Automatic Data Layout
Using 0-1 Interger Programming

Robert Bixby
Ken Kennedy
Ulrich Kremer

CRPC-TR93349-S
November 1993

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

Automatic Data Layout Using 0–1 Integer Programming*

Robert Bixby
Dept. of Computational and Applied Mathematics
Rice University

Ken Kennedy
Dept. of Computer Science
Rice University

Ulrich Kremer†
Dept. of Computer Science
Rice University

Abstract: The goal of languages like Fortran D or 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 will not be able to compensate for a poorly chosen data layout since many compiler optimizations 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 for regular problems 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 a search space 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 such that their overall cost is minimal has been shown to be NP-complete. Instead of resorting to heuristics to determine a possibly suboptimal selection of data layouts, 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 show that even though we use a general purpose integer programming tool, there is a formulation that can be solved very efficiently. Similar 0–1 problems have been significantly improved by using a special-purpose solver, which indicates that such an approach could be used with this problem as well.

*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 DARPA 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.

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

1 Introduction

The advent of languages like High Performance Fortran [Hig93], in which the programmer specifies parallelism implicitly by specifying the layout of an applications 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 novice programmers have no idea how to choose a good data layout, even in a language like HPF. This is because 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 [FHK+90], 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 an approach to building a data layout tool using a number of techniques from linear and integer programming. Integer programming is required because, as we show, the problem as we formulate it is NP-complete. Evidence will be presented that this approach will be efficient enough for use in a programming assistance tool.

There has been remarkable improvement in our ability to solve integer programs over the last 5 to 10 years, particularly pure 0–1 integer programs such as those being generated here. The basic technique for solving integer programs 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 5 years ago, particularly for larger problems [Bix93]. Combined with the improvements in computing speed over that same period these codes represent an approximate four orders of magnitude improvement in our ability to solve linear programming problems.

The second major development is in so-called cutting-plane technology. Motivated by work of Dantzig, Johnson and Fulkerson in the 50's [DFJ54], Padberg, Groetschel and others have shown how cutting-plane techniques could be used to strengthen the linear programming relaxations of many pure 0–1 integer programming problems [GH91, PR91, HP92]. 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 leau of branching.

The third major area of improvement has come in the application of parallel processing

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

2 Framework for Automatic Data Layout

This section is an overview of our proposed framework for automatic data layout for regular problems. 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.

The initial step of the proposed strategy for automatic data layout in the presence of dynamic remapping is to partition the program into code segments, called program *phases*. Phases are intended to represent program segments that perform operations on entire data objects. Data remapping is allowed only between phases. An operational definition of a phase is given elsewhere [KMCKC93]. Strategies for identifying program phases are a topic of current research.

A data layout for a single phase is specified by the alignment and distribution of the arrays referenced in the phase. The data layout for an entire program consists of a collection of data layouts, one data layout for each phase in the program. This implies that an array has a unique data layout at any point in the program. The overall data layout is determined in three steps. First, alignment analysis builds a search space of reasonable alignment schemes for each phase based on the unique alignment space of the program. 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 [KK93]. Finally, a single candidate data layout scheme for each phase is selected, resulting in a data layout for the entire program. 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 [BFKK91]. The selection process must consider the tradeoff between the exploitable parallelism of data layouts for each phase and the remapping costs of data layouts between phases. This last step solves the so-called *inter-phase* data layout problem. The inter-phase data layout problem is proven to be NP-complete [Kre93]. The proof is based on a reduction from the 3-CNF satisfiability problem (3-SAT) [CLR90].

In this paper, we focus on methods to compute the optimal solution of the inter-phase data layout problem. An instance of the inter-phase data layout problem is translated

