Array Distribution in Data-Parallel Programs

Siddhartha Chatterjee * John Gilbert † Robert Schreiber * Thomas Sheffler *

Abstract

We consider distribution at compile time of the array data in a distributed-memory implementation of a data-parallel program written in a language like Fortran 90. We allow dynamic redistribution of data and define a heuristic algorithmic framework that chooses distribution parameters to minimize an estimate of program completion time. We represent the program as an alignment-distribution graph. We propose a divide-and-conquer algorithm for distribution that initially assigns a common distribution to each node of the graph and successively refines this assignment, taking computation, realignment, and redistribution costs into account. We explain how to estimate the effect of distribution on computation cost and how to choose a candidate set of distributions. We present the results of an implementation of our algorithms on several test problems.

1 Introduction

One of the major decisions in compiling data-parallel programs for distributed-memory parallel computers is the mapping of data and computation to the multiple processors of the machine. A good mapping minimizes program completion time by balancing the opposing needs of parallelism and communication: spreading the data and work over many processors increases available parallelism, but also increases communication time.

Most compilation systems (e.g., Fortran D [10] and High Performance Fortran [9]) divide the data mapping problem into two phases: *alignment*, in which the relative positions of arrays are determined within a Cartesian grid called a *template*, and *distribution*, in which the template is partitioned and mapped to a processor grid. We have dealt with the alignment problem previously [2, 4, 5]. This paper focuses on the distribution problem.

1.1 Distribution

The distribution of a template specifies for each of its dimensions the number of processors p it is spread across and the block size k used in the distribution. Template cell i is located at processor $(i \text{ div } k) \bmod p$. The distribution of a multidimensional template is the tensor product of the distributions of each of its dimensions. The HPF declarations

```
CHPF$ TEMPLATE T(100,200)
CHPF$ PROCESSORS P(4,8)
CHPF$ DISTRIBUTE T(CYCLIC(1),CYCLIC(10)) ONTO P
```

specify the distribution of a template to a 4×8 processor grid with block sizes 1 and 10 in the two dimensions. The mapping of a data array is determined by the composition of its alignment to the template and the distribution of the template to the processor grid. Note that distributing a template dimension to one processor is equivalent to making that dimension resident in memory.

*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).

The distribution problem is to determine template distribution parameters that minimize the completion time of the program. There are two variants of the problem, depending on whether or not we allow dynamic redistribution of templates and data objects.

Problem 1 (Static Distribution). Given a program and a number of processors, determine a common distribution for all array objects that minimizes the completion time.

Problem 2 (Dynamic Distribution). Given a program and a number of processors, determine the distribution at each definition and each use of every array object so that the completion time is minimized.

1.2 Problem formulation

In our system, program data flow is represented as a directed, edge-weighted graph called the Alignment-Distribution Graph (ADG) [3]. It consists of ports, nodes, and edges. Ports represent array objects manipulated by the program, nodes represent program operations, and edges connect definitions of array objects to their uses. See Section 6 for examples. Alignment and distribution are attributes of ports (array objects). Each port has an alignment to a template, and a distribution of the template to a processor grid. With one exception, which is noted in Section 5.3, we treat the ADG in this paper as an *undirected* graph.

Nodes constrain the alignments and distributions of their constituent ports. Alignment constraints have been discussed elsewhere [3]. The distribution constraint at a node is particularly simple: all of its ports must be aligned to the same template, with the same distribution parameters. It is therefore sensible to speak of "the distribution at a node".

The ADG makes communication explicit. Communication occurs when the alignment or distribution is different at the end points of an edge. ADG edges connect the definition of an object to its uses. *Realignment* occurs along an edge when the alignment of the object at the tail of the edge is different from its alignment at the head. *Redistribution* occurs along an edge when the distribution of the template at the tail is different from the distribution at the head. Alignments are chosen in a previous compilation phase, and are considered fixed here. The interaction of alignments and distributions is quite important in determining good distributions. We cover this topic in Section 5.3.

Let δ be a mapping from ADG nodes to distributions. The computation time T_{comp} of an ADG node u is a function of its computation, the sizes of the array objects it computes, the alignments of its ports, and the distribution $\delta(u)$ at the node. The cost of any realignment that occurs on out-edges of an ADG node can be accounted for at the node, as we show in Section 5.3. Thus, we include a term T_{node}, called a *node cost*, that accounts for computation and realignment, in our model of completion time. All our techniques assume that the dependence of node cost T_{node} on distribution can be made explicit. Section 5.3 discusses techniques for doing this.

The redistribution time T_{redist} of an edge (u, v) depends on the weight w_{uv} of the edge, and the distributions $\delta(u)$ and $\delta(v)$ of nodes u and v. Specifically, we assume that the cost of redistribution along edge (u, v) is the product of three terms: a machine parameter ρ that gives the cost per data item of all-to-all personalized communication; the edge data weight w_{uv} (the total data volume carried by the ADG edge); and the *discrete distance* between the distributions $\delta(u)$ and $\delta(v)$, defined to be 0 if $\delta(u)$ equals $\delta(v)$ and 1 if they differ. (Section 5.1 provides some empirical evidence justifying this model.)

Our model of completion time of the ADG $G = (V, E)$ with alignment map α and distribution map δ is

$$T_{finish}(G, \alpha, \delta) = \sum_{v \in V} T_{node}(v, \alpha, \delta(v)) + \sum_{(u,v) \in E} T_{redist}(w_{uv}, \delta(u), \delta(v)). \tag{1}$$

In the sequel whenever we refer to a "cost" we mean either the node cost T_{node} of an ADG node, or the redistribution time T_{redist} of an edge. To simplify notation, we shall suppress the α in T_{finish} and T_{node} in the sequel, since we assume that α has already been determined, and is not subject to further change.

Given the ADG G and the alignment mapping α, the goal is to determine the distribution mapping δ that minimizes the completion time $T_{finish}(G, \alpha, \delta)$. We shall call a distribution mapping *optimal* if it minimizes T_{finish}, given G and α.

We collect here some definitions and facts from graph theory and linear algebra that we will use in the paper. Let $S \subseteq V$ be a subset of the nodes of the ADG. We denote by $G(S)$ the ADG subgraph induced by S. We assume that the set of candidate distributions D has been determined. (We show in Section 5.2

how to do this.) A distribution mapping of S is a function $\delta : S \mapsto D$. It is *static* if $\delta(s)$ is the same for all elements s of S, otherwise it is *dynamic*. If δ is defined on S, we call the set S static or dynamic depending on whether the restriction of δ to S is static or dynamic.

Let $S \subseteq V$ be given. Define $T_{node}(S, d) = \sum_{v \in S} T_{node}(v, d)$. We denote by $T_{opt}(S)$ the minimum over all $d \in D$ of $T_{node}(S, d)$, and call this the *static cost* of $G(S)$. It is the minimum completion time of $G(S)$, given that all nodes in S are constrained to have the same distribution, since there is no redistribution when S is static. Define a *best static distribution* of the subgraph $G(S)$ as some distribution d that achieves the static cost of $G(S)$.

A partition of S is a set of disjoint subsets of S whose union equals S. We shall primarily deal with partitions of the nodes of an ADG subgraph $G(S)$ consisting of two subsets, S_1 and S_2. For such a two-way partition, define $\text{cut}(S_1, S_2) \equiv \{ (u, v) \in E \mid u \in S_1,\ v \in S_2 \}$, and a redistribution cost $\text{cut-cost}(S_1, S_2) \equiv \rho \sum_{(u,v) \in \text{cut}(S_1,S_2)} w_{uv}$.

