Automatic Data Layout for Distributed-Memory Machines

KEN KENNEDY
Rice University
and
ULRICH KREMER
Rutgers University

The goal of languages like Fortran D or High Performance Fortran (HPF) is to provide a simple yet efficient machine-independent parallel programming model. After 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. If languages such as HPF are to find general acceptance, the need for data layout selection support has to be addressed. We believe that the appropriate way to provide the needed support is through a tool that generates data layout specifications automatically. This article discusses the design and implementation of a data layout selection tool that generates HPF-style data layout specifications automatically. Because layout is done in a tool that is not embedded in the target compiler and hence will be run only a few times during the tuning phase of an application, it can use techniques such as integer programming that may be considered too computationally expensive for inclusion in production compilers. The proposed framework for automatic data layout selection builds and examines 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 are needed to predict the execution time of each candidate layout and the costs of possible remappings between candidate data layouts. 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, the framework capitalizes on state-of-the-art 0-1 integer programming technology to compute optimal solutions of these NP-complete problems. A prototype data layout assistant tool based on our framework has been implemented as part of the D system currently under development at Rice University. The article reports preliminary experimental results. The results indicate that the framework is efficient and allows the generation of data layouts of high quality.

This research was supported by the Center for Research on Parallel Computation (CRPC), a Science and Technology Center funded by NSF through Cooperative Agreement Number CCR-9120008. This work was also sponsored by ARPA under contract #DABT63-92-C-0038. The content of this article does not reflect the position or the policy of the U.S. Government.
Authors' addresses: K. Kennedy, Department of Computer Science, Rice University, 6100 Main Street, Houston, TX 77005-1892; email: ken@cs.rice.edu; U. Kremer, Department of Computer Science, Hill Center, Busch Campus, Rutgers University, Piscataway, NJ 08855; email: uli@cs.rutgers.edu

Categories and Subject Descriptors: C.1.2 [**Processor Architecture**]: Multiple Data Stream Architectures (Multiprocessors)—*SIMD; MIMD*; D.3.4 [**Programming Languages**]: Processors—*compilers; optimization*; E.1 [**Data Structures**]: Arrays; G.1 [**Numerical Analysis**]: Optimization—*integer programming*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: High Performance Fortran, U. Kremer, Department of Computer Science, Hill Center, Busch Campus, Rutgers University, Piscataway, NJ 08855; email: uli@cs.rutgers.edu programming environments

1. INTRODUCTION

Compilers for data-parallel languages such as High Performance Fortran (HPF) [Koelbel et al. 1994] typically perform many optimizations based on the specified data layout. The actual optimizations performed may vary between compilers and different compiler optimization levels. Different parallel machine architectures have different communication and computation costs, resulting in distinct optimal balances between communication and computation for each architecture. On some machines and for some compilers, remapping data between different parts of an application program may be profitable. All these factors make it extremely difficult for a user to predict the performance impact of a particular data layout for an application program. Users, particularly users without previous exposure to HPF, will need help to port Fortran programs to an HPF environment. This article addresses the problem of automatic data layout in the context of a data layout assistant tool. We believe that such a tool is crucial, since it provides the support needed to make HPF generally accessible.

This article discusses a framework for automatic data layout for regular problems as part of a data layout assistant tool in a data-parallel programming environment such as the D system [Adve et al. 1994]. A data layout assistant tool can be used to determine an initial data layout for a sequential Fortran application. If the quality of the automatically generated layout is not satisfactory, static and dynamic performance analysis tools can help the user to understand the performance characteristics of the program. Once the user has chosen data layouts for program parts crucial for the overall performance, the layout assistant tool can be used to extend these data layouts to a data layout for the entire program. The application scenario for our proposed data layout assistant tool is shown in Figure 1, with HPF as the representative data-parallel language. Typically, regular problems represent data objects as dense arrays as opposed to a sparse representation. Regular problems allow compile time analysis to determine most of the communication requirements and to perform a variety of program optimizations at compile time. The automatically selected data layout may contain remappings if they are found profitable. Because the tool is not part of a compiler and will be run only a few times during the tuning phase of an application, techniques can be used that may be too computationally expensive to be used inside a compiler.

Our framework for automatic data layout has been designed to support experimentation with different data layout strategies in the context of a data layout assistant tool. The framework divides the problem of automatic data layout into

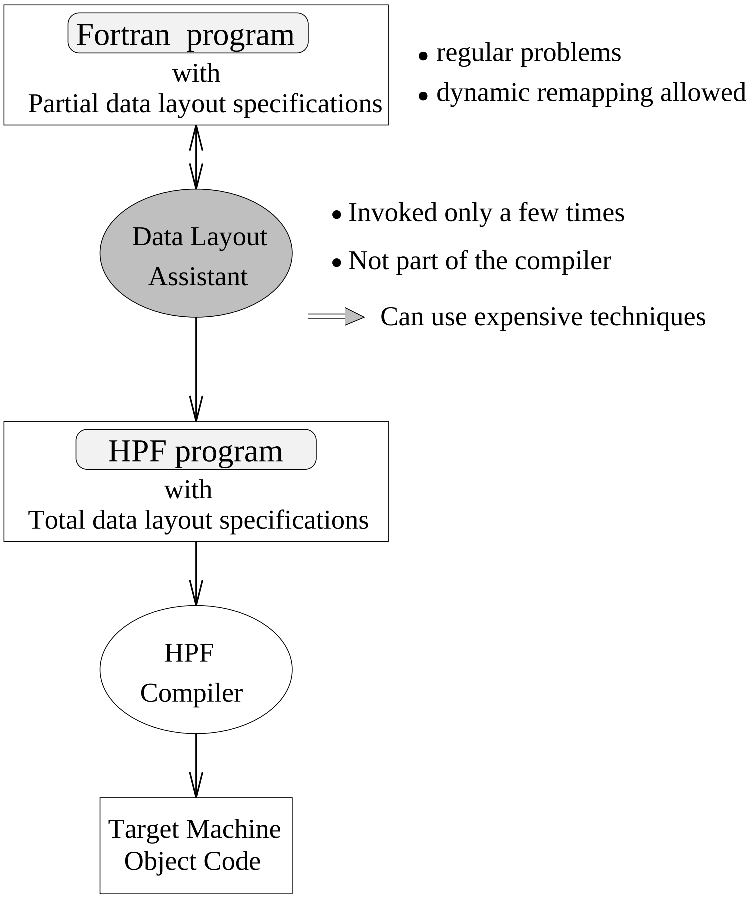


Fig. 1. Automatic data layout as part of a programming environment.

five consecutive steps: program partitioning, candidate-layout search spaces construction, performance prediction, data layout selection, and code generation. The framework defines the interfaces between these steps and provides infrastructure to support their solutions.

In the first step the input program is partitioned into program segments. Remapping is only allowed between 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 may be used to generate the candidate-layout search spaces. In the third step each candidate layout is evaluated in terms of its predicted execution time. In addition, costs of possible remappings between candidate layouts are determined. The performance prediction is based on a compiler model, execution model, and machine model. Given the predicted 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 step of our framework. Note that the optimal selection may consist of candidate data layouts that are each suboptimal for their phases. In the fifth and last step, HPF data layout specifications are inserted into the program that correspond to the computed candidate-layouts selection.

The framework is parameterized with respect to the HPF compiler, the machine architecture, the problem size, and the number of available processors. As a consequence, these entities have to be known at tool invocation time. Note that an automatically generated data layout can be used for different problem sizes and numbers of processors, although its quality may be suboptimal.

Actual implementations based on our framework may differ, for instance, in the way programs are partitioned in the first step, in the approach to candidate search spaces construction in the second step, or in the accuracy and cost of the used performance prediction techniques in the third step. However, the data layout selection and code generation steps are the same for all implementations. We show that the data layout selection step involves the solution of an NP-complete problem as described in Appendix B. Rather than resorting to heuristics, the framework capitalizes on 0-1 integer programming technology to compute optimal solutions of this problem and a second NP-complete problem that may occur during candidate search space construction. This article presents preliminary experimental results that indicate that optimal solutions can be computed efficiently. Computing optimal solutions for the data layout selection step implies that for any framework implementation the quality of the generated overall data layout depends only on the choices made during the first three steps of the framework, allowing a better understanding of the trade-offs between possible choices through experimentation. It is important to note that for any given implementation, using optimal techniques cannot compensate for poor design choices in the first three steps of the framework. For instance, using performance prediction techniques that are not able to distinguish the candidate data layouts in the search spaces will make an optimal solution to the data layout selection problem rather pointless, even though its computation may be efficient.

This article discusses an efficient formulation of the data layout selection problem as a 0-1 integer programming problem. In order to assess the overall design and efficiency of our framework, a prototype data layout assistant tool based on the framework has been implemented. Experiments for four benchmark programs using a single target compiler and target architectures showed that the prototype is efficient and allows the generation of data layouts of high quality.

The remainder of the article is organized as follows. A short introduction to data layout directives supported in HPF is given in Section 2. An overview of the first four steps is given in Section 3. A prototype implementation based on the framework is described in Section 4. The results of applying the prototype to four application programs are discussed in Section 4.3. The article concludes with a discussion of related work, followed by a summary and discussion of future work in Section 5 and Section 6, respectively.

2. HPF DATA LAYOUT DIRECTIVES

This section contains a short introduction to data mappings supported in HPF. A complete and detailed discussion of HPF data layout directives can be found elsewhere [Koelbel et al. 1994].

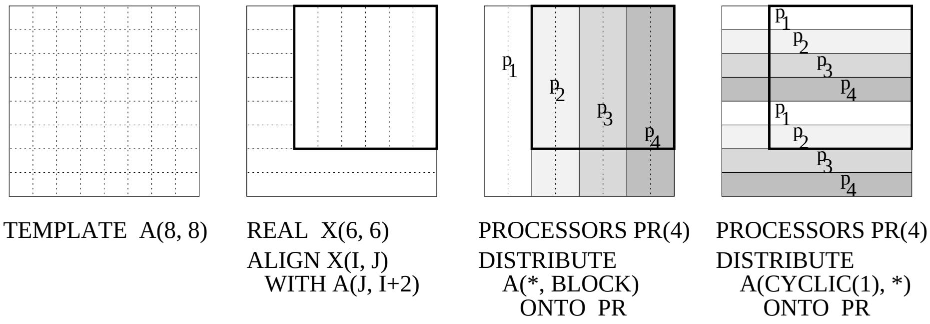


Fig. 2. HPF data layout specifications.

The task of distributing data across processors can be approached by considering two levels of data mappings. First, there is the question of how arrays should be *aligned* with respect to one another, both within and between array dimensions (problem mapping). Second, there is the question of how aligned arrays should be *distributed* onto the actual parallel machine (machine mapping). The alignment of arrays for a computation represents the minimal requirements for reducing data movement for a computation, i.e., aligning two array elements guarantees that both elements will be mapped to the same processor for all possible distributions. However, array elements that are not aligned may be mapped to the same processor as well depending on the subsequently selected distribution. Alignment is largely independent of any machine considerations. The choice of a distribution is typically affected by the topology, communication mechanisms, size of local memory, and number of processors of the underlying target machine. The distribution of arrays in the program depends on the coarse-grain parallelism defined by the target parallel machine and on load-balancing considerations for the application. It is important to note that only the combination of alignment and distribution will determine the necessary communication and exploitable parallelism in the program.

HPF provides data layout specifications for these two levels of data mapping using `TEMPLATE`, `ALIGN`, `DISTRIBUTE`, and `PROCESSORS` statements. A template is an abstract problem or index domain used as a target for alignments. A template does not require any storage. 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 (*intradimensional alignment*) or across dimensions (*interdimensional alignment*). Interdimensional alignment specifies how an array dimension or array axis is mapped to the dimensions or axes of the target template. For example, interdimensional alignment may specify that the transpose of a two-dimensional array is mapped onto a two-dimensional template $[array(i,j) \mapsto template(j,i)]$ or that a one dimensional array is embedded within a two-dimensional template $[array(i) \mapsto template(i,d)]$, where d is an integer constant. Intradimensional alignment specifies the mapping of array elements to template elements within an aligned dimension. Common forms of intradimensional alignment are *offset alignment* $[array(i) \mapsto template(i+d)]$ or *stride alignment* $[array(i) \mapsto template(e*i)]$, where d and e are integer constants. In a *canonical* offset and stride alignment, the offset is 0, and the stride is 1 [$array(i) \mapsto template(i)$]. 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-2,N-2)
!HPF$   TEMPLATE A(N,N)
!HPF$   ALIGN X(I,J) WITH A(J,I+2)
```

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 an offset within the second dimension of A.

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)
!HPF$   PROCESSORS PR(4)
!HPF$   DISTRIBUTE A(*,BLOCK) ONTO PR
!HPF$   DISTRIBUTE A(CYCLIC(1),*) ONTO PR
```

distributing template A by `(*,BLOCK)` results in a column partition of A and induces a corresponding distribution of all arrays aligned to A. Alternatively, distributing A by `(CYCLIC(1),*)` distributes the rows of A in a round-robin fashion among the processors. These sample data alignment and distributions for the case `N=8` are shown in Figure 2. If load-balancing issues dominate the overall execution time, the `CYCLIC` distribution may perform better than the `BLOCK` distribution, since the latter distribution does not map any elements of A to the first processor p_1 and therefore keeps p_1 idle.

