International Journal of Parallel Programming, Vol. 24, No. 6, 1996

Efficient Distribution Analysis via Graph Contraction

Thomas J. Sheffler,[1,6] Robert Schreiber,[2,6] William Pugh,[3] John R. Gilbert,[4] and Siddhartha Chatterjee[5,6]

Alignment and distribution of array data should be managed by optimizing compilers for parallel computers, but current approaches to the distribution problem formulate it as an NP-complete graph optimization problem. The graphs arising in applications are large and difficult to optimize. 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 70 to 99 percent reductions in problem size, the larger number being more typical, without loss of solution quality.

KEY WORDS: Parallel computing; array distribution; optimizing compilers; graph algorithms.

[1] Rambus Inc., 2465 Latham Street, Mountain View, California 94040. (E-mail: sheffler@rambus.com).

[2] Hewlett Packard Laboratories, 1501 Page Mill Road, Palo Alto, California 94304-1126. (E-mail: schreiber@hpl.hp.com).

[3] Department of Computer Science, University of Maryland, College Park, Maryland 20742. (E-mail: pugh@cs.umd.edu).

[4] Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304-1314. (E-mail: gilbert@parc.xerox.com). Copyright © 1995 by Xerox Corporation. All rights reserved.

[5] Department of Computer Science, University of North Carolina, Chapel Hill, North Carolina 27599-3175. (E-mail: sc@cs.unc.edu).

[6] The work was supported by the NAS Systems Division via Contract NAS 2-13721 between NASA and the Universities Space Research Association (USRA) while he was at RIACS.

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 the programmer, who 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. For very large programs or very detailed analyses (employing interprocedural 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 the size of the graph may be reduced through graph contraction without affecting the best solutions. We propose algorithms that identify regions of the program (the vertices of a subgraph) that may be performed under the same distribution. Once identified, the algorithm collapses each such subgraph into a single vertex that captures all of the information present in the original problem. In contrast, other authors have advocated controlling the size of the data flow graphs modeling distribution problems, but they have used coarse criteria, such as forcing any loop to be executed without redistribution.(1)

This paper has two components: a rigorous analysis of an abstract graph model, and a discussion of the implementation of the algorithms in our compilation system. The graph model and the contraction lemmas discussed could be used in any other compilation system as a pre-processing step to further distribution analysis. We use the results collected from our own implementation to show that the theoretical results presented are effective in practice.

In our experiments, 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 very effective. It is possible, moreover, that stronger contraction operations will further reduce the size of problems to the point where they could be solved exactly.

1.1. Related Work

While we are aware of no other approaches to the distribution problem through graph contraction, there is a significant amount of work done concerning other aspects of automatic distribution. Some work regarding alignment analysis has actually considered the effects of distribution.[3, 4] However, these and other papers regarding alignment do not address the problems introduced by allowing a rich set of distribution parameters that includes block, cyclic, and block-cyclic distributions by array axis.

Formulations of the distribution optimization problem may be categorized as either solving the *static* or *dynamic* distribution problem. Previous researchers have focused on the *static* version, in which the distribution of each array remains fixed throughout the execution of the program. Wholey[5] uses a hillclimbing procedure to successively refine the distribution parameters for a given program until the program can no longer be improved. His algorithm attempted to select the best number of processors and dimensions to be distributed, but can get caught in local minima. Gupta[6] uses heuristic methods to decide between block and cyclic partitioning, and other parameters are selected based on estimated program costs. Our formulation of the distribution optimization problem allows the distribution of an array object to change over its lifetime. This is the *dynamic* distribution problem.

In contrast to some previous approaches, which directly generate the distributions for arrays, our analysis begins with a set of candidate distributions and selects one of these for each array. The resulting solution is naturally dynamic.