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

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
```

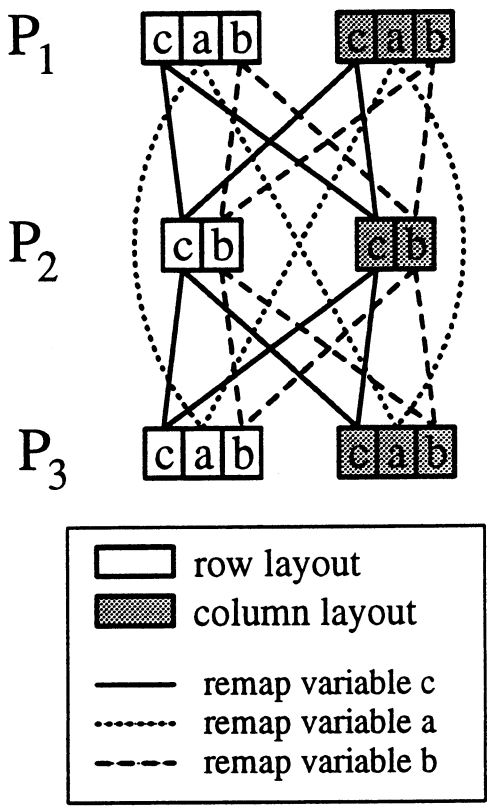

Figure 1: Example code fragment with three phases and two candidate data layouts per phase. In the data layout graph, nodes represent candidate layouts and edges represent possible remappings. Nodes and edges are associated with costs (not shown here).

into a 0–1 integer programming problem suitable to be solved by *CPLEX*[1], a linear integer programming tool partly developed by Robert Bixby at Rice University [Bix92]. We give experimental results for different 0–1 integer programming formulations for an 800 line ADI integration code.

3 Simple Example

The following example illustrates the framework for automatic data layout. The program on the left-hand side of Figure 1 shows a fragment of an ADI integration kernel, namely the forward and backward sweeps along rows. ADI integration is a technique frequently used to

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

solve partial differential equations (PDEs). Each loop nest is represented by a single phase. The code fragment has a two-dimensional alignment space. To simplify the example, we assume that alignment and distribution analysis generate only two candidate data layout schemes for each phase, namely a row layout and a column layout. The data layout graph (shown on the right-hand side of Figure 1) has one node for each candidate data layout. Edges represent possible remappings of variables between candidate data layouts. Nodes and edges have weights representing the cost of each layout and single variable remapping, respectively, in terms of execution time. A static performance estimator will determine the node and edge weights.

To solve the resulting 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.

For this simple example, choosing row layouts for all three phases results in the best performance since no communication is necessary inside phases or between phases. However, in the complete ADI integration kernel, the forward and backward sweeps along the rows is followed by corresponding downward and upward sweeps along the columns. For the sweeps along the columns, a column layout has the best performance. 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. In contrast, transposing the arrays between the row and column sweeps eliminates communication inside all 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.

4 Related Work

The problem of finding an efficient data layout for a distributed-memory multiprocessor has been addressed by many researchers [LC91, KLS90, KLD92, Gup92, Who91, CGST93, AL93, Keß93, PM93, DHR93, KD93, CHZ91, RS89, HA90, SS90, Sus91]. 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. Our work is one of the first to provide a framework for automatic data layout that considers dynamic remapping. However, many researchers have recognized the need for dynamic remapping and are planning to develop solutions.

Knobe, Lukas, and Dally [KLD92], and Chatterjee, Gilbert, Schreiber, and Teng [CGST93] address the problem of dynamic alignment in a framework particularly suitable for SIMD machines. More recently, Anderson and Lam [AL93] have proposed techniques for automatic data layout for distributed and shared address space machines. Their approach considers dynamic remapping.

In contrast to most of the published work, our framework for automatic data layout is designed to work in the context of a programming assistance tool, not inside a compiler. As a consequence, our framework can use techniques considered too expensive to be included in a compiler. Our framework is unique in the sense that it selects the data layout of a

program in two steps. First, it uses pruning algorithms to generate explicit alignment and distribution search spaces of reasonable sizes. In the second step, the final selection between the remaining data layouts is performed by solving an NP-complete problem, namely the inter-phase data layout problem. Preliminary experimental results indicate that computing the optimal solution of the inter-phase data layout problem is practical in the context of a programming tool.

5 Inter-phase Data Layout Problem

This section discusses the details of the mathematical formulation of the inter-phase data layout problem and its translation into distinct 0–1 integer programming problems. To simplify the discussion, all programs are assumed to have no control flow between phases, i.e. the program can be represented as a linear sequence of phases, $P_1, \ldots P_n$. Control flow issues will be discussed in Section 7.