HPF supports dynamic remapping, i.e., the data layout of arrays can change during the execution of the program. Once an array has been declared as `DYNAMIC`, `REALIGN` and `REDISTRIBUTE` statements can be used to specify remappings. In contrast to their "static" versions, realignment and redistribution statements are executable statements and not declarative. However, remapping can also be introduced by explicitly specifying "static" copies of an array, each with a different data layout. A simple assignment statement involving two copies will lead to the desired remapping.

We should note that the goal in designing HPF is not to support the most general data layouts possible. Instead, the intent is to provide data mappings that are both powerful enough to express data parallelism in scientific programs, and simple enough to permit the compiler to produce efficient programs. Typically, HPF compilers use some variant of the *owner computes rule* to generate a computation mapping for the specified data layout. The rule states that if a data item is mapped to a processor, i.e., the processor owns the data item, it will perform any computation that will assign a value to the data item [Callahan and Kennedy 1988; Rogers and Pingali 1989; Zima et al. 1988]. The design of HPF [Koelbel et al. 1994] has been influenced by languages such as Fortran D [Fox et al. 1990], CM Fortran [Thinking Machines Corp. 1989], C∗ [Rose and Steele 1987], Kali [Koelbel and Mehrotra 1991], and Vienna Fortran [Chapman et al. 1992].

3. FRAMEWORK FOR AUTOMATIC DATA LAYOUT

The framework for automatic data layout consists of five steps, namely program partitioning, candidate-layout search spaces construction, performance prediction, data layout selection, and code generation. In the remainder of this section each of the first four steps is discussed in more detail. The final code generation step is straightforward and therefore is not discussed any further. A small program kernel will serve as an example to illustrate each of the four steps.

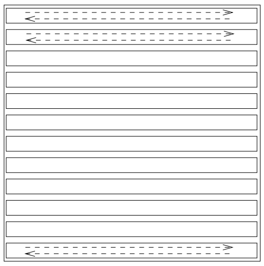
3.1 ADI Example Program

The following example illustrates the framework for automatic data layout. Figure 3 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 speed of the communication hardware and software of the target distributed-memory machine, and the ability of the compiler to exploit the available parallelism efficiently. For example, for a static rowwise data layout the compiler has to insert nearest-neighbor communication during the column sweeps. Two commercially available HPF compilers (IBM's xlhpf and PGI's pghpf) use different communication generation strategies for the IBM SP–2 multiprocessor [Presberg 1996]. For a nearest-neighbor *shift* communication pattern, PGI's pghpf compiler generates a single call to a global communication routine that implements the pattern. In contrast, IBM's xlhpf compiler generates explicit calls to lower-level send and receive communication routines for the pattern. Using a single global communication routine allows runtime communication optimization, since the decisions regarding execution order and type of lower-level send and receive routines may be postponed until the global routine is actually called. Inserting send and receive communications directly into the program allows compile time optimizations such as message latency hiding by moving sends ahead of receives across program segments. However, runtime optimizations are more restricted, since the placement and type of communication routines are determined at compile time. The PGI and IBM compilers also differ in the sizes of messages that

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

// loop nest performing READ (c, a, b) operation

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

// loop nest performing WRITE (c, b) operation
```



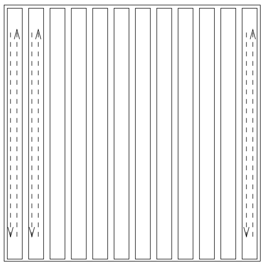


Fig. 3. ADI integration kernel with computation illustration.

are generated for the *shift* communication pattern. The PGI compiler uses the declared size of an array to determine the sizes of nearest-neighbor messages while the IBM compiler only communicates the actual data needed. As a result, the IBM's compiler generates smaller messages. In addition to the compiler dependence, the performance characteristics of the underlying I/O system should be considered in the data layout choice, 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 shows two possible data layout choices. The left-hand side shows a static, rowwise data layout. The right-hand side depicts a dynamic data layout where transpose operations will be performed between the row and column sweeps. As will be discussed in Section 4.3, our prototype data layout assistant tool generates a static or dynamic data layout for ADI depending on the data type of the arrays, number of available processors, and problem size.

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

// Static rowwise 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 columnwise 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
      . . .
```

Fig. 4. Two possible data layouts for the ADI integration kernel. The left-hand side shows a static rowwise data layout, and the right-hand side shows a dynamic layout that performs transposes between the sweeps along rows and columns.

3.2 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 identifies 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 framework supports many different phase definitions. The following phase definition has been used in our prototype implementation that is described in Section 4. Although the definition has its limitations it worked well on all programs in our test suite of Fortran 77 programs and has been adopted by other researchers [Palermo and Banerjee 1995].

Definition 3.2.1. 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.

A phase is a central concept in our current framework. All subsequent steps in the framework rely on the given decomposition of the program into phases, in particular the performance prediction and data layout selection steps. Performance prediction may model compiler optimizations and predict partial parallelism within phase boundaries. Parallelism resulting from overlapped phase executions is currently not modeled by the data layout selection step. Other strategies for identifying program phases are a topic of current research. For instance, the current phase definition may represent an entire group of loop nests as a single phase although remapping between the loops nests may be profitable. This case can occur when an outermost loop induction variable is used in an array subscript expression. The current phase definition may also combine too few loop nests within a single phase.

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 [Sheffler et al. 1996]. Transformations to influence phase recognition, for instance by aggressively distributing all loops in the program before the program partitioning step, are beyond the scope of this article.

The phase structure of the program is represented in the *phase control flow graph* (PCFG), an augmented control flow graph [Aho et al. 1986] where each phase is represented by a single node. The graph is annotated with branch probabilities and loop control information. Branch probabilities can either be supplied by the user or determined based on a guessing heuristic.

Figure 5 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.

3.3 Layout Search Spaces Construction

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

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

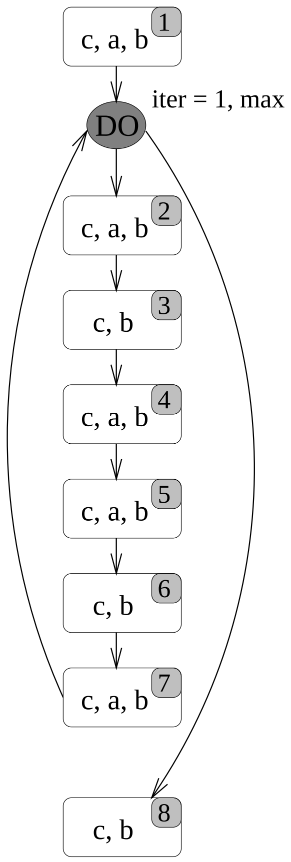
The use of search spaces is an important design decision in our framework for automatic data layout. Alignment search spaces allow the framework to postpone the evaluation of an alignment candidate until all distribution candidates are known. Therefore, promising alignment candidates are not eliminated prematurely, but are evaluated in combination with the selected candidate distributions. Only the combination of an alignment and a distribution determines the final communication requirements and exploitable parallelism for a program.

The current framework uses a single template for the entire program. All alignments and distributions are specified relative to this *program template*. Corresponding to the two-stage mapping, the framework supports the construction of the search spaces of promising candidate layouts through an alignment analysis step, followed by distribution analysis. Alignment and distribution analyses are discussed in the next two sections.

3.3.1 *Alignment Analysis.* Alignment analysis takes the phase control flow graph as input and generates alignment search spaces for each phase. Different strategies may be used to determine a reasonably sized set of alignment candidates that will guarantee a good overall performance for most applications. Alignment analysis may be 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* [Anderson and Lam 1993].

