Algorithms for Automatic Alignment of Arrays *

Siddhartha Chatterjee † John R. Gilbert ‡ Leonid Oliker § Robert Schreiber ¶
Thomas J. Sheffler ||

*This work was performed while Chatterjee and Sheffler were postdoctoral scientists at RIACS, and Schreiber was a senior scientist at RIACS. This work was supported by the NAS Systems Division via Contract NAS 2-13721 between NASA and the Universities Space Research Association. Sections of this paper have previously appeared in the *Supercomputing '93* and *Frontiers '95* conferences.

†Department of Computer Science, Campus Box 3175, Sitterson Hall, The University of North Carolina, Chapel Hill, NC 27599-3175. sc@cs.unc.edu.

‡Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304-1314. gilbert@parc.xerox.com.

§Research Institute for Advanced Computer Science, Mail Stop T27A-1, NASA Ames Research Center, Moffett Field, CA 94035-1000. oliker@riacs.edu.

¶HP Labs 3L-5, Hewlett-Packard Company, 1501 Page Mill Road, Palo Alto, CA 94304-1126. schreiber@hpl.hp.com.

||Rambus Inc., 2465 Latham Street, Mountain View, CA 94040. sheffler@rambus.com.

Proposed running head: Algorithms for Automatic Alignment of Arrays
Direct all correspondence to:

Prof. Siddhartha Chatterjee
Department of Computer Science
CB #3175, Sitterson Hall
The University of North Carolina
Chapel Hill, NC 27599-3175
Phone: (919) 962-1766
Email: sc@cs.unc.edu

Abstract

When a data-parallel language like Fortran 90 is compiled for a distributed-memory machine, aggregate data objects (such as arrays) are distributed across the processor memories. The mapping determines the amount of communication needed to bring operands of parallel operations into alignment with each other. A common approach is to break the mapping into two stages: first, an *alignment* that maps all the objects to an abstract template, and then a *distribution* that maps the template to the processors. This paper describes algorithms for solving the various facets of the alignment problem: axis and stride alignment, static and mobile offset alignment, and replication labeling.

We show that optimal axis and stride alignment is NP-complete for general program graphs, so we study heuristic methods. We give a heuristic that can explore the space of possible solutions in a number of ways. We show that some of these strategies can give better solutions than a simple greedy approach proposed earlier. We also show how local graph transformations can reduce the size of the problem significantly without changing the best solution. This allows more complex and effective heuristics to be used.

We show how to model the static offset alignment problem using linear programming, and we show that loop-dependent mobile offset alignment is sometimes necessary for optimum performance. We describe an algorithm with for determining mobile alignments for objects within `do` loops. We also identify situations in which replicated alignment is either required by the program itself (via `spread` operations) or can be used to improve performance. We describe an algorithm based on network flow that determines which objects to replicate so as to minimize the total amount of broadcast communication in replication.

We present experimental results showing the effect of our axis/stride alignment algorithm on the performance of some example programs running on the CM-5.

List of symbols

Symbol	Interpretation
$\mathcal{I}$, $\mathcal{Z}$	Uppercase I and Z in calligraphic font
$\alpha, \beta, \phi, \theta, \sigma$	Lowercase Greek letters
ℓ	Lowercase letter l
$\in$	Set membership symbol
$\sum$	Summation symbol
∞	Infinity symbol

1 Introduction

Parallelism is expressed in data-parallel array languages like Fortran 90 [1] in the form of operations on arrays and array sections. Compiling such a program for a distributed-memory parallel machine requires a model for the mapping of the data to the machine. We view the mapping as an initial *alignment* to a Cartesian index space called a *template*, followed by a *distribution* of the template to the processors. The alignment phase positions all array objects in the program with respect to each other so as to reduce realignment communication cost. In the distribution phase that follows, the template is distributed to the processors. This two-phase approach separates the language issues from the machine issues, and is used in Fortran D [15], High Performance Fortran [19], CM-Fortran [29], and Vienna Fortran [6].

Placing arrays to enhance data locality is important when compiling array-parallel languages for distributed-memory parallel computers. Languages such as High Performance Fortran [19] require the user to provide data placement directives in the source code. There has also been considerable interest in automating the task of data placement [3, 8, 11, 22, 23, 24, 30]. This compiler optimization is important for insuring the portability of new scientific codes and for supporting old codes developed without a distributed memory model in mind.

Data placement optimization is a two-step process. First, the *alignment* phase examines relationships between array objects in a program and determines the manner in which corresponding array elements should be co-located to reduce communication costs. Second, the *distribution* phase partitions arrays across processor memories. Alignment deals with the relative positions of array objects in an architecture-independent framework, while distribution considers their absolute positions in a distributed memory.

The goal of compilation is to produce data and work mappings that reduce completion time. Much of this goal can be achieved by judicious alignment of the arrays. We consider only alignment here. Our work on automatic determination of distribution parameters can be found elsewhere [10, 27].

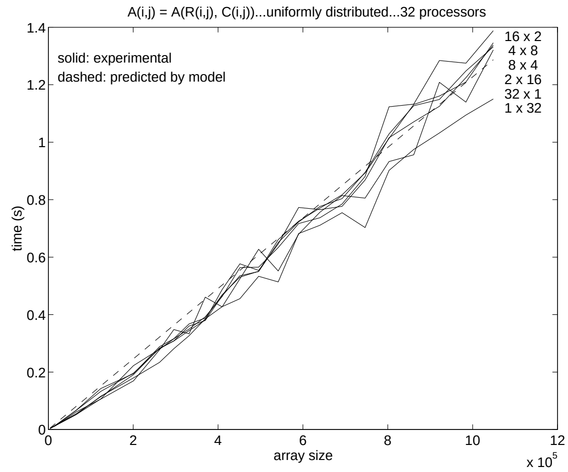
Completion time has two components: computation and communication. Communication can be separated into *intrinsic* and *residual* communication. Intrinsic communication arises from computational operations such as reductions that require data motion as an integral part of the operation. Residual communication arises from nonlocal data references required in a computation whose operands are not mapped to the same processors. As we only consider alignment in this paper, we take the view that objects are mapped identically to processors if and only if they are aligned. We use the term *realignment* to refer to residual communication due to misalignment; we seek to determine array alignments that minimize realignment cost. Communication for transpose, spread, and vector-valued subscript operations can in some cases be removed by suitable alignment choices. Our theory makes these forms of communication residual rather than intrinsic, and thus encompasses such optimizations [12].

1.1 A formal model of data layout

Alignment maps an array to a template with an affine mapping. The array coordinate $a \in \mathcal{Z}^d$ maps to the template coordinate $t \in \mathcal{Z}^t$ in the following way [2]:

$$Rt = La + f. \tag{1}$$

The matrix L encodes the orientation and spacing of array elements in the template, the column vector f encodes the offset of the array in the template, and the projection matrix R encodes the template axes along which the array is replicated. We constrain the matrix L to have exactly one nonzero per column and at most one nonzero per row. We call a matrix with exactly one nonzero in each column and at most one nonzero in each row a *D-matrix*. This implies that arrays cannot be collapsed or skewed with respect to the template. HPF allows the former option, which we feel is better cast as a distribution issue; Anderson and Lam [3] and Bau *et al.* [4] allow skewed alignment.



(a) (b)

Figure 1: Collective communication costs on the CM-5. (a) All-to-all communication. (b) Shift communication.

A similar linear algebraic formulation is available for distribution [2, 7]. We do not explicitly mention it here as distribution is beyond the scope of this paper.

1.2 A formal statement of the data layout problem

Given an array-parallel program and a target number of processors, our goal is to determine the quantities R, L, and f for each array and template at each point during program execution so as to minimize the completion time of the program on the parallel machine. This is a complex discrete optimization problem for several reasons: any realistic estimate of completion time is usually a nonlinear function of the free variables; the variables are coupled in their effect on completion time; and each variable has a large search space associated with it. We therefore use an approximate model of completion time containing only the most significant terms, develop a sequence of subproblems by weakening or outright ignoring the coupling among the variables, and resort to heuristic solutions for each of the subproblems. For the alignment phase, we have three such subproblems that we discuss in this paper. These subproblems correspond to the three unknowns in equation (1), and are called axis/stride alignment, offset alignment, and replication labeling.

We first determine the matrix L in equation (1) that encodes the axis/stride alignment parameters. Axis and stride play the biggest role in reducing communication costs because correcting axis and stride misalignment requires general all-to-all communication. Modeling the cost of this communication as the product of the data size and a large constant representing the per-unit cost of invoking the router is good enough for our analysis. (We call this the weighted 0-1 discrete metric.) Experimental evidence on the CM-5 presented in Figure 1(a) validates this model.

We then determine the matrix R in equation (1) that encodes the replication parameters. Replicating an array is performed using some kind of broadcast communication. We model this communication cost again as a metric on a space with two positions. We show that the problem of determining the optimal replication strategy can be reduced to a network flow probplem.

Finally, we determine the vector f in equation (1) that encodes the offset parameters. Our model of the cost of offset realignment assumes that it grows linearly in both the data size and in the number of template cells between the source and destination positions. (This is the weighted L_1 metric.) Experimental evidence on the CM-5 presented in Figure 1(b) validates this model for small shift distances.

```
real A(100,100), V(200)



do k = 1, 100

   A(k,1:100) = A(k,1:100) + V(k:k+99)
enddo
```

(a)

```
real A(100,100), V(200)
template T
align A(i,j) with T(i,j)

do k = 1, 100
   realign V(i) with T(k,i-k+1)
   A(k,1:100) = A(k,1:100) + V(k:k+99)
enddo
```

(b)

Figure 2: (a) A Fortran 90 program fragment requiring mobile alignment. (b) A mobile alignment for the program fragment.

In addition to static offsets, we also allow the components of f to be affine functions of loop induction variables. We call such an offset alignment *mobile*. Figure 2(b) shows an example of a mobile alignment for the code fragment of Figure 2(a). Note that V moves at each iteration of the loop; it has a mobile alignment.

We present an algorithm based on linear programming to determine static offset alignments, and extend it to determine good mobile alignments. Our program model provides a detailed and realistic model of realignment cost that accounts for control flow in loops. We present approximate solutions for mobile offset alignment for array objects occurring within loops, where we allow the offset alignment to be a compiler-determined affine function of loop induction variables.

1.3 Related work

Knobe, Lukas, and Steele [23] laid the foundation for data layout optimization of parallel programs. They addressed axis, stride, and offset alignment in a unified framework. Our algorithm for axis and stride alignment amplifies their claims of the importance of data layout optimization, and improves upon their methods in several ways.