We previously introduced a divide-and-conquer approach to the dynamic distribution problem.[3] In this approach, the program is recursively divided into regions to which static distributions are independently assigned. The conquer stage merges two regions, choosing a static distribution when the cost of the dynamic distribution is worse. Recently, Gupta's techniques have been applied by Palermo to the dynamic distribution problem.[7] Palermo's is also a divide-and-conquer approach, and uses Gupta's static analyzer to assign distributions to the regions generated.

Kremer[8] and Bixby, *et al.*[1] use essentially the same data flow graph and cost model as we. (We allow only a single distribution candidate set, while they allow a different set for each program operation; one formulation may be easily transformed into the other.) The first of these papers showed that this formulation of the distribution problem is NP-complete. In the second, the problem is reduced to 0,1-integer programming, and it is shown that for modest sized problems heuristic methods (in the form of the CPLEX package) are effective at finding an optimal solution.

Our approach may be viewed as a preprocessing step for these methods. We reduce the problem size so as to speed up the subsequent analysis.

2. MODELING DISTRIBUTION

This section first describes how data mapping is specified in HPF through alignment and distribution directives. We then propose an array data flow graph with a cost model that estimates the running time of a program as a function of the alignments and distributions of its arrays. We discuss the generation of a set of candidate distributions. Finally, we discuss the graph contraction process that we develop in the following sections.

2.1. Alignment and Distribution

Our model of data mapping follows that of 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 or distribution of an array object may change throughout its lifetime in a program. 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, offset realignment requires shift communication, and replication realignment requires a spread (broadcast) communication operation. In general, redistribution requires all-to-all personalized communication.

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. Previously, we presented algorithms that efficiently determine each of these alignment parameters.(9, 10) 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, typical of a finite-difference stencil computation.

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

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 fragment. 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 than a cyclic distribution, since this reduces data traffic when shift communication is performed. It would be difficult to determine this fact about distribution without first having established alignment information.

2.2. The Alignment Distribution Graph

A data-parallel program may be modeled as a directed graph (V, E). The members of the vertex set V of the graph correspond to array operations 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. Only true dependences are modeled by edges.

A *weight* labels each edge; it is an estimate of the cost of the redistribution communication that would occur if the array carried by the edge is redistributed along it. In our system we compute the edge weight as a function of a constant communication overhead, the number of elements in the array carried 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 vertex, specifying the distribution that is applied to each of the arrays involved in the vertex computation. The graph, along with these labels, is called an alignment-distribution graph (ADG).(11) In our previous papers about the ADG it was important to distinguish between different vertex types for alignment analysis. Distribution analysis does not require such differentiation.

2.3. 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. Vertex 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, the machine topology, and other factors that might play a role) experiments have shown that this metric accurately reflects the cost of performing a redistribution step.[12] Furthermore, any edge that carries an axis or stride realignment has its weight changed to zero prior to distribution analysis, since redistribution of the value carried by the edge can be accomplished as part of the all-to-all personalized communication required for the realignment.

We seek to give each ADG vertex a distribution in D, i.e., we seek a mapping $m\colon V \rightarrow D$. For a particular distribution map, we estimate the execution time of the program it models by the sum over the vertices of the cost of performing the vertex computation in its 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 vertex to a distribution so that this cost is minimized.

The model's vertex cost component is trivially minimized by mapping each vertex to its distribution of smallest vertex cost, but this can result in many edges carrying redistribution communication. At the other extreme, edge costs may be avoided entirely by mapping every vertex to some one distribution, the best of these being the distribution that minimizes the sum of the vertex costs. The optimal solution typically lies at neither of these extremes. The conflict between reducing vertex costs (by labeling vertices independently) and eliminating edge costs (by labeling vertices identically) makes the problem difficult.

