Efficient Distribution Analysis via Graph Contraction

Thomas J. Sheffler [*] Robert Schreiber [*] William Pugh [†] John R. Gilbert [‡]
Siddhartha Chatterjee [§]

Abstract

Alignment and distribution of data by an optimizing compiler is a dream of both manufacturers and users of parallel computers. The distribution problem has been formulated as an NP-complete graph optimization problem. The graphs arising in applications are large, and the optimization problem does not lend itself to traditional heuristic optimization techniques. In this paper, we improve some earlier results on methods that use graph contraction to reduce the size of a distribution problem. We report on an experiment using seven example programs that show these contraction operations to be effective in practice; we obtain from 60 to 99 percent reductions in problem size, the larger number being more typical, without loss of solution quality.

1 Introduction

Programmers expect that array parallel languages such as High-Performance Fortran (HPF) will provide high performance on distributed memory parallel computers, if they pay careful attention to the distribution of arrays to the available processors. Currently, array distribution must be performed by a programmer, who then annotates a program with distribution directives. This difficult task is further complicated by the fact that the optimal distribution for a program is dependent on the target machine. In the interest of simplifying the task of the programmer and enhancing the portability of array parallel programs, distribution should be handled by the compiler.

Unfortunately, distribution is a difficult combinatorial optimization problem [1]. Heuristic algorithms can be effective for small programs. However, for very large programs or very detailed analyses (employing inter-procedural analysis, for example) these algorithms may become less effective or unacceptably slow.

In this paper, we show how to reduce the size of a distribution problem. We recall the formulation [1, 2] of the distribution problem as a graph labeling problem, then show how parts of the graph may be eliminated through graph contraction operations. The contraction operations are based on identifying program regions (the nodes of a subgraph) that may be performed under the same distribution. Once identified, we collapse these regions into a single node that captures all of the information present in the original problem. Our contraction operations are lossless: they do not diminish the quality of solutions that may be found.

[*]Research Institute for Advanced Computer Science, Mail Stop T27A-1, NASA Ames Research Center, Moffett Field, CA 94035-1000. (sheffler@riacs.edu, schreiber@riacs.edu). The work of this author was supported by the NAS Systems Division via Contract NAS 2-13721 between NASA and the Universities Space Research Association (USRA).

[†]Department of Computer Science, University of Maryland, College Park MD 20742. pugh@cs.umd.edu

[‡]Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304-1314. gilbert@parc.xerox.com.

[§]Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175. sc@cs.unc.edu

We examine different strategies for applying the contraction operations and evaluate their relative merit. Initial experiments conducted with example programs show that these contraction operations are effective in practice. It is possible, moreover, that stronger contraction operations could further reduce problem sizes to the point where they could be solved exactly.

2 The graph model

A data-parallel program may be modeled as a directed graph (V, E). Each node $v \in V$ of the graph corresponds to an array operation in the program. An operation consumes one or more arrays, and produces one or more arrays as a result. A directed edge $(v, w) \in E$ connects a definition of an array object in array operation v to a use by operation w.

In our representation, there are also three special node types. A *fanout* node produces copies of an array with many uses; a *branch* node produces copies of an array with mutually exclusive uses, and a *merge* node combines mutually exclusive definitions of an array value (usually due to branches to a particular point in a program).

A *weight* labels each edge; it is (an estimate of) the number of elements in the array represented by the edge multiplied by an estimated trip-count of the edge in an execution of the program. In this way, the weight incorporates information about control flow.

Alignment is specified for the head and tail of each edge, giving the alignment of an array at its definition and use. Distribution must also be given for each node, specifying the distribution that is applied to each of the arrays involved in the node computation. The graph, along with these labels, is called an Alignment Distribution Graph (ADG) [3].

We allow the same mappings as HPF: an array is aligned to a template, which is distributed over the available processors. A template is an abstract array used as a target in alignment directives. An alignment is specified by four separate components. These are *axis*, *stride*, *offset*, and *replication*. *Axis* alignment determines the correspondence of array axes to template axes. *Stride* alignment specifies the factor by which the array is stretched across the template. *Offset* is a vector specifying the distance of an array from the origin of the template, and *replication* specifies certain axes of the template over which an array might be copied.

An example of alignment and distribution as specified in an HPF program follows. The first two directives declare the template and describe the alignment of the array. The third and fourth lines together describe the distribution. The `PROCESSORS` directive describes the allocation of arrays to the axes of the template, while the `DISTRIBUTE` directive specifies how the template is divided over the processors. In this case, the `BLOCK` directive says that the first dimension of the template is distributed in blocks of 25 over 4 processors, and the CYCLIC `directive` specifies that the second dimension is distributed in blocks of 10 over 8 processors. Since the extent of the template is 200 in the second dimension, the blocks will wrap around the processors.

```
real :: A(100,100)

!HPF$ TEMPLATE T(100,200)
!HPF$ ALIGN A(i,j) WITH T(i,j+100)
!HPF$ PROCESSORS P(4, 8)
!HPF$ DISTRIBUTE T(BLOCK, CYCLIC(10)) ONTO P
```

The alignment of an array object may change between its definition and use (as represented by the tail and head of an edge in the graph model). A change in alignment effectively changes the data mapping of an array and results in realignment communication. The type of communication needed to implement the realignment is determined by the component of an alignment that changes. Axis or stride realignment requires all-to-all personalized communication (AAPC), offset realignment requires shift communication, and replication realignment requires a spread (broadcast) communication operation. In general, redistribution requires AAPC.