Our framework provides basic operations for interdimensional alignment analy-

```
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

Fig. 5. Result of the program partitioning step.

sis. Infrastructure to support intradimensional alignment analysis is currently not available. Interdimensional alignment is considered more important, since a potential misalignment can lead to global communication, e.g., a transpose operation. In contrast, missing an intradimensional alignment such as offset alignment may only result in nearest-neighbor communication. Typically, intradimensional alignment is performed after interdimensional alignment. The simplest form of intradimensional alignment does not perform any analysis, but uses canonical offset and stride alignments for all aligned array dimensions.

The basic operations for interdimensional alignment supported in our framework can be used 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. These basic operations and methods form the building blocks for implementing different strategies and heuristics for the alignment search space construction. The heuristic implemented in our prototype tool is discussed in Section 4.2.

3.3.1.1 *Identification and Representation of Relative Alignment Preferences.* A central representation for the relative, interdimensional alignment problem is the weighted, undirected *component affinity graph* (CAG) introduced by Li and Chen at Yale University [Li and Chen 1991b]. It represents the alignment preferences of arrays that are coupled in a computation. A d-dimensional array is represented in the CAG by d nodes, one node for each dimension. Alignment preferences between dimensions of distinct arrays are represented as edges between the corresponding nodes. The weights of the edges reflect the relative importance of alignment preferences. Typically, an edge weight represents the expected performance penalty if the corresponding alignment preferences is not satisfied. Therefore, methods to determine edge weights are based on some performance model. Our framework is independent of the actual performance model used. The alignment analysis performance model in our prototype implementation is discussed in Section 4.1.

3.3.1.2 *Detection of Relative Alignment Conflicts.* Assume that d is the dimensionality of the program template. A solution to the interdimensional 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 test for alignment conflicts is linear in the size of the CAG, since it involves solutions of reachability problems between nodes in the CAG.

3.3.1.3 *Relative Alignment Conflict Resolution.* A conflict implies that every solution of the corresponding interdimensional 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 interdimensional alignment problem is NP-complete [Li and Chen 1991b]. Instead of using a heuristic, the current framework formulates the interdimensional alignment problem as an efficient 0-1 integer programming problem. The 0-1 formulation is given in Appendix A. A similar 0-1 formulation for another NP-complete problem in our framework is discussed in more detail in Section 3.5.

3.3.1.4 *Comparison of Relative Alignment Preferences.* The interdimensional alignment information of a conflict-free CAG can be represented as a partitioning of its nodes. Each partition in the partitioning of a conflict-free CAG is a connected component in the CAG. The set of all possible conflict-free, interdimensional alignments of a set of arrays forms a semilattice [Hecht 1977]. 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 6 shows an example lattice for two two-dimensional arrays.

The partial order $\sqsubseteq$ defined over the set of conflict-free CAGs is that of partition-

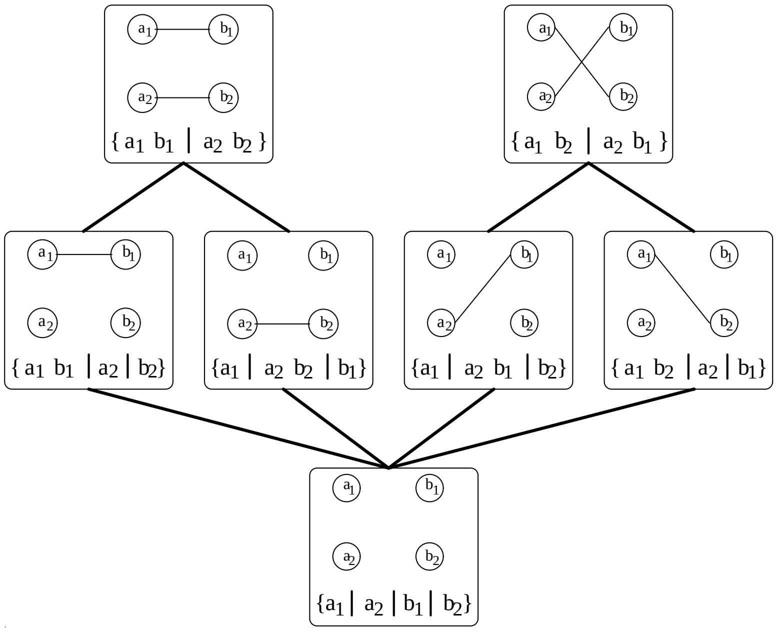


Fig. 6. Interdimensional alignment information lattice for two arrays a and b. Both arrays have two dimensions. Each conflict-free CAG is shown with its node partitioning. The bottom element of the lattice is the CAG without edges, i.e., the partitioning $\{a_1 \ |a_2 \ |b_1 \ |b_2\}$.

ing 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 assuming the elements in X and Y are tagged with their partition membership, then the test "X is refinement of Y" is linear in the average case 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 the average case in the number of nodes in CAG_1.

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

3.3.1.5 *Orientation Selection.* A conflict-free CAG represents relative interdimensional alignment preferences of arrays. A final stage is needed to determine the orientation of the CAG relative to the unique program template. An orientation of a conflict-free CAG maps the CAG's connected components, i.e., the sets of aligned array dimensions to the dimensions of the program template. For a d–dimensional program template and a conflict-free CAG with d connected components, there are $d!$ possible orientations.

Any orientation of a conflict-free CAG satisfies the alignment preferences represented in the CAG. However, in the presence of dynamic realignment, the orientation of two distinct CAGs may influence the potential remapping costs between the two alignments. Therefore an algorithm is needed that matches the orientations of the CAGs in the alignment search spaces as closely as possible. Anderson and Lam propose a greedy strategy to determine orientations [Anderson and Lam 1993]. We have developed a similar strategy based on meet operations over the lattice of conflict-free CAGs [Kremer 1995]. A meet operation of two CAGs identifies common alignment preferences represented in the CAGs. The idea is to map array dimensions that are part of the common alignment preferences to the same template dimensions. Evaluating different strategies for orientation selection is still part of ongoing research.

3.3.2 *Distribution Analysis.* Distribution analysis is performed after alignment analysis. A candidate distribution may map single template dimensions either by block, cyclic, or block–cyclic onto the target architecture, or replicate dimensions on each processor. In addition, a candidate distribution specifies the number of processors in each distributed dimension. Different strategies and 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 search spaces of candidate data layouts for each phase.

The heuristics for the construction of candidate distribution search spaces can be roughly divided into two classes, *exhaustive* and *constructive*. The sizes of some exhaustive search spaces are discussed in [Kremer 1995]. Exhaustive heuristics approximate the exhaustive set of all possible distributions of the program template. An approximation is a subset of candidate distributions that can be considered a representative sample of the exhaustive set. Constructive heuristics may consider load-balancing issues and the actual alignments in the alignment search spaces.

The quality of the performance estimator in the framework determines a bound on the granularity of the distribution search space. If the estimator is not able to distinguish two similar distributions, then only a single representative distribution should be added to the search space. The framework is not restricted to a particular heuristic for distribution search space construction. For instance, due to restrictions with respect to our target compiler, our prototype implementation uses a distribution search space restricted to all possible one-dimensional block distributions.

3.4 Performance Prediction

After the generation of the search spaces, each candidate data layout must be evaluated in terms of its expected execution time for its phase. In addition to the phase execution times, execution time predictions are needed for possible remappings between candidate data layouts. The goal of the performance estimator is not to give *absolute* performance predictions for different candidate data layouts, but to rank candidate data layout choices correctly according to their performance, i.e., to predict their *relative* performance. 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 required that can target different compilers and architectures.

A performance estimator suitable for our framework has been implemented as part of our data layout assistant prototype. The estimator uses a compiler model to determine where and what kind of communication will be generated for a given candidate data layout and its phase. The compiler model is parameterized with respect to the transformations and communication optimizations that may be performed by the target compiler. The compilation process needs to be simulated for performance purposes only. Special cases that have a small impact on the the overall performance, but must be handled by a real compiler in order to generate correct code, may be ignored in the performance estimation compiler model. For instance, the compiler model implemented as part of our prototype ignores code that is generated for boundary processors in loops. The implemented compiler model is discussed in Section 4.

Once locations and types of compiler-generated communications are known for a candidate layout and its phase, an execution model is used to predict the performance effects of synchronizations induced by the communications. Communication inside a phase may lead to a reduction or pipelined execution of the loop. In contrast, communication outside of the phase may result in a loosely synchronous execution scheme [Fox et al. 1988]. The execution model relies on a machine model to provide the costs for basic communication and computation patterns.

The machine model implemented in our prototype uses the training set approach [Balasundaram et al. 1991; Hiranandani et al. 1991]. 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 execution model in the performance estimator. By utilizing training sets, the performance estimator achieves both accuracy and portability across different target compilers and machine architectures.

3.5 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.

The general data layout selection problem as formulated in Section 3.5.1 is NP-complete. The proof can be found in Appendix B. In Section 3.5.3 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 3.5.4.

3.5.1 *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 ongoing research [Kremer 1998]. 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 [Aho et al. 1986] and reaching decomposition problems [Tseng 1993]. Phases from which data layouts can be inherited are determined based on the solution of the *most recently referenced* (MR–Refs) data flow problem. A reference is a pair consisting of an array name and a phase that references the array. MR–Refs(x) denotes the set of all most recent references that can reach phase x. The data flow flow equation for MR–Refs for a phase b has the following form:

$$\text{MR–Refs}(b) = \bigcup_{x \in pred(b)} (\text{Refs}(x) \cup (\text{MR–Refs}(x) - \text{Kill–Refs}(x)))$$

where Refs(x) are the local references for phase x and where Kill–Refs(x) contains all references of arrays outside of x that are also referenced locally in x. Each reference in MR–Refs(b) is associated with a probability that it will actually reach phase b. If a reference to array a in phase p_1 is the MR–reference of a for phase p_2, and phase p_2 references a, then the DLG will contain remapping edges from all candidate layouts in phase p_1 to all candidate layouts in phase p_2. Note that for each array and control flow path that reaches a phase there is a unique MR–reference of the array.

3.5.2 *Example.* The example ADI program kernel introduced in Section 3.1 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 4.2. 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 7. 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 predicted 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

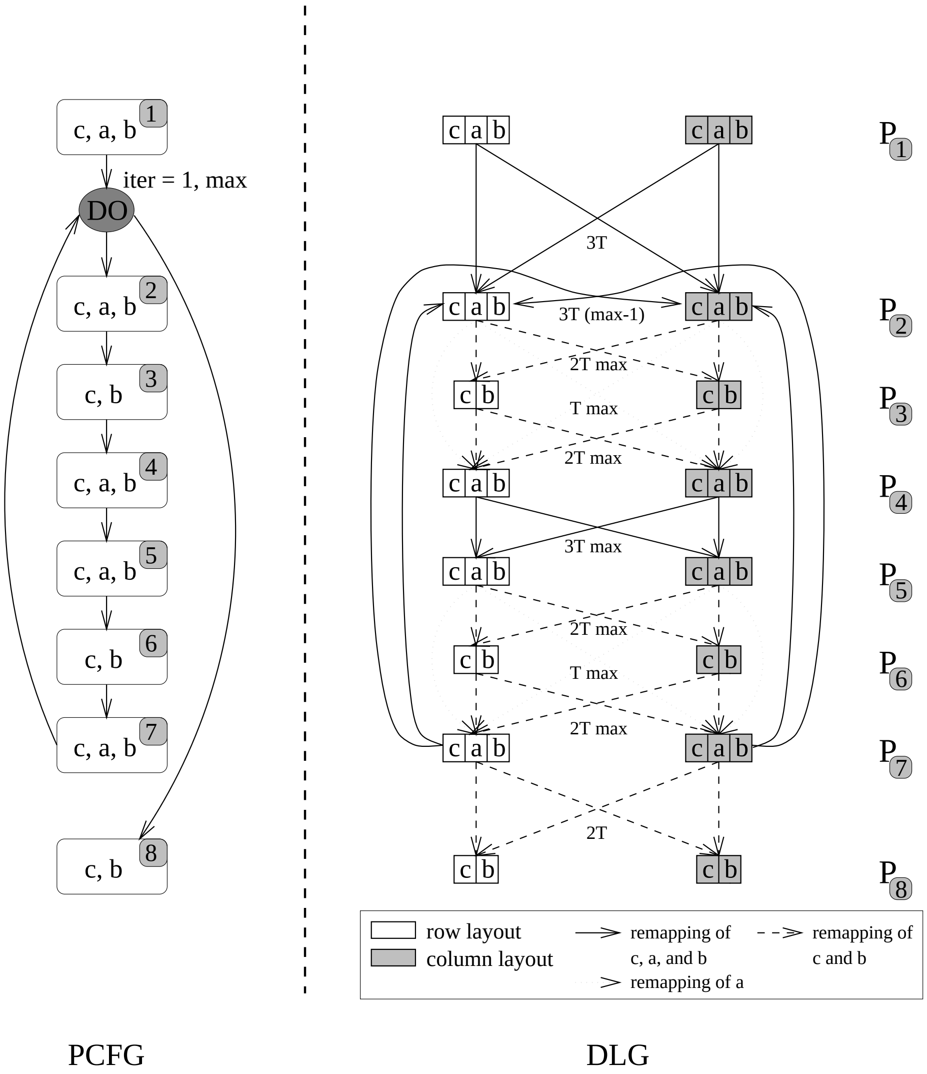


Fig. 7. PCFG and DLG for the ADI kernel example. To simplify the example, we assume that there are only two candidate data layouts in each search space. Weights in the DLG represent static performance prediction of overall execution times. Node weights are not shown. Unlabeled edges have zero weight. `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.

denoted by `T`, and `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 values of `T` and `max` are known. As shown in Appendix B, the general data layout selection problem is NP-complete.

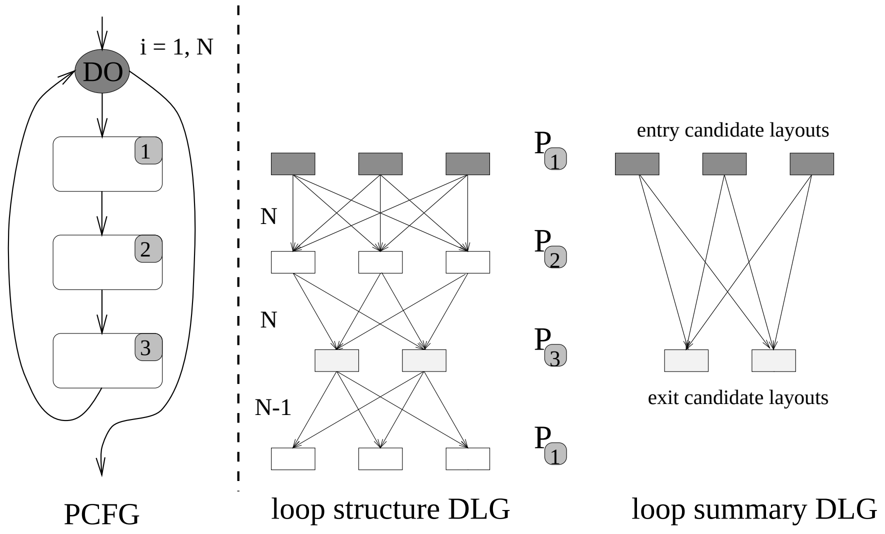


Fig. 8. Example loop summary DLG. The layout search spaces for the first two phases in the loop body are assumed to have three candidate layouts. The search space for the third phase is assumed to have two layouts.

3.5.3 *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 assume 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. Our framework assumes that data layout candidates for a phase specify only the layout of arrays referenced in the phase. Therefore, the polynomial algorithm can only be applied if every program phase contains a reference to every array in the program. The polynomial-time algorithm may also be applied if candidate layouts specify the data mappings for all program arrays even though only a small subset of the arrays may actually be referenced in each phase. Recent work done by Damian-Jordache and Pemmaraju at the University of Iowa illustrates the trade-offs between using a polynomial time algorithm and the sizes of the data-layout search spaces for each phase [Damian-Iordache and Pemmaraju 1998].

