Modeling Data-Parallel Programs with the Alignment-Distribution Graph *

Siddhartha Chatterjee † John R. Gilbert ‡ Robert Schreiber † Thomas J. Sheffler †

Corresponding author: Siddhartha Chatterjee

*An earlier version of this paper was presented at the Sixth Annual Workshop on Languages and Compilers for Parallelism, Portland, OR, 12–14 August 1993, and appears in *Languages and Compilers for Parallel Computing*, volume 768 of Lecture Notes in Computer Science series, Springer-Verlag, 1994.

†Research Institute for Advanced Computer Science, Mail Stop T27A-1, NASA Ames Research Center, Moffett Field, CA 94035-1000 (sc@riacs.edu, schreibr@riacs.edu, sheffler@riacs.edu). The work of these authors was supported by the NAS Systems Division via 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).

Abstract

Implementing a data-parallel language such as Fortran 90 on a distributed-memory parallel computer requires distributing aggregate data objects (such as arrays) among the processor memories. The mapping of objects to the machine determines the amount of residual communication needed to bring operands of parallel operations into alignment with each other. We present a program representation called the *alignment-distribution graph* that makes these communication requirements explicit. We describe the details of the representation, explain its construction from source text, show how to use it to model communication cost, and outline several algorithms for determining mappings that approximately minimize residual communication.

Keywords: To be supplied.

1 Introduction

When a data-parallel language such as Fortran 90 is implemented on a distributed-memory parallel computer, the aggregate data objects (arrays) have to be distributed among the multiple memory units of the machine. The mapping of objects to the machine 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 Cartesian grid called a *template*, and then a *distribution* that maps the template to the processors. This two-phase approach separates language issues from machine issues; it is used in Fortran D [1], High Performance Fortran [2], and CM-Fortran [3].

A compiler for a data-parallel language attempts to produce data and work mappings that reduce completion time. Completion time has two components: computation and communication. Communication can be separated into *intrinsic* and *residual* communication. Intrinsic communication arises from operations such as reductions that must move data as an integral part of the operation. Residual communication arises from nonlocal data references in an operation whose operands are not mapped to the same processors. We use the term *realignment* to refer to residual communication due to changes in alignment, and *redistribution* to refer to residual communication due to changes in distribution.

In this paper, we describe a representation of array-based data-parallel programs called the *Alignment-Distribution Graph*, or *ADG* for short. We show how to model residual communication cost with the ADG, describe how to construct the ADG from source code, and discuss algorithms for analyzing the alignment requirements of a program. The ADG is closely related to the *static single assignment* (*SSA* for short) form of programs developed by Cytron *et al.* [4], but is tailored for alignment and distribution analysis. In particular, it uses new techniques to represent the residual communication due to loop-carried dependences, assignments to sections of arrays, and transformational array operations such as reductions and spreads. Alignment and distribution can be phrased as discrete optimization problems on the ADG.

Section 2 of this paper defines the ADG and presents the most general version of our model of communication cost in terms of the ADG. Section 3 shows how to construct the ADG from source code. Section 4 discusses the use of the ADG in alignment analysis. Section 5 compares the ADG representation with SSA form and with the *preference graph*, another representation used in alignment and distribution analysis. Section 6 discusses open problems and future work.

2 The ADG representation of data-parallel programs

The ADG is a directed graph in which nodes represent computation, and edges represent flow of data. Figures 1–3 are examples of three Fortran 90 programs and their ADGs.

Alignments are associated with endpoints of edges, which we call *ports*. A node constrains the relative alignments of the ports that represent its operands and its results. Realignment occurs whenever the ports of an edge have different alignments. The goal of alignment analysis is to determine alignments for the ports that satisfy the node constraints and minimize the total realignment cost, which is a sum over edges of the cost of all realignment that occurs on that

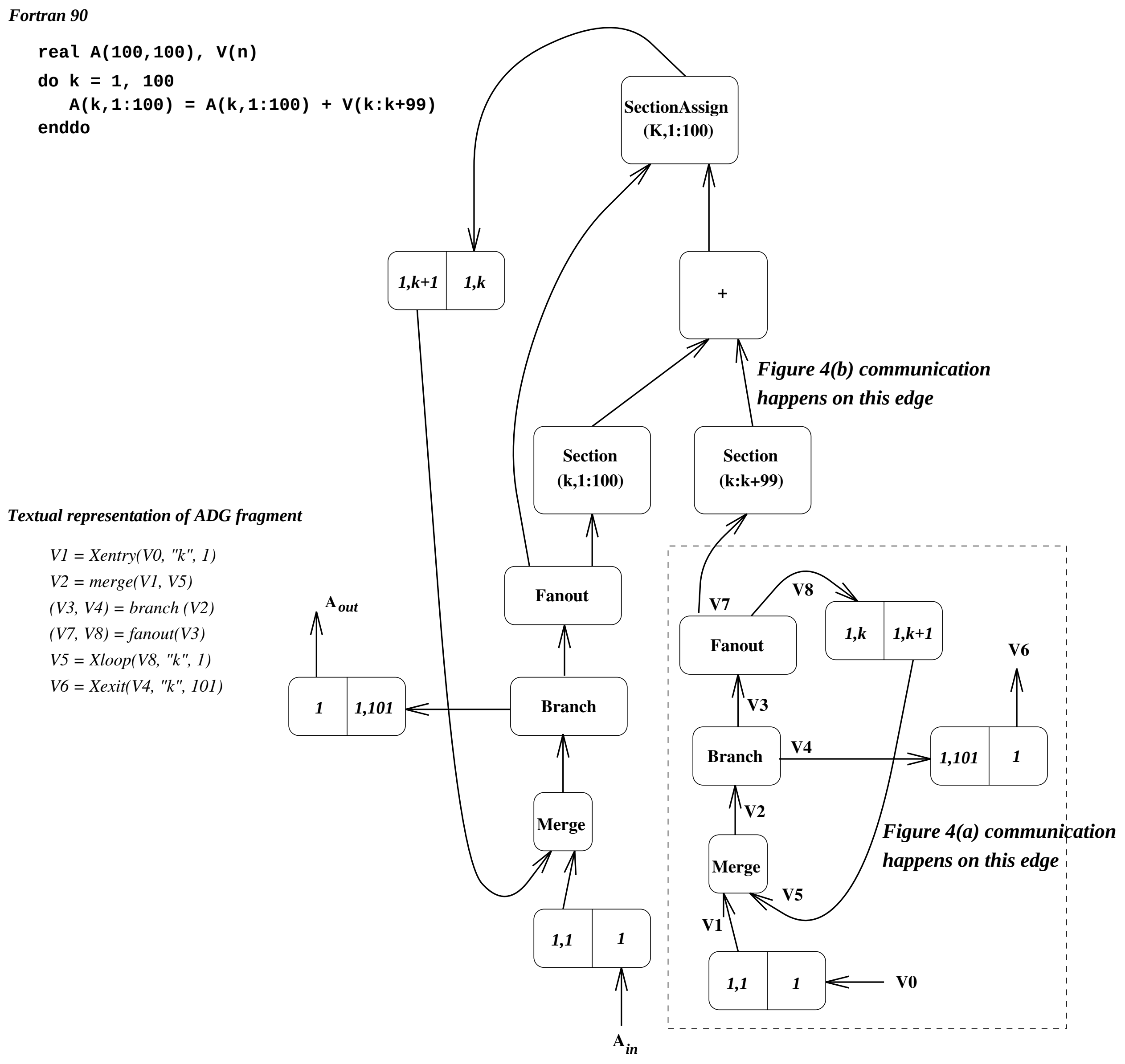


Figure 1: A Fortran 90 program fragment, its ADG, and the textual representation of the fragment of the ADG enclosed in the dashed box. $V0 (\equiv V_{in})$ represents the values and position of the vector V at the entry point of the code, and $V6 (\equiv V_{out})$ represents the values and position of V at the exit point of the code. A_{in} and A_{out} are the corresponding quantities for the array A.

