_\alpha^s, d_\beta^t) = \sum_{v_i \in p_\alpha \cap p_\beta} c(d_{\alpha i}^s, d_{\beta i}^t),$$

where $c(d_{\alpha i}^s, d_{\beta i}^t)$ is the cost for remapping the single variable v_i.

Let $f_i : p_i \rightarrow \{1, ...n\}$ be a mapping that determines for each variable $v \in p_i$ the phase that most recently referenced v before P_i. If no such phase exists, then $f_i(v)$ has the value i. A data remapping of v may occur between phase $P_{f_i(v)}$ and phase P_i.

Definition 4.4 An instance of the data layout selection problem consists of a program with a linear sequence of n phases, a set of program variables $V = \{v_1, ...v_r\}$, sets p_i and D_i for each phase P_i, and cost functions $c(P_i, d_i)$, $d_i \in D_i$, and $c(d_{ij}, d_{f_i(v_j)j})$ for each $v_j \in p_i$ and $d_i \in D_i$, with $1 \leq i \leq n$ and $1 \leq j \leq r$.

Definition 4.5 A solution of an instance of the data layout selection problem is a set $\{d_1, d_2, ...d_n\}$ of data layout schemes $d_i \in D_i, 1 \leq i \leq n$, such that

$$\sum_{i=1}^{n} c(P_i, d_i) \quad + \quad \sum_{i=1}^{n} \sum_{v_j \in p_i} c(d_{ij}, d_{f_i(v_j)j})$$

is minimized, where $c(d_{ij}, d_{ij})$ is 0. Note that this implies not associating any cost with an initial data layout.

The data layout selection problem is an optimization problem. We will prove that the corresponding decision problem DYN-REMAP(k) is NP-complete.

Definition 4.6 DYN-REMAP(k) represents a decision problem defined as follows:

DYN-REMAP(k) := set of all instances of the data layout selection problem such that there exists a set of data layouts $\{d_1, d_2, ...d_n\}$, $d_i \in D_i, 1 \leq i \leq n$, with a cost less or equal to k, where k is a non-negative integer.

Definition 4.7 An instance of the 3 Conjunctive Normal Form Satisfiability Problem consists of a boolean expression B in conjunctive normal form,

$$B = \bigwedge_{i=1}^{t} F_i,$$

where $F_i = l_i^1 \vee l_i^2 \vee l_i^3, \;\; 1 \leq i \leq t$, and each literal is a variable or its negation in the set of variables $V = \{v_1, ...v_r\}$.

The decision problem 3SAT is represented as follows:

3SAT := set of all instances of the 3 Conjunctive Normal Form Satisfiability Problem for which there exists a truth value assignment $w : V \rightarrow \{\texttt{true}, \texttt{false}\}$ such that B evaluates to `true` under w.

Theorem 4.1 DYN-REMAP(k) is NP-complete.

Proof The proof consists of two parts. Lemma 4.5 shows that DYN-REMAP(k) is in NP. Lemma 4.6 states that 3SAT can be reduced to DYN-REMAP(k) in polynomial time. Since 3SAT is NP-complete, DYN-REMAP(k) has to be NP-complete. $\square$

Lemma 4.5 DYN-REMAP(k) is in NP.

Proof Let $\{d_1, d_2, ...d_n\}$, $d_i \in D_i, 1 \leq i \leq n$, be a set of data layouts for an instance of the data layout selection problem, one data layout for each phase in the program. The overall cost of this set can be computed in polynomial time. Therefore it can be verified in polynomial time whether a given set of data layouts has a cost smaller or equal to a given cost k. Hence, DYN-REMAP(k) is in NP. $\square$

Lemma 4.6 3SAT can be reduced in polynomial time to DYN-REMAP(k).

Proof Part 1 of the proof defines the function g that maps an instance B of the 3SAT problem onto an instance $g(B)$ of the DYN-REMAP(0) problem. Part 2 contains the proof that $B \in$ 3SAT $\Leftrightarrow g(B) \in$ DYN-REMAP(0). Finally, Part 3 shows that g can be computed in polynomial time.

Part 1: Let B be an arbitrary instance of the 3 Conjunctive Normal Form Satisfiability Problem, $B = \bigwedge_{i=1}^{t}(\ l_i^1 \vee l_i^2 \vee l_i^3\ )$. g maps the instance B to an instance of the data layout selection problem as follows:

- $V = \{v_1, ...v_r\}$, i.e. the sets of variables are the same.
- Each clause F_i is represented by a distinct phase P_i, $1 \leq i \leq t$.
- $p_i = \{v_j \mid l_i^k \textit{ is a literal of variable } v_j,\ 1 \leq k \leq 3\}$, where $1 \leq i \leq t$. Note that $|p_i| \leq 3$.
- Each variable $v \in p_i$ has 2 possible data layouts, called `T` and `F`. D_i contains $2^{|p_i|}$ candidate layouts, one layout for each possible combination of the single variable layouts. In other words, each $d_i \in D_i$ represents a truth value assignment w_i for all variables in p_i:

$$w_i(v_j) = \begin{cases} \texttt{true} & \text{if } d_{ij} = \texttt{T} \\ \texttt{false} & \text{if } d_{ij} = \texttt{F} \end{cases}$$

- Assume $D_i = \{d_i^1, ...\ d_i^m\}$.

  $$c(P_i, d_i^k) = \begin{cases} 0 & \text{if } F_i \text{ is true under the truth value assignment represented by } d_i^k\ , \\ 1 & \text{otherwise}\ , \end{cases}$$

  where $1 \leq i \leq t$ and $1 \leq k \leq m$.

- Assume $d_{ij}^k \in d_i^k \in D_i$ and $d_{i'j}^{k'} \in d_{i'}^{k'} \in D_{i'}$, where $i' = f_i(v_j)$.

  $$c(d_{ij}^k, d_{i'j}^{k'}) = \begin{cases} 0 & \text{if both data layouts are identical} \\ 1 & \text{otherwise} \end{cases}$$

  In other words, $c(d_{ij}^k, d_{i'j}^{k'}) = 0$ if and only if no remapping of v_j is required between the two data layouts.

Note that there are at most eight data layouts for each phase. An example of the application of g to an instance of 3SAT is discussed in the next section.

Part 2a: Claim: $B \in$ 3SAT $\Rightarrow g(B) \in$ DYN-REMAP(0).

Proof: Let $w : V \rightarrow \{\texttt{true}, \texttt{false}\}$ be a truth value assignment that satisfies the problem instance B. There is exactly one data layout scheme in each phase P_i of $g(B)$ that represents w restricted to the variables in p_i. Call this data layout scheme d'_i. For all i, $1 \leq i \leq t$, $c(P_i, d'_i) = 0$. Since w specifies a unique data layout for each single program variable $v_j \in V$, redistribution between the set of data layouts $\{d'_1, d'_2, \ldots d'_t\}$ does not occur. Therefore the set has an overall cost of 0. Hence $g(B) \in$ DYN-REMAP(0).

Part 2b: Claim: $g(B) \in$ DYN-REMAP(0) $\Rightarrow B \in$ 3SAT.

Proof: Let $\{d_1, d_2, \ldots d_t\}$ be a set of data layouts, one data layout for each phase P_i, with an overall cost of 0. Therefore no remapping can occur between the data layouts and each data layout d_i has to represent a truth value assignment that satisfies F_i. Hence, there exists a unique truth value assignment w that satisfies all $F_i, 1 \leq i \leq t$. The existence of such a truth value assignment means that B is in 3SAT.

Part 3: Claim: $g(B)$ can be computed in polynomial time.

Proof: The collection of functions f_i can be computed in $\mathcal{O}(t\ r)$, where t and r are the number of phases and program variables, respectively.

There are at most 8 data layouts per phase. Therefore there are at most $t * 8$ data layout costs of the form $c(P_i, d_i^k)$ for the entire program. For each data layout d_i^k of phase P_i, at most $3 * 8$ remapping costs for individual variables $v_j \in p_i$ of the form $c(d_{ij}^k, d_{i'j}^{k'})$ have to be computed, resulting in at most $3 * 8^2$ remapping costs for each phase and at most $t * 3 * 8^2$ for the entire program. Hence, g can be computed in polynomial time.

$\square$

Example Reduction

The function g maps the instance $B = (v_1 \vee \neg v_2 \vee v_3) \wedge (\neg v_1 \vee v_2 \vee v_4) \wedge (v_1 \vee v_3 \vee \neg v_4)$ of the 3SAT problem into an instance of the decision problem DYN-REMAP(0) as follows:

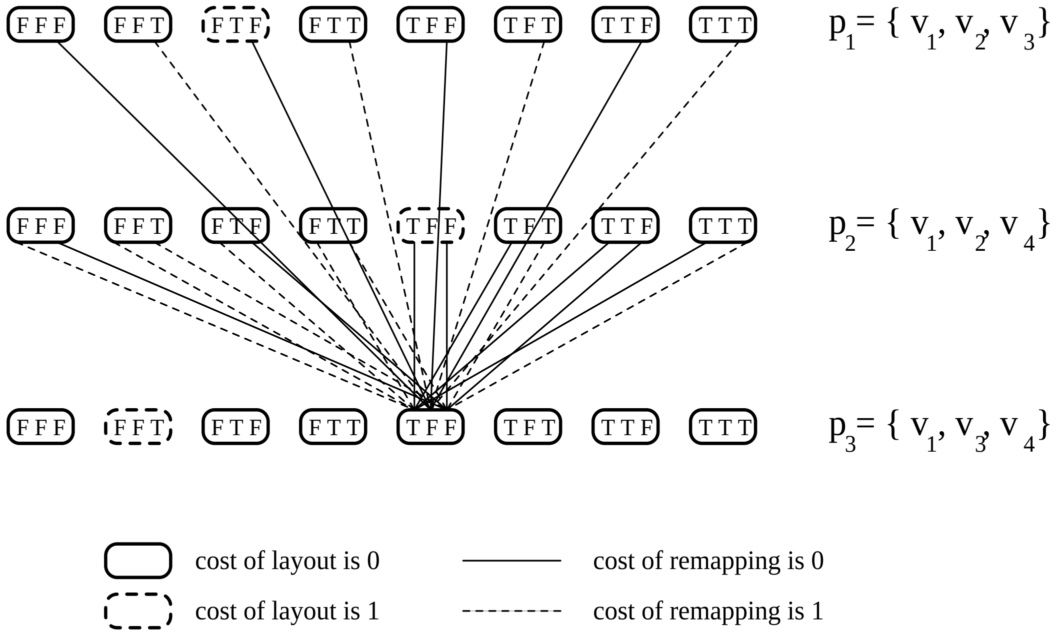


Figure 4.16 Sample costs for $g(B)$,
$B = (v_1 \vee \neg v_2 \vee v_3) \ \wedge\ (\neg v_1 \vee v_2 \vee v_4) \ \wedge\ (v_1 \vee v_3 \vee \neg v_4)$

- $V = \{v_1, v_2, v_3, v_4\}$
- There are three phases, P_1, P_2,and P_3. The ordering of the phases is given by their indices.
- $p_1 = \{v_1, v_2, v_3\}$, $p_2 = \{v_1, v_2, v_4\}$, and $p_3 = \{v_1, v_3, v_4\}$.
- $D_1 = \{ \{(v_1, F), (v_2, F), (v_3, F)\}, \{(v_1, F), (v_2, F), (v_3, T)\}, \{(v_1, F), (v_2, T), (v_3, F)\},$
$\{(v_1, F), (v_2, T), (v_3, T)\}, \{(v_1, T), (v_2, F), (v_3, F)\}, \{(v_1, T), (v_2, F), (v_3, T)\},$
$\{(v_1, T), (v_2, T), (v_3, F)\}, \{(v_1, T), (v_2, T), (v_3, T)\} \}$,
$D_2 = \{ \{(v_1, F), (v_2, F), (v_4, F)\}, \{(v_1, F), (v_2, F), (v_4, T)\}, \{(v_1, F), (v_2, T), (v_4, F)\},$
$\{(v_1, F), (v_2, T), (v_4, T)\}, \{(v_1, T), (v_2, F), (v_4, F)\}, \{(v_1, T), (v_2, F), (v_4, T)\},$
$\{(v_1, T), (v_2, T), (v_4, F)\}, \{(v_1, T), (v_2, T), (v_4, T)\} \}$, and