Let X be any undirected, edge-weighted, n vertex graph with vertices V and edges E. The *Laplacian matrix* of X is the symmetric $n \times n$ matrix defined by

$$L(X)_{ij} = \begin{cases} -w_{ij}, & i \neq j \text{ and } (i, j) \in E \\ \sum_{k \neq i} w_{ik}, & i = j \\ 0, & \text{otherwise.} \end{cases}$$

If we define the weighted adjacency matrix $A(X)$ in the usual manner and the weighted degree matrix $D(X) = \text{diag}(d_i)$, where d_i is the sum of the weights of edges incident on vertex i, then $L(X) = D(X) - A(X)$. A Laplacian matrix has nonnegative eigenvalues, one of which is always zero. If the graph X is connected, then the eigenvalue zero is simple, and the corresponding eigenvector is the vector $e = (1, \ldots, 1)^T$.

Let x be a real n-vector. We shall make use of p-norms of vectors, defined by

$$\|x\|_p = (\sum_i |x_i|^p)^{1/p},$$

and the ∞-norm, defined by

$$\|x\|_\infty = \max_i |x_i|.$$

Note that the 2-norm is the usual euclidean norm in real n-space, $\mathbb{R}^n$.

1.3 Previous work

Most of the previous work in this area has concentrated on the static version or on simplified dynamic versions of the problem. Wholey [16] uses a hill-climbing procedure to determine the distributions for named variables. (A named variable corresponds, in our formulation, to a subset of the nodes of the ADG. Thus, Wholey considers a restricted version of the dynamic problem.) Gupta [8] uses heuristic methods to determine the distribution parameters. He analyzes communication patterns to determine whether block or cyclic distributions are preferable. He then uses an affinity graph framework to determine block sizes. Finally, he allows at most two array dimensions to be distributed across the processors, and determines the proper aspect ratio by exhaustive enumeration.

Kremer [12] shows that dynamic distribution is NP-complete. Bixby et al. [1] and Kremer et al. [13] present a partial, heuristic solution method. They assume that the user provides a decomposition of the program into phases, which are program fragments that are executed without changing distribution. Dynamic redistribution therefore only occurs between phases. Their system chooses the distribution of each phase. They solve this restricted problem by reducing it to 0-1 integer programming.

The techniques we present here, in contrast, do not require the user to provide any decomposition of the program. We propose, instead, a fast algorithm that first determines node subsets to be given the same distribution and then determines their distributions.

1.4 Organization of the paper

The remainder of the paper is organized as follows. Sections 2 through 4 discuss our algorithm for determining distributions. Section 5 explains details of cost modeling. Section 6 shows experimental results from our implementation of the algorithm. Finally, Section 7 presents conclusions and future work.

2 The distribution algorithm

The dynamic distribution problem is to choose a distribution mapping that minimizes completion time T_{finish} with given ADG and alignment mapping. Our formulation of the distribution problem resembles graph partitioning; but unlike classical graph partitioning problems, there is no intrinsic balance criterion in our problem formulation. Moreover, the tension between T_{node} and T_{redist} makes this problem interesting. Kremer [12] proved that dynamic distribution NP-complete. We therefore seek good heuristics rather than exact algorithms for the problem. This section gives an overview of our algorithm; the next three sections discuss details.

Assume that we have chosen a set $D = \{\delta_1, \ldots, \delta_d\}$ of candidate distributions. (Section 5.2 discusses how we do this.)

Algorithm 1 uses divide-and-conquer to find an approximately optimal solution to the dynamic distribution problem. If it determines that the whole ADG should have a static distribution, then it chooses a best static distribution of G. Otherwise it partitions the ADG into two subgraphs and recursively determines dynamic distributions for each of them. The conquer step which follows chooses the better of two alternatives: either the union of these two dynamic subset distributions, or the best possible static distribution for the whole ADG. Thus, rather than requiring user intervention, Algorithm 1 automatically finds sets of nodes that it constrains to share a common distribution. It then determines distributions for these sets. The key to the algorithm is the partitioner, which must choose a partition that groups strongly related nodes in the same subset.

Algorithm 1 (Top-down partitioning of ADG for distribution analysis.)
Input: S, a set of ADG nodes; D, a set of candidate distributions.
Output: δ, a distribution mapping from S to D; ct, the completion time of set S.

1 $sc \leftarrow$ *static-cost*$(G(S))$
2 $sd \leftarrow$ *best-static-distribution*$(G(S))$
3 **if** *termination condition is reached* **then**
4     $ct \leftarrow sc$
5     $\delta(v) \leftarrow sd$ **for each** v **in** S
6 **else**
7     $[S_1, S_2] \leftarrow$ *partition*(S)
8     $[\delta_1, ct_1] \leftarrow$ *Algorithm 1*(S_1, D)
9     $[\delta_2, ct_2] \leftarrow$ *Algorithm 1*(S_2, D)
10   *dynamic-cut* $\leftarrow \{(u, v) \in E \mid u \in S_1, v \in S_2, \delta_1(u) \neq \delta_2(v)\}$
11   *dynamic-cut-cost* $\leftarrow \rho \sum_{(u,v) \in \textit{dynamic-cut}} w_{uv}$
12     $\kappa_D \leftarrow ct_1 + ct_2 +$ *dynamic-cut-cost*
13   **if** $\kappa_D < sc$ **then**
14         $ct \leftarrow \kappa_D$
15         $\delta \leftarrow \delta_1 \cup \delta_2$
16   **else**
17         $ct \leftarrow sc$
18         $\delta(v) \leftarrow sd$ **for each** v **in** S
19   **endif**
20 **endif**
21 **return** $[\delta, ct]$

The divide phase calls a partitioning routine on line 7 that returns a partition of the node set S into S_1 and S_2. The conquer phase calls Algorithm 1 recursively on the subsets S_1 and S_2 and determines κ_D, the completion time achieved using the given partition, with dynamic distribution of each subset (line 12). If this is less than the static cost of $G(S)$, we define the distribution mapping as the (disjoint) union of the mapping returned by the recursive calls. Otherwise, each node of S is mapped to a best static distribution of $G(S)$.

Section 3 discusses partitioning strategies, and Section 4 discusses the termination condition.

One might also employ "bottom-up" clustering procedures for finding static subgraphs of G. Such a procedure would begin with single node subgraphs and merge them. We do not, however, have any experience with such procedures.

We show in Section 3 that the costs of our implementations of the function *partition* for an n-vertex subset S are $O(n^3)$. Hence, Algorithm 1 has a worst-case complexity of $O(n^4)$. If the partitions we choose turn out to be well-balanced, however, then its complexity can be as small as $O(n^3)$.

3 Partitioning

A core routine of Algorithm 1 is the partitioner, which takes a subset of ADG nodes S and partitions it into two subsets S_1 and S_2. Algorithm 1 then independently computes candidate distribution mappings for the induced subgraphs. Our key partitioning heuristic is the following: choose a partition that minimizes $\kappa_S(S_1, S_2)$, which we define to be the sum of the static costs of S_1 and S_2 and cut-cost(S_1, S_2). (The subscript S stands for "static".)

In this section we present two algorithms of polynomial complexity for approximately solving this problem. Both encode the problem as the minimization of a real-valued function of the vertices of the unit n-dimensional cube, and both embed this minimization in an easier continuous problem.