In our model, each node in the graph is assigned a distribution. If an edge connects two nodes with different distributions, then the array carried by the edge is redistributed between its definition at the tail of the edge and its use at the head.

We choose to perform distribution analysis after alignment analysis. In our system, we first optimize axis and stride alignment, then replication alignment, followed by shift alignment. The order of the optimizations is motivated by the relative costs of the communication required by these types of realignment. It might be possible to find a better alignment if distribution information were known, but distribution analysis is difficult without some model of realignment communication costs. Consider the following example code fragment.

```
integer, parameter :: N = 1000
real a(N), left(N-2), right(N-2), cl, cr
t1 = cl * a(1:N-2)
t2 = cr * a(3:N)
a(2:N-1) = a(2:N-1) + t1 + t2
```

In our system, we would perform alignment first. The axis and stride alignments chosen here cause no realignment, but there is offset realignment in this typical finite-difference stencil computation. Because of the necessary offset realignment, which comes to light in the alignment optimization phase, we would prefer to give the arrays a block rather that a cyclic distribution, since this reduces data traffic when shift communication if performed. It would be difficult to determine this fact about distribution without having established alignment information first.

2.1 Modeling redistribution cost

The Alignment-Distribution Graph (ADG) $G = (V, E)$ may be used to model the effects of distribution decisions. Our system first finds a set D of candidate distributions. Node costs are recorded in a matrix, C, and edge costs are recorded in a matrix, W. Entry $C(d, v)$ estimates the time required to perform operation $v \in V$ under distribution $d \in D$. Realignment costs are also incorporated in this model by adding an estimate of the cost of performing the realignment, if any, on directed edge (u, v), for each distribution d, into the cost entry $C(d, u)$. Each edge, $(u, v) \in E$, has an associated weight, $W(u, v)$, which is an estimate of the time required to redistribute the array value communicated along the edge. Even though this single weight is a simplistic measure of redistribution (since it is insensitive to the actual starting and ending layouts) experiments have shown that this discrete metric accurately reflects the cost of performing a redistribution step [7].

We seek to give each ADG node a distribution in D, *i.e.*, we seek a mapping $m : V \rightarrow D$. For a particular distribution map, the cost of performing the computation of the graph is the sum over the nodes of the cost of performing their computation in the given distribution, plus the sum of the weights of all edges

whose endpoints have different distributions:

$$\text{cost}(m) = \sum_{v \in V} C(m(v), v) + \sum_{(u,v) \in E,\ m(u) \neq m(v)} W(u, v).$$

The goal of distribution analysis is to map each node to a distribution so that this cost is minimized.

The model's node cost component is trivially minimized by mapping each node to its distribution of smallest node cost; this can result in many edges carrying redistribution communication. Edge costs may be avoided entirely by mapping every node to some one distribution, the best of these being the distribution that minimizes the sum of the node costs. The optimal solution typically lies at neither of these extremes. The centrifugal tendencies toward reducing node costs (by labeling nodes independently) and eliminating edge costs (by labeling nodes identically) are what make the problem difficult. Bixby, *et al.* [1] in fact show that this formulation of the distribution problem is NP-complete.

2.2 The set of distributions

A distribution $d \in D$ specifies both the deployment of processors to the axes of the template and the blocking factor with which each axis is distributed to the processors (in a cyclic fashion). Our analysis requires a set D of candidate distributions.[1] The set may be specified by a programmer, or may be generated by a compiler as it analyzes the program. We adopt the latter approach.

The generation of a set of distributions requires care. The achieved cost is in general reduced by allowing a larger set of candidate distributions. But the running time of our optimization algorithms is sensitive to the size of D. We have previously shown how to select candidate distributions and how to limit their number [2].

2.3 Static and dynamic mappings

We introduce two terms to describe a distribution map for a subset of ADG nodes S. Let m be a given distribution map. Then S is static under m if m maps each element of S to the same distribution; S is *dynamic* under m otherwise. We say that the map m is static if V is static under m. Since all nodes in a static subset have the same distribution, no edges in a static subset carry redistribution costs and the cost of a static distribution is completely determined by the node costs.

The way in which we attempt to reduce the size of the graph is to identify *optimally static* (O.S.) subgraphs of the distribution graph.

Definition 1 (Optimally Static) *A subset $S \subseteq V$, is* optimally static *if for any map $m : V \rightarrow D$ there exists a map m' such that m' and m take identical values on $V - S$, S is static under m' and cost$(m') \leq$ cost(m).*

(Note the difference between this and the following definition: S is *weakly optimally static* if there is a minimum cost distribution map m under which S is static. The strong definition allows one to identify a family of disjoint O.S. subsets and collapse them simultaneously, the latter does not.)

Our overall plan for finding a distribution map m is this. We first determine a collection of subsets that we require to be static under m. This partially determines m. Then, for these subsets, we can ignore internal edges and aggregate them into a single "super" node whose node cost is the sum of the node costs

[1] This approach differentiates our strategy from others. For example, Wholey employed a search strategy to determine the optimal distribution for a program [8], and Gupta used heuristic methods to determine distribution parameters [6].

of its elements, thus reducing the size of the graph. Clearly, any O.S. subset should be so contracted, as this does not increase the cost of the best distribution that can be found. The remainder of the paper will be concerned with finding O.S. subsets.