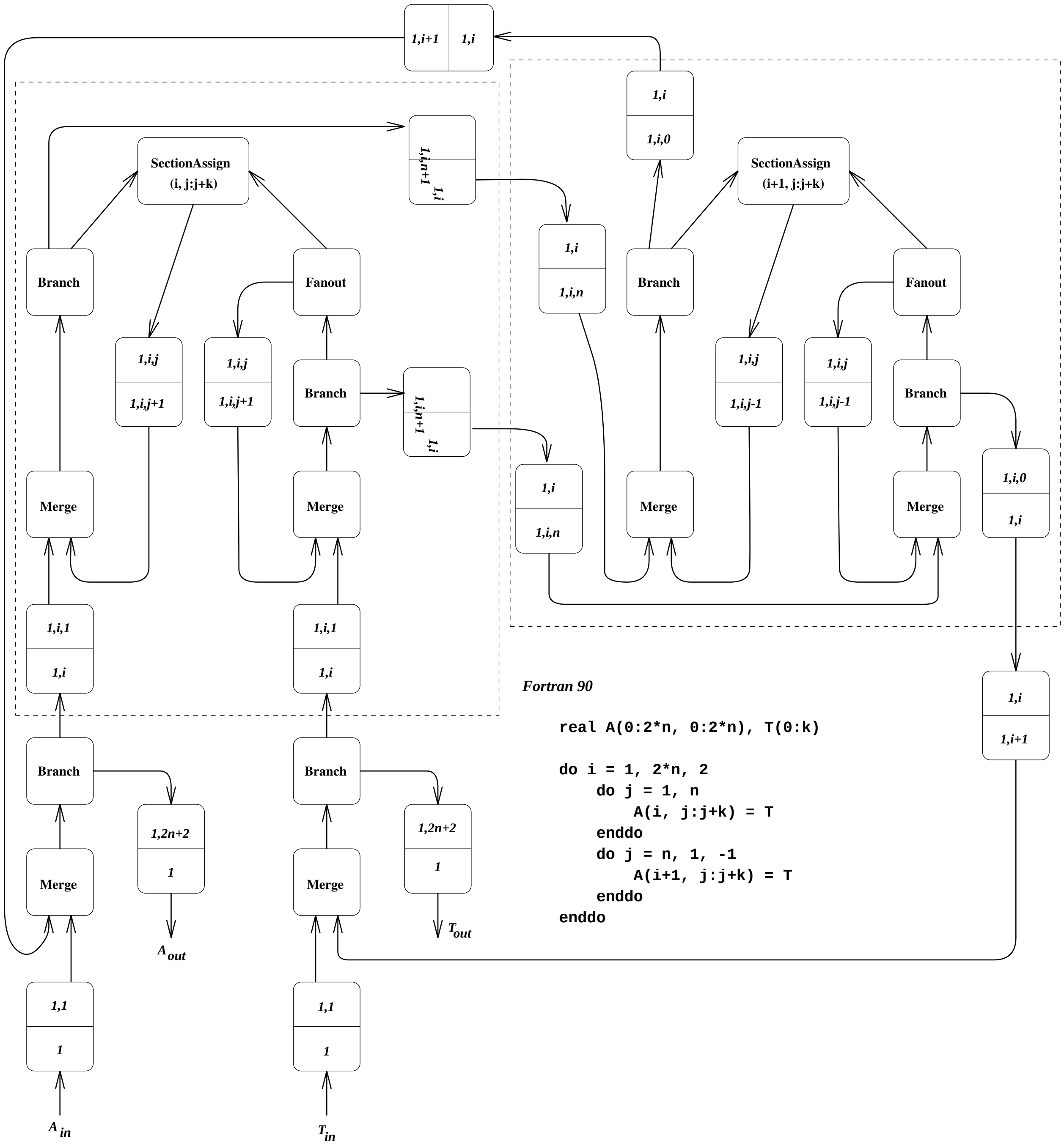


Figure 2: A Fortan 90 program fragment with a doubly-nested loop and its ADG. The nodes in the dashed boxes correspond to the inner loops.

```
do i = 1,5
   if (cond(i)) then
      A = A + B(i,:)
   else
      A = A + B(:,i)
   endif
enddo
```

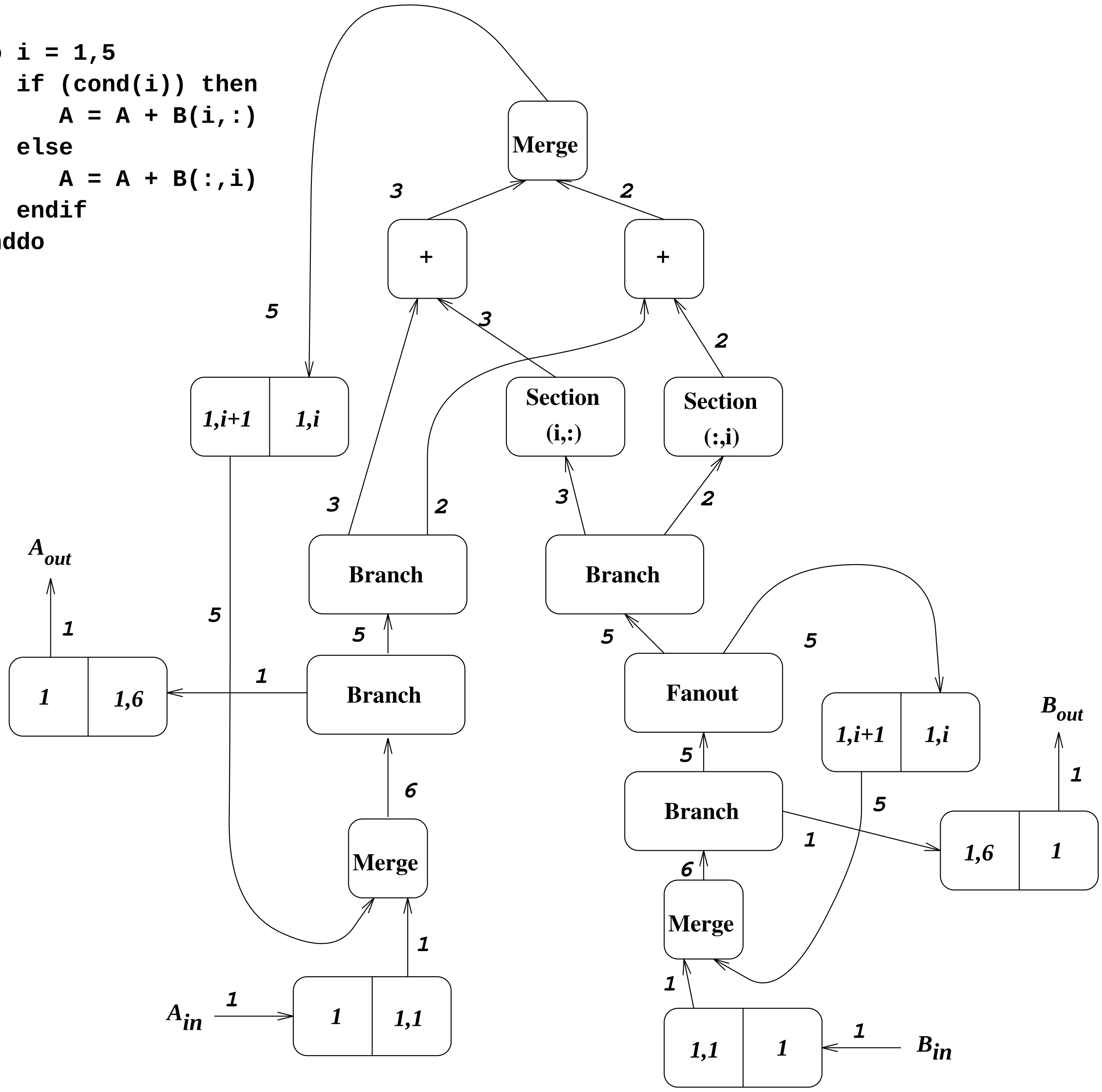
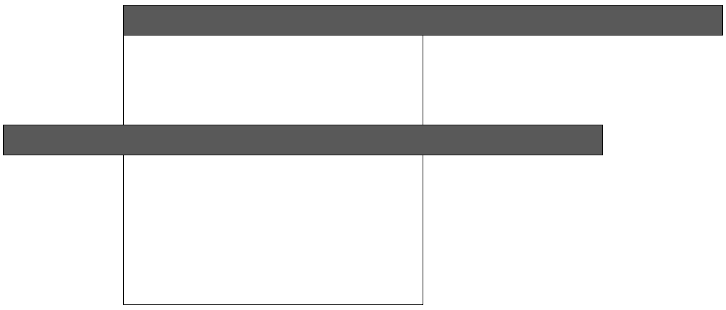


Figure 3: The ADG for a program with conditional branches. The labels on the edges are their control weights, assuming that the *then* branch of the conditional is taken 60% of the time, and that the *else* branch is taken 40% of the time.

(a)



(b)

Figure 4: Optimizing shift communication in the ADG of Figure 1. The optimum solution depends on the size of the vector V. (a) If V is small, the least communication involves moving all of V at each loop iteration. (b) If V is large, the least communication involves keeping it stationary at the central row of the array and moving the individual sections as needed.

edge during program execution. Similar mechanisms can be used for determining distributions. The remainder of this paper, however, focuses on alignment analysis.

To see the use of the ADG in alignment analysis, consider optimizing shift communication in the program fragment of Figure 1. (We will discuss the optimization algorithm in Section 4.3.) The point of interest is that the optimum depends of the size of the vector V, as shown in Figure 4. If V is small, it is optimal to shift all of V at each loop iteration. If V is very large, the optimal solution keeps V stationary at row 50 of the array A and move the sections as needed at each iteration. Depending on the value of the parameter n, our optimization algorithm finds one or the other solution. The edges that carry communication in the two cases are marked in Figure 1.

The ADG distinguishes between program array variables and array-valued objects in order to separate names from values. An *array-valued object* (object for short) is created by every array operation and by every assignment to a section of an array. Assignment to a whole array, on the other hand, merely names an existing object. The algorithms in Section 4.3 determine an alignment for each object in the program rather than for each program variable.

2.1 Position semantics

Traditional program analysis (*e.g.*, data flow analysis or dependence analysis) is based on *value semantics*: two objects are considered identical if their values are provably the same, and distinct otherwise. We need to strengthen the notion of identity of objects by considering both the values in the array and the position of the array in the machine. A communication action typically changes an object's position but not its values. Thus, we consider two objects identical if they have the same values and position. We call this nonstandard semantic interpretation of a program its *position semantics*. Gupta and Schonberg [5] consider position semantics in their work on data availability analysis.

Converting a source program to ADG form makes its position semantics explicit. The ADG thus contains position-transforming operations in addition to the usual value-transforming program operations.

2.2 Ports and alignment

The ADG has a *port* for each textual definition or use of an object. Ports are joined by edges as described below. The ports that represent the inputs and outputs of an operation are grouped together to form a *node*. Some ports are named and correspond to program variables; others are anonymous and correspond to intermediate values produced by the computation.

The principal attribute of a port is its *alignment*, which is an injective mapping of the elements of the object into the cells of a template. We use the notation

$$A(\theta_1, \ldots, \theta_d) \boxplus [g_1(\theta_1, \ldots, \theta_d), \ldots, g_t(\theta_1, \ldots, \theta_d)]$$

to indicate the alignment of the d-dimensional object A to the t-dimensional (unnamed) template. High Performance Fortran allows a program to use more than one template. Our theory extends to multiple templates, but in this paper, for simplicity, we assume that all array objects are aligned to a single template. The index variables θ_1 through θ_d are (implicitly) universally quantified in this formula.

An object in a nest of *do*-loops may have an alignment that depends on the loop induction variables (LIVs). For an object nested inside k loops with induction variables $\xi_1, \ldots, \xi_k$, we extend the notation to

$$A(\theta_1, \ldots, \theta_d) \boxplus_\xi [g_1(\theta_1, \ldots, \theta_d, \xi), \ldots, g_t(\theta_1, \ldots, \theta_d, \xi)],$$

where $\xi = (1, \xi_1, \ldots, \xi_k)^T$. The additional 1 at the beginning of ξ signifies that an object outside any loop nests has a position independent of any loop iteration variables. (See Section 4.2.2 for more details.) In this notation, the index variables are universally quantified, but the induction variables are free. An alignment that depends on LIVs is said to be *mobile*.