The approximately optimal partitions that these two algorithms provide could be improved further by a subsequent improvement procedure of Kernighan-Lin type, in which nodes are moved from one subset to the other singly, in order to further reduce κ_S. We do not yet know its complexity, or whether such postprocessing is worthwhile.

3.1 The partitioning problem as nonlinear multidimensional optimization

In recasting the partitioning problem in matrix terms, we use techniques seen in spectral graph partitioning methods [14].

Let S be an n-vertex subset of ADG nodes. Assume we are given a partition of S into two disjoint subsets S_1 and S_2. Let x be an n-vector with elements $+1$ and -1 encoding this partition; nodes corresponding to the $+1$ elements of x belong to S_1, while those corresponding to the -1 elements belong to S_2. Let e be the n-vector of all 1's.

As above, $D = \{\delta_1, \ldots, \delta_d\}$ is a given set of candidate distributions. Construct the $d \times n$ node-cost matrix C, such that $C_{ij} = T_{node}(j, \delta_i)$ is the node cost of node j with distribution δ_i.

We now formulate the various costs in terms of matrix expressions. The sum of the weights of the edges crossing the cut is given by $\frac{1}{4}\sum_{(u,v)\in E} w_{uv}(x_u - x_v)^2$, since the term $(x_u - x_v)^2$ contributes zero if x_u and x_v have the same sign and 4 if they have different signs. To write this as a matrix expression, we simplify as follows:

$$\begin{aligned}
\sum_{(u,v)\in E} w_{uv}(x_u - x_v)^2 &= \sum_{(u,v)\in E} w_{uv}(x_u^2 + x_v^2) - \sum_{(u,v)\in E} w_{uv} 2x_u x_v \\
&= \sum_{(u,v)\in E} 2w_{uv} - \sum_u \sum_v x_u A_{uv} x_v \\
&= \sum_u x_u^2 d_u - \sum_u x_u \sum_v A_{uv} x_v \\
&= x^T D x - x^T A x \\
&= x^T L x.
\end{aligned}$$

The redistribution cost of edges crossing the cut is therefore $\frac{1}{4}\rho x^T L x$. To get the static costs of S_1 and S_2, we need to extract the nodes belonging to each set. Given that the elements of x are $+1$ and -1, we see that the corresponding elements of the vector $\frac{1}{2}(e + x)$ are 1 and 0, while those of the vector $\frac{1}{2}(e - x)$ are 0 and 1. Element i of the matrix-vector product $\frac{1}{2}C(e + x)$ gives the cost of the $+1$ partition if each node in that partition has distribution δ_i. The static cost of S_1 is therefore $\frac{1}{2}\min_i(C(e+x))_i$, while that of S_2 is $\frac{1}{2}\min_i(C(e-x))_i$. We therefore seek to minimize

$$\frac{1}{4}\rho x^T Lx + \frac{1}{2}\min_i(C(e+x))_i + \frac{1}{2}\min_i(C(e-x))_i \tag{2}$$

over the set of vectors with elements ± 1, not all elements the same. This is the set of vertices of the n-dimensional unit hypercube, without the two elements e and $-e$ whose elements are all of the same sign. We denote this set by H_n.

Rewrite the matrix $C = B - M$, where B is a matrix whose entries are a constant b, larger than $\max_{ij} C_{ij}$; M is the "savings" matrix.

Note that all elements of the products $M(e+x)$ and $M(e-x)$ are nonnegative, as the elements of M are positive and the elements of $(e+x)$ and $(e-x)$ are nonnegative. Hence,

$$\begin{aligned}
\min_i(Cy)_i &= \min_i((B-M)y)_i \\
&= \min_i(b\sum_j y_j - (My)_i) \\
&= b\sum_i y_i - \max_i(My)_i \\
&= b\sum_i y_i - ||My||_\infty .
\end{aligned}$$

The following constrained minimization problem is therefore equivalent to equation (2):

$$\min_{x\in H_n} \frac{1}{4}\rho x^T Lx - \frac{1}{2}||M(e+x)||_\infty - \frac{1}{2}||M(e-x)||_\infty . \tag{3}$$

Note the two kinds of terms in the cost function. The redistribution term $\frac{1}{4}x^T Lx$ is sensitive only to the edges of the ADG. The other two node-cost terms consider only the node characteristics. To see how these various terms affect the minimization, consider the ladder graph shown in Figure 1. Assume that there are three candidate distributions, that $\rho = 1$, and that the node cost matrix C has the following form:

$$C = \begin{bmatrix} 100 & 100 & 20 & 100 & 1 & 1 \\ 100 & 100 & 20 & 100 & 100 & 100 \\ 1 & 1 & 1 & 1 & 100 & 100 \end{bmatrix} .$$

Each column of C gives the cost of a single node in each of the three candidate distributions.

Minimizing just the communication term gives the partition vector $x = [1, -1, 1, -1, 1, -1]^T$, with completion time $= 3 + 102 + 102 = 207$. Minimizing only the node terms gives the partition vector $x = [1, 1, 1, 1, -1, -1]^T$, with completion time $= 200 + 4 + 2 = 206$. Minimizing all three terms together gives the partition vector $x = [1, 1, -1, 1, -1, -1]^T$, with completion time $= 121 + 3 + 22 = 146$. This demonstrates the insufficiency of minimizing the node or communication terms individually.

The minimization problem (3) is combinatorial, and may be as hard as the original distribution problem! Our heuristic approach is to first change the search space to a convex, closed, bounded region of $\mathbb{R}^n$, and then to replace the objective function by a differentiable approximation. We present two algorithms of this type for approximately solving (3).

3.2 Algorithm NL

The first algorithm (called NL for nonlinear) uses techniques from constrained nonlinear optimization to solve (3). We first change the problem to a continuous version, replacing the ∞-norm by a $2p$-norm for sufficiently large integer p, and minimizing over the surface of the n-dimensional unit $2p$-norm ball rather than over the vertices of the n-dimensional unit hypercube.[1] We return to the discrete domain by taking the

[1] The definition of the p-norm of a vector involves the absolute values of its components. An even-integer p-norm makes the absolute values unnecessary and is smooth at $x = 0$.

