Published in the Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT94), Montréal, Canada, August 1994.

Automatic Data Layout Using 0–1 Integer Programming[1]

Robert Bixby[a], Ken Kennedy[b], and Ulrich Kremer[b]

[a] Department of Computational and Applied Mathematics, Rice University, U.S.A.

[b] Department of Computer Science, Rice University, U.S.A.

Abstract: The goal of languages like Fortran D or High Performance Fortran (HPF) is to provide a simple yet efficient machine-independent parallel programming model. By shifting much of the burden of machine-dependent optimization to the compiler, the programmer is able to write data-parallel programs that can be compiled and executed with good performance on many different architectures. However, the choice of a good data layout is still left to the programmer. Even the most sophisticated compiler may not be able to compensate for a poorly chosen data layout since many compiler decisions are driven by the data layout specified in the program.

The choice of a good data layout depends on many factors, including the target machine architecture, the compilation system, the problem size, and the number of processors available. The option of remapping arrays at specific points in the program makes the choice even harder. Current programming tools provide little or no support for this difficult selection process.

This paper discusses automatic data layout techniques in the context of a programming environment and an advanced compilation system that allows dynamic data remapping. Our proposed framework for automatic data layout builds and examines search spaces of candidate data layouts. A candidate layout is an efficient layout for some part of the program. Choosing a single layout for each program part among its candidate data layouts such that their overall cost is minimal has been shown to be NP-complete. Instead of resorting to heuristics, this paper investigates methods to determine the optimal selection. The data layout selection problem is formulated as a 0–1 integer programming problem, which is then fed to a state-of-the-art, general purpose integer programming solver. Our experiments

[1]**Corresponding author**: Ulrich Kremer, Department of Computer Science, Rice University, 6100 S. Main, Houston, Texas 77005; e-mail:kremer@cs.rice.edu. 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, and the IBM corporation. The content of this paper does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be inferred.

show that even though we use a general purpose integer programming tool, there exists a formulation that can be solved very efficiently. Comparisons with similar 0–1 problems and their special purpose solvers indicate that our results can be improved on significantly as well if a special purpose solver is used.

Keyword Codes: C.1.2; D.2.6; D.3.4
Keywords: Multiple Data Stream Architecture (Multiprocessors); Programming Environment; Processors – compilers, optimization

1 Introduction

The advent of languages like High Performance Fortran (HPF) [20], in which the programmer specifies parallelism implicitly by specifying the layout of an application's data across the processor array, has focused renewed attention on the problem of choosing a good data layout for parallel execution. Many experts believe that the choice of data layout is one of the two most important steps, in addition to choice of a suitable algorithm, toward a successful parallel implementation.

However, many programmers are not sure what data layout to choose. Many complex considerations must be taken into account if the program is to perform at high efficiency. For example, it is almost essential to consider the program as a whole rather than a series of independent subroutines. Of particular importance and complexity is the problem of determining when dynamic data redistribution will enhance overall performance.

Fortunately, the designers of languages like HPF and Fortran D [14], by requiring that data layout specifications be provided by the programmer, have opened the door for powerful new tools which can use intensive computation to determine a first approximation to a good data layout automatically. Because these tools are not embedded in the compiler and will be run only a few times during the implementation phase of a project, they can use techniques that would be considered too computationally intensive for inclusion in compilers, even on today's powerful supercomputers. Furthermore, by providing a high-level target language for these tools, HPF and similar languages have dramatically simplified the implementation of data layout tools.

In this paper, we will describe a data layout tool that uses a number of techniques from linear and integer programming. Integer programming is required because the automatic data layout problem that we solve is NP-complete. Evidence will be presented that our approach will be efficient enough for use in a programming assistance tool.

Our ability to solve integer programming problems has been remarkably improved over the last five to ten years, particularly pure 0–1 integer problems such as those being generated here. The basic technique for solving integer problems is to apply intelligent branch-and-bound using linear programming at the nodes. Important improvements have come in three areas. First, linear programming codes are on average approximately two orders of magnitude faster than they were five years ago, particularly for larger problems [6]. Combined with the improvements in computing speed over that same period these codes represent an approximate four orders of magnitude improvement in our ability to solve linear programming problems.