We restrict our attention to alignments in which each axis of an object maps to a different axis of the template, and elements are evenly spaced along template axes. Such an alignment has three components: *axis* (the mapping of object axes to template axes), *stride* (the spacing of successive elements along each template axis), and *offset* (the position of the object origin along each template axis). Each g_j is thus either a constant f_j (in which case the axis is called a *space axis*), or a function of a single array index of the form $s_j \theta_{a_j} + f_j$ (in which case it is called a *body axis*). There are d body axes and $(t-d)$ space axes. In matrix notation, the alignment $g_A(\theta)$ of object A can be written as

$$g_A(\theta) = L_A \theta + f_A \tag{1}$$

where L_A is a $t \times d$ matrix whose columns are orthogonal and contain exactly one nonzero element each, f_A is a t-vector, and $\theta = (\theta_1, \ldots, \theta_d)^T$. The elements of L_A and f_A are expressions in ξ. The nonzero structure of L_A gives the axis alignment, its nonzero values give the stride alignment, and f_A gives the offset alignment.

As an example, consider an array A aligned to template T using the HPF directive `ALIGN A(I,J) WITH T(3*J+6,1,I-10)`. In this case, $\theta = \begin{bmatrix} i \\ j \end{bmatrix}$, matrix $L_A = \begin{bmatrix} 0 & 3 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}$, and vector $f_A = \begin{bmatrix} 6 \\ 1 \\ -10 \end{bmatrix}$. The second template axis is the single space axis of A.

We also allow replication of objects. The offset of an object in a space axis of the template, rather than being a scalar, may be a set of values.

2.3 Nodes

ADG node types fall into three categories. The first category comprises the simple arithmetic and assignment operations such as array addition, array reduction, array assignment, and section operations. The second category deals with control flow and comprises the branch, merge, and fanout nodes. The third category comprises the transformer nodes, which handle mobile objects. Nodes in the first category are constructed directly from source program statements. The nodes of the last two categories are added during ADG construction and do not correspond directly to constructs visible to the programmer.

Every array operation is a node of the ADG, with one port for each operand and result. Figure 1 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. (*Section* and *SectionAssign* correspond to the *Access* and *Update* functions of SSA [4, §3.1].)

A word about *SectionAssign*: Although we have modeled updating a section (or element) of an array as the creation of a new array that differs from the old array only on the updated section, this does not constrain the implementation of this operation. This artifact is simply a means to model the communication possibilities in performing the operation: either the RHS object can be moved to the position of the section that it updates, or the whole array can be moved so that the section being updated aligns with the RHS object. Note that the only form of parallelism in our model is array parallelism. Had we been trying to determine noninterfering array updates to extract other forms of parallelism, we would want to use array dependence analysis [6]. Such analysis is orthogonal to the issues considered in this paper.

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. Intuitively, a merge node occurs at every join point in a program where alternate object definitions could converge. (This node corresponds to the ϕ-function of SSA [4].) Conversely, when a single definition can reach at most one of several possible uses, the ADG contains a *branch node*. Branch nodes have no counterpart in SSA form, because a sequential program stores a value in the same location regardless of which branch requires it. However, in a parallel distributed-memory model, alternate uses of a particular value may require different memory mappings. Figures 1, 2, and 3 contain examples of merge and branch nodes.

Now consider the situation where a single definition actually reaches multiple uses. This is different from the branching situation, in which a definition has several *alternative* uses. Given alignments for the definition and for all the uses, the optimal way to make the object available at the positions where it is used is through a *Steiner tree* [7] spanning the alignment at the definition, the alignments at the uses, and minimizing the sum of the edge lengths in the metric space of possible alignments. Determining the Steiner tree is NP-hard for most metric spaces. We therefore approximate the Steiner tree as a star, adding one additional node called a *fanout node* at the center of the star. Figures 1 and 3 contain examples of fanout nodes. There remains the possibility of replacing this star by a true Steiner tree in a later compilation phase.

Finally, for a program with *do*-loops, we need to characterize the introduction, removal, and update of LIVs, since data weights and alignments may be functions of these LIVs. Array objects become mobile upon the introduction of a new LIV, lose mobility upon the removal of an LIV, and change position upon the update of an LIV. Accordingly, for every edge that carries data into, out of, or around a loop, we insert a *transformer node* that enforces a relationship between the iteration spaces at its two ports. Figures 1, 2, and 3 contain examples.

ADG nodes define relations among the alignments of their ports, as well as among the data weights, control weights, and iteration spaces of their incident edges. The relations on alignments constrain the solution provided by alignment analysis. They must be satisfied for computation to be performed at the nodes. An alignment (of all ports, at all iterations of their iteration spaces) that satisfies the node constraints is said to be *feasible*.

The constraints force all realignment communication onto the edges of the ADG. By suitable choice of node constraints, the apparently "intrinsic" communication of operations such as transpose and spread can be exposed as realignment and subjected to optimization as well.

Only intrinsic communication and computation happens within the nodes. In our current language model, the only program operations with intrinsic communication are reduction and vector-valued subscripting (scatter and gather), which access values from different parts of an object as part of the computation.

2.4 Edges, iteration spaces, and control weights

An edge in the ADG connects the definition of an object with its use. Multiple definitions or uses are handled with merge, branch, and fanout 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. An edge has three attributes: data weight, iteration space, and control weight.

The *data weight* of an edge is the size of the object whose definition and use it connects. As the objects in our programs are rectangular arrays, the size of an object is the product of its extents. If an object is mobile, we allow its extents, and hence its weight, to be functions of the LIVs. We write the data weight of edge (x, y) at iteration ξ as $w_{xy}(\xi)$.

The ADG is a static representation of the data flow in a program. However, to model communication cost accurately, we must take control flow into account. The *branch* and *merge* nodes in the ADG represent forks and joins in control flow. Control flow has two effects: data may not always flow along an edge during program execution (due to conditional constructs), and data may flow along an edge multiple times during program execution (due to iterative constructs). An *activation* of an edge is an instance of data flowing along the edge during program execution. To model the communication cost correctly, we attach *iteration space* and *control weight* attributes to edges.

First consider a singly nested *do*-loop, as in Figure 1. Data flows once along the edges from the preceding computation into the loop, along the forward and loop-back edges of the loop at every iteration, and once, after the last iteration, out of the loop to the following computation. Summing the contribution of each edge over its iterations correctly accounts for the realignment cost of an execution of the loop construct. In general, an edge (x, y) inside a nest of k *do*-loops is labeled with an *iteration space* $\mathcal{I}_{xy} \subset \mathbb{Z}^{k+1}$, whose elements are the vectors ξ of values taken by the LIVs.[1] Both the size of the object on an edge and the alignment of the object at a port can be functions of the LIVs. The realignment and redistribution cost attributed to an edge is the sum of these costs over all iterations in its iteration space. An edge outside any *do*-loops has the trivial iteration space $\{(1)\}$, with one one-dimensional element.

For a program where the only control flow occurs in nests of *do*-loops, iteration spaces exactly capture the activations of an edge. However, with *while*- and *repeat*-loops, *if-then-else* constructs, conditional *goto*s, and so on, iteration spaces can both underestimate and overestimate communication. First, consider a *do*-loop nested within a *repeat*-loop. In this case, the iteration space indicated by the *do*-loop may underestimate the actual number of activations of the edges in the loop body. Second, because of *if-then-else* constructs in a loop, an edge may be activated on only a subset of its iteration space. For this reason, we associate a control weight $c_{xy}(\xi)$ with every edge (x, y) and every iteration ξ

[1] To be completely formal, iteration spaces should be associated with ports rather than edges. However, iteration spaces can change only between the input and output ports of transformer nodes. Thus, the two ports comprising an edge must have the same iteration space, and the iteration space can be associated directly with the edge.

in its iteration space. We think of $c_{xy}(\xi)$ as the expected number of activations of the edge (x, y) on an iteration with LIVs equal to ξ. Control weights enter multiplicatively into our estimate of communication cost.

Consider the *if-then-else* construct in the code of Figure 3. In the ADG, we have introduced two *branch* nodes, since the values `A` and `B` can flow to one, but not both, of two alternative uses, depending on the outcome of the conditional. If the outcomes were known, we could simply partition the iteration space accordingly, and assign to each edge leaving these branch nodes the exact set of iterations on which data flows over the edge. Since this is impractical, we label these edges with the whole iteration space $\{(1,1),(1,2),(1,3),(1,4),(1,5)\}$. The control weights of these edges shown in Figure 3 approximate the dynamic behavior of the program.

Iteration spaces and control weights both model multiple activations of an ADG edge. The iteration space approach gives the more accurate model of communication cost. When an exact iteration space can be determined statically, as in the case of *do*-loops, we use it to characterize control flow. We use control weights only when an exact iteration space cannot determined statically.

2.5 Modeling residual communication cost using the ADG

The ADG describes the structural properties of the program that we need for alignment and distribution analysis. Residual communication occurs on the edges of the ADG, but so far we have not indicated how to estimate this cost. This missing piece is the *distance function* d, where the distance $d(p, q)$ between two alignments p and q is a nonnegative number giving the cost per element to change the alignment of an array from p to q. The set of all alignments is normally a metric space under the distance function d [8]. We discuss the structure of d in Section 4.2.

We model the communication cost of the program as follows. Let E be the edge set of the ADG G, and let $\mathcal{I}_{xy}$ be the iteration space of edge (x, y). For a vector ξ in $\mathcal{I}_{xy}$, let $w_{xy}(\xi)$ be the data weight, and let $c_{xy}(\xi)$ be the control weight of the edge. Finally, let π be a feasible alignment for the program. Then the realignment cost of edge (x, y) at iteration ξ is $c_{xy}(\xi) \cdot w_{xy}(\xi) \cdot d(\pi_x(\xi), \pi_y(\xi))$, and the total realignment cost of the ADG is

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

This cost model contains two main assumptions.

1. We assume that communications happen one at a time. This assumption is justifiable if the problem size is much greater than the machine size (so that each communication action fills up the entire machine), which is the usual mode of operation on parallel machines. Further, allowing simultaneous disjoint communication actions would make bothe the analysis and the subsequent code generation more complicated.

2. We have ignored the possibility of overlapping computation and communication. Bouchitté *et al.* [9] show that allowing for such overlap renders even the simplest version of the alignment optimization problem NP-complete.

Our goal is to choose π to minimize the cost in (2), subject to the node constraints. An analogous framework can be used to model redistribution cost.

3 Constructing the ADG from source code

This section describes how to translate a source program into ADG form through a series of program transformations (shown in Figure 5). ADG form is closely related to SSA form but incorporates position semantics into the notion of object identity as expressed by branch, merge, transformer, and fanout nodes.

We use a textual representation for the ADG that describes the graph as a program consisting of what appear to be invocations of ADG-node functions. The right-hand-side arguments to each node are its input ports, and the left-hand-side values of each assignment statement are the node's output ports. Two ports with the same name define an edge. Figure 1 shows the textual form for a fragment of the ADG.