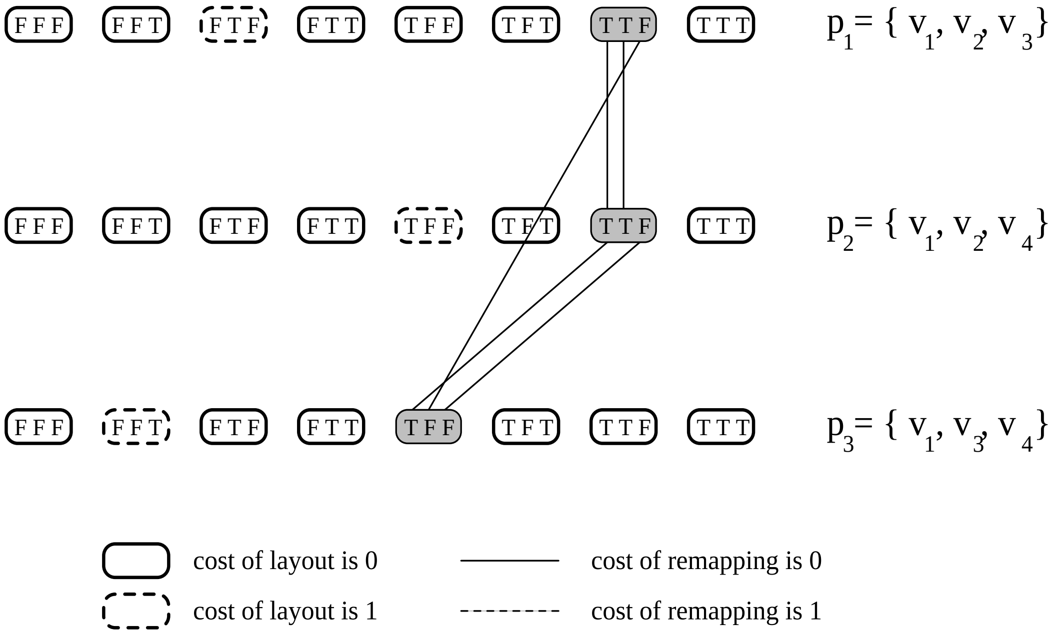


Figure 4.17 A solution for $g(B)$,
$B = (v_1 \vee \neg v_2 \vee v_3) \ \wedge \ (\neg v_1 \vee v_2 \vee v_4) \ \wedge \ (v_1 \vee v_3 \vee \neg v_4)$

$$
\begin{aligned}
D_3 \ = \ \{ & \{(v_1,F),(v_3,F),(v_4,F)\},\{(v_1,F),(v_3,F),(v_4,T)\},\{(v_1,F),(v_3,T),(v_4,F)\}, \\
& \{(v_1,F),(v_3,T),(v_4,T)\},\{(v_1,T),(v_3,F),(v_4,F)\},\{(v_1,T),(v_3,F),(v_4,T)\}, \\
& \{(v_1,T),(v_3,T),(v_4,F)\},\{(v_1,T),(v_3,T),(v_4,T)\} \ \}.
\end{aligned}
$$

- The costs for the data layouts for the phases and the costs for remapping of individual variables is shown in Figure 4.16. Individual remapping costs are only shown for $d_3^5 \in D_3$, $d_3^5 = \{ \ (v_1,T), \ (v_3,F), \ (v_4,F) \ \}$. Each edge in the graph represents a cost function value $c(d_{3j}^5, d_{f_3(v_j)j}^k)$, $j \in \{1,3,4\}$.

Figure 4.17 shows a solution s to the example data layout selection problem, $s = \{d_1^7, d_2^7, d_3^5\} \in$ DYN-REMAP(0). Note that all costs are 0. The corresponding truth value assignment is $\{(v_1, \texttt{true}),\ (v_2, \texttt{true}),\ (v_3, \texttt{false}),\ (v_4, \texttt{false})\}$. This truth value assignment satisfies B.

4.5.4 Polynomial Time Solution for Special Case

In some cases, the solution to the data layout selection problem can be determined in polynomial time. For instance, assume that the PCFG contains only structured control flow such as loops and branches, and each data layout candidate specifies the layout of every array in the program. In addition, subgraphs in the PCFG that represent branches have unique entry and exit phase nodes. We will show that in this case the data layout selection problem can be solved in polynomial time in the sizes of the candidate layout search spaces. The main idea is to decompose the data layout selection problem into subproblems that can be solved as single-source shortest path problems [CLR90]. The "length" of a path includes the weights on the nodes and the weights of the edges along the path.

The PCFG is decomposed in a hierarchical fashion, from innermost to outermost loop or branch levels. The possible contributions of each loop or branch to the solution of the data layout selection problem for the entire program is summarized in special DLGs, called *loop DLGs* or *branch DLGs*. The summary DLGs are computed based on the DLGs of the subgraphs in the PCFG that correspond to the control flow structures. A summary DLGs represent its control flow structure on the next higher loop or branch level. The DLG for the entire program is constructed bottom-up, based on the summary DLGs and the DLGs for the phases at each loop or branch level.

Each summary DLG has a unique set of entry and exit candidate layout search spaces. The requirement that each branch must have a unique entry and exit phase in the PCFG is a sufficient condition for this property. Edges between entry and exit candidate layouts are weighted by the cost that will occur if the control flow structure is entered using the entry layout and left using the exit layout. In addition, each edge is annotated with its associated minimum cost path in the DLG of its control flow structure.

The construction of the loop summary DLG is illustrated in Figure 4.18. The loop structure DLG is only an intermediate step in computing the loop summary DLG. The weight of an edge in the summary DLG from a candidate layout of the first phase (entry layout) to a candidate layout of the third phase (exit layout) is the result of computing the shortest path in the loop structure DLG from the entry layout to the exit layout, and adding the cost of remapping between the exit and entry layout. Note that the edge weights in the loop structure DLG reflect the frequencies of execution.

A dynamic programming solution of the single-source shortest paths problem for a directed acyclic graph is described in [CLR90]. The time complexity of the algorithm is linear in the number of edges in the graph. Let k denote the maximal number of layout candidates for each phase and p the number of phases. The number of edges in the loop structure DLG is $\mathcal{O}(pk^2)$. For each entry candidate layout, a single single-source shortest paths problem has to be solved. The solution provides the shortest paths from the entry candidate layout to each exit candidate layout. As a result, the overall complexity of constructing a loop summary DLG is $\mathcal{O}(pk^3)$.

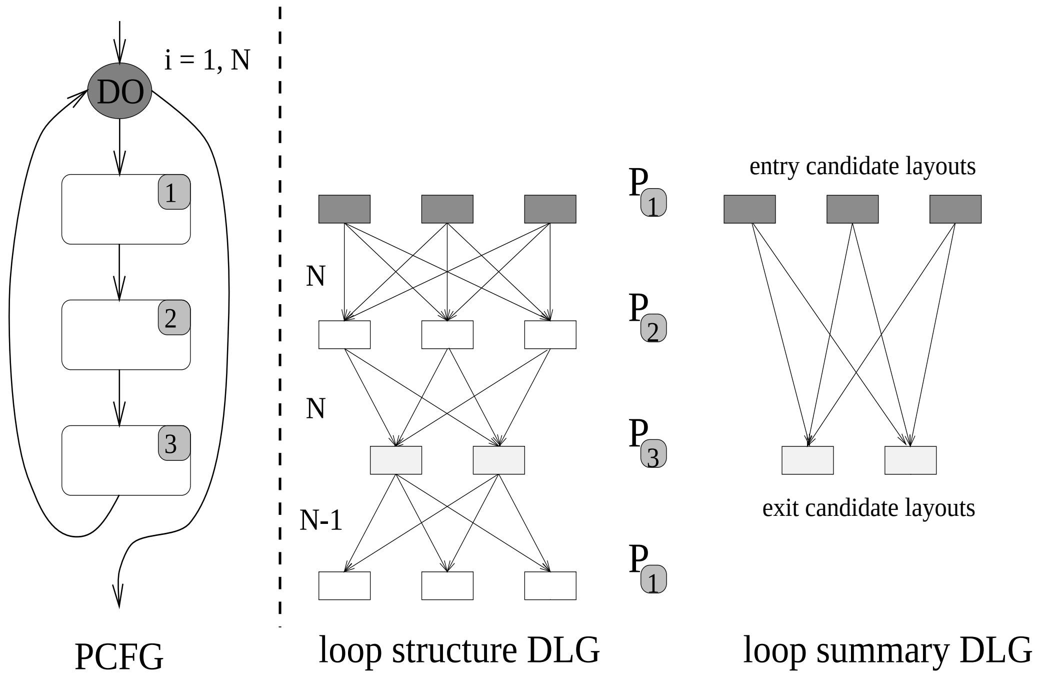


Figure 4.18 Example loop summary DLG. The layout search spaces for the first two phase in the loop body are assumed to have three candidate layouts. The search space for the third phase is assumed to have two layouts.

The construction of a branch summary DLG is shown in Figure 4.19. As during the construction of the summary loop DLG, k single-source shortest paths problems are solved, one for each entry candidate layout in the branch structure DLG. However, to compute the weight of an edge in the branch summary DLG, the costs of the distinct shortest paths from an entry candidate layout to an exit candidate layout through the *true* and *false* branches are summed up. The overall time complexity of constructing a branch summary DLG is $\mathcal{O}(pk^3)$.

Once the hierarchical algorithm reaches the outermost nesting level, a single single-source shortest path problem is solved to compute the overall solution. Artificial entry and exit nodes are introduced to the outermost DLG. The artificial nodes and their adjacent edges have zero weight. This last step is shown in Figure 4.20. The time complexity to solve a single single-source shortest paths problem is $\mathcal{O}(pk^2)$. Note that each summary edge is annotated with its cost and associated minimum cost paths. Based on this information, the shortest path in the outermost DLG determines a minimal cost path for the DLG of the entire program.

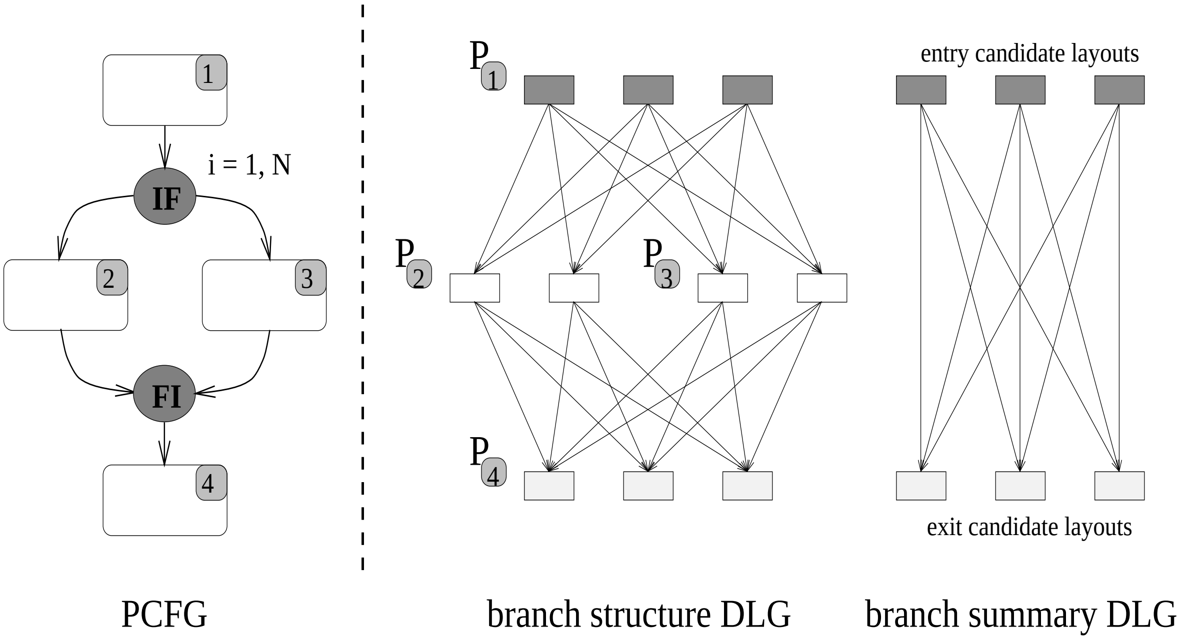


Figure 4.19 Example branch summary DLG. The layout search spaces for the entry and exit phase are assumed to have three candidate layouts. The phases in the true and false branches are assumed to have two candidate layouts in their search spaces.

Theorem 4.2 Assume that the PCFG contains only structured control flow such as loops and branches, and each data layout candidate specifies the layout of every array in the program. In addition, assume that subgraphs in the PCFG that represent branches have unique entry and exit phase nodes. The data layout selection problem for the resulting DLG can be solved in polynomial time in the sizes of the candidate layout search spaces.

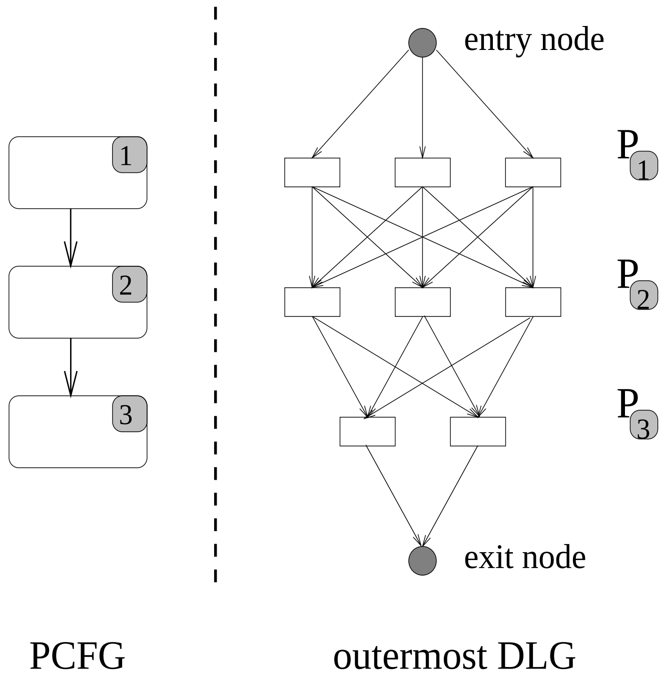


Figure 4.20 Example DLG for outermost program level. The candidate layout search spaces are assumed to have three and two layouts for the first two phases and the third phase, respectively.