Graph contraction based on O.S. subsets requires first finding a candidate subset and then testing whether it is O.S. The next section develops a theory of O.S. subsets. A later section discusses heuristic strategies for finding candidate subsets.

3 Node amalgamation for the distribution problem.

An understanding of properties of the distribution graph allows us to develop theorems that describe how subsets of nodes can be collapsed or amalgamated into super nodes, without changing the problem in an essential way. In this manner, we will reduce the size of the ADG as a first step in distribution analysis.

3.1 Definitions

Each cost matrix entry, $C(d, v)$, gives an estimate of the time required to perform the computation of node v under distribution d. It is convenient to speak of the cost table of a node, which is simply the column of entries pertaining to the node, denoted C_v. We extend this term to sets of nodes, S, where the cost table of a subset is simply the vector sum of the columns of the nodes in S, denoted C_S. Thus, $C_S(d)$ is the cost of performing all of the operations of S in distribution d.

Recall that a set of nodes S is O.S. if some optimal program distribution assigns the same distribution to all nodes in S. We may also speak of a static cost of a set of nodes, which is simply the cost of the set of nodes if they were to be placed at the same distribution. Because the cost table C_S gives the cost of the subset for each candidate distribution, we may obviously select the smallest value. The static cost of a set of nodes S is

$$\text{static}(S) = \min\{C_S(d) \mid d \in D\}.$$

We will also occasionally be interested in the worst static cost that a set could incur:

$$\text{worst}(S) = \max\{C_S(d) \mid d \in D\}.$$

The difference between the maximum and minimum cost of a single node is called the *range* of the node.

$$\text{range}(v) = \max_D C_v(d) - \min_D C_v(d).$$

Let a subset $S \subseteq V$ be given. We shall use a simple lower bound on the cost of a distribution map under which S is dynamic. Such a distribution map could potentially assign every element of S to its minimum cost distribution; or it could avoid redistribution costs on edges leaving S at the expense of some redistribution cost internally (assuming that S is connected). In either case, a dynamic solution divides S into at least two partitions with different distributions, and every edge crossing the partition carries redistribution communication. This redistribution cost is at least as great as the weighted minimum-cut (min-cut) of the undirected subgraph induced by S. Thus, for a distribution map m such that S is dynamic under m,

$$\text{dynamic}(S) \equiv \sum_{v \in S} \min C_v + \text{mincut}(S)$$

is a lower bound on cost(m).

Many of our proofs require consideration of the edges crossing from one set S to another set T. Define the function weight(S, T) as follows:

$$\text{weight}(S, T) = \sum_{v \in S, w \in T} W(v, w) + W(w, v).$$

Thus, weight$(S, \overline{S})$ is the sum of the weights of all edges entering or leaving S. We will also need a function that determines the maximum weight with which a set is attached to any particular neighbor.

$$\text{maxweight}(S) = \max_{v \notin S} \text{weight}(S, \{v\}).$$

3.2 Optimally static subsets

We present a number of tests that may be used to verify that a subset of nodes is O.S. Each of the lemmas below gives an explicit construction showing, for a class of subsets S, how a map with dynamic S can be modified on S to make S static and not increase the cost. A following section discusses the implementation of the tests and the expected running time of each.

Lemma 1 (Accretion) *[2] Let S be O.S. and assume $v \notin S$. If*

$$\textit{weight}(\{v\}, \overline{S}) + \textit{range}(v) \leq \textit{weight}(\{v\}, S)$$

then $S \cup \{v\}$ is O.S.

Proof: Any map may, by assumption, be modified on S to make S static without increasing its cost. Now consider a map 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 cost of the mapping by $w(v, S)$ and raises it by at most $w(v, \overline{S}) + \text{range}(v)$. By the hypotheses, this change also does not increase the cost. Hence $S \cup \{v\}$ is O.S. □

Corollary 1 (Series) *Given a node y with only two distinct neighbors, x and z, the set $S = \{x, y\}$ is O.S. if $W(y, z) + \textit{range}(y) \leq W(x, y)$.*

Proof: Since any singleton node is an O.S. subset, the corollary follows immediately from Lemma 1. □

This trivial corollary of Lemma 1 turns out to be very useful in practice: it identifies pairs of nodes that should be merged. In particular, unary operations representing SPREAD and REDUCE functions often have small ranges and have input and output edges of very different weights. Elementwise unary operations may have zero range with equal weights on their two incident edges.

Lemma 2 (Min-cut) *[2] A set S is O.S. if $\textit{static}(S) + \textit{weight}(S, \overline{S}) \leq \textit{dynamic}(S)$.*

Proof: Assume that S is dynamic under a given distribution map. If the inequality holds, then the cost of this map is not increased by assigning S to its best static distribution, incurring the static node cost as well as potential redistribution on all edges leaving S. □

The strategy of the previous lemma was to remap all of S to its preferred single location. As an alternative, we consider remapping all of S to the distribution of one of its neighbors, so as to make the union of S and that neighbor static. Define adj(S) to be those nodes outside of S with one or more neighbors in S.