The second major development is in so-called cutting-plane technology. Motivated by work of Dantzig, Johnson and Fulkerson in the 50's [12], Padberg, Groetschel and others have shown how cutting-plane techniques could be used to strengthen the linear programming relaxations of many pure 0–1 integer programming problems [25]. The strengthening is effected by studying the facets of the underlying polytope generated by the convex hull of 0–1 solutions. Knowledge of these facets leads to subroutines for recognizing inequalities violated by the current fractional solution. These violated inequalities can then be added to the linear programming formulation in lieu of branching.

The third major area of improvement has come in the application of parallel processing to handle the branching when cutting planes do not succeed in sufficiently strengthening the linear programming formulation. Parallelism is particularly appropriate for current cutting-plane methods because cuts are computed not only at the root node but at all nodes in the branching tree. The extra computation at the nodes has the effect of making the computations sufficiently coarse grained that communication costs need not be significant. The most striking example of an integer programming success story exploiting all of the above advances is the recent work of Applegate, Bixby, Cook and Chvatal in which a 4461 city traveling salesman problem was solved to exact optimality using a complex branch-and-cut code running on a network of up to 60 loosely connected workstations [3].

2 Framework for Automatic Data Layout

The choice of a good data layout for a program depends on the compilation system, the problem size, the number of processors used, and the performance characteristics of the target machine architecture. Our proposed framework for automatic data layout will determine a good data layout for a given compilation system. Although this framework is not designed to be used inside the compilation system, it knows about the transformations and optimizations performed by the compilation system. The data layout is optimized for a target distributed-memory machine, a specific problem size, and the number of available processors, which implies that these entities have to be known at tool invocation time.

This section gives an overview of the proposed framework for automatic data layout for regular problems. Typically, regular problems represent data objects as dense arrays as opposed to a sparse representation. Regular problems allow the compilation system to determine the communication requirements and to perform a variety of program optimizations at compile time. The framework assumes that different data layouts can be specified for different program sections. Our proposed strategy for automatic data layout in the presence of dynamic data remapping consists of several steps, each discussed briefly in the remainder of this section. A detailed program example can be found in Section 3.

2.1 Overview

The initial step partitions the program into code segments, called program *phases*. Data remapping is allowed only between phases. A *phase* is the outermost loop in a loop nest such that the loop defines an induction variable that occurs in a subscript expression of an array reference in the loop body. This operational definition does not allow the overlapping or nesting of phases. Other strategies for identifying program phases are a topic of current research. Note that loop transformations such as loop fusion or loop distribution can change the partitioning of a program into phases. The discussion of transformations in the context of phase recognition is beyond the scope of this paper.

The phase structure of the program is represented in the *phase control flow graph*, an augmented control flow graph [1] where each phase is represented by a single node. The graph is annotated with branch probabilities and loop control information.

In the second step, the *program alignment space* is determined. The program alignment space corresponds to a single HPF template or a single Fortran D decomposition. All alignments and distributions are specified based on this unique program alignment space.

In the next two steps, data layout search spaces are constructed for each phase. A data layout for a single phase is specified by the alignment and distribution of all arrays referenced in the phase. First, alignment analysis builds a search space of reasonable alignment schemes for each phase. If arrays have fewer dimensions than the program alignment space, alignment analysis may generate different embeddings for the arrays. Then, distribution analysis uses the alignment search spaces to build candidate data layout search spaces of reasonable alignments and distributions for each phase. A preliminary discussion of possible pruning heuristics and the sizes of their resulting search spaces can be found in [16].

After the generation of the search spaces, a single candidate data layout is selected for each phase, resulting in a data layout for the entire program. This step solves the so-called *inter-phase* data layout problem. The inter-phase data layout problem will be described in more detail in the next section. The selected candidate layouts have minimal overall cost. The overall cost is determined by the costs of each selected candidate layout, and the required remapping costs between selected candidate layouts. Note that the optimal data layout for a program may consist of candidate data layouts that are each suboptimal for their phases. The selection process is based on static performance estimates of the candidate data layouts and of data remappings between layouts. A static performance estimator suitable for automatic data layout has been discussed elsewhere [4, 14]. The performance estimator will mimic the compilation process for single phases. It will use the training set approach to determine communication and computation costs of the target machine architecture.