Proof

Without loss of generality, we assume that each candidate layout search space has exactly k candidate layouts. Let p denote the number of phases in the PCFG for the entire program. The hierarchical algorithm described in this section visits each edge in the DLG at most k times. The worst case occurs if the edge is part of a loop or branch structure DLG and therefore k single-source shortest paths problems have to be solved. Each summary step substitutes a DLG subgraph by a summary DLG that has at most half as many edges. As any original edge, each newly introduced edge will be visited at most k times. Therefore, the overall time complexity of the entire algorithm is

$$\sum_{i=0}^{\infty} \mathcal{O}(k \, \frac{1}{2^i} \, pk^2) \;=\; \mathcal{O}(2 \, pk^3)$$

□

4.5.5 0–1 Integer Programming Formulations

This section discusses the details of translating an instance of a data layout selection problem and its data layout graph (DLG) into an instance of a 0–1 integer programming problem with linear constraints, or *0–1 problem* for short. An overview of 0–1 integer programming can be found in Section 2.2.

For the purpose of this discussion, we assume that the input program has n phases, $P_1, \ldots P_n$. The corresponding DLG has m_i nodes for each phase i, $1 \leq i \leq n$. Each node in the DLG represents a particular candidate data layout that specifies the alignment and distribution of all variables referenced in the phase.

> **Definition 4.8** An *instance of a 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 the objective function while respecting the constraints.

There are many possible translations of the data layout selection problem into an equivalent 0–1 problem. The resulting 0–1 problem instances may differ significantly in the time needed by an integer programming tool to compute an optimal solution. In this section we will discuss three different translations. Experimental results for the three formulations and a general-purpose integer programming tool are presented in Section 5.1.

All three translations introduce a variable for each node and edge in the DLG. These variables can be thought of as switches that are *on* if and only if the represented nodes or edges are part of a solution of the data layout selection problem. The set X is the union of two sets of variables, $X = X_{layout} \cup X_{remap}$. X_{layout} contains a single switch for each node in the DLG, and X_{remap} has one switch for each edge. The switch $x_{ik} \in X_{layout}$ represents the k-th node of the i-th phase. The switch $x_{ik}^{jl} \in X_{remap}$

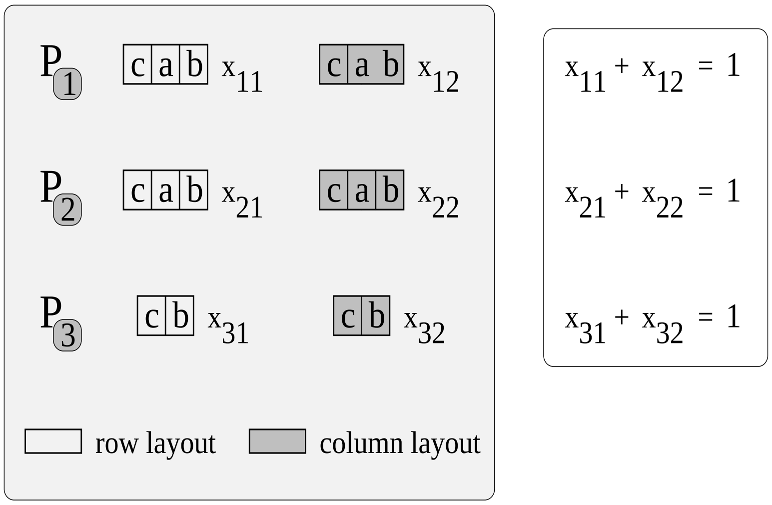


Figure 4.21 Layout constraints for the first three phases of our ADI kernel example.

represents the remapping edge between the l-th node of phase j and the k-th node of phase i.

Similar to the variable set X, the set of constraints is partitioned into two classes. Constraints that ensure the selection of only a single node for each phase are called *layout constraints*. *Remapping constraints* guarantee that all remapping edges between selected nodes are considered, i.e. are also selected. The three translations differ only in the formulation of their remapping constraints.

All three translations define their layout constraints as follows. For each phase i, $1 \leq i \leq n$, there is a constraint of the form

$$\sum_{k=1}^{m_i} x_{ik} = 1$$

In other words, exactly one switch has to be *on* for each phase and all other switches for the phase have to be *off*. The layout constraints for the first three phases of our ADI kernel example DLG (Figure 4.15) are shown in Figure 4.21.

Two translations base their remapping constraints formulation on a counting argument over the number of incoming and outgoing edges of each node in the DLG. We will refer to these translations as *node-based* formulations. In contrast, the third translation introduces a pair of constraints for each edge in the DLG. It is therefore referred to as an *edge-based* formulation. The different remapping constraints are discussed in detail in the next section.

A solution of an instance of the generated 0–1 problem *minimizes* the following objective function under the layout and remapping constraints:

$$\sum_{x_{ik} \in X_{layout}} x_{ik}\ cost_{layout}\ (x_{ik}) \quad + \sum_{x_{ik}^{jl} \in X_{remap}} x_{ik}^{jl}\ cost_{remap}\ (x_{ik}^{jl})\ ,$$

where $cost_{layout}$ and $cost_{remap}$ represent the node and edge weights of the data layout graph, respectively.

Remapping Constraints Formulation

The node-based formulations introduce two types of remapping constraints for each node in the data layout graph, namely IN-constraints and OUT-constraints. The edge-based formulation has a pair of constraints for each edge in the DLG.

Compact node-based formulation: Define P_i^{in} as the set of phases that contain source nodes of edges with sink nodes in phase P_i. For the node represented by x_{ik} and all incoming edges with nodes in phases P_i, IN-constraints of the form

$$\sum_{j \in P_i^{in}} \sum_{l=1}^{m_j} x_{ik}^{jl} = x_{ik}\ |P_i^{in}|$$

are generated. In other words, if switch x_{ik} is *on* then exactly $|P_i^{in}|$ switches representing incoming edges from phases in P_i must be *on*. If x_{ik} is *off* then all incoming edge switches have to be *off*.

Similarly, define P_i^{out} as the set of phases that contain sink nodes of edges with source nodes in phase P_i. For the node represented by x_{ik} and all outgoing edges with nodes in phases P_i, OUT-constraints of the form

$$\sum_{j' \in P_i^{out}} \sum_{l'=1}^{m_{j'}} x_{j'l'}^{ik} = x_{ik}\ |P_i^{out}|$$

are generated. In other words, if switch x_{ik} is *on* then exactly $|P_i^{out}|$ switches representing outgoing edges from phases in P_i must be *on*. If x_{ik} is *off* then all outgoing edge switches have to be *off*.

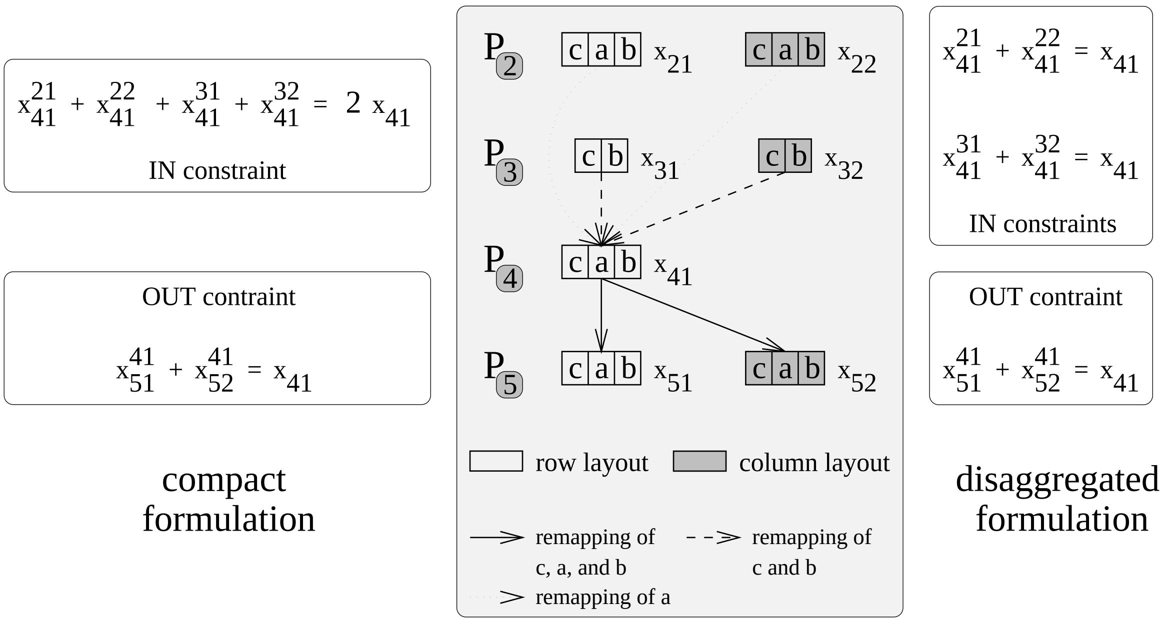


Figure 4.22 Two node-based remapping constraints formulations for the node x_{41} representing the first layout in the fourth phase of our ADI kernel example.

Disaggregated node-based formulation: For the node represented by x_{ik} and all incoming edges with nodes in the same phase j as their sources, IN-constraints of the form

$$\sum_{l=1}^{m_j} x_{ik}^{jl} = x_{ik}$$

are generated. In other words, if switch x_{ik} is *on* then exactly one switch representing an incoming edge from phase j must be *on*. If x_{ik} is *off* then all incoming edge switches have to be *off*.

Similarly, for the node represented by x_{ik} and all outgoing edges with nodes in the same phase j' as their sinks, OUT-constraints of the form

$$\sum_{l'=1}^{m_{j'}} x_{j'l'}^{ik} = x_{ik}$$

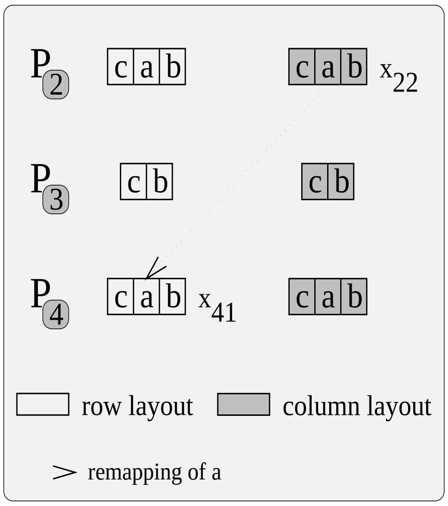
are generated. In other words, if switch x_{ik} is *on* then exactly one switch representing an outgoing edge to phase j must be *on*. If x_{ik} is *off* then all outgoing edge switches have to be *off*.

The node-based remapping constraints formulations for the node representing the first candidate layout in the fourth phase of the ADI integration example, x_{41}, are listed in Figure 4.22.

<u>Edge-based formulation</u>: For each edge x_{ij}^{kl} in the DLG the pair of constraints

$$x_{ik} + x_{jl} \geq 2\ x_{ik}^{jl} \quad and \quad x_{ik} + x_{jl} \leq 1 \ + \ x_{ik}^{jl}$$

are introduced. These two constraints model the constraint $x_{ik}\ x_{jl} = x_{ik}^{jl}$ which is non-linear and therefore cannot be used in our 0–1 problem formulation. A pair of edge-constraints for a single edge in our example DLG for the ADI kernel is shown in Figure 4.23.



$$x_{22} + x_{41} \geq 2\ x_{41}^{22}$$

$$x_{22} + x_{41} \leq 1 + x_{41}^{22}$$

Figure 4.23 Edge-based remapping constraints formulations for the edge x_{41}^{22} representing a remapping between the second and fourth phase of our ADI kernel example.

A correctness proof of the three remapping constraints formulations can be found in the appendix in Section A.2. The compact and disaggregated node-based formulations differ in the number and size of individual constraints. Typically, the compact formulation will have fewer constraints with more variables than the disaggregated formulation. However, the constraints of the disaggregated formulation are tighter in the sense that a single set of disaggregated IN or OUT constraints imply the validity of the corresponding set of compact constraints, but not vice versa. For instance, in the example shown in Figure 4.22, if switches x_{41}^{21}, x_{41}^{22}, and x_{41} are *on* and switches x_{41}^{31} and x_{41}^{32} are *off*, the compact IN constraint is satisfied, but the corresponding set of disaggregated IN constraints are not. Experiments based on the different formulations are discussed in Section 5.1.

Chapter 5

Experimental Results

Experiments were conducted to show that our framework is efficient and can be used to generate good data layouts. In order to verify the quality of data layouts generated by the framework, a prototype data layout assistant tool has been implemented.

5.1 Efficiency of 0–1 Integer Programming Formulations

5.1.1 Description of Experiments

The efficiency experiments of the different 0–1 integer programming formulations for the data layout selection problem are based on Erlebacher, a 500 line benchmark program written by Thomas Eidson at the Institute for Computer Applications in Science and Engineering (ICASE). The program performs 3-dimensional tridiagonal solves using Alternating Direction Implicit (ADI) integration. The code contains computational wavefronts across all three dimensions. Array kill analysis was performed by hand and arrays were renamed and replicated appropriately. The resulting program contains 40 phases and 25 arrays. There are arrays with one, two, and three dimensions.

