Automatic Array Alignment in Data-Parallel Programs

Siddhartha Chatterjee * John R. Gilbert † Robert Schreiber* Shang-Hua Teng ‡

Abstract

Data-parallel languages like Fortran 90 express parallelism in the form of operations on data aggregates such as arrays. Misalignment of the operands of an array operation can reduce program performance on a distributed-memory parallel machine by requiring nonlocal data accesses. Determining array alignments that reduce communication is therefore a key issue in compiling such languages.

We present a framework for the automatic determination of array alignments in data-parallel languages such as Fortran 90. Our language model handles array sectioning, reductions, spreads, transpositions, and masked operations. We decompose alignment functions into three constituents: axis, stride, and offset. For each of these subproblems, we show how to solve the alignment problem for a basic block of code, possibly containing common subexpressions. Alignments are generated for all array objects in the code, both named program variables and intermediate results. The alignments obtained by our algorithms are more general than those provided by the "owner-computes" rule. Finally, we present some ideas for dealing with control flow, replication, and dynamic alignments that depend on loop induction variables.

1 Introduction

Consider the Fortran 90 [2] code fragment shown in Figure 1. Parallelism is expressed in this code in the form of operations on whole arrays and array sections. Compiling such a program for a distributed-memory parallel machine requires aligning the various array objects (named variables as well as compiler-generated temporaries for intermediate results) with each other, partitioning the data, and distributing the work among the processors. The goal of compilation is to produce an alignment and distribution that reduces completion time. One such alignment is shown in Figure 2, where we have also made the array temporaries explicit. In this paper, we present algorithms to automatically transform unannotated code to annotated code.

```
real a(50,100), b(100), c(100,200)
real d(100,50), e(50), f(50), g(50)

d = c(:,1:200:4)
e = sum(a + transpose(d), dim=2)
f = sum(spread(b, dim=2, ncopies=50) + d,
  dim=1)
g = b(1:100:2) + e + f
```

Figure 1: **A Fortran 90 program fragment involving arrays, regular sections of arrays, and array intrinsics.**

Completion time has two components: computation and communication. Communication can again be separated into *intrinsic* and *residual* communication. Intrinsic communication arises from computational operations such as reductions and vector-valued subscripts 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 aligned with respect to each other. Not all communication patterns are equally expensive in distributed-memory machines. Typically, there is a three-level hierarchy of patterns: nearest-neighbor (shifts), tree (reductions, parallel prefix, spreads), and unstructured (transpose, change of stride, vector-valued subscripts). Tree patterns are used for intrinsic communication and are not considered in this paper. The goal of alignment is to provide a correspondence between array objects that minimizes the residual communication in the program.

We view the compilation process as two phases: *alignment* and *distribution*. In the alignment phase, all array objects in the program are positioned with respect to each other so as to reduce residual communication. We achieve this by aligning all array objects within an auxiliary Cartesian grid called a *template*. In the distribution phase that follows, the template is distributed onto the parallel machine; the distributions of the arrays are determined by their

*Research Institute for Advanced Computer Science, Mail Stop T045-1, NASA Ames Research Center, Moffett Field, CA 94035-1000 (sc@riacs.edu, schreibr@riacs.edu). The work of these authors was supported by the NAS Systems Division via Cooperative Agreement NCC 2-387 between NASA and the Universities Space Research Organization (USRA).

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

‡Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA 02139 (steng@theory.lcs.mit.edu). This work was performed when author was a postdoctoral scientist at Xerox PARC.

To appear in the Proceedings of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Charleston, SC, January 1993.

```
real a(50,100), b(100), c(100,200)
real d(100,50), e(50), f(50), g(50)
real t1(50,100), t2(50,100), t3(100,50),
  t4(100,50), t5(50), t6(50)

template T(10000,10000)
align a(i,j), t1(i,j), t2(i,j) with
  T(4*i-3,4*j-3)
align b(i) with T(*,4*j-3)
align c(i,j) with T(4*j-3,i)
align d(i,j), t3(i,j), t4(i,j) with
  T(4*j-3,4*i-3)
align e(i), f(i), t5(i), t6(i), g(i) with
  T(4*i-3,*)

d = c(:,1:200:4)
t1 = transpose(d)
t2 = a + t1
e = sum(t2, dim=2)
t3 = spread(b, dim=2, ncopies=50)
t4 = t3 + d
f = sum(t4, dim=1)
t5 = b(1:100:2)
t6 = t5 + e
g = t6 + f
```

Figure 2: **The program of Figure 1 with temporaries made explicit and alignment annotations added.**

alignment with respect to the template and the distribution of the template. This two-phase approach separates the language issues form the machine issues, and has been adopted by Fortran D [9], High Performance Fortran [15], CM-Fortran [24], and Vienna Fortran [4]. Our work is distinguished by performing these phases automatically rather than requiring them to be done by the user. In this paper, we deal only with the alignment phase.

Alignment and distribution are the primary determinants of the performance of data-parallel languages on massively parallel machines. This is particularly critical for distributed-memory or NUMA machines, where the cost of accessing nonlocal memory is high. This is a difficult problem: in current compilers, data alignments and distributions are either supplied by the user, or canonical defaults are used; task mappings are derived from data mappings using the "owner-computes" rule [16]. Requiring the programmer to specify alignment and distribution is onerous to the programmer and is insufficient for several reasons:

- The best mapping for a problem changes with the interconnection structure of the machine, the size of the machine, and the size of the problem [25]. This means that the user must either write new mappings for each case, or be satisfied with poor performance.
- Even if the programmer maps named variables, the compiler must still map temporaries. Mapping according to the owner-computes rule can cause unnecessary communication [18].
- The best data and task mappings may be different in different portions of a program. Having to deal with this explicitly would complicate the programmer's job still further.
- The owner-computes rule provides no guidance about placement for array transformations such as reductions and spreads.

We believe that the compiler should deal with data alignment and distribution. Some early partial results in this direction are the following.

- Knobe, Lucas, and Steele [19] developed a heuristic for data distribution based on alignment preferences for arrays. Their work addresses automatic distribution, but is restricted by the owner-computes rule. We remove this restriction.
- Wholey [25] demonstrated that optimal mappings are often non-obvious even for a simple array-based programming model. He further showed that a compiler can derive good mappings without programmer intervention. His algorithm used a combination of clustering and hill-climbing techniques to achieve these results. He dealt only with axis alignment. We use different techniques to handle axis, stride, and offset alignment.
- Li and Chen [21] presented a heuristic solution for axis alignment by reducing it to weighted bipartite graph matching.
- Gupta [13] developed techniques along the lines of Li and Chen for axis and stride alignment of Fortran 77 programs. He uses a more detailed cost model for edge weights but is restricted by the owner-computes rule. He also does not deal with offset alignment.
- Gilbert and Schreiber [11] developed a fast optimal algorithm for aligning temporaries in expression trees (single program statements) involving array sections, by characterizing interconnection networks as metric spaces. Their algorithm applies to a class of so-called *robust* metrics, which includes multidimensional grids, hypercubes, and the discrete metric. They showed that the presence of common subexpressions renders the problem NP-complete, making heuristic solutions necessary for basic blocks. Chatterjee *et al.* [6] extended Gilbert and Schreiber's work in two different directions. First, they reformulated the alignment problem in terms of dynamic programming and developed a family of compact dynamic programming (CDP) algorithms to handle both robust and non-robust metrics such as rings and fat-trees [20]. Second, they relaxed the restriction that all arrays be conformable, modeled this by allowing different weights on the edges of the expression tree, and extended the CDP algorithms to edge-weighted expression trees. Both those papers concentrated on offset alignment. Our paper extends this previous work in the following ways: it relates changes in array size to the semantics of transformational array operators, it provides solutions for axis and stride alignments, it generalizes the solution to basic blocks represented as edge-weighted DAGs, and it handles alignments for both variables and temporaries.