Each variable name represents a single edge, implying that *each variable must have a single definition and a single use*. Source programs in their original form rarely obey this constraint. However, as program text is converted into ADG form, the transformation steps ensure that this property is achieved.

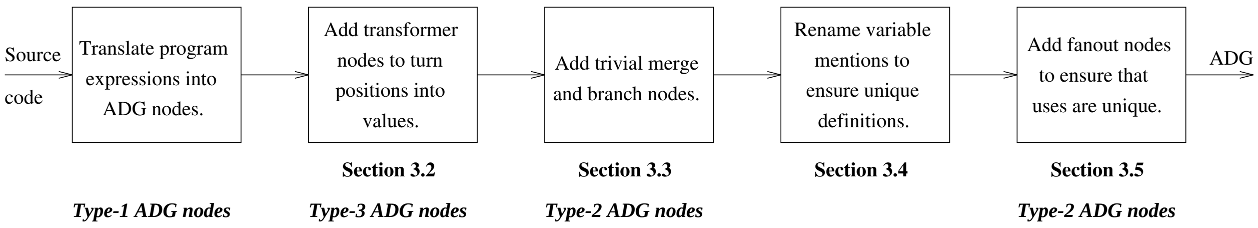


Figure 5: Translation of source code into ADG form.

As stated in Section 2.3, ADG nodes fall into three categories. The first step in the conversion from source code into ADG form is the statement-by-statement translation of array statements into ADG nodes. Complex expressions are flattened into primitive operations and temporaries are generated in this step. This translation phase generates all ADG nodes of the first type.

The remainder of this section develops the necessary algorithms for the placement of nodes of the second and third types. Section 3.1 recapitulates some basic compiler algorithms and representations. Using the idea of position semantics, we identify the locations of transformer nodes in Section 3.2. The placement of control flow nodes requires a significant amount of program analysis. Section 3.3 develops the algorithms to insert these nodes, and Section 3.4 shows how to rename all variables to ensure the uniqueness constraint discussed above.

3.1 Preliminaries

The basis for ADG translation is a representation of the source program as a control flow graph (CFG) [10]. The translation process strongly resembles the translation of a program into SSA form [4]. For the sake of completeness, we now review some standard compiler terminology and the properties of SSA form.

Basic blocks and the CFG A basic block is a code sequence in which control flow enters at the first statement and exits at the last statement. For our purposes, it does not matter whether basic blocks are maximal or not. A CFG of a program is a graph with a node for each basic block and an edge representing the possibility of control flow from one block to another. In addition, the CFG has two additional nodes called ENTRY and EXIT. Program execution begins in ENTRY and terminates in EXIT. We assume that each variable in the program is initialized in ENTRY.

Paths Each basic block B has a set of successor blocks in the CFG, denoted $\mathit{Succ}(B)$, and a set of predecessors, $\mathit{Pred}(B)$. A *path* in the CFG of length k is a sequence of $k+1$ nodes $B_1, \ldots, B_{k+1}$ and k edges denoted $((B_1, B_2), (B_2, B_3), \ldots, (B_k, B_{k+1}))$. A path from node B_i to node B_j is written as $B_i \Rightarrow B_j$. A path is *simple* if all nodes on the path are distinct; a *null* path has length 0. In this discussion, all paths are assumed to be non-null unless otherwise stated.

Two paths *converge* at a node Z if there are paths $X \Rightarrow Z$ and $Y \Rightarrow Z$ such that $X \neq Y$ and the set of nodes visited on each path are disjoint except for Z. Similarly, two paths may originate at the same node and *diverge*.

Dominators A node X is said to *dominate* node Y if all paths from ENTRY to Y pass through X [4]. The dominance relation is written as $X \underline{\gg} Y$. If $X \underline{\gg} Y$ but $X \neq Y$, then X *strictly* dominates Y, written $X \gg Y$. The *immediate dominator* of Y, denoted $\mathit{idom}(Y)$, is the closest strict dominator of Y. In a *dominator tree* of the CFG nodes, Y is a child of X if $X = \mathit{idom}(Y)$. In this paper, the terms "child" and "parent" refer to relations in the dominator tree.

Loops For a program represented as a CFG, a *loop* is identified as a strongly connected component with one block that dominates all of the other blocks of the component [10]. This block is called the loop HEADER block. Each loop also has at least one BACK edge, which is identified as an edge whose head dominates its tail. Finally, edges that leave a loop are called EXIT edges.

Many loops may be characterized by a loop induction variable (LIV), which is a variable whose value is incremented by a fixed amount each trip through the loop. Loop analysis is an extensively studied topic for which there are well-known algorithms for determining LIVs [10]. We assume that loops have been identified and that LIV recognition has been performed.

Dominance frontiers The dominance frontier function relates a node in the CFG to nodes immediately beyond those nodes that it dominates. Specifically, the *dominance frontier* of a node X, $DF(X)$, is the set of all CFG nodes Y such that X dominates a predecessor of Y but does not strictly dominate Y [4]. Thus, if Z is in the dominance frontier of X, then there is a path from X to Z, but there is some other path from ENTRY to Z that avoids X entirely.

Dominance frontiers extend to sets of nodes. If S is a set of nodes, $DF(S)$ is the union of the dominance frontiers of the members of S. The *iterated dominance frontier* $DF^{+}(S)$ is the limit of the increasing sequence of sets of nodes

$$DF_1 = DF(S)$$

$$DF_{i+1} = DF(S \cup DF_i).$$

Efficient algorithms are known for finding iterated dominance frontiers without enumerating this sequence explicitly [4].

Static single assignment (SSA) form A program is in *SSA form* if each variable is the target of exactly one assignment statement in the program text [4]. Any program can be translated to SSA form by renaming variables and introducing a pseudo-assignment called a ϕ-function at some of the join nodes in the control flow graph of the program. Cytron *et al.* [4] present an efficient algorithm to compute minimal SSA form (*i.e.*, SSA form with the smallest number of ϕ-functions inserted) for programs with arbitrary control flow graphs. Johnson and Pingali [11] have recently improved the complexity of SSA-conversion.

SSA form is commonly defined for sequential scalar languages, but this is not fundamental. It can be used for array languages if care is taken to properly model references and updates to individual array elements and array sections [4, §3.1]. The ADG uses SSA form in this manner.

A major contribution of SSA form is the separation between the values manipulated by a program and the storage locations where the values are kept. This separation, which allows greater opportunities for optimization, is the primary reason for basing the ADG on SSA form. Cytron *et al.* discuss two optimizations (dead code elimination and storage allocation by coloring) that produce efficient object code [4, §7]. After optimization, the program must be translated from SSA form to object code.

3.2 Transformer nodes

Data object creation is easy to recognize in imperative programming languages: an expression or the result of a function application creates a new object. However, no textual representation of mobile objects exists in a source program because their creation and update are not represented explicitly. Instead, these operations occur as a function of some LIV. Thus, the second phase in the translation from source code to ADG form is to make mobile objects explicit in the source code as new values.

There are three types of transformer nodes, corresponding to the introduction of mobility to an object, the removal of mobility from an object, and an update in a mobile object's alignment. An entry transformer function introduces a new object whose alignment is mobile with respect to a given LIV.

```
A = Xentry(A, <LIV>, <entry value>)
```

Similarly, the exit transformer function removes a degree of mobility from an object.

```
A = Xexit(A, <LIV>, <exit value>)
```

The result of this function is an object whose alignment is no longer a function of the LIV. Because of the symmetric use of transformer-entry and transformer-exit functions, they must be properly nested in a textual representation of the ADG.

The third function is the loop-back transformer. The increment with which the LIV is updated each trip through the loop is required as an argument to this type of transformer.

```
A = Xloop(A, <LIV>, <increment value>)
```

The return values of transformer "functions" reflect the potential effects of mobility on the identity of objects. This simplifies further analysis of the program by reducing position semantics to familiar value semantics. In particular, the insertion of transformer nodes reduces the placement of merge nodes to the placement of ϕ-nodes in SSA form. We now describe the algorithm that introduces transformer nodes in the CFG of a program.

To simplify the transformer node placement algorithm, we insert a number of empty blocks are inserted into the CFG. The transformer insertion phase adds code to these blocks. To each loop, we add a PRE-HEADER [10], an empty code block into which all arcs entering the loop HEADER block are re-directed. This ensures that each loop HEADER block has only a single preceding block. Similarly, we add a POST-BODY block that is executed after each iteration, and PRE-EXIT blocks that are executed prior to traversing each exit edge.

Loop analysis identifies the blocks of each loop and determines those that have loop induction variables (LIVs). Loops without LIVs are not candidates for mobile objects. Loop analysis should provide the following information for each loop: its LIV and increment value INCR; its ENTRY, PRE-HEADER, and POST-BODY blocks; the blocks of the loop BODY; and a list of all variables referenced (used or defined) within the loop body.

The transformer node placement algorithm examines each loop in turn, in any order. In fact, the order of visiting nested loops does not matter. For each variable V referenced in the BODY of the loop, we add three new ADG nodes.

- In the PRE-HEADER block, `V = Xentry(V, LIV, INIT)`
- In the POST-BODY block, `V = Xloop(V, LIV, INCR)`
- For each PRE-EXIT block, `V = Xexit(V, LIV, FINAL)`

Later phases rename the variables to ensure the ADG uniqueness criterion.

These steps ensure that prior to entering a loop, each variable referenced in the loop is transformed into a mobile object. For each iteration of the loop, a loop-back transformer updates the position of the object by using the increment value. Finally, objects made mobile upon loop entry lose their mobility on any path exiting the loop.

3.3 Merge, branch, and fanout nodes

The final phase of program transformation adds merge, branch, and fanout nodes, and also renames variables to ensure the single-definition, single-use criterion. Merge, branch, and fanout nodes are special because they reflect the effects of control flow on the flow of data values through a program.

A merge node is introduced when alternate definitions of the same object could possibly reach the same point in a program. A branch node is complementary: if a single definition can reach mutually exclusive alternate references, then a branch node supplies a copy of the object to each alternate branch. Lastly, fanout nodes create copies of an object when one definition reaches many references.

The following criteria define the required locations for these node types.