Erlebacher has a three-dimensional alignment space. For our experiments, we assume that alignment analysis and distribution analysis generate seven candidate layouts for each phase, one layout for each possible combination of distributed dimensions. However, if a phase contains only one-dimensional arrays, its candidate search space has only four layouts since some layouts are the projection of two distribution schemes. The corresponding data layout graphs with different weights were generated by hand. Weights were chosen to model different communication costs and the presence or absence of compiler optimizations. For instance, a compiler may be able to generate a coarse-grain pipelined loop if the data layout induces cross-processor dependences [Tse93]. Whether the compiler performs such an optimization or not is represented by different weights of the nodes and edges in the DLG.

We wrote a tool that generates the three different 0–1 problem formulations discussed in Section 4.5.5 for an input DLG. The tool was written in *Scheme*, a dialect of *Lisp* [Dyb87]. The sole purpose of this tool was to provide a testbed to determine the efficiency of the different 0–1 problem formulations. The following table shows the sizes of the automatically generated 0–1 problem instances:

	compact node-based	**disaggregated node-based**	**edge-based**
#layout variables	253	253	253
#remapping variables	2133	2133	2133
#constraints	485	715	4306

For the experiment, 6 compact node-based, 12 disaggregated node-based, and 6 edge-based 0–1 problem formulations were automatically generated by the tool from different input DLGs. Each of the 0–1 problem instances was solved by *CPLEX*[†], a state-of-the-art linear integer programming tool and library, partly developed by Robert Bixby at Rice University [Bix92]. *CPLEX* includes an implementation of a general-purpose branch-and-bound code for mixed integer programming. Being general purpose, this code does not exploit the structural properties of our particular 0–1 problems.

The following table gives the solution times in seconds of the 24 0–1 problem instances using *CPLEX* on a SPARC-10. A "*" indicates that *CPLEX* took more than 30 minutes. The reported averages exclude experiments that took longer than 30 minutes. Note that some experiments took more than 4 hours.

compact node-based			**disaggregated node-based**			**edge-based**		
best	worst	avg.	best	worst	avg.	best	worst	avg.
8.2	*	292	2.6	4.8	3.8	*	*	*

The experiments show that the disaggregated node-based formulation can be solved by the general-purpose *CPLEX* in less than 4 seconds on average. For one 0–1 problem instance of the disaggregated formulation, *CPLEX* determined the op-

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

timal solution in 2.6 seconds. *CPLEX* did not take longer than 4.8 seconds on any of the twelve disaggregated 0–1 problem instances.

The improvement between the compact node-based and the disaggregated node-based formulations is subtle, but fundamental. The disaggregated formulation is perhaps less elegant, and certainly larger. It is also equivalent to the node-based formulation when integrality is imposed. However, when integrality is relaxed, it provides a much better approximation of the polytope of 0–1 solutions. The extra cost in the size of the linear programming relaxations is more than compensated for by the improved integrality properties of these relaxations. The edge-based formulation can be viewed as initial attempts to find cutting planes. It has yet to be proven that these inequalities are indeed independent of the inequalities in the disaggregated formulation. In addition, no studies have yet been made of the quality of these inequalities, that is, of the dimension of their intersection with the underlying polytope. Such research is outside of the scope of this thesis.

In order to assess the scalability of the disaggregated node-based formulation, four different DLGs based on Erlebacher were generated by hand. The DLGs correspond to programs consisting of multiple copies of the Erlebacher code. Remapping edges between individual copies were either introduced only for the four main, 3-dimensional program arrays (decoupled version), or for all arrays in the program (coupled version). The DLGs of the coupled versions contain more edges than of the decoupled versions. Typically, the additional edges in the DLGs of the coupled versions span across multiple phases due to the use of replicated and renamed array variables in our Erlebacher code. The resulting sizes of the 0–1 problem instances and their solution times using *CPLEX* on a SPARC-10 are given below.

	3 copies, 120 phases		**5 copies, 200 phases**	
	coupled	**decoupled**	**coupled**	**decoupled**
#number of variables	8668	7438	14950	12490
#number of constraints	2689	2227	4663	3739
solution times (seconds)	**115**	**34**	**357**	**87**

In Section 4.3.1 we have shown how the problem of relative alignment conflict resolution can be translated into a 0–1 integer programming problem. Evaluating the efficiency of our 0–1 alignment conflict resolution formulation based on real program examples is difficult. It is our experience that inter-dimensional alignment conflicts are not very common. Since the structure and number of variables and constraints is similar for both 0–1 problems, we expect the efficiency of the 0–1 conflict resolution formulation to be comparable to the efficiency of the disaggregated node-based formulation of the data layout selection problem for CAGs and DLGs of roughly equal sizes.

5.1.2 Discussion of Results

0–1 integer programming is NP-complete. Therefore, it is unrealistic to expect a solution for all instances in minimal computation time. However, for instances of our framework's two 0–1 problems that occur in practice, the experiments indicate that a general-purpose integer programming tool can compute optimal solutions with good efficiency.

The efficiency of the solution of our 0–1 integer programming problems can be improved on by developing special purpose solvers that exploit the particular structures of the 0–1 problem instances at hand. Recent experience with other NP-hard problems formulated as 0–1 integer programs — principally the TSP — indicate that a careful study of structure of the particular integer program can lead to very effective practical procedures. *CPLEX* is also designed to be applied as a callable library of linear-programming routines that can be conveniently built into a branch-and-cut code using cutting plane methods, such as a special purpose solver for the data layout selection problem or the inter-dimensional alignment problem. The development of such special purpose solvers is beyond the scope of this thesis.

In addition, exploiting the parallelism in the branch-and-bound nature of the solution process will allow additional, substantial efficiency improvements.

If the efficiency of the solution process is still not acceptable, or if an optimal solution is not desired, the 0–1 problem formulations can be used to implement efficient heuristics for potentially suboptimal solutions. By computing both upper and lower bounds during the branch-and-cut solution process, a solver can determine the first feasible solution found within some predefined percentage of optimal. Another possible heuristic can be implemented by a solver that returns the best feasible solution within a predefined time limit. In this case, the computed bounds will provide an estimate of the solution quality.

In summary, state-of-the-art integer programming technology allows the efficient solution of practical instances of the two 0–1 problems of our framework in the context of a data layout assistant tool. A "fire wall" can be built into a practical solver to terminate the solution process after a predefined time limit. If an optimal solution is not found within the time limit, the solver returns the best feasible solution found.

5.2 Quality of Generated Data Layouts

To show that our framework can be used to generate efficient data layouts, we have implemented a prototype data layout assistant tool based on our framework. The prototype tool was applied to four programs and program kernels, and the automatically generated data layouts were compared with the best possible data layout choice. The prototype and our experiments are discussed below.

5.2.1 Prototype Implementation

A prototype data layout assistant tool has been implemented as part of the D system [ACG+94]. The prototype is a batch system. It takes Fortran 77 programs as input and generates Fortran D data layout specifications for each phase in the program. 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 phase control flow graph (PCFG) is built based on the operational phase definition presented in Section 4.2. `Do` nodes and `If` nodes are annotated with loop control information and branch probabilities, respectively. The current implementation assumes that each outgoing edge from a `If` node has the same execution probability.

Search Space Construction

The alignment analysis performance model and the heuristic for alignment search space construction are discussed in Section 4.3.2. Distribution analysis generates exhaustive search spaces of 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 distribution [Tse93]. Note that since the current prototype generates exhaustive search spaces for 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.

Performance Estimation

The implemented performance estimator does not use compiler level training sets. In this section we discuss the compiler, execution and machine model of our prototype layout assistant tool.

Compiler Model

The prototype's compiler model supports the simulation of communication optimizations such as message vectorization, message aggregation, and message coalescing. Flags are used to enable or disable individual communication optimizations. For phases where each processor takes part in the computation, the compilation process is simulated for a single, non-boundary processor. If a phase is executed by only a single processor or pair of processors, for instance due to boundary computations, the simulation is done for the boundary processor or processors. As the result of the compiler simulation, each loop nesting level in a phase is annotated with information about the required communication at that level.

Execution Model

The execution model uses this communication information to detect processor synchronization. Phases are classified as either pipelined, loosely synchronous, or reductions. The current prototype can only handle single statement reductions.

For a loosely synchronous phase the estimated communication and computation costs are added up for the processor with the largest local segment involved in executing the phase.

For a pipelined phase the innermost level that carries a true dependence determines the granularity of the pipeline. This level is referred to as the *pipeline_level*. The computation estimate for a single pipeline stage, T_{stage}, is the cost for a single iteration at the pipeline level. Let *outer_iterations* denote the number of stages executed by each processor. The execution time for the entire pipeline, $T_{pipeline}$, is determined by the time needed by the last processor to finish all its stages and by its starting delay relative to the first processor. While the last processor executes its stages, we assume a complete overlap between communication and computation, i.e., the actual observable message latency is assumed to be zero. However, communication overhead may occur for each stage due to operations such as message copying. The communication overhead $T_{comm}^{low\ latency}$ for zero latency pipeline communication and

$T_{comm}^{high\ latency}$ for pipeline communication with fully observable latency are determined based on separate training sets. $T_{comm}^{high\ latency}$ is used to compute the starting delay of the last processor. Our overall performance model is similar to the model described by Mellor-Crummey, Adve, and Koelbel [MCAK94]. We compute $T_{pipeline}$ as follows:

$$\begin{aligned} T_{pipeline} = \quad & (\#processors - 1) * (T_{stage} + T_{comm}^{high\ latency}) \quad + \\ & outer\_iterations \quad * (T_{stage} + T_{comm}^{low\ latency}) \end{aligned}$$

For a reduction phase, the kind of reduction operation, for instance `min`, `max`, or `sum`, and its data type is determined. Since the current prototype assumes the owner-computes rule, the overall cost is the sum of a single local computation and the global reduction operation. The costs of global reductions are determined by training sets.

Machine Model

The identification of basic computations is based on the phase's abstract syntax tree representation (AST [ASU86]). Performance estimates for basic computations and communication patterns are based on machine level training sets for Intel's iPSC/860 or Paragon. A general discussion of machine level training sets can be found in Section 4.4.2.

The current machine model implementation uses more than 100 training sets that measure basic computations such as `real` and `double` floating point operations, and basic communication patterns such as nearest neighbor communication (`shift`), 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 usually requires message buffering. Low latency message costs are used to estimate the communication costs $T_{comm}^{low\ latency}$ in pipelined phases where computation and communication can be overlapped. In contrast, message costs $T_{comm}^{high\ latency}$ for loosely synchronous phases are based on high latency training sets.

The communication pattern training sets are based on calls to Intel's native communication library. The training sets can be compiled and executed on Intel's family of distributed-memory multiprocessors. Training set node programs are compiled using the highest level of optimization (if77 –O4).

Figure 5.1 shows the structure of the arithmetic training set. For each arithmetic operation and data type there is separate timing loop in the training set. Arithmetic operations include intrinsic functions. The measured memory access patterns are based on one-dimensional array accesses with unit stride. As a result, most array accesses will be in cache. The cost function generated by the arithmetic training set used in our prototype tool is optimistic in the sense that it assumes in cache accesses.

```
C       – << determine loop overhead >> –
        timeLoopOverhead = dclock()
        do iter = 1, maxiter
        enddo
        timeLoopOverhead = (dclock() – timeLoopOverhead) / maxiter
C       – << determine cost of addition >> –
        do iter = 1, maxiter
            x(iter) = x(iter) + y(iter)
        enddo
        time = (dclock() – time) / maxiter   –   timeLoopOverhead
```

Figure 5.1 Structure of training set for arithmetic operations

The structures of the training sets for unit stride nearest neighbor `shift` communication patterns with high latency and low latency are shown in Figure 5.2 and Figure 5.3, respectively. The resulting cost functions and cost functions for the corresponding non-unit stride training sets are shown in Figure 5.4. The cost functions were generated by executing the training sets on 8 processors. Note that due to cache effects, communication costs can vary significantly for only slightly different message sizes.

Figure 5.6 shows the cost functions generated by the training sets for `broadcast` communication patterns with unit stride memory accesses on 2, 4, 8, 16, and 32 processors. The structure of the training sets is shown in Figure 5.5.