First, we use a more comprehensive cost model. This is inherited directly from our alignment-distribution graph representation of data-parallel programs [11]. We also defer offset concerns to a later phase of alignment, because the shift communication needed to change offset is typically much less expensive than the general communication needed to change axis or stride.

Second, we develop a heuristic optimization framework that is more flexible than the strictly greedy algorithm of Knobe, Lukas, and Steele. Our experimental results confirm that the greedy heuristic can miss solutions that our algorithm finds.

Third, we show how to use local graph transformations to reduce the size of the optimization problem without changing its best solution. This reduction allows us to use more complex and effective heuristics than would be feasible for the unreduced graph.

In other related work, Li and Chen [24] addressed axis alignment alone, using a representation called a component affinity graph. Edges of this graph represent axis constraints to be satisfied. Their optimization algorithm is also greedy, but it is their cost model that most differentiates their work from ours. They formulate the problem as a graph with *large* and *small* weight edges, such that large edges are infinitely heavier than small edges. The optimization procedure finds a maximal weight set of edges that satisfy the constraints. Our cost model reflects the actual communication cost of a parallel program more accurately.

Huang and Sadayappan [20] introduced a linear algebraic formulation of the alignment problem. This is the basis of more recent papers by Anderson and Lam [3] and Bau *et al.* [4]. Anderson and Lam provide an inefficient iterative algorithm for solving these matrix equations. Bau *et al.* solve these equations by reducing this problem to the problem of computing a basis for the null space of an integer matrix. These authors permit a broader class of alignments than we do, but often sacrifice parallelism to reduce communication. The tradeoff between communication and parallelism is intimately related to parameters of the target machine. Our approach discovers alignment constraints that depend only on the source program, providing information that is useful on any target machine. As a result, we retain as much parallelism as is present in the source code.

Earlier, we developed an exact algorithm called compact dynamic programming [13] for finding minimum cost alignments of tree-structured computations (namely, expressions). We suggested using that algorithm as a heuristic for arbitrary programs, but experiments showed that it often makes poor alignment decisions because it uses only local information. Our present algorithm makes better use of global connectivity information.

Several other authors have considered static offset alignment [3, 18, 23, 24, 30]. Our earlier research [13, 12, 17] dealt with static alignment. We extend that work to handle mobile alignment here. Knobe, Lukas, and Steele [23] and Knobe, Lukas, and Dally [22] address the issue of dynamic alignment. Their notion of dynamic alignment is alignment depending on quantities whose values are known only at runtime, which may include loop induction variables as well as other arbitrary runtime values. This paper focuses on mobile alignment in the context of loops, where the alignment of an object is an affine function of the loop induction variables.

2 Representing the Alignment Problem

Previously, we developed a representation of data-parallel programs called the alignment-distribution graph (ADG) [11] to evaluate data layout decisions made during compilation. The ADG is based on static single assignment form [14], but incorporates a "position semantics" that makes each communication operation of the program explicit. This section reviews the ADG and then shows how to distill the ADG to a simpler graph that represents only axis and stride alignments.

2.1 The alignment-distribution graph

The *alignment-distribution graph*, or ADG for short, is a modified and annotated data flow graph. It is closely related to the static single-assignment form of programs developed by Cytron *et al.* [14]. Figure 3 shows the ADG for the program fragment in Figure 2.

Nodes in the ADG represent computation; edges represent flow of data. An endpoint of an edge is called a *port* and represents an array object with a specified position. Thus, an edge relocates an array object from one position to another. Alignments are associated with ports. Realignment occurs only when the two ports of an edge have different positions. A node constrains the relative alignments of the ports representing its operands and its results.

The ADG has a port for each (static) definition or use of an object. An edge joins the definition of an object with its use. Multiple definitions or uses are handled with merge, fanout, and branch nodes. Thus every edge has exactly two ports. All communication necessary for realignment is associated with edges; if the two ports of an edge have different alignments, then the edge incurs a cost that depends on the alignments and the total amount of data that flows along the edge during program execution.

Every array operation is a node of the ADG, with one port for each operand and one port for the result. Figure 3 contains examples of a "+" node representing elementwise addition, a *Section* node whose input is an array and whose output is a section of the array, and a *SectionAssign* node whose inputs are an array

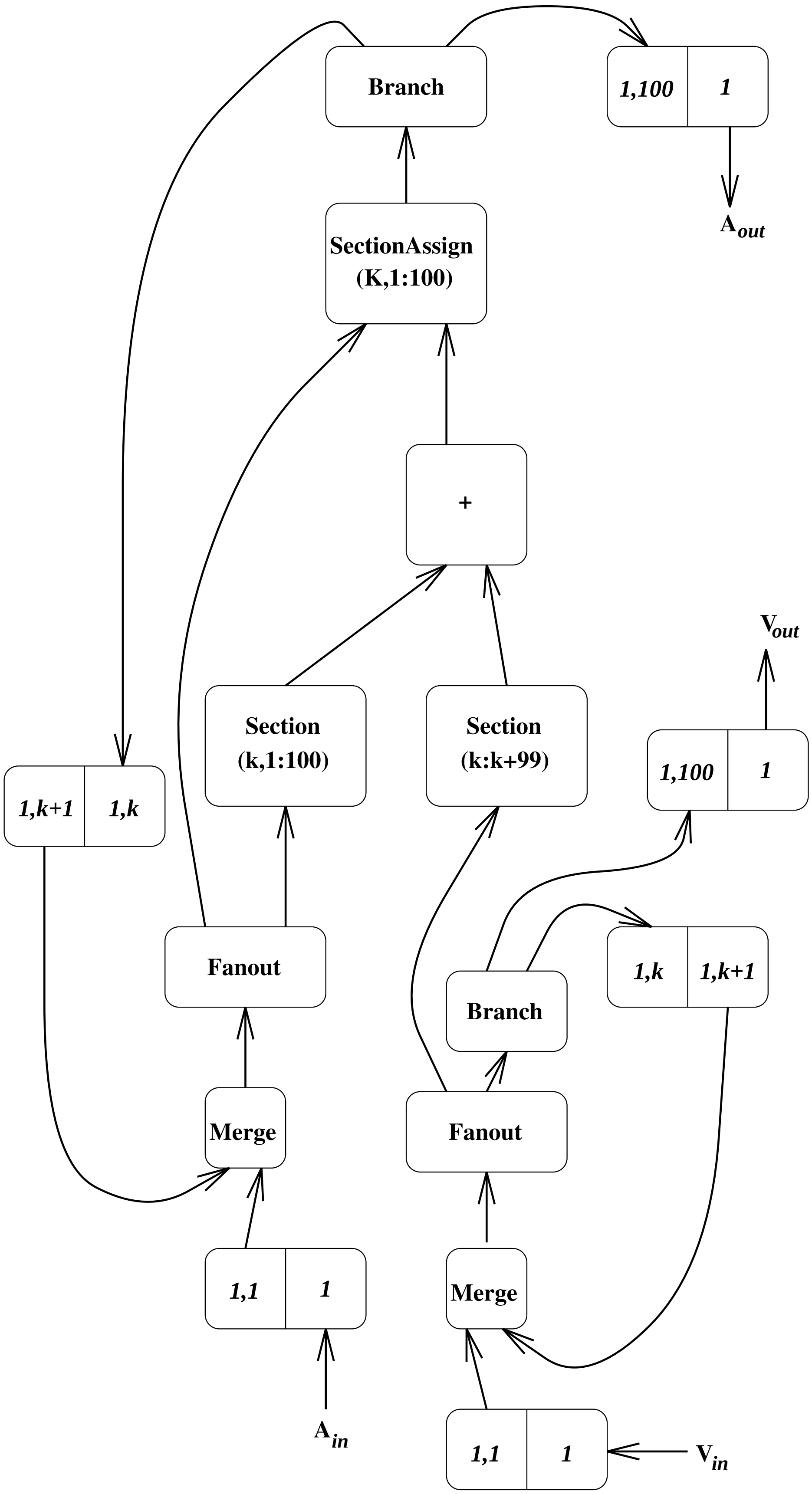


Figure 3: The ADG corresponding to the program fragment of Figure 2.

and a new object to replace a section of the array, and whose output is the modified array. (*SectionAssign* is called *Update* by Cytron *et al.* [14].)

When a single use of a value can be reached by multiple definitions, the ADG contains a *merge node* with one port for each definition and one port for the use. (This node corresponds to the ϕ-function of Cytron *et al.* [14].) When a single definition reaches multiple uses within the same basic block, the ADG contains a *fanout node*. When a single definition can reach multiple alternate uses (*e.g.*, due to conditional constructs), the ADG contains a *branch node*. Figure 3 contains examples of merge, fanout, and branch nodes. Finally, the ADG for a program with loops contains *transformer nodes* that delimit iteration spaces.

Nodes constrain the alignments of their ports. An elementwise operation like "$+$" constrains all its ports to have the same alignment. A merge or fanout node enforces the same constraint. If `A` is a two-dimensional array in a two-dimensional template, a node `transpose(A)` constrains its output to have the opposite axis alignment from its input; thus any communication necessary to transpose the array is assigned to the input or output edges rather than to the node itself. As a trivial example, this allows us to determine that `sum(transpose(A))` can be computed without actual transposition of the array A. *Section* and *SectionAssign* nodes enforce constraints that describe the position of a section relative to the position of the whole array; for example, the node for the section `A(10:50:2)` constrains its output object to have the same axis alignment as its input, twice the stride of its input, and an offset equal to 8 times the stride of A plus the offset of A.

The ADG represents data flow, not control flow. To model communication cost accurately, we must account for the fact that data can flow over a particular edge many times during the program's execution, and each time the data object may have a different size. Section 6 discusses how to model arbitrary control flow. Here we deal with the important special case in which the only control flow is in the form of `do` loops.

An edge inside a nest of k loops is labeled with a k-dimensional *iteration space*, whose elements are the vectors of values taken by the loop induction variables (LIVs). Both the size of the data object on an edge and the alignment of the data object at a port are functions of the LIVs, so they may vary over the iteration space.

For every edge that carries data into, out of, or around a loop, we insert a *transformer node* to describe the relationship between the iteration spaces at the two ports. Figure 3 contains examples. A loop-back transformer node, in a loop `do k = l:h:s`, constrains the alignment of its input as a function of $k + s$ to equal the alignment of its output as a function of k. Consider a $\frac{(1,k)}{(1,k+1)}$ transformer node as in Figure 3. An offset alignment of $2k + 3$ on the input ("k") port and of $2k + 1$ on the output ("$k + 1$") port satisfies the node's constraints. The $\frac{(1)}{(1,1)}$ transformer node on entry to this loop constrains its input position (which does not depend on k) to equal its output position for $k = 1$.

