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`. Copyright ©1993, 1994, 1995 by Xerox Corporation. All rights reserved.

[§]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

Aggregate data objects (such as arrays) are distributed across the processor memories when compiling a data-parallel language for a distributed-memory machine. 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: an *alignment* that maps all the objects to an abstract template, followed by 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, and give a heuristic method 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 contractions 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 or can be used to improve performance. We describe an algorithm based on network flow that replicates objects so as to minimize the total amount of broadcast communication in replication.

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 ell
$\in$	Set membership symbol
$\sum$	Summation symbol
∞	Infinity symbol

1 Introduction

Data-parallel array languages express parallelism as operations on arrays and array sections. Compiling such a program for a distributed-memory parallel machine requires a model for mapping 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 relative 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 [10], High Performance Fortran [14], CM-Fortran [21], and Vienna Fortran [4].

Placing arrays to enhance data locality is important when compiling array-parallel languages for distributed-memory parallel computers. The languages mentioned above 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 [2, 5, 6, 16, 17, 18, 22]. 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.

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 consider alignment only in this paper, we take the view that arrays 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 [7].

1.1 A formal model of data layout

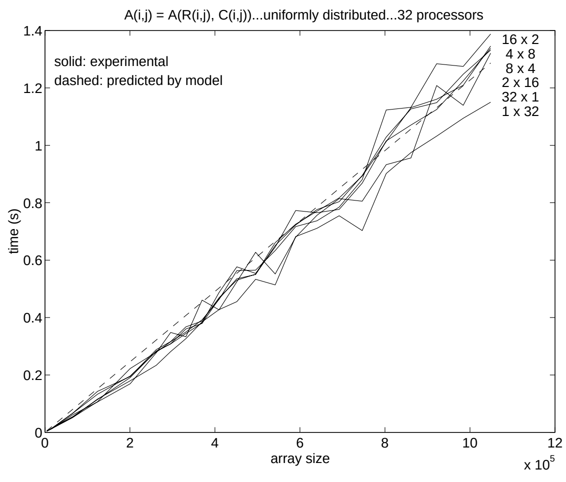
Alignment maps an array to a template with an affine mapping. The array coordinate $a \in \mathcal{Z}^d$ maps thus to the template coordinate $t \in \mathcal{Z}^t$ [1]:

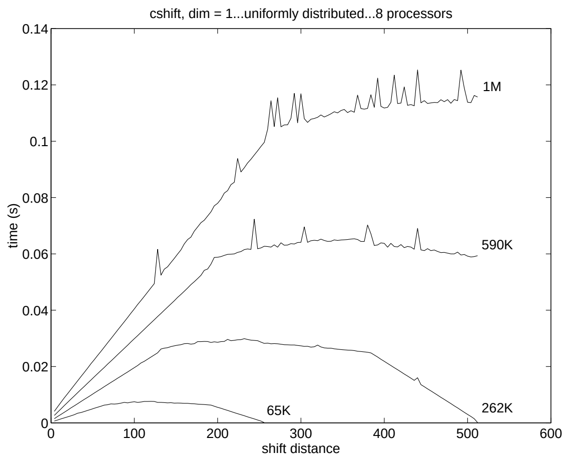
$$Rt = La + f. \tag{1}$$

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 such a matrix 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 [2] and Bau *et al.* [3] allow skewed alignment.

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





(a) (b)

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

to minimize the completion time of the program. 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 problem.

Finally, we determine the vector f in equation (1) that encodes the offset parameters. Our cost model of offset realignment assumes linear growth 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.

In addition to static offsets, we also allow the components of f to be affine functions of loop induction variables. We call such alignments *mobile*. We use linear programming to determine static offset alignments, and extend this algorithm to determine good mobile alignments.

1.3 Related work

Knobe, Lukas, and Steele [17] laid the foundation for data layout optimization. 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, inherited from our alignment-distribution graph representation of data-parallel programs [6]. 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 use local graph contractions 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 [18] addressed axis alignment alone, using a representation called a component affinity graph. 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 [15] introduced a linear algebraic formulation of the alignment problem. This is the basis of more recent papers by Anderson and Lam [2] and Bau *et al.* [3]. 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.