In the current prototype, the raw training set data is represented internally as cost tables. Linear interpolation is used if a requested data point has no entry in the cost table.

```
      time = dclock()

      do iter = 1, maxiter
         if (my$p .gt. 0) then
C           − << send message of length "bytes" to left neighbor >> −
            call csend(115, x, bytes, my$p−1, my$pid)
         endif
         if (my$p .lt. nprocs−1) then
C           − << receive message of length "bytes" from right neighbor >> −
            call crecv(115, y, bytes)
         endif
      enddo

      time = (dclock() − time) / maxiter
```

Figure 5.2 Structure of training set for high latency, unit stride shift communication pattern for iPSC/860 or Paragon

```
C     − << computation to hide message latency >> −
      time2 = dclock()
      < computation Comp_overlap >
      time2 = dclock() − time2

      time = dclock()

      do iter = 1, maxiter
         if (my$p .gt. 0) then
C           − << send message of length "bytes" to left neighbor >> −
            call csend(115, x, bytes, my$p−1, my$pid)
         endif

         < computation Comp_overlap >

         if (my$p .lt. nprocs−1) then
C           − << receive message of length "bytes" from right neighbor >> −
            call crecv(115, y, bytes)
         endif
      enddo

      time = (dclock() − time) / maxiter  −  time2
```

Figure 5.3 Structure of training set for low latency, unit stride shift communication pattern for iPSC/860 or Paragon

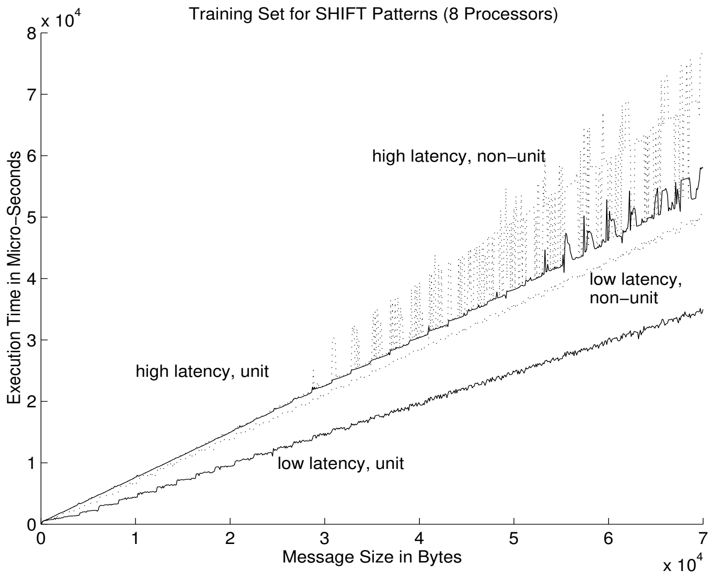


Figure 5.4 Results generated by training sets for shift communication pattern for 8 processors on iPSC/860

```
      time = dclock()

      do iter = 1, maxiter
         if (my$p .eq. 0) then
C           − << processor #0: broadcast message of length “bytes” >> −
            call csend(115, x, bytes, −1, my$pid)
         else
C           − << receive message of length “bytes” from processor #0 >> −
            call crecv(115, y, bytes)
         endif
      enddo

      time = (dclock() − time) / maxiter
```

Figure 5.5 Structure of training set for unit stride broadcast communication pattern for iPSC/860 or Paragon

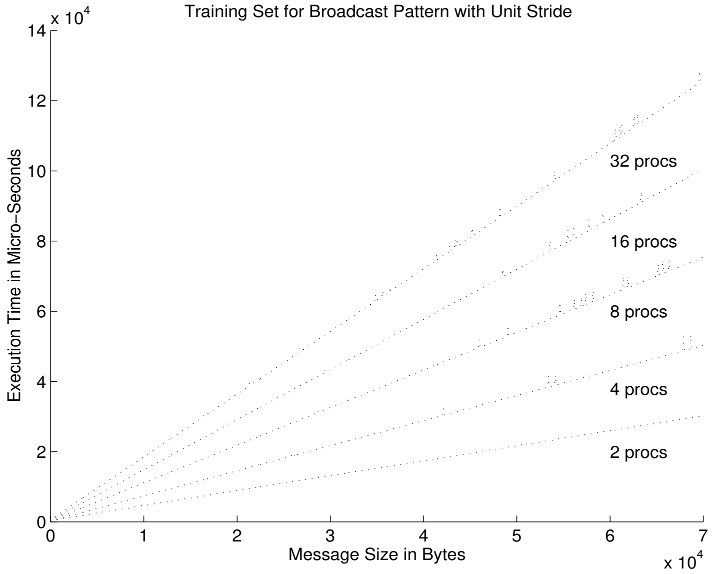


Figure 5.6 Results generated by training sets for unit stride broadcast communication pattern for different numbers of processors on iPSC/860

Data Layout Selection

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 the general-purpose version of *CPLEX*. The data layout selection step uses the disaggregated node-based formulation as discussed in Section 4.5.5. The prototype tool builds the required constraint matrices internally, and directly calls the CPLEX routines without creating any intermediate files.

5.2.2 Description of Experiments

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.

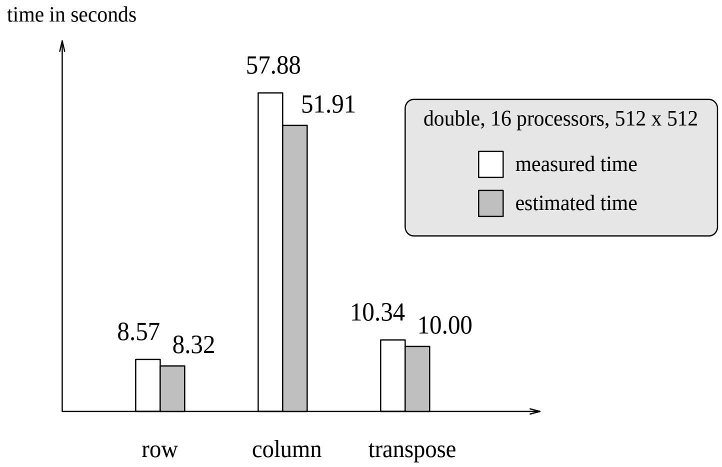


Figure 5.7 Example test case for ADI with three possible data layouts

It is important to note that it is not the goal of the 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). All programs are written in a data-parallel programming style that allows good compile time analysis [KR94].

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 5.7 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 5.7 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.

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 5.7. The results of all five test cases for problem size 256×256 and double precision arrays can be found in Figure 5.8. 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 37 cases. In all these cases the relative rankings of data layout alternatives were correct. In 3 cases the automatically chosen layout was suboptimal. The worst case performance loss due to the suboptimal selection was 7.4% as compared to the best possible choice.

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.

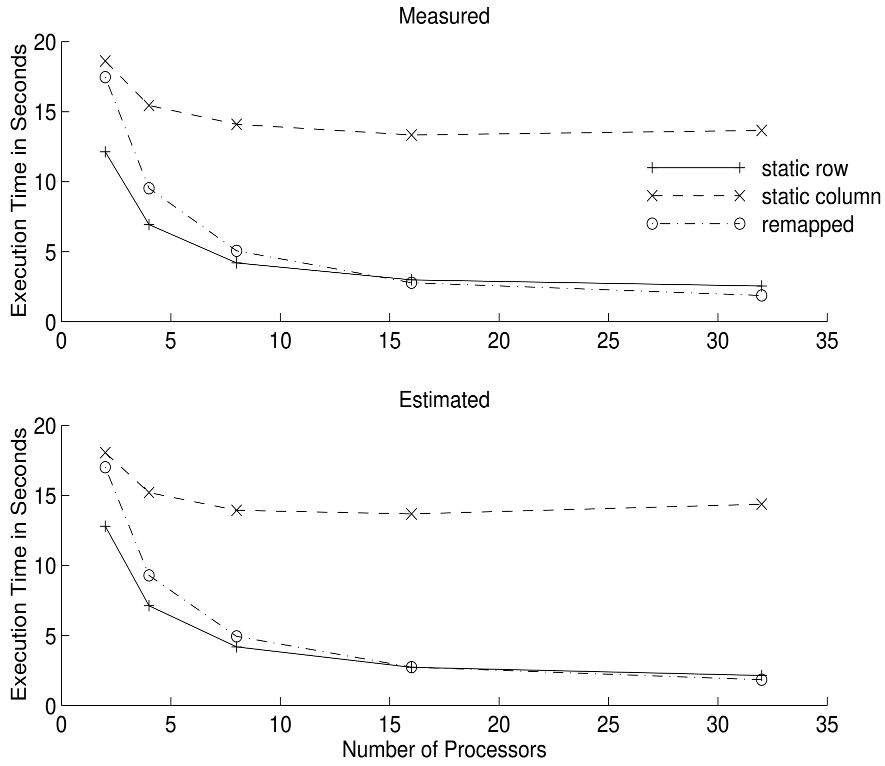


Figure 5.8 Measured and estimated execution times for Adi kernel with problem size 256×256, double precision

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.

We measured 20 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 9 cases.

The prototype tool determined the best layout in 12 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.

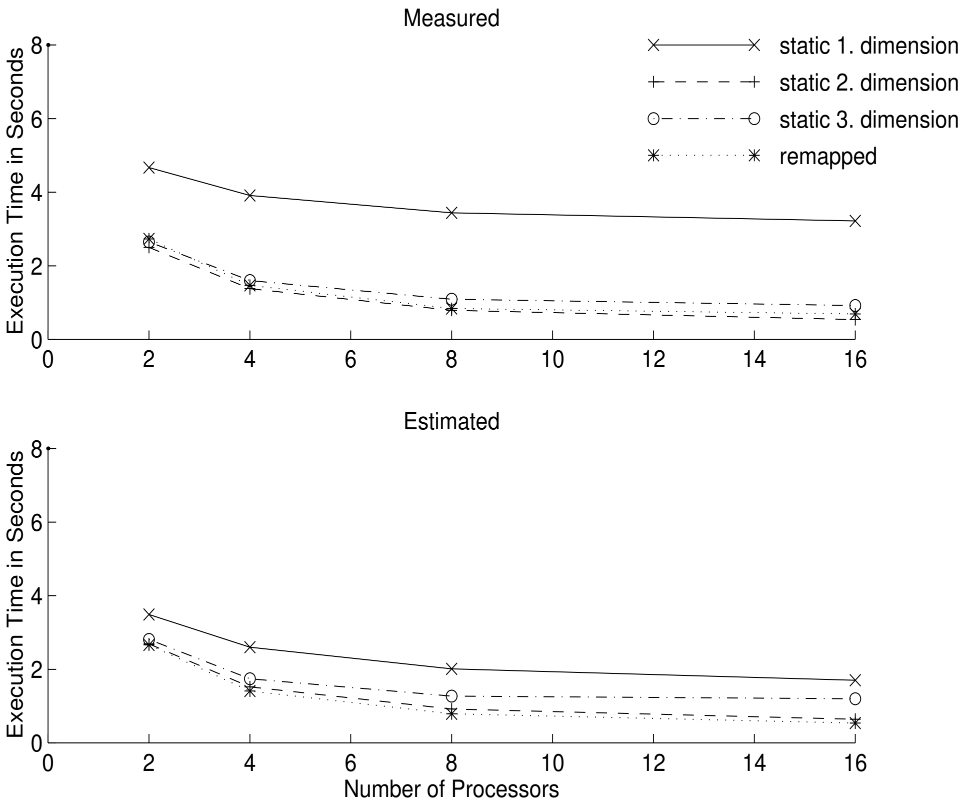


Figure 5.9 Measured and estimated execution times for Erlebacher with problem size $64 \times 64 \times 64$, real

Figure 5.9 shows the measured and estimated execution times for the test cases of problem size $64 \times 64 \times 64$ and data type real. Although the different candidate layouts are ranked correctly, the layout that distributes the third dimension is overestimated by up to 23% and the layout that distributed the first dimension is underestimated by up to 47%.

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.

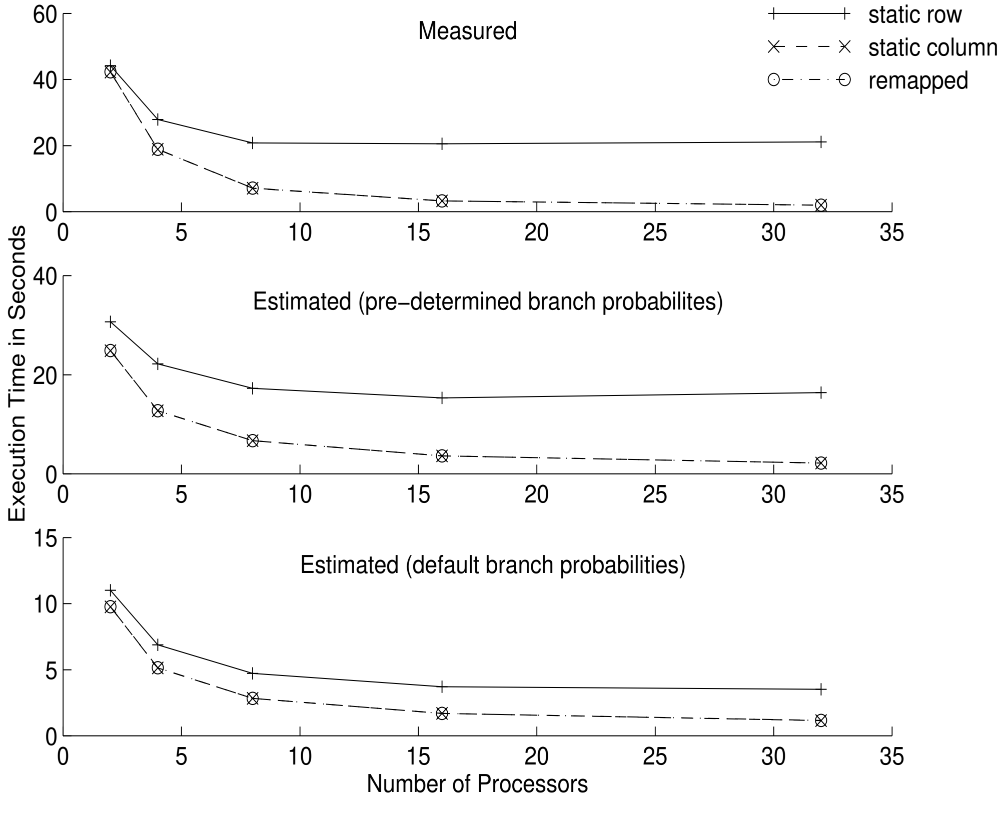


Figure 5.10 Measured and estimated execution times for Tomcatv with problem size 128×128, double precision (Note the different time scales)

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 18 test cases. In a single test case, a static row layout was the best choice. Distributing the second dimension was the best data layout selection for 12 cases. For the remaining 5 test cases, the dynamic data layout was the best choice. In all cases the prototype tool selected the column-wise data layout. The six suboptimal choices resulted in a performance degradation of at most 2.0% as compared to the best choice.

