Mobile and Replicated Alignment of Arrays in Data-Parallel Programs

Siddhartha Chatterjee * John R. Gilbert † Robert Schreiber *

Abstract

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

We solve two facets of the problem of finding alignments that reduce residual communication: we determine alignments that vary in loops, and objects that should have replicated alignments. We show that loop-dependent mobile alignment is sometimes necessary for optimum performance, and we provide algorithms with which a compiler can determine good mobile alignments for objects within `do` loops. We also identify situations in which replicated alignment is either required by the program itself (via `spread` operations) or can be used to improve performance. We propose an algorithm based on network flow that determines which objects to replicate so as to minimize the total amount of broadcast communication in replication. This work on mobile and replicated alignment extends our earlier work on determining static alignment.

1 Introduction

Parallelism is expressed in data-parallel array languages like Fortran 90 [1] in the form of operations on arrays and array sections. Compiling such a program for a distributed-memory parallel machine requires a model for the mapping of the data to the machine. We view the mapping as an *alignment* to a Cartesian index space called a *template*, followed by a *distribution* of the template to the processors. The alignment phase positions all array objects in the program with respect to each other so as to reduce realignment communication cost. In the distribution phase that follows, the template is distributed to the processors. This two-phase approach separates the language issues from the machine issues, and is used in Fortran D [7], High Performance Fortran [10], and CM-Fortran [16].

The goal of compilation is to produce data and work mappings that reduce completion time. Much of this goal can be achieved by judicious alignment of the arrays. We consider only alignment here.

Completion time has two components: computation and communication. Communication can be separated into *intrinsic* and *residual* communication. Intrinsic communication arises from computational operations such as reductions that require data motion as an integral part of the operation. Residual communication arises from nonlocal data references required in a computation whose operands are not mapped to the same processors. As we only consider alignment in this paper, we take the view that objects are mapped identically to processors if and only if they are aligned. We use the term *realignment* to refer to residual communication due to misalignment; we seek to determine array alignments that minimize realignment cost. Communication for transpose, spread, and vector-valued subscript operations can in some cases be removed by suitable alignment choices. Our theory makes these forms of communication residual rather than intrinsic, and thus encompasses such optimizations [5].

A suitable alignment for the code fragment of Figure 1(a) is shown in Figure 1(b). Note that *V* moves at each iteration of the loop; it has a mobile alignment.

In this paper, we present algorithms to automatically determine good mobile alignments. We develop a detailed and realistic model of realignment cost that accounts for control flow in loops, and we formulate the alignment problem as a constrained optimization of the realignment cost. We present approximate solutions for mobile stride and offset alignment for array objects occurring within loops, where we allow the offset alignment to be a compiler-determined affine function of loop induction variables. We also show that replication may be viewed as an extension of offset alignment, and show that the problem of determining

*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 and Contract NAS 2-13721 between NASA and the Universities Space Research Association (USRA).

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

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



do k = 1, 100

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

(a)

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

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

(b)

Figure 1: (a) A Fortran 90 program fragment requiring mobile alignment. (b) A mobile alignment for the program fragment.

the optimal replication strategy can be reduced to a network flow problem.

Several other authors have considered static alignment [2, 9, 12, 13, 17]. Our earlier research [4, 5, 8] dealt with static alignment. We extend that work to handle mobile alignment here. Knobe, Lukas, and Steele [12] and Knobe, Lukas, and Dally [11] address the issue of dynamic alignment. Their notion of dynamic alignment is alignment depending on quantities whose values are known only at runtime, which may include loop induction variables as well as other arbitrary runtime values. This paper focuses on mobile alignment in the context of loops, where the alignment of an object is an affine function of the loop induction variables.

The paper is organized as follows. Section 2 formalizes the notion of alignment and defines mobile alignment. It also introduces our graph model for the alignment problem. Section 3 poses and solves the problem of mobile stride alignment. Section 4 poses and solves the problem of mobile offset alignment, covering fixed- and variable-sized objects and loop nests. Section 5 describes an algorithm for determining replicated offset alignments. Finally, Section 6 presents conclusions, open problems, and future work.

2 The alignment problem

An alignment is a mapping that takes each element of an array to a cell of a *template*. The template is a conceptually infinite Cartesian grid, with as many dimensions as necessary; it is a piece of "graph paper" on which all the array objects in a program are positioned relative to each other. The *alignment phase* of compilation aligns all array objects of the program to the template. The *distribution phase* then assigns template cells to actual processors. This paper discusses only the alignment phase.