2.4. The Set of Distributions

A distribution $d \in D$ specifies both the deployment of processors to the axes of arrays 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. 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 never increased, and may be reduced, by allowing a larger set of candidate distributions, but the running time of our optimization algorithms is sensitive to the size of D. Thus we want a small set D that nevertheless includes those distributions that are best for the given program. We have previously shown how to select candidate distributions and how to limit their number.[3] To summarize these ideas, consider a data object of rank three, and let the number of processors be P. We allow distribution onto processors arrangements of shape (p_1, p_2, p_3) with each p_i allowed to be in the set $1, P^{1/3}, P^{1/2}, P$ subject to the constraint $p_1 p_2 p_3 = P$. In addition, we consider processor shape vectors that are proportional to the sizes of the large arrays in the program. In each dimension i for which $p_i > 1$ we allow either block or cyclic or cyclic(k) distribution, for a few values k. We choose the allowed distribution block sizes by examining the sizes of array sections in the given dimension, in order to determine the algorithm's "natural" block sizes.

2.5. Static and Dynamic Mappings

We introduce two terms to describe a distribution map for a subset of ADG vertices S. Let m be a given distribution map. Then S is *static* under m if m maps each member 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 vertices in a static subset have the same distribution, no edge internal to a static subset can carry redistribution cost; the cost of a static distribution is completely determined by the vertex costs.

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

The way in which we attempt to reduce the size of the graph is to identify *optimally static* (O.S.) subsets of vertices in 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 $\text{cost}(m') \leqslant \text{cost}(m)$.

An optimally static subset usually corresponds to a region of code, one in which it is provably not profitable to change distributions internally.

Our overall plan for finding a distribution map m is this. We first find a collection of optimally static subsets of V; we seek to cover V with the smallest possible number of them. Then, we aggregate each subset into a single "super" vertex whose cost vector is the sum of the cost vectors of its members. Clearly, any O.S. subset can be so contracted, as this does not increase the cost of the best distribution. We then use some heuristic or exact method to obtain a distribution mapping on the contracted graph, and take m to be its extension to the full graph.

The remainder of the paper is concerned with finding O.S. subsets. Our approach is to first find a candidate subset, and then test whether it is O.S. The next section develops a theory of O.S. subsets. Section 4 discusses heuristic strategies for finding candidate subsets.

3. SUFFICIENT CONDITIONS FOR OPTIMALLY STATIC SUBSETS

An understanding of properties of the distribution graph allows us to develop theorems that describe how subsets of vertices can be collapsed or amalgamated into super vertices, 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

In discussing whether or not a set S is O.S., we need to look at the largest and smallest entries of C_S, i.e., the largest and smallest aggregate vertex costs for S when it is distributed statically. Let

$$C_{\min}^{(\text{stat})}(S) \equiv \min_{d \in D} C_S(d)$$

and

$$C_{\max}^{(\text{stat})}(S) \equiv \max_{d \in D} C_S(d)$$

In contrast, we need to compare these with the smallest possible vertex cost total for S. Let

$$C_{\min}^{(\text{dyn})}(S) \equiv \sum_{s \in S} \min_{d \in D} C_s(d)$$

Last, let the difference between the maximum and minimum cost of a single vertex be called the *range* of the vertex,

$$\text{range}(v) \equiv \max_{d \in D} C_v(d) - \min_{d \in D} C_v(d)$$

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

$$w(S, T) \equiv \sum_{v \in S,\, w \in T} W(v, w) + W(w, v)$$

Thus, $w(S, \bar{S})$ is the sum of the weights of all edges entering or leaving S. We will commit the obvious abuse of using $w(s, T)$ instead of $w(\{s\}, T)$ for a single vertex s. Note that w is a symmetric function of its arguments.

3.2. Optimally Static Subsets

We present a number of tests that may be used to verify that a subset of vertices is O.S. Each of the next lemmas give 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*[3]). Let S be O.S. and assume $v \notin S$. If

$$w(v, \bar{S}) + \text{range}(v) \leqslant w(v, S)$$

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

Proof. Any map may, by assumption, be modified on S to make S static without increasing the cost of the map. Now consider a map in which S is static with distribution d and v has a different distribution d'. Changing the distribution of vertex v to d reduces the cost of the mapping by $w(v, S)$ and raises it by at most $w(v, \bar{S}) + \text{range}(v)$. By the hypotheses, this change also does not increase the cost. Hence $S \cup \{v\}$ is O.S. □

Corollary 1 (*Edge Contraction*). Let s and v be distinct vertices. The set $\{s, v\}$ is O.S. if $w(v, \{\bar{s}\}) + \text{range}(v) \leqslant w(v, s)$.