Lemma 3 (Adjacent Vertex) *Let $S \subseteq V$. If*

$$(\textit{worst}(S) - \sum_{v \in S} \min C_v) + \textit{weight}(S, \bar{S}) - \textit{maxweight}(S) \leq \textit{mincut}(S), \tag{1}$$

then S is O.S.

Proof: Let S be dynamic under some map m, and let v be as in the hypotheses. Remap all nodes in S to the distribution of node v. The node costs can increase to the worst static cost of S, thus increasing by no more than the first term of the inequality 1; the edges from S except those touching v may now incur redistribution costs. By the hypothesis, these added costs are at least recompensed by the absence of redistribution along the edges internal to S. □

Note that although the construction in the proof guarantees that $S \cup \{v\}$ is static after the relabeling, we cannot conclude that $S \cup \{v\}$ is O.S., since we claimed a gain of mincut(S) after relabeling a map for which S is dynamic. The following reformulation allows us to conclude that $S \cup \{v\}$ is O.S.

Lemma 4 (Border) *Let $S \subseteq V$. If*

$$(\textit{worst}(S) - \sum_{v \in S} \min C_v) + \textit{weight}(S, \bar{S}) - \textit{maxweight}(S) \leq \textit{mincut}(S \cup \{v\}), \tag{2}$$

then $S \cup \{v\}$ is O.S.

Nodes that do elementwise computation often do not prefer any distribution to any other, as long as all the distributions in D balance the load (number of elements per processor).

Definition 2 (Apathy) *A set of nodes, S, is* apathetic *if range$(v) = 0$, for all $v \in S$.*

For a set S of apathetic nodes, worst$(S) = \sum_{v \in S} \min C_v$, so the first term of inequalities 1 and 2 vanish.

When S is an apathetic singleton set. Lemma 4 holds when one of the edges touching the node outweighs the sum of all others. In this simplified form, this lemma is useful for identifying single nodes that should be merged with a heavily connected neighbor into an O.S. subset.

4 Locating subsets

The lemmas developed in the preceding section verify that a subset of nodes is O.S., but do not reveal how to find candidate subsets. It is clearly impractical to consider all possible subsets of V, so we develop heuristics to help locate subsets with the potential to be O.S.

Lemmas 2, 3 and 4 compare static costs to dynamic costs. Static distributions must tolerate redistribution communication on edges leaving the subsets, while the dynamic distributions only incur the min-cut cost on edges internal to the set. In order for a subset to pass any of these tests, it must be highly connected internally (leading to a large min-cut value), with low-weight connections to nodes outside of the subset. Using this observation, we construct the connected components of the subgraph obtained by deleting from E all edges whose wieght is less than or equal to some specified threshold, t. The mincut of any such componenet is therefore not less than t, while the weight of each external edge is less than t.

We examine a set of thresholds, T, which is generated by histogramming the edge weights of the graph and dividing the histogram into buckets. The minimum value in each bucket becomes a threshold value in

the set T. To use this algorithm, we work through the thresholds in T from heaviest to lightest. We apply the O.S. tests to each connected component at the current threshold.

This heuristic is effective because of the way in which the edge weights of the ADG are calculated. Recall that the ADG incorporates the effects of control flow into its weight calculation by multiplying the weight of an edge by its estimated trip count. In the ADG, nodes corresponding to operations within loops are connected by high weight edges, and values are communicated into and out of loops by low weight edges (because they are traversed only once). The strategy above tends to find connected components encompassing the operations inside the bodies of loops. The buckets tend to correspond to different levels in loop nests.

The complexity of this subset finding algorithm is proportional to the number of edges, $|E|$, and the number of thresholds, $|T|$. The histogramming phase of the algorithm can be performed in time proportional to $|E|$, and connected components can be found in time $t = O(|E|)$ by using depth-first search. The enumeration of all subsets using this technique can be performed in $t = O(|T|\,|E|)$ time.

4.1 A slight modification

The procedure above locates subsets based only on edge weights, but it also beneficial to incorporate cost table information into the heuristic. If we look at the components of the static and dynamic costs due to the cost table entires, we note that we would like to minimize the following difference:

$$\Delta(S) = \min_{d \in D} \sum_{v \in S} C(d, v) - \sum_{v \in S} \min_{d \in D} C(d, v).$$

Elsewhere, we have called this quantity the "dissension" of a set of nodes[2]. A set has a dissension of 0 if all nodes agree on the best distributon.

We initially tried modifying the subset selection procedure to find subsets with a dissention of 0, but this strict criterion did not work well in practice. We also tried heuristics for find subsets having low dissension; these approaches are still under investigation.

5 Implementing the O.S. tests and the contraction operation

This section suggests data structures and algorithms for implementing the tests of the preceding section. While none of the algorithms presented is difficult to implement, a naive implementation of the tests could lead to poor running times. In particular, dense representations of the matrices C and W are inappropriate not only because of the waste of storage that would occur, but because suitable graph traversals are not supported. Sparse matrix algorithms are necessary because of the sparse structure of the ADG. Contraction operations maintain sparsity as well.