2.2 Cost model

Finally, we describe the communication cost of the program in terms of the ADG. A *position* is an encoding of a legal alignment. The *distance* $d(p, q)$ between two positions p and q is a nonnegative number giving the cost per element to change the position of an array from p to q. We require the set of all positions to be a metric space under the distance function d [13].[1]

We model the communication cost of the program as follows. Let E be the set of edges of the ADG, and let $\mathcal{I}_{xy}$ be the iteration space for edge (x, y). For a vector i in $\mathcal{I}_{xy}$, let $w_{xy}(i)$ be the data weight, which is the size of the data object on edge (x, y) at iteration i. Finally, let π be a feasible mobile alignment for the program—that is, for each port x let $\pi_x(i)$ be an alignment for x at iteration i that satisfies all the node constraints. Then the realignment cost of edge (x, y) at iteration i is $w_{xy}(i) \cdot d(\pi_x(i), \pi_y(i))$, and the total

[1]That is, for any distinct positions p, q, and r, we have $d(p,p) = 0$, $d(p,q) > 0$, $d(p,q) = d(q,p)$, and $d(p,q) + d(q,r) \geq d(p,r)$.

realignment cost of the program is

$$C(\pi) = \sum_{(x,y)\in E} \sum_{i\in\mathcal{I}_{xy}} w_{xy}(i) \cdot d(\pi_x(i), \pi_y(i)). \tag{2}$$

Our goal is to choose π to minimize this cost, subject to the node constraints.

2.3 Restrictions on mobile alignment functions

So far we have not constrained the form that mobile alignments may take. In principle, we could allow them to be arbitrary functions of the LIVs. For reasons of tractability, we consider only the (important) case in which mobile alignments of objects to be affine functions of the LIVs. Thus, the mobile offset or stride alignment function for an object within a k-deep loop nest with LIVs $i_1, \ldots, i_k$ is of the form $\alpha_0 + \alpha_1 i_1 + \cdots + \alpha_k i_k$, where the coefficient vector $\alpha = (\alpha_0, \ldots, \alpha_k)$ is what we must determine. We write this alignment succinctly in vector notation as αi^T, where $i = (1, i_1, \ldots, i_k)$. Both α and i are $(k+1)$-vectors. This reduces to the constant term α_0 for an object outside any loops.

Likewise, we restrict the extents of objects to be affine in the LIVs, so that the size of an object is polynomial in the LIVs.

3 Axis and Stride Alignment

In this section, we discuss algorithms for axis and stride alignment.

3.1 The constraint graph

An example ADG for a Fortran 90 code fragment appears in Figure 4. In the figure, positions are represented as the matrices, L_1, L_2 and L_3.

The ADG represents alignment and distribution for arrays in a parallel program. We now transform the ADG into a simpler graph, called the constraint graph (CG), that is specific to axis/stride alignment. The CG unifies the representations of positions and constraints, and succinctly captures the costs associated with each constraint. Each port in the ADG becomes a vertex in the CG. Edges and nodes in the ADG contribute constraints, which are represented as edges in the CG. Figure 5 illustrates the transformation of a line of source code into an ADG node and then into a CG, with positions and constraints given as matrices. The rest of this section describes the construction and use of the CG.

A constraint is a mapping from the coordinate space of one array object to another. If a constraint is imposed on the position of y with respect to x, then this constraint may be written as

$$L_y = L_x C_{xy},$$

where C_{xy} is a *constraint matrix*. Position matrices are D-matrices (defined in Section 1.1), and it turns out that constraint matrices are also D-matrices.

An ADG edge imposes an equality constraint (C_{xy} is an identity matrix) that may be violated for a specified cost. The cost W_{xy} reflects the communication cost in moving the array object from position x to position y.

An ADG node imposes constraints between the positions of its ports that cannot be violated (all constraints from ADG nodes have $W_{xy} = \infty$). For a node involving three or more ports, one is designated the reference port, and the constraints are expressed relative to it. The construction of constraint matrices for the various node types of the ADG is straightforward [11].

```
real A(1:100, 1:100)
r1 = reduce(A, dim=2)
r2 = reduce(A, dim=1)
s1 = spread(r1, dim=2, ncopies=100)
s2 = spread(r2, dim=1, ncopies=100)
out = s1 * s2
```

$$L_1 = \begin{bmatrix} 2 & 0 \\ 0 & 0 \\ 0 & 3 \end{bmatrix} \quad L_2 = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} \quad L_3 = \begin{bmatrix} 0 \\ 0 \\ 3 \end{bmatrix}$$

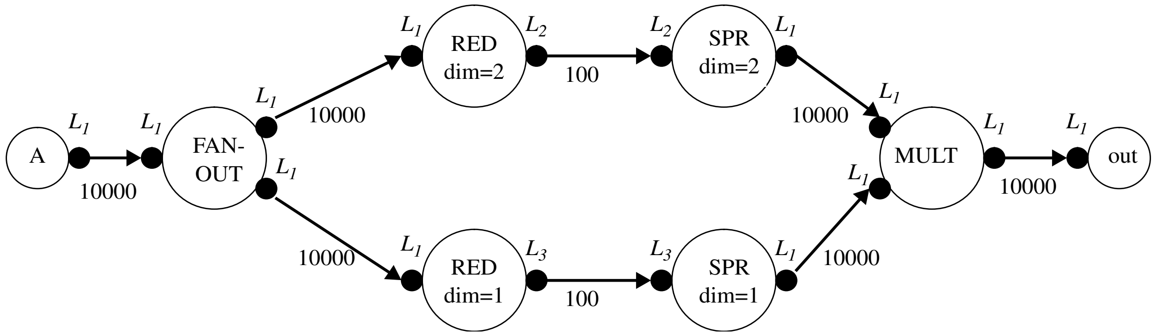
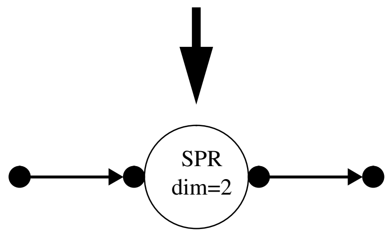


Figure 4: A code fragment using `reduce` and `spread` and its ADG. Data weights on edges represent the cost of communication. Each port has a position label. The ADG represents data flow in a program. It can also include nodes that reflect control flow due to branches or loops.

```
s1=spread(r1,dim=2,ncopies=100)
```



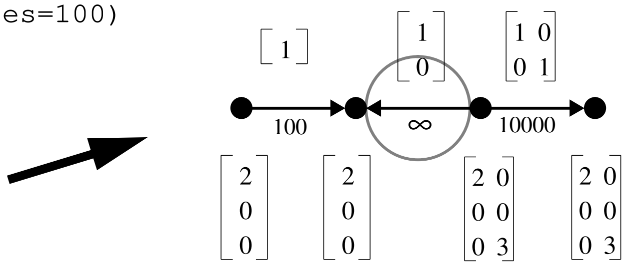


Figure 5: A code fragment, its translation into an ADG node, and the resulting constraint graph. Each port has a position (shown below) and each edge imposes a constraint (shown above). A position labeling *satisfies* an edge if the head position is the product of the tail position and edge constraint.

The CG is constructed from these constraints. For each array object x (that is, each ADG port), there is a vertex v_x. For each constraint $L_y = L_x C_{xy}$, there is a directed edge from v_x to v_y with label C_{xy}. Each edge also has an associated weight W_{xy}, which is the communication cost of moving an object from position L_x to L_y if the constraint is not satisfied.

A labeling of the CG is called *communication-free* if it satisfies every edge constraint, and a CG is called *satisfiable* if at least one such labeling exists.

This simple formulation captures all of the possible constraints pertinent to alignment analysis among array objects in High Performance Fortran. For example, constraint matrices can express relations between arrays that are projections, reductions, or sections of one another. The CG may be simplified even further. Section 3.3 discusses graph contraction operations that often reduce an alignment problem to a graph of only a few vertices.

3.2 An axis/stride labeling algorithm

The alignment problem is surprisingly hard. It is NP-complete even when restricted to only axis alignment for two-dimensional arrays in a two-dimensional template. Thus, we must be satisfied with heuristic or approximate solutions.

Theorem 1 *Min-cost labeling of an ADG is NP-complete, even considering only straight-line programs involving two-dimensional arrays with the transpose and addition operations.*

Proof: It is easy to see that min-cost labeling is in NP since the cost of a given axis labeling can be computed in polynomial time. We proceed by reduction from "Bipartite Subgraph (GT25)" [16], which is the following problem: Given a graph G and an integer k, is there a bipartite subgraph of G with at least k edges? (Equivalently, is there a way to 2-color the nodes of G that violates the color condition for at most $t = e - k$ edges?) This is NP-complete even if G has vertices of degree 2 and 3 only.

We first transform G into a graph with two kinds of edges, "opposite" edges whose endpoints are to be colored differently and "same" edges whose endpoints are to be colored the same. Each original edge of G is an "opposite" edge. Split each degree-2 node of G into two nodes joined by a new "same" edge. Split each degree-3 node of G into a 4-vertex star, each of whose leaves is incident on an original edge, with 3 new "same" edges. It is easy to see that G can be 2-colored in a way that violates at most t edges if and only if the transformed graph can be. Furthermore. since each "opposite" edge shares a degree-2 endpoint with a "same" edge, the transformed graph has an optimal 2-coloring that violates only "same" edges.

Now each degree-2 vertex is incident on one "opposite" edge and one "same" edge, and each degree-3 vertex is incident on three "same" edges. Construct an ADG by replacing each "opposite" edge and its two endpoint vertices with a transpose node, and by replacing each degree-3 vertex with an addition node. Each edge of the ADG is a "same" edge of the transformed graph. Each "opposite" edge of the transformed graph (i.e., each edge of the original graph) is a transpose node of the ADG. A min-cost labeling of the ADG corresponds to an optimal 2-coloring of the transformed graph that violates only "same" edges. □

3.2.1 Outline of the algorithm

Let G be a given CG. Like the algorithms of Knobe, Lukas, and Steele [23] and Li and Chen [24], our algorithm finds a maximal satisfiable subgraph of G by starting with the empty graph (which is trivially satisfiable) and growing it by adding edges one at a time while maintaining satisfiability at each step. Unlike the other algorithms, however, our algorithm is not strictly greedy. It can discard previously added edges as well as add new edges, and therefore will ordinarily explore a larger set of feasible solutions.