Several authors [2, 13, 17, 18, 22], including ourselves [7, 8, 12], have considered static offset alignment. We extend that work to handle mobile alignment here. Knobe, Lukas, and Steele [17] and Knobe, Lukas, and Dally [16] address the issue of dynamic alignment, which may depend on quantities whose values are known only at runtime. We focus 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) [6] to evaluate data layout decisions made during compilation. The ADG is a modified and annotated data flow graph based on static single assignment form [9], but incorporates a "position semantics" that makes each communication operation of the program explicit. 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 in the usual manner with merge, fanout, and branch nodes. 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.

We can now 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 [8].[1]

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

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 realignment cost of the program is

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

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

While mobile alignments could be arbitrary functions of the LIVs, we consider only the (important) case in which they are affine in 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 $i = (1, i_1, \ldots, i_k)$. We also 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

An example ADG for a Fortran 90 code fragment appears in Figure 2. 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 trans-

$d(q, r) \geq d(p, r)$.

Figure 2: 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.

form 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.

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*. Both position and constraint matrices are D-matrices (defined in Section 1.1). 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 semantic 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

Figure 3: 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.

constraints are expressed relative to it. The construction of constraint matrices for the various node types of the ADG is straightforward [6].

The CG is constructed from these constraints. For each ADG port x, 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 *communication-free* if it satisfies every edge constraint, and a CG is *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.

3.1 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 transpose and addition operations.*

Proof: The proof is by reduction from "Bipartite Subgraph (GT25)" [11].[2] □

3.1.1 Outline of the algorithm

Let G be a given CG. Like the algorithms of Knobe, Lukas, and Steele [17] and Li and Chen [18], our algorithm, shown in Figure 4, builds a maximal satisfiable subgraph G' 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. The algorithm terminates because the weight of the current graph increases at every iteration (though its size may not).

Lemma 1 *In step 3 of the algorithm in Figure 4, the graph $G(e, E)$ with e included and E removed is satisfiable.* □

3.1.2 The predicate is-satisfiable

Given a CG, G, is-satisfiable determines in linear time if there is a communication-free labeling of G. We first construct a new graph, where 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.

[2] Space limitations preclude the inclusion of full proofs in this article. The interested reader can find the complete proofs at the URL `http://www.cs.unc.edu/~sc/research/papers.html`.

1. Initialize G' to contain all of the vertices of G, but none of the edges. 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'.

2. Include an excluded edge e.

3. If the resulting graph is satisfiable (see Section 3.1.2), accept the new edge and go to Step 2.

   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.

4. If the weight of edge e is greater 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 unchanged.

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

6. Find a labeling that satisfies the final graph (see Section 3.1.3).

Figure 4: The algorithm for optimizing axis and stride alignments.

We then find the connected components of this new graph. If two vertices of an 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.

For each connected component of the graph, we determine 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. We do this as follows: find any spanning tree of the connected component; label an arbitrary vertex “1” and label the rest of the vertices by multiplying (or dividing) by the stride label of the spanning tree edges; for each non-tree edge, check whether the stride transformation it describes is satisfied by its endpoints.

3.1.3 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, with each color corresponding to a template axis. Finding an optimal coloring is hard, as shown below, but as with most graph coloring problems, we expect and that standard heuristics will find an optimal or near-optimal coloring.

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

Proof: The proof is by reduction from "Graph k-colorability (GT4)" [11]. $\square$

3.2 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 3.1, 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 $XC = Y$.* $\square$

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

Contraction 1: Let vertex v be adjacent to only one other vertex w, with the edge having a directed constraint C_{vw} or C_{wv}. Contract G into G' by removing v and the edge. To convert an alignment for G' into one for G with the same cost, choose P_v for v as follows. If the edge was (w, v), then compute $P_v = P_w C_{wv}$. If the edge was (v, w), solve $P_v C_{vw} = P_w$ for P_v by Lemma 2.