5.1 Data structures

The matrices C and W are stored as sparse matrices. An element in a matrix is a record structure storing its row, column, and value, and pointers threading it into two doubly-linked lists: a list of elements in the same row, and another list of elements in the same column. For each matrix, two vectors of pointers record the first element in each row and column. The elements of the lists are unordered.

Finding a particular matrix element in this data structure requires potentially searching through an entire row or column list. However, our algorithms do not require finding individual elements quickly, but rather

depend on a data structure that supports efficient traversals of various types. The main operations required are the following.

insert: Given a pointer to an matrix element, insert it into the appropriate row and column lists. This may be accomplished in $O(1)$ time.

delete: Given a pointer to an matrix element, delete it from both of its lists. This may be accomplished in $O(1)$ time.

neighbors: Enumerate the neighbors of a given node, v. These may be enumerated by iterating through the row and column lists of v in the weighted adjacency matrix, W.

A set of nodes S is represented in a simple linked list structure so that its nodes may be enumerated. Our algorithms also require two mark vectors, called $mark$ and $border$, each of length $|V|$. These two vectors are cleared once at the beginning of the program. Some of the O.S. tests will set and clear marks in these vectors, but none will be required to touch the entire vector.

Our algorithms use a Sparse Accumulator (SPA) to add sparse vectors [4]. A sparse vector is a sequence of `<index,value>` pairs, where the largest possible index is some value N. The SPA maintains 3 internal vectors of length N called `accum`, `flags` and `elts`. At program startup time, these vectors are allocated and cleared. The vector `accum` holds the values of all elements of a sparse vector, `elts` is used as a stack that records the indices of nonzero elements in `accum`, and `flags`, a boolean vector, is set if the corresponding element has been pushed onto `elts`. A SPA is used to compute the sum of several sparse vectors in time proportional to the number of nontrivial arithmetic operations actually performed.

5.2 Contracting nodes

The node contraction operation replaces a set of nodes, S, with a single node s in a reduced graph. The node cost vector C_S of the new node is the sum of the node costvectors of its elements, and the weight of each edge incident to S is the sum of the weights of all edges between the adjacent node and elements of S. Precisely, this is written as

$$C(d,s) = \sum_{v \in S} C(d,v), W(s,v) = \sum_{v \in S, w \notin S} W(v,w), W(v,s) = \sum_{v \in S, w \notin S} W(w,v).$$

Our technique contracts S by adding the sparse vectors that encode the edge weight and adjacency information for the nodes in S. Merging the cost table entries for the nodes requires adding the corresponding columns of C. Thus, the cost table entries for a set can be computed in $O(|S| \cdot |D|)$ time.

Merging the adjacency table entries requires merging both the row and column lists for the nodes of S. A row or column is a sparse vector, and so we may use a SPA to implement the merger. In the row merge phase, add each element of the S row lists into the SPA in unit time per element. Enumerate the elements of the SPA, creating a new matrix element for each and insert it into its row and column lists in unit time. Clear the SPA when done. Column contraction proceeds in the same manner.

Define degree(v) as the number of nodes adjacent to node v in the graph, and let $n = \sum_{v \in S}$ degree(v). The total running time of the contraction of a set of nodes is $O(|S| \cdot |D| + n)$.

5.3 Series test

The series test is easy to implement. Determine if a node is a series node by traversing its adjacency list. If it has two or fewer neighbors, record the edges and the weights. Compute the range of the node by examining its $|D|$ cost table entries. The series test can be applied all nodes in $O(|V| \cdot |D|)$ time.

We will show later that the series test is effective at reducing the size of the graph. Because it is so simple to implement, this test should always be used.

5.4 Apathetic test

We only apply this test to single nodes, even though it is defined for sets. Determine if a node is apathetic by examining each of its $|D|$ cost table entries. If the node is apathetic, then visit its neighbors, keeping a running sum, w, of the weights of all edges encountered, and recording the heaviest edge, *maxweight*. If *maxweight* $\geq (w -$ *maxweight*$)$, then merge the node with its most heavily connected neighbor. The apathetic test can be applied to all nodes in time $O(|V| \cdot |D| + |E|)$ time.

5.5 Dynamic cost

The rest of the tests require computing the dynamic cost of a set of nodes, S. Clearly, the sum of the minima of the node costs can be computed in $O(|S| \cdot |D|)$ time. The difficult part is computing the min-cut value. There are two options: use an easily obtained lower bound on the min-cut, or compute it exactly.

If S is connected, the mincut(S) is not less than than the minimum weight edge in S. There are at most $|S|^2$ edges and we may find the lightest one by examining all edges internal to S. To be precise, we first enumerate S setting $mark[v]$ for each v in S, and then look at all edges connected to vertices in S. Any whose other endpoint is marked in $mark$ is an internal edge. We must also clear $mark$ when we are done. The running time of this computation is $O(|S|^2)$.

In the second case, we compute the global min-cut of the set exactly, using an algorithm of Goldberg and Tarjan which runs in $O(|S|^4)$ time [5]. In practice, when using this option, we only invoke the min-cut procedure when the size of the set is smaller than some predefined value – because the running time of the min-cut procedure becomes unnaceptable for large sets.

In summary, an estimate of the dynamic cost of a set of nodes may be found in $O(|S| \cdot |D| + |S|^2)$, or $O(|S| \cdot |D| + |S|^4)$ time. In estimating the time of the following tests, we will simply write $dyn(S)$ as the running time for finding the dynamic cost of a set of nodes.