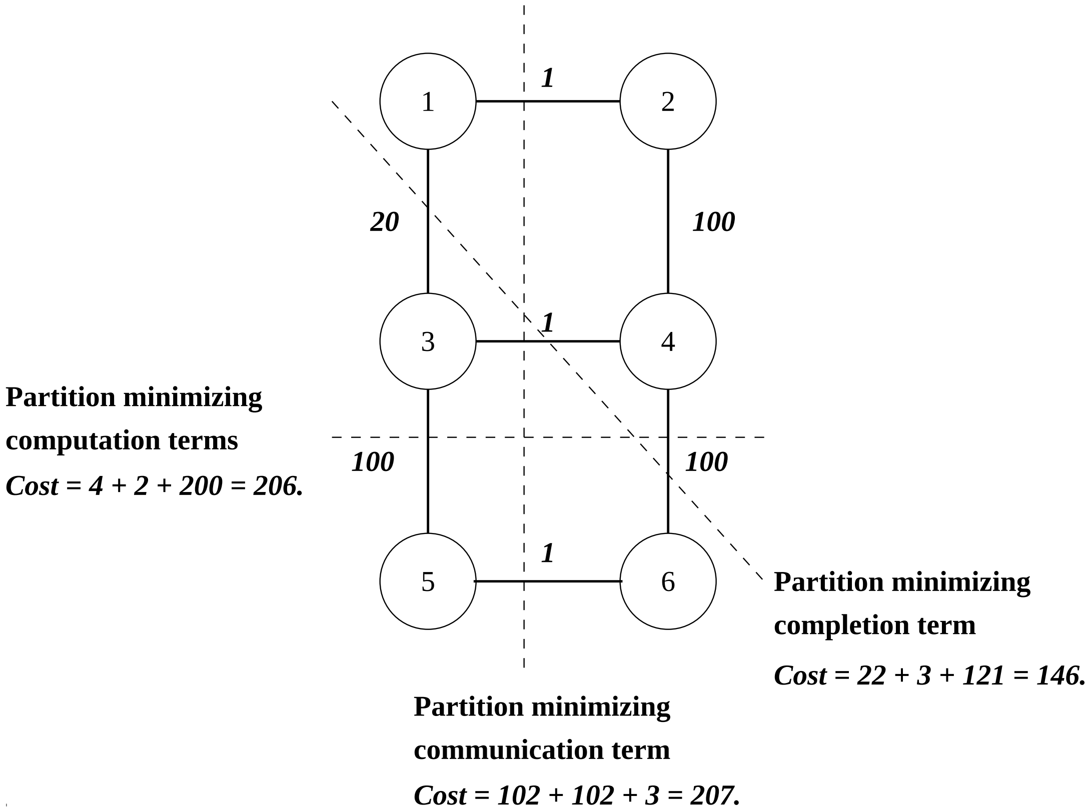

Figure 1: Ladder graph illustrating the roles of computation and communication in partitioning.

sign of the elements of the solution of the continuous problem. We need to exclude the positive and negative orthants, since if all elements of x have the same sign, no partition is produced. A simple constraint, which we use, is to require that x be orthogonal to e. Let $U_p = \{x \in \mathbb{R}^n \mid e^T x = 0, \quad \|x\|_p = 1\}$. The problem thus becomes

$$\min_{x \in U_{2p}} \frac{1}{4}\rho x^T L x - \frac{1}{2}\|M(e+x)\|_{2p} - \frac{1}{2}\|M(e-x)\|_{2p}. \tag{4}$$

Nonlinear optimization problem (4) can be solved by standard iterative methods like successive quadratic programming [6]. While the complexity of each iterative step is independent of p, convergence depends on p. In practice, choosing $p = 2$ appears to be adequate.

The cost of partitioner NL depends on p, on the particular minimization procedure, and on n; the dependence on n is $O(n^3)$ for standard optimization procedures.

3.3 Algorithm CC

In this section we describe an approximate solution technique that may be significantly less costly than Algorithm NL. The idea in this algorithm (called CC for convex combinations) is to solve approximately the communication-only problem and the node-only problem in the continuous domain and to then search for the minimizer of equation (3) among the convex combinations of the two extremal vectors.

The vector that minimizes the communication term $x^T L x$ among vectors of unit 2-norm is e (since $e^T L e = 0$); among vectors of unit 2-norm orthogonal to e, its minimizer is the eigenvector of L corresponding to the smallest nonzero eigenvalue (called the Fiedler vector). We rescale the Fiedler vector to have unit infinity norm, and call it x_L. (Note that finding a single eigenvector is somewhat cheaper than finding all of them.)

As for the node term, we require a vector x such that $M(x+e)$ and $M(x-e)$ are simultaneously large.[2] Note that M is elementwise positive, so that e maximizes the infinity norm of Mz among vectors z with unit infinity norm. We therefore expect that e will be far from orthogonal to the right singular vector x_1 of M corresponding to the largest singular value. (This is the vector z of unit 2-norm that maximizes the 2-norm of Mz.) By the Perron-Frobenius theorem [15], x_1 must have positive elements. The second right singular vector of M maximizes the norm of Mz among unit vectors z orthogonal to x_1. Therefore, if we choose x of infinity norm one in the direction of x_2, then both $x+e$ and $x-e$ will inherit the large component that e has in the direction x_1, making both $M(x+e)$ and $M(x-e)$ large. Call this vector x_M.

We now search along the line between x_L and x_M for the minimizer, that is, we seek λ in $[0,1]$ such that the vector $x = \lambda x_L + (1-\lambda)x_M$ minimizes the objective function in (3). Since both x_L and x_M are only determined up to sign, we replace x_M by $-x_M$ if necessary to make the two vectors agree in sign in at least one element before beginning this line search. (Consider, otherwise, the effect of $x_M = -x_L$.) We explore the search space using golden-section search [7]. Finally, we revert to the discrete domain by taking the sign of the continuous solution.

Our implementation of CC runs in time $O(n^3)$.

4 Termination

The second unspecified element of Algorithm 1 is the termination criterion that determines when to stop dividing. We could recurse all the way down to single nodes, but this is often unnecessary. In this section, we develop certain lemmas regarding structural properties of ADG subgraphs that tell us when we can safely stop the recursion.

First, some definitions. A subgraph $G(S)$ of the ADG G is *optimally static* if some optimal distribution mapping for G assigns the same distribution to all nodes in S. $G(S)$ is *necessarily static* if every optimal distribution mapping for G assigns the same distribution to all nodes in S.

For $v \in V$, let $\delta_{opt}(v)$ be some distribution d that minimizes $T_{node}(v,d)$. We call $\delta_{opt}(v)$ local minimum cost distribution at v. For any $S \subseteq V$, let $\delta_{opt}(S)$ be a best static distribution of $G(S)$, that is, some value of d that minimizes $\sum_{v \in S} T_{node}(v,d)$.

[2] We measure length in the infinity norm; but because of the bound $\|z\|_\infty \leq \|z\|_2 \leq n^{1/2}\|z\|_\infty$ that holds for all $z \in \mathbb{R}^n$, we can switch to the 2-norm $\|z\|_2$ without danger.

S is *unanimous* if there is a best static distribution $\delta_{opt}(S)$ that is also a local minimum cost distribution for each node $v \in S$; that is, there is some distribution d such that, for all nodes $v \in S$, $d = \delta_{opt}(v) = \delta_{opt}(S)$.

Define $\Delta(S)$ as $\min_d \sum_{v \in S} T_{node}(v, d) - \sum_{v \in S} \min_d T_{node}(v, d)$. $\Delta(S)$ gives the difference in node-cost between placing the entire node set at its best static distribution and placing each node at its local minimum cost distribution. Clearly, $\Delta(S) = 0$ if and only if S is unanimous; it is a measure of "dissention" in S. Define $\theta(S)$ as $\min_{v \in S}(\min_{d \neq \delta_{opt}(v)} T_{node}(v, d) - T_{node}(v, \delta_{opt}(v)))$. $\theta(S)$ gives the least possible cost of changing the distribution of a node in S from its local minimum to the distribution of next lowest cost. Let $w(S)$ be the total weight of edges with exactly one endpoint in S, multiplied by the communication parameter ρ. Finally, define $\text{mincut}(S)$ to be the smallest possible redistribution cost incurred when S is dynamic, that is, $\text{mincut}(S) = \min \text{cut-cost}(S_1, S_2)$ where the minimum is taken over all partitions of S into two nonempty subsets.

Now we prove some lemmas regarding the static properties of subgraphs. Our goal is to prove that a subgraph is optimally or necessarily static, because then we know that it is safe to terminate the recursion.

If $\Delta(S)$ is large, then it is hard to satisfy all nodes with a single, static distribution. Also, if $w(S)$ is also large, then the nodes that border S may "pull" its elements toward different distributions. On the other hand, when $G(S)$ has no low-weight edge cutset, it will be expensive to allow it to be dynamic. These competing factors are directly comparable, as we now demonstrate.