If A is a d-dimensional array, and g_1 through g_t are integer-valued functions, we write

$$A(i_1,\ldots,i_d) \boxplus T[g_1(i_1,\ldots,i_d),\ldots,g_t(i_1,\ldots,i_d)]$$

to mean that the specified element of A is aligned to the specified element of the t-dimensional template T. Multiple templates may be useful in some cases, but this paper only considers alignment to a single template. Thus we omit the template name and just write $A(i) \boxplus [g(i)]$, where i is a d-vector and g is a function from d-vectors to t-vectors.

We restrict our attention to alignments in which each axis of the array maps to a different axis of the template, and array elements are evenly spaced along template axes. Such an alignment has three components: *axis* (the mapping of array axes to template axes), *stride* (the spacing of array elements along each template axis), and *offset* (the position of the array origin along each template axis). Each g_k is thus either a constant f_k or a function of a single array index of the form $s_k i_{a_k} + f_k$. The array is aligned one-to-one into the template. (In Section 5, we extend this to one-to-many alignments in which an array can be replicated across some template axes.)

An *array-valued object* (object for short) is created by every array operation and by every assignment to a section of an array. The compiler determines an alignment for each object of the program rather than to each program variable. The alignment of an object in a loop may be a function of the loop induction variable; such an alignment is *mobile*.

2.1 Examples

We now give examples of the various kinds of alignment.

Example 1 (Offset alignment) Consider the statement

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

If the alignments are $A(i) \boxplus [i]$ and $B(i) \boxplus [i]$, then a one-unit nearest-neighbor shift is necessary. However, the statement can be executed without communication if $A(i) \boxplus [i]$ and $B(i) \boxplus [i-1]$.

Example 2 (Stride alignment) Consider the statement

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

If $A(i) \boxplus [i]$ and $B(i) \boxplus [i]$, then general communication is needed to bring *A* and the section of *B* together. The alignments $A(i) \boxplus [2i]$ and $B(i) \boxplus [i]$ avoid communication.

Example 3 (Axis alignment) Consider the statement

```
B = B + transpose(C),
```

where *B* and *C* are two-dimensional arrays. If $B(i_1, i_2) \boxplus [i_1, i_2]$ and $C(i_1, i_2) \boxplus [i_1, i_2]$, then general communication is needed to transpose *C*. However, if $B(i_1, i_2) \boxplus [i_1, i_2]$ and $C(i_1, i_2) \boxplus [i_2, i_1]$, then the operands are aligned, and no communication is necessary.

Example 4 (Mobile offset alignment) Consider the code fragment in Figure 1. This can be executed optimally if $A(i_1, i_2) \boxplus_k [i_1, i_2]$, and $V(i_1) \boxplus_k [k, i_1 - k + 1]$. We use the symbol $\boxplus_k$ to emphasize the dependence of the alignment on the loop induction variable k.

Example 5 (Mobile stride alignment) Consider the code fragment

```
real A(1000), B(1000), V(20)

do k = 1, 50
   V = V + A(1:20*k:k)
   B(1:20*k:k) = V
enddo
```

Suppose $A(i) \boxplus_k [i]$ and $B(i) \boxplus_k [i]$. If the stride alignment of *V* is static, then any alignment of V is equally good, with a cost of two general communications per iteration. The cost drops to one general communication per iteration with the mobile stride alignment $V(i) \boxplus_k [ki]$.

2.2 Alignment-distribution graphs

Our main tool in this paper is a modified and annotated data flow graph that we call the *alignment-distribution graph*, or ADG for short. In this section we briefly describe the ADG and formulate the alignment problem as an optimization problem on the ADG. A companion paper [3] presents a more formal and complete treatment of the ADG. The ADG is closely related to the static single-assignment form of programs developed by Cytron *et al.* [6]. Figure 2 shows the ADG for the program fragment in Figure 1.

Nodes in the ADG represent computation; edges represent flow of data. Alignments are associated with endpoints of edges, which we call *ports*. A node constrains the relative alignments of the ports representing its operands and its results. An edge carries residual communication cost if its ports have different alignments. The goal is to provide alignments for the ports that satisfy the node constraints and minimize the total edge cost.