Our algorithm builds a maximal satisfiable subgraph G'. Initially G' contains all of the vertices of G, but all edges are excluded. At each step, an excluded edge e is conditionally added to G' and a function is-satisfiable is called to determine if a communication-free labeling exists for the augmented graph G'.

The optimization algorithm proceeds as follows:

1. Include an excluded edge e.

2. If the resulting graph is satisfiable (see Section 3.2.2) then accept the new edge and go to Step 1.

   Otherwise, find a minimum-weight cut set E in G' of edges between the endpoints of the edge e. The graph including e but with E removed is guaranteed to be satisfiable (see below). However, there may be edges in E whose inclusion does not prohibit satisfiability. Try including each edge in E back into the graph in turn, and retain in E only those edges that prohibit satisfiability. E is now a *minimal* set of edges whose removal allows a communication-free labeling of the graph with edge e.

3. If the weight of edge e is bigger than the total weight of edge set E then insert e in the graph and remove the edges in E. Otherwise, reject e and leave the graph as it was.

4. Repeat this procedure until no more edges can be added to the graph.

5. Find a labeling that satisfies the final graph (see Section 3.2.3).

Lemma 1 *In step 3, the graph with e included and E removed is satisfiable.*

Proof: Each path in G' describes a constraint between its endpoints. Multile paths between two vertices may describe incompatible constraints, in which case the graph is not satisfiable.

Prior to adding edge $e = (v, w)$, the graph G' is satisfiable. Thus, the constraint on edge e must be incompatible with that of some other path from v to w. Edge set E is a mincut set of edges breaking all paths from v to w. Removing E leaves no constraints between v and w to conflict with that of edge e. Thus, $G' + \{e\} - E$ is satisfiable. □

The procedure terminates because the weight of the current graph increases at every iteration (though its size may not).

3.2.2 The predicate is-satisfiable

Given a CG, G, is-satisfiable determines in linear time if there is a communication-free labeling of G. (It does not actually have to find the labeling.) The algorithm first decides axis satisfiability, then stride satisfiability.

Axis Satisfiability We first construct a new graph (that is closely related to the component affinity graph of Li and Chen [24]). Each vertex of the CG is split into one vertex for each axis of its array object. Directed edges are introduced between the vertices of this new graph corresponding to the non-zero elements of the constraint matrices of the CG. This transformation is illustrated in Figure 6 for a small constraint graph fragment.

We then find the connected components of this new graph. If two vertices of the same array object are in the same connected component, then there is no communication-free solution (because the two axes would have to be mapped to the same template axis). Otherwise, there is a communication-free axis alignment that assigns each connected component to a different template axis. (This may use more template axes than necessary—see Section 3.2.3.)

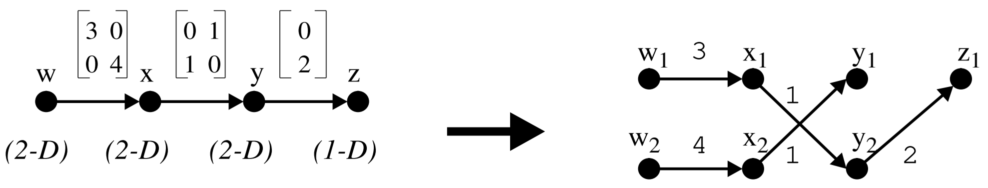


Figure 6: A new graph is constructed to verify the existence of a legal axis and stride assignment. One vertex is constructed for each axis of an array object (whose ranks are shown in italics), and edges map one axis to another with a given stride.

Stride Satisfiability For each connected component of the graph, the algorithm determines whether there is a labeling that satisfies the stride constraints by the following steps. An edge in this graph is satisfied if the product of its tail and stride labels equals its head label.

1. Find any spanning tree of the connected component.
2. Label an arbitrary vertex "1" and label the rest of the vertices by multiplying (or dividing) by the stride label of the edges.
3. For each non-tree edge, check whether the stride transformation it describes is satisfied by its endpoints.

The running times of both the axis and stride phases of this algorithm are linear in the number of edges in the graph. Connected components can be found in linear time by depth-first search. Checking the vertices corresponding to each array object to determine if any are in the same component is trivial. The stride constraints can be verified during the depth-first search: as each node is visited, propagate the appropriate stride value to each non-visited neighbor. For all other neighbors, verify that the constraint on the edge is satisfied.

This algorithm performs operations similar to the conformance checking procedure of Knobe, Lukas, and Steele [23], but is much simpler. They find axis conflicts by an incremental approach based on merging sets, and their stride satisfiability test involves complicated array allocation functions. These differences stem from the differences in the two optimization frameworks. Our approach can generate large-scale changes to the constraint graph, requiring that we recompute satisfiability anew each time. For this reason, we developed the efficient linear-time algorithm given here. In contrast, they considered the addition of only a single edge at each step, and incremental techniques were more appropriate.

3.2.3 Providing a labeling

The function is-satisfiable implicitly finds an axis and stride labeling, but its axis labeling may use more template axes than necessary. When the final maximal satisfiable subgraph is found, we label the axes by a coloring procedure as follows.

We construct another graph to describe the coloring problem: the axis quotient graph. This graph has one vertex for each *connected component* of the axis/stride satisfiability graph, and an undirected edge between vertices representing two connected components that occur in the same array object. A k-coloring of this graph corresponds to an assignment of the axes of each array object to k template axes. Each color corresponds to an axis of the template, and a k-coloring of the graph assigns the axes of each array object to different axes of the template. Finding an optimal coloring is hard:

Theorem 2 *Given an ADG that admits a communication-free labeling, axis assignment to minimize the number of template dimensions is NP-complete.*

Proof: It is easy to see that this problem is in NP since we can use the procedure is-satisfiable to verify the validity of a given axis alignment in polynomial time. We proceed by reduction from "Graph k-colorability (GT4)" [16], which is the following problem: Given a graph $G = (V, E)$ and a positive integer $k \leq |V|$, is G k-colorable, i.e., does there exist a function $f : V \to \{1, 2, \ldots, k\}$ such that $f(u) \neq f(v)$ whenever $(u, v) \in E$? This is solvable in polynomial time for $k = 2$, but remains NP-complete for all fixed $k \geq 3$.

Given an instance G of GT4, we reduce it to an instance of our problem as follows. For each vertex v_i of G, construct an input node in the ADG for a variable called t_i of dimension `(1:N)`. For each edge $e_m = (i, j)$, construct an input node in the ADG for a two-dimensional array called A_m with dimensions `(1:N, 1:N)`, and construct two ADG section assignment nodes that express the assignments $A_m(:, 1) = t_i$ and $A_m(1, :) = t_j$.

Let each color correspond to a template dimension, and let k be the dimensionality of the template. Each array t_i is assigned to dimension $f(v_i)$ in the template, and each array A_m spans the two dimensions $f(v_i)$ and $f(v_j)$ in the template. If G is k-colorable, then the mapping described above yields an assignment of template dimensions to array axes that does not violate the constraint that axes of a given object must lie in different template dimensions. On the other hand, if the ADG has a legal assignment of axes to k template dimensions and the mapping described above is constructed, then the assignment satisfies the constraints of the colorability problem. Thus, the ADG has an axis assignment with k template dimensions iff G is k-colorable. □

As with most graph coloring problems, we expect that the axis quotient graph will usually be easy to color and that standard heuristics [5] will find an optimal or near-optimal coloring.

3.3 Contracting the constraint graph

The constraint graph may be contracted into a smaller graph that captures all of the alignment constraints and costs of the original graph. We can then use the algorithm of Section 3.2, or any other method, to align the contracted graph, and propagate the results back to the original graph by reversing the contractions. For many examples, the contracted constraint graph has only a few vertices. Since performing the contractions is inexpensive compared to determining the alignment, contraction makes the total running time much smaller. The contractions rely on the following property of D-matrices, which were defined in Section 1.1.

Lemma 2 *Let Y and C be given D-matrices with the same number of columns. There is always at least one D-matrix X such that $XC = Y$.*

Proof: Let X be a $(p \times q)$ matrix, C be $(q \times r)$, and Y be $(p \times r)$. Without loss of generality, the rows and columns of the matrices may be permuted to place the nonzeros of C on the diagonal. Now, if $q = r$, then C is a diagonal matrix and X is uniquely determined. Otherwise, $q > r$ and the problem may be written as follows:

$$Y = [X_1 X_2] \begin{bmatrix} D \\ 0 \end{bmatrix}.$$

D is a nonsingular $(r \times r)$ matrix, X_1 is $(p \times r)$, and X_2 is $(p \times (q - r))$. Matrix X_1 is determined given Y and D. Any value for X_2 satisfies the equation, but it must be chosen so that X is a D-matrix, which is simple. The r columns of X_1 are multiples of r standard basis vectors $\{e_1, e_2, \cdots, e_p\}$. The $(q - r)$ columns of X_2 must be multiples of the remainder and there are $(p - r)$ such columns remaining. Since $(p \geq q)$, there is at least one way in which this can be achieved. □

We now present four situations where the CG can be contracted.

Contraction 1: Suppose vertex v has degree 1, so v is adjacent to only one other vertex w. The edge between them has a directed constraint C_{vw} or C_{wv}. In either case, contract the graph G into a smaller graph G' by removing v and the edge. To convert an alignment for G' into one for G with the same cost, choose a position P_v for v as follows. If the edge was directed (w, v), then compute $P_v = P_w C_{wv}$. If the edge was directed (v, w), solve $P_v C_{vw} = P_w$ for P_v by Lemma 2.

Contraction 2: The second contraction applies when a vertex v is adjacent to only two different vertices. In this case there is an edge (u, v) and an edge (v, w), and $u \neq v \neq w \neq u$. Construct G' by eliminating v and contracting the two edges into a new edge (u, w) with edge label $C_{uw} = C_{uv} C_{vw}$, and weight $W_{uw} = \min(W_{uv}, W_{vw})$. To convert an alignment for G' into one for G with the same cost, choose a position for v as follows.

There are two cases. If the alignment for G' satisfies edge (u, w), then compute $P_v = P_u C_{uv}$. This satisfies (u, v) in G, and (v, w) is also satisfied because $P_w = P_u C_{uw} = P_u C_{uv} C_{vw} = P_v C_{vw}$. If the alignment for G' does not satisfy (u, w), then (u, w) contributes cost $W_{uw} = \min(W_{uv}, W_{vw})$ to G'. We can construct an alignment for G with the same cost by failing to satisfy the less expensive of (u, v) and (u, w): If $W_{vw} < W_{uv}$ then let $P_v = P_u C_{uv}$, and if $W_{uv} \leq W_{vw}$ the solve $P_w = P_v C_{vw}$ for P_v by Lemma 2.