Lemma 1 (Min-cut) *If* $\text{mincut}(S) \geq w(S) + \Delta(S)$ *for a set* S*, then* S *is optimally static.*

Proof: Suppose $\text{mincut}(S) \geq w(S) + \Delta(S)$, and consider a distribution mapping that assigns different distributions to two nodes x and y in S. If we modify the given mapping to assign every node in S the distribution $\delta_{opt}(S)$, we pay at most $\Delta(S)$ (for the penalty in node costs) plus $w(S)$ (for the possible increase on edges joining S to the rest of the graph). Since the original mapping assigned different distributions to x and y, there is some set of edges with differently-distributed endpoints that separates x and y in S. This cut has cut-cost at least $\text{mincut}(S)$; making S static eliminates that cost. Thus we gain at least as much as we pay. □

The next lemma establishes sufficient conditions for a subgraph $G(S)$ to be optimally static in the special case where it is unanimous. This strengthens Lemma 1 for this case.

Lemma 2 (Unanimous) *If set* S *is unanimous and* $\text{mincut}(S) + \theta(S) \geq w(S)$*, then* S *is optimally static.*

Proof: Consider a distribution mapping of G in which S is dynamic. It suffices to show that under the conditions of the lemma, we can produce a distribution mapping of G that has lower completion time and in which S is static.

Given the proposed dynamic mapping of S, change it to a mapping in which S is colored $\delta_{opt}(S)$. In doing so, we can increase the completion time by at most $w(S)$. (Since S is unanimous, we could not have increased any node cost by mapping all nodes to distribution $\delta_{opt}(S)$.) On the other hand, we have reduced the completion time in two ways: first, by avoiding the redistribuion costs of the dynamic mapping; and second, by remapping some nodes to their local minimum cost distributions. The reduction from the first source is at least $\text{mincut}(S)$, and that from the second source is at least $\theta(S)$. Thus, we have decreased the completion time by at least much as we could possibly have increased it, producing a static mapping of S of lower completion time than the initial dynamic mapping. □

The final lemma shows how an optimally static subgraph may be enlarged while remaining optimally static.

Lemma 3 (Accretion) *Let* S *be optimally static and assume* $v \notin S$*. Define* $w(v, S)$ *to be* ρ *multiplied by the sum of the weights of edges connecting* v *to* S*, and similarly define* $w(v, \bar{S})$ *as* ρ *times the sum of the weights of all other* v*-incident edges. Finally, define* $range(v)$ *as* $(\max_d T_{node}(v, d) - \min_d T_{node}(v, d))$*. If* $w(v, S) \geq w(v, \bar{S}) + range(v)$*, then* $S \cup \{v\}$ *is optimally static.*

Proof: Consider a distribution mapping in which S is static with distribution d and v has a different distribution d'. Changing the distribution of node v to d reduces the completion time of the mapping by $w(v, S)$ and raises it by $w(v, \bar{S}) + \text{range}(v)$. Given the conditions on v, this results in a net reduction in completion time. Hence $S \cup \{v\}$ is optimally static. □

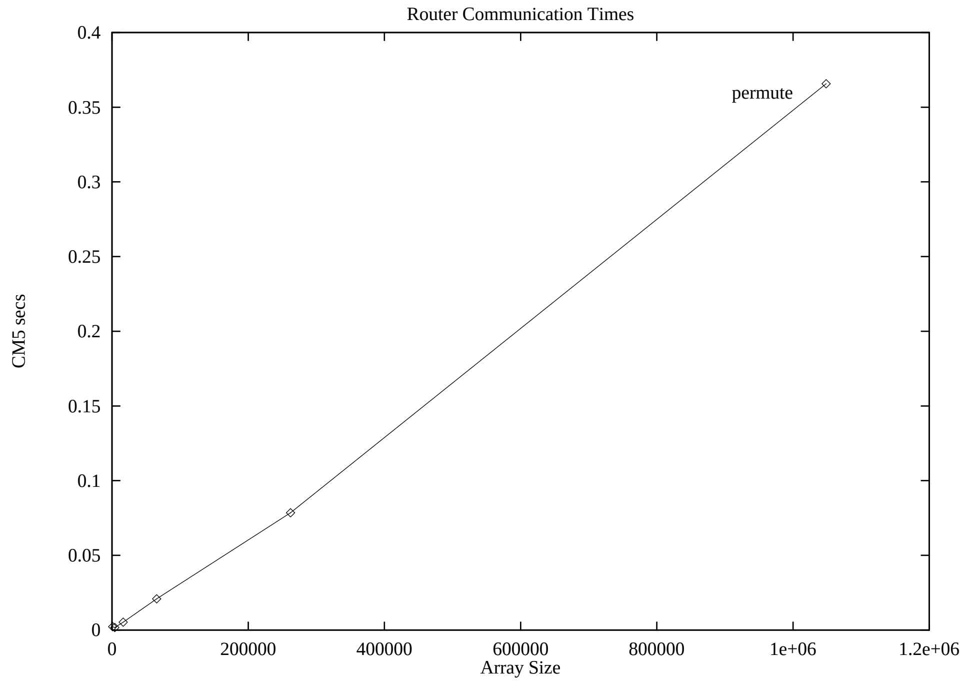

Figure 2: Performance of all-to-all personalized communication on the CM-5.

Note that the computation involved in verifying the inequalities is dominated by the time taken to find the global minimum cut mincut(S). A naive algorithm for this would run n single-source single-sink minimum cut computations (n being the number of nodes in S) for a total cost of $O(n^3)$ or more. Recently, Karger and Stein [11] have recently developed a probabilistic algorithm for this problem with $\tilde{O}(n^2)$ running time.

5 Modeling

This section fills in certain details concerning the modeling of the distribution problem. The specific issues covered here are use of the discrete metric for redistribution cost, choosing candidate distributions, and building the node-cost matrix C.

5.1 Redistribution cost

Changing distributions typically involves all-to-all personalized communication involving the router in a parallel machine. Each processor goes through two steps to complete the process: first, it must examine the data it currently holds, compute the identity of the processor that will hold it in the new distribution, and add it to the message buffer for that processor; then it must send out all the messages to the network. Rather than building a detailed model of the network incorporating routing algorithms, congestion, and the like, we model such communication using the discrete metric. The communication cost is also proportional to the size of the object whose distribution is being changed. Experimental evidence on the CM-5 (shown in Figure 2) reveals that this is an adequate model in practice. The program timed was written in CM Fortran. It performs a permutation of the columns of a (BLOCK, BLOCK) mapped square array on a 4×8 processor grid.