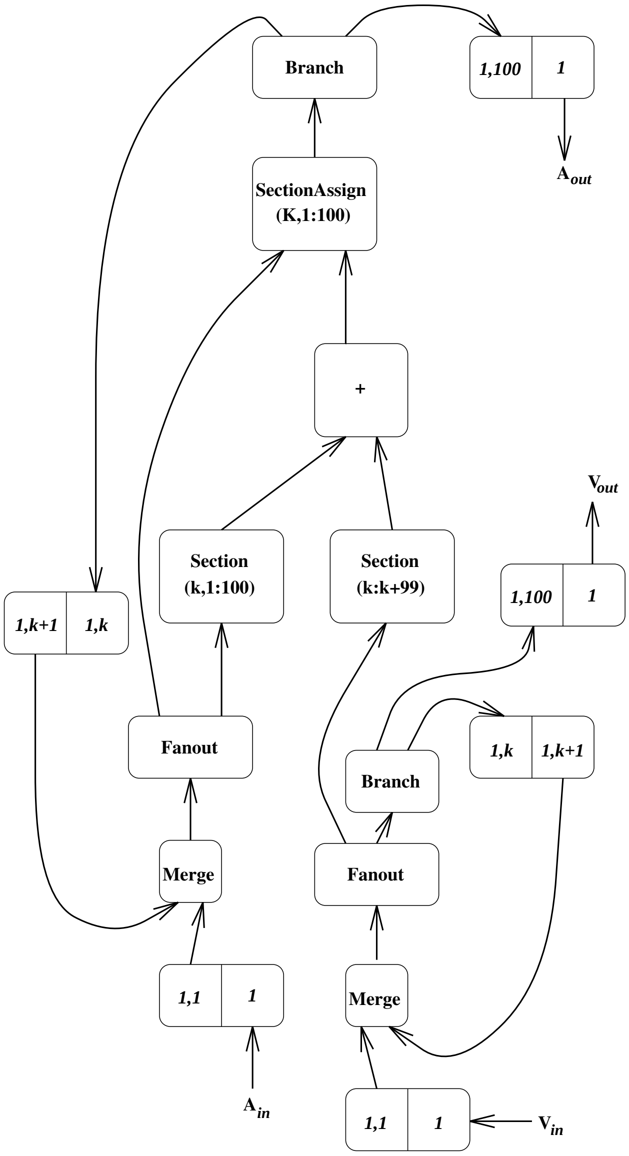

Figure 2: The ADG corresponding to the program fragment of Figure 1.

2.2.1 Edges

The ADG has a port for each (static) definition or use of an object. An edge joins the definition of an object with its use. Multiple definitions or uses are handled with merge, fanout, and branch nodes as described below. Thus every edge has exactly two ports. The purpose of the alignment phase is to label each port with an alignment. 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.

2.2.2 Nodes

Every array operation is a node of the ADG, with one port for each operand and one port for the result. Figure 2 contains examples of a "+" node representing elementwise addition, a *Section* node whose input is an array and whose output is a section of the array, and a *SectionAssign* node whose inputs are an array and a new object to replace a section of the array, and whose output is the modified array. (*SectionAssign* is called *Update* by Cytron *et al.* [6].)

When a single use of a value can be reached by multiple definitions, the ADG contains a *merge node* with one port for each definition and one port for the use. (This node corresponds to the ϕ-function of Cytron *et al.* [6].) When a single definition reaches multiple uses within the same basic block, the ADG contains a *fanout node*. When a single definition can reach multiple alternate uses (*e.g.*, due to conditional constructs), the ADG contains a *branch node*. Figure 2 contains examples of merge, fanout, and branch nodes. Fanout nodes represent opportunities for so-called *Steiner optimization*, as discussed in Section 6. Finally, the ADG for a program with loops contains *transformer nodes* that delimit iteration spaces as described below.

Nodes constrain the alignments of their ports. An elementwise operation like "+" constrains all its ports to have the same alignment. A merge or fanout node enforces the same constraint. If `A` is a two-dimensional array in a two-dimensional template, a node `transpose(A)` constrains its output to have the opposite axis alignment from its input; thus any communication necessary to transpose the array is assigned to the input or output edges rather than to the node itself. *Section* and *SectionAssign* nodes enforce constraints that describe the position of a section relative to the position of the whole array; for example, the node for the section `A(10:50:2)` constrains its output object to have the same axis as its input, twice the stride of its input, and an offset equal to 10 times the stride of A plus the offset of A.