The main idea of our polynomial-time solution is to decompose the data layout selection problem into subproblems that can be solved as single-source shortest-path problems using dynamic programming such as, for instance, a variant of Dijkstra's algorithm for directed acyclic graphs as described in Cormen et al. [Cormen et al. 1990]. 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

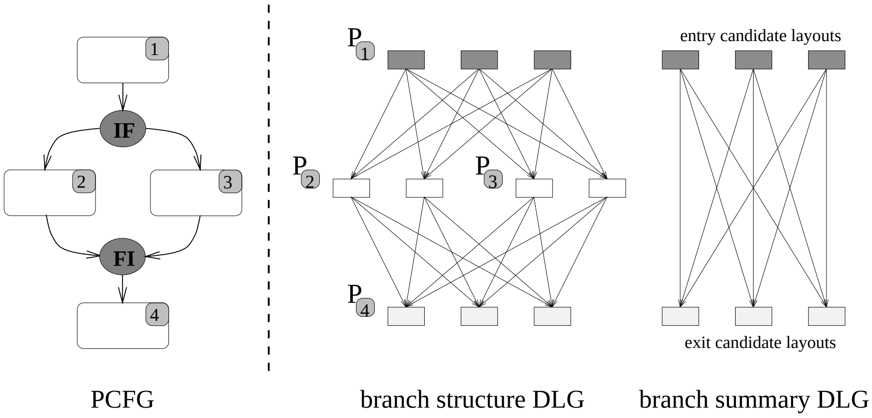


Fig. 9. Example branch summary DLG. The layout search spaces for the entry and exit phase are assumed to have three candidate layouts. The phases in the true and false branches are assumed to have two candidate layouts in their search spaces.

program are 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 DLG represents its associated 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 8. 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.

The time complexity of Dijkstra's algorithm for directed acyclic graphs is linear in the number of nodes and 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 nodes in the loop structure DLG is $\mathcal{O}(pk)$, and the number of edges 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 $\mathcal{O}(pk^3)$.

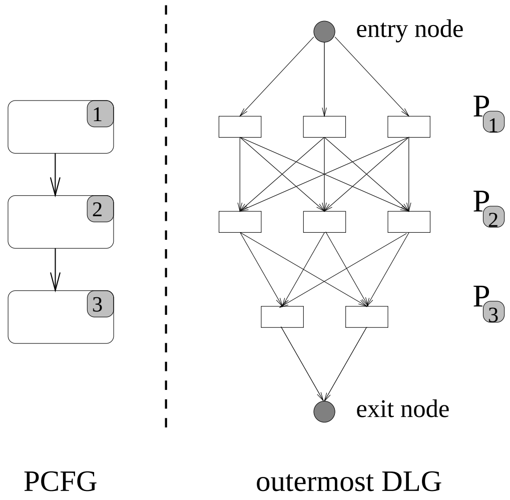


Fig. 10. Example DLG for outermost program level. The candidate-layout search spaces are assumed to have three and two layouts for the first two phases and the third phase, respectively.

The construction of a branch summary DLG is shown in Figure 9. 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, since edge weights reflect the execution probabilities for each branch. 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 10. 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.

Theorem 3.5.1. *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.*

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 $\mathcal{O}(\sum_{i=0}^{\infty} \ k \ 1/2^i \ pk^2) \ = \ \mathcal{O}(pk^3)$. □

3.5.4 *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 in-depth discussion of integer programming can be found elsewhere [Nemhauser and Wolsey 1988].

For the purpose of this discussion, we assume that the input program has n phases, $P_1, ... \, P_n$. 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 3.5.2. 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 $s : X \rightarrow \{0, 1\}$ 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 3.5.4.2.

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 kth node of the ith phase. The switch $x_{ik}^{jl} \in X_{remap}$ represents the remapping edge between the lth node of phase j and the kth 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 7) are shown in Figure 11.

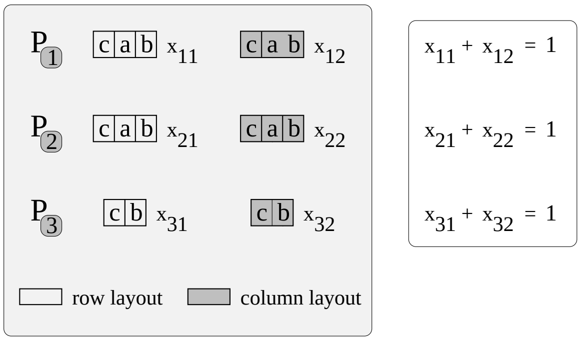


Fig. 11. Layout constraints for the first three phases of our ADI kernel example.

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:

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

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

3.5.4.1 *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 each node x_{ik}, an IN-constraint of the form

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

is generated for all incoming incident edges x_{ik}^{jl}. 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

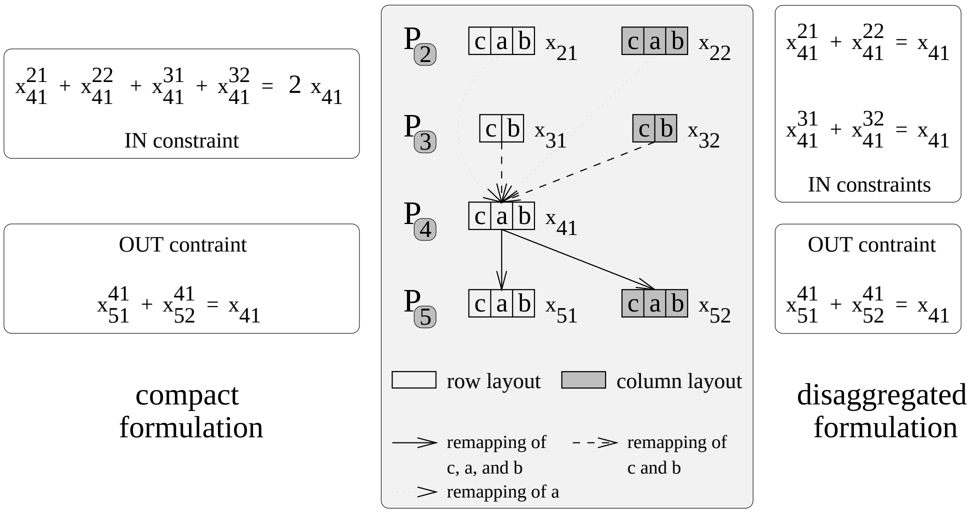


Fig. 12. Two node-based remapping constraints formulations for the node x_{41} representing the first layout in the fourth phase of our ADI kernel example.

phase P_i. For each node x_{ik}, an OUT-constraint of the form

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

is generated for all outgoing incident edges $x_{j'l'}^{ik}$. 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*.

Disaggregated Node-Based Formulation. For each node x_{ik}, IN-constraints of the form

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

are generated for all incoming incident edges x_{ik}^{jl} with nodes in phase P_j as their sources. 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 each node x_{ik}, OUT-constraints of the form

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

are generated for all outgoing incident edges $x_{j'l'}^{ik}$ with nodes in phase P_j' as their sinks. 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

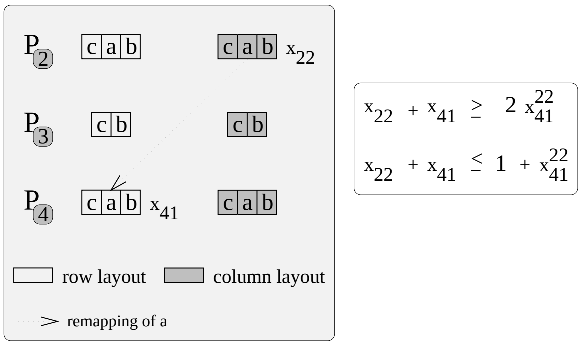


Fig. 13. Edge-based remapping constraints formulations for the edge x_{41}^{22} representing a remapping between the second and fourth phase of our ADI kernel example.

the first candidate layout in the fourth phase of the ADI integration example, x_{41}, are listed in Figure 12.

Edge-Based Formulation. 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 nonlinear 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 13.

All three remapping constraints formulations can be used to compute an optimal solution. A correctness proof of the three formulations can be found in Kremer [1995]. 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 IN-constraints of the disaggregated formulation are tighter in the sense that a single set of disaggregated IN-constraints implies the validity of the corresponding single compact IN-constraint, but not vice versa. The same holds for the OUT-constraints. For instance, in the example shown in Figure 12, 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-onstraint is satisfied, but the corresponding set of disaggregated IN-constraints is not. The notion of tightness has only an impact on the solution speed of the resulting 0-1 problem instances. For a given data layout selection problem, both compact and disaggregated formulations together with the layout constraints are equivalent with respect to the set of solutions they allow. Experiments based on the different formulations are discussed in the next section.

3.5.4.2 *Efficiency of 0-1 Integer Programming Formulations.* 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 three-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 [Tseng 1993]. 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 standalone tool that generates the three different 0-1 problem formulations discussed in Section 3.5.4 for an input DLG. The tool was written in *Scheme*, a dialect of *Lisp* [Dybvig 1987]. The sole purpose of this tool was to provide a testbed to determine the efficiency of the different 0-1 problem formulations. The top part of Table I shows the sizes of the automatically generated 0-1 problem instances.

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. We initially used 6 DLGs to evaluate the formulations. The weights of the DLGs were computed by hand assuming different balances of communication and computation costs for the target architecture and different abilities of the target compiler to exploit partial parallelism through pipelining. Additional 6 DLGs were used to allow a better evaluation of the disaggregated node-based formulation. Each of the 0-1 problem instances was solved by *CPLEX*[1], a state-of-the-art linear integer programming tool and library, partly developed by Robert Bixby at Rice University [Bixby 1992]. *CPLEX* includes an implementation of a general-purpose branch-and-bound code for mixed integer programming. Being general purpose, this code does not use apriori knowledge about the structural properties of our particular 0-1 problems.

Table I gives the solution times in seconds of the 24 0-1 problem instances using *CPLEX* on a SUN SPARC–10 (*CPLEXv3.0*) and SUN UltraSparc–1 (*CPLEXv4.0*). By default, *CPLEX* uses a primal simplex for the initial LP relaxation. The table includes solution times for the dual simplex instead of the primal simplex for *CPLEXv4.0* on the UltraSparc–1[2]. The reported averages exclude experiments that took longer than 30 minutes. Note that some experiments took more than 4 hours to finish.

[1] *CPLEX* is a trademark of CPLEX Optimization, Inc.
[2] Ed Rothberg from SGI pointed out the potential benefit of using the dual instead of the default primal simplex.

Table I. CPLEX Solution Times on Sparc–10 and UltraSparc–1 Workstations for Three Different 0-1 Problem Formulations

	Compact Node-Based			**Disaggregated Node-Based**			**Edge-Based**		
sizes of 0-1 problem instances									
#layout variables	253			253			253		
#remapping variables	2133			2133			2133		
#constraints	485			715			4306		
solution times in seconds									
Sparc–10 CPLEXv3.0 (primal)	best	worst	avg.	best	worst	avg.	best	worst	avg.
	8.2	$*$	292	2.6	4.8	3.8	$*$	$*$	$*$
UltraSparc–1 CPLEXv4.0 (primal)	best	worst	avg.	best	worst	avg.	best	worst	avg.
	7.7	1018	261	0.44	1.05	0.74	293	$*$	293†
UltraSparc–1 CPLEXv4.0 (dual)	best	worst	avg.	best	worst	avg.	best	worst	avg.
	2.4	337	104	0.20	0.35	0.28	754	$*$	754†

† Only a single problem instance could be solved optimally within 30 minutes. A "$*$" indicates that *CPLEX* took more than 30 minutes.

The experiments show that the three formulations have significantly different solution times. The same DLG problem instance can be solved in a matter of seconds, minutes, or hours, depending on the chosen 0-1 formulation. Selecting the right 0-1 formulation is therefore crucial. The disaggregated node-based formulation can be solved by the general-purpose *CPLEX* with dual initial simplex in 300 milliseconds on average on a UltraSparc–1. *CPLEX* did not take longer than 0.35 seconds on any of the 12 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.

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, three-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

Table II. Timings for Disaggregated 0-1 Problem Instances

	3 Copies, 120 Phases		**5 Copies, 200 Phases**	
	Coupled	**Decoupled**	**Coupled**	**Decoupled**
sizes of 0-1 problem instances				
#layout variables	745	745	1237	1237
#remapping variables	7923	6693	13713	11253
#number of constraints	2689	2227	4663	3739
solution times in seconds				
Sparc–10 CPLEXv3.0 (primal)	115	34	357	87
UltraSparc–1 CPLEXv4.0 (primal)	13.2	4.9	28.6	13.2
UltraSparc–1 CPLEXv4.0 (dual)	1.4	1.1	3.4	2.8

variables in our Erlebacher code. The resulting sizes of the 0-1 problem instances and their solution times using *CPLEXv3.0* on a SUN SPARC–10 and *CPLEXv4.0* on a SUN UltraSparc–1 are given in Table II. As in the previous experiments, *CPLEXv3.0* numbers are based on a primal simplex for the initial relaxation, and *CPLEXv4.0* results are reported for both primal and dual simplex.

Up to 50% of the speedup of *CPLEXv4.0* over *CPLEXv3.0* (both using primal simplex) can be attributed to algorithmic improvements in the presolver step of *CPLEXv4.0* as opposed to hardware improvements (private communication, I. Lustig, CPLEX Optimization Inc., 1996). The second dramatic decrease in solution times is due to the use of dual instead of primal simplex for the initial LP relaxation. It is important to note that the reported speedups—ranging from 31 to 105—were achieved for the same 0-1 problem instances and therefore came to some extent *for free*. Once an efficient 0-1 problem formulation has been found, algorithmic improvements in 0-1 integer programming technology can be immediately exploited.

In Appendix A we show 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 turned out to be difficult. It is our experience that interdimensional 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.

3.5.4.3 *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. For example, applications that reference only a small number of global variables across the entire program and use mostly local variables within procedures will result in 0-1 data layout selection problems that are of a decoupled nature. Decoupled 0-1 problems promise faster solution times (see Table II).

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 interdimensional alignment problem. 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. For instance, all edge-based problem instances shown in Table I that did not find an optimal solution within 30 minutes using *CPLEXv4.0* found a feasible solution within 5% of optimal in less than a minute.

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 time limit can be built into a practical solver to terminate the solution process after the limit has been reached. If an optimal solution is not found within the time limit, the solver returns the best feasible solution found.

4. PROTOTYPE IMPLEMENTATION

A prototype data layout assistant tool has been implemented as part of the D system [Adve et al. 1994]. The prototype tool is a batch system and performs only intraprocedural analysis. Nonlinear control flow in input Fortran programs is restricted to `Do` loops and `If` statements. I/O operations are not supported.

The alignment analysis performance model and the heuristic for alignment search space construction are discussed in Section 4.1 and Section 4.2, respectively. Distribution analysis generates exhaustive search spaces for 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 multidimensional distributions [Tseng 1993]. Fortran D [Fox et al. 1990] shares many features with HPF, since it was one of the main contributors to the HPF language proposal. The execution model uses data dependence information to detect processor synchronization. Phases are classified as either pipelined, loosely synchronous, or reductions. Performance predictions for basic computations and communication patterns are based on machine-level training sets for Intel's iPSC/860 or Paragon [Balasundaram et al. 1991]. The training set node programs were compiled using the highest level of optimization (if77 –O4).