Contraction 2: If a vertex v is adjacent to only two different vertices, so that 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 $W_{uw} = \min(W_{uv}, W_{vw})$. Convert an alignment for G' into one for G with the same cost as follows.

If the alignment for G' satisfies edge (u, w), then compute $P_v = P_u C_{uv}$. If the alignment for G' does not satisfy (u, w), then (u, w) contributes cost $W_{uw} = \min(W_{uv}, W_{vw})$ to G'. 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 set $P_v = P_u C_{uv}$, otherwise 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. (All square D-matrices are invertible.) This may enable other contraction operations.

```
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 5: Two example programs.

3.3 Experimental results

To illustrate our algorithm, we constructed the two small example programs shown in Figure 5, 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 [21], 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.

3.3.1 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 can yield superior results. For instance, for PROGRAM1, the maximum-weight ordering requires communication on two edges, for a total cost of 1,280,000. The optimal solution, requiring communication on only one edge and costing 1,000,000, was found using the minimum edge-weight ordering heuristic.

3.3.2 Execution time

We show in Table 1 the execution time of each program on the CM-5 with each alignment our algorithm generated, and also without axis or stride optimization as a baseline. 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.

3.3.3 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

Table 1: The estimated and actual times of two programs under differing axis and stride alignments. The times measured were averaged over ten runs. 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.

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

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. Graph contraction is an inexpensive operation, and the time spent reducing the size of the graph is easily recovered by the time saved in optimization.

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 contraction leads to better solutions. However, this is not necessarily the case. Figure 6 shows a histogram of the frequency with which alignments of different costs were found by running the

Table 2: The effect of contraction on the quality of solutions produced and the running time of the algorithm. The table reports running times for the entire optimization program, including contraction if any, on a Sun-4/370.

Example/ Method	FULL GRAPH Comm. Cost	FULL GRAPH Time (s)	CONTRACTED Comm. Cost	CONTRACTED 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

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.

4 Offset Alignment

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 applicable when the communication is bandwidth limited, which holds 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

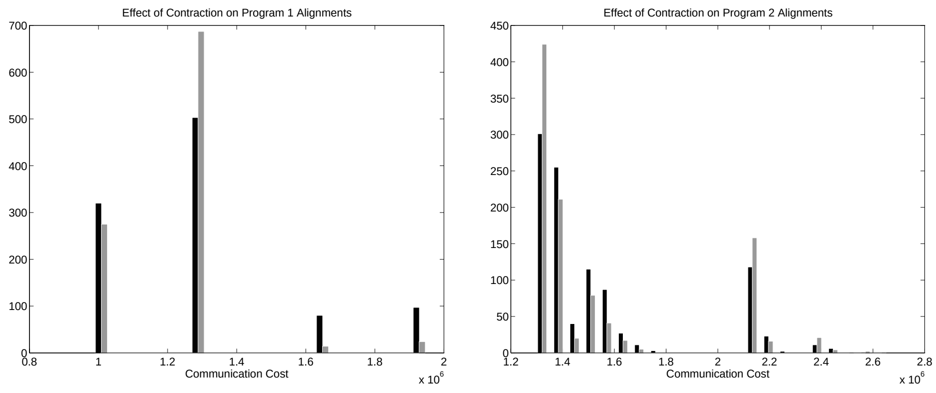


Figure 6: The effect of contraction on the quality of solutions found over 1000 runs of the algorithm using the random edge selection rule. Black bars show the histogram of alignment costs found when using contraction; gray bars show costs found without contraction.

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 [6]. 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) $C(\pi) = \sum_{(x,y)\in E} w_{xy}|\pi_x - \pi_y|$. Nodes introduce linear constraints relating the offsets of their ports [6]. 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, $\theta_{xy} + \pi_x - \pi_y \geq 0$ and $\theta_{xy} - \pi_x + \pi_y \geq 0$, 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.

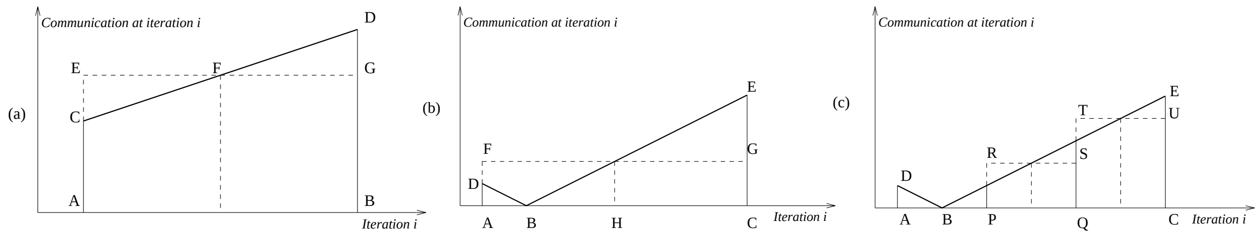

(a)
Communication at iteration i
D
E
F
G
C
A
B
Iteration i
(b)
Communication at iteration i
E
F
G
D
A
B
H
C
Iteration i
(c)
Communication at iteration i
E
T
U
R
S
D
A
B
P
Q
C
Iteration i

Figure 7: 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.

4.2 Mobile offsets

Consider an object with offset alignment αi^T, as defined in Section 2. 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 [7].

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.