Proof. Since any singleton vertex is an O.S. subset, the corollary follows immediately by applying Lemma 1 to $S = \{s\}$. □

This simple corollary of Lemma 1 turns out to be very useful in practice: it identifies pairs of vertices 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*[3]). A set S is O.S. if

$$w(S, \bar{S}) + C_{\text{min}}^{(\text{stat})}(S) - C_{\text{min}}^{(\text{dyn})}(S)) \leqslant \text{mincut}(S)$$

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. For we gain at least mincut(S) in edge costs, and lose at most $C^{(\text{stat})}_{\min}(S) - C^{(\text{dyn})}_{\min}(S)$ in added vertex costs and at most $w(S, \bar{S})$ in additional redistribution on 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 S static and avoid redistribution on edges connecting it to that neighbor. These results are new to this paper.

Lemma 3 (*External Vertex*). A set S is O.S. if for some vertex $v \notin S$,

$$(w(S, \bar{S}) - w(S, v)) + (C^{(\text{stat})}_{\max}(S) - C^{(\text{dyn})}_{\min}(S)) \leqslant \text{mincut}(S) \qquad (1)$$

Proof. Let S be dynamic under some map m, and let v be connected to S by a set of edges of largest total weight. Remap all vertices in S to the distribution of vertex v. The vertex costs can increase, but not by more than the second term of the inequality (1); the edges from S except those touching v may now incur redistribution costs, but these added costs are not more than the first term of the inequality. Since, again, we gain at least mincut(S) in avoided redistribution on edges internal to S, the remapping cannot increase the total cost. □

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 is O.S. by considering remapping S to the same distribution as one of its own vertices.

Lemma 4 (*Internal Vertex*). A set S is O.S. if, for some vertex $v \in S$,

$$w(S - \{v\}, \bar{S}) + (C^{(\text{stat})}_{\max}(S - \{v\}) - C^{(\text{dyn})}_{\min}(S - \{v\})) \leqslant \text{mincut}(S) \qquad (2)$$

The proof is analogous to that of the External Vertex lemma.

4. LOCATING CANDIDATE SUBSETS

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

Our heuristic creates a series of partitions of the graph by deleting edges whose weight falls below a given threshold. Note that a single threshold value uniquely defines a partition of the graph. To generate a large number of candidate subsets, our heuristic examines all partitions defined by a set of thresholds, T.

The set of thresholds is generated by histogramming the edge weights of the graph and then gathering the histogram points into clusters. The minimum value in each cluster 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 the O.S. Lemmas are constructed and the way that the edge weights in the ADG are calculated. Lemmas 2–4 prefer sets that have no small edge cut. The heuristic finds subsets that are highly connected internally (leading to a large min-cut value), with low weight connections to vertices outside of the subset. For a given threshold value t, the mincut of any such component is not less than t, while the weight of each external edge is less than t.

The ADG tends to have clusters of heavy weight edges bordered by lighter weight edges. Recall that the ADG incorporates the effects of control flow into its edge weight calculation by multiplying the weight of an edge by its estimated trip count. In the ADG, vertices corresponding to operations within loops are connected by heavy edges, and values are communicated into and out of loops by lighter edges (because they are traversed only once). This strategy tends to find connected components encompassing the operations inside the bodies of loops, and the threshold values 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 $O(|E|)$, and connected components can be found in time $O(|E|)$ by using depth-first search. The enumeration of all subsets using this technique can be performed in time $O(|T||E|)$.