1. If two nonnull paths $X \Rightarrow Z$ and $Y \Rightarrow Z$ converge at basic block Z and both X and Y modify variable V, then Z must contain a merge node for V.
2. If two nonnull paths $Z \Rightarrow X$ and $Z \Rightarrow Y$ diverge at basic block Z and both X and Y contain references to variable V, then Z must contain a branch node for V.
3. No single definition has more than one use.

Merge and branch node insertion is a two-step process based on the method of Cytron *et al.* for the translation of programs to SSA form [4]. The first step determines the locations of the nodes and inserts trivial merge and branch functions. The second step renames the variables. Trivial merge or branch functions have the following form.

```
V = merge(V, ..., V)
(V, ..., V) = branch(V)
```

Each instance of the variable name is called a *mention* of *V*. A merge node contains as many mentions of *V* on the RHS as there are predecessors to the block containing the merge node. Branch nodes have as many mentions of *V* on the LHS as there are successors of the block in which the node occurs. The *renaming* algorithm will later replace mentions of variable names with new, unique names.

Fanout nodes are added last, as described in Section 3.5.

For each variable V, let D be the set of CFG nodes that *modify* V,[2] and let R be the set of CFG nodes that contain *references* to V. Using the dominance frontier relation, determining a minimal set of locations for merge nodes is simple. The following lemma is fundamental.

Lemma 1 (Cytron *et al.* [4]) *Let $X \neq Y$ be two nodes in the CFG and suppose that nonnull paths $p : X \Rightarrow Z$ and $q : Y \Rightarrow Z$ in the CFG converge at Z. Then $Z \in \mathit{DF}^+(\{X\}) \cup \mathit{DF}^+(\{Y\})$.*

Using this fact, the set of blocks that require ADG merge nodes for V is

$$M_1 = \mathit{DF}^+(D \cup R).$$

Lemma 2 *The set M_1 satisfies criterion 1 for merge node placement.*

Proof: This follows directly from Lemma 1 and the observation that if X and Y are any two CFG nodes that define or reference variable V then $\{X, Y\} \subseteq D \cup R$, which implies $(\mathit{DF}^+(\{X\}) \cup \mathit{DF}^+(\{Y\})) \subseteq \mathit{DF}^+(D \cup R)$. □

The location of branch nodes is determined using the reverse control flow graph (RCFG), which is the CFG with all edges reversed. The dominance frontier function computed on the RCFG is denoted RDF^+. The set of blocks that require branch nodes is

$$B_1 = \mathit{RDF}^+(R).$$

This set of nodes is sufficient to satisfy criterion 2 for branch node placement because of Lemma 1 on the RCFG.

Lemma 3 *Set B_1 satisfies criterion 2 for branch node placement.*

Proof: Note that if two nonnull paths $Z \Rightarrow X$ and $Z \Rightarrow Y$ diverge at node Z, then the RCFG contains converging paths $X \Rightarrow Z$ and $Y \Rightarrow Z$. The proof follows directly from Lemma 1 and the observation that if X and Y are any two nodes in the RCFG that reference variable V then $\{X, Y\} \subseteq R$, which implies $(\mathit{DF}^+(\{X\}) \cup \mathit{DF}^+(\{Y\})) \subseteq \mathit{DF}^+(R)$. □

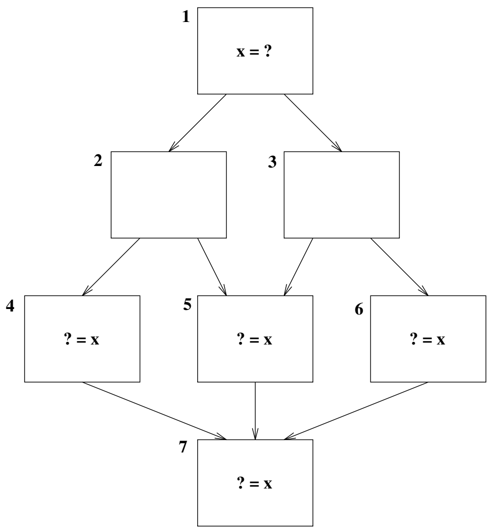
The placement of merge and branch nodes has the effect of introducing both new definitions and new references to variables. While the algorithm of Cytron *et al.* can handle this for either merge nodes or branch nodes individually, the interaction of the two is more complex. In effect, new merge nodes introduce the requirement for more branch nodes, and new branch nodes require new merge nodes. Consider the CFG shown in Figure 6, where the introduction of branch nodes in blocks 2 and 3 induces a merge node in block 5. Thus we have a mutually recursive definition of the locations of the required nodes.

$$M_{i+1} = \mathit{DF}^+(M_i \cup B_i) \cup M_i$$

$$B_{i+1} = \mathit{RDF}^+(M_i \cup B_i) \cup B_i$$

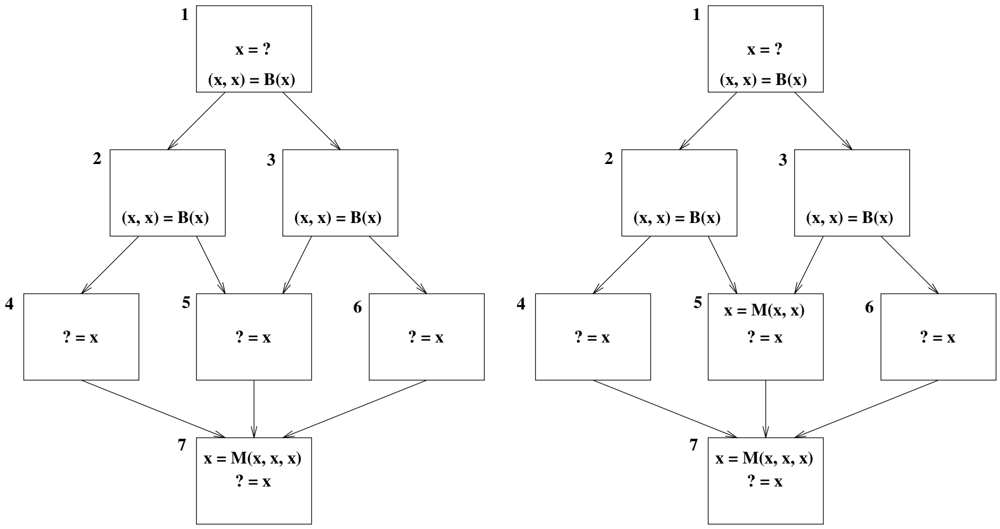
The locations of merge and branch nodes are the limits of the increasing sequence of sets defined by these equations. Unlike Cytron *et al.*, we are unable to avoid calculating the recurrences iteratively. In practice, however, the computation of the sets frequently terminates in a few iterations.

[2]Because an object's alignment can change when it is referenced, both assignments and references modify an object.



Dominance Frontiers		
Block	DF()	RDF()
1		
2	5, 7	1
3	5, 7	1
4	7	2
5	7	2, 3
6	7	3
7		

(a) An example CFG with blocks that define or use the value of variable x. The dominance frontiers of the blocks are shown in the table.



(b) Merge and branch nodes required in blocks 2, 3, and 7.

(c) Adding branch nodes in blocks 2 and 3 induces a merge node in block 5.

Figure 6: The interaction between merge and branch nodes.

3.4 The renaming algorithm

The placement of merge and branch nodes is only the first half of the problem. The mentions of variables in each of these trivial merge and branch nodes must be replaced with new variable names. New names retain the original name ("V") as a root, but add a sequence number ("V_1", "V_2", *etc.*). The algorithm that follows is essentially the naming algorithm given by Cytron *et al.* [4], modified to handle branch nodes.

For each variable V a counter $C[V]$ gives its current sequence number in the vector C. Another vector, S, holds a stack of names for each variable. Functions *Push* and *Top* operate on an individual stack, $S(V)$, while *MarkStacks* and *PopToMarks* manipulate all of the stacks in S. Function *MarkStacks* pushes a special mark on each of stacks so that a matching call to *PopToMarks* pops all names pushed since the last call to *MarkStacks*. Each node has an ordered list of successors and predecessors. Two functions assign numbers to the members of these lists: function *WhichSucc*(*X,Y*) returns the position of *Y* in the list of node *X* if it is a successor of *X*, and function *WhichPred*(*X,Y*) returns the position of *Y* in the predecessor list of *X*.

Each statement *S* is an assignment with left- and right-hand sides LHS(*S*) and RHS(*S*). A copy of each original statement before renaming is also stored in oldLHS and oldRHS. Since an LHS may have more than one variable, the notation LHS(*S*)[*j*] is used to refer to the *j*'th variable in the list.

The renaming algorithm (Algorithm 3.4) works by keeping a stack of the current sequence number for each variable. Recall that each variable definition must be given a unique name. For each regular assignment, a new name is given for each variable on the LHS by incrementing the sequence number and appending it to the root. As this is done, the names are pushed on the stack. Visiting blocks in dominator tree order ensures that the names at the top of the stack reflect the most recent definition of each variable. Each reference is simply replaced with the top name on the appropriate stack.

Statements containing merge and branch nodes are handled slightly differently. A merge node has many variables of the same name on the RHS. There is no "current" name that can be used to replace these mentions. Instead, each predecessor of the block in which a merge occurs is responsible for the renaming of one of these mentions. Branch nodes placed at the bottom of a basic block provide a different new variable name for each successor. It is the successor's responsibility to look "backward" and to retrieve the correct name if it is preceded by a block containing a branch node.

The main algorithm begins by initializing global data structures and then calls the routine *Search*, which recursively visits the nodes of the dominator tree. Upon entering a block, if there are branch nodes in any of its immediate predecessors, then the appropriate names are extracted from the particular branch followed and pushed onto the stack.

For each regular statement in the block, references are replaced with the name on the top of the stack of the appropriate variable. Definitions receive new names by incrementing the sequence number of the variable.

Following this step, the algorithm handles the renaming of mentions in the LHS of branch statements and the RHS of merge statements in successors. Each successor of the block is visited in turn. For each successor, new names are generated for the LHS mentions of each branch node and are pushed on the stack. This step ensures that the stack reflects the appropriate variable names upon visiting the merge nodes of a particular successor in the next statement. Then, the successor is examined for any merge nodes. The RHS mentions are replaced with the current names as reflected in the tops of the stacks in S. After visiting a successor, all names pushed on the stacks are popped with the function *PopToMarks*, and the next successor is visited if there is one.

Finally, the children of the block (in the dominator tree) are visited. Any definitions made in the current block, or any that are still in effect from parents, are on the tops of the stacks and become the names that replace references.