In the final step, data layout specifications are generated. The program alignment space is represented by a single `decomposition` or `template` statement. A data flow problem over the phase control flow graph is solved to avoid redundant specifications of alignments and/or distributions for the selected candidate data layouts. The final data layout specifications are translated into corresponding `align` and `distribute` statements.

2.2 Inter-phase Data Layout Problem

The inter-phase data layout problem is modeled as an optimization problem over the *data layout graph*. The data layout 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. A data flow problem over the phase control flow graph can be solved to determine these probabilities. The proposed data flow problem is similar to a reaching definitions problem [1] that additionally determines the probability that a definition reaches a given point in the program.

The initial formulation of the inter-phase data layout problem as an optimization problem over the data layout graph does not model the possible overlap of communication and computation between phases. However, the static performance estimator for each single phase will take the effect of compiler generated wavefronts inside a phase into account [27]. In addition, the initial formulation requires 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 current research.

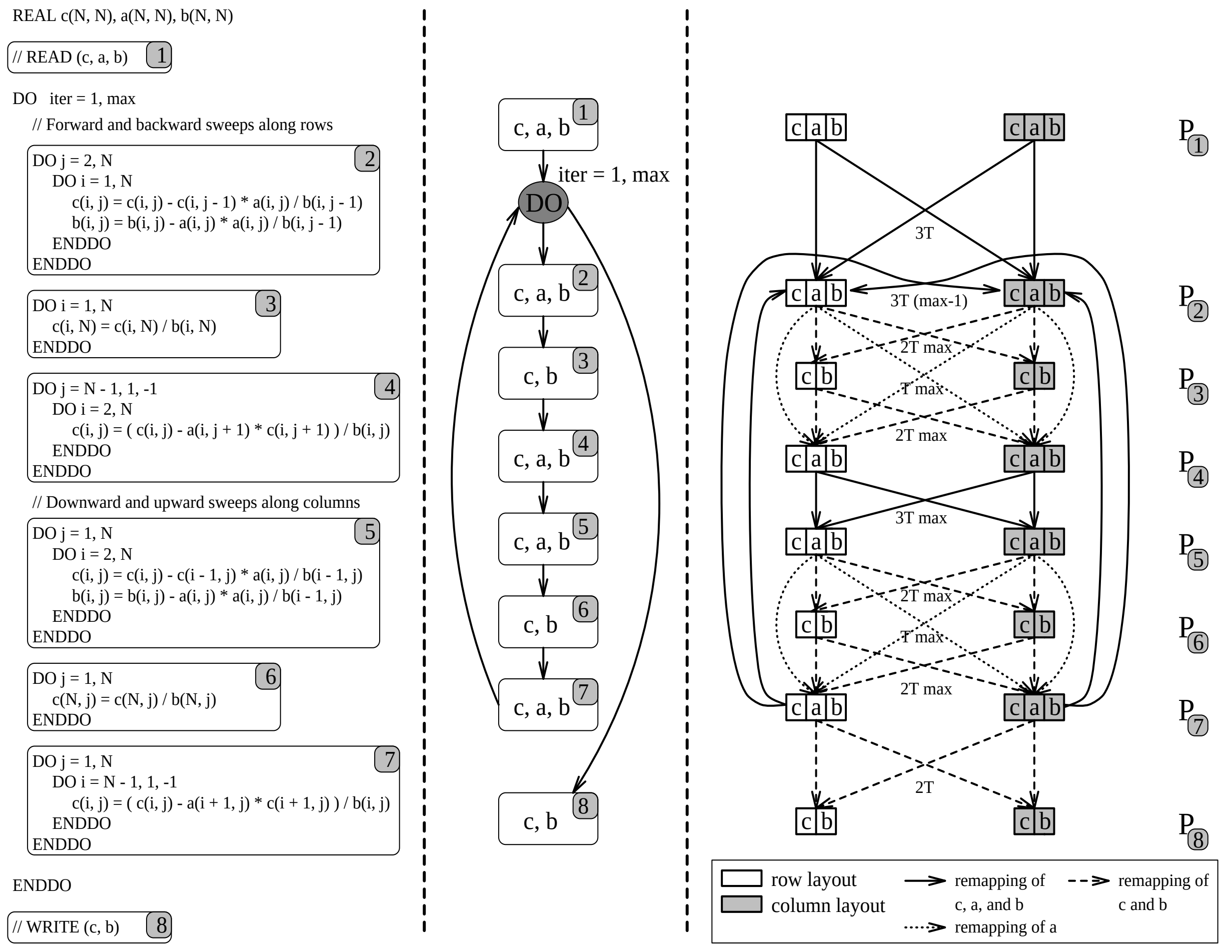
The inter-phase data layout problem is proven to be NP-complete [21]. The proof is based on a reduction from the 3-CNF satisfiability problem (3-SAT) [11]. However, in the special case where each candidate layout specifies a mapping for every array in the program, the inter-phase data layout problem can be solved in polynomial time in the size of the data layout graph. The polynomial time algorithm uses dynamic programming to solve multiple single-source, shortest path problems over the data layout graph [22].