This paper is the first systematic and unified treatment of the alignment problem. We develop a detailed and realistic model of communication cost, and formulate the alignment problem as a constrained optimization of the residual communication cost function. We identify three components of alignment, present efficient algorithms for finding optimal and near-optimal alignments of both temporaries and program variables, and show how to deal with reduction, spread, transposition, array sections, and masked operations. Finally, we present some ideas on how to deal with control flow, replication, and dynamic alignments that depend on loop induction variables.

The paper is organized as follows. Section 2 introduces the language model. Section 3 defines the alignment problem, and reviews our previous work [6]. Section 4 treats the axis and stride alignment problem, and Section 5 treats the offset alignment problem for grids. Section 6 extends the algorithms of the previous sections to handle situations where named variables must be aligned in addition to temporaries. Section 7 presents extensions to this work to handle control flow, dynamic alignments, and replication. Finally, Section 8 presents conclusions.

2 The language model

We consider an imperative language with array-based data parallelism, incorporating the array features of Fortran 90 and APL [17]. There are three types of entities in the language: array objects, section specifiers, and permutations. *Array objects* are multidimensional rectangular collections of data. A *section specifier* is a list of *section expressions*. Section expressions have three forms. The form $\ell:h:s$ represents the sequence of indices from ℓ to h in steps of s. The form ℓ is equivalent to $\ell:\ell:1$, and the form : is equivalent to *lower-bound* : *upper-bound* : *1*. A section specifier may be applied to an array to create a new array (this is called *array sectioning*). Assignment to a section of an array conceptually produces a new array that is identical to the original array everywhere except on the section.

Let A be an $e_1 \times \cdots \times e_r$ array (*i.e.*, an r-dimensional array with extents e_1 through e_r), let i and n be integers, let op be a binary associative operator, and let $\mathtt{P} = [p_1, \ldots, p_r]$ be a permutation vector. The following three unary operators can be applied to arrays to produce new arrays.

1. `REDUCE(op, A, DIM =` i`)` Produces an $e_1 \times \cdots \times e_{i-1} \times e_{i+1} \times \cdots \times e_r$ array.
2. `SPREAD(A, DIM =` i`, NCOPIES =` n`)` Produces an $e_1 \times \cdots \times e_{i-1} \times n \times e_i \times \cdots \times e_r$ array.
3. `[P]A` Produces an $e_{p_1} \times \cdots \times e_{p_r}$ array. This is a generalization of the Fortran 90 `TRANSPOSE` intrinsic function and can be done with the `RESHAPE` intrinsic of Fortran 90.

Binary elementwise operations such as array addition are permitted. In fact, any n-ary operation whose output size is determinable from its input sizes is permissible. We assume the usual control flow mechanisms: conditionals (`IF-ELSE-ENDIF`) and iteration (`DO` loops).

The `WHERE` construct of Fortran 90 can be readily incorporated in our model. The language standard [2] specifies a set of rules to convert a `WHERE-ELSEWHERE-ENDWHERE` block into a sequence of `WHERE` statements, and also specifies how the mask argument is propagated into the right-hand side expression of the `WHERE` statement. For each elemental operation, we define a corresponding *masked* version that takes an additional mask argument. The algorithms presented in the remainder of the paper assume unary or binary operations, but can be extended in the obvious way to handle ternary masked operations.

We use a directed acyclic graph (DAG) representation of the basic block of code we are aligning. The standard algorithm [1] is used to create the DAG representation of the basic block, special attention being paid to `WHERE` statements and assignment to array sections. A node in the graph represents an array operation, and the input and output ports of a node represent its inputs and outputs. Edges connect ports, representing data flow. (We do not use control flow edges for alignment.) Our algorithms provide an alignment at each port; any edge whose endpoints have different alignments causes *residual communication*.

3 Alignment

Alignment is a relation (written $\boxplus$) that specifies a 1-1 correspondence between elements of a pair of array objects. Rather than specifying this correspondence directly, we introduce an auxiliary frame of reference called a *template* and specify all correspondences with respect to it. A template is a Cartesian grid of sufficiently high dimension and (conceptually) infinite extent into which all array objects can be mapped. Let A be an m-dimensional array and T be the n-dimensional template. The general form of the alignment relation is

$$A(i_1, \ldots, i_m) \boxplus T(f_1(i_1, \ldots, i_m), \ldots, f_n(i_1, \ldots, i_m)).$$

We restrict our attention to alignments in which every function f_k is an affine function of a single distinct index variable, $f_k = s_k i_{a_k} + o_k$. The parameters a_k, s_k, and o_k correspond to the three components of the alignment function: *axis*, *stride*, and *offset*. This restriction does not permit replication or skew alignments (alignments of the form $A(I) \boxplus T(I, I)$). Misalignment in the axis or stride components causes unstructured communication, and misalignment in the offset component causes nearest-neighbor communication. We now give examples of the various kinds of alignment.

Example 1 (Axis alignment) Consider the statement

```
B = B + [2,1]C,
```

where B and C are two-dimensional arrays. Let T be a two-dimensional template. If $B(I, J) \boxplus T(I, J)$ and $C(I, J) \boxplus T(I, J)$, execution of the statement will require general communication to transpose C. If, however, $B(I, J) \boxplus T(I, J)$ and $C(I, J) \boxplus T(J, I)$, the operands of the array addition and assignment are aligned, and no communication is necessary.

Example 2 (Stride alignment) Consider the statement

```
A(1:N) = A(1:N) + B(2:2*N:2),
```

where A and B are one-dimensional arrays. Let T be a one-dimensional template. If $A(I) \boxplus T(I)$ and $B(I) \boxplus T(I)$, the operands of the addition will be misaligned, and general communication will result. The alignments $A(I) \boxplus T(2I)$ and $B(I) \boxplus T(I)$ avoid residual communication.

Example 3 (Offset alignment) Consider the statement

```
A(1:N-1) = A(1:N-1) + B(2:N).
```

Let T be a one-dimensional template. If we have $A(I) \boxplus T(I)$ and $B(I) \boxplus T(I)$, nearest-neighbor communication will result. However, the statement can be executed without communication if $A(I) \boxplus T(I)$ and $B(I) \boxplus T(I-1)$.

Example 4 Consider the statement

```
A(0:2*N:2,1) = A(0:2*N:2,1) + B(1,3:N+3).
```

Let T be a two-dimensional template. Residual communication can be avoided if $A(I,J) \boxplus T(I+6,J)$ and $B(I,J) \boxplus T(2J,I)$.

While we were able to completely eliminate residual communication in the examples above, this is usually not possible. The only case guaranteed to have a zero-communication solution is where the graph of the code block is a tree, and source positions are free. In general, our aim is to find an alignment that minimizes residual communication.

There are several variants of the alignment problem. Table 1 lists some of these variants. The code fragment we are aligning may be a tree (single statement) or a DAG (basic block). A variety of distance functions may be used to model communication costs. Alignment for the source nodes may be supplied as input, or we may be asked to find alignments for some or all of them as well as for internal nodes. The *free-source* variant occurs when we consider the alignment of a basic block in isolation. The *fixed-source* variant models the dependence between basic blocks, where inputs to a block can be constrained by the actions of preceding code blocks.