5. IMPLEMENTING THE O.S. TESTS AND THE CONTRACTION OPERATION

This section suggests data structures and algorithms for implementing the tests of Section 3. We make use of basic sparse matrix techniques to keep the running times of the O.S. tests and the contraction operations low.

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. 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 unit time insertion and deletion of single elements, and finding the neighbors of a given vertex. The data structure previously described supports these operations.

Our algorithms use a Sparse Accumulator (SPA) to add sparse vectors.(13) A SPA is used to compute the sum of several sparse vectors in time proportional to the number of nonzero elements in the vectors. This capability is important when contracting vertices.

5.2. Contracting Vertices

The vertex contraction operation replaces a set of vertices, S, with a single vertex s in a reduced graph. The vertex cost vector C_s of the new vertex is the sum of the vertex cost vectors of its members, and the weight of each edge incident to s is the sum of the weights of all edges between the adjacent vertex and members of S. Precisely, this is written as

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

Our technique contracts S by adding the sparse vectors that encode the edge weight and adjacency information for the vertices in S. Merging the cost table entries for the vertices 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 vertices of S. When merging row lists, we treat each row as a sparse vector and use the SPA to add the vectors, deleting the elements from the matrix as we go. We then enumerate the nonzero elements of the SPA and insert these new values in the contracted matrix. Column merging proceeds the same way. By using a SPA, the modification of the adjacency matrix requires time linear in the number of nonzeros processed.

5.3. Complexity Analysis

We now consider the overall complexity of the contraction algorithms we propose. These algorithms consist of the application of a sequence of transformations in some prespecified or adaptively chosen order, until some stopping criterion is satisfied. The three transformations we use follow.

Edge Contract Test all edges using Corollary 1.

Min-Cut Generate a set T of thresholds and for each, generate a set of subsets, as in Section 4. Apply the Min-Cut lemma to each subset and contract it if it is O.S.

Distinguished-Vertex Generate a set T of thresholds and for each, generate a set of subsets, as in Section 4. For each subset, and for each vertex either adjacent to or internal to the subset, apply the relevant Distinguished Vertex lemma and contract it if it is O.S.

A single pass refers to an application of the "Edge Contract" test to every edge in the graph, or an application of the "Min-Cut" and "Distinguished-Vertex" to all of the subgraphs of a partition of the graph defined by a single threshold value t. We have already shown that the modification of the data structures takes linear time; here we show that the application of the tests is efficient too.

We make the following assumption about our algorithm: The number $|T|$ of thresholds and the number $|D|$ of candidate distributions are bounded above by constants, and are not a function of graph size. For instance, $|D|$ is related to the complexity of the distribution requirements of a program. Some small programs may perform few arithmetic operations but possess complex distribution requirements, while larger programs may perform many more arithmetic operations with simple distribution requirements. See Table I for details.

Table I. Properties of the Program Graphs[a]

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

[a] Each is quite sparse. In general, the number of distributions used in the analysis of each program is small, with the exception of BlockLU.

5.3.1. Edge Contraction

Edge contraction is easy to implement. In order to facilitate it, we can store the range, the weight of all incident edges, and the weight of the heaviest incident edge in the vertex data structure. Then we can immediately tell whether a given vertex can be contracted into a neighbor. An important observation is that when a contraction occurs, vertices not adjacent to the two merged vertices are unaffected: if they could not be contracted into a neighbor before, they cannot after. Vertices adjacent to one of the merged vertices are likewise unaffected. Vertices adjacent to both *may* become contractable into the new vertex, and our implementation checks such vertices and contracts them in, if possible, immediately. Thus, we may examine the vertices, including those created by contraction, once each. When we finish, no more edges can be contracted. The total number of vertices we need to examine is therefore $O(|V|)$. The cost of the contraction of an edge is dominated by the cost of the addition of two rows and columns of W, which grows as the degree of the new vertex. Let B be the largest degree of any vertex created during the process; obviously $B < |V|$. The edge contraction algorithm runs in time $O(B \cdot |V|)$.