2.2.3 Iteration spaces

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

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

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

2.3 Cost model

Finally, we describe the communication cost of the program in terms of the ADG. A *position* is an encoding of a legal alignment. The *distance* $d(p, q)$ between two positions p and q is a nonnegative number giving the cost per element to change the position of an array from p to q. The set of all positions is a metric space under the distance function d [4].

In this paper we will use two metrics: the *discrete metric*, in which $d(p, q) = 0$ if $p = q$ and $d(p, q) = 1$ otherwise, and the *grid metric*, in which p and q are grid points and $d(p, q)$ is the L_1 (or Manhattan) distance between them. We use the discrete metric to model axis and stride alignment, since any change of axis or stride requires general communication. The discrete metric is a simple model of general communication that abstracts away from such machine-specific details as routing, congestion, and software overhead. We use the grid metric to model offset alignment. The grid metric is *separable*, meaning that the distance between two points in a multidimensional grid is equal to the sum of the distances between their corresponding coordinates in one-dimensional grids. This property allows allows us to solve the offset alignment problem independently for each axis [4].

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

$$C(\pi) = \sum_{(x,y)\in E} \sum_{i\in\mathcal{I}_{xy}} w_{xy}(i) \cdot d(\pi_x(i), \pi_y(i)). \qquad (1)$$

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

2.4 Restrictions on mobile alignment functions

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

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

3 Mobile stride alignment

We use the discrete metric to model communication costs arising from stride changes. Let the strides at the ports of an edge be αi^T and $\alpha' i^T$. If $\alpha = \alpha'$, then the ports will be aligned at every iteration; if the constant terms α_0 and α'_0 differ but all other components are equal, then they are always misaligned; otherwise, they are almost always misaligned. We approximate this situation by considering the objects to be misaligned in all iterations unless $\alpha = \alpha'$.

As the distance function in equation (1) is independent of the LIV, we can move it outside the summation over the iteration space, and write the communication cost of edge (x, y) as the product of a weight and a distance. The distance is the discrete metric on $(k+1)$-vectors; the weight is the sum over all iterations of the size of the object at each iteration, $W = \sum_{i\in\mathcal{I}_{xy}} w_{xy}(i)$. Since the weight is polynomial in the LIVs, the sum can be evaluated in closed form. We can now use compact dynamic programming, a technique we have previously developed for static axis and stride alignment [5], to solve this problem.

4 Mobile offset alignment

Consider an object with offset alignment αi^T. 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 [5].

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

$$C_{xy} = \sum_{i\in\mathcal{I}_{xy}} w_{xy}(i)|(\alpha - \alpha')i^T|, \qquad (2)$$

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 (2) 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. After reviewing the solution for static offset alignment, we show the solution for fixed-size objects in singly-nested loops ($k = 1$), and then generalize to variable-size objects and to loop nests.

4.1 Offset alignment by linear programming

We review how the static offset alignment problem for the grid metric can be reduced to linear programming [5]. Let the integer π_x be the offset alignment of port x. Then the residual communication cost (which is the function we want to minimize) is $C(\pi) = \sum_{(x,y)\in E} C_{xy}(\pi)$; so

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

Nodes introduce linear constraints relating the offsets of their ports. See [3] for more details. To remove the absolute value from the objective function, we introduce a variable θ_{xy} for every edge (x, y) of the ADG, and add two inequality constraints,

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

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

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

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

If the offsets that result from the linear program are fractional, we round them to integers. The rounded solutions are not necessarily optimal integer solutions; in general, rounding an LP solution may not even preserve feasibility. However, in the case of offset alignment with the grid metric, we argue that rounding is a reasonable approach. It is straightforward to round the offsets so as to satisfy all the node constraints. The template can be thought of as a discrete approximation to a continuous L_1 metric space in which the edge costs are continuous functions of real-valued offsets. The unrounded LP optimizes this problem exactly, so we expect that the discrete optimum is not very sensitive to rounding. We will refer to this algorithm as rounded linear programming, or RLP. (We have also experimented with using mixed integer linear programming.)