5.2 Choosing candidate distributions

The optimization framework described for the distribution problem requires a set of candidate distributions. We now present a heuristic method for generating a reasonable set of distributions based on the characteristics

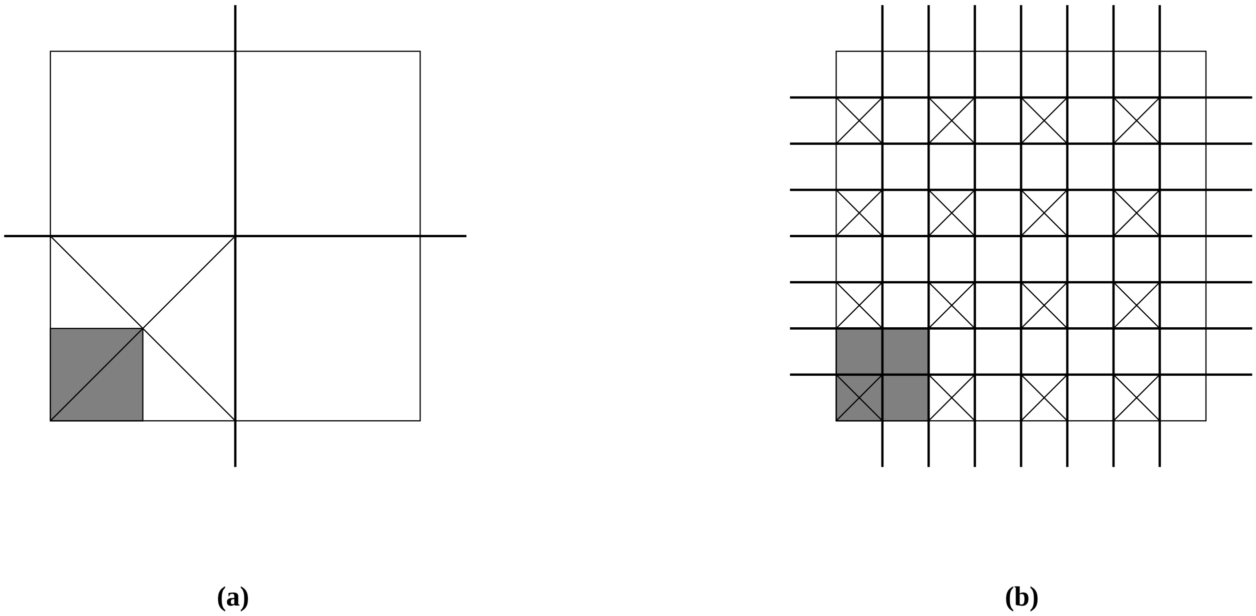

Figure 3: The interaction of features sizes and block sizes. The program has two arrays, one of size 1000 $\times$ 1000, and the other of 250 $\times$ 250 (shown shaded in the lower left corner). The template is distributed across a 2 $\times$ 2 grid of processors. (a) Block sizes are chosen to match the feature size of the larger array. The smaller array ends up on a single processor. (b) Block sizes are chosen to match the feature size of the smaller array. Both arrays are equally distributed across the processor grid (the parts of the large array marked X are all on one processor).

of the array objects present in a program.

A distribution is a partitioning of a t-dimensional template onto the available processors. A distribution may be identified with an ordered pair of t-vectors:

$$((p_1, \ldots, p_t), (k_1, \ldots, k_t)).$$

The first element describes allocation of the processors to the dimensions of the template, and the second gives the block size in each dimension. Thus, template cell $(i_1, \ldots, i_t)$ is located at processor coordinate $((i_1 \text{ div } k_1) \bmod p_1, \ldots, (i_t \text{ div } k_t) \bmod p_t)$.

Generating block sizes requires care. A naive algorithm might simply find the size of the template occupied by the array objects and generate a few block sizes based on that size. However, it is important to recognize and consider the different *feature sizes* of different objects. Consider an example program with one large array of size 1000×1000, and a small array of size 250×250, both aligned to the lower left corner of the template. This arrangement is shown in Figure 3.

Let there be four processors that are to be allocated with two per dimension, that is, $(p_1, p_2) = (2, 2)$. The size of the occupied template is the entire large array. A simple blocked distribution over these four processors has block sizes $(500, 500)$. This distribution would balance the distribution of the large array, but would leave the small array on only one processor. Now consider a distribution suited to the small array: block sizes $(125, 125)$. This distribution balances the elements of both the small array and the large array. The cost matrix entries computed for operations on the smaller array would reflect this difference in load balance between the two cases. We must generate distributions that are suited to objects with many different feature sizes and allow the divide-and-conquer partitioning algorithm to choose the right distribution from this set for each node in the program.

We first calculate the extents of all objects in the program. The extent of an array object is the size of the smallest t-dimensional box that encloses it over its iteration space. (Note that the size and position of an object may be functions of loop induction variables.) The extent e of an object is a t-vector.

The collection E of the extents of all objects in the program typically forms a small number of clusters in $\mathbb{R}^t$. We use histogramming to identify these clusters and to select a representative vector for each cluster. Call this set of representative extents R.

The set R is used to generate a set A of processor allocations. Each element of R gives a ratio in which processors are divided among the dimensions of the template. We add to this set an allocation that equally divides the processors if it is not already present. We also add allocations that give only one processor in dimensions for which some extent has a small value.

Block sizes must be chosen based on the extents of objects and also on the manner in which they are used. If an array object varies in size over the course of a program (for example, the active part of the matrix in an LU-decomposition) then a block size of 1 should be examined to achieve load balance over the whole iteration space. Similarly, an array object used in a stencil computation should have a large block size to minimize shift communication.

We extract sets F_1 through F_t of feature sizes from R by projecting individual components. Thus,

$$F_i = \{r_i \,|r \in R\}\,.$$

Additionally, we add a feature size of 1 to F_i if the size of any object varies in dimension i during program execution.

The set D of candidate distributions is constructed from the sets of processor allocations and feature sizes as follows:

$$D = \left\{ ((p_1, \ldots, p_t), (\lceil f_1/p_1 \rceil, \ldots, \lceil f_t/p_t \rceil)) \;\middle|\; \begin{array}{ccc} (p_1, \ldots, p_t) & \in & A \\ f_1 & \in & F_1 \\ & \vdots & \\ f_t & \in & F_t \end{array} \right\}$$

The resulting set of distributions could potentially be very large. This could render the divide-and-conquer algorithm unusable, because the running times of both our partitioners are sensitive to the number of distributions. In practice, programs typically have only a few feature sizes, and we generate only a few processor allocations. Our belief is that most programs can be analyzed using a few tens of candidate distributions.

5.3 Building the node-cost matrix C

The model, equation (1), for the completion time of an ADG separates the time into a component depending on the nodes and another component depending on the edges of the ADG.

Nodes perform computation and *intrinsic* communication. Intrinsic communication is communication that is performed as a part of the node computation. An example is the communication of values that happens during the summation of a distributed array. The value of $T_{comp}(v, \alpha, d)$ for an ADG node v with distribution d, and with a given alignment $\alpha(v)$, is determined by finding the largest number of elements held by one processor of the array computed at v under the mapping (of array elements to processors) $d \circ \alpha(v)$, and weighting it by the time per element of the computation done by node v. The only thing that complicates this is that the computation may be performed within a loop nest, and the sizes of the objects being computed can be functions of loop induction variables. So, in general, $T_{comp}(v, \alpha, d)$ is a sum over iterations of the compute time of node v at each iteration.

For a given iteration, the maximum processor load is a product over array axes of an array extent divided by an effective number of processors. The effective number of processors to which an array axis maps may be less that the corresponding element p_i in the distribution vector, since, with non-unit stride alignments and strided array sections, subsets of processors may actually hold array data. As an example, consider the following program fragment.

```
REAL A(4,8), S(4)
INTEGER I, ST
PROCESSORS P(2,4)
DISTRIBUTE A(BLOCK, BLOCK) ONTO P

ST = 1
DO I = 1, 3
    S = S + SUM(A(:, 1:8:ST), DIM = 2)
    ST = ST * 2
ENDDO
```

The maximum processor load is four at iteration one, and two at iterations two and three.