5.3.2. Min-cut Based Contraction

Application of the min-cut and the distinguished vertex lemmas to each subset S in a partition of the graph requires knowledge of the weight of edges leaving S, mincut(S), $C^{(\text{stat})}_{\text{min}}$, $C^{(\text{stat})}_{\text{max}}(S)$, and $C^{(\text{dyn})}_{\text{min}}(S)$. The weight of edges leaving each subset can be computed in a single pass over the graph in $O(|E|)$ time. Clearly, the sum of the minima and the minimum and maximum of the sum of the vertex costs for each subset can be computed in $O(|V| \cdot |D|)$ time. The difficult part is computing the mincut. There are two options: use an easily obtained lower bound on the mincut, or compute it exactly.

If S is connected, then mincut(S) is not less than than the minimum weight edge in S. We may find the lightest one by examining all edges internal to S. For a particular partition of the graph (defined by a threshold value t), we may determine the lightest edge of each subset in $O(|E|)$ time. With this simple lower bound on the min-cut, a single pass of the "Min-Cut" test takes $O(|V| \cdot |D| + |E|) = O(|V| + |E|)$ time.

In the second case, we compute the global min-cut of each set S exactly, using an algorithm of Goldberg and Tarjan which runs in $O(|S|^4)$ time.(14) 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 unacceptable for large sets.

5.3.3. Distinguished Vertex Tests

The application of the external vertex test requires finding the external vertex whose weight connecting it to a subgraph S in a partition of the graph is greater than that of any other vertex. (Computation of the other quantities is straightforward.) Using a SPA, for each subgraph S, we can compute the total weight with which it is connected to each external vertex. This operation is the same as the contraction step, except that we do not modify the matrix W. In doing this for each subset, each edge will be traversed at most twice: once from each vertex. Using the simple lower bound on the min-cut (as earlier), a single pass of the external vertex test takes $O(|V| + |E|)$ time.

The internal vertex test is similar to the external vertex test except that for each vertex $v \in S$ we compute $C_{\max}^{(\mathrm{stat})}(S - \{v\})$ and min C_v. This may be done in the following way to make the test efficient. Record the cost vector for the set, C_S. Now, as each vertex is visited, make use of the fact that $C_{\max}^{(\mathrm{stat})}(S - \{v\}) = \max(C_S - C_v)$ 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 external vertex test. Thus, with the simple lower bound on the min-cut a single pass of the internal vertex test takes $O(|V| + |E|)$ 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 as preliminary, pending better tools and a larger base of experimental programs.

Using program analysis tools we have developed earlier,[11] 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 the 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 I 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 algorithm. This 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 algorithm. This differs from ADI in that it uses Gaussian elimination to solve the tridiagonal systems.

LU: Unblocked *LU* factorization of a dense matrix.

Shallow: A benchmark weather prediction program; finite-difference approximation of 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 Jacobi over-relaxation method for the subdomain solver.

6.2. Cost Matrix Construction

In Section 2, we differentiated between three communication patterns: all-to-all personalized communication, 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, with all arithmetic operations requiring unit time per element. 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 all-to-all), 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 II shows the correspondence 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 vertex cost matrix, while redistribution costs affect adjacency information. Varying the parameters ρ the same

Table II. Mapping of High-Level HPF Operations to Low-Level Communication Types[a]

Coefficients of proportionality		
High-level operation	Low-level communication type	Constant
Redistribute	all-to-all	ρ
Stride realign	all-to-all	ρ
Axis realign	all-to-all	ρ
Offset realign	shift	ν
Replication realign	broadcast	σ
Subscript	all-to-all	ρ
Reduction	fan-in	σ

[a] 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.