In this paper, we focus on methods to compute the optimal solution of the inter-phase data layout problem with only a single copy of each array at any time during the execution of the program. An instance of the inter-phase data layout problem is translated into a 0–1 integer programming problem suitable to be solved by *CPLEX*[2], a linear integer programming tool partly developed by Robert Bixby at Rice University [5]. We give experimental results for our 0–1 integer programming formulations for an 800 line benchmark code developed by Thomas Eidson at ICASE.

3 Example Program

The following example illustrates the framework for automatic data layout. Figure 1A shows an Alternating Direction Implicit (ADI) integration kernel. ADI integration is a technique frequently used to solve partial differential equations (PDEs). The operational phase definition partitions the code into eight phases (see Section 2.1). The first phase and the last phase are loop nests that perform input and output operations, respectively. The resulting phase control flow graph is shown in Figure 1B. The program has a two-dimensional alignment space of size `N` in each dimension. To simplify the example, we assume that alignment analysis builds alignment search spaces that map the three arrays `a, b`, and `c` canonically onto the program alignment space. To simplify the example even further, we assume that distribution analysis generates only two candidate data layouts for each phase, namely a row layout and a column layout. The resulting data layout graph is shown in Figure 1C. The edges represent possible remappings of arrays between candidate data layouts. To model the effects of the iterative loop, the data layout graph contains edges between the layouts of the seventh phase and the second phase. The node and edge weights are the estimated costs for phase execution and remapping, respectively, in terms of execution time, multiplied by their predicted execution frequencies. The cost for transposing a single array is denoted by `T.max`

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



(A) (B) (C)

Figure 1: ADI integration kernel with three array variables. A: Source code with phase partitioning. There are eight phases. The loops representing the first and the last phase are not shown. Each phase references either two or three arrays. B: The phase control flow graph for the phase partitioning. C: The data layout graph for the candidate data layout search spaces of the phases. To simplify the example, we assume that there are only two candidate data layouts in each search space. Weights represent static performance estimates of overall execution times. Node weights are not shown. Unlabeled edges have zero weight. `T` is the cost of performing a single array transpose, and `max` is the number of iterations of the outermost loop of the ADI integration kernel.

is the number of iterations in the iterative loop. A static performance estimator together with the computed phase execution frequencies will be used to determine the node and edge weights [4, 14].

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

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 row and column sweeps, i.e. between phases 4 and 5, and between phases 7 and 2, eliminates communication within phases. In contrast, choosing the same data layout for both, row and column sweeps will avoid communication between phases but will make communication necessary inside some of the phases. The right 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 pipelined parallelism efficiently. In addition, the performance characteristics of the underlying I/O system has to be considered for the candidate layouts of the first and the last phase. Figure 2 shows two different data layout specifications that may be generated by the automatic tool. The left hand side shows a static, column-wise data layout. The right hand side depicts a dynamic data layout where transpose operations will be performed between the row and column sweeps.