Figure 5.10 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 5.10 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 5.11 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.

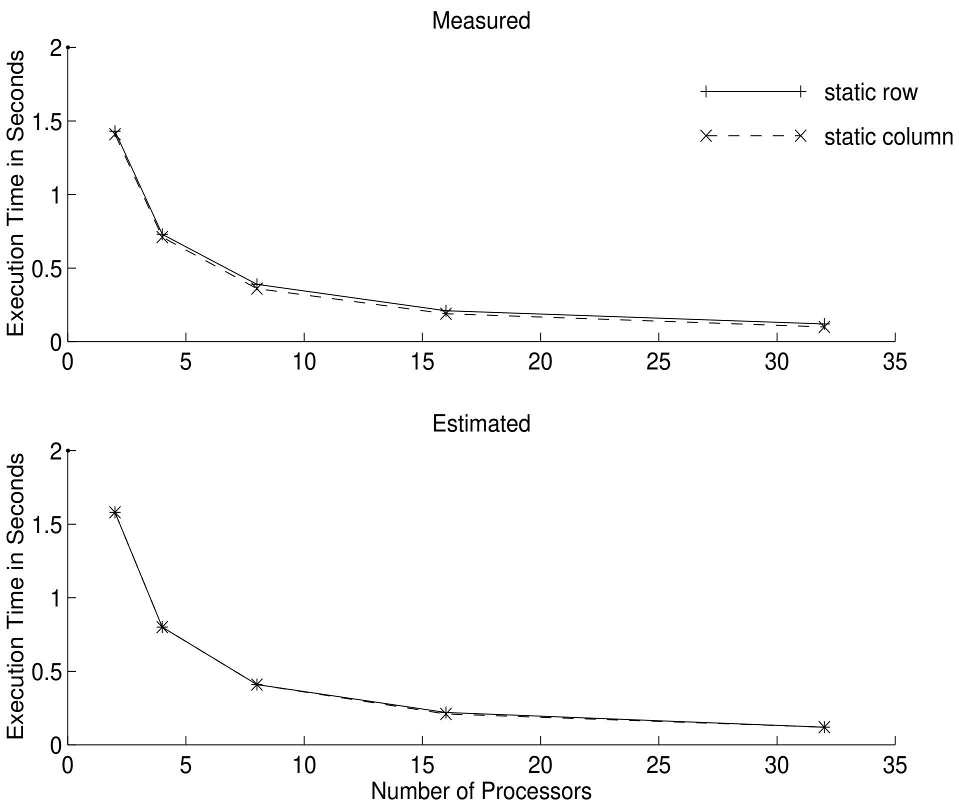


Figure 5.11 Measured and estimated execution times for Shallow with problem size 384×384, real

5.2.3 Discussion of Results

A total of 97 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, since the compiler performs the work mapping and inserts the necessary communication. In addition, the trade-off decisions depend on the performance characteristics of the target machine, the problem size, and the number of processors used.

Erlebacher and Adi are program examples where the best data layout choices depend on the problem size and number of processors used. Tomcatv contains branches inside its main iterative loop, making static performance prediction without predetermined branch probabilities difficult. However, the relative performance characteristics of the different layout choices were predicted with high accurracy. Shallow shows the precision of relative performance estimates for a program where reasonable data layout choices have nearly identical performance.

In 79 cases, the tool selected the optimal data layout. In the cases where the tool selected a suboptimal layout, the performance loss incurred was within 8.6% of the optimal layout. All encountered instances of the interdimensional 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.

Chapter 6

Conclusions and Future Work

Choosing a good 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. The trade-off decisions become even harder if dynamic remapping is allowed.

We have developed a new framework for automatic data layout for regular problems. The framework optimizes the data layout for a given problem size and number of available processors. The automatically generated data layouts may contain dynamic data remapping. Instead of resorting to heuristics, the framework 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.

We have implemented a data layout assistant tool based on our framework. The prototype implementation uses a static performance estimator consisting of an efficient compiler, execution and machine model. The machine model is based on our novel training set approach. Experiments showed that the prototype tool is efficient and generates data layouts of high quality.

6.1 Framework for Automatic Data Layout

Our framework divides the problem of automatic data layout into four well defined subproblems, namely program partitioning, candidate layout search spaces construction, performance estimation, and data layout selection. We presented an efficient, optimal solution to the data layout selection subproblem. For the other subproblems we discussed basic tools and methods that can be used to implement efficient solutions.

Most automatic data layout algorithms discussed in the literature can be supported in our framework. Each such algorithm has a specific strategy for the choice of candidate data layouts and for performance prediction, and uses a heuristic to determine the final, overall data layout. Implementing a data layout algorithm in our framework has the advantage that our final selection step uses an optimal solution instead of a heuristic. Therefore, the overall quality of the data layouts generated by an automatic data layout algorithm embedded in our framework will be improved.

Although the four subproblems in our framework can be solved independently, the overall precision and efficiency of an implementation based on the framework will depend on a balanced design of its solutions to the four subproblems. For instance, the search spaces construction algorithm and the performance estimator should be calibrated with respect to each other. The performance estimator has to be able to efficiently process every candidate layout that is generated during the search spaces construction step. In addition, if the performance estimator cannot distinguish the performance of two similar candidate data layouts, both layouts should not be inserted into the same search space. Other overall efficiency issues are related to the program partitioning algorithm. For instance, if it can be shown that remapping between two neighboring code segments can never be profitable, the phase definition should assign both segments to the same phase. Fewer phases will reduce the time needed for search spaces construction, performance prediction, and final candidate selection.

In the remainder of this section we discuss some of the unique features of our framework in more detail.

6.1.1 Fixed Problem Size and Number of Processors

Our experiments showed that the choice of a good data layout may depend on the problem size and the number of available processors. Optimizing a data layout for specific values of these entities is therefore desirable or even necessary.

A tool that allowed the specification of a range of problems sizes or numbers of processor could also be implemented based on our framework. The tool could apply our framework multiple times for different problem sizes and numbers of processors in the specified range. The resulting data layouts could be used to identify the data layout that has the best average performance for the specified ranges. In the case where no range is specified, the tool could either choose the ranges or just select an arbitrary problem size and number of processors.

6.1.2 Explicit Search Spaces Construction

Explicit candidate layout search spaces are the main interface between the last three subproblems in our framework. In addition, the search spaces provide the basis for an effective user interface. Such an interface may allow the user to browse through the search spaces, delete candidate layouts, or insert additional candidates.

The current prototype implementation uses a heuristic to construct alignment search spaces. Explicit alignment search spaces allow the postponement of tradeoff decisions between different alignments until the distributions are known. Distribution candidates are chosen from an exhaustive search space of one-dimensional distributions.

Exhaustive search space construction methods may not be efficient if block-cyclic and multi-dimensional distributions are considered. One possible approach involves an iterative search process based on multiple invocations of our framework. Initially, a coarse distribution search space is generated. Once the solution for the coarse spaces has been determined, the next iteration builds distribution search spaces that refine the distributions close to the optimal distributions in the previous solution. Note that such an approach is not guaranteed to find the optimal solution since it may converge in a local minimum. Experiments will be needed to verify the efficiency of the described approach.

We were not able to implement distribution search spaces for block-cyclic and multi-dimensional distributions as part of this thesis since this would have required the availability of efficient compilation and performance models for these distributions. At this point in time, such models are still a topic of active research in the compiler and parallel programming environment community.

6.1.3 Optimal or Near-Optimal Solutions to NP-complete Problems

We have shown that today's 0–1 integer programming technology combined with a careful 0–1 formulation of NP-complete problem instances can be used to solve these problems efficiently in practice. Data layout selection problems as large as 200 phases with a total of over 1200 candidate layouts and 13500 possible remappings were solved in a matter of minutes. To obtain this result we investigated different equivalent 0–1 formulations of the NP-complete data layout selection problem and the inter-dimensional alignment problem. Our experiments were based on CPLEX, arguably the most powerful 0–1 integer programming tool available today.

If the structure of a 0–1 problem instance is so complex that even the most advanced integer programming techniques fail to compute the optimal solution efficiently, or if the optimal solution is not desired, feasible solutions can be used as a basis for near-optimal approximations. Information about how closely a feasible solution approximates the optimal solution is available without additional cost due to the branch-and-bound nature of the solution process. Returning the best feasible solution found within a pre-specified time bound or returning the first feasible solution that is within some pre-defined epsilon of the optimal solution, are examples of heuristics that could be implemented easily.

Although we discussed polynomial time solutions for problem instances with special characteristics, we did not investigate algorithms that combine special purpose and general solution methods.

6.1.4 Static Performance Estimation

Performance estimation is crucial for any automatic data layout technique. We have developed a new approach to static performance estimation based on an efficient compiler, execution, and machine model. The compiler model simulates the compilation process at an abstract level. The machine model uses machine-level "training sets" to estimate the costs of basic computations and communication patterns.

Instead of using a general theoretical performance model, training sets probe the underlying system for its performance characteristics. The execution of a machine-level training set produces a cost function that is used by the execution model to estimate the costs of basic computations and communication patterns. Changes in the underlying hardware/software system can be handled by reexecuting the training sets on the target system. If a general theoretical model is used, changes to the underlying system will nearly always result in changes to the performance model itself, making the updating of the machine model difficult.

Although the use of training sets simplifies the task of performance prediction significantly, its complexity now lies in the design of the training sets themselves. Our current implementation uses over 100 machine-level training sets.

6.2 Experimental Results

We conducted experiments to determine the efficiency of our framework and the quality of data layouts it generates. The results show that our framework is efficient and generates data layouts of high quality.

The efficiency experiments were based on hand-generated data layout graphs for the 500 line program Erlebacher. The resulting data layout selection problems had 40 search spaces with a total of 253 candidate layouts and 2133 possible remappings. The optimal solution of each problem instance was computed in less than 4.8 seconds on a SPARC-10. We generated bigger problem instances by using multiple copies of Erlebacher. Data layout selection problems with 200 search spaces and a total of over 1200 candidate layouts and 13500 possible remappings were solved optimally in less than 6 minutes.

To determine the quality of the generated data layouts, we implemented a prototype data layout assistant tool. A total of 97 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. In 79 cases, the tool selected the optimal data layout. In the cases where the tool selected a suboptimal layout, the performance loss incurred was within 8.6% 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.

6.3 Future Work

6.3.1 Data Layout Selection

Our current framework does not allow replication of read-only data, i.e., it assumes that only a single copy of an array can exist at any time during program execution, unless the array is replicated due to multiple ownership. The data layout graph (DLG) construction algorithm can be modified to support read-only replication. The 0–1 formulation of the data layout selection problem has to be changed to deal with the additional remapping edges. Constraints over the remapping edges can be used to specify an upper bound on the number of possible read-only replications.

6.3.2 Interprocedural Analysis

In order to be able to evaluate our automatic data layout techniques for larger programs, the framework should be extended to handle programs that consist of multiple procedures. Multiple passes over the call graph may be necessary to perform program partitioning, search spaces construction, and DLG construction.

During the DLG construction for the entire program, DLGs of subroutines can be propagated bottom-up along edges in the call graph. If a procedure is cloned, each procedure clone is represented by a separate copy of the procedure's DLG. Otherwise, each procedure is represented by a single copy of its DLG in the DLG for the entire program.

6.3.3 Distributed Shared Memory Systems

In the longer term, we propose investigating the possibility of using our framework for automatic data layout selection for distributed shared memory systems (DSM). In a DSM system, the objects to be mapped are not user-level arrays, but operating system objects such as pages or cache lines.

6.3.4 0–1 Integer Programming

Encouraged by our experience with 0–1 integer programming, we propose to investigate 0–1 formulations of other NP-complete problems that occur in optimizing compilers and programming environments, such as register allocation, instruction scheduling, and parallel code generation. There will be several benefits of this research.

Translating practical instances of NP-complete problems into equivalent 0–1 formulations will allow a classification of the problems based on the structure of their 0–1 formulations. For some structures, advanced general purpose techniques will suffice to determine the optimal solution within an acceptable time bound. For other structures, the use of special purpose solvers, possibly combined with parallel processing may allow an efficient solution.

Bibliography

[ABCC93] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. The traveling salesman problem. 1993. In preparation.

[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.

[AG94] R. Govindarajan E. R. Altman and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, San Jose, CA, December 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.

[AGG95] E. R. Altman, R. Govindarajan, and G. R. Gao. Scheduling and mapping: Software pipelining in the presence of structural hazards. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995.

[AK87] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[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.

[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.

[AMCA+95] V. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J-C. Wang, and D. Reed. An integrated compilation and performance analysis environment for data parallel programs. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

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

[Bal90] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[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.