3.1 Distance functions and residual communication

Let $G = (N, E, W)$ be a weighted DAG representing a basic block of code. Here, N is the set of nodes representing computations, E is the set of data flow edges, and $W(x,y)$ is a nonnegative weight on edge (x,y) reflecting the number of data items being communicated along that edge. Since fanout corresponds to multiple uses of the same array object, the weight of an edge in fact depends only on its source node. We will therefore abbreviate $W(x,y)$ to w_x. The DAG for the code in Figure 1 in shown in Figure 3. (Control flow edges may also be present, but these will not enter into the discussion of alignment.) G may be connected or consist of several connected components. Let $\mathbb{P}$ be a set of abstract "positions" (called the *position space*) encoding possible alignments. For instance, in Example 1, the position space consists of two positions, corresponding to the two alignments $A(I,J) \boxplus T(I,J)$ and $A(I,J) \boxplus T(J,I)$ of an arbitrary array A. (Put another way, the positions correspond to the "row-wise" and "column-wise" orientations of two-dimensional arrays.) Let d be a distance function on $\mathbb{P}$. The "distance" $d(p,q)$ between two positions p and q in $\mathbb{P}$ is a nonnegative number corresponding to the cost per unit data of changing position from p to q. Given a DAG edge with weight w and endpoints at positions p and q, the contribution of the edge to the residual communication is therefore $w \cdot d(p,q)$.

The distance function must be a metric, *i.e.*, satisfy the following three conditions for all positions p, q, and r in $\mathbb{P}$:

- **(Symmetry)** $d(p,q) = d(q,p)$.
- **(Nonnegativity)** $d(p,q) \geq 0$; $d(p,q) = 0$ if and only if $p = q$.

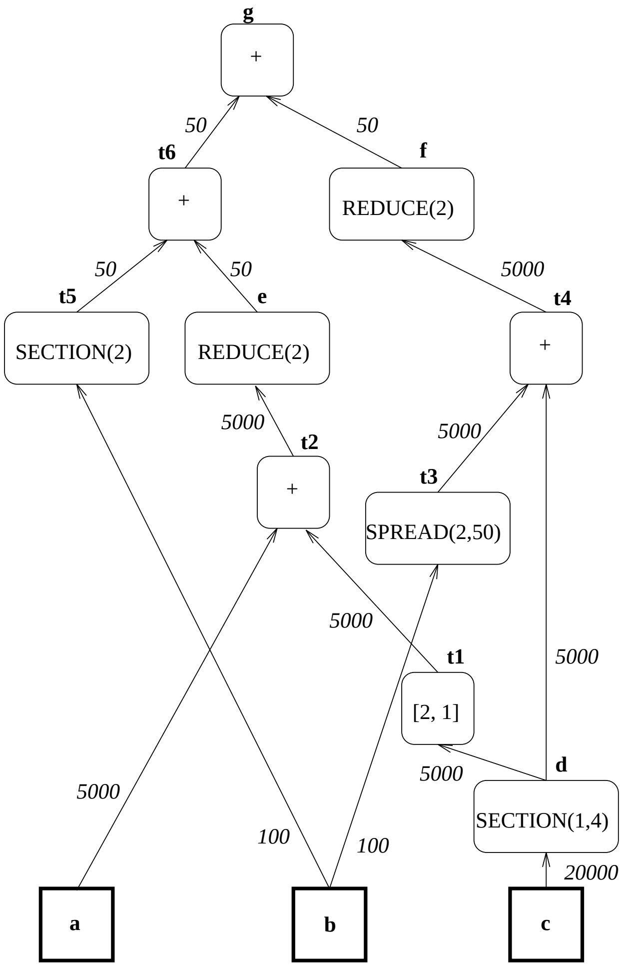

Figure 3: **Weighted computation DAG for program in Figure 1.**

- **(Triangle inequality)** $d(p,q) + d(q,r) \geq d(p,r)$.

Some examples of distance functions are given in Table 2. For multidimensional grids, hypercubes, and rings, the distance between two positions can be written as the sum of the distances in the multiple dimensions. This allows the alignment problem for such metrics to be separated into independent subproblems for each dimension [6].

Let $\pi : N \to \mathbb{P}$ be an alignment function specifying the locations of computation nodes in the position space. Then

$$C(G, d, \pi) = \sum_{(x,y) \in E} w_x \cdot d(\pi(x), \pi(y)) \qquad (1)$$

is the residual communication required for the given code block under the alignment π. The alignment problem is to determine a π that minimizes this residual communication. Note that the problem could be phrased equally well on the undirected graph, since the distance function d is symmetric. Although our algorithms will proceed in the direction of data flow, that is not inherent to the optimization problem.

Strictly speaking, this formulation of residual communication cost is correct only for trees. Consider the node shown in Figure 4. Let w be the weight of the edges leaving the node. Given positions for all the nodes as shown,

Graph structure	Source positions	Weights	Distance function	Solution method
tree	fixed	unweighted	grid, discrete	G-S, CDP
tree	fixed	unweighted	ring, fat-tree	CDP
tree	fixed	weighted	grid, ring, discrete, fat-tree	CDP
tree	free	either	any	trivial
DAG	fixed	weighted	discrete	CDP (heuristic)
DAG	free	weighted	discrete	CDP + CSP (heuristic)
DAG	free	weighted	grid	LP (heuristic)

Table 1: **Taxonomy of alignment problems. G-S = Gilbert-Schreiber algorithm for robust metrics, CDP = compact dynamic programming, CSP = cyclic source placement, LP = linear programming.**

Name	$d(p,q)$	Comments
Grid	$\|p-q\|$	Also for hypercube. Used for shift alignment.
Ring	$\min(\|p-q\|, P-\|p-q\|)$	Not robust. Used for shift alignment.
Discrete	$\begin{cases} 0 & p=q \\ 1 & p \neq q \end{cases}$	Models general communication. Used for axis and stride alignment.
Fat-tree	$2 \cdot \text{HEIGHT}(\text{LCA}(p,q))$	Not robust.

Table 2: **Examples of distance functions.** P **is the size of the ring.**

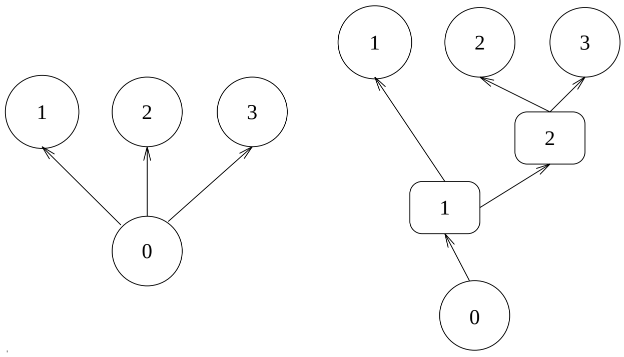

Figure 4: **The effect of the Steiner tree. The numbers within the nodes are their positions. Assume the grid metric. The residual communication cost of the graph fragment on the left is 6. However, if the Steiner tree on the right is used for the multicast, the residual communication cost is 3. For a one-dimensional grid, a Steiner tree is a minimum spanning tree of the positions, but that is not so in more than one dimension.**

the above formulation would estimate the residual communication cost as the sum of the individual distances, *i.e.*, $w(d(0,1) + d(0,2) + d(0,3))$. A better solution is to find the minimum spanning tree of the positions 0, 1, 2, 3 in the underlying metric space and use this tree to distribute the values. The optimal solution uses a *Steiner tree* [23], which is a tree of lowest possible total edge length connecting the fanout vertex to its successors. Let $\mathcal{S}(x,d)$ is the cost of the Steiner tree on the positions of node x and its successors in the DAG. If x has one successor y, then $\mathcal{S}(x,d) = d(\pi(x), \pi(y))$. Then we rewrite the residual cost expression as follows:

$$C(G,d,\pi) = \sum_{x \in N} w_x \cdot \mathcal{S}(x,d). \qquad (2)$$

Finding Steiner trees is NP-complete for most metrics [10] (the discrete metric is an important exception). In practice, the "Steiner effect" is usually minor, heuristics such as the use of a minimum spanning tree perform quite well, and bounds on the performance of the heuristics are available for many metrics of interest [26].