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 matrix. 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 all, we experimented with 21 different combinations of the contraction operations. In the discussion of the combinations of contraction operations we use a shorthand. The character "e" means repeated application of edge contraction to all edges until the graph does not change. The character "*m*" stands for the min-cut test, and "*d*" for the distinguished vertex tests. By default, each of the "*m*" and "*d*" tests use the simple lower-bound on the min-cut value. We also ran these tests using the exact min-cut algorithm, but only for subsets whose size is smaller than a specified threshold. These thresholds are indicated by a number following the test combination, e.g. "eme:50." The contraction combinations examined are listed here:

```
m m:25 m:50 em em:25 em:50 eme eme:25 eme:50
d d:25 d:50 ed ed:25 ed:50 ede ede:25 ede:50
e ememe emedeme:50
```

6.4. Results

For each of the seven programs, we generated test data assuming a 64 processor target using these five sets of communication parameter values shown in Table III.

Case 1 reflects an 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, preferring dynamic distributions over static ones. Thus, this case should thwart many of our contraction operations.

Case 2 reflects an architecture where communication is only slightly expensive. Cases 3, 4, and 5 describe architectures with progressively more expensive communication. We expect that the cases with higher redistribution costs will encourage static solutions to the distribution problem, and thus expect our contraction operations to do well.

Rather than present tables containing all 735 data points, the contraction data is summarized in a scatter plot in Fig. 1. Each of the test program and communication parameter combinations appears along the X-axis, with test programs abbreviated by the first letter of their name. A single column of points illustrates the contraction obtained by all of the various contraction combinations tried. The amount of contraction achieved by combination "eme:50" is shown as a box, and "eme" is shown as a star. All other combinations are simply shown as a dot. From this graph it is clear that combination "eme:50" achieved the best contraction for almost all of the tests. For only the first two test programs of Case 5 did it not achieve the highest contraction rate.

The "eme" combination performs nearly as well as "eme:50" in many of the tests. Using the min-cut approximation ensures that the contraction tests run in linear time, and these results show that this crude approximation to the min-cut value is effective in practice. If run time is not a factor, however, then using the exact min-cut produces better results in a few of the few tests.

Table III. Communication Parameter Values Used for Generating Test Data

Case 1	$\rho = 1$	$\sigma = 1$	$\nu = 1$
Case 2	$\rho = 10$	$\sigma = 1$	$\nu = 1$
Case 3	$\rho = 100$	$\sigma = 1$	$\nu = 1$
Case 4	$\rho = 100$	$\sigma = 10$	$\nu = 1$
Case 5	$\rho = 1000$	$\sigma = 1$	$\nu = 1$

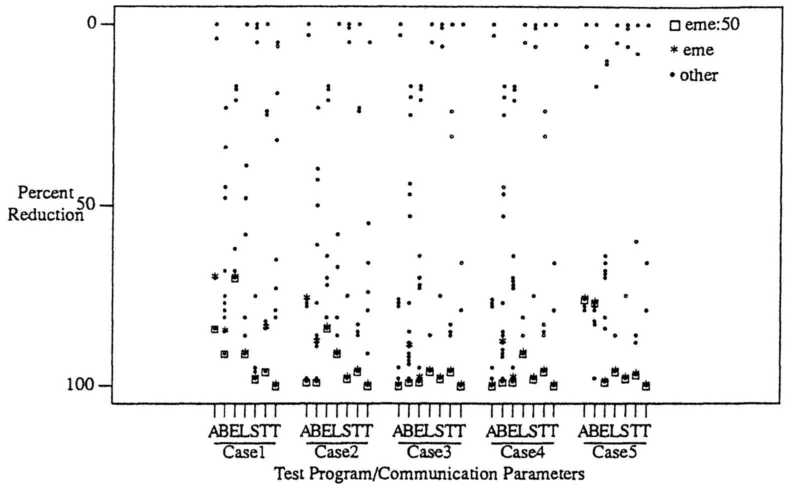

Fig. 1. Scatter plot of the contraction data. Combination "eme:50" is highlighted as a box, and "eme" appears as a star; all others are dots. Combination "eme:50" most consistently achieves the highest contraction rate.