The prototype uses over 100 training sets that measure basic computations such as `real` and `double` floating-point operations, and basic communication patterns such as nearest-neighbor communication, single send/receive pairs, broadcasts, reductions, and transpose operations. The basic computation patterns contain only unit–stride data accesses, reflecting the implicit assumption that most memory accesses will result in cache hits. 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 nonunit stride memory access pattern, and high- or low-latency messages. A nonunit memory access pattern requires message buffering. Low-latency message costs are used to predict the communication costs in pipelined phases where computation and communication can be overlapped. In contrast, message costs for loosely synchronous phases are based on high-latency training sets. An in-depth discussion of the performance estimator is presented elsewhere [Kremer 1995].

The prototype tool solves NP-complete problems during alignment analysis and the final data layout selection step. Instances of these problems are translated into 0-1 integer programming problems suitable to be solved by *CPLEX* version 4.0 with dual simplex as the initial LP relaxation. The prototype tool builds the required constraint matrices internally, and directly calls the *CPLEX* routines without creating any intermediate files.

4.1 CAG Edge Weights

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

During the process of determining edge weights, our CAG representation is a directed graph. The edge directions keep track of the flow of values within an alignment preference. The flow of values is important to assess the communication costs in the presence of the owner–computes rule. For example, our performance model treats an array reference pattern `A=B` followed by the same reference pattern `A=B` differently than the pattern `A=B` followed by `B=A`. If the alignment preferences cannot be satisfied, the latter pattern sequence may result in two communications while the former patterns may only generate a single communication. Therefore, if the same alignment preference is encountered, our algorithm checks whether the preference has the same direction as the one already represented in the CAG. If the directions are not identical, the edge weight is increased by the predicted communication cost, and the edge direction is reversed. If the directions are the same, the CAG remains unchanged. The predicted communication costs reflect the volume of the communication, i.e., correspond to the sizes of the arrays that are communicated. Once the edge weights of a CAG have been determined, all edge directions are removed.

4.2 Heuristic for Construction of Alignment Search Spaces

The program template is computed based on the maximal dimensionalities and maximal dimensional extents of the arrays in the program. The alignment search spaces are initialized with the undirected, weighted CAGs of their phases. If a phase CAG has interdimensional alignment conflicts, the conflicts are resolved, and the resulting CAG is used for the initialization. In the next step, phases in the PCFG are grouped into classes such that each class of phases has alignment preferences that do not conflict. Candidate alignment search spaces are built for these metaphases by exchanging candidate alignments between metaphases. As a final step, each phase search space is derived from its metaphase search space through a projection mapping. Metaphases are mainly constructed to improve the efficiency of our search space construction algorithm. The remainder of this section discusses the algorithm in more detail.

After initialization, the phases are partitioned into classes such that their merged CAGs are conflict-free. Each class of phases is represented by its joined, conflict-free CAG. The current prototype uses a greedy strategy to determine the phase CAGs to be merged next. The implemented algorithm visits the phases, i.e., the nodes in the PCFG in reverse postorder [Hecht 1977], and joins their CAGs as long as no conflict is detected. Once a conflict is encountered, a new class is created and initialized with the CAG of the single phase that led to the conflict. The partitioning algorithm terminates after all phases have been visited.

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

The current prototype imports each optimal alignment candidate of another phase class into the search space of a phase class. If the phase partitioning has p classes, then each final class alignment search space can have at most p candidates. In order to avoid duplication of alignment information, the imported alignment candidate is only inserted into the class search space if its alignment information is not weaker or equal to any alignment information already in the search space. A more detailed discussion of the alignment heuristic can be found elsewhere [Kremer 1995]. Finally, candidate alignment schemes for phase classes are translated into candidate alignments for each individual phase in the class.

Since the current prototype generates exhaustive search spaces of only one-dimensional `block` distributions, the orientation selection is trivial due to the symmetry between orientations and distribution candidates. For instance, in the two-

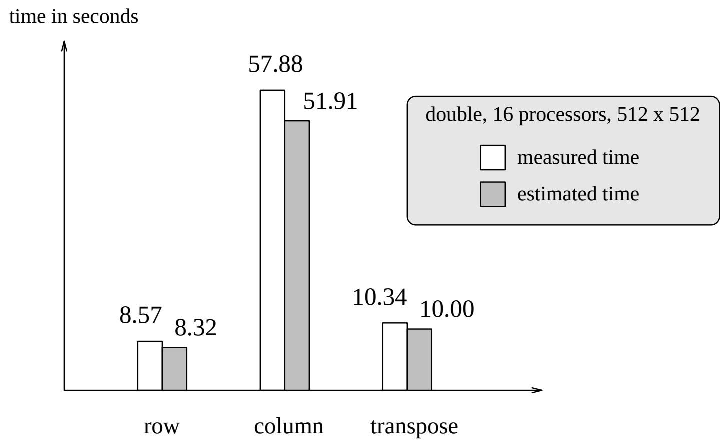


Fig. 14. Example test case for ADI with three possible data layouts.

dimensional case, the candidate layout resulting from a transposed orientation and distribution by row is the same as from a canonical orientation and distribution by column.

4.3 Experimental Results

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

It is important to note that it is not the goal of 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 predict 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 [Kremer and Ramé 1994].

The automatic data layout tool was applied to each program for a set of 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 14 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. The following two sections discuss the

Table III. Breakdown of Times Spent in Different Parts of the Prototype Implementation

	Adi	**Erlebacher**	**Tomcatv**	**Shallow**
Execution times summary				
Total execution time	**2.97**	**9.49**	**7.06**	**5.22**
General D System infrastruture	1.01	5.48	2.83	2.52
Automatic data layout	**1.96**	**4.01**	**4.23**	**2.70**
% of total	66%	42%	60%	52%
Automatic data layout module				
Initialization of data structures†	0.03	0.23	0.12	0.18
Initialization of performance estimator (training set machine model)	1.61	1.61	1.61	1.61
PCFG and search space construction	0.09	0.49	1.56‡	0.26
Performance estimation	0.17	1.50	0.71	0.48
DLG construction and data layout selection (0-1 problem solver)	0.05 (0.01)	0.17 (0.03)	0.22 (0.01)	0.16 (0.03)
Code generation	0.01	0.01	0.01	0.01

† Excludes the initialization time for the performance estimation machine model (training sets).
‡ Time spent to solve two 0-1 interdimensional alignment problems was 0.3 and 0.13 seconds.

efficiency and effectiveness of our prototype implementation.

4.3.1 *Efficiency.* Table III contains a breakdown of the average, total execution times of the tool on a SUN UltraSparc–1 for all test cases of the four test programs. The granularity of the used timing routine was 10 milliseconds.

The general D System infrastructure includes parsing, data dependence analysis, and the construction of other internal program representations. For all test cases, the tool determined a data layout in less than 10 seconds. The tool spent up to 66% of its execution time on tasks directly related to automatic data layout. Between 50% and 90% of the time spent on automatic data layout was due to static performance estimation. As part of the initialization step of the tool, more than 100 training sets were read from disk and represented internally. This cost of 1.61 seconds is fixed and does not change with input program size. The solutions to the generated 0-1 problem instances did not contribute more than 12.5% to the execution time of the automatic data layout module, and not more than 7.5% to the overall execution time.

4.3.2 *Effectiveness.* 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 14 the prototype tool picked the best data layout, namely a static rowwise data layout, and 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 in the generated search spaces correctly, and whether the best predicted data layout alternative was also the best measured alternative. In the following, the notion of *best layout* is always relative to the program data layouts that can be generated based on the candidate data layout search spaces.

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 [Tseng 1993] 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.

4.3.2.1 *Adi.* Adi solves a two-dimensional problem (see Figure 3). The program has 9 phases. Since the current prototype does not support I/O operations, the write operation has been eliminated, and the read operation has been substituted by three initialization loops, one for each array in the program. There are no interdimensional alignment conflicts. The solution of the 0-1 data layout selection problem took *CPLEX* 10 milliseconds on average on a UltraSparc–1. The problem had 61 variables and 53 constraints.

We measured 40 test cases, one of which is shown in Figure 14. The results of all five test cases for problem size 256×256 and double-precision arrays can be found in Figure 15. 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 due to imprecisions in the performance prediction. The worst-case performance loss due to the suboptimal selection was 7.4% as compared to the best choice.

4.3.2.2 *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 interdimensional alignment conflicts. The data layout selection step generated a 0-1 problem with 327 variables and 190 constraints. *CPLEX* solved the problem in 30 milliseconds on average on a UltraSparc–1.

The program consists of a three symmetric computations, each along one of the dimensions of the problem. The computations share access to a three-dimensional, read-only array. All four three-dimensional arrays are aligned canonically, i.e., there is no interdimensional 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

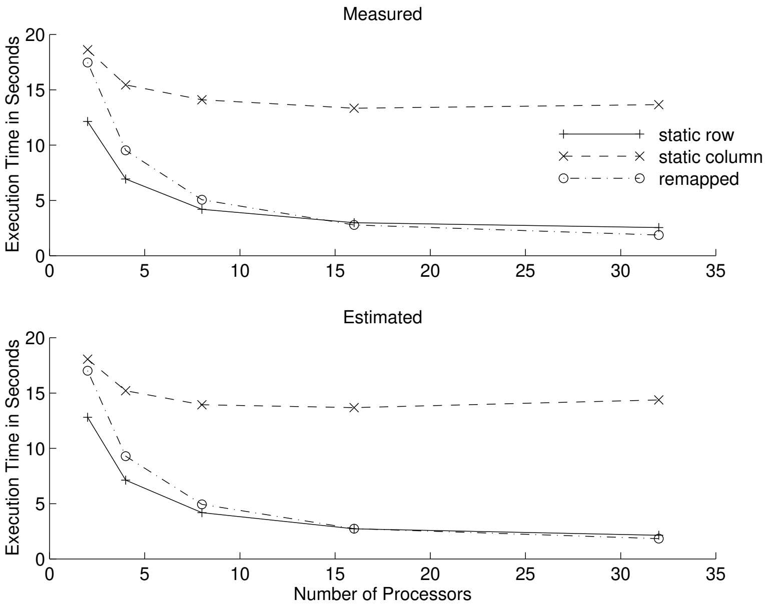


Fig. 15. Measured and predicted execution times for Adi kernel with problem size 256 × 256, double-precision.

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 16 shows the measured and predicted 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%.

4.3.2.3 *Tomcatv.* In contrast to Erlebacher and Adi, Tomcatv has interdimensional alignment conflicts for two of its two-dimensional arrays. The assistant tool partitioned the 17 phases into two classes and exchanged their interdimensional alignment information. The resulting alignment search spaces for each phase had two entries. Together with the two possible single-dimensional 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.

The two interdimensional alignment conflicts were translated into 0-1 problems with 312 variables and 530 constraints. Although the sizes of the two problems

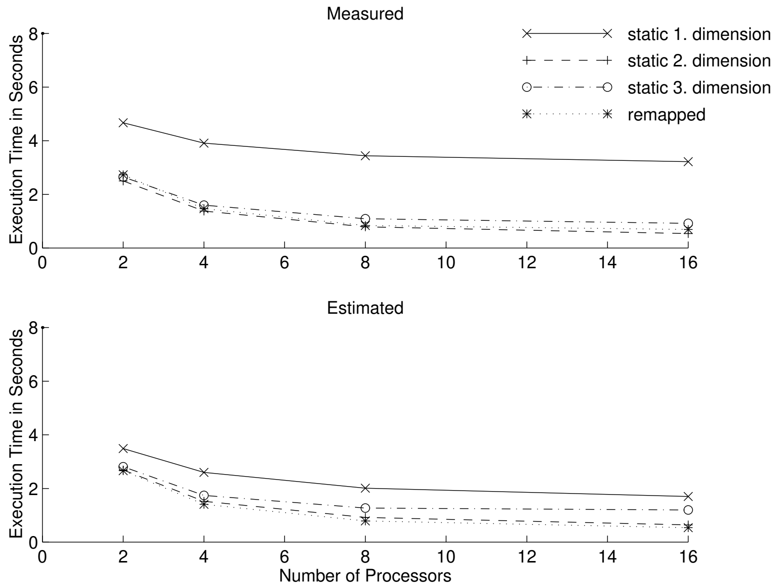


Fig. 16. Measured and predicted execution times for Erlebacher with problem size $64 \times 64 \times 64$, real.

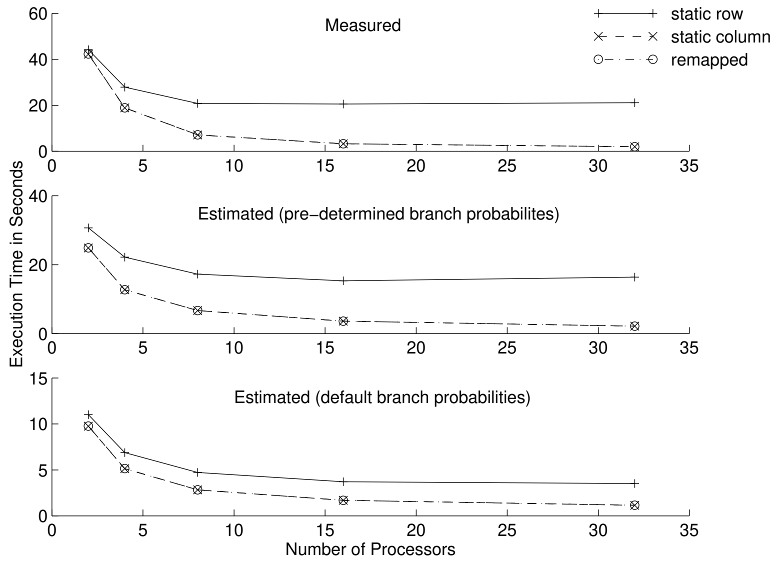


Fig. 17. Measured and predicted execution times for Tomcatv with problem size 128×128, double-precision (note the different time scales).

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 UltraSparc–1, *CPLEX* solved the two problems in 300 and 130 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 10 milliseconds on average on a UltraSparc–1.

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 columnwise data layout. The six suboptimal choices resulted in a performance degradation of at most 2.0% as compared to the best choice.

Figure 17 shows the measured and predicted 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 17 shows the resulting predictions. However, if the actual branch probabilities are used, the performance prediction is more precise although still lower than the actually measured timings.

4.3.2.4 *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 30 milliseconds on average on a UltraSparc–1. 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 18 shows the 5 test cases for data type *real* and a problem size of 384×384. The static performance predictions slightly overestimate the measured timings. However, the relative performance is predicted with high accuracy.

4.4 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 nontrivial, 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

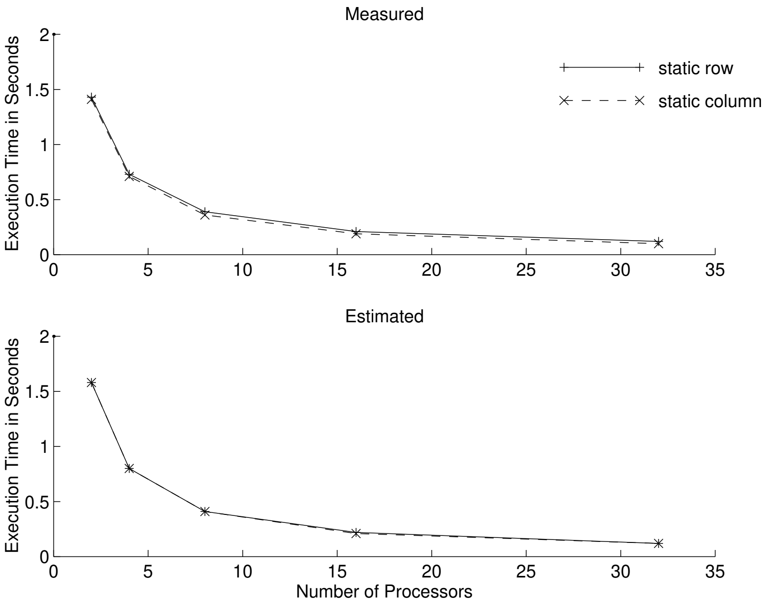


Fig. 18. Measured and predicted execution times for Shallow with problem size 384 × 384, real.

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 accuracy. Shallow shows the precision of relative performance predictions 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 instances of the interdimensional alignment problem and all instances of the data layout selection problem were solved in less than 30 milliseconds and 300 milliseconds, respectively. At most 7.5% of the overall execution time of the tool was spent on solving these 0-1 integer programming problems. The tool took between 3 and 10 seconds for every test case. These results show that the prototype implementation of our framework is efficient and generates good data layouts for our test programs.