Algorithm 3.4 would be simple to understand if it were true that upon entry to a block the current name on the tops of the stacks reflected the true current name for all references. But this is not always the case because a predecessor block with a branch node renames a variable but does not push it onto the stack. Hence the first part of the algorithm retrieves variable names from the branch nodes of predecessors and pushes them onto the stacks. However, because the blocks are visited in dominator order, it is not clear that a predecessor is necessarily visited before a successor attempts to retrieve new names from its branch statements.

There are some variable mentions, though, for which renaming status does not matter. Because merge nodes are placed at the top of a basic block, variables with a merge function are renamed immediately before any regular statements can reference them, and any name retrieved for such a variable would be immediately hidden by a new

Algorithm 1 (Renaming the occurrences of variable names in the ADG.)

```
for each variable V do
   C(V) = 0
   S(V) = empty
enddo
call Search(ENTER)

Search(X):
MarkStacks(S)
for each Y in pred(X) do                                   /* Retrieve predecessor branch names */
   j = WhichSucc(Y, X)
   for each branch node B in Y do
      V = oldLHS(B)[j]
      if (V ∉ {var | X has a merge function for var}) then
         Push(S(V), LHS(B)[j])                             /* Push the current name of j'th occurrence */
      endif
   enddo
enddo
for each statement A in X do                               /* Rename regular statements */
   if A is not a merge statement then
      replace each RHS var V with TOP(S(V))
   endif
   if A is not a branch statement then
      replace each LHS var V with PUSH(S(V), ++C(V))
   endif
enddo
k = 0
for each Y in Succ(X) do                                   /* Rename branch */
   k = k + 1
   MarkStacks(S)
   for each branch function B in X do
      V = RHS(B)
      LHS(B)[k] = PUSH(S(V), ++C(V))
   enddo
   j = WhichPred(Y, X)
   for each merge node M in Y do                           /* Fix merge nodes in successor */
      replace the j-th operand V in RHS(M) by Top(S(V))
   enddo
   PopToMarks(S)                                           /* Pop all Pushes done for branch nodes */
enddo
for each Y in Children(X) do
   SEARCH(Y)                                               /* Recursively search dominator tree children */
enddo
PopToMarks(S)                                              /* Forget context pushed by this block */
```

Program	Operation nodes	Transformer nodes	Control flow nodes
fig1	4 (25.0%)	6 (37.5%)	6 (37.5%)
fig2	2 (5.9%)	18 (52.9%)	14 (41.2%)
fig3	4 (22.2%)	6 (33.3%)	8 (44.4%)
dflux	277 (56.3%)	57 (11.6%)	158 (32.1%)
eflux	220 (91.3%)	0 (0.0%)	21 (8.7%)
shal	269 (60.4%)	63 (14.2%)	113 (25.4%)
erle	380 (57.1%)	126 (18.9%)	160 (24.0%)

Table 1: Distribution of nodes by type in the ADG representation of several programs.

name pushed on the stack.

This observation leads to a guarantee of the correctness of the renaming algorithm. The following lemma ensures that upon entry to a block only the names of variables of branch nodes for which there are no corresponding merge statements must be pushed on the stack.

Lemma 4 *Upon entering a block B, any variable V for which there is a predecessor with a branch node either (1) has already been named, or (2) is not named, but B has a merge node for V so that the current name for V is not needed.*

Proof: Consider any predecessor A of B. There are two cases. If $A \gg B$, then A is visited before B and any branch nodes are renamed before entering B. If $A \not\gg B$, then A could be visited before or after B. However, since A is a predecessor of B but does not dominate B, the dominance frontier of A contains B. Block B must therefore have a merge node for V corresponding to the definition of V in the branch statement at the bottom of A. □

3.5 Fanout node placement

Fanout nodes are added last to ensure that every object has exactly one definition and one use. After the preceding steps, if a variable V has a single definition but multiple uses, a fanout node is added in a line immediately following the definition of the variable in question. The input to the node is the variable V, and the rest of the references to V are renamed by incrementing the counter associated with V.

3.6 Size of the ADG

Table 1 shows the distribution of node types in the ADGs for several programs. Programs `fig1`, `fig2`, and `fig3` are the program frgaments shown in Figures 1–3. The routines `dflux` and `eflux` are two of the three most computation-intensive procedures of the `flo52` program from the Perfect Club benchmarks, and are two of the test programs used by Gupta [12]. `shal` and `erle` are part of the Fortran D compiler test suite [13]. `shal` is a benchmark weather prediction program originally written by Paul Swarztrauber at NCAR. It is a stencil computation that applies finite-difference methods to solve shallow-water equations. `erle` is a benchmark program written by Thomas Eidson at ICASE. It performs 3D tridiagonal solves using ADI integration.

The original Fortran 77 programs were converted to Fortran 90 array syntax using the CMAX translator on the Connection Machine CM-5. Additional hand optimization was performed in the case of the `dflux` routine. Transformer and control flow nodes were then added manually.

Considering the four fragments of real programs, we see that the majority of ADG nodes are operation nodes. All of the codes were structured, making the placement of control flow nodes simple. The predominant kind of control flow node is the fanout node. The routine `eflux` is a single basic block of array statements; its ADG therefore contains no transformer nodes, and all the control flow nodes are fanout nodes. Overall, the total number of ADG nodes is less than a factor of two larger than the number of program operations.

4 Using the ADG in alignment analysis

This section discusses the use of the ADG in alignment analysis. We first describe the constraints that nodes impose among the alignments, control weights, and iteration spaces of their ports. We then specialize the fully general model to the patterns of control flow and data access of greatest importance. Finally, we survey algorithms for determining the various components of array alignment.

4.1 Nodal relations and constraints

We now list the constraints on alignment (the matrix L and the vector f in equation (1)) and the relations on iteration spaces and control weights that hold at each type of node.

4.1.1 ADG nodes of the first kind

ADG nodes of the first kind correspond to program operations. Control weights and iteration spaces are the same at every port, but alignment constraints may be complicated.

Elementwise operation nodes An elementwise operation on congruent objects A_1 through A_k produces an object R of the same shape and size, so all ports of such a node have the same alignment.

$$L_{A_1} = \cdots = L_{A_k} = L_R$$

and

$$f_{A_1} = \cdots = f_{A_k} = f_R.$$

Array section nodes Let A be a d-dimensional object, and S an array section specifier, that is, a d-vector $(\sigma_1, \ldots, \sigma_d)$, where each σ_i is either a scalar ℓ_i or a triplet $\ell_i : h_i : s_i$. Array axes corresponding to positions where the section specifier is a scalar are projected away, while the axes where the specifier is a triplet form the axes of the array section $R \equiv A(S)$. Let the elements of S that are triplets be in positions $\lambda_1 < \cdots < \lambda_c$. Let e_i be a column vector of length d whose only nonzero entry is 1 at position i.

The axis alignment of R is inherited from the dimensions of A that are preserved (not projected away) by the sectioning operation. Strides are multiplied by the sectioning strides. The offset alignment of R is equal to the position of $A(\ell_1, \ldots, \ell_d)$:

$$L_R = L_A \cdot [s_{\lambda_1} e_{\lambda_1}, \ldots, s_{\lambda_c} e_{\lambda_c}]$$

and

$$f_R = g_A((\ell_1, \ldots, \ell_d)^T) = f_A + L_A(\ell_1, \ldots, \ell_d)^T$$

where $\cdot$ denotes matrix multiplication.

Array section assignment nodes An assignment to a section of an array, as in `A(1:100:2, 1:100:2) = B`, takes an input object A, a section specifier S, and a replacement object B conformable with $A(S)$, and producing a result object R that agrees with B on $A(S)$ and with A elsewhere. The result aligns with A, and the alignment of B must match that of $A(S)$, as defined in the preceding paragraph.

$$L_R = L_A, \;\; f_R = f_A$$

and

$$L_B = L_{A(S)}, \;\; f_B = f_{A(S)}.$$

Transposition nodes Let A be a d-dimensional object, and let ρ be a permutation of $(1,\ldots,d)$. The array object $R \equiv \rho A$ (produced by an ADG *transpose* node) is the array $R(\theta_{\rho_1},\ldots,\theta_{\rho_d}) = A(\theta_1,\ldots,\theta_d)$. (Fortran 90 uses the `reshape` and `transpose` intrinsics to perform general transposition.) The offset of the transposed array is unchanged, but its axes are a permutation of those of A:

$$L_R = L_A \cdot [e_{\rho_1},\ldots,e_{\rho_d}], \;\; f_R = f_A.$$

Reduction nodes Let A be a d-dimensional object. Then the program operation `sum(A, dim=k)` produces the $(d-1)$-dimensional object R by reducing along axis k of A. (The operation could be `prod`, `max`, *etc.* instead of `sum`.) Let n_k be the extent of A in axis k. Then R is aligned identically with A except that the template axis to which axis k of A was aligned is a space axis of R. The offset of R in this axis may be any of the positions occupied by A.

$$L_R = L_A \cdot [e_1,\ldots,e_{k-1},e_{k+1},\ldots,e_d]$$

and

$$f_R = f_A + \beta L_A e_k, \text{ where } 0 \le \beta < n_k.$$

Spread nodes Let A be a d-dimensional object. Then the program operation `spread(A, dim=k, ncopies=n)` produces a $(d+1)$-dimensional object R with the new axis in position k, and with extent n along that axis. The alignment constraints are the converse of those for reduction. The new axis of R aligns with an unused template axis, and the other axes of R inherit their alignments from A.

$$L_A = L_R \cdot [e_1,\ldots,e_{k-1},e_{k+1},\ldots,e_{d+1}].$$

In order to make the communication required to replicate A residual rather than intrinsic, we require the offset alignment of A (the input) in dimension k to be replicated. This condition sounds strange, but it correctly assigns the required communication to the input edge of the *spread* node. In this view, a *spread* node performs neither computation nor communication, but transforms a replicated object into a higher-dimensional non-replicated object. Thus,

$$f_A = f_R + f_r$$

where the vector f_r has one nonzero component, a triplet in the axis j spanned by the replicated dimension k:

$$f_r = (0,\ldots,0\!:\!\ell_{jk}(n-1):\ell_{jk},\ldots,0)^T.$$

4.1.2 ADG nodes of the second kind

These nodes express the effect of control flow. They have the same alignment at every port, but have more complicated constraints on their control weights and iteration spaces.

Merge nodes A merge node occurs when multiple definitions of a value converge. This happens on entry to a loop and as a result of conditional transfers. Merge nodes enforce identical alignment at their ports.