5.1 Optimization Problem

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

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

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

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

Let $f_i : p_i \rightarrow \{1, \ldots n\}$ be a mapping that determines for each variable $v_t \in p_i$ the phase P_j such that $v_t \in p_j$, $j < i$, and for all $q \in \{j+1, j+2, \ldots i-1\}$, $v_t \notin p_q$ holds. If such a j does not exist, f_i returns the value i. The phase $P_{f_i(v_t)}$ is the phase that most recently referenced $v_t \in p_i$ with respect to phase P_i. If $f_i(v_t) = i$, then P_i contains the first reference to v_t in the program.

Let $g_i : p_i \rightarrow \{1, \ldots n\}$ be a mapping that determines for each variable $v_t \in p_i$ the phase $P_{j'}$ such that $v_t \in p_{j'}$, $j' > i$, and for all $q \in \{i+1, i+2, \ldots j'-1\}$, $v_t \notin p_q$ holds. If such a j' does not exist, g_i returns the value i. The phase $P_{g_i(v_t)}$ is the phase that will reference $v_t \in p_i$ next with respect to phase P_i. If $g_i(v_t) = i$, then P_i contains the last reference to v_t in the program.

Definition 1 *An instance of the inter-phase data layout problem consists of a program with a linear sequence of n phases, a set of program variables* $V = \{v_1, \ldots v_r\}$, *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 2 *A solution s_{ddl} of an instance of the inter-phase data layout problem is a set $s_{ddl} = \{d_1, d_2, ...d_n\}$ 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.

Definition 2 results in an optimization problem. As mentioned above, the problem of determining a solution s_{ddl} of an inter-phase data layout problem is NP-complete [Kre93].

5.2 0–1 Integer Programming Problem

This section describes the translation of an instance of the inter-phase data layout problem as defined in Definition 2 into an instance of a 0–1 integer programming problem with linear constraints, or *0–1 problem* for short.

Definition 3 *An instance of the 0–1 problem consists of a set of variables X, a set of linear constraints over the variables in X, and a linear objective function with domain X. A solution to an instance of the 0–1 problem is a function $s_{01} : X \rightarrow \{0, 1\}$ that minimizes the objective function while respecting the constraints.*

Let A be an instance of the inter-phase data layout problem. The translation function m maps A into its corresponding instance of the 0–1 problem, $m(A)$. The specification of m consists of the rules for constructing the variable set, the set of constraints, and the objective function.

To simplify the notation for constraints and objective functions, each variable $x \in X$ represents its value under a solution s_{01}. 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 candidate data layout in a phase, and X_{remap} has one switch for each possible remapping between phases. For all 0–1 problems discussed in this paper, X_{layout} is defined as follows:

$$X_{layout} = \{x_i^k \mid d_i^k \in D_i, 1 \leq i \leq n, 1 \leq k \leq m_i\}$$

Similar to the variable set X, the set of constraints is partitioned into two classes. Constraints that ensure the selection of only a single data layout for each phase are called *layout constraints*. For all 0–1 problems discussed in this paper, the layout constraints are defined identically. For each phase i, $1 \leq i \leq n$, there is a constraint of the form:

$$\sum_{k=1}^{m_i} x_i^k = 1. \qquad (1)$$

Remapping constraints guarantee that all remapping costs between selected data layouts are considered. A solution of an instance of the 0–1 problem must satisfy the constraints and therefore can be directly translated back into a solution of the corresponding dynamic data layout problem by choosing all candidate data layouts whose variables have been switched on:

$$s_{ddl} = \{\ d_i^k \mid s_{01}(x_i^k) = 1,\ 1 \leq i \leq n \ and\ 1 \leq k \leq m_i\}.$$

In the remainder of this section, several possible translation functions m are discussed.

5.2.1 Initial Node-based Formulation

This formulation is the direct translation of the inter-phase data layout problem as given in Definitions 1 and 2. The proof of the correctness of the formulation is given in Appendix A. Remapping constraints are constructed for each candidate data layout, i.e. each node in the data layout graph.

Remapping Variables: X_{remap} has one variable for each possible remapping of a single variable between candidate data layouts:

$$X_{remap} = \{x_{ij}^{klv_t} \mid v_t \in p_i \cap p_j, j = f_i(v_t), j \neq i, 1 \leq i \leq n, 1 \leq k \leq m_i, 1 \leq l \leq m_j\}$$

Remapping Constraints:

- Define $j = f_i(v_t)$ and $p_i^{in} = \{v_t | j \neq i\}$. For each phase i, $1 \leq i \leq n$, and choice k, $1 \leq k \leq m_i$, there is a constraint of the form:

$$\sum_{v_t \in p_i^{in}} \sum_{l=1}^{m_j} x_{ij}^{klv_t} = x_i^k |p_i^{in}| \qquad (2)$$

- Define $j' = g_i(v_t)$ and $p_i^{out} = \{v_t | j' \neq i\}$. For each phase i, $1 \leq i \leq n$, and choice k, $1 \leq k \leq m_i$, there is a constraint of the form:

$$\sum_{v_t \in p_i^{out}} \sum_{l'=1}^{m_{j'}} x_{j'i}^{l'kv_t} = x_i^k |p_i^{out}| \qquad (3)$$

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

$$\sum_{i=1}^{n} \sum_{k=1}^{m_i} x_i^k\ c(P_i, d_i^k) \ + \ \sum_{i=1}^{n} \sum_{v_t \in p_i} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} x_{ij}^{klv_t}\ c(d_{it}^k, d_{jt}^l),$$

where $j = f_i(v_t)$ and $j \neq i$.

5.2.2 Improved Initial Node-based Formulation

The improved formulation reduces the number of 0–1 variables needed, i.e. reduces the size of X. The correctness proof is analogue to the proof for the initial phase-based formulation.

Remapping Variables: Instead of including in X_{remap} one switch for each possible remapping of a *single* program variable between candidate layouts, a single switch represents all variable remappings between a possible pair of candidate data layouts (d_i^k, d_j^l), where $j = f_i(v_t)$ for some $v_t \in p_i$, $j \neq i$:

$$X_{remap} = \{x_{ij}^{kl} \mid 1 \leq i \leq n, \exists\ v_t \in p_i \cap p_j\ (j = f_i(v_t), j \neq i, 1 \leq k \leq m_i, 1 \leq l \leq m_j)\ \}$$

Remapping Constraints: The constraints have to reflect the changes to X_{remap}. The remapping constraints of type (2) and (3) are modified as follows:

- Define $P_i^{in} = \{j \mid \exists v_t \in p_i \cap p_j\ (j = f_i(v_t), j \neq i)\ \}$. For each phase i, $1 \leq i \leq n$, and choice k, $1 \leq k \leq m_i$, there is a constraint of the form:

$$\sum_{j \in P_i^{in}} \sum_{l=1}^{m_j} x_{ij}^{kl} = x_i^k\ |P_i^{in}| \qquad (2)$$

- Define $P_i^{out} = \{j' \mid \exists v_t \in p_i \cap p_{j'}\ (j' = g_i(v_t), j' \neq i)\ \}$. For each phase i, $1 \leq i \leq n$, and choice k, $1 \leq k \leq m_i$, there is a constraint of the form:

$$\sum_{j' \in P_i^{out}} \sum_{l'=1}^{m_{j'}} x_{j'i}^{l'k} = x_i^k\ |P_i^{out}| \qquad (3)$$

Objective Function: Define V_{remap}^{ij} as the set of all program variables that may be remapped between phases P_i and P_j as follows:

$$V_{remap}^{ij} = \{v_t \mid v_t \in p_i \cap p_j, 1 \leq i \leq n, j = f_i(v_t), j \neq i\}$$

Let $c(d_i^k, d_j^l)$ denote the cost of remapping all variables in V_{remap}^{ij} from candidate layout d_j^l of phase P_j to candidate layout d_i^k of phase P_i:

$$c(d_i^k, d_j^l) = \sum_{v_t \in V_{remap}^{ij}} c(d_{it}^k, d_{jt}^l).$$

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

$$\sum_{i=1}^{n} \sum_{k=1}^{m_i} x_i^k\ c(P_i, d_i^k)\ \ +\ \ \sum_{i=1}^{n} \sum_{j \in P_i^{in}} \sum_{k=1}^{m_i} \sum_{l=1}^{m_j} x_{ij}^{kl}\ c(d_i^k, d_j^l)\ .$$

5.2.3 Disaggregated Improved Node-based Formulation

In this formulation, the remapping constraints are disaggregated. This is the only difference between this version and the improved initial node-based formulation.

Remapping Constraints:

- Define $P_i^{in} = \{j \mid \exists v_t \in p_i \cap p_j \ (j = f_i(v_t), j \neq i)\ \}$. For each phase i, $1 \leq i \leq n$, for each choice k, $1 \leq k \leq m_i$, and for each $j \in P_i^{in}$ there is a constraint of the form:

$$\sum_{l=1}^{m_j} x_{ij}^{kl} = x_i^k \qquad (2)$$

- Define $P_i^{out} = \{j' \mid \exists v_t \in p_i \cap p_{j'} \ (j' = g_i(v_t), j' \neq i)\ \}$. For each phase i, $1 \leq i \leq n$, for each choice k, $1 \leq k \leq m_i$, and for each $j' \in P_i^{out}$ there is a constraint of the form:

$$\sum_{l'=1}^{m_{j'}} x_{j'i}^{l'k} = x_i^k \qquad (3)$$

5.2.4 Edge-Based Formulation

This formulation uses a different approach to remapping constraints. The correctness proof of the formulation is given in Appendix B. Remapping constraints are constructed for each possible remapping in the improved node-based formulation, i.e. for each edge in the corresponding data layout graph,

Remapping Constraints: For each phase i, $1 \leq i \leq n$, for each phase $j \in P_i^{in}$, for each choice k, $1 \leq k \leq m_i$, and for each choice l, $1 \leq l \leq m_j$, there is a constraint of the form:

$$x_i^k + x_j^l \geq 2\, x_{ij}^{kl} \quad \text{and} \quad x_i^k + x_j^l \leq 1 + x_{ij}^{kl}.$$