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 7(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 7(b). In this case, we partition the iteration space 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 $C_{xy} = \sum_{j=1}^{m}\sum_{i\in\mathcal{I}_j}|(\alpha - \alpha')i^T|$. We then pretend that the span does not change sign within any subrange, which leads to the approximate cost model $C_{xy} \approx \widehat{C}_{xy} = \sum_{j=1}^{m}|\sum_{i\in\mathcal{I}_j}(\alpha - \alpha')i^T|$. Now we fix m, expand the outer sum explicitly, and evaluate each inner sum using equation (4), as shown in Figure 7(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. The cost C at the approximate solution exceeds the cost at the best possible solution by at most a factor of $(1 + 2/m^2)$.

The discussion above suggests several possible algorithms for solving the mobile offset alignment problem, such as loop unrolling, state space search, tracking zero crossings, recursive refinement, and fixed partitioning [5]. In our implementation, we 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 do this as a good compromise between speed, reliability, and quality.

Extensions of this approach to variable-sized objects and loop nests are straightforward [5].

4.3 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, but this guarantees neither optimality nor 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.

3. **(TLP)** We accept the solutions produced by LP, and truncate them at runtime. 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.

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.

Table 3: Running times of offset alignment for three test programs. The "Iterations" column refers to the number of steps the simplex method took to solve the LP. The times reported are MINOS run times only, and do not include any I/O time, which was generally insignificant.

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

4.4 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 [19], which solves linear programs using a reliable implementation of the primal simplex method. We used a file interface between our software and MINOS.

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

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. Like HPF, we use the symbol $*$ to indicate replication across an entire 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: first, a `spread` operation causes replication; second, the use of lookup tables indexed by vector-valued subscripts is more efficient if the lookup table is replicated across the processors, which we may do with the programmer's permission; finally, a read-only object with mobile offset alignment in a space axis can be replicated.

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 8 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.

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

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

Figure 8: Replication of the array *A*.

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**.[3]
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.

[3]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.

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 goal is to complete the labeling, satisfying the constraints, and minimizing the sum of the weights of the edges directed from **N** to **R** ports. This is a min-cut problem and can therefore be solved efficiently by standard network flow techniques.

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

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 enables 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.

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

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 propose handling such interactions by iterating the two phases until quiescence.

Acknowledgments

We thank David Bau, Patty Hough, Jingke Li, Michael Saunders, Shang-Hua Teng, the anonymous referees, and the guest editors of this special issue for their help.

References

[1] 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.

[2] 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.

[3] 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.

[4] B. M. Chapman, P. Mehrotra, and H. P. Zima. Vienna Fortran—a Fortran language extension for distributed memory multiprocessors. Technical Report 91-72, ICASE, NASA Langley Research Center, Hampton, VA, Sept. 1991.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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

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