3.2 Edge weights

Edge weights model different amounts of communication along different edges. Operations such as sectioning, reduction, and spread change the sizes of array objects and motivate the use of weights. Consider the rank-one update $A = A + \alpha x y^T$, where A is a square matrix, x and y are column vectors, and α is a scalar. The multiplication of x by y^T takes two m-element vectors as input, and produces an m^2-element matrix as output. Thus the edges from x and y^T to the outer product node would reflect the cost of moving m elements each, and the edge from the multiplication to the addition would reflect the cost of moving m^2 elements. An alignment might then move the $2m$ elements of x and y^T to align with the columns and rows of A, spread them, and perform both the multiplication and the addition in place, in preference to multiplying x by y^T and then aligning the m^2 elements of the product with A.

We assume that the sizes of all named variables are known and that the size of the output of an operation depends only on the sizes of the inputs, not their values. With this restriction, we can easily propagate sizes of array objects using Chatterjee's size inference algorithm [5] to obtain the sizes of all array objects in the program. We can then label each edge with the size of the array object it represents. These edge weights are shown in Figure 3 for the program of Figure 1, where the unit of edge weight is 50 array elements.

3.3 Compact dynamic programming for trees

We now review the dynamic programming formulation of the alignment problem for weighted trees with fixed source node (*i.e.*, leaf) alignments [6]. Consider an expression tree T and a position space $\mathbb{P}$. For each internal node x, define the *subtree cost function* C_x such that $C_x(p)$ is the minimum cost of evaluating the subtree T_x rooted at node x, subject to the constraint that the root x of the subtree be placed at position p. Thus,

$$C_x(p) = \min_{\pi}\{C(T_x, d, \pi) : \pi(x) = p\}.$$

We can express the subtree cost function at a node x in terms of the subtree cost functions at its children y and z. It is just a minimum over all possible placements of y and z.

$$C_x(p) = \min_{r}\Big(C_y(r) + w_y \cdot d(p,r)\Big) + \min_{r}\Big(C_z(r) + w_z \cdot d(p,r)\Big).$$

The key observation is that $C_x(p)$ is the sum of two independent terms, one for each subtree below x. To emphasize this, we rewrite the recurrence in terms of the *contribution function* g_y, where $g_y(p)$ is the cost of computing the value of the subtree T_y and delivering it to position p. Function g_y is just the $(\min, +)$ convolution of C_y with the distance function. This gives us the following master recurrence.

$$\begin{aligned} C_x(p) &= g_y(p) + g_z(p) & (3)\\ g_y(p) &= \min_{r\in\mathbb{P}}(C_y(r) + w_y \cdot d(p,r)) & (4)\\ g_z(p) &= \min_{r\in\mathbb{P}}(C_z(r) + w_z \cdot d(p,r)). \end{aligned}$$

Finally, we define a position for y that realizes the minimum contribution $g_y(p)$:

$$r_y^*(p) = \text{some } r \text{ that minimizes } C_y(r) + w_y \cdot d(p,r). \qquad (5)$$

These equations lead to the following dynamic programming algorithm for solving the alignment problem.

Algorithm 1 (Optimal alignment of an expression tree using dynamic programming.)
Input: An edge-weighted rooted expression tree T, alignments for all source nodes of the tree, and a position space $\mathbb{P}$.
Output: An alignment function π that minimizes residual communication.
Method:

```
for x ∈ NODES(T) in postorder do
   for all p ∈ ℙ do
      evaluate C_x(p) and r*_y(p) using equations
         (3) and (5), where x = PARENT(y)
   enddo
enddo
π(ROOT(T)) ← some p at which C_ROOT(T)(p) is
   minimized
for y ∈ NODES(T) in preorder do
   π(y) ← r*_y(π(PARENT(y)))
enddo
return π
```

Metric	Unweighted tree	Weighted tree
Discrete	$O(Nh)$	$O(Nh)$
Grid	$O(kN)$	$O(kN\log^2 N)$
Ring	$O(kNh)$	$O(kN^3)$
Fat-tree	$O(N^2h)$	$O(N^3)$

Table 3: **Summary of asymptotic running times of CDP algorithms for various distance functions. N is the number of nodes in the expression tree, and h is its height. For the grid and ring metrics, k is the dimensionality of the grid or ring.**

Algorithm 1 exploits the structure of the tree but not of the distance function. It works for arbitrary distance functions and has time complexity $O(NP^2)$ and space complexity $O(NP)$, where $N = |\text{NODES}(T)|$ and $P = |\mathbb{P}|$. Mace [22] originally used this algorithm to choose the best memory layout for arrays, from a small set of possibilities. In our application we typically have $N \ll P$; hence Algorithm 1 is not practical. However, we can exploit the structure of some specific distance functions to represent the cost functions compactly, and substantially reduce the cost of dynamic programming. We call this family of algorithms *compact dynamic programming* (CDP). These various CDP algorithms are described in detail by Chatterjee *et al.* [6], and the asymptotic running times for the various metrics are presented in Table 3.

4 Axis and stride alignment with fixed source positions

We first solve the following problem: Given a weighted directed acyclic graph $G = (N, E, W)$ representing a basic block, and the sizes and positions of the source nodes of G (the named variables), find an axis and stride alignment π for all nodes in N that minimizes the residual communication cost of the DAG.

Changing axis or stride requires general communication using the interconnection network. It is generally difficult to predict the cost of such communication, since it depends on factors such as contention and average transit distance. However, a reasonable model of this kind of communication is to consider it as having some fixed, high cost. Therefore, we use the discrete metric to model general communication throughout this section.

Having decided on the discrete metric as the distance function, we must now specify the position space. Let t be the highest rank of any array object. This is also the rank of the template. We specify the position of an r-dimensional array object A by an ordered pair $\pi(A) = \mathbb{A}(A)|\mathbb{S}(A)$, with $\mathbb{A}(A)$ being the axis specifier and $\mathbb{S}(A)$ the stride specifier. Each specifier is an r-tuple with an entry for each dimension of the array object. For instance, suppose that $t = 3$, and the template axes are labeled i, j, and k. Then a 2-dimensional array could have six possible axis orientations: (i,j), (j,i), (i,k), (k,i), (j,k), (k,j). In general, the array A has $t!/(t-r)!$ possible axis orientations. Scalars are considered free, with the universe as the possible set of positions. Note that this does not give all the information we need about placing the object. If $A(:,:)$ is oriented $(i,j)|(1,1)$, then every row object $A(\ell,:)$ is oriented $(j)|(1)$ irrespective of the row number ℓ. The additional information we need is precisely the offset alignment.

There are only finitely many possible axis alignments, since the template has finite rank. However, there is a potentially infinite space of stride alignments. We use $\mathbb{Q}$, the set of rationals, as the space of strides. We choose $\mathbb{Q}$ rather than $\mathbb{Z}$ because $\mathbb{Q}$ is closed under multiplication and division by nonzero integers, which is necessary for dealing with strided sections. Our algorithm can generate fractional strides, but it multiplies through at the end to make them all integers.

Consider the array alignment $A(I,J) \boxplus T(3J-2, 10-6I)$. The axis|stride position corresponding to this alignment is $(j,i)|(3,-6)$. The negative stride indicates that the second array dimension is reversed with respect to the template axis.