5. RELATED WORK

Compiling a single name space program for a distributed–memory architecture requires a 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 parallelism and load balancing.

The problem of automatic data and computation mapping for scientific applications has been discussed extensively in the literature. The proposed solutions for regular problems with dense arrays differ significantly in the assumptions that are made about the possible sets of data and computation mappings, the compilation system, and the target machine architecture. For instance, the data and computation mappings may be rather general or chosen from a restricted set; the compilation system may perform a variety of 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.

Most previous approaches to automatic data and computation mapping have been developed as part of an optimizing compiler, i.e., have been designed as an compiler optimization pass and not as part of an automatic data or computation mapping tool. Although most techniques may be used in both contexts, there are significant differences between the two scenarios.

Techniques embedded in a compiler allow a tight coupling between the data and computation mapping pass and other compilation phases. Compiler-generated temporaries may also be mapped in addition to the user-specified arrays. Since the automatic data and computation mapping pass is invoked for each compile, the design of the pass has to guarantee an acceptable response time. A response time may be considered acceptable if it is comparable to the execution times of other compilation passes, i.e., typically in the order of seconds or a few minutes. Finally, compiler-embedded approaches are based on the assumption that the "compiler knows best", and user feedback is not needed to generate efficient data layouts.

Automatic data and computation mapping techniques that are part of a tool for languages such as HPF support user feedback. We believe that there are applications where user feedback is needed to compensate for the imprecision of compile time analysis techniques which may lead to bad data- and computation-mapping choices. Many HPF programming environments support performance debugging through runtime profiling (e.g. PGI's *pgprof* and IBM's *xprofiler*), allowing the user to focus on performance critical program regions. We believe that for regular problems, HPF and its data layout directives provide a suitable interface to relate the user's input back to the compiler.

It is important to note that the task of automatic data and computation mapping is harder for a tool than for a compiler-embedded scenario, since the mapping techniques have to be able to target different HPF compilers and their supported architectures in order to allow program portability across different HPF environments. The techniques have to accommodate an entire class of compilers through, for instance, a parameterized compiler model. What optimizations to include in the compiler model will depend on the required precision of the model and on the constraints of the tool's user interface. The latter problem is similar to the problem of debugging optimized code, since the automatically selected data layout has to be presented to the user in the context of the original program. Finally, since the tool will not be invoked each time a program is compiled, automatic mapping techniques are not subject to the same time constraints as compiler-embedded techniques.

A unique aspect of our work 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 [Pugh 1991]. Using integer programming for instruction scheduling under resource constraints for superscalar machines has been discussed by Feautrier [Feautrier 1994] and Ning, Govindarajan, Altman and Gao [Altman and Gao 1994; Altman et al. 1995; Ning and Gao 1993]. Integer programming techniques in the context of a distributed–memory compiler have been proposed by Phillipsen [Philippsen 1995] and Garcia, Ayguadé and Labarta [Garcia et al. 1995]. The latter two works have been based on our previously published experience with 0-1 integer programming [Bixby et al. 1994; Kennedy and Kremer 1995].

In the remainder of this section different approaches to automatic data and computation mappings that were mainly designed for use within an optimizing compiler will be discussed. Many aspects of these approaches, for instance the different heuristics to generate candidate alignments and distributions, may be fully or partially implemented within our framework. The discussion is divided into techniques that only allow a single data and computation mapping for the entire program (*static mappings*) or allow different mappings for different parts of the program (*dynamic mappings*). Finding static data and computation mappings automatically is considered a simpler problem than allowing dynamic changes of the mappings.

5.1 Static Mappings

The problem of finding efficient data and computation mappings for special cases such as single loop nests with regular data access patterns (e.g., stencils) has been addressed by a few researchers. In these cases mathematical frameworks can be used effectively to find communication-free mappings [Bau et al. 1994; D'Hollander 1989; Huang and Sadayappan 1991] or mappings with minimal communication [Hudak and Abraham 1990; Ramanujam and Sadayappan 1989]. In the more general case, most solutions to the static mapping problem are graph based. Weighted graphs are used to represent trade-offs between different data and computation mappings. The majority of approaches emphasize the data-mapping view of the mapping problem, i.e., try to find a data mapping that allows an efficient computation mapping. The decomposition of the data mapping problem into alignment and distribution leads to a two-step strategy, where alignment analysis is typically followed by distribution analysis.

Early work done by Knobe et al. at Compass, Inc., focuses on the alignment of arrays as part of a family of Fortran compilers for SIMD target architectures [Knobe et al. 1990]. Intradimensional alignments and interdimensional alignments without transpositions can be generated. The alignment algorithm is based on the usage patterns of arrays and Fortran 90-style array sections in the source program. Each pattern generates mapping requests, called *preferences*, that indicate the optimal layout of the arrays relative to each other. Preferences may specify that array elements should be mapped to the same virtual processor in order to avoid communication, or should be mapped across multiple processors to allow parallel execution. The preferences of the program are represented by the undirected *preference graph* where the arcs correspond to the preferences and where 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. 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. Knobe et al. later extended their work to handle dynamic realignments [Knobe and Natarajan 1993].

Li and Chen investigated techniques for automatic data layout as part of the Crystal compiler and language project at Yale University [Li 1991; Li and Chen 1991a; 1991b]. Crystal is a high-level, purely functional language. Their compiler work focuses on the interdimensional alignment problem, modeled as a graph problem. An undirected, weighted graph, called the *component affinity graph* (CAG), is constructed based on normalized reference patterns in the source program. Each dimension of an array that is referenced in the program 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 program. 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. Li and Chen show that the solution of this alignment problem is NP-complete, and they propose a heuristic to solve the problem [Li and Chen 1991b].

Gupta and Banerjee at the University of Illinois at Urbana-Champaign developed techniques for automatic data layout as part of a Fortran 77 compiler based on the Parafrase-2 program restructurer [Gupta 1992; Gupta and Banerjee 1992]. The compiler generates SPMD node programs with explicit communication. It performs alignment and distribution analysis based on *constraints* for each statement in the program. Constraints represent properties of the data layout and are associated with a quality measure. The compiler performs alignment analysis based on Li and Chen' s approach. The alignment quality measures, i.e., the edge weights in the CAG, are penalty functions. They represent the additional costs that will occur if the array dimensions 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. 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 automatic techniques handle cyclic, block, and block–cyclic distributions. In addition, partial replication of arrays is considered.

Wholey at Carnegie Mellon University developed a compiler for the ALEXI language [Wholey 1992; 1991]. In this language, communication and parallelism are expressed explicitly by primitive operations that are similar to Fortran 90 array constructs and intrinsic communication functions. Alignment analysis is done at compile time, based on the approach by Knobe, Lukas, and Steele. 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 search space is restricted to noncyclic, 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 predicted execution time.

5.2 Dynamic 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 [Chatterjee et al. 1993; Chatterjee et al. 1994; Philippsen 1995], loop nests [Ayguadé et al. 1994; Anderson and Lam 1993; Lee and Tsai 1993; Ning et al. 1995; Palermo and Banerjee 1995; Tandri and Abdelrahman 1997], or groups of statements or loop nests for which it can be shown that remapping between them can never be profitable [Sheffler et al. 1996]. More recent work tries to make the mapping decisions independent of the particular program structure [Kelly and Pugh 1996]. 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.

Palermo and Banerjee extended previous work on automatic data layout by Gupta and Banerjee to handle dynamic remapping as part of the Paradigm project at the University of Illinois at Urbana-Champaign [Palermo 1996; Palermo and Banerjee 1995]. Their approach is based on a hierarchical decomposition of the program into components such that remapping is profitable only between these components. 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 predicted, 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. More recently, Palermo and Banerjee have included interprocedural analysis into their framework [Palermo and Banerjee 1996].

Anderson and Lam at Stanford University have developed automatic data and computation mapping techniques as part of the SUIF compiler project at Stanford University [Anderson 1997; Anderson and Lam 1993]. SUIF accepts Fortran and C programs as input and generates code with and without explicit communication. In the first step, their approach determines an efficient data and computation mapping onto a virtual processor array, i.e., an efficient data and computation alignment. In their approach, the input program is represented as a *communication graph*, a weighted undirected graph where the nodes correspond to loop nests in the program. The node weights are initialized with the costs of a mapping determined by their *static algorithm*. The static algorithm is based on a linear algebra framework and computes communication-free mappings with the largest degree of parallelism. If this mapping is trivial, i.e., all data and computation are mapped to the same processor, the static algorithm tries to find a mapping that can exploit partial parallelism through wave-front-style execution. The edge weights represent remapping costs between node mappings. The dynamic mapping problem is formulated as a partitioning problem of the communication graph into static regions such that the sum of the region costs determined by the static algorithm and the remapping costs between static regions is minimal. Anderson and Lam showed that their problem formulation is NP-complete [Anderson 1997]. They use a greedy algorithm as a heuristic to find a low-cost graph partitioning. For each loop nest level, from innermost to outermost, edges in the communication graph are visited in the order of decreasing weights. Incident nodes are merged into a single static region if the cost of the mapping determined by the static algorithm for the combined nodes is less than the costs of the two individual nodes with remapping between them. Based on the resulting alignments, the virtual processors are mapped onto physical processors using either a block, cyclic, or block-cyclic distributions. In her thesis, Anderson extended the approach to work within an interprocedural framework [Anderson 1997].