Contractions 3 and 4: There are two final contraction operations. Merge parallel edges if their constraint matrices are equal and add their edge weights. Finally, reverse edges with invertible constraints. (Note that all square D-matrices are invertible.) This may enable other contraction operations.

3.4 Experimental results

To illustrate our algorithm, we constructed the two small example programs shown in Figure 7, which have nontrivial axis alignment issues.

We generated alignments for the programs using our algorithms with various edge selection rules and ran the optimized programs on the CM-5 to measure the effect of alignment on their running times. Because the CMF compiler does not allow axis-changing alignments [29], we broke alignment into two parts. We performed axis alignment manually by changing the orientation of the arrays in the program and including explicit array transpose operations for unsatisfied CG edges. We specified stride alignment by adding ALIGN directives to the source code.

We did three kinds of experiments. First, we examined the effect of edge selection strategy on the quality of solutions found. Second, we examined the effect of axis and stride alignment on running time, and the correlation between the discrete metric of the optimization problem and the actual running time on the CM-5. Third, we examined the effect of graph contraction on the time required to find a solution.

3.4.1 Edge selection ordering

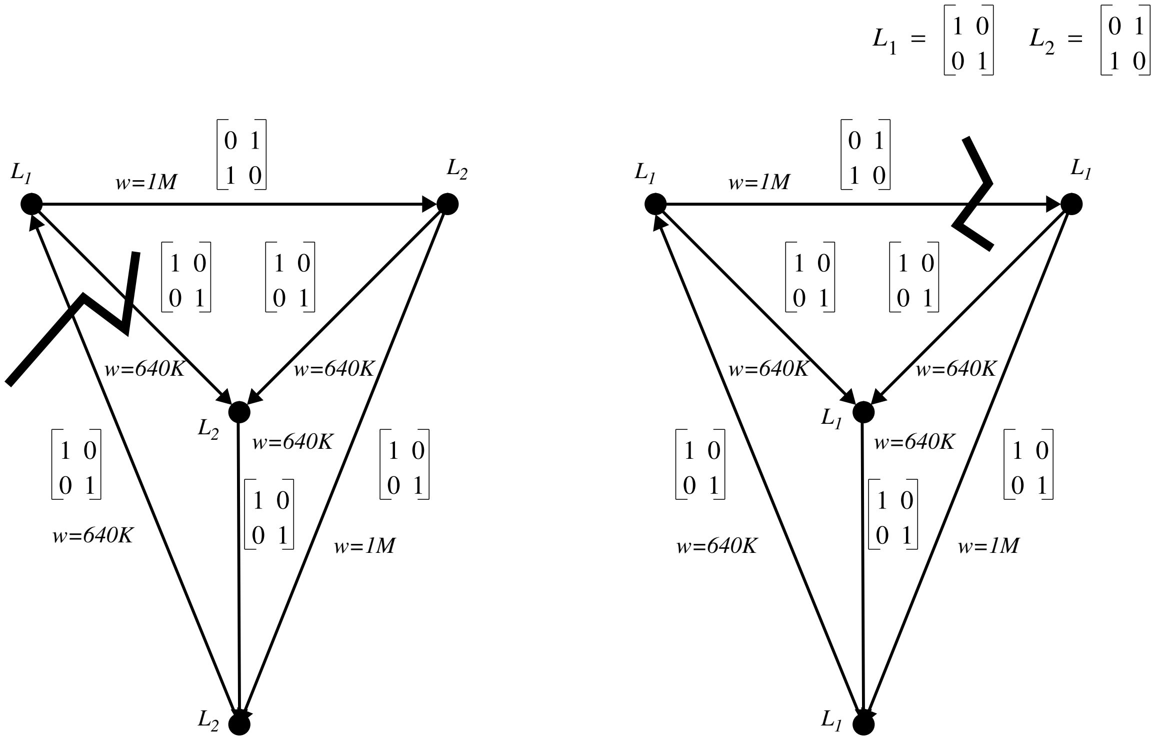
At each iteration the optimization algorithm tries to add an excluded edge to the graph. We examined three edge selection strategies: maximum weight first, minimum weight first and random selection. With the maximum weight ordering, our algorithm reduces to the greedy heuristic proposed by Knobe, Lukas and Steele. However, our experimental results show that other orderings (combined with the min-cut procedure) can yield superior results.

The contracted graph for PROGRAM1 appears in Figure 8 with two different alignments. One requires communication on two edges, for a total cost of 1,280,000. The optimal solution requires communication on only one edge and costs 1,000,000. This optimal solution was found using the minimum edge-weight heuristic and the min-cut procedure. This example demonstrates a shortcoming of the maximum-weight

```
PROGRAM program1
REAL, ARRAY(1000, 1000) :: A, B
C = A + B
B(1:800,1:800) = A(1:800,1:800) -
    transpose(B(1:800,1:800))
A(1:800,1:800) = transpose(A(1:800,1:800)) -
    B(1:800,1:800)
END PROGRAM
```

```
PROGRAM program2
PARAMETER(N=1000)
REAL, ARRAY(N, N) :: A, B
SUM  = A + transpose(B)
DIFF = transpose(A) - B
A2 = SUM(1:N/2, 1:N/2)
B2 = DIFF(1:N/2, 1:N/2)
HALFSUM  = A2 + B2
HALFDIFF = A2 - B2
A3 = HALFSUM(1:N/4, 1:N/4)
B3 = HALFDIFF(1:N/4, 1:N/4)
QUARTSUM  = A3 + transpose(B3)
QUARTDIFF = transpose(A3) - B3
A(1:N/4, 1:N/4) = transpose(QUARTSUM)
B(1:N/4, 1:N/4) = transpose(QUARTDIFF)
AVG = (A + B) / 2.0
END PROGRAM
```

Figure 7: Two example programs.



Maximum-weight ordering.

COST=1,280,000

Min-cut heuristic with minimum-weight ordering.
COST=1,000,000

Figure 8: The contracted graph for PROGRAM1. Alignments are shown by a "cut" through the unsatisfied edges. The optimal alignment requires cutting one of the heaviest edges. The maximum-weight heuristic cannot find this alignment.

heuristic: the optimal solution in this example cannot be found if edges are added in order of decreasing weight, because the subgraph of heaviest edges is satisfiable, but not optimal.

3.4.2 Execution time

We measured the execution time of each of the programs on the CM-5 with each alignment our algorithm generated, and also without axis or stride optimization as a baseline. The times measured were averaged over ten runs. Table 1 presents the data. The solution reported for the random edge selection heuristic reflects the best of five trials. The table shows the estimated cost according to the discrete metric and the actual execution time of the program.

We draw two conclusions. First, optimizing axis and stride alignment can significantly improve the running time of the programs. Second, our discrete metric of communication cost is an accurate enough measure to correctly predict the relative running times with different alignments.

Table 1: The estimated and actual times of two programs under differing axis and stride alignments.

Example	Method	Comm. Cost	CM5 running time (secs)
PROGRAM1	(none)		.25
	max-wt	1280000	.25
	min-wt	1000000	.13
	random	1280000	.25
	(optimal)	1000000	.13
PROGRAM2	(none)		.62
	max-wt	1750000	.56
	min-wt	1375000	.40
	random	1312500	.34
	(optimal)	1312500	.34

Table 2: The effect of contraction on the quality of solutions produced and the running time of the algorithm.

	FULL GRAPH		CONTRACTED	
Example/ Method	Comm. Cost	Time (s)	Comm. Cost	Time (s)
PROGRAM1	(44 nodes, 47 edges)		(4 nodes, 6 edges)	
max-wt	1280000	.95	1280000	.23
min-wt	1000000	1.50	1000000	.23
random-best	1280000	2.22	1280000	.19
random-worst	1920000	1.72	1640000	.25
PROGRAM2	(70 nodes, 77 edges)		(14 nodes, 21 edges)	
max-wt	1678500	2.47	1750000	.40
min-wt	2500000	6.12	1375000	.58
random-best	1312500	7.14	1312500	.63
random-worst	2375000	4.93	2625000	.65

3.4.3 Graph contraction

Graph contraction, which has not been suggested elsewhere for this problem, significantly reduces the size of the problem and the solution space that must be examined. Using contraction leads to a large decrease in the running time of the algorithm. In many cases the resulting graphs are small enough that their alignment problems could even be solved exactly by an exhaustive search.

Table 2 shows the running time of the optimization algorithm and the quality of solutions produced with and without contraction. The table reports running times for the entire optimization program, including contraction if any, on a Sun-4/370. Graph contraction is an inexpensive operation, and the time spent reducing the size of the graph is easily recovered by the time saved in the optimization phase.

Although contraction preserves the cost of the optimal alignment, it can change the result of our algorithm because the heuristic is sensitive to the order of selection of equal-weight edges. When optimizing PROGRAM2 with the maximum-weight ordering, a slightly worse solution is found with contraction enabled. In some cases, the contraction phase had the unfortunate effect of reordering the edges so that a worse solution is found.

We explored the effect that contraction has on the quality of solutions found, initially suspecting that

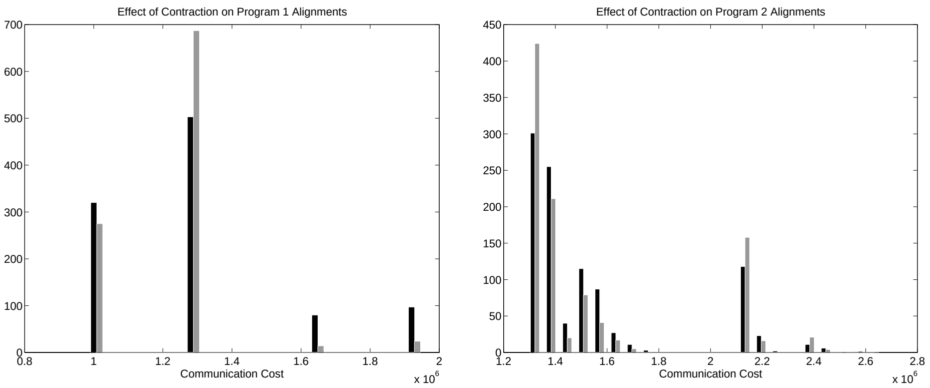


Figure 9: The effect of contraction on the quality of solutions found over 1000 runs of the algorithm using the random edge selection rule. Black bars show alignment costs found when using contraction; gray bars show costs found without contraction. Surprisingly, the distribution of results found is unchanged when contraction is used.

contraction leads to better solutions. However, this is not necessarily the case. Figure 9 shows a histogram of the frequency with which alignments of different costs were found by running the algorithm 1000 times with the random edge selection rule. Black bars are alignment costs found when using contraction, and gray bars are costs found without contraction. Using the random edge selection rule, contraction had little effect on the distribution of results.