4.1 Unary operations

Given a set of positions for an array object, we can derive sets of positions for the array objects created by unary array operations such as projection, strided section, generalized transpose, reduction, and spread. Let the template T have rank t. Let A be an array object of rank $n \leq t$, and B an array object of rank $m < t$. Let the template axes be labeled $\Gamma = [\gamma_1, \ldots, \gamma_t]$. Let i range over integers and p over rationals. Let $R_i = [\rho_1, \ldots, \rho_i]$ be a permutation of $[1, \ldots, i]$. Let $S_i = [\sigma_1, \ldots, \sigma_i]$ be a section expression, where each σ_i is a section specifier. We define the function APPLY shown in Figure 5 that applies a section or a permutation to a position for a given object to produce the corresponding position(s) of the derived object. This leads to the set of rules for positions of derived array objects shown in Figure 5.

Note that the output of a SPREAD operation can have a set of possible positions rather than a single position if there are multiple degrees of freedom between the ranks of the template and the input of the SPREAD. Consider `SPREAD(V, DIM=2, NCOPIES=n)`, where the vector `V` is aligned (i)|(1) in a 3-dimensional template. The possible orientations of the result are $(i,j)|(1,\mathbb{Q})$ and $(i,k)|(1,\mathbb{Q})$. The notation $(i,j)|(1,\mathbb{Q})$ is really an abbreviation of the set of positions $\{(i,j)|(1,p) : p \in \mathbb{Q}\}$. The stride in the spread direction is set to $\mathbb{Q}$ since there is no *a priori* reason to constrain it further.

4.2 The alignment algorithm

We will use a CDP algorithm as a heuristic to determine approximate costs at nodes. Note that this does not guarantee an optimal solution any longer, since the dynamic programming formulation does not accurately model the effect of fanout. First, the costs of subtrees are no longer independent: equation (3) no longer holds. However, we will continue take equations (3)–(5) as definitions of C_x, g_y, and r_y^* in the DAG case, realizing that $C_x(p)$ is only an approximation to the minimum cost of the subgraph below node x. Second, even supposing that we had exact cost functions for all nodes, the presence of multiple sink vertices poses another problem. An optimal solution for the DAG does not necessarily place each sink node in a minimum-cost position. Consider the following piece of code and its program graph shown in Figure 6.

```
real A(100,100), B(100,100), C(100,100),
  T(100,100), X(100)
T = A + transpose(A)
B = spread(X, dim=1, ncopies=100) + T
C = spread(x, dim=2, ncopies=100) + T
```

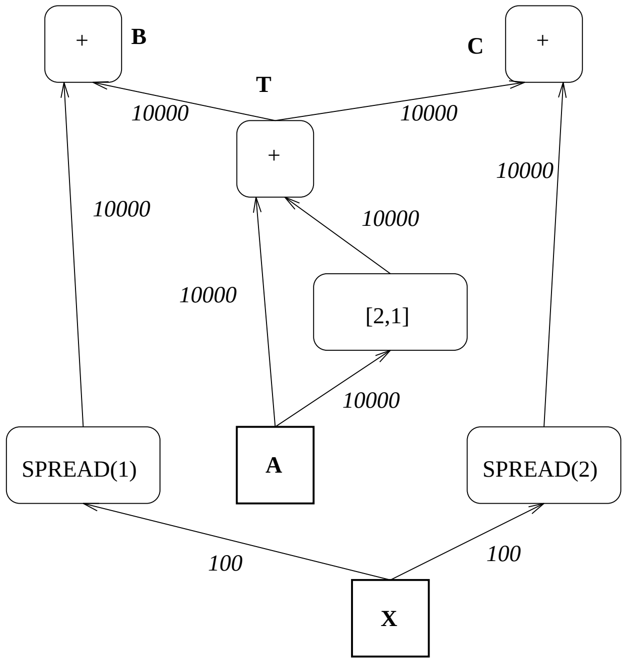

Figure 6: **A DAG with multiple sink nodes. The optimal solution for the DAG does not necessarily place the sink nodes at their minimum-cost positions.**

The minimum-cost solution would orient both arrays `B` and `C` either in orientation (i,j) or in orientation (j,i), with a residual cost of 100. However, this does not place the two sink nodes in their minimum-cost positions. Insisting that the sink nodes be placed "optimally" would actually result in a residual cost of 10000.

After we have an approximate solution, we will use the Steiner tree optimization to further reduce communication costs. Steiner trees can be found trivially for the discrete metric: determine the set of distinct positions of the destination nodes, and find any spanning tree of this set in the discrete metric space.

The contribution function of a node has a particularly simple form in the discrete metric.

Lemma 1 *Let $\mu_y = \min_r C_y(r)$. Then, in the discrete metric space,*

$$g_y(p) = \min(C_y(p), \mu_y + w_y).$$

Thus, a node is placed either at its parent's location or at a location where its cost function is minimized. Lemma 1 also shows that the convolution defining g_y (equation (4)) can be computed in constant time for the discrete metric.

The cost function C_x for a node x takes on only a small number of different values. Therefore, we represent it compactly by tabulating its *level sets.* A level set of node x is a set L_x of positions p where the cost value $C_x(p)$ is constant. (We do not require that level sets be maximal.)

Lemma 2 *Let x be an internal node with children y and z. Let L_y be a level set of y, with $C_y(p) = \beta$, and L_z a level set of z, with $C_z(p) = \gamma$. Then $L_y \cap L_z$ is a level set of x, with $C_x(p) = \min(\beta, \mu_y + w_y) + \min(\gamma, \mu_z + w_z)$.*

Proof: Let p be a position in $L_y \cap L_z$. By two applications of Lemma 1, $C_x(p) = \min(C_y(p), \mu_y + w_y) + \min(C_z(p), \mu_z +$

(Transpose)

$$
\begin{aligned}
\text{APPLY}(R_n, \mathbb{A}|\mathbb{S}) &= \text{APPLY1}(R_n, \mathbb{A})|\text{APPLY1}(R_n, \mathbb{S}) \\
\text{APPLY1}((\rho_1, \ldots, \rho_n), (\alpha_1, \ldots, \alpha_n)) &= (\alpha_{\rho_1}, \ldots, \alpha_{\rho_n})
\end{aligned}
$$

(Array sectioning)

$$
\begin{aligned}
\text{APPLY}(S, \mathbb{A}|\mathbb{S}) &= \text{APPLY2}(S, \mathbb{A})|\text{APPLY3}(S, \mathbb{S}) \\
\text{APPLY2}((\sigma_1, \ldots, \sigma_n), (\alpha_1, \ldots, \alpha_n)) &= \text{REMOVE-ZEROS}(\beta_1, \ldots, \beta_n) \\
\beta_i &= \begin{cases} \alpha_i & \text{where } \sigma_i = l_i : h_i : s_i \text{ or } : \\ 0 & \text{otherwise} \end{cases} \\
\text{APPLY3}((\sigma_1, \ldots, \sigma_n), (p_1, \ldots, p_n)) &= \text{REMOVE-ZEROS}(q_1, \ldots, q_n) \\
q_i &= \begin{cases} p_i \odot s_i & \text{where } \sigma_i = l_i : h_i : s_i \text{ or } : \\ 0 & \text{otherwise} \end{cases} \\
p \odot s &= \begin{cases} \mathbb{Q} & \text{where } p = \mathbb{Q} \\ p \cdot s & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\text{Let } \pi(A) &= (\alpha_1, \ldots, \alpha_n)|(p_1, \ldots, p_n) \\
\text{and } \pi(B) &= (\alpha_1, \ldots, \alpha_m)|(p_1, \ldots, p_m)
\end{aligned}
$$

$$
\begin{aligned}
\pi(A(S)) &= \text{APPLY}(S, \pi(A)) & (6) \\
\pi([R_n]A) &= \text{APPLY}(R_n, \pi(A)) & (7) \\
\pi(\text{REDUCE}(A, i)) &= (\alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_n)|(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n) & (8) \\
\pi(\text{SPREAD}(B, i)) &= \{(\alpha_1, \ldots, \alpha_{i-1}, \gamma, \alpha_i, \ldots, \alpha_m)|(p_1, \ldots, p_{i-1}, \mathbb{Q}, p_i, \ldots, p_m) : \gamma \in \Gamma - \mathbb{A}(B)\} & (9)
\end{aligned}
$$