5.6 Best static test

The static cost of a set of nodes is computed in $O(|S| \cdot |D|)$ time by first creating the cost vector for the set of nodes and then finding the minimum value. Computing the weight of edges leaving S requires first setting $mark[v]$ for each $v \in S$ and then traversing all edges adjacent to vertices in S. This test may be implemented in $O(|S| \cdot |D| + \sum_{v \in S} \text{degree}(v) + dyn(S))$ time.

5.7 Border outside test

The worst static cost of a set of nodes is computed as easily as computing the best static cost, and the weight of all the edges leaving S can be computed as before. This test is different in that all vertices external to S must be visited in turn. When computing the weight of edges external to S, set $border[v]$ for any edge

encountered connecting S to an external node v. If $border[v]$ was not set when encountered, then add v to the set of border nodes called B. (The marks may be cleared by traversing B at the end of the Border Outside test.) For each external border node, compute the weight that connects it to S using the $mark$ vector to determine nodes in and out of S in $O(\text{degree}(v))$ time for each border node. The running time of this test is $O(|S| \cdot |D| + \sum_{v \in S} \text{degree}(v) + \sum_{v \in B} \text{degree}(v) + dyn(S))$. This test is not much more expensive than the Best Static test if B is not too big.

5.8 Border inside test

The Border Inside test is similar to the Border Outside test except that for each node v in the border we must compute worst$(S - \{v\})$ and min C_v. This may be done in the following way to make the test run in the same time as the previous one. Initially, instead of computing worst(S), record the cost vector for the set, C_S. Find the border nodes as before. Now, as each border node is visited, make use of the fact that worst$(S - \{v\})$ = max$(C_S - C_v)$ and compute both this value and min C_v in $O(|D|)$ time. The rest of the implementation of the test is the same as the Border Outside test and can be implemented in $O(|S| \cdot |D| + \sum_{v \in S} \text{degree}(v) + \sum_{v \in B} \text{degree}(v) + dyn(S))$ time.

6 Experiments

We now present an experimental study of the effectiveness of the contraction operations developed earlier. The process of locating subsets and verifying that they are O.S. is heuristic; such a study is therefore mandated, and we view the data below as preliminary, pending better tools and a larger base of experimental programs.

Using program analysis tools we have developed earlier [3], we constructed the distribution graphs for seven test programs and applied various combinations of the contraction operations. The contraction operations are sensitive to the adjacency structure of the graph as well as values of the cost entries. For this reason, it is important to understand how the test cases were generated. We begin by describing the example programs and how the cost values were calculated. We then discuss contraction strategies and examine the results of these strategies.

6.1 The example programs

We chose seven example programs that represent typical scientific applications. A brief description of each of the seven follows. In addition, Table 1 describes properties of the cost and adjacency tables for each of the programs. Each of the graphs is quite sparse. With the exception of `BlockLU`, each program was analyzed with a relatively small number of distributions. Because BlockLU has many different feature sizes, a large number of distributions are generated by our automatic system.

`ADI`: A two-dimensional alternating-direction implicit (ADI) algorithm. This algorithm uses cyclic reduction to solve tridiagonal systems.

`BlockLU`: A blocked algorithm for LU factorization of a dense matrix.

`Erle`: A three-dimensional alternating-direction implicit (ADI) algorithm. This differs from the one above in that it uses Gaussian elimination to solve the recurrences.

`LU`: The LU factorization on a dense matrix.

Table 1: Properties of the example program graphs. Each is quite sparse. In general, the number of distributions used in the analysis of each program is relatively small, with the exception of `BlockLU`.

Properties of the Programs			
Program	$\|V\|$	$\|E\|$	$\|D\|$
`ADI`	232	308	12
`BlockLU`	108	131	41
`Erle`	666	845	7
`LU`	21	25	12
`Shallow`	445	545	3
`Tred`	105	124	9
`TwoZone`	335	411	12

`Shallow`: A benchmark weather prediction program; finite-difference approximation of the the shallow water equations.

`Tred`: Reduction of a dense matrix to tridiagonal form using Householder transformations.

`TwoZone`: Solution of Poisson's equation in an L shaped domain by Schwartz alternating procedure, using a JOR (Jacobi Over-Relaxation Method) for the subdomain solver.

6.2 Cost table construction

In Section 2, we differentiated between three communication patterns: all-to-all personalized communication (AAPC), offset communication (shift), and reduction/replication communication. When analyzing a program, we estimate the time of an elementwise operation to be proportional to the amount of data on the most heavily loaded processor. We estimate the time of a communication operation to be proportional to the maximum amount of data sent or received by any one processor, with the constant of proportionality determined by the type of operation. The three constants are ρ (for AAPC), σ (for reduction/replication) and ν (for shift). (The names recall the now ancient and disappearing Connection Machine jargon: *router, scan, NEWS*). High-level operations in HPF give rise to one of these three types of low-level communication. Table 2 shows the correspondance between high-level and low-level communication operations.