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

[Bix94] R. Bixby. Progress in linear programming. *ORSA Journal on Computing*, 6(1), 1994.

[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.

[BKK94] 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.

[Cal87] D. Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Rice University, March 1987.

[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 expressions 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.

[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.

[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.

[CMT94] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, October 1994.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran - a Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[DFJ54] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large scale traveling salesman problem. *Operations Research*, 7:58–66, 1954.

[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.

[Dyb87] R. K. Dybvig. *The Scheme Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[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.

[Fea94] P. Feautrier. Fine-grain scheduling under resource constraints. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.

[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.

[FZ93] T. Fahringer and H.P. Zima. A static parameter based performance prediction tool for parallel programs. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[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.

[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.

[GB92a] M. Gupta and P. Banerjee. Compile-time estimation of communication costs on multicomputers. In *Proceedings of the 6th International Parallel Processing Symposium*, Beverly Hills, CA, March 1992.

[GB92b] 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.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine.* The MIT Press, Cambridge, MA, 1994.

[Ger90] M. Gerndt. Updating distributed variables in local computations. *Concurrency—Practice & Experience*, 2(3):171–193, September 1990.

[GLS94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with Message-Passing Interface.* The MIT Press, Cambridge, MA, 1994.

[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.

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

[Hig93] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, May 1993. To appear in *Scientific Programming*, vol. 2, no. 1.

[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.

[HKTW94] S. Hiranandani, K. Kennedy, C.-W. Tseng, and S. Warren. The D Editor: A new interactive parallel programming tool. In *Proceedings of Supercomputing '94*, Washington, DC, November 1994.

[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.

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

[KK95] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. To appear. The paper is also available as technical report CRPC-TR94-498-S, Center for Research on Parallel Computation, Rice University.

[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.

[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.

[KR94] U. Kremer and M. Ramé. Compositional oil reservoir simulation in Fortran D: A feasibility study on Intel iPSC/860. *International Journal of Supercomputer Applications*, 8(2):119–128, Summer 1994. Also available as Technical Report CRPC-TR93335, Center for Research on Parallel Computation, Rice University.

[Kre93a] U. Kremer. Automatic data layout for distributed-memory machines. Technical Report CRPC-TR93-299-S, Center for Research on Parallel Computation, Rice University, February 1993. (thesis proposal).

[Kre93b] 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.

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

[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.

[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.

[MCAK94] John M. Mellor-Crummey, Vikram S. Adve, and Charles Koelbel. The Compiler's Role in Analysis and Tuning of Data-Parallel Programs. In *Proceedings of The Second Workshop on Environments and Tools for Parallel Scientific Computing*, pages 211–220, Townsend, TN, May 1994. Also available via anonymous ftp from softlib.cs.rice.edu in `pub/CRPC-TRs/reports/CRPC-TR94405.ps`.

[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.

[NG93] Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, Albuquerque, NM, January 1993.

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

[PB95] D. Palermo and P. Banerjee. Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. Technical Report CRHC-95-09, Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, April 1995.

[PFTV88] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C: the Art of Scientific Computing.* Cambridge University Press, 1988.

[PHHF94] M. Parashar, S. Hariri, H. Haupt, and G. Fox. Interpreting the performance of HPF/Fortran90D. In *Proceedings of Supercomputing '94*, Washington, DC, November 1994.

[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.

[PIJJ88] D.J. Pease, R. Inamdar, A. Joshi, and S. Jejurikar. Predicting the performance of a scalar program converted to execute on a vector processor. In *Proceedings of the 3rd ACM SIGSoft/SIGPlan Conference on Parallel Processing*, pages 355–361, August 1988.

[PR91] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33:60–100, 1991.

[PSCB94] D. Palermo, E. Su, J. A. Chandy, and P. Banerjee. Communication optimizations used in the PARADIGM compiler for distributed-memory multicomputers. In *Proceedings of the 1994 International Conference on Parallel Processing*, St. Charles, IL, August 1994.

[Pug91] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[Ree94] Daniel A. Reed. Experimental Performance Analysis of Parallel Systems: Techniques and Open Problems. In Gunter Haring and Gabriele Kotsis, editors, *Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 25–51. Springer-Verlag, May 1994.

[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.

[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.

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

[Tol95] S. Toledo. PERFSIM: A tool for automatic performance analysis of data-parallel Fortran programs. In *Frontiers95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 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.

[Wan93] K-Y. Wang. A framework for static, precise performance prediction for superscalar-based parallel computers. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[Wan94] K-Y. Wang. Precise compile-time performance prediction for superscalar-based computers. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, Orlando, FL, June 1994.

[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.

[WL91] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[Wol89] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

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

Appendix A

A.1 Correctness of 0–1 Alignment Problem Formulation

Assume that the CAG may contain nodes of arrays with different numbers of dimensions. Each edge represents an alignment preference, weighted by its importance. The inter-dimensional alignment problem is a node partitioning problem such that (1) nodes that represent the same array cannot be in the same partition, and (2) the sum of the weights of edges between distinct partitions is minimal. The correctness criteria for a solution of the inter-dimensional alignment problem over the CAG are:

- *(C1) :* The solution is a node partitioning of the CAG, i.e., each node belongs to exactly one partition.
- *(C2) :* Nodes representing distinct dimensions of the same array must not belong to the same partition.
- *(C3) :* The partitioning of the nodes has the property that the sum of the weights of all edges between nodes in different partitions is minimal.

The correctness criteria *C1* and *C2* are enforced by the node constraints. The edge constraints must make sure that the cost of an edge is considered in the objective function, if and only if both, the source and the sink of the edge are in the same partition. By maximizing the sum of all edge weights inside partitions, the sum of all edge weights crossing distinct partitions is minimized. Therefore, the correctness criterion *C3* holds. What remains to be shown is that the edge constraints formulation satisfies the following lemma:

> **Theorem A.1** Let $e = (a_i, b_j)$ be an edge in the CAG. For any solution of the 0–1 problem formulation the following holds:
>
> $$a\$b^{ik}_{jk} = 1 \quad \text{if and only if} \quad a_{ik} = 1 \text{ and } b_{jk} = 1,$$

where k is the index of an arbitrary partition, $1 \leq k \leq d$.

Proof
"$\Longrightarrow$": Assume $a\$b_{jk}^{ik}$ is switched on. Assume without loss of generality that the edge is directed from a_i to b_j. $a\$b_{jk}^{ik}$ occurs in exactly two constraints, namely the IN constraint for node b_j and partition k, and the OUT constraint for node a_i and partition k. The right hand side of the inequality constraints is b_{jk} and a_{ik}, respectively. Therefore, both a_{ik} and b_{jk} have to be switched on.

"$\Longleftarrow$": Assume a_{ik} and b_{jk} are both switched on. Assume without loss of generality that the edge is directed from a_i to b_j. $a\$b_{jk}^{ik}$ occurs in exactly two constraints, namely the IN constraint for node b_j and partition k, and the OUT constraint for node a_i and partition k. The right hand side of the inequality constraints is b_{jk} and a_{ik}, respectively. Due to the node constraints, all $a_{i'k}$ and $b_{j'k}$, $i' \neq i$ and $j' \neq j$, have to be switched off. This implies that all variables of the form $a\$b_{jk}^{i'k}$ have to be switched off since they occur in OUT constraints of switched off variables $a_{i'k}$. Therefore, in the IN constraint for CAG node b_j and partition k,

$$\sum_{a_{i'} \in SRC(a,b_j)} a\$b_{jk}^{i'k} \leq b_{jk} \ ,$$

only $a\$b_{jk}^{ik}$ may be switched on.

Since the solution of the 0–1 problem determines the maximal solution of the objective function and since all edge weights are greater than zero, $a\$b_{jk}^{ik}$ must be switched on. Assume $a\$b_{jk}^{ik}$ is not switched on in an optimal solution of the 0–1 problem. Since a_{ik} and b_{jk} are both switched on, $a\$b_{jk}^{ik}$ can be switched on, resulting in a solution that is greater than the optimal solution. This is a contradiction. □

A.2 Correctness of 0–1 Remapping Constraints

The three formulations of the remapping constraints introduce a distinct switch for each candidate data layout and each possible remapping between different candidate data layouts. The switches can either be turned on or off. The constraints have to ensure that the following three correctness criteria hold for any solution of the corresponding 0–1 integer programming problem:

- *(C1) :* Exactly one data layout is selected for each phase, i.e., only a single node switch is on for each phase.

- *(C2)* : Only edges between selected candidate data layouts are selected, i.e., an edge is switched on if and only if its sink and source nodes are switched on.

For all three formulations, the first correctness criterion is enforced by the layout constraints as discussed in Section 4.5.5. In the following we will show the validity of correctness criterion *(C2)* for the node-based and edge-based formulations.

Theorem A.2 The compact and disaggregated node-based formulations of the remapping constraints ensure the validity of the second correctness criterion *(C2)*.

Proof Assume that x_{ik}^{jl} is an edge with source node x_{jl} and sink node x_{ik}. We have to show that edge x_{ik}^{jl} is switched on iff x_{jl} and x_{ik} are both switched on.

"$\Longrightarrow$": Assume that x_{jl} or x_{ik} are switched off. Without loss of generality, assume that x_i^k is switched off. Since the edge x_{ik}^{jl} occurs in an IN constraint for node x_i^k, the edge has to be switched off as well.

"$\Longleftarrow$": Assume x_{jl} and x_{ik} are both switched on. Due to the layout constraints of phases i and j, no other nodes in these phases can be switched on. Therefore, the edge x_{ik}^{jl} is the only edge between nodes in phases i and j that may be switched on. In the disaggregated formulation, the IN constraint of node x_{ik} for phase j forces the edge x_{ik}^{jl} to be switched on. In the compact formulation, the right hand side of the IN constraint for node x_{ik} contains the factor $|P_i^{in}|$. Since each phase in P_i^{in} can only contribute a single edge that is switched on, an edge between phase j and i has to be switched on. Edge x_{ik}^{jl} is the only candidate for such an edge and therefore has to be switched on.

$\square$

Theorem A.3 The edge-based formulations of the remapping constraints ensures the validity of the second correctness criterion *(C2)*.

Proof Assume that x_{ik}^{jl} is an edge with source node x_{jl} and sink node x_{ik}. We have to show that edge x_{ik}^{jl} is switched on iff x_{jl} and x_{ik} are both switched on.

"$\Longrightarrow$": Assume x_{ik}^{jl} is switched on. Due to constraint $x_{ik} + x_{jl} \geq 2\ x_{ik}^{jl}$, $x_{ik} + x_{jl} \geq 2$ has to hold. This is only possible if both, x_{jl} and x_{ik}, are switched on.

"$\Longleftarrow$": Assume x_{jl} and x_{ik} are switched on. Due to constraint $x_{ik} + x_{jl} \leq 1 \ + \ x_{ik}^{jl}$, $x_{ik} + x_{jl} \leq 1 \ + \ x_{ik}^{jl}$, $1 \leq x_{ik}^{jl}$ has to hold. Therefore x_{ik}^{jl} must be switched on.

$\square$

A.3 Exhaustive Search Spaces for BLOCK Distributions

Lemma A.1 Let *procs* denote the number of processors used. Assume that the number of processors is a power of some prime number p multiplied by a prime number c, i.e. $procs = p^k * c$, with $c \neq p$. Assuming that at least one dimension is partitioned, the number of distribution schemes for a d-dimensional program template , *size*, is

$$
\begin{aligned}
size &= \sum_{i=1}^{d} \begin{pmatrix} d \\ i \end{pmatrix} * \begin{pmatrix} k-1 \\ i-1 \end{pmatrix} * i \ + \ \sum_{i=2}^{d} \begin{pmatrix} d \\ i \end{pmatrix} * \begin{pmatrix} k-1 \\ i-2 \end{pmatrix} * i \\
&= d \ * \ \begin{pmatrix} k+d-1 \\ d-1 \end{pmatrix}
\end{aligned}
$$

Proof The index i keeps track of the number of dimensions to be partitioned. The first sum gives the number of possible distribution schemes if each distributed dimension is assigned a number of processors that is a multiple of p. In terms of Figure 4.10, the additional factor c can end up in any of the i groups created by the partitioning. The second sum takes care of the case where a single dimension has exactly c processors assigned to it. This means that the p^k processors need to be partitioned into only $i-1$ groups, since the i-th group will consist of c alone. There are i possible positions for the group consisting of c with respect to the other $i-1$ groups.