Figure 5: **Definition of alignments for array objects created by unary operations.**

$w_z) = \min(\beta, \mu_y + w_y) + \min(\gamma, \mu_z + w_z)$. Thus, $C_x(p)$ is independent of the position p, and $L_y \cap L_z$ is therefore a level set of x. □

On the downward pass of the CDP algorithm, we choose the position of a node based on the position of its successors. In the case of a node with fanout, placement is complicated by the Steiner problem. We implicitly construct the Steiner tree in the discrete metric as follows. The cost of a Steiner tree on the set of positions $\{\pi_1, \ldots, \pi_n\}$ in the discrete metric is $n-1$, which depends on the cardinality of the set but is independent of its elements. Thus, we minimize the contribution of the node fanning out to these positions by choosing the position in this set that minimizes the cost function of the node.

The final cost of the alignment is not necessarily given by the sum of the minimum costs of the sink nodes. Rather, once we have placed the nodes, we need to make a post-processing pass over the DAG and compute the cost of the alignment using equation (2).

The above lemmas lead to the following CDP algorithm for the axis and stride alignment problem. The algorithm finds a minimum-cost solution if the DAG is in fact a forest, but in the general case it is only a heuristic.

Algorithm 2 (Axis and stride alignment for a computation DAG.)
Input: A weighted computation DAG $G = (N, E, W)$.
Output: An access and stride alignment function π that reduces residual communication cost.
Method:

for $x \in N$ **in** topological order **do**
  if x is a unary operation node **then**
    Use equations (6)–(9) to propagate the cost information.
  else /* x is a binary operation node */
    LEVEL-SETS$(x) \leftarrow \varnothing$
    $y \leftarrow$ LCHILD(x); $z \leftarrow$ RCHILD(x)
    for all $L_y \in$ LEVEL-SETS(y) **and** $L_z \in$ LEVEL-SETS(z) **do**
      $L_x \leftarrow L_y \bigcap L_z$
      If $L_x \neq \varnothing$, add L_x to LEVEL-SETS(x), using Lemma 2 to compute the cost.
    enddo
  endif
enddo
Place sink nodes in minimum-cost positions.
for $x \in N$ **in** reverse topological order **do**
  $\sigma(x) \leftarrow \{r_x^*(\pi(y)) : (x, y) \in E\}$
  $\pi(x) \leftarrow$ any $p \in \sigma(x)$ that minimizes C_x
enddo
return π

The only concern in the upward sweep of the algorithm is the possibly exponential growth of the number of level sets. However, for the two most important cases of fixed source positions and unconstrained source positions, we can choose a level set partition of the universe that has a compact representation and linear growth rate.

Lemma 3 *Let $G = (N, E, W)$ be a computation DAG with s source nodes. If the alignment at each source node is either free or fixed, then the maximum number of level sets at any node of G is s.*

Proof: We proceed by induction on the height of the node. The base case is a node x of height 0, *i.e.*, a source node. If the node is free, the level set partition is $L_x = \{\mathbb{P}\}$. If the node is fixed at position π_x, the level set partition is $L_x = \{\{\pi_x\}, \mathbb{P} - \{\pi_x\}\}$.

For the induction case, assume that an internal node x has children y and z, and the level set partition of y is $L_y = \{B_0, B_1, \ldots, B_{m-1}\}$ and that of z is $L_z = \{C_0, C_1, \ldots, C_{n-1}\}$. The sets $B_1, \ldots, B_{m-1}, C_1, \ldots, C_{n-1}$ are singletons, $B_0 = \mathbb{P} - \bigcup_{i=1}^{m-1} B_i$, and $C_0 = \mathbb{P} - \bigcup_{i=1}^{n-1} C_i$. (This is the nature of level sets produced by the base case.) Now, by Lemma 2, the level sets of x are intersections of level sets of y and z. Hence the level set partition of x is $L_x = \{B_0 \cap C_0, B_1, \ldots, B_{m-1}, C_1, \ldots, C_{n-1}\}$, which proves the lemma. $\square$

Theorem 1 *Algorithm 2 runs in $O(|N|s^2)$ time on a DAG with node set N and s source nodes.*

We illustrate this algorithm on the DAG of Figure 3. Let T be the two-dimensional template with axes named i and j to which all array objects will be aligned. Let the position of array a be fixed at $(i,j)|(1,1)$, and the positions of b and c be free. Let $\mathbb{Q}_1 = \mathbb{Q} - \{1\}$. Then the level sets of the cost function for nodes a, b, and c are as shown in the second column of Table 4. We indicate a level set and its cost as a colon-separated pair.

In the first phase of Algorithm 2, we traverse the DAG from sources to sinks computing cost functions at each node. We obtain the cost functions shown in the second column of Table 4. The final choices of positions made in the second phase of Algorithm 2 for the nodes are shown in the third column of Table 4.

Nodes b and d have fanout and therefore require the Steiner optimization in the second phase. In the case of node b, there are multiple minimum-cost positions.

Nodes $t3$ and $t4$ are examples where strides are unconstrained. In such cases, we arbitrarily set the strides to 1.

Nodes b and c show that strides can be fractional. In this case, we multiply all strides by the least common multiple of the denominators of the rational strides.

After multiplying all the strides by 4 to make them integral, we obtain the final alignments shown in Figure 2. With this alignment the only residual communication occurs along the edge from node b to node $t5$: here general communication is used to simultaneously transpose the array b and change its stride. This is one of two minimum-cost solutions for this example.

5 Offset alignment

Axis and stride alignment require general communication, which is more expensive than the nearest-neighbor communication used in offset alignment. We therefore perform axis and stride alignment before offset alignment. If a DAG edge carries communication from axis and stride alignment, this subsumes communication due to offset alignment; we can perform both the axis/stride change and the shift in one general communication step. We model this during offset alignment by setting to zero the weights of those edges that already carry general communication from axis or stride alignment.

For the offset alignment problem, we use the metric (grid, ring, *etc.*) appropriate to the machine architecture. In this section, we concentrate on the grid (and hypercube) metric, $d_G(p,q) = |p-q|$. A k-dimensional grid is the Cartesian product of k one-dimensional grids, and the distance between two positions in the k-dimensional grid is the sum of the distances in the k one-dimensional grids between the appropriate projections of those positions. This allows the multidimensional problem to be separated into k independent one-dimensional problems. For this reason, it suffices to present the solution for a one-dimensional grid.

The offset alignment problem for grids can be reduced to integer programming (except for the Steiner correction). Let π_x be the position of node x in some alignment π. Then the residual communication cost (which is the function we want to minimize) is

$$F(\pi) = \sum_{(x,y)\in E} w_x |\pi_x - \pi_y|.$$

Program operations such as sectioning and shifts introduce a set of linear equality constraints. For instance, the statement `B = A(5:N)` introduces the constraint $\pi_B = \pi_A + 5$. To reduce the offset alignment problem to integer programming, introduce the variable θ_{xy} for every edge (x,y) of the DAG. For each such θ_{xy}, add the following two inequality constraints:

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

These constraints guarantee that $\theta_{xy} \geq |\pi_x - \pi_y|$. Also, change the objective function to be

$$G(\theta) = \sum_{(x,y)\in E} w_x \theta_{xy}.$$

The transformed problem is equivalent to the original one, because the minimum of $G(\theta)$ will always have $\theta_{xy} = |\pi_x - \pi_y|$.