In general, it is impossible to predict how varying the parameters, ρ, σ, and ν, will affect the contraction operations. Even the interaction between this model of communication and the cost values generated is quite complex. Realignment costs are incorporated into the node cost table, while redistribution costs affect adjacency information. Varying the parameters by the same factor changes the relationship between elementwise computation and communication. Varying the parameter ρ can affect values in both, while varying σ or ν can only affect values in the cost table. Because of these complex interactions, we ran tests of the contraction operations for a number of values of the parameters to see how the results changed.

6.3 Contraction operation strategies

The contraction operations may be applied individually, or in combinations. In the discussion of the combinations of contraction operations we will use a shorthand. The character "a" means an application

Table 2: Mapping of high level HPF operations to low-level communication types. Each of the three low-level operations is modeled as requiring time proportional to the amount of data communicated, with the constant of proportionality as shown.

Coefficients of Proportionality		
High-Level Operation	Low-Level Communication Type	Constant
Redistribute	AAPC	ρ
Stride Realign	AAPC	ρ
Axis Realign	AAPC	ρ
Offset Realign	shift	ν
Replication Realign	broadcast	σ
Subscript	AAPC	ρ
Reduction	fan-in	σ

of the Apathetic test, an "s" the Series test. The character "S" stands for the Best Static test, and "B" for the Border tests. Each of the last two tests are parameterized by the subset selection method used, "cc" or "ccc", and the method of estimating the min-cut. The "cc" subsets are connected components of edge-weight thresholded subgraphs. The "ccc" data employ a heuristic designed to improve the dissention of these subgraphs by removing additional edges before finding the connected components. The keyword "min" means the minimum weight edge in the subset was used as a bound, and an integer P means that an exact min-cut was computed for subsets of size less than P.

6.4 Results

For each of the seven programs, we generated test data assuming a 64 processor target using three sets of values for the communication parameters as shown below.

Case1	$\rho = 1$	$\sigma = 1$	$\nu = 1$
Case2	$\rho = 10$	$\sigma = 1$	$\nu = 1$
Case3	$\rho = 100$	$\sigma = 10$	$\nu = 1$

Case 1 reflects a target architecture where communication costs as much as computation. There is no such machine widely available today, but such a machine would tolerate a lot of redistribution, prefering dynamic distributions over static ones. Thus, this case should thwart many of our contraction operations.

Case 2 reflects a case where communication is only slightly expensive. Scans and shifts are still very inexpensive. Case 3 is the most realistic model of current machines. Shift communication is very cheap, but any general AAPC communication is quite expensive. We would expect both of these cases to enourage static solutions to the distribution problem, and thus expect our contraction operations to do well.

Tables 3, 4 and 5 show the results of the contraction experiments for the three different cases. The size of the original program is shown at the top of each column, and the results of applying nineteen different contraction combinations below. The combinations are divided into groups. The first group includes only the single-vertex tests ("a" and "s"). The next group shows the subset tests ("S" and "B") alone. The next groups show the "S" or "B" tests preceded by the "as" combination. It is clear from the data

Table 3: The size of the contracted graphs with the communication parameters of Case1. With ρ set low, we expect to see graphs that prefer dynamic distributions, thus it should be difficult to prove any that any subgraphs are O.S. Many of the test graphs are contracted significantly.

Contraction Results for Case 1: $\rho = 1, \sigma = 1, \nu = 1$							
method	ADI	Block	Erle	LU	Shal	Tred	TwoZ
ORIGINAL	232	108	666	21	445	105	335
s	82	41	295	5	125	32	150
a	220	84	548	16	339	77	310
as	82	41	289	5	125	32	150
asas	81	39	285	5	125	29	150
S(cc,min)	232	57	555	3	445	80	73
S(ccc,min)	232	84	552	21	441	105	319
B(cc,min)	224	84	555	13	445	80	273
B(ccc,min)	224	72	529	11	425	79	317
asS(cc,min)	82	17	253	2	22	23	2
asS(cc,25)	42	10	253	2	22	8	2
asS(cc,50)	42	10	253	2	22	8	2
asS(ccc,min)	82	29	253	4	125	29	74
asB(cc,min)	74	27	253	4	37	23	105
asB(cc,25)	74	24	253	2	37	23	74
asB(cc,50)	74	24	253	2	37	23	74
asB(ccc,min)	74	30	253	3	125	23	105
asB(ccc,25)	72	27	253	3	114	21	102
asB(ccc,50)	72	27	253	3	114	21	74
asS(cc,min)B(cc,min)	72	17	253	2	22	23	2

that the combination that nearly always achieves the best contraction of all those tried is the "asS(cc,50)" combination. The "asS(cc,min)" combination also fared well many of the times, but occasionally the true min-cut values were critical to obtaining further contraction.

The contraction programs that involve only the Apathetic and Series contraction operations ("a" and "s") are actually very effective considering just how inexpensive they are to implement. Recall that these tests examine only the neighbors of single nodes. These two contraction operations should *always* be applied first, as they are *enabling* contractions for the more powerful Best Static and Border contractions. Notice that in most of the test cases, the Best Static or Border tests alone are ineffective, while pre-contracting with the Apathetic and Series tests helps them considerably. Also notice that repeating the Apathetic and Series tests ("asas") is not worthwhile.

The overall percentage of reduction achieved by the "asS(cc,50)" combination is shown in Table 6. Initially, we did not expect to be able to contract the graphs very much for Case1 because redistribution is fairly inexpensive in this case. With low edge-weights, we did not expect to find many O.S. subgraphs. The results show that, on the contrary, the tests are effective even when ρ is small.