The iteration space of the out edge is the union of the iteration spaces of in edges. The expected number of activations of the out edge with LIVs ξ is just the sum of the expected number of activations with LIVs ξ of the inedges. Therefore, the control weight of the outedge is the sum of the control weights of the inedges. Let the iteration spaces of the inedges be $\mathcal{I}_1$ through $\mathcal{I}_m$, and the corresponding control weights be $c_1(\xi)$ through $c_m(\xi)$. Let the iteration space of the outedge be $\mathcal{I}_R$ and its control weight be $c_R(\xi)$. Extend the c_i for each input edge to $\mathcal{I}_r$ by defining $c_i(\xi)$ to be 0 for all $\xi \in \mathcal{I}_r - \mathcal{I}_i$. Then

$$\mathcal{I}_R = \bigcup_{i=1}^{m} \mathcal{I}_i$$

and

$$\forall \xi \in \mathcal{I}_R, \;\; c_R(\xi) = \sum_{i=1}^{m} c_i(\xi).$$

Branch nodes A branch node occurs when multiple mutually exclusive uses of a value diverge. Following the activation of the in edge, one of the out edges activates, depending on program control flow. Branch nodes enforce identical alignment at their ports.

The relations satisfied by iteration spaces and control weights of the incident edges are dual to those of merge nodes. Let the iteration spaces of the outedges be $\mathcal{I}_1$ through $\mathcal{I}_m$, and the corresponding control weights be $c_1(\xi)$ through $c_m(\xi)$. Let the iteration space of the inedge be $\mathcal{I}_A$ and its control weight be $c_A(\xi)$. Then

$$\mathcal{I}_A = \bigcup_{i=1}^{m} \mathcal{I}_i$$

and

$$\forall \xi \in \mathcal{I}_A, \ \ c_A(\xi) = \sum_{i=1}^{m} c_i(\xi).$$

Fanout nodes Fanout nodes have identical alignments, control weights, and iteration spaces at all ports.

4.1.3 ADG nodes of the third kind

Transformer nodes express the effect of mobility.

Transformer nodes Transformer nodes are of three types, corresponding to the introduction, removal, and update of a LIV. The first two types relate iterations at different nesting levels while the third type relates iterations at the same nesting level.

Transformer nodes of the first and second types are called *entry* and *exit* transformer nodes and have the form

$$(1, \xi_1, \ldots, \xi_{k-1} | 1, \xi_1, \ldots, \xi_{k-1}, \xi_k = v)$$

and

$$(1, \xi_1, \ldots, \xi_{k-1}, \xi_k = v | 1, \xi_1, \ldots, \xi_{k-1}),$$

corresponding to the introduction or removal of the LIV ξ_k in a loop nest. Let $\xi = (1, \xi_1, \ldots, \xi_{k-1})$. Let the alignment on the input ("ξ") port be $L_A\theta + f_A$, and let the alignment on the output port be $L_R\theta + f_R$. Let $\mathcal{I}_A$ be the iteration space of the input port and $\mathcal{I}_R$ be the iteration space of the output port. Then the alignment constraints are

$$\forall \xi \in \mathcal{I}_A, \ \ L_A(\xi) = L_R((\xi, v)^T)$$

and

$$\forall \xi \in \mathcal{I}_A, \ \ f_A(\xi) = f_R((\xi, v)^T).$$

The relation satisfied by the iteration spaces is

$$\mathcal{I}_R = \mathcal{I}_A \times \{v\},$$

where $\times$ denotes the Cartesian product. Thus, the $(1|1, 1)$ transformer node in Figure 1 constrains its input position (which does not depend on k) to equal its output position for $k = 1$. An offset alignment of $f_A = 1$ and $f_R = 2i - 1$ satisfies the node's constraints.

Transformer nodes of the third kind are called *loop-back* transformer nodes and have the form

$$(1, \xi_1, \ldots, \xi_k | 1, \xi_1, \ldots, \xi_k + s),$$

corresponding to a change in the value of the LIV ξ_k by the loop stride s. Let $\xi = (1, \xi_1, \ldots, \xi_k)$; define $\xi + s = (1, \xi_1, \ldots, \xi_k + s)$. Let the alignment on the input ("ξ") port be $L_A\theta + f_A$, and let the alignment on the output port be

$L_R\theta + f_R$. Let $\mathcal{I}_A$ be the iteration space of the input port and $\mathcal{I}_R$ be the iteration space of the output port. Then the alignment constraints are

$$\forall \xi \in \mathcal{I}_A, \;\; L_A(\xi) = L_R(\xi + s)$$

and

$$\forall \xi \in \mathcal{I}_A, \;\; f_A(\xi) = f_R(\xi + s).$$

The relation between the iteration spaces is

$$\widehat{\mathcal{I}} = \mathcal{I} + s(0, e_k)^T.$$

Consider one of the $(1, k|1, k+1)$ transformer nodes in Figure 1. An offset alignment $f_A = 2k + 3$ and $f_R = 2k + 1$ satisfies the node's alignment constraints. If the input iteration space is $\mathcal{I}_A = \{(1,1)^T, \ldots, (1, n-1)^T\}$, then the output iteration space is $\mathcal{I}_R = \{(1,2)^T, \ldots, (1,n)^T\}$.

4.2 Approximations and specializations in the model

The definition of the ADG in Section 2 assumed complete knowledge of control flow, and did not consider the complexity of the optimization problem. In this section, we discuss approximations to the model to address questions of practicality. The approximations are of two kinds: those that make it possible to compute the parameters of the model, and those that make the optimization problem tractable.

4.2.1 Control weights

Our model of control weights as a function of LIVs is formally correct but difficult to use in practice. We therefore approximate the control weight $c_{xy}(\xi)$ as an averaged control weight c'_{xy} that does not depend on ξ. We now relate this averaged control weight to the execution counts of the basic blocks of the program.

Assume that we have a control flow graph (CFG) of the program and an estimate of the branching probabilities of the program. (These probabilities can be estimated using heuristics, profile information, or user input.) Let p_{ij} be this estimate for edge (i, j) of the CFG, with $p_{ij} = 0$ if (i, j) is not an edge of the CFG. Let there be n basic blocks in the CFG, with ENTRY numbered 1 and EXIT numbered n. We first determine execution counts of the basic blocks by solving a linear system expressing conservation of control flow:

$$P^T u - u = e_1$$

where $u = (u_1, \ldots, u_n)^T$ is the vector of execution counts, and the e_1 on the right hand side forces the execution count of ENTRY to be 1. The (averaged) control weight c'_{xy} of the ADG edge (x, y) coming from a computation in basic block b is then $u_b/|\mathcal{I}_{xy}|$.

4.2.2 Mobile alignment

So far we have not constrained the form that mobile alignments may take. In principle, they could be arbitrary functions of the LIVs. We restrict mobile alignments to be affine functions of the LIVs. Thus, the alignment function for an object within a k-deep loop nest with LIVs $\xi_1, \ldots, \xi_k$ is of the form $a_0 + a_1\xi_1 + \cdots + a_k\xi_k$, where the coefficient vector $a = (a_0, \ldots, a_k)^T$ is what we must determine. We write this alignment succinctly in vector notation as $a^T\xi$. Both a and ξ are $(k+1)$-vectors. This reduces to the constant term a_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.

4.2.3 Replicated alignments

In Section 2, we introduced triplet offset positions to represent replication. In practice, we treat the replication component of offset separately from the scalar component. The alignment space for replication has two elements, called $\mathbf{R}$ (for replicated) and $\mathbf{N}$ (for non-replicated). The extent of replication for an $\mathbf{R}$ alignment is the entire template extent in that dimension. In this approximate model of replication, communication is required only when changing from a non-replicated alignment to a replicated one. Thus, the distance function is given by $d(\mathbf{N}, \mathbf{R}) = 1$ and $d(\mathbf{R}, \mathbf{R}) = d(\mathbf{R}, \mathbf{N}) = d(\mathbf{N}, \mathbf{N}) = 0$. We call the process of determining these restricted replicated alignments *replication labeling*.

4.2.4 Distance functions

In introducing the distance function in Section 2.5, we defined it to be the cost per element of changing from one alignment to another. Now consider the various kinds of such changes, and their communication costs on a distributed-memory machine. A change in axis or stride alignment requires unstructured communication. Such communication is hard to model accurately as it depends critically on the topological properties of the network (bisection bandwidth), the interactions among the messages (congestion), and the software overheads. Offset realignment can be performed using shift communication, which is usually substantially cheaper than unstructured communication. Replicating an object involves broadcasting or multicasting, which typically uses some kind of spanning tree. Such broadcast communication is likely to cost more than shift communication but less than unstructured communication.

We could conceivably construct a single distance function capturing all these various effects and their interactions, but this would almost certainly make the analysis intractable. We therefore split the determination of alignments into several phases based on the relative costs of the different kinds of communication, and introduce simpler distance functions for each phase.

We determine axis and stride alignments (or the matrix L of equation (1)) in one phase, using the *discrete metric* to model axis and stride realignment. This metric, in which $d(p, q) = 0$ if $p = q$ and $d(p, q) = 1$ otherwise, is a crude but reasonable approximation for the per-element cost of unstructured communication.

We determine scalar offset alignment in another phase, using the *grid metric* to model shift realignment. In this metric, alignments are the vectors f of equation (1), and $d(f, f')$ is the Manhattan distance between them, $d(f, f') = \sum_{i=1}^{t} |f_i - f'_i|$. Note that the distance between f and f' is the sum of the distances between their individual components. This property of the metric, called *separability*, allows us to solve the offset alignment problem independently for each axis [8].

We mention in passing that although we have developed algorithms to optimize latency-dominated communication (discrete metric) and distance-dominated communication (grid metric), optimizing communication containing both a startup term and a distance term appears to be much more difficult.

Finally, we use yet another phase to determine replicated offsets, using the alignments and distance function described in Section 4.2.3.

The ordering of these phases is as follows: we first perform axis and stride alignment, then replication labeling, and finally offset alignment.

The various kinds of communication interact with one another. For instance, shifting an object in addition to changing its axis or stride alignment does not increase the communication cost, since the shift can be incorporated into the unstructured communication needed for the change of axis or stride. We model such effects by ignoring certain edges. During replication labeling and offset alignment, we ignore edges carrying residual axis or stride realignment. Similarly, we ignore edges carrying replication communication during offset alignment.