4.2 Fixed-size objects and singly-nested loops

Assume for this section that the data weight of edge (x, y) is constant and equal to 1, and that $\mathcal{I}_{xy} = \ell : h : s$. Call $(\alpha - \alpha')i^T$ the *span* of edge (x, y) at iteration i. If the span does not change sign in the interval $[\ell, h]$ (as shown in Figure 3(a)), the summation and the absolute value in equation (2) can be interchanged. Then $C_{xy} = |\sum_{i \in \ell : h : s} (\alpha - \alpha')i^T|$, the closed form for which is

$$C_{xy} = \frac{h - \ell + s}{s} |(\alpha_0 - \alpha_0') + \frac{\ell + h}{2}(\alpha_1 - \alpha_1')|. \quad (3)$$

Note that the term inside the absolute value is the average distance spanned by edge (x, y). We can reduce this to RLP 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 3(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 follows:

$$C_{xy} = \sum_{j=1}^{m} \sum_{i \in \mathcal{I}_j} |(\alpha - \alpha')i^T|. \quad (4)$$

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

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

Now we fix m, expand the outer sum explicitly, and evaluate each inner sum using equation (3), as shown in Figure 3(c). Clearly, the span can change sign in at most one

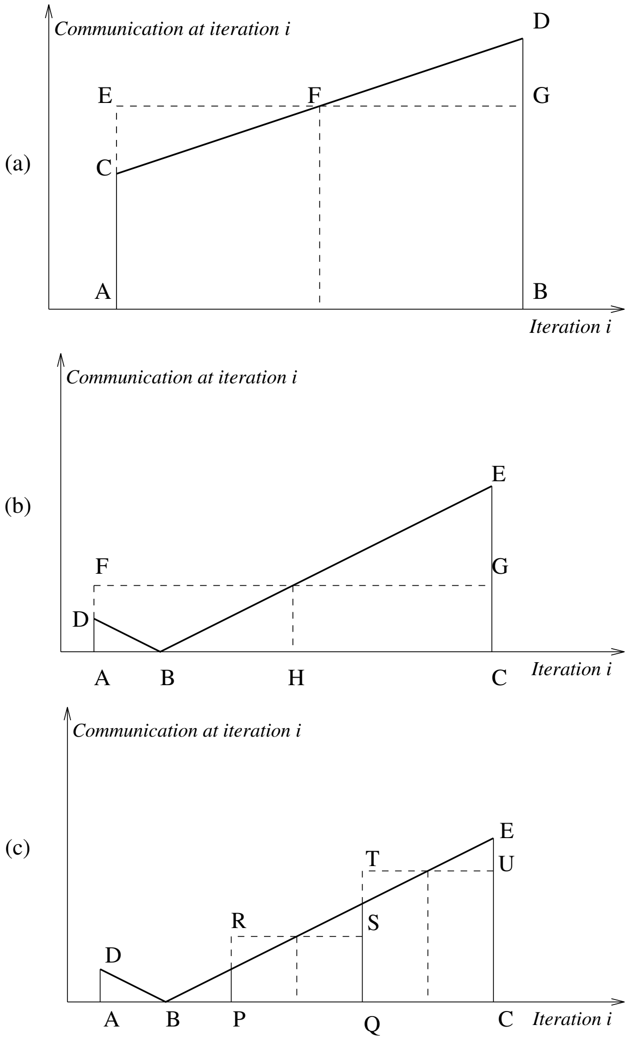

Figure 3: Approximating the cost of communication in loops. The actual communication cost is equal to the area under the heavy curve. (a) If the communication function does not have a zero crossing, then $ABDC \equiv ABGE$, and our approximation is exact. (b) If the communication function has a zero 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.

subrange; therefore, at least $(m-1)$ of the subrange sums are correct. We then reduce to RLP with m new variables per edge.

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

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

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

2. **State space search:** Approximate the iteration space as a single subrange, and use RLP. Using this solution as an initial guess, optimize the exact cost equation (4) by, for example, steepest descent.

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

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

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

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

4.3 Variable-size objects in singly-nested loops

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

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

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

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

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

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

We then determine the alignment coefficients as in Section 4.2.

4.4 Loop nests

The method generalizes to loop nests as follows. Divide the index range for each LIV into three subranges. The Cartesian product of this decomposition divides the iteration space into 3^k subranges, over each of which we assume that there is no sign change in the span; we sum the cost over each subrange, yielding one term in the approximate cost. We then solve for the minimizer of the approximate cost as in Section 4.2. It is also possible to use other quadrature rules to approximate the cost over each subrange.

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

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

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. We restrict the possible sets of positions to be triplets `l:h:s`.