The approach proposed by Chatterjee et al. as part of the Excalibur project performs alignment and distribution analysis for data–parallel programs written in a language such as Fortran 90 [Chatterjee et al. 1995; Chatterjee et al. 1994; Sheffler et al. 1996]. The main program representation is the *Alignment–Distribution Graph* (ADG). Alignment and distributions are associated with the endpoints of edges in the ADG, called ports. ADG nodes represent array operations, and edges represent flow of data between operations. A node constrains the relative alignment and distribution of its ports, i.e., its operands and result. Communication operations are explicit in the ADG. Chatterjee et al. discuss a family of alignment algorithms for different alignment cost metrics [Chatterjee et al. 1995]. Compiler-generated and user-defined arrays are considered. Once the alignment has been fixed, a heuristic is used to determine a suitable set of candidate distributions. Candidate distributions include cyclic, block, and block–cyclic mappings. Weights are used to represent the costs for the distributions and remappings in terms of predicted execution times [Chatterjee et al. 1994]. Similar to our approach, the final solution consists of the selection of a single distribution for each port such that the overall cost is minimal. Chatterjee et al. use a divide–and–conquer strategy to determine a minimal-cost path. Starting from a static distribution for the entire program, the algorithm tries to find a partitioning of the nodes into two disjoint sets such that dynamic remapping between them is profitable. More recently, the approach was extended to allow ADG graph contraction without loss of solution precision [Sheffler et al. 1996].

Garcia et al. at the Universitat Politècnica de Catalunya in Barcelona developed automatic mapping techniques in the context of a Fortran programming environment [Garcia 1997; Garcia et al. 1996]. Their approach considers the effects of data alignment, data distribution, and computation mapping at the same time. Possible data distributions are one-dimensional and may be block or cyclic. Fortran programs are partitioned into phases according to our phase definition. Each phase is represented as a subgraph in the Communication–Parallelism Graph (CPG), which contains information about possible data movements, exploitable parallelism, and remapping costs between phases. Similar to Li's and Chen's CAG, nodes in the CPG subgraph associated with a phase represent dimensions of arrays referenced in the phase. Edges represent alignment preferences weighted by the predicted execution time, assuming that the corresponding array dimensions are aligned and distributed. Additional *hyperedges* link all nodes of array dimensions that have to be aligned and distributed for the phase to be executed in parallel. Hyperedges are weighted with the saved execution time if the phase is executed in parallel. Different phase distributions are modeled by replicating the CPG subgraphs for each distribution alternative. Once the entire CPG has been built, the data-mapping problem is solved as a minimal-cost path problem with a few additional constraints. A linear 0-1 integer programming formulation together with a general-purpose 0-1 integer programming solver are used to compute the optimal solution.

6. SUMMARY AND FUTURE WORK

This article discussed a new framework for automatic data layout designed to be used in a data layout assistant tool for languages such as HPF. The framework divides the problem of automatic data layout into five main steps: program partitioning, candidate-layout search spaces construction, performance prediction, data layout selection, and code generation. The interfaces between the steps are defined, and basic infrastructure is provided to allow experimentation with different step implementation strategies. In contrast to compiler-embedded automatic data layout techniques, our framework has to be able to target different compilers and parallel architectures in order to allow program portability across different HPF environments. Therefore, the framework contains a parameterized compiler model in addition to an execution and machine model. Tool-based techniques are also not subject to the same time constraints as compiler-embedded approaches, since they will not be invoked each time a program is compiled.

A prototype data layout assistant tool has been implemented based on our framework. The prototype implementation uses the latest and most powerful general-purpose techniques for linear and integer programming to solve two NP-complete problems optimally, namely the interdimensional alignment problem and the data layout selection problem. Efficiency experiments for the data layout selection problem showed that problem instances consisting of up to 200 search spaces with over 1200 data layout candidates can be solved optimally within seconds. This result was achieved through a careful design of the problem's 0-1 integer programming formulation and an appropriate selection of a single parameter for the general-purpose integer programming solver.

To determine the quality of the generated data layouts, a total of 97 experiments based on four scientific programs and program kernels using a single target compiler and target architectures were conducted. In all cases, the prototype tool took less than 10 seconds to determine the data layout. For three out of the four programs, the data layout choice is nontrivial, i.e., choosing the wrong layout results in substantial performance loss. In 79 cases, the tool selected the best data layout from all possible choices in the candidate data layout search spaces. In the cases where the tool selected a suboptimal layout, the performance loss incurred was within 8.6% of the best layout. All encountered instances of the interdimensional alignment problem and the data layout selection problem were solved in less than 0.3 seconds. The experiments indicate that the framework is efficient and can generate data layouts of high quality.

Although the results of our experiments are encouraging, more experiments are needed to evaluate the efficiency and effectiveness of our framework on larger programs and on a range of different target HPF compilers and parallel platforms. Our prototype implementation is currently being extended at Rutgers University to perform interprocedural search space construction, performance prediction, and final data layout selection. The performance prediction component is being augmented by a memory hierarchy and software pipeline model. The target compilers of the extended implementation will include commercially available HPF compilers from IBM and PGI, and Rice University's dHPF compiler [Adve and Mellor-Crummey 1998]. These compilers support multidimensional block and cyclic distributions. Experiments with the interprocedural prototype will allow a better understanding of the interactions between different steps in the framework and their solution strategies.

Other ongoing research addresses the problem of automatic data layout in the presence of memory constraints [Kremer 1998], PCFG graph contraction similar to the work done by Sheffler et al. [Sheffler et al. 1996], and special-case solutions for the data layout selection step such as techniques proposed by Damian–Iordache and Pemmaraju [Damian-Iordache and Pemmaraju 1998]. Encouraged by our experience with 0-1 technology, we are planning to investigate the benefits of 0-1 technology for other programming environment and compilation problems [Kremer 1997].

Finally, research is needed to design a user interface for our framework. A user should 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. Since any information generated by the framework has to be presented to the user in terms of the original source code, designing an interface will have to address issues similar to the ones encountered in debugging optimized code.

ONLINE-ONLY APPENDIX

A. FORMAL CORRECTNESS PROOFS

Appendix A is available only online. You should be able to get the online-only appendix from the citation page for this article:

http://www.acm.org/pubs/citations/journals/toplas/1998-20-4/p869-kennedy/

Alternative instructions on how to obtain online-only appendices are given on the back inside cover of current issues of ACM TOPLAS or on the ACM TOPLAS web page:

http://www.acm.org/toplas

ACKNOWLEDGMENTS

We would like to thank Robert Bixby, John Mellor-Crummey, Vikram Adve, and Hristo Djidjev for many discussions and helpful suggestions, and the anonymous reviewers for their comments and suggestions.

REFERENCES

Adve, V., Carle, A., Granston, E., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Mellor-Crummey, J., Tseng, C.-W., and Warren, S. 1994. Requirements for data-parallel programming environments. *IEEE Parallel Distrib. Tech. 2,* 3, 48–58.

Adve, V. and Mellor-Crummey, J. 1998. Advanced code generation for High Performance Fortran. In *Compilation Techniques and Run Time Systems for Scalable Parallel Systems.* Lecture Notes in Computer Science, vol. 1366. Springer-Verlag, Berlin. To appear.

Aho, A. V., Sethi, R., and Ullman, J. 1986. *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, Reading, Mass.

Altman, E. R., Govindarajan, R., and Gao, G. R. 1995. Scheduling and mapping: Software pipelining in the presence of structural hazards. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation.* ACM, New York, 139–150.

Altman, R. G. E. R. and Gao, G. R. 1994. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the 27th Annual International Symposium on Microarchitecture.*

Anderson, J. 1997. Automatic computation and data decomposition for multiprocessors. Ph.D. thesis, Stanford University, Palo Alto, Cal.

Anderson, J. and Lam, M. 1993. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation.* ACM, New York, 112–125.

Ayguadé, E., Garcia, J., Girones, M., Labarta, J., Torres, J., and Valero, M. 1994. Detecting and using affinity in an automatic data distribution tool. In *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing.* Lecture Notes in Computer Science, vol. 892. Springer-Verlag, Berlin, 61–75.

Balasundaram, V., Fox, G., Kennedy, K., and Kremer, U. 1991. A static performance estimator to guide data partitioning decisions. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, New York, 213–223.

Bau, D., Kodukula, I., Kotlyar, V., Pingali, K., and Stodghill, P. 1994. Solving alignment using elementary linear algebra. In *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing.* Lecture Notes in Computer Science, vol. 892. Springer-Verlag, Berlin, 46–60.

Bixby, R. 1992. Implementing the Simplex method: The initial basis. *ORSA J. Comput. 4,* 3, 267–284.

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

Callahan, D. and Kennedy, K. 1988. Compiling programs for distributed-memory multiprocessors. *J. Supercomput. 2*, 151–169.

Chapman, B., Mehrotra, P., and Zima, H. 1992. Vienna Fortran – A Fortran language extension for distributed memory multiprocessors. In *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, J. Saltz and P. Mehrotra, Eds. North-Holland, Amsterdam, The Netherlands, 39–62.

Chatterjee, S., Gilbert, J., Schreiber, R., and Teng, S.-H. 1993. Automatic array alignment in data-parallel programs. In *Proceedings of the 20th Annual ACM Symposium on the Principles of Programming Languages.* ACM, New York, 16–28.

Chatterjee, S., Gilbert, J., Schreiber, R., and Teng, S.-H. 1995. Optimal evaluation of array expressions on massively parallel machines. *ACM Trans. Program. Lang. Syst. 17,* 1 (Jan.), 123–156.

Chatterjee, S., Gilbert, J. R., Schreiber, R., and Sheffler, T. 1994. Array distribution in data-parallel programs. Lecture Notes in Computer Science, vol. 892. Springer-Verlag, Berlin, 76–91.

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

Damian-Iordache, M. and Pemmaraju, S. 1998. Automatic data decompositions for message–passing machines. In *Proceedings of the 10th Workshop on Languages and Compilers for Parallel Computing.* Lecture Notes in Computer Science, vol. 1366. Springer-Verlag, Berlin, 64–78.

D'Hollander, E. 1989. Partitioning and labeling of index sets in do loops with constant dependence. In *Proceedings of the 1989 International Conference on Parallel Processing.* II139–II144.

Dybvig, R. K. 1987. *The Scheme Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, N.J.

Feautrier, P. 1994. Fine-grain scheduling under resource constraints. In *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing.* Lecture Notes in Computer Science, vol. 892. Springer-Verlag, Berlin, 1–15.

Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C., and Wu, M. 1990. Fortran D language specification. Tech. Rep. TR90-141, Dept. of Computer Science, Rice University, Houston, Tex.

Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D. 1988. *Solving Problems on Concurrent Processors.* Vol. 1. Prentice-Hall, Englewood Cliffs, N.J.

Garcia, J. 1997. Automatic data distribution for massively parallel processors. Ph.D. thesis, Universitat Politècnica de Catalunya, Barcelona, Spain.

Garcia, J., Ayguadé, E., and Labarta, J. 1995. A novel approach towards automatic data distribution. In *Proceedings of the Workshop on Automatic Data Layout and Performance Prediction (AP'95).* Available as Rice University Tech. Rep. CRPC-TR95-548.

Garcia, J., Ayguadé, E., and Labarta, J. 1996. Dynamic data distribution with control flow analysis. In *Proceedings of Supercomputing '96.* Available at URL http://www.supercomp.org/sc96/proceedings.

Gupta, M. 1992. Automatic data partitioning on distributed memory multicomputers. Ph.D. thesis, University of Illinois at Urbana-Champaign, Ill.

Gupta, M. and Banerjee, P. 1992. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Trans. Parallel Distrib. Syst. 3,* 2 (Apr.), 179–193.

Hecht, M. S. 1977. *Flow Analysis of Computer Programs.* North Holland.

Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., and Tseng, C. 1991. An overview of the Fortran D programming system. In *Proceedings of the 4th Workshop on Languages and Compilers for Parallel Computing.* Lecture Notes in Computer Science, vol. 589. Springer-Verlag, Berlin.

HUANG, C.-H. AND SADAYAPPAN, P. 1991. Communication-free hyperplane partitioning of nested loops. In *Proceedings of the 4th Workshop on Languages and Compilers for Parallel Computing.* Lecture Notes in Computer Science, vol. 589. Springer-Verlag, Berlin.

HUDAK, D. AND ABRAHAM, S. 1990. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing.* ACM, New York, 187–200.

KELLY, W. AND PUGH, W. 1996. Minimizing communication while preserving parallelism. In *Proceedings of the 1996 ACM International Conference on Supercomputing.* ACM, New York, 52–60.

KENNEDY, K. AND KREMER, U. 1995. Automatic data layout for High Performance Fortran. In *Proceedings of Supercomputing '95.* Available at URL http://www.supercomp.org/sc95/proceedings.

KNOBE, K., LUKAS, J., AND STEELE, G. 1990. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *J. Parallel Distrib. Comput. 8,* 2 (Feb.), 102–118.

KNOBE, K. AND NATARAJAN, V. 1993. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. *J. Supercomput. 7*, 387–415.

KOELBEL, C., LOVEMAN, D., SCHREIBER, R., STEELE, JR., G., AND ZOSEL, M. 1994. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, Mass.

KOELBEL, C. AND MEHROTRA, P. 1991. Compiling global name-space parallel loops for distributed execution. *IEEE Trans. Parallel Distrib. Syst. 2,* 4 (Oct.), 440–451.

KREMER, U. 1995. Automatic data layout for distributed memory machines. Ph.D. thesis, Rice University, Houston, Tex. Available as CRPC-TR95-559-S.

KREMER, U. 1997. Optimal and near–optimal solutions for hard compilation problems. *Parallel Process. Lett. 7,* 2, 371–378.

KREMER, U. 1998. Automatic data layout with read–only replication and memory constraints. In *Proceedings of the 10th Workshop on Languages and Compilers for Parallel Computing.* Lecture Notes in Computer Science, vol. 1366. Springer-Verlag, Berlin, 419–422.

KREMER, U. AND RAMÉ, M. 1994. Compositional oil reservoir simulation in Fortran D: A feasibility study on Intel iPSC/860. *Int. J. Supercomput. Appl. 8,* 2 (Summer), 119–128.

LEE, P. AND TSAI, T.-B. 1993. Compiling efficient programs for tightly-coupled distributed memory computers. In *Proceedings of the 1993 International Conference on Parallel Processing.* II161–II165.

LI, J. 1991. Compiling Crystal for distributed memory machines. Ph.D. thesis, Dept. of Computer Science, Yale University, New Haven, Conn.

LI, J. AND CHEN, M. 1991a. Compiling communication-efficient programs for massively parallel machines. *IEEE Trans. Parallel Distrib. Syst. 2,* 3 (July), 361–376.

LI, J. AND CHEN, M. 1991b. The data alignment phase in compiling programs for distributed-memory machines. *J. Parallel Distrib. Comput. 13,* 2 (Oct.), 213–221.

NEMHAUSER, G. L. AND WOLSEY, L. A. 1988. *Integer and Combinatorial Optimization.* John Wiley & Sons, New York.

NING, Q., DONGEN, V. V., AND GAO, G. R. 1995. Automatic data and computation decomposition for distributed memory machines. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences.* IEEE, New York.

NING, Q. AND GAO, G. R. 1993. A novel framework of register allocation for software pipelining. In *Proceedings of the 20th Annual ACM Symposium on the Principles of Programming Languages.* ACM, New York, 29–42.

PALERMO, D. 1996. Compiler techniques for optimizing communication and data distribution for distributed–memory multicomputers. Ph.D. thesis, University of Illinois at Urbana–Champaign, Ill. Available as CRHC-96-09.

PALERMO, D. AND BANERJEE, P. 1995. 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, Ill.

PALERMO, D. AND BANERJEE, P. 1996. Interprocedural array redistribution with data–flow analysis. In *Proceedings of the 9th Workshop on Languages and Compilers for Parallel Computing.* Lecture Notes in Computer Science, vol. 1239. Springer-Verlag, Berlin, 435–449.

PHILIPPSEN, M. 1995. Automatic alignment of array data and processes to reduce communication time on DMPPs. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, New York, 156–165.

PRESBERG, D. L. 1996. Comparison of 3 HPF compilers for the IBM SP. *NHSE Rev. 1996,* 2. Available at http://www.crpc.rice.edu/NHSEreview/96-2.html.

PUGH, W. 1991. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing '91.* 4–13.

RAMANUJAM, J. AND SADAYAPPAN, P. 1989. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of Supercomputing '89.* 637–646.

ROGERS, A. AND PINGALI, K. 1989. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation.* ACM, New York, 69–80.

ROSE, J. AND STEELE, G. 1987. C*: An extended C language for data parallel programming. Technical Report PL87-5, Thinking Machines, Inc., Cambridge, Mass.

SHEFFLER, T. J., SCHREIBER, R., PUGH, W., GILBERT, J. R., AND CHATTERJEE, S. 1996. Efficient distribution analysis via graph contraction. *Int. J. of Parallel Prog. 24,* 6 (Dec.), 599–620.

TANDRI, S. AND ABDELRAHMAN, T. 1997. Automatic partitioning of data and computation on scalable shared memory multiprocessors. In *Proceedings of the 1997 International Conference on Parallel Processing.* 64–73.

Thinking Machines Corp. 1989. *CM Fortran Reference Manual*, Version 5.2-0.6 ed. Thinking Machines Corp., Cambridge, Mass.

TSENG, C. 1993. An optimizing Fortran D compiler for MIMD distributed-memory machines. Ph.D. thesis, Rice University, Houston, Tex. Available as Rice COMP TR93-199.

WHOLEY, S. 1991. Automatic data mapping for distributed-memory parallel computers. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa.

WHOLEY, S. 1992. Automatic data mapping for distributed-memory parallel computers. In *Proceedings of the 1992 ACM International Conference on Supercomputing.* ACM, New York, 25–34.

ZIMA, H., BAST, H.-J., AND GERNDT, M. 1988. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Comput. 6*, 1–18.

Received May 1997; revised October 1997; accepted December 1997

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

Automatic Data Layout for Distributed-Memory Machines

KEN KENNEDY
Rice University
and
ULRICH KREMER
Rutgers University

A. ALIGNMENT CONFLICT RESOLUTION AS A 0-1 PROBLEM

This section describes the translation of an instance of the interdimensional alignment problem into an instance of a 0-1 integer programming problem with linear constraints. Note that there are many possible translations. Experiments showed that our formulation is very promising. A proof of the correctness of our formulation is given in [Kremer 1995].

Definition A.0.1. An instance of the interdimensional 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 A.0.2. 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 $s : X \to \{0, 1\}$ 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 ith 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 kth 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 $\sum_{k=1}^{d} a_{ik} = 1$ $(type1)$ are introduced for each node a_i. Two dimensions of the same array must not be in the same partition. This property is enforced by the following constraints, one constraint for each pair of an array a and a partition k, $1 \leq k \leq d$: $\sum_{i=1}^{dim(a)} a_{ik} \leq 1$ $(type2)$. Note that in the case of an embedding of array a, $dim(a) < d$, some partition k will not contain any CAG node associated with a. There are $|N|$ node constraints of $(type1)$ and $d|Arrays|$ node constraints of $(type2)$, where N is the set of nodes in the CAG, and $Arrays$ is the set of arrays represented in the CAG.

Edge Constraints. The formulation of the interdimensional 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".

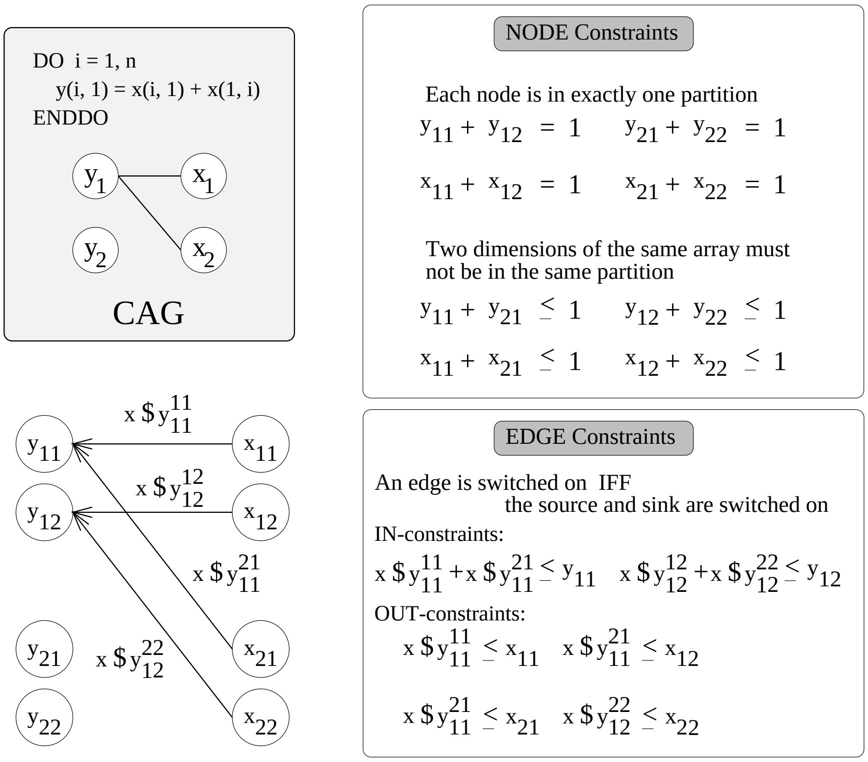


Fig. 19. Alignment conflict resolution of an example CAG as a 0-1 integer programming problem.

Let $SRC(b, a_i)$ denote the set of all nodes b_j that represent a dimension of array b, and there is an edge from b_j to a_i. For each k, $1 \leq k \leq d$, and each nonempty set $SRC(b, a_i)$, IN-constraints of the form $\sum_{b_j \in SRC(b,a_i)} \; b\$a^{jk}_{ik} \leq a_{ik}$ 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 nonempty set $SINK(a_i, c)$, OUT-constraints 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 19 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 19 is not actually generated during the translation process, but illustrates our 0-1 formulation.

Objective Function. A solution of the 0-1 problem formulation of the interdimensional 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.

B. NP-COMPLETENESS OF DATA LAYOUT SELECTION PROBLEM

B.1 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, $P_1, \ldots P_n$, with no control flow between phases. Showing that the restricted problem is NP-complete implies that the general problem is NP-complete as well.

Let V denote the set of variables in the program, $V = \{v_1, \ldots v_r\}$. Let p_i denote the variables referenced in the ith 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 $D_i = \{d^1_i, \ldots\ d^{m_i}_i\}$. Note that two phases may have a different number of candidate data layouts. A single candidate data layout $d^k_i = \{d^k_{ij_1}, \ldots d^k_{ij_{q_i}}\}, 1 \leq k \leq m_i$, is a set of layouts, one layout for each variable $v \in p_i = \{v_{j_1}, \ldots v_{j_{q_i}}\}$.

The cost of executing phase P_i under the data layout $d^k_i \in D_i$ is denoted by $c(P_i, d^k_i)$. 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^s_α and d^t_β be two candidate data layouts for phase P_α and phase P_β, respectively. The remapping cost is given below:

$$c(d^s_\alpha, d^t_\beta) = \sum_{v_i \in p_\alpha \cap p_\beta} c(d^s_{\alpha i}, d^t_{\beta i}),$$

where $c(d^s_{\alpha i}, d^t_{\beta i})$ 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 B.1.1. 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 B.1.2. 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 B.1.3. 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 nonnegative integer.

Definition B.1.4. 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 B.1.5. *DYN-REMAP(k) is NP-complete.*

PROOF. The proof consists of two parts. Lemma B.1.1 shows that DYN-REMAP(k) is in NP. Lemma B.1.2 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. □

LEMMA B.1.1. *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. □

LEMMA B.1.2. *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.

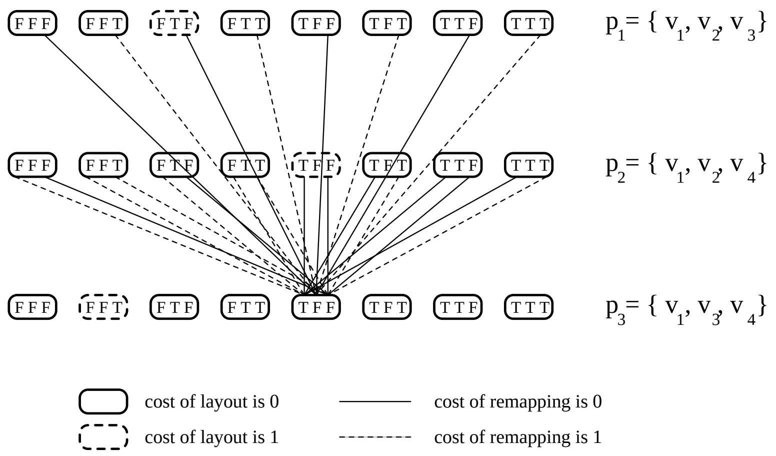


Fig. 20. Sample costs for $g(B)$, $B = (v_1 \vee \neg v_2 \vee v_3) \ \wedge \ (\neg v_1 \vee v_2 \vee v_4) \ \wedge \ (v_1 \vee v_3 \vee \neg v_4)$.

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 $\{d'_1, d'_2, ...\ d'_t\}$ 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 $\{d_1, d_2, ...\ d_t\}$ 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. □

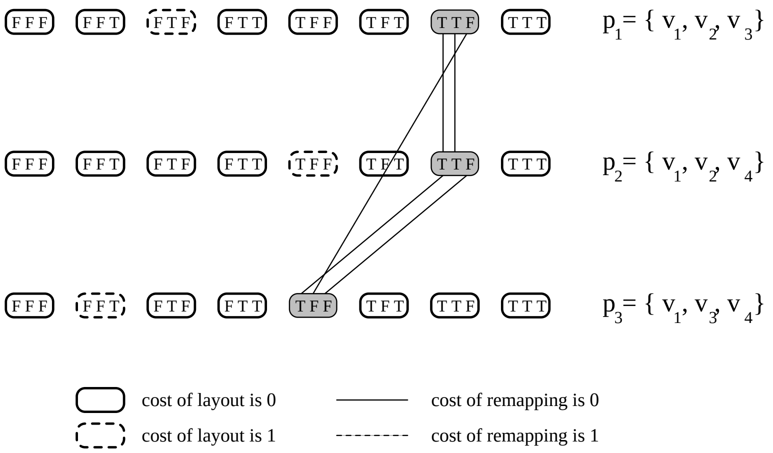


Fig. 21. A solution for $g(B)$, $B = (v_1 \vee \neg v_2 \vee v_3)\ \wedge\ (\neg v_1 \vee v_2 \vee v_4)\ \wedge\ (v_1 \vee v_3 \vee \neg v_4)$.

B.2 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:

—$V = \{v_1, v_2, v_3, v_4\}$

—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 $p_3 = \{v_1, v_3, v_4\}$.

—$$\begin{aligned} D_1 = \{ & \{(v_1,F),(v_2,F),(v_3,F)\},\{(v_1,F),(v_2,F),(v_3,T)\},\{(v_1,F),(v_2,T),(v_3,F)\}, \\ & \{(v_1,F),(v_2,T),(v_3,T)\},\{(v_1,T),(v_2,F),(v_3,F)\},\{(v_1,T),(v_2,F),(v_3,T)\}, \\ & \{(v_1,T),(v_2,T),(v_3,F)\},\{(v_1,T),(v_2,T),(v_3,T)\}\ \}, \\ D_2 = \{ & \{(v_1,F),(v_2,F),(v_4,F)\},\{(v_1,F),(v_2,F),(v_4,T)\},\{(v_1,F),(v_2,T),(v_4,F)\}, \\ & \{(v_1,F),(v_2,T),(v_4,T)\},\{(v_1,T),(v_2,F),(v_4,F)\},\{(v_1,T),(v_2,F),(v_4,T)\}, \\ & \{(v_1,T),(v_2,T),(v_4,F)\},\{(v_1,T),(v_2,T),(v_4,T)\}\ \}, \text{ and} \\ D_3 = \{ & \{(v_1,F),(v_3,F),(v_4,F)\},\{(v_1,F),(v_3,F),(v_4,T)\},\{(v_1,F),(v_3,T),(v_4,F)\}, \\ & \{(v_1,F),(v_3,T),(v_4,T)\},\{(v_1,T),(v_3,F),(v_4,F)\},\{(v_1,T),(v_3,F),(v_4,T)\}, \\ & \{(v_1,T),(v_3,T),(v_4,F)\},\{(v_1,T),(v_3,T),(v_4,T)\}\ \}. \end{aligned}$$

—The costs for the data layouts for the phases and the costs for remapping of individual variables are shown in Figure 20. 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 21 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.

C. EXPERIMENTAL RESULTS

This section contains the entire data generated for the effectiveness experiments described in Section 4.3.2. Table entries are pairs of the form

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

All timings are given in seconds with Intel's iPSC/860 as the target system. 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.

C.1 Adi

C.1.1 *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

C.1.2 *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

C.1.3 *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**

C.2 Erlebacher

C.2.1 *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

C.2.2 *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

C.2.3 *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

C.2.4 *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**

C.3 Tomcatv

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

C.3.1 *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

C.3.2 *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**

C.3.3 *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

C.4 Shallow

C.4.1 *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

C.4.2 *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**