In all cases, contraction significantly decreases the running time required by the algorithm. This makes it possible to run the optimization algorithm with a number of edge ordering heuristics and then choose the best solution found.

4 Offset Alignment

We now investigate algorithms for offset alignment. We first disregard the possibility of mobile offsets and show how to solve this static version of the problem using linear programming. We then extend this algorithm to handle mobile offset alignments.

Our model of the cost of offset communication is that it is linear both in the size of the object being shifted and in the magnitude of the shift (the latter being measured, for instance, in number of template cells). This model is application when the communication is bandwidth limited, which is applicable for many parallel machines for regular shift communications.

The use of this L_1 or Manhattan metric for communication cost also implies that that the shift cost can be independently computed for each template axis. This separability of the cost function may be more pessimistic than what some communication systems can deliver, but it is nonetheless a reasonable starting point and serves furthermore to keep the optimization problem manageable.

4.1 Static offset alignment by linear programming

We review how the static offset alignment problem can be reduced to linear programming [11]. Let the integer π_x be the offset alignment of port x. Then the residual communication cost (which is the function we want to minimize) is $C(\pi) = \sum_{(x,y)\in E} C_{xy}(\pi)$; so

$$C(\pi) = \sum_{(x,y)\in E} w_{xy}|\pi_x - \pi_y|.$$

Nodes introduce linear constraints relating the offsets of their ports [9]. To remove the absolute value from the objective function, we introduce a variable θ_{xy} for every edge (x, y) of the ADG, and add two inequality constraints,

$$\begin{aligned}\theta_{xy} + \pi_x - \pi_y &\geq 0\\ \theta_{xy} - \pi_x + \pi_y &\geq 0,\end{aligned}$$

that guarantee that $\theta_{xy} \geq |\pi_x - \pi_y|$. The new objective function is then

$$\sum_{(x,y)\in E} w_{xy}\theta_{xy}.$$

The transformed problem is equivalent to the original one, because $\theta_{xy} = |\pi_x - \pi_y|$ at optimality. This transformation introduces $|E|$ new variables and $2|E|$ new constraints.

We discuss several computational concerns with this approach in Section 4.6.

4.2 Mobile offsets

Consider an object with offset alignment αi^T, as defined in Section 2.3. Since the problem is separable, we can determine offsets with respect to one template axis at a time. If there are no loops in the code, the solution reduces to our earlier solution for static offset alignment [12].

The contribution of edge (x, y) to the residual communication is

$$C_{xy} = \sum_{i\in\mathcal{I}_{xy}} w_{xy}(i)|(\alpha - \alpha')i^T|, \tag{3}$$

where $\pi_x(i) = \alpha i^T$, $\pi_y(i) = \alpha' i^T$, and $\mathcal{I}_{xy}$ is the iteration space associated with the edge. Even if $w_{xy}(i)$ is constant, the absolute value in equation (3) makes its closed form complicated. Rather than seek an algorithm to minimize this cost function, we choose instead to approximate it by one for which the solution is straightforward. We first show the solution for fixed-size objects in singly-nested loops ($k = 1$), and then generalize to variable-size objects and to loop nests.

4.3 Fixed-size objects and singly-nested loops

Assume for this section that the data weight of edge (x, y) is constant and equal to 1, and that $\mathcal{I}_{xy} = \ell : h : s$. Call $(\alpha - \alpha')i^T$ the *span* of edge (x, y) at iteration i. If the span does not change sign in the interval $[\ell, h]$ (as shown in Figure 10(a)), the summation and the absolute value in equation (3) can be interchanged. Then $C_{xy} = |\sum_{i\in\ell:h:s}(\alpha - \alpha')i^T|$, the closed form for which is

$$C_{xy} = \frac{h - \ell + s}{s}|(\alpha_0 - \alpha_0') + \frac{\ell + h}{2}(\alpha_1 - \alpha_1')|. \tag{4}$$

Note that the term inside the absolute value is the average distance spanned by edge (x, y). We can reduce this to LP with one new variable per edge.

In general, however, the span may change sign in the iteration space, and interchanging the summation and the absolute value is incorrect, as shown in Figure 10(b). In this case, we partition the iteration space

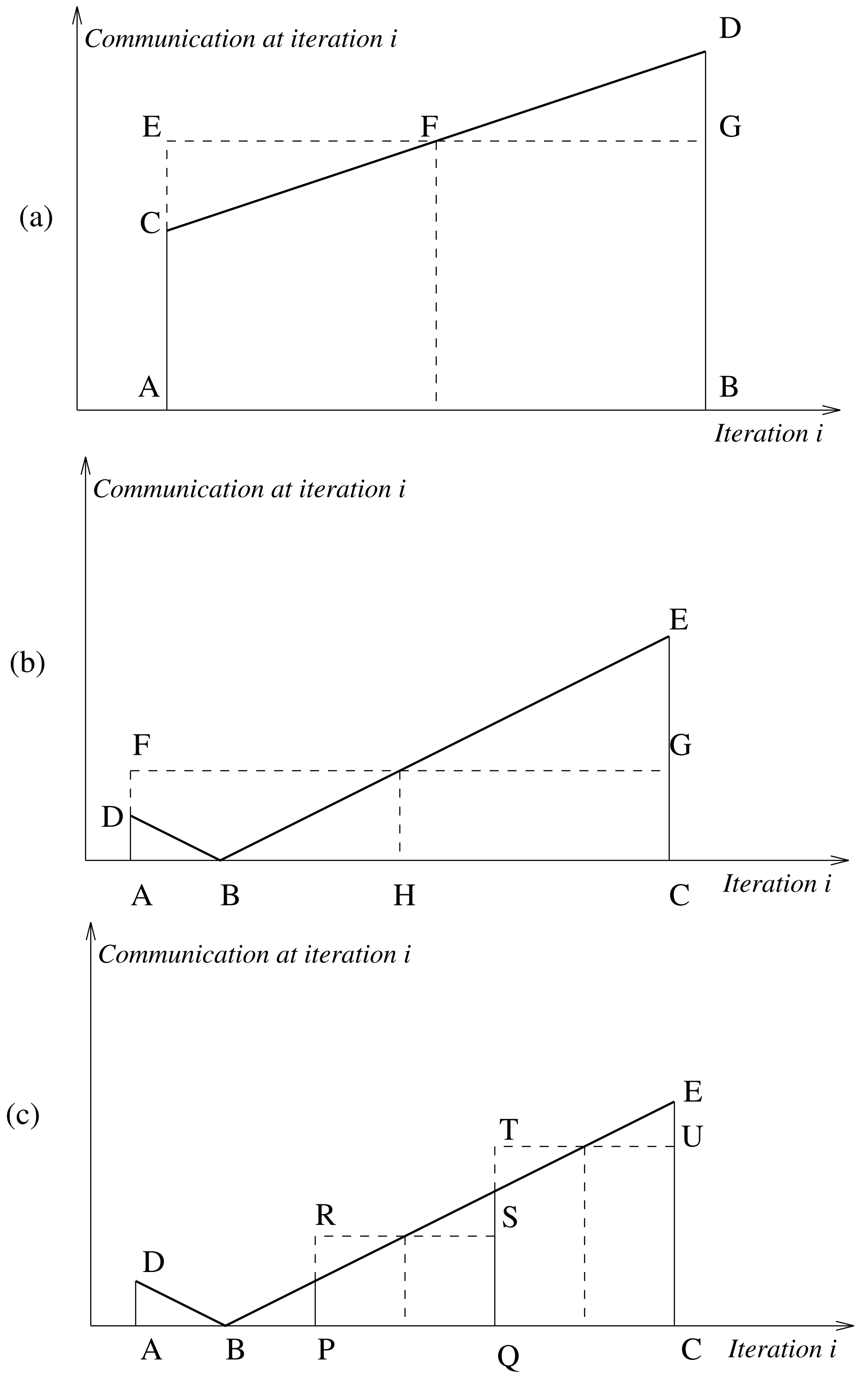


Figure 10: Approximating the cost of communication in loops. The actual communication cost is equal to the area under the heavy curve. (a) If the communication function does not have a zero crossing, then $ABDC \equiv ABGE$, and our approximation is exact. (b) If the communication function has a zero crossing, then $ABD + BCE \not\equiv ACGF$. The maximum relative error in approximation occurs when B coincides with H, and is proportional to AC. (c) To reduce the maximum relative error, we partition the iteration space AC into subranges AP, PQ, and QC. As there are no zero crossings in subranges PQ and QC, the approximations there are exact. The approximation in subrange AP is incorrect, but the maximum relative error is reduced. In general, at most one of the subranges can have a zero crossing.

into m equal subranges $\mathcal{I}_1, \ldots, \mathcal{I}_m$, each subrange corresponding to a set of consecutive iterations, and decompose the communication cost as follows:

$$C_{xy} = \sum_{j=1}^{m} \sum_{i \in \mathcal{I}_j} |(\alpha - \alpha')i^T|. \tag{5}$$

We then pretend that the span does not change sign within any subrange, which leads to the approximate cost model

$$C_{xy} \approx \widehat{C}_{xy} = \sum_{j=1}^{m} |\sum_{i \in \mathcal{I}_j} (\alpha - \alpha')i^T|. \tag{6}$$

Now we fix m, expand the outer sum explicitly, and evaluate each inner sum using equation (4), as shown in Figure 10(c). Clearly, the span can change sign in at most one subrange; therefore, at least $(m-1)$ of the subrange sums are correct. We then reduce to LP with m new variables per edge.

We now bound the error. We can show that the cost C at the approximate solution exceeds the cost at the best possible solution by at most a factor of $(1 + 2/m^2)$. (We can further reduce the error bound by using unequal intervals.)

The discussion above suggests several possible algorithms for solving the mobile offset alignment problem, which we now list.

1. **Unrolling:** Make every iteration a subrange, and use LP. This is equivalent to unrolling the loop. It is exact, but is impractical unless the number of iterations is small.

2. **State space search:** Approximate the iteration space as a single subrange, and use LP. Using this solution as an initial guess, optimize the exact cost equation (5) by, for example, steepest descent.

3. **Tracking zero crossings:** Split the iteration space into two equal subranges, and use LP. If the span has a zero crossing in the range, locate it, and move the subrange boundaries to coincide with this point. Now solve the new LP and iterate until convergence. This solves a sequence of fixed-size problems, each with $2|E|$ new variables. Convergence of this method is not guaranteed.