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, or three dimensions.

ERLEBACHER has a three-dimensional alignment space. For our experiments, we assume that alignment and distribution analysis generate 7 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 4 layouts since some layouts are the projection of two distribution schemes. The corresponding data layout graphs

with different weights were generated by hand. Weights were chosen to model different communication costs and the presence or absence of compiler optimizations. For instance, a compiler may be able to generate a coarse-grain pipelined loop if the data layout induces cross-processor dependences [Tse93]. Whether the compiler performs such an optimization or not is represented by different node weights.

We wrote a tool that generates the four distinct 0–1 problem formulations (see Section 5.2) for each input, weighted data layout graph. The remapping costs for individual arrays are given to the tool in the form of a cost table. The tool automatically generates the edge weights of the corresponding data layout graph based on this cost table.

The following table gives an insight into the sizes of the automatically generated 0–1 problems:

	initial node-based	**improved node-based**	**disaggregated**	**edge-based**
#layout variables	253	253	253	253
#remapping variables	3012	2133	2133	2133
#constraints	485	485	715	4306

For the experiment, 6 improved node-based, 12 disaggregated, and 6 edge-based 0–1 problem formulations 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 following table gives the solution times in seconds of the 24 0–1 problems using *CPLEX* on a SPARC-10. A "*" indicates that *CPLEX* took more than 30 minutes. The reported averages exclude these experiments. Initial node-based formulations are omitted since they are inferior to the improved node-based formulations.

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

The experiments show that the disaggregated formulation can be solved by the general-purpose *CPLEX* in less than 4 seconds on average. The improvement between the node-based and the disaggregated 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.

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 Other Issues

So far, the discussion of the inter-phase data layout problem has ignored procedure calls or control flow between phases. 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 systems 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 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 of the entire program.

Structured control flow between phases such as single entry/exit loops or branches can be easily represented in the data layout graph. Edges in the data layout graph represent possible communication between phases. To deal with control flow, edge weights are chosen to reflect the probability and frequency of the program's execution paths. The treatment of general control flow is a topic of current research.

8 Conclusions

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 take advantage of the particular structure of our formulation of the data layout problem.

Acknowledgements

We would like to thank Hristo Djidjev, Reinhard von Hanxleden, and John Mellor-Crummey for inspiring many of the ideas in this work.

References

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

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

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

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

[Bix93] R. Bixby. Progress in linear programming. Technical Report TR93-40, Dept. of Computational and Applied Mathematics, Rice University, September 1993.

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

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

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

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

[DHR93] Anne Dierstein, Roman Hayer, and Thomas Rauber. Automatic parallelization for distributed memory multiprocessors. In Christoph W. Kessler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 192–217. Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden, Germany, 1993.

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

[GH91] M. Grötschel and O. Holland. Solution of large-scale symmetric traveling salesman problems. *Mathematical Programming*, 51:141–202, 1991.

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

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

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

[HP92] K. L. Hoffman and M. Padberg. Solving airline crew-scheduling problems by branch-and-cut. 1992. preprint.