In most of the trials, we used the Best Static and Border tests in a mutually exclusive manner. The final case chains the two together. In almost all of the trials, we observe the the results produced by the Border

Table 4: The size of the contracted graphs with the communication parameters of Case2.

Contraction Results for Case 1: $\rho = 10, \sigma = 1, \nu = 1$							
method	ADI	Block	Erle	LU	Shal	Tred	TwoZ
ORIGINAL	232	108	666	21	445	105	335
s	68	35	283	5	125	32	144
a	220	84	548	16	339	77	310
as	68	35	277	5	125	32	144
asas	67	33	273	5	125	28	144
S(cc,min)	232	65	555	3	445	16	73
S(ccc,min)	232	84	552	21	441	105	319
B(cc,min)	232	62	555	9	445	80	153
B(ccc,min)	226	54	529	7	425	81	319
asS(cc,min)	68	20	46	2	22	8	2
asS(cc,25)	68	2	46	2	22	8	2
asS(cc,50)	68	2	21	2	22	8	2
asS(ccc,min)	68	24	241	4	125	29	70
asB(cc,min)	68	19	241	2	22	23	37
asB(cc,25)	68	12	241	2	22	23	37
asB(cc,50)	68	12	241	2	22	23	37
asB(ccc,min)	66	20	241	3	125	21	70
asB(ccc,25)	66	11	235	3	114	21	70
asB(ccc,50)	66	11	235	3	114	21	70
asS(cc,min)B(cc,min)	68	19	46	2	22	8	2

Table 5: The size of the contracted graphs with the communication parameters of Case3.

Contraction Results for Case 1: $\rho = 100, \sigma = 10, \nu = 1$							
method	ADI	Block	Erle	LU	Shal	Tred	TwoZ
ORIGINAL	232	108	666	21	445	105	335
s	68	35	283	5	125	32	144
a	200	84	548	16	339	77	310
as	68	35	277	5	125	32	144
asas	67	33	273	5	125	28	144
S(cc,min)	232	87	555	3	445	16	73
S(ccc,min)	232	90	552	21	441	105	335
B(cc,min)	232	81	555	3	445	80	73
B(ccc,min)	226	60	529	20	421	73	335
asS(cc,min)	5	24	241	5	22	8	2
asS(cc,25)	5	3	235	2	22	8	2
asS(cc,50)	5	3	235	2	22	8	2
asS(ccc,min)	68	28	241	4	125	29	144
asB(cc,min)	16	18	241	2	22	23	2
asB(ccc,min)	66	18	241	3	125	21	144
asB(ccc,25)	64	14	241	32	114	21	136
asB(ccc,50)	64	14	241	32	114	21	136
asB(cc,25)	16	3	241	2	22	8	2
asB(cc,50)	16	3	241	2	22	8	2
asS(cc,min)B(cc,min)	5	18	241	2	22	8	2

Table 6: The amount of contraction as a percentage of the total size for the combination "asS(cc,50)." This particular combination proved the most effective overall.

Percentage Contraction using asS(cc,50)							
	ADI	Block	Erle	LU	Shal	Tred	TwoZ
Case1	82%	91%	62%	90%	95%	92%	99%
Case2	71%	98%	95%	90%	95%	92%	99%
Case3	98%	97%	65%	90%	95%	92%	99%

tests are worse than those produced by the Best Static test. This is not surprising, because the Border tests involve the *worst* static value which is generally a loose bound. What is surprising is that in some cases the Border tests do quite well.

7 Conclusions

When we began formulating algorithms for solving the distribution problem, we originally felt that sophisticated optimization techniques would be needed. We now believe that contraction operations can dramatically reduce the size of a distribution problem without losing information. With effective contraction operations, problem sizes become so small that less powerful optimization strategies may suffice. Indeed, some problems become small enough that it may be possible to find optimal solutions exactly.

Some issues that remain open are these. Should one relax the requirement that the contraction operations remain lossless — contraction operations may be accepted for subgraphs that are not necessarily O.S. What is the tradeoff, if this is done, between compile time and run-time? Is it better to do a heuristic optimization of a big but exact distribution problem or an exact optimization of a small but approximate problem? We also need to reexamine our subset selection procedure. It is interesting whether, in the few cases in which the contracted graph remains large, the reason is that we haven't found the right subsets to test, or our lemmas are not powerful enought to detect that the sets we select *O.S.*, or simply that there aren't any more O.S. sets left to be found.

References

[1] R. Bixby, K. Kennedy, and U. 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] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler. Array distribution in data-parallel programs. In K. Pingal, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, number 892 in Lecture Notes in Computer Science, pages 76–91, Ithaca, NY, August 1994. Springer-Verlag. Also available as RIACS Technical Report 94.09.

[3] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler. Modeling data-parallel programs with the alignment-distribution graph. *Journal of Programming Languages*, 2:227–258, 1994. Special issue on compiling and run-time issues for distributed address space machines.

[4] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, January 1992.

[5] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, October 1988.

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

[7] P. Hough and T. J. Sheffler. A performance analysis of collective communication on the CM-5. Excalibur project meeting note.

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