The overall percentage of reduction achieved by the "eme:50" combination is shown in Table IV. Initially, we did not expect to see high contraction rates for Case 1 because redistribution is inexpensive and the low edge-weights lead to small min-cut values relative to the node costs. The results show that, on the contrary, the tests are effective even when p is small.

Table IV. The Percentage Contraction for the Combination "eme:50"[a]

	ADI	Block	Erle	LU	Shal	Tred	TwoZ
Case 1	83%	91%	70%	90%	98%	95%	99%
Case 2	99%	98%	83%	90%	98%	95%	99%
Case 3	99%	98%	99%	95%	98%	95%	99%
Case 4	99%	98%	99%	90%	98%	95%	99%
Case 5	75%	77%	99%	95%	98%	96%	99%

[a] This particular combination proved the most effective overall.

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. If one should relax the requirement that the contraction operations remain lossless—that is, subgraphs that are not necessarily O.S. are contracted anyway—what is the tradeoff 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. In the few cases in which the contracted graph remains large, is it because we haven't found the right subsets to test, or are our lemmas not powerful enough to prove that these subsets are indeed O.S?

8. SOFTWARE

Software implementing the graph contraction algorithms presented here is available from the authors, or at URL ftp://riacs.edu/pub/Excalibur/excalibur.html.

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, Texas (November 1993).
2. J. M. Anderson and M. S. Lam, Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. *Proc. ACM SIGPLAN '93 Conf. PLDI*, Albuquerque, New Mexico, pp. 112–125 (June 1993).
3. S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler, Array Distribution in Data-Parallel Programs. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, (eds)., *Proc. of the Seventh Ann. Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, Springer-Verlag, *Lecture Notes in Computer Science*, **892**:76–91, (August 1994). Also available as RIACS Technical Report 94.09.
4. J. Li and M. Chen, The Data Alignment Phase in Compiling Programs for Distributed-Memory Machines. *J. of Parallel and Distrib. Comput.* **13**(2):213–221 (October 1991).
5. S. Wholey, Automatic Data Mapping for Distributed-Memory Parallel Computers. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania (May 1991). Available as Technical Report CMU-CS-91-121.

6. M. Gupta, Automatic Data Partitioning on Distributed Memory Multicomputers. Ph.D. Thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois (September 1992). Available as Technical Reports UILU-ENG–92-2237 and CRHC–92–19.
7. D. J. Palermo and P. Banerjee, Automatic Selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers. Talk presented at the Workshop on Automatic Data Layout and Performance Prediction. Center for Research on Parallel Computing, Rice University. (April 1995).
8. U. Kremer, NP-Completeness of Dynamic Remapping. Technical Report CRPC-TR93-330-S, Center for Research on Parallel Computation, Rice University, August 1993. Appears in the *Proc. of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands (December 1993).
9. S. Chatterjee, J. R. Gilbert, and R. Schreiber, Mobile and Replicated Alignment of Arrays in Data-Parallel Programs. *Proc. of Supercomputing '93*, Portland, Oregon, pp. 420–429, (November 1993).
10. T. J. Sheffler, R. Schreiber, J. R. Gilbert, and S. Chatterjee, Aligning Parallel Arrays to Reduce Communication. *Proc. of Frontiers '95: The Fifth Symp. on the Frontiers of Massively Parallel Computation* pp. 324–331 (February 1995).
11. S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler, Modeling Data-Parallel programs with the Alignment-Distribution Graph. *J. Programming Languages*, **2**:227–258 (1994). Special issue on compiling and run-time issues for distributed address space machines.
12. P. Hough and T. J. Sheffler, A Performance Analysis of Collective Communication on the CM-5. Excalibur project meeting note.
13. 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).
14. A. V. Goldberg and R. E. Tarjan, A New Approach to the Maximum-Flow Problem. *J. ACM*, **35**(4):921–940 (October 1988).

Printed in Belgium