[KD93] K. Knobe and W. Dally. Subspace optimizations. In Christoph W. Kessler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 153–176. Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden, Germany, 1993.

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

[KK93] 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.

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

[KLS90] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed*

Computing, 8(2):102–118, February 1990.

[KMCKC93] U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle. Automatic data layout for distributed-memory machines in the D programming environment. In Christoph W. Kessler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 136–152. Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden, Germany, 1993. Also available as technical report CRPC-TR93-298-S, Rice University.

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

[LC91] 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.

[PM93] Michael Philippsen and Markus U. Mock. Data and process alignment in modula-2*. In Christoph W. Kessler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 177–191. Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden, Germany, 1993.

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

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

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

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

[Tse93] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, Houston, TX, January 1993. Rice COMP TR93-199.

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

Appendix

A Correctness of Initial Node-based Formulation

The translation introduces a distinct variable for each candidate data layout and each possible remapping of a single variable between different candidate data layouts. These variables are switches that can be turned on or off. The constraints ensure that

1. exactly one variable is switched on for each phase, i.e. only a single candidate layout is selected for each phase, and
2. all remapping variables are switched on for candidate data layouts that have been switched on, i.e. the remapping costs between selected candidate data layouts are considered, and
3. no other variables are switched on, i.e. no other costs are considered.

Constraints of type (1) ensure that any solution selects exactly one data layout for each phase. Constraints of type (2) and (3) guarantee the remaining two properties of the remapping switches as shown in Theorem 1.

The objective function of the 0–1 problem is a parameterized version of the cost function in Definition 2, where the parameters are the introduced switches.

Theorem 1 *Let A be an instance of a dynamic data layout problem and $m(A)$ the translation of A into a corresponding instance of the 0–1 problem. If s_{01} is a solution of $m(A)$ with variables $x_1^{k_1}, x_2^{k_2}, \ldots x_n^{k_n}$ switched on, $1 \leq k_i \leq m_i$ for $1 \leq i \leq n$, then all remapping switches have to be on between the selected data layouts and no other remapping switches are on.*

Proof: If the switch x_i^k is off, the right hand sides of its associated constraints of type (2) and (3) evaluate to 0. Therefore, all switches that model the remapping for data layout d_i^k, namely $x_{ij}^{klv_t}$ and $x_{j'i}^{l'kv_t}$, have to be off. In other words, only remapping switches between selected data layouts can be activated. If the switch x_i^k is on, the right hand sides of its associated constraints evaluate to the number of incoming remapping edges and the number of outgoing remapping edges for constraints of type (2) and (3), respectively. Therefore, a single incoming remapping switch and a single outgoing remapping switch for each variable $v_t \in p_i$ must be on, since only remapping switches between selected data layouts can be activated. In other words, all remapping switches between selected data layouts have to be activated.

□

B Correctness of Edge-based Formulation

Theorem 2 *Let $1 \leq i \leq n$, $1 \leq k \leq m_i$, $j \in P_i^{in}$, and $1 \leq l \leq m_j$. x_{ij}^{kl} is switched on iff x_i^k and x_j^l are both switched on.*

Proof:
"$\Longrightarrow$": Assume x_{ij}^{kl} is switched on. Due to constraint $x_i^k + x_j^l \geq 2\ x_{ij}^{kl}$, $x_i^k + x_j^l \geq 2$ has to hold. This is only possible if both, x_i^k and x_j^l, are switched on.

"$\Longleftarrow$": Assume x_i^k and x_j^l are switched on. Due to constraint $x_i^k + x_j^l \leq 1\ +\ x_{ij}^{kl}$, $1 \leq x_{ij}^{kl}$ has to hold. Therefore x_{ij}^{kl} must be switched on.
□