4. **Recursive refinement:** Approximate the iteration space as a single subrange, and use LP. Now examine the solution to determine subranges (at most one per edge) in which the span has a zero crossing. Break each subrange in two at the zero crossing, and formulate and solve a new LP. Continue the refinement until some stopping criterion is satisfied (*e.g.*, there are no more subranges to be refined, the objective function shows no further improvement, we run out of time). This requires solving a sequence of progressively larger problems.

5. **Fixed partitioning:** Partition the iteration space into three subranges, and use LP. The solution is guaranteed to be within 22% of optimal. This requires solving a single problem with $3|E|$ new variables. (A five-way partition would reduce the error bound to 8%.)

We advocate the fixed partitioning method as a good compromise between speed, reliability, and quality.

4.4 Variable-size objects in singly-nested loops

Now suppose that $\mathcal{I}_{xy} = \ell : h : s$ and that the data weight of edge (x, y) at iteration i is $\beta_0 + \beta_1 i$, where β_0 and β_1 are integer constants. Then the communication cost of the edge is

$$C_{xy} = \sum_{i \in \ell : h : s} (\beta_0 + \beta_1 i)|(\alpha - \alpha')i^T|.$$

Assuming the span does not change sign in $[\ell, h]$, we can write the communication cost of edge (x, y) as

$$C_{xy} = |(\beta_1\sigma_1 + \beta_0\sigma_0)(\alpha_0 - \alpha_0') + (\beta_1\sigma_2 + \beta_0\sigma_1)(\alpha_1 - \alpha_1')|,$$

where $\sigma_0 = \sum_{i\in\ell:h:s} 1$, $\sigma_1 = \sum_{i\in\ell:h:s} i$, and $\sigma_2 = \sum_{i\in\ell:h:s} i^2$ can be evaluated in closed form:

$$\begin{aligned}
\sigma_0 &= (h - \ell + s)/s. \\
\sigma_1 &= (s\sigma_0^2 + (2\ell - s)\sigma_0)/2. \\
\sigma_2 &= (2s^2\sigma_0^3 + (6s\ell - 3s^2)\sigma_0^2 + (6\ell^2 - 6s\ell + s^2)\sigma_0)/6.
\end{aligned}$$

We then determine the alignment coefficients as in Section 4.3.

4.5 Loop nests

The method generalizes to loop nests as follows. Divide the index range for each LIV into three subranges. The Cartesian product of this decomposition divides the iteration space into 3^k subranges, over each of which we assume that there is no sign change in the span; we sum the cost over each subrange, yielding one term in the approximate cost. We then solve for the minimizer of the approximate cost as in Section 4.3. It is also possible to use other quadrature rules to approximate the cost over each subrange.

For a k-deep loop nest, the problem has $3^k|E|$ variables. This technique will therefore not scale well for deep loop nests. We do not expect this to be a problem for Fortran 90, where array operations and `forall` loops are used to express in parallel what would be loop code in a sequential language.

The Cartesian product formulation handles imperfect and trapezoidal loop nests quite naturally. The key to this is the transformer nodes that bridge the different levels of the loop nest.

4.6 Computational issues

4.6.1 Reducing the problem size

In our original discussion of linear programming, we introduced linear constraints among all the ports of an ADG node. Since there is no communication within a node, we can easily reduce the number of constraints and variables by choosing one reference port to represent each node, and assigning the appropriate alignment to the remaining ports relative to the reference port. This modification was used in our implementation as it speeds up the runtime of the LP algorithm.

4.6.2 Dealing with non-integral offset values

Linear programming could produce non-integral values for some of the parameters, which would be incompatible with our model, which is discrete. There are several possibilities for this situation.

1. **(MILP)** We could use mixed integer linear programming, adding integrality constraints on the solution variables. This avoids the situation altogether, but results in an NP-complete optimization problem.

2. **(RLP)** We could round the LP solutions to integers. The rounded solutions are not necessarily optimal integer solutions; in general, rounding an LP solution may not even preserve feasibility. It can also magnify rounding errors. Consider the situation where two ports p_1 and p_2 should have equal mobile offsets. With RLP, rounding errors could produce completely wrong offsets, as in the case where the offset for p_1 is 0.49999999999999 i + 0 while that for p_2 is 0.50000000000001 i + 0.

Table 3: Running times of offset alignment for three test programs.

Program	Unknowns	Equations	Dimension	Iterations	Time (sec)
LU	28	5	0	4	0.04
			1	4	0.03
Twozone	1500	418	0	450	6.79
			1	262	3.72
Erle	3652	1044	0	975	35.06
			1	921	32.86
			2	1099	39.53

3. **(TLP)** We accept the solutions produced by LP, and truncate them at runtime. This is equivalent to RLP for the static case, but can be very different in the case of mobile offsets. Thus, given a mobile alignment $\alpha i+c$, RLP would round α and c at compile time to give the alignment **trunc**$(\alpha)i+$**trunc**(c), while TLP would evaluate the expression at runtime and truncate, giving the alignment trunc$(\alpha i + c)$. This could move an object with a non-uniform step at each trip through the loop, but the average velocity of the object would be α, which is exactly the solution computed by the LP. By forcing α to be an integer (either through rounding or using an IP solver), the velocity of the object will always remain constant, which is not necessarily the best solution.

   TLP is simpler than MILP, requires very simple runtime support, and is less sensitive to floating point problems than RLP. This is the method used in our implementation.

4.6.3 Runtime support for mobile offsets

Mobile offset alignments require runtime support to implement. The `REALIGN` directive available in HPF is precisely what is needed. This directive is executable, and would result in data motion at run time.

Another alternative method would be to insert explicit `SHIFT` operations into the generated code. This is not disjoint with the notion of inserting `REALIGN` directives—in fact, the latter may be implemented using the former technique!

One other possibility is to insert temporary arrays at the new positions and to use an array assignment to perform the data motion. This technique is likely to be more useful for redistribution, which requires all-to-all communication.

4.7 Performance results

Table 3 shows the running times of our offset alignment algorithm on a Sun-4/370 running version 5.5 of the MINOS solver [25], which solves linear programs using a reliable implementation of the primal simplex method. The "Iterations" column in the table refers to the number of steps the simplex method took to solve the LP. For these experiments, we used a file interface between our software and MINOS. The times reported are MINOS run times only, and do not include any I/O time, which was generally quite insignificant.

We have not investigated the issue of graph contractions for offset alignment. Such contractions, if they exist, could significantly reduce the time required to solve the LP.

```
real A(100), B(100,200)

do K = 1,200
   A = cos(A)
   B = B + spread(A, dim=2, ncopies=200)
enddo
```

Figure 11: Replication of the array A.

5 Replication

Until now we have considered alignment as a one-to-one mapping from an object to the template. We now relax our definition and make it a one-to-many mapping, introducing the notion of *replication*. We define replication as an offset alignment that is a set of positions rather than a single position. Replication can be viewed equivalently as a projection operation; this is the view we took in equation (1), where the replication actions were represented as the matrix R.

Suppose that a d-dimensional object is aligned to a t-dimensional template. We call the d template axes to which the object is aligned the *body axes* and the remaining $(t - d)$ axes *space axes*. Body axes require the specification of axis, stride, and offset alignments, while space axes require only offset alignments. Our notion of replication allows the offset alignment along a space axis of an object to be a regular section of the corresponding template axis. We use the symbol $*$ to indicate replication across an entire template axis. For example, the directive `ALIGN A(i) WITH T(i,10)` would indicate alignment of A with one position along the second template axis, while `ALIGN A(i) WITH T(i, *)` would indicate replication of A across all of the second template axis. A broadcast communication occurs on an edge along which data flows from a fixed offset to a replicated offset.

5.1 Replication labeling

Our goal in replication labeling is to decide which ports of the ADG should have replicated positions. In this section, we describe an algorithm for replication labeling that labels ports as being replicated or non-replicated.

There are three sources of replication:

- A `spread` operation causes replication.
- The use of lookup tables indexed by vector-valued subscripts is more efficient if the lookup table is replicated across the processors; we may replicate them with the programmer's permission.
- A read-only object with mobile offset alignment in a space axis can be realized through replication.

Subject to these sources, we want to determine which other objects should be replicated, in order to minimize broadcast communication during program execution. We model the problem as a graph labeling problem with two possible labels (replicated, non-replicated) and show that it can be solved efficiently as a min-cut problem.

Figure 11 shows why replication labeling is useful. In the example, a broadcast will occur in every iteration if A is not replicated, while a single broadcast will occur (at loop entry) if it is replicated. This is the solution found by our method.

After replication labeling, we discard from the ADG every edge with a replicated endpoint and proceed to find offsets for the non-replicated ports as described in Section 4. The justification for this is that an edge whose tail is replicated requires no communication, while an edge whose head is replicated requires the same amount of communication regardless of the offset of the (non-replicated) tail.

5.2 Labeling by network flow

As the cost function for replication communication is separable by dimension, we determine offsets independently for each template axis. We call the axis we are currently labeling the *current axis*. We must label every port of the ADG either "replicated" (**R**) or "non-replicated" (**N**). The constraints on this labeling are as follows:

1. A port for which the current axis is a body axis has label **N**.
2. The node for a `spread` along the current axis has its input port labeled **R** and its output port labeled **N**.[2]
3. A port for a read-only object with a mobile alignment in the current axis, and for which the current axis is a space axis, has label **R**.
4. Some other ports have specified labels, such as ports at subroutine boundaries, and ports representing replicated lookup tables.
5. At every other node, all ports must have the same label.

Subject to these constraints, we want to complete the labeling to minimize replication communication. We associate with each ADG edge a weight that is the expected total communication cost (over time) of having the tail non-replicated and the head replicated; the weight is therefore the sum over all iterations of the size of the object communicated.

The object is to complete the labeling, satisfying the constraints, and minimizing the sum of the weights of the edges directed from **N** to **R** ports. We now show that this is a min-cut problem and can be solved by standard network flow techniques.

Theorem 3 *An optimal replication labeling can be found by network flow.*

Proof: We define a weighted, directed graph G, which is a slightly modified version of the ADG. The vertices of G are as follows: Each node of the ADG except current-axis spreads is a vertex of G. If the node has a port labeled **N** or **R**, the vertex of G has the same label. (No node except a current-axis spread can have two ports with different labels.) Each current-axis spread corresponds to two vertices of G, one for each port, with the input-port vertex labeled **R** and the output-port vertex labeled **N**. Finally, G has a new source vertex s labeled **N** and a new sink vertex t labeled **R**. The edges of G are as follows: Each directed edge of the ADG corresponds to an edge of G with the same weight. Also, there is a directed edge of infinite weight from the source s to every vertex with label **N**, and a directed edge of infinite weight from every vertex with label **R** to the sink t.