The proof of the second part of the equation is as follows:

(1) :

$$
\textstyle\sum_{i=1}^{d} \begin{pmatrix} d \\ i \end{pmatrix} \ * \ \begin{pmatrix} k-1 \\ i-1 \end{pmatrix} \ * \ i \qquad \textstyle\sum_{i=1}^{d} \ \frac{d!}{i! \ (d-i)!} \ * \ \begin{pmatrix} k-1 \\ i-1 \end{pmatrix} \ * \ i
$$

$$
= \ \textstyle\sum_{i=1}^{d} \ \frac{d!}{(i-1)! \ (d-i)!} * \begin{pmatrix} k-1 \\ i-1 \end{pmatrix}
$$

$$= \; d \; * \; \sum_{i=1}^{d} \; \frac{(d-1)!}{(i-1)! \; (d-i)!} \; * \; \binom{k-1}{i-1}$$

$$= \; d \; * \; \sum_{i=1}^{d} \binom{d-1}{i-1} \; * \; \binom{k-1}{i-1}$$

$$= \; d \; * \; \binom{d-1}{d-1} \; * \; \binom{k-1}{d-1} \; + \; d \; * \; \sum_{i=1}^{d-1} \binom{d-1}{i-1} \; * \; \binom{k-1}{i-1}$$

$$= \; d \; * \; \binom{d}{d} \; * \; \binom{k-1}{d-1} \; + \; d \; * \; \sum_{i=1}^{d-1} \binom{d-1}{i-1} \; * \; \binom{k-1}{i-1}$$

(2) :

$$\sum_{i=2}^{d} \binom{d}{i} \; * \; \binom{k-1}{i-2} \; * \; i \; = \; \sum_{i=2}^{d} \; \frac{d!}{i! \; (d-i)!} \; * \; \binom{k-1}{i-2} \; * \; i$$

$$= \; \sum_{i=2}^{d} \; \frac{d!}{(i-1)! \; (d-i)!} * \binom{k-1}{i-2}$$

$$= \; d \; * \; \sum_{i=2}^{d} \; \frac{(d-1)!}{(i-1)! \; (d-i)!} \; * \; \binom{k-1}{i-2}$$

$$= \; d \; * \; \sum_{i=2}^{d} \binom{d-1}{i-1} \; * \; \binom{k-1}{i-2}$$

$$= \; d \; * \; \sum_{i=1}^{d-1} \binom{d-1}{i} \; * \; \binom{k-1}{i-1}$$

(1) $+$ **(2)** :

$$\begin{aligned}
\text{size} &= d * \binom{d}{d} * \binom{k-1}{d-1} + d * \textstyle\sum_{i=1}^{d-1} \binom{k-1}{i-1} * \left[\binom{d-1}{i-1} + \binom{d-1}{i}\right] \\
&= d * \binom{d}{d} * \binom{k-1}{d-1} + d * \textstyle\sum_{i=1}^{d-1} \binom{k-1}{i-1} * \binom{d}{i} \\
&= d * \textstyle\sum_{i=1}^{d} \binom{d}{i} * \binom{k-1}{i-1} \\
&= d * \binom{k+d-1}{d-1} \text{ (due to Lemma 4.2).}
\end{aligned}$$

□

The following table gives the values of *size* for different numbers of processors used and program templates of up to seven dimensions. The number of processors is assumed to be a power of two times a single number, i.e. $procs = 2^k * c$, where c is relative prime to two and no factorization of c is allowed.

#procs	#dimensions						
	1	2	3	4	5	6	7
$2 * c$	1	4	9	16	25	36	49
$4 * c$	1	6	18	40	75	126	196
$8 * c$	1	8	30	80	175	336	588
$16 * c$	1	10	45	140	350	756	1470
$32 * c$	1	12	63	224	630	1512	3234
$64 * c$	1	14	84	336	1050	2772	6468
$128 * c$	1	16	108	480	1650	4752	12012
$256 * c$	1	18	135	660	2475	7722	21021
$512 * c$	1	20	165	880	3575	12012	35035

Lemma A.2 Let d be constant. Then $size = \Theta(k^{d-1})$.

Proof Follows immediately from Lemma 4.3 and Lemma A.1. □

A.4 Experimental Results

This section contains the entire data generated for the quality experiments described in Section 5.2. Table entries are pairs of the form:

"<measured execution time> <predicted execution time>"

All timings are given in seconds. The best measured and estimated timings for a specific test case are printed in boldface. The entry "problem" indicates that the particular problem size and number of processors could not be handled by the Fortran D compiler prototype. An entry "memory" for a problem size and number of processors means that the node program executable did not fit into the memories of the node processors.

A.4.1 Adi

Static Row Layout

procs	128 × 128 real		256 × 256 real		384 × 384 real		512 × 512 real	
2	**2.492**	**2.470**	**9.254**	**9.069**	**20.284**	**19.798**	**35.577**	**34.659**
4	1.708	1.597	**5.467**	**5.248**	**11.374**	**10.964**	**19.377**	**18.745**
8	1.297	1.123	**3.487**	3.232	**6.718**	**6.374**	**11.004**	**10.549**
16	1.196	0.914	2.716	2.242	4.613	4.087	**7.061**	6.447
32	1.217	0.928	2.406	1.888	3.979	3.106	5.748	4.582

procs	128 × 128 double		256 × 256 double		384 × 384 double		512 × 512 double	
2	**3.274**	**3.407**	**12.132**	**12.806**	**27.340**	**28.198**	**48.545**	**49.584**
4	**2.082**	**2.071**	**6.937**	**7.127**	**14.865**	**15.180**	**25.824**	**26.229**
8	1.510	1.367	**4.198**	**4.186**	**8.451**	**8.503**	**14.164**	**14.318**
16	1.266	1.041	2.988	**2.729**	**5.411**	**5.166**	**8.571**	**8.352**
32	1.245	0.996	2.552	2.143	4.146	3.664	6.221	5.560

Static Column Layout

procs	128 × 128 real		256 × 256 real		384 × 384 real		512 × 512 real	
2	3.179	3.126	12.810	12.444	28.771	27.947	55.754	49.646
4	2.676	2.659	10.720	10.452	24.061	23.408	46.339	41.529
8	2.418	2.491	9.704	9.552	21.708	21.263	41.570	37.632
16	2.388	2.556	9.350	9.330	20.748	20.484	39.485	36.066
32	2.493	2.868	9.225	9.692	20.457	20.753	38.565	36.097

procs	128 × 128 double		256 × 256 double		384 × 384 double		512 × 512 double	
2	4.360	4.547	18.619	18.058	39.833	40.555	79.173	72.047
4	3.678	3.869	15.444	15.202	33.349	34.019	66.818	60.336
8	3.390	3.627	14.093	13.936	30.101	30.968	60.540	54.773
16	3.358	3.733	13.335	13.686	28.819	29.943	57.884	52.664
32	3.586	4.260	13.657	14.375	28.448	30.556	56.764	53.110

Dynamic Layout with Remapping

procs	128 × 128 real		256 × 256 real		384 × 384 real		512 × 512 real	
2	3.174	2.685	12.437	10.752	28.487	24.318	50.846	44.040
4	**1.704**	**1.466**	6.678	5.825	15.109	12.991	26.515	23.091
8	**0.961**	**0.806**	3.579	**3.081**	7.901	6.819	13.783	12.065
16	**0.712**	**0.510**	**2.017**	**1.791**	**4.143**	**3.631**	7.178	**6.271**
32	**0.775**	**0.477**	**1.471**	**1.339**	**2.475**	**2.301**	**4.150**	**3.658**

procs	128 × 128 double		256 × 256 double		384 × 384 double		512 × 512 double	
2	4.361	4.243	17.465	17.018	40.309	39.257	73.541	71.689
4	2.430	2.369	9.527	9.289	21.397	20.884	38.376	37.415
8	**1.354**	**1.330**	5.066	4.942	11.251	10.988	19.875	19.406
16	**0.908**	**0.879**	**2.781**	2.738	5.883	5.756	10.342	10.088
32	**0.909**	**0.884**	**1.871**	**1.836**	**3.430**	**3.389**	**5.643**	**5.559**

A.4.2 Erlebacher

Static Distribution of First Dimension

procs	32 × 32 × 32 real		48 × 48 × 48 real		64 × 64 × 64 real	
2	0.647	0.567	2.139	1.619	4.677	3.496
4	0.687	0.501	1.929	1.289	3.918	2.601
8	0.721	0.433	1.807	1.045	3.441	2.004
16	problem		problem		3.223	1.693

procs	32 × 32 × 32 double		48 × 48 × 48 double		64 × 64 × 64 double	
2	0.917	0.982	2.936	2.974	6.630	6.651
4	0.824	0.737	2.415	2.050	4.931	4.320
8	0.817	0.582	2.131	1.482	4.177	2.998
16	problem		problem		3.646	2.330

Static Distribution of Second Dimension

procs	32 × 32 × 32 real		48 × 48 × 48 real		64 × 64 × 64 real	
2	**0.319**	0.369	**1.019**	1.179	**2.503**	**2.699**
4	0.199	0.232	**0.581**	0.696	**1.380**	1.525
8	**0.146**	**0.165**	**0.368**	0.453	**0.785**	0.926
16	problem		problem		**0.537**	**0.644**

procs	32 × 32 × 32 double		48 × 48 × 48 double		64 × 64 × 64 double	
2	0.589	0.793	1.896	2.534	4.607	5.854
4	0.343	0.486	1.057	1.463	2.440	3.254
8	0.245	0.333	0.678	0.900	1.488	1.935
16	problem		problem		0.958	1.298

Static Distribution of Third Dimension

procs	**32 × 32 × 32** **real**		**48 × 48 × 48** **real**		**64 × 64 × 64** **real**	
2	0.337	0.381	1.038	1.217	2.651	2.813
4	0.213	0.263	0.638	0.782	1.597	1.744
8	0.163	0.224	0.456	0.602	1.093	1.273
16	problem		problem		0.920	1.206

procs	**32 × 32 × 32** **double**		**48 × 48 × 48** **double**		**64 × 64 × 64** **double**	
2	0.572	0.824	**1.805**	2.656	**4.382**	6.162
4	0.361	0.557	1.066	1.689	2.579	3.801
8	0.280	0.457	0.759	1.277	1.765	2.747
16	problem		problem		1.531	2.555

Dynamic Layout with Remapping

procs	**32 × 32 × 32** **real**		**48 × 48 × 48** **real**		**64 × 64 × 64** **real**	
2	0.321	**0.349**	1.036	**1.142**	2.729	2.706
4	**0.197**	**0.211**	0.584	**0.621**	1.473	**1.449**
8	0.159	1.694	0.400	**0.383**	0.845	**0.865**
16	problem		problem		0.690	0.686

procs	**32 × 32 × 32** **double**		**48 × 48 × 48** **double**		**64 × 64 × 64** **double**	
2	**0.567**	**0.742**	1.811	**2.461**	4.451	**5.820**
4	**0.313**	**0.415**	**0.977**	**1.304**	**2.373**	**3.055**
8	**0.233**	**0.278**	**0.633**	**0.739**	**1.415**	**1.693**
16	problem		problem		**0.952**	**1.128**

A.4.3 Tomcatv

The presented execution time estimates are based on a branch prediction heuristic that assigns the same probability to each branch.

Static Row Layout

procs	64 × 64 double		128 × 128 double		256 × 256 double		384 × 384 double	
2	7.939	3.181	44.215	11.020	**165.792**	40.890	memory	
4	8.294	2.348	27.891	6.899	98.823	23.148	191.006	48.903
8	9.462	1.941	20.810	4.728	66.504	13.882	96.587	27.775
16	10.377	1.824	20.543	3.712	49.863	9.289	70.684	17.233
32	problem		21.145	3.521	45.023	7.370	72.379	12.394

Static Column Layout

procs	64 × 64 double		128 × 128 double		256 × 256 double		384 × 384 double	
2	6.684	**2.517**	**42.231**	**9.757**	168.835	**38.407**	memory	
4	3.018	**1.451**	18.923	**5.142**	**85.593**	**19.686**	**168.240**	**43.742**
8	**1.724**	**0.924**	**7.095**	**2.841**	**43.229**	**10.279**	63.629	**22.454**
16	**1.058**	**0.649**	**3.248**	**1.693**	**19.281**	**5.587**	**25.393**	**11.811**
32	problem		**1.963**	**1.157**	7.672	**3.293**	**13.405**	**6.573**

Dynamic Layout with Remapping

procs	64 × 64 double		128 × 128 double		256 × 256 double		384 × 384 double	
2	**6.547**	2.521	42.354	9.774	176.864	38.482	memory	
4	**2.990**	1.452	**18.904**	5.153	85.809	19.734	memory	
8	1.750	0.925	7.136	2.848	43.362	10.305	**63.182**	22.513
16	1.069	0.650	3.257	1.698	19.336	5.599	25.513	11.839
32	problem		1.975	1.164	**7.591**	3.302	13.439	6.588

A.4.4 Shallow

Static Row Layout

procs	128 × 128 real		256 × 256 real		384 × 384 real		512 × 512 real	
2	0.172	0.182	0.643	0.712	1.426	1.587	memory	
4	0.093	0.097	0.346	0.366	0.731	0.807	**1.287**	1.423
8	0.055	0.055	0.189	0.193	0.390	0.416	0.675	0.727
16	0.036	0.034	0.106	0.107	0.213	0.221	0.371	0.380
32	0.026	0.024	0.067	0.064	0.126	0.124	0.206	0.206

Static Column Layout

procs	128 × 128 double		256 × 256 double		384 × 384 double		512 × 512 double	
2	**0.162**	**0.181**	**0.640**	**0.710**	**1.405**	**1.585**	memory	
4	**0.085**	**0.096**	**0.327**	**0.364**	**0.711**	**0.805**	1.310	**1.420**
8	**0.047**	**0.053**	**0.170**	**0.190**	**0.367**	**0.414**	**0.670**	**0.724**
16	**0.028**	**0.032**	**0.090**	**0.104**	**0.191**	**0.219**	**0.345**	**0.377**
32	**0.019**	**0.022**	**0.052**	**0.062**	**0.104**	**0.122**	**0.183**	**0.204**