4.3 Determining alignments using the ADG

We briefly describe our algorithms for determining alignment. Full descriptions of these algorithms are in companion papers [8, 14, 15, 16].

4.3.1 Axis and stride alignment

To determine axis and stride alignment, we minimize the communication cost $K(G, d, \pi)$ using the discrete metric as the distance function. The position of a port in this context is the matrix L of the alignment function. We have developed two separate algorithms for this problem: *compact dynamic programming* [8] and *the constraint graph method* [16].

Compact dynamic programming is based on the dynamic programming approach of Mace [17]. The "compact" in the name refers to the way we exploit properties of the distance function to simplify the computation of costs and to compactly represent the cost tables. The method is exact if the ADG is a tree, but in the general case (where the optimization problem is NP-complete) it is an approximation.

The constraint graph method is more suitable when the ADG has arbitrary structure. In this formulation, the objective is to find positions for each of the ports that maximizes the weight of the edges whose constraints are satisfied. The algorithm derives a *constraint graph* from the ADG, performs a series of contraction steps on this graph, and then finds a maximal subgraph that satisifes the nodal constraints of the ADG. Its strength lies in its non-local nature: it may reverse earlier decisions by removing previously added edges. Further, each step examines entire sets of edges and can discover interactions between constraints that are not handled adequately by strictly local approaches.

4.3.2 Offset alignment

For offset alignment, a position is the vector f of the alignment function, and the Manhattan metric is the distance function. We can solve independently for each component of the vector. For code where the positions do not depend on any LIVs, the constrained minimization can be solved exactly using integer programming [14, 15]. With mobile alignments, the residual communication cost can be approximated and this approximate cost minimized exactly using integer programming.

4.3.3 Replication labeling

For replication labeling, the position space has two positions for each template axis, called $\mathbf{R}$ (for replicated) and $\mathbf{N}$ (for non-replicated). The distance function, as given in Section 4.2.3, ia asymmetric and thus is not a metric. The sources of replication are *spread* operations, certain vector-valued subscripts, and read-only objects with mobile offset alignment. As in the offset alignment case, each axis can be treated independently.

A minimum-cost replication labeling can be computed efficiently using network flow [14].

5 Comparison with other work

In this section, we compare the ADG with SSA form and with the *preference graph*, another representation that has been used for automatic determination of alignments.

5.1 SSA form

While the ADG is based on SSA form, it has a number of extra features. All of the differences stem from the fact that the ADG representation was designed to manipulate positions of objects, while SSA form was designed to manipulate values. In fact, an SSA form based on a nonstandard position semantics (see Section 2.1) would probably look exactly like the ADG.

The ADG annotates the ports and edges with data weights, control weights, and iteration spaces, none of which are present in SSA form. However, the substantive difference between the two representations lies in the nodes. Every ϕ-function of SSA corresponds to a merge node in ADG, but certain merge nodes (*e.g.*, for read-only objects within a loop) do not correspond to any ϕ-functions in SSA. Similarly, fanout and branch nodes have no analog in SSA form.

The fanout and branch nodes of the ADG resemble similar nodes in the Program Dependence Web representation developed by Ballance *et al.* [18]. Their μ and η functions are also similar to ADG transformer nodes. However, the motivations behind them are very different.

5.2 The preference graph

Another representation that has been used in alignment analysis is the *preference graph*, which has several variants [12, 19, 20, 21]. The preference graph is an undirected, weighted graph constructed from the reference patterns in the program. The nodes of the preference graph correspond to dimensions of array occurrences in the program, the edges reflect beneficial alignment relations, and the weights reflect the relative importance of the edges. The edges of the preference graph encode axis alignment, while additional node attributes are used to encode stride and offset alignment.

The preference graph has two kinds of edges, corresponding to the two sources of alignment decisions. Each statement considered in isolation provides some relations among nodes that avoid residual communication in computing the given statement. The edges corresponding to these relations are called *conformance preference edges*. A conformance preference edge between two array occurrences indicates that if they are not aligned, residual communication will be needed to align them in preparation for the operation. Conformance preferences are similar to node constraints in the ADG.

The second kind of alignment preference comes from relating definitions and uses of the same array variable. The edges corresponding to these relations are called *identity preference edges*. An identity preference edge between two array occurrences indicates that if they are not positioned identically, residual communication will be needed to make the values from the definition available at the use. Identity preferences are analogous to ADG edges.

Alignment decisions are made by contracting graph edges. At each step, an edge is chosen, and if the two nodes connected by the edge satisfy certain conditions, the edge is contracted and one of the nodes is merged into the other. When an edge is contracted, we say that the alignment preference it carries has been *honored*. The contraction process stops when no more edges can be contracted. An acyclic preference graph can be contracted to a single node, guaranteeing a communication-free implementation. If, however, there are cycles in the preference graph, some alignment preferences may remain unhonored and will result in residual communication. In this case, we must decide which preferences to honor and which to break. There are two principal variants of this general framework.

The Knobe-Lukas-Steele algorithm Knobe, Lukas, and Steele [19] call the nodes of the preference graph *cells*. They introduce an additional attribute of cells (called *independence anti-preference*) to model the constraint that a dimension occurrence has sufficient parallelism and should not be serialized.

An alignment conflict can only occur within a cycle of the preference graph. We therefore need to locate a cycle, determine whether it causes a conflict, and resolve the conflict if it exists. Actually, we only need to examine and resolve conflicts in a set of fundamental cycles of the preference graph, since if such a cycle basis is conflict-free, then all cycles in the graph are conflict-free. A cycle basis can be easily determined from a spanning tree of the graph: each non-tree edge determines a unique fundamental cycle of the graph with respect to the spanning tree. This ignores the fact that the preference graph is weighted. Given a choice between honoring two preferences, we would choose to honor the preference with higher weight. The Knobe-Lukas-Steele algorithm uses the nesting depth of an edge as its weight and constructs a maximum-weight spanning tree.

An unhonored identity preference implies that an object will live in distinct locations in different parts of the program. An unhonored conformance preference implies that not all of the operations in a statement are necessarily performed in the location of the "owner" of the LHS. In other words, breaking a conformance preference is the mechanism for improving upon the "owner-computes rule" [22].

The Li-Chen algorithm Li and Chen [20] use the preference graph model to determine axis alignment. Their algorithm was developed in the context of the Crystal language, which allows more general reference patterns than we have considered, for instance, a pattern like $A(i, j) = B(i, i)$. This indicates conformance preferences between the first axis of A and both axes of B; such sets of preferences that are generated from the same reference pattern and are incident upon a common node are said to be *competing*.

Li and Chen call the preference graph the *Component Affinity Graph* (CAG), and call preferences *affinities*. Competing edges have infinitesimal weight ϵ, while noncompeting edges have unit weight. The axis alignment problem is framed as the following graph partitioning problem:

> Partition the node set of the CAG into n disjoint subsets $V_1, \ldots, V_n$ (where n is the maximum dimensionality of any array) with the restriction that no two axes of the same array may be in the same partition, so as to minimize the sum of the weights of edges connecting nodes in different subsets of the partition.

The idea here is to align array dimensions in the same subset of the partition, with edges between nodes in different partitions corresponding to residual communication. Hence we minimize the sum of the weights of such edges.

This graph partitioning problem is NP-complete [20]. Li and Chen solve it heuristically by finding maximum-weight matchings of a sequence of bipartite subgraphs of the CAG.

Discussion Conformance preferences, while similar to the node constraints of our approach, are not clearly related to the data flow and the intermediate results of the computation. Intermediate results have no explicit representation in the conformance graph model. They could, however, be made explicit by preprocessing the source.

The preference graph model distinguishes between objects at different levels of a loop nest, but does not consider the size of an object, or the shift distance, in computing the cost of moving it. In fact, it can be thought of as using the unweighted discrete metric to model all communication cost.

Knobe *et al.* [19] discuss handling control flow by introducing alignments for arrays at merge points in the program. Our use of static single assignment form provides a sound theoretical basis for the placement of merge nodes.

Finally, to the best of our knowledge, the preference graph method has not been used to determine when objects should be replicated.

6 Conclusions and further work

We have motivated and described the alignment-distribution graph, a representation of data-parallel programs that provides a mechanism for explicitly representing and optimizing residual communication cost. The representation is based on a separation of variables and values, extended from the static single assignment form of programs. We have implemented the algorithms mentioned in Section 4.3 and are experimenting with them to test the validity of this approach.

In the distribution analysis method that we foresee in the ADG framework, the template has a distribution at each node of the ADG; the alignment of an object to the template and the distribution of the template jointly define the mapping of the objects at that node on to the processors. The distributions are chosen to minimize an execution time model that accounts for redistribution cost (again associated with edges of the ADG) and computation cost, associated with nodes of the ADG and dependent on the distribution as well.

Ongoing and future work in this area include developing algorithms for distribution analysis using the ADG framework, understanding the interactions between the alignment and distribution phases, developing interprocedural optimization techniques in this framework, and allowing for the possibility of skew alignments [23].

References

[1] Geoffrey C. Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report Rice COMP TR90-141, Department of Computer Science, Rice University, Houston, TX, December 1990.

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

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

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[5] Manish Gupta and Edith Schonberg. A framework for exploiting data availability to optimize communication. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 216–233. Springer-Verlag, 1994.

[6] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[7] Pavel Winter. Steiner problem in networks: A survey. *Networks*, 17:129–167, 1987.

[8] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Shang-Hua 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, October 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.

[9] Vincent Bouchitté, Pierre Boulet, Alain Darte, and Yves Robert. Evaluating array expressions on massively parallel machines with communication/computation overlap. ??, 1994.

[10] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1986.

[11] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 78–89, Albuquerque, NM, June 1993.

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

[13] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing'93*, pages 338–350, Portland, OR, November 1993.

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

[15] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Shang-Hua 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, January 1993. Also available as RIACS Technical Report 92.18 and Xerox PARC Technical Report CSL-92-13.

[16] Thomas J. Sheffler, Robert Schreiber, John R. Gilbert, and Siddhartha Chatterjee. Aligning parallel arrays to reduce communication. Submitted to Supercomputing'94, Washington DC, November 1994.

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

[18] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 257–271, White Plains, NY, June 1990.

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

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

[21] Skef 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.

[22] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler support for machine-independent parallel programming in Fortran D. Technical Report Rice COMP TR90-149, Department of Computer Science, Rice University, Houston, TX, February 1991.

[23] Jennifer M. Anderson and Monica 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.