4 Related Work

The problem of finding an efficient data layout for a distributed-memory multiprocessor has been addressed by many researchers [2, 8, 9, 10, 13, 15, 17, 18, 19, 23, 24, 26, 28]. The presented solutions differ significantly in the assumptions that are made about the input language, the possible set of data layouts, the compilation system, and the target distributed-memory machine. Even though many researchers have recognized the need for dynamic remapping and are planning to develop solutions, our work is one of the first to provide a framework for automatic data layout that considers dynamic remapping.

Knobe, Lukas, and Dally [19], and Chatterjee, Gilbert, Schreiber, and Teng [9, 10] address the problem of dynamic alignment in a framework particularly suitable for SIMD machines. Anderson and Lam discuss techniques for automatic data layout for distributed and shared address space machines [2]. Their approach considers dynamic remapping. Lee and Tsai propose a dynamic programming algorithm to determine a data layout for a sequence of loop nests, allowing remapping between the loop nests [23].

In contrast to most of the previously published work, our framework is designed to work in the context of a programming assistance tool, not inside a compiler. As a consequence, the framework can use techniques that may be too expensive to be included in a compiler.

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

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

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

            . . .

    // Downward and upward sweeps along columns

            . . .

ENDDO
        . . .
```

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

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

Figure 2: Example output of the proposed framework for automatic data layout for the ADI integration kernel. The left hand side shows a static column-wise data layout, and the right hand side shows a dynamic layout that performs transposes between the sweeps along rows and columns.

5 Inter-phase Data Layout as a 0–1 Problem

This section discusses the details of translating an instance of an inter-phase data layout problem and its data layout graph into an instance of a 0–1 integer programming problem with linear constraints, or *0–1 problem* for short. For the purpose of this discussion, we assume that the input program has n phases, $P_1, \ldots P_n$. The corresponding data layout graph has m_i nodes for each phase i, $1 \leq i \leq n$. Remember that each node represents a particular candidate data layout that specifies the alignment and distribution of all variables referenced in the phase.

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

The translation introduces a variable for each node and edge in the data layout graph. 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 inter-phase data layout 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 data layout graph, and X_{remap} has one switch for each edge. The switch $x_{ik} \in X_{layout}$ represents the k-th node of the i-th phase. The switch $x_{ik}^{jl} \in X_{remap}$ represents the remapping edge between the l-th node of phase j and the k-th node of phase i.

Similar to the variable set X, the set of constraints is partitioned into two classes. Constraints that ensure the selection of only a single node for each phase are called *layout constraints*. 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

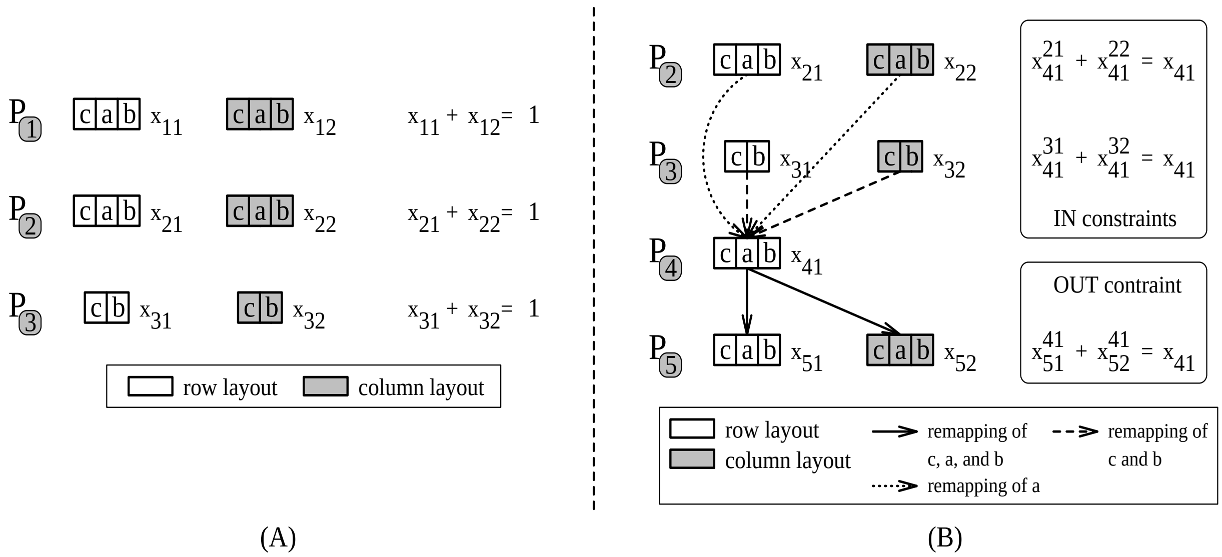


Figure 3: A: Layout constraints for the first three phases. B: Remapping constraints for the first candidate layout in the fourth phase. Switch x_{41} represents this layout.

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 example data layout graph (Figure 1C) are shown in Figure 3A.

Remapping constraints guarantee that all remapping edges between selected nodes are considered, i.e. are also selected. For each node in the data layout graph, there are two types of remapping constraints, namely IN-constraints and OUT-constraints.

For the node represented by x_{ik} and all incoming edges with nodes in the same phase j as their sources, IN-constraints of the form $\sum_{l=1}^{m_j} x_{ik}^{jl} = x_{ik}$ are generated. In other words, if switch x_{ik} is *on* then exactly one switch representing an incoming edge from phase j must be *on*. If x_{ik} is *off* then all incoming edge switches have to be *off*.

Similarly, for the node represented by x_{ik} and all outgoing edges with nodes in the same phase j' as their sinks, OUT-constraints of the form $\sum_{l'=1}^{m_{j'}} x_{j'l'}^{ik} = x_{ik}$ are generated. The remapping constraints for the node representing the first layout in the fourth phase of the ADI integration example, x_{41}, are listed in Figure 3B.