A *cut* in G is a partition of its vertices into two sets X and $\overline{X}$, with $s \in X$ and $t \in \overline{X}$. The *cost* of a cut is the total weight of the edges that cross it in the forward direction, that is, the total weight of directed edges (x, y) with $x \in X$ and $y \in \overline{X}$.

[2] This sounds strange, but it correctly assigns any necessary communication to the input edge rather than to the node. Thus a `spread` node performs neither computation nor communication, but just converts a replicated object to a higher-dimensional non-replicated one.

Every replication labeling is a cut, and the cost of the labeling is the same as the cost of the cut. Every cut of finite cost is a replication labeling (since no infinite-cost edge can cross it in the forward direction), and hence a minimum-cost cut is an optimum replication labeling. The max flow/min cut theorem [26, Theorem 6.2] says that the cost of a minimum cut is the same as the value of a maximum flow from the source to the sink. □

Both the max flow and the min cut can be found in low-order polynomial time by any of several algorithms [26, 28, 21]. In particular, it can be solved using linear programming [26, chapter 6].

6 Remarks and Conclusions

This paper presents algorithms for determining alignment parameters for array data in a data-parallel language such as Fortran 90. We presented algorithms for the various components of the problem: axis and stride alignment, offset alignment (including mobile offsets), and replication labeling. Our algorithms extend those previously reported in a number of ways.

Our algorithms use a problem formulation based on the ADG representation. The ADG makes explicit all array objects generated by a program—named arrays as well as unnamed temporaries. Thus, the optimization algorithm has complete control over the placement of every array generated. The ADG also incorporates the effects of control flow into its data flow representation; this information can affect alignment decisions. Other work has not treated control flow as rigorously.

The graph contraction operations greatly reduce the computation time of the program. For many examples, the contracted constraint graph becomes a graph of only a few vertices, and the alignment problem can be solved exactly. Even when an exact method is not feasible, the reduced size of the contracted graph makes possible a more complete search of the space of possible solutions. We believe that even more powerful graph contractions are possible; indeed we hope eventually to define a set of contractions that reduces most programs enough that optimal alignments can be found by an exponential search procedure.

Axis and stride alignment is a discrete optimization problem. The optimization algorithm we propose is actually a family of optimization algorithms parameterized by an edge ordering. For one particular ordering, our algorithm reduces to the algorithm of Knobe, Lukas, and Steele. However, we have shown that other orders, coupled with the min-cut procedure, can lead to superior solutions. Edge orderings based on graph structure may be possible, and there may be more efficient means of finding conflicting edge sets.

We have proved that an optimal replication labeling can be found by network flow. For mobile offset alignment in the context of loops, we have presented an approximate reduction to linear programming, with error bounds on the solution quality.

While we have concentrated on loop programs, our framework can in fact deal with arbitrary control flow. Static single-assignment form can be constructed for programs with arbitrary control flow graphs. In the presence of arbitrary control flow, we can associate a control weight c_e of execution with every edge e of the ADG (using user input, profiling, or other heuristics), and minimize the *expected realignment cost*

$$\sum_{(x,y)\in E} \sum_{i\in\mathcal{I}_{xy}} c_{xy}(i) \cdot w_{xy}(i) \cdot d(\pi_x(i), \pi_y(i)).$$

A chicken-and-egg situation exists between replication labeling and determining mobile offset alignment, as replication can be motivated by a mobile alignment for a read-only object. Our current proposal is to iterate the replication labeling and mobile alignment phases until quiescence.

The only reason for restricting replication to space axes is that we do not yet completely understand the ramifications with regard to storage and communication of allowing replication in body axes. Extending the notion of replication to body axes would provide a more elegant theory.

We do not, however, foresee extending the definition of alignment to make it a many-to-one mapping (collapsing). This complicates the alignment phase, and we feel that it is best handled in the distribution phase by mapping some template axes to memory. Clearly, there are interactions between alignment and distribution, as decisions taken in the distribution phase (such as mapping certain template axes to memory) can radically alter the assumptions made in the alignment phase. We again propose handling such interactions by iterating the two phases until quiescence.

Acknowledgements

David Bau and Lenny Oliker implemented an earlier version of the axis, stride, and offset alignment algorithms, and our experiments here are built on their testbed. The proof of Theorem 1 is also due to David Bau. Patty Hough performed the communication cost studies on the CM-5. Michael Saunders provided access to the MINOS software. Jingke Li and Shang-Hua Teng participated in stimulating discussions of all these problems.

References

[1] American National Standards Institute. *Fortran 90: X3J3 internal document S8.118 Submitted as Text for ANSI X3.198-1991, and ISO/IEC JTC1/SC22/WG5 internal document N692 Submitted as Text for ISO/IEC 1539:1991*. American National Standards Institute, May 1991.

[2] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. In *Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, Dec. 1993.

[3] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 112–125, Albuquerque, NM, June 1993.

[4] D. Bau, I. Kodukula, V. Kotlyar, K. Pingali, and P. Stodghill. Solving alignment using elementary linear algebra. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallelism*, pages 4.1–4.15, Ithaca, NY, Aug. 1994.

[5] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[6] B. M. Chapman, P. Mehrotra, and H. P. Zima. Vienna Fortran—a Fortran language extension for distributed memory multiprocessors. Technical Report 91-72, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA, Sept. 1991.

[7] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S.-H. Teng. Generating local addresses and communication sets for data-parallel programs. *Journal of Parallel and Distributed Computing*, 26(1):72–84, Apr. 1995.

[8] S. Chatterjee, J. R. Gilbert, and R. Schreiber. Mobile and replicated alignment of arrays in data-parallel programs. In *Proceedings of Supercomputing'93*, pages 420–429, Portland, OR, Nov. 1993. Also available as RIACS Technical Report 93.08 and Xerox PARC Technical Report CSL-93-7.

[9] S. Chatterjee, J. R. Gilbert, and R. Schreiber. The alignment-distribution graph. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 234–252. Springer-Verlag, 1994.

[10] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler. Array distribution in data-parallel programs. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallelism*, pages 6.1–6.17, Ithaca, NY, Aug. 1994. Also available as RIACS Technical Report 94.09.

[11] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler. Modeling data-parallel programs with the alignment-distribution graph. *Journal of Programming Languages*, 2:227–258, Sept. 1994. Special issue on compiling and run-time issues for distributed address space machines.

[12] 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 SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 16–28, Charleston, SC, Jan. 1993. Also available as RIACS Technical Report 92.18 and Xerox PARC Technical Report CSL-92-13.

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

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[15] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. Fortran D language specification. Technical Report Rice COMP TR90-141, Department of Computer Science, Rice University, Houston, TX, Dec. 1990.

[16] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco, CA, 1979.

[17] J. R. Gilbert and R. Schreiber. Optimal expression evaluation for data parallel architectures. *Journal of Parallel and Distributed Computing*, 13(1):58–64, Sept. 1991.

[18] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, Sept. 1992. Available as technical reports UILU-ENG-92-2237 and CRHC-92-19.

[19] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1–2):1–170, 1993.

[20] C.-H. Huang and P. Sadayappan. Communication-free hyperplane partitioning of nested loops. *Journal of Parallel and Distributed Computing*, 19:90–102, 1993.

[21] D. Karger and C. Stein. On $\tilde{O}(n^2)$ algorithm for minimum cuts. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 757–765, 1993.

[22] 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*, pages 394–404, Vienna, Austria, July 1992. Austrian Center for Parallel Computation.

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

[24] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(2):213–221, Oct. 1991.

[25] B. A. Murtagh and M. A. Saunders. *MINOS 5.4 User's Guide*. Department of Operations Research, Stanford University, Mar. 1993. Report SOL 83-20R.

[26] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., 1982.

[27] T. J. Sheffler, R. Schreiber, W. Pugh, J. R. Gilbert, and S. Chatterjee. Efficient distribution analysis via graph contraction. In *Proceedings of the Eighth Annual Workshop on Languages and Compilers for Parallel Programming*, pages 25–1–25–15, Columbus, OH, Aug. 1995.

[28] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[29] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual Versions 1.0 and 1.1*, July 1991.

[30] S. Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1991. Available as Technical Report CMU-CS-91-121.

Biographies

Siddhartha Chatterjee has been an Assistant Professor of Computer Science at The University of North Carolina at Chapel Hill since October 1994. Prior to that he was a Postdoctoral Scientist at the Research Institute for Advanced Computer Science at NASA Ames Research Center in Moffett Field, California. He received his Ph.D. in computer science from Carnegie Mellon University in 1991. His research interests include the design and implementation of data-parallel languages for irregular problems, and compiler techniques for automatic data placement on distributed-memory parallel machines.

John R. Gilbert received his bachelor's degree in mathematics from the University of New Mexico in 1973 and his Ph.D. in computer science from Stanford in 1981. From 1981 to 1988 he waas Assistant and Associate Professor of Computer Science at Cornell. Since 1988 he has been a member of the Computer Science Laboratory at the Xerox Palo Alto Research Center. He has held visiting appointments at the University of Iceland and at the University of Bergen, Norway. His research is in parallel computing, geometric and combinatorial algorithms, and sparse matrix computation.

Leonid Oliker is a doctoral candidate at the University of Colorado at Boulder, and is currently a Visiting Researcher at the Research Institute for Advanced Computer Science at NASA Ames Research Center in Moffett Field, California. He received his bachelor's degrees in computer science and finance from University of Pennsylvania in 1991.

Robert Schreiber is a Project Scientist at Hewlett Packard Laboratories. He received a PhD in computer science from Yale University in 1977. He is known for important basic research in sequential and parallel algorithms for sparse matrix computation, is a contributor to the Matlab scientific computing environment, and is a codeveloper of the High Performance Fortran programming language. He has written over eighty scientific papers. He is on the editorial boards of three journals in the scientific and supercomputing fields. Schreiber has held faculty positions at Stanford University and Rensselaer Polytechnic Institute, was Chief Scientist at Saxpy Computer Corporation, and was a senior scientist with the Research Instititute for Advanced Computer Science at NASA Ames Research Center. He has broad experience as a consultant in the scientific and technical computing industry.

Thomas J. Sheffler is currently working on the formal verification of VLSI devices for Rambus Inc. Prior to that he was a Postdoctoral Scientist at the Research Institute for Advanced Computer Science at NASA Ames Research Center in Moffett Field, California. He received his Ph.D. in computer engineering from Carnegie Mellon University in 1992. His research interests include parallel algorithms, high-performance computer architectures, and object-oriented languages and systems.