Edges, on the other hand, perform realignment and redistribution. An individual edge may carry zero or more of these forms of communication. Since alignments are determined in a previous compilation phase, we would like to treat the realignment communication as a known quantity. However, the realignment communication is still a function of distribution. For example, if there is shift communication along an array axis that is memory-resident, the realignment cost is in fact zero. Should an edge carry both realignment and redistribution communication, the realignment communication can be folded into the general redistribution communication at no additional cost.

With this in mind, we use the following approximation in building the matrix C. Here only, we must be aware of the direction of the edges of the ADG, which correspond to the direction of data flow. We find the realignment cost for edge (u, v) assuming distribution d at both head and tail, and add it into the cost $T_{node}(u, d)$ of node u. Thus, in equation (1),

$$T_{node}(G, \alpha, \delta) \equiv \sum_{u \in V} \left(T_{comp}(u, \alpha, \delta(u)) + \sum_{(u,v) \in E} T_{realign}(u, v, \alpha(u), \alpha(v), \delta(u)) \right). \tag{5}$$

The cost matrix C is calculated using this definition of T_{node}. It includes the realignment cost of any ADG edge that carries realignment in the node-cost for the node that is the source of the data communicated. The model is approximate; it overestimates communication time when an edge carries both realignment and redistribution communication.

Realignment communication comes in three forms. A change of the array axis to template axis map, or of an alignment stride, requires general all-to-all personalized communication; going from a nonreplicated to a replicated alignment (which is how the spread operator of Fortran 90 manifests itself in our system *after* the alignment phase) requires broadcast communication (possibly using a spanning tree algorithm); a change in array offset requires grid communication. We calibrate the communication characteristics of the machine using three parameters ρ, σ, and ν, which give the time per word transferred per processor in the three modes of communication. We use these parameters to scale the maximum processor load in computing realignment time.

6 Experiments

In this section, we compare the performance of partitioning algorithms NL and CC. The test graphs are small, but their characteristics are representative of genuine applications. We implemented both partitioners in MATLAB, and used the number of floating point operations (flops) as measured by MATLAB as a measure of the computation involved in solving a test case.

The first example ADG is the ladder graph shown in Figure 1. In this case, both NL and CC found the minimum-cost solution shown in Figure 1. However, our implementation of partitioner NL required 59836 flops to find this solution while our implementation of partitioner CC required 8326 flops.

The next example ADG is shown in Figure 4. It represents the structure seen in multizone applications such as the simulation of both the fluid dynamics and the structural mechanics on an airplane wing. In such a simulation, we have two or more data structures that undergo local computation and communication, with occasional transfers of smaller sets of data between them. A schematic of such a code is as follows.

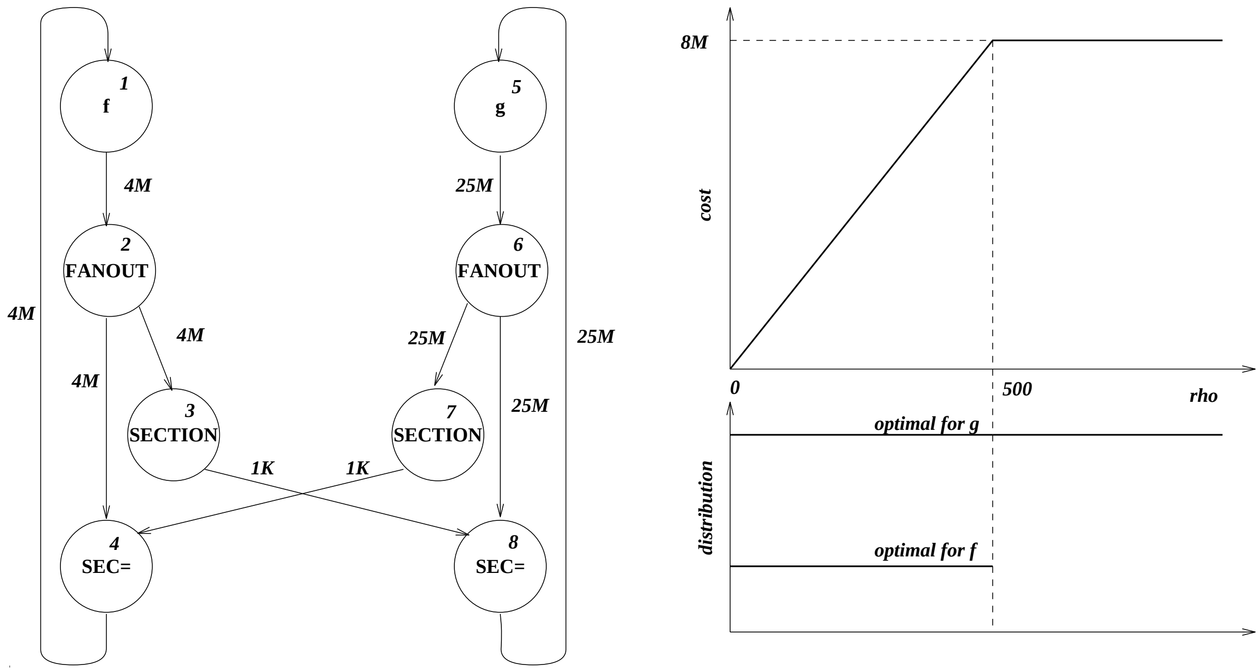

Figure 4: The ADG for the two-zone example, and the cost of its optimal partition and the distributions as a function of ρ.

```
REAL A(2000,2000), B(5000,5000)

DO I = 1,N
    A = f(A)
    B(1001:2000, 1) = A(:, 2000)
    B = g(B)
    A(:, 2000) = B(1001:2000, 1)
ENDDO
```

The function f encapsulates the structures computation, and function g encapsulates the fluids computation. We consider two candidate distributions, one being optimal for f and the other being optimal for g. Let the cost vector for the f-node be $10^6[1, 2]^T$ and that for the g-node be $10^6[12, 6]^T$. The two fanout nodes and two section nodes have cost zero (no computation is performed there). The cost of the section-assign (SEC= in the figure) nodes is negligible compared to the cost of the f- and g-nodes, so we take them to be zero as well. The node-cost matrix of the ADG is

$$C = 10^6 \begin{bmatrix} 1 & 0 & 0 & 0 & 12 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 6 & 0 & 0 & 0 \end{bmatrix}.$$

Finally, observe that the size of the left and right sections whose values are interchanged is 1,000.

Applying Lemma 1 to the entire ADG, we see that $w(G) = 0$, $\Delta(G) = 10^6$, and $\text{mincut}(G) = 2000\rho$. If $2000\rho > 10^6$, then the graph is optimally static, with cost 8×10^6. Otherwise, the ADG should be split down the middle, for a cost of $2000\rho + 10^6 + 6 \times 10^6$.

Algorithm 1 with partitioner CC finds this behavior, as shown in Figure 4. Algorithm CC always splits the ADG into two parts down the middle, but the Algorithm 1 checks this against the best static distribution and chooses the best static distribution when $\rho > 500$. Algorithm NL did not always function reliably; the solution depends on its initial starting point, and it often seemed to get stuck in local minima. Algorithm CC required about 7,700 flops, while algorithm NL required between 2.3×10^5 and 2.7×10^6 flops.