A solution s_{01} of an instance of the 0–1 problem *minimizes* the following objective function under the above constraints:

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

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

Note that we also investigated other valid formulations of the data-layout problem as a 0–1 problem. However, these other formulations turned out to be inferior to the presented formulation in terms of the time *CPLEX* needed to compute the optimal solution. A more detailed discussion of the different formulations can be found elsewhere [7].

6 Experiments

All of our experiments are based on ERLEBACHER, a 800 line benchmark program written by Thomas Eidson at the Institute for Computer Applications in Science and Engineering (ICASE). The program performs 3-dimensional tridiagonal solves using Alternating Direction Implicit (ADI) integration. The code contains computational wavefronts across all three dimensions. Array kill analysis was performed by hand and arrays were renamed and replicated appropriately. The resulting program contains 40 phases and 25 arrays. There are arrays with one, two, and three dimensions.

ERLEBACHER has a three-dimensional alignment space. For our experiments, we assume that alignment analysis and distribution analysis generate seven candidate layouts for each phase, one layout for each possible combination of distributed dimensions. However, if a phase contains only one-dimensional arrays, its candidate search space has only four layouts since some layouts are the projection of two distribution schemes. The corresponding data layout graphs with different weights were generated by hand. Weights were chosen to model different communication costs and the presence or absence of compiler optimizations. For instance, a compiler may be able to generate a coarse-grain pipelined loop if the data layout induces cross-processor dependences [27]. Whether the compiler performs such an optimization or not is represented by different weights given to the nodes.

We wrote a tool that generates the 0–1 problem formulation (see Section 5) for a given, weighted data layout graph. The remapping costs for individual arrays are handed to the tool as a cost table. The tool automatically generates the edge weights of the corresponding data layout graph based on this cost table. The generated 0–1 problems have 253 layout switches, 2133 remapping switches, and 715 layout and remapping constraints.

For the experiment, twelve 0–1 problems were automatically generated. Each of the 0–1 problems was solved by *CPLEX*, a linear integer programming tool. *CPLEX* includes an implementation of a general-purpose branch-and-bound code for mixed integer programming. Being general purpose, this code does not exploit the structural properties of our particular 0–1 problems. The experiments show that our 0–1 formulation can be solved by the general-purpose *CPLEX* in less than 4 seconds on average on a SPARC-10 workstation. For one 0–1 problem instance, *CPLEX* determined the optimal solution in 2.6 seconds. *CPLEX* did not take longer than 4.8 seconds on any of the twelve 0–1 problem instances.

0–1 integer programming is NP-complete. Therefore, it is unrealistic to expect a solution for all instances in minimal computation time. However, 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. Recent work on the TSP again provides a good example of what one can hope for. Using the well-known TSPLIB test set of problems, we have been able to solve to exact optimality all instances with fewer than 2000 cities, with the notable exception of one 225 city instance for which our cutting plane methods simply do not seem to be effective. However, it is interesting to note that for this one instance, it is symmetry that makes that problem difficult for our algorithms. That very symmetry implies that various heuristic procedures easily find the optimal solution (provably optimal by an independent analysis of problem structure). For the problem at hand, namely the inter-phase data layout problem, a similar approach is proposed in which inexact heuristic procedures would be applied if integer programming fails to find a solution within acceptable time limits. We remark in this context that we would expect our branch-and-cut algorithm to be computing both upper and lower bounds as it proceeds. If the computation is terminated prior to optimality, these bounds would provide estimates of the solution quality.

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, such as a special purpose solver for the inter-phase data layout problem.

7 Future Work