Integer programming is NP-complete. However, we can relax the requirement that the solutions be integers, and solve the corresponding linear programming problem to obtain a solution for π. Since the solution is not necessarily integral, we must examine the integer-valued positions in the neighborhood of the solution and choose a position with minimum function value.

6 The free-source alignment problem

Once we have an algorithm for the fixed-source alignment problem, we can conceptually create an algorithm for the free-source alignment problem by searching through all possible placements of the source nodes. This exhaustive search is not practical. We therefore resort to a relaxation technique we call *cyclic source placement* that is related to Alternating Direction Search used in multivariate optimization [12]. In this method, we freeze the locations of all but

Node n	C_n	$\pi(n)$
a	$\{(i,j)\vert(1,1):0, (i,j)\vert(\mathbb{Q},\mathbb{Q}_1):\infty, (i,j)\vert(\mathbb{Q}_1,\mathbb{Q}):\infty, (j,i)\vert(\mathbb{Q},\mathbb{Q}):\infty\}$	$(i,j)\vert(1,1)$
b	$\{(i)\vert(\mathbb{Q}):0, (j)\vert(\mathbb{Q}):0\}$	$(i)\vert(1/2)$ or $(j)\vert(1)$
c	$\{(i,j)\vert(\mathbb{Q},\mathbb{Q}):0, (j,i)\vert(\mathbb{Q},\mathbb{Q}):0\}$	$(j,i)\vert(1,1/4)$
d	$\{(i,j)\vert(\mathbb{Q},\mathbb{Q}):0, (j,i)\vert(\mathbb{Q},\mathbb{Q}):0\}$	$(j,i)\vert(1,1)$
e	$\{(i)\vert(1):0, (i)\vert(\mathbb{Q}_1):5000, (j)\vert(\mathbb{Q}):5000\}$	$(i)\vert(1)$
f	$\{(i)\vert(\mathbb{Q}):0, (j)\vert(\mathbb{Q}):0\}$	$(i)\vert(1)$
g	$\{(i)\vert(1):0, (i)\vert(\mathbb{Q}_1):100, (j)\vert(\mathbb{Q}):100\}$	$(i)\vert(1)$
$t1$	$\{(j,i)\vert(\mathbb{Q},\mathbb{Q}):0, (i,j)\vert(\mathbb{Q},\mathbb{Q}):0\}$	$(i,j)\vert(1,1)$
$t2$	$\{(i,j)\vert(1,1):0, (i,j)\vert(\mathbb{Q}_1,\mathbb{Q}_1):5000, (j,i)\vert(\mathbb{Q},\mathbb{Q}):5000\}$	$(i,j)\vert(1,1)$
$t3$	$\{(j,i)\vert(\mathbb{Q},\mathbb{Q}):0, (i,j)\vert(\mathbb{Q},\mathbb{Q}):0\}$	$(j,i)\vert(1,\mathbb{Q})$
$t4$	$\{(i,j)\vert(\mathbb{Q},\mathbb{Q}):0, (j,i)\vert(\mathbb{Q},\mathbb{Q}):0\}$	$(j,i)\vert(1,\mathbb{Q})$
$t5$	$\{(i)\vert(\mathbb{Q}):0, (j)\vert(\mathbb{Q}):0\}$	$(i)\vert(1)$
$t6$	$\{(i)\vert(1):0, (i)\vert(\mathbb{Q}_1):50, (j)\vert(\mathbb{Q}):50\}$	$(i)\vert(1)$

Table 4: **Cost functions and final positions of nodes in the computation DAG of Figure 3.**

one of the source nodes, and compute an alignment and cost. This also provides an alignment for the free source node. If this results in a lower cost, we use this as the new alignment of the free source node, and move on to another source node. We cycle through the source nodes until the cost does not improve any further. The order of cycling is from source nodes of heavier weight to those of lighter weight, and, within the same weight, from nodes with larger outdegree to those of smaller outdegree. We have found that in practice the algorithm converges in 2 or 3 cycles and produces very good solutions. The details of the algorithm are given below. We only use cyclic source placement for the axis and stride components; linear programming directly solves the free-source variant of the axis alignment problem in the grid metric.

Algorithm 3 (Heuristic solution of free-source alignment problem with cyclic source placement.)
Input: A computation DAG $G = (N, E, W)$, and a position space $\mathbb{P}$.
Output: A low-cost alignment function π.
Method:

Sort the source nodes of the DAG by rank and, within rank, by outdegree into array L.
Place the variable $L[0]$ in an arbitrary orientation, and let all other variables be free.
Apply Algorithm 2 to obtain an initial estimate for π and $\kappa = C(G, d, \pi)$.
for iter = 1 **to** maxiter **do**
  for var = 0 **to** maxvar **do**
    $\pi^{(1)} \leftarrow \pi$ with $\pi(L[\text{var}])$ set to free
    Apply Algorithm 2 with source node placements from $\pi^{(1)}$ to obtain $\pi^{(2)}$ and $\kappa' = C(G, d, \pi^{(2)})$.
    if $\kappa' < \kappa$ **then**
      $\pi \leftarrow \pi^{(2)}$; $\kappa \leftarrow \kappa'$
    endif
  enddo
enddo
return π

7 Extensions and future work

This section deals with extensions the basic framework that make the analysis more powerful and allow the modeling of features such as replication and dynamic alignments. The ideas in this section are topics of current and future research and are subject to change based on future experience.

7.1 Control flow

Previous work by Chen and Wu [7] focused on finding alignments for π-blocks of code in isolation and then performing the necessary realignments at π-block boundaries. However, it is usually the case that a code block gives a set of constraints that must be satisfied rather than a unique solution, and there may be several alignments satisfying the constraints. Arbitrarily choosing one solution without reference to the uses of the variables in later blocks could cause more communication than necessary. This suggests the need for a more global approach.

Interblock alignment is a consideration only for variables that are live at the beginning or end of a block. Dead variables can be aligned optimally within their active block. This requires solving a standard data flow problem to identify live and dead variables [14].

Our approach to interblock alignment is based on the notion of traces [8] of the flow graph of the program. Our task is considerably simpler than that of classical trace scheduling for two reasons: our traces are more coarse-grained, and we are not performing code motion. We use traces to generate a prioritized list of larger "basic blocks" on which to apply our algorithms.

We perform alignment across the basic blocks in the current trace subject to constraints imposed by previous (higher-priority) traces. To handle off-trace constraints, we introduce *split* and *merge* nodes, whose purpose is to handle the necessary realignment in case of control flowing back and forth between a trace and a different, higher-priority trace. If a variable can arrive at a point on the current trace either from within the trace or from another trace, we represent it with a merge node. Suppose the weight of the data on each edge is D, and the edge counts for the on- and off-trace edges are N and F. Then the modified weights are $ND/(N+F)$ for the on-trace edge and $FD/(N+F)$ for the off-trace edge. The extension to multiway merge nodes is obvious. We do a similar thing, but in reverse, for split nodes. The addition of these nodes means that all constrained nodes of a trace are either source or sink nodes. Our previous algorithms can be applied to this modified graph. During code generation, the merge and split nodes generate any communication resulting from realignment across basic blocks.