The final example is the ADG shown in Figure 5. This example shows the essential features of an alternating-direction implicit (ADI) iteration. We consider three distributions representing row orientation,

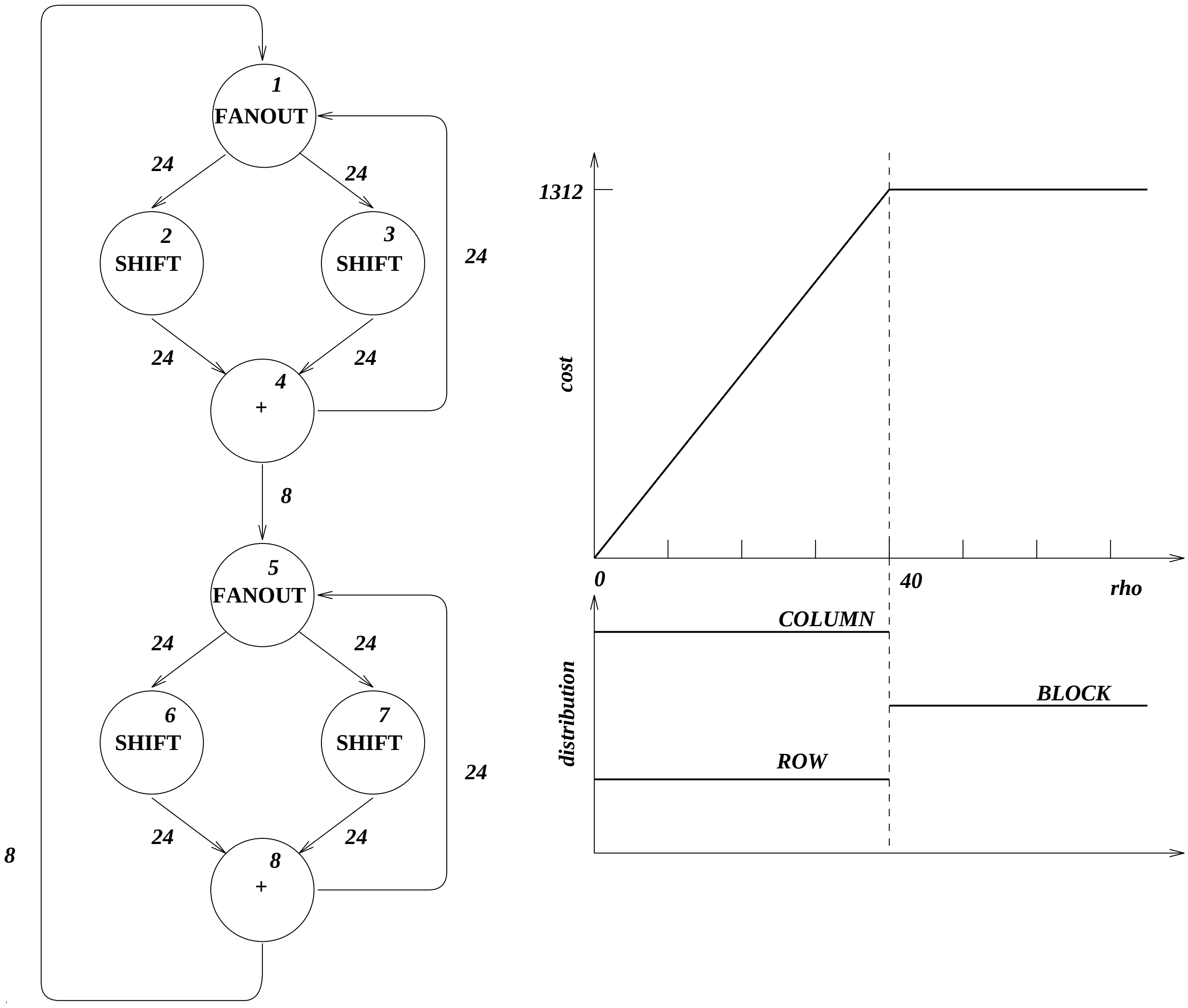

Figure 5: The ADG for the ADI example, the cost of its optimal partition and the distributions as a function of ρ.

block orientation, and column orientation. The node-cost matrix of the ADG is

$$C = \begin{bmatrix} 0 & 160 & 160 & 16 & 0 & 640 & 640 & 16 \\ 0 & 320 & 320 & 16 & 0 & 320 & 320 & 16 \\ 0 & 640 & 640 & 16 & 0 & 160 & 160 & 16 \end{bmatrix}.$$

An application of Lemma 1 shows that the ADG should be static at the block orientation if $\rho > 40$ (for a cost of 1,312), and should be dynamic with the first portion of the computation being performed in the row orientation and the second portion in the column orientation (for a cost of $672 + 16\rho$).

Again, Algorithm 1 finds the best of the distributions considered, for all ρ between 10 and 70. Partitioner CC was reliable and required about 1.1×10^4 flops, while partitioner NL occasionally failed to find a minimum, and required between 9.8×10^4 and 1.4×10^6 flops.

These examples show that our implementation of partitioner NL is still far from stable. We are investigating the reasons for its aberrant behavior. In any case, partitioner CC requires considerably less computation. The tradeoff between solution time and solution quality is unclear from these small examples; the heuristic used in partitioner CC is quite simple-minded, and it seems possible that partitioner NL may outperform it in solution quality for larger problems.

7 Conclusions

We have formulated the problem of determining data distributions as a partitioning problem on a graph representation of a program, and have presented a divide-and-conquer algorithm to solve the problem. We have developed two different partitioning algorithms for use in this method, and have implemented prototypes of both algorithms. Our tests on some small example programs reveals that these heuristics are reasonable.

We view this work as preliminary. We are currently looking into the effect of weakening the termination criterion on Algorithm 1 in order to limit the number of static subsets explored. This may produce a worthwhile acceleration of Algorithm 1 without worsening the resulting distribution mapping. We are trying to speed up Algorithm CC and are auditioning other hopefuls for the role of the vector x_M in it. We are also experimenting with a procedure that will find optimally static subsets *a priori*, and collapse them before Algorithm 1 is invoked. Finally, we are looking for more difficult and representative problems.

Acknowledgments

Dan Feng suggested the nonlinear optimization techniques in Algorithm NL.

References

[1] Robert Bixby, Ken Kennedy, and Ulrich Kremer. Automatic data layout using 0-1 integer programming. Technical Report CRPC-TR93349-S, Center for Research on Parallel Computation, Rice University, Houston, TX, November 1993.

[2] 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.

[3] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Thomas J. Sheffler. Modeling data-parallel programs with the alignment-distribution graph. To appear in *Journal of Programming Languages*, 1994.

[4] 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.

[5] 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.

[6] Roger Fletcher. *Practical Methods of Optimization.* John Wiley & Sons, second edition, 1989.

[7] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization.* Academic Press, Orlando, FL, 1981.

[8] 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.

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

[10] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

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

[12] Ulrich Kremer. NP-completeness of dynamic remapping. Technical Report CRPC-TR93-330-S, Center for Research on Parallel Computation, Rice University, Houston, TX, August 1993. Appears in the *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[13] Ulrich Kremer, John Mellor-Crummey, Ken Kennedy, and Alan Carle. Automatic data layout for distributed-memory machines in the D programming environment. Technical Report CRPC-TR93-298-S, Center for Research on Parallel Computation, Rice University, Houston, TX, February 1993. Appears in *Proceedings of the First International Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution and Automatic Parallel Performance Prediction (AP'93)*, Vieweg Verlag, Wiesbaden, Germany.

[14] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis and Applications*, 11(3):430–452, July 1990.

[15] Richard S. Varga. *Matrix Iterative Analysis.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1962.

[16] 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.