A d-dimensional object aligned to a t-dimensional template has d *body axes* (which require axis, stride, and offset alignments) and $(t-d)$ *space axes* (which require only offset alignments). Our notion of replication allows the offset alignment along a space axis of an object to be a regular section of the corresponding template axis. We use the symbol $*$ to indicate replication across an entire template axis. For example, $A(i) \boxplus [i, 10]$ aligns A with one position along the second template axis; $A(i) \boxplus [i, 10\!:\!20\!:\!2]$ aligns A with a subset of the second template axis; and $A(i) \boxplus [i, *]$ replicates A across all of the second template axis. A broadcast communication occurs on an edge along which data flows from a fixed offset to a replicated offset.

5.1 Replication labeling

Offset alignment begins with a phase called *replication labeling*, whose purpose is to decide which ports of the ADG should have replicated positions. In this section, we propose an algorithm for replication labeling. Our algorithm labels ports as being replicated or non-replicated, but does not determine the extent of replication. Instead, we plan to generate the extents of replicated alignments in a storage optimization phase that follows replication.

There are three sources of replication:

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

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

Figure 4 shows why replication labeling is useful. In the example, a broadcast will occur in every iteration if A is not replicated, while a single broadcast will occur (at loop entry) if it is replicated. This is the solution found by our method.

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

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

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

Figure 4: Replication of the array *A*.

5.2 Labeling by network flow

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

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

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

Theorem 1 *An optimal replication labeling can be found by network flow.*

[1] 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.

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

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

Every replication labeling is a cut, and the cost of the labeling is the same as the cost of the cut. Every cut of finite cost is a replication labeling (since no infinite-cost edge can cross it in the forward direction), and hence a minimum-cost cut is an optimum replication labeling. The max flow/min cut theorem [14, Theorem 6.2] says that the cost of a minimum cut is the same as the value of a maximum flow from the source to the sink. □

Both the max flow and the min cut can be found in low-order polynomial time by any of several algorithms [14, 15]. In particular, it can be solved using linear programming. This is ideal for us, since we already require a linear programming package for determining mobile offset alignments. This is less efficient asymptotically than other methods, but should be adequate for our purposes.

6 Remarks and Conclusions

We have presented compiler optimizations for determining replication and mobile offsets within loops. We have proved that an optimal replication labeling can be found by network flow. For mobile alignment, we have presented an approximate reduction to rounded linear programming, with error bounds on the solution quality.

We now describe several extensions we are currently pursuing.

The framework for determining mobile offset alignment can be extended to handle user-defined runtime functions. The idea is to incorporate such functions in the offset alignment, and treat a mismatch in positions as a shift of unknown distance. This allows us to use techniques similar to those used in Section 4 to solve the problem.

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 use the control dependence graph [6] to associate a control weight c_e of execution with every edge e of the ADG, and minimize the *expected realignment cost*

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

Fanout nodes in an ADG represent the possibility of Steiner optimization, in which we determine an optimum fanout tree for communicating an object from the position in which it is defined to the positions in which it is used [5]. The fanout node is an approximation to a Steiner tree, which should be constructed in a pass after alignments have been determined.

Our replication algorithm does not determine the extent of replication for an object. This could be handled after replication labeling by propagating lower bounds on such extents. The algorithm also does not deal with storage allocation issues for replicated objects. In particular, it does not deal with the possibility of storing just one copy per physical processor rather than a copy per template cell. We feel that this decision fits with other storage optimization decisions in a separate phase of the compiler.

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

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

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

We now have a comprehensive theory of alignment analysis within a single procedure. Our next major efforts are to validate our approach by implementing these techniques, to develop a theory of distribution, and to understand the interprocedural aspects of alignment and distribution analysis.

References

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

[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] S. Chatterjee, J. R. Gilbert, and R. Schreiber. The alignment-distribution graph. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallelism*, Portland, OR, Aug. 1993. To appear.

[4] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. 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. Published in SIGPLAN Notices, 28(1), January 1993, pages 68–71. An expanded version is available as RIACS Technical Report TR 92.17 and Xerox PARC Technical Report CSL-92-11.

[5] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 16–28, Charleston, SC, Jan. 1993. Also available as RIACS Technical Report 92.18 and Xerox PARC Technical Report CSL-92-13.

[6] 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.

[7] 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.

[8] 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.

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

[10] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Draft, Jan. 1993. Also available as technical report CRPC-TR 92225, Center for Research on Parallel Computation, Rice University.

[11] 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.

[12] 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.

[13] 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.

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

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

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

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