So far, the discussion of the our proposed framework has ignored procedure calls. All steps of the framework as discussed in Section 2.1 will have to be performed interprocedurally. For the inter-phase data layout problem, we plan to use the following approach in the absence of recursion. The data layout graphs of subroutines can be propagated bottom-up along the edges of the call graph, resulting in a single data layout graph for the entire program. If the compilation system performs procedure cloning, a distinct copy of a procedure's data layout graph is propagated along each edge in the call graph. This strategy has been used to hand generate the data layout graph of the ERLEBACHER code used for the experiments discussed in Section 6. In the absence of cloning, each procedure is represented by a single copy of its data layout graph in the data layout graph for the entire program.

We are currently investigating the support of data replication in our framework. There are two forms of data replication. Data replication that consists of multiple copies of an array with distinct owners for each copy will be handled during the construction of the candidate search spaces, i.e. search spaces may contain candidate layouts that specify multiple owners of an array. Data replication that refers to multiple copies of an array with only a single owner can be modeled by additional remapping edges in the data layout graph. The IN and OUT-constraints of our presented 0–1 problem formulation will have to be slightly modified to accommodate the new edges. Additional constraints can be used to enforce an upper bound on the number of copies of an array.

The proposed framework for automatic data layout is currently being implemented as part of the D programming environment. The implementation will provide the basis for the validation of our proposed techniques.

8 Conclusion

We have presented an approach to automatic data layout in the context of a programming tool that produces High Performance Fortran or a similar language as output. This has permitted us to explore exact solutions to the problem of automatic data layout, even though our formulation of the problem is NP-complete. Through the use of the latest and most powerful general purpose techniques for linear and integer programming, we have shown the technique to be practical for a full-sized application.

Recent experiences with similar NP-complete problems indicate that special purpose linear and integer programming techniques can be used to compute the exact solution even faster. These special purpose techniques will take advantage of the particular structure of our formulation of the data layout problem.

Acknowledgement

We would like to thank Hristo Djidjev, Reinhard von Hanxleden, and John Mellor-Crummey for providing many helpful comments.

References

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

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

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

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

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

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

[7] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0–1 integer programming. Technical Report CRPC-TR93-349-S, Center for Research on Parallel Computation, Rice University, November 1993.

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

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

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

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

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

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

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

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

[16] K. Kennedy and U. Kremer. Initial framework for automatic data layout in Fortran D: A short update on a case study. Technical Report CRPC-TR93-324-S, Center for Research on Parallel Computation, Rice University, July 1993.

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

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

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

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

[21] U. Kremer. NP-completeness of dynamic remapping. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[22] U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle. Automatic data layout for distributed-memory machines in the D programming environment. In C.W. Keßler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 136–152. Verlag Vieweg, Wiesbaden, Germany, 1993.

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

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

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

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

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

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