7.2 Dynamic alignment

Consider the following loop, taken from Knobe *et al.* [19].

```
do i = 1, n
    t = a(i,:)
    a(i,:)  = b(i,:)
    b(i,:)  = t
enddo
```

It is possible to execute this loop without communication, but not with a fixed alignment for the vector `t`. The zero-communication solution is achieved by aligning the vector `t` with the i^{th} row of arrays `a` and `b` at the i^{th} iteration of the loop. This shows the need for dynamic alignment.

We propose dynamic alignments that have some structured dependence on loop induction variables, rather than those that are dependent on arbitrary runtime values of variables. Knobe *et al.* [18] deal with the latter kind of dynamic alignments. In particular, we consider making the stride and offset alignments of arrays affine functions of loop induction variables.

The example above was particularly simple since it had no loop-carried dependences. Each instance of the vector `t` was live only within a single loop iteration and could be optimized independently. However, consider a loop with loop-carried dependences on some array variables, and think of two iterations i and j spanned by such a dependence edge e on an array A. Given that the alignment of A is an affine function of the loop induction variable, we know the alignments of A at entry to iterations i and j, and we can compute the possible alignments and their associated costs for A at exit from iteration i. If the alignment of A at its exit from iteration i does not match its desired alignment on entry to iteration j, the edge e will carry residual communication, reflected in the dynamic realignment of A.

Clearly, we cannot actually unroll loops to perform this analysis; we must therefore find a representation that compactly encapsulates such interactions between loop iterations while maintaining the acyclic nature of the program graph if possible. Restricting alignments to affine functions of loop induction variables and assuming constant distance vectors gives a uniform pattern to the interaction between loop iterations. We generate a closed-form representation by introducing *transformer* and *comparator* nodes that encapsulate the interaction between loop iterations. A transformer node takes the alignment function of a variable on entry to the current iteration i and produces its desired alignment function on entry to the iteration j affected by the loop-carried dependence. A comparator node compares the alignments of a variable at exit from iteration i and on entry to iteration j (the latter being derived from a transformer node), and resolves the cost of realignment when the two are different. This captures inter-iteration dependences while allowing the optimization to be formulated as a DAG problem. However, some symbolic manipulation is required to solve this problem. We are currently investigating the symbolic capabilities required for such a system.

7.3 Replication

Replication can be modeled by allowing offsets to be sets of positions rather than a single position. These sets can be compactly represented as section specifiers. Thus, an alignment of the form $A(I) \boxplus T(I, 0:100:2)$ would cause the one-dimensional vector A to be replicated across a subset of the even-numbered columns of template T. We have identified two distinct kinds of replication: *value replication* and *name replication*. In the former, the values of a smaller array are replicated across a larger array; this is the case with SPREAD. In the second case, the locus of alignments of a smaller array span a larger array; this is the case with dynamic alignment functions, as in the example in Section 7.2, and is related to notions of ownership transfer proposed by Bala and Ferrante [3].

While regular sections are a reasonable syntactic mechanism for specifying replication, it is unclear how to use them for analysis. Replication is a vexing problem that has not been dealt with in current research, but is clearly important enough to merit serious attention.

7.4 Automatic distribution

This paper deals only with alignment, and has omitted a discussion of automatic distribution. The issues there are determining the number of processors to be assigned to each dimension of the template, and the block size of the distribution for each template dimension. Distribution offers the possibility of further reducing communication by "serializing" certain template dimensions in memory; however, such communication reduction must be balanced against the possible loss in parallelism. Automatic distribution can also be formulated as a constrained optimization problem. We have made some progress in modeling this problem. Gupta [13] has developed a constraint-based approach to automatic distribution.

8 Conclusions

We have presented a framework for the automatic determination of array alignments in data-parallel languages such as Fortran 90. We have decomposed alignment functions into three constituents: axis, stride, and offset. For each of these subproblems, we have shown how to solve the alignment problem for a basic block of code, possibly containing common subexpressions. The language model handles sectioning operations, reductions, spreads, transpositions, and masked operations. We make few assumptions and are not limited by the owner-computes rule. We have also presented some ideas on extending the analysis to handle control flow, dynamic alignments, and replication.

While alignment and distribution directives are being proposed in High Performance Fortran [15], most of the proponents agree that such user specification is a stopgap solution, and the ultimate solution should involve compiler analysis. In many ways, the situation is similar to that in the early days of vectorizing compilers. Vectorizers have grown in sophistication in the last decade, and are much less dependent on user directives. Our work has put the alignment problem on a firm theoretical foundation and provided solutions to some of the problems. However, many open problems remain to be solved before these techniques become part of the standard optimization toolkit in compilers for distributed-memory machines.

Acknowledgments

We gratefully acknowledge a helpful conversation with Mike Overton, who pointed out the transformation to reduce the offset alignment problem to integer programming. We thank Jay Sipelstein for his comments on an early draft of this paper.

References

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

[2] 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*, May 1991.

[3] Bala, V., and Ferrante, J. Explicit data placement (XDP): A methodology for explicit compile-time representation and optimization of data movement. In *Proceedings of the Second Workshop on Languages, Compilers, and Runtime Environments for Distributed Memory Multiprocessors* (Boulder, CO, Oct. 1992).

[4] Chapman, B., Mehrotra, P., and Zima, H. Programming in Vienna Fortran. *Scientific Programming 1*, 1 (Fall 1992), 31–50.

[5] Chatterjee, S. *Compiling Data-Parallel Programs for Efficient Execution on Shared-Memory Multiprocessors.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Oct. 1991. Available as Technical Report CMU-CS-91-189.

[6] Chatterjee, S., Gilbert, J. R., Schreiber, R., and Teng, S.-H. Optimal evaluation of array expressions on massively parallel machines. In *Proceedings of the Second Workshop on Languages, Compilers, and Runtime Environments for Distributed Memory Multiprocessors* (Boulder, CO, Oct. 1992). Also available as RIACS Technical Report TR 92.17.

[7] Chen, M. C., and Wu, J.-J. Optimizing FORTRAN-90 programs for data motion on massively parallel systems. Tech. Rep. YALEU/DCS/TR-882, Department of Computer Science, Yale University, New Haven, CT, Jan. 1992.

[8] Ellis, J. R. *Bulldog: A Compiler for VLIW Architectures.* ACM Doctoral Dissertation Awards. The MIT Press, Cambridge, MA, 1986.

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

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

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

[12] Gill, P. E., Murray, W., and Wright, M. H. *Practical Optimization.* Academic Press, Orlando, FL, 1981.

[13] Gupta, M. *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.

[14] Hecht, M. S. *Flow Analysis of Computer Programs.* North-Holland, New York, NY, 1977.

[15] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Draft, Sept. 1992.

[16] Hiranandani, S., Kennedy, K., and Tseng, C.-W. Compiler support for machine-independent parallel programming in Fortran D. Tech. Rep. Rice COMP TR90-149, Department of Computer Science, Rice University, Houston, TX, Feb. 1991.

[17] Iverson, K. E. *A Programming Language.* Wiley, New York, NY, 1962.

[18] Knobe, K., Lukas, J. D., and Dally, W. J. Dynamic alignment on distributed memory systems. In *Proceedings of the Third Workshop on Compilers for Parallel Computers* (Vienna, Austria, July 1992), Austrian Center for Parallel Computation, pp. 394–404.

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

[20] Leiserson, C. E. Fat-Trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput. C-34*, 10 (Oct. 1985), 892–901.

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

[22] Mace, M. E. *Memory Storage Patterns in Parallel Processing.* Kluwer international series in engineering and computer science. Kluwer Academic Press, Norwell, MA, 1987.

[23] Melzak, Z. On the problem of Steiner. *Canadian Mathematical Bulletin 4* (1961), 143–148.

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

[25] Wholey, S. *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.

[26] Winter, P. Steiner problem in networks: A survey. *Networks 17* (1987), 129–167.