In *Proc. 2nd Symp. Frontiers Massively Parallel Computation* (1988), pp. 195–198.

Portable Parallel Algorithms for Geometric Problems

(Preliminary Version)

Russ Miller*
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260 USA

Quentin F. Stout†
Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122 USA

Abstract

Because the interconnection scheme among processors (or between processors and memory) significantly affects the running time, efficient parallel algorithms must take the interconnection scheme into account. This in turn entails tradeoffs between efficiency and portability among different architectures. Our goal is to develop algorithms that are portable among massively parallel fine grain architectures such as hypercubes, meshes, and pyramids, while yielding a fairly efficient implementation on each. Our approach is to utilize standardized operations such as prefix, broadcast, sort, compression, and crossproduct calculations. This paper describes an approach for designing efficient, portable algorithms and gives sample algorithms to solve some fundamental geometric problems. The difficulties of portability and efficiency for these geometric problems have been redirected into similar difficulties for the standardized operations. However, the cost of developing efficient implementations of them on the various target architectures can be amortized over numerous algorithms.

Keywords Portable parallel algorithms, computational geometry, data movement operations, distributed memory parallel computers.

1 Introduction

Massively parallel computers consisting of perhaps millions of processors are now becoming available. While such machines offer significantly faster solutions to many problems, they also impose severe programming requirements to utilize their potential. Old "dusty decks" do not typically work on such machines, and hence new algorithms and programs need to be developed. Since each processor contains only a small fraction of the total data, for most problems there must be extensive communication among processors. This communication often dominates the total running time of the program, and efficient programs must be developed with this in mind.

If the introduction of massive parallelism only brought about a one-time need to reprogram, then the reprogramming costs would at least be fairly well understood and for a variety of applications would be affordable. However, extensive additional costs are introduced due to the significant differences among massively parallel architectures. Different massively parallel architectures have significantly different communication characteristics, and hence have significantly different running times on the same programs. For example, on a square two-dimensional mesh with n processors, it takes $\Theta(\sqrt{n})$ time on average for two processors to exchange information, while on a hypercube or pyramid it takes $\Theta(\log n)$. For n processors to exchange data takes $\Theta(\sqrt{n})$ time on the mesh and pyramid, or $O(\log^2 n)$ time (worst-case) on the hypercube. Notice that in one case the pyramid is similar to the hypercube, while in the other case it is similar to the mesh. Due to such differences, for a single problem one may have two programs A and B with the property that on one massively parallel machine A is significantly faster than B, while on another massively parallel machine B is significantly faster than A.

This paper is concerned with developing algorithms which can be ported among different fine grain, massively parallel architectures and yield reasonably good implementations on each. Our approach is to write algorithms in terms of general data movement operations, and then implement the data movement operations on the target architecture. Efficient implementation of the data movement operations requires careful programming, but since the data movement operations form the foundation of many programs the cost of implementing them can be amortized. The use of data movement operations also helps programmers think in terms of higher-level programming units, in the same way the use of standard data structures helps programmers of serial computers.

*Partially supported by NSF grants DCR-8608640 and IRI-8800514.

†Partially supported by NSF grant DCR-8507851 and an Incentives for Excellence Award from Digital Equipment Corporation.

In Section 2 we give several data movement operations, and in Section 3 we illustrate our approach by giving some geometric algorithms written in terms of these data movement operations. Many data movement operations have been proposed, and the list is still growing as programmers acquire experience in parallel programming. Our list is intended as an illustrative sample, not an exhaustive collection. Further, the types of problems for which this approach is useful is quite large, and in this short paper we make no attempt to even survey such problems.

2 Data Movement Operations

A variety of data movement operations have been proposed for parallel computers. Often they originated as steps in the midst of some algorithm, and then later it was realized that they might have widespread utility. More recently there have been attempts to promote specific data movement operations as a programming aid [2, 3], or to develop a collection of data movement operations particularly useful for a specific architecture [5].

Several of the operations are defined in terms of some semigroup operation $\otimes$ over a set B, and our analyses of running time will assume that $\otimes$ can be computed in constant time. Sorting is a central operation, with several operations assuming that the data is already in sorted order. For such operations we assume that there is a linear ordering of the processors and a linear ordering of the set from which the items are chosen. Some operations are performed in parallel on disjoint consecutive sequences of items in sorted order, which are called *(ordered) intervals*.

Due to space limitations, we can give only a few of the proposed data movement operations. Two of the operations given below, namely, reducing a function and searching, originated with geometric problems, while the others have had somewhat wider usage. Interested readers might consult [2, 3, 5, 6, 7] for additional operations and extensive uses of the operations discussed here. Implementations of these operations for a variety of architectures appear in [7].

1. *Sort:* Given data distributed arbitrarily one per processor, order the data with respect to the processors.

2. *Merge:* Suppose that a set of data D is chosen from a linearly ordered set. Further, suppose D_1 is ordered one item per processor with respect to one subset of the processors, and D_2 is ordered one item per processor with respect to a disjoint subset of the processors, where $D = D_1 \cup D_2$. The merge operation combines D_1 and D_2 to yield D ordered with respect to the entire set of processors.

3. *Semigroup Computation:* Suppose each processor has a record with data from B and a label, and that the records form ordered intervals with respect to their label. Each processor ends up with the result of applying $\otimes$ to all data items with its label.

4. *Broadcast/Report:* Broadcast and report are often viewed and implemented as inverse operations. Both operations involve moving data within disjoint ordered intervals. They also both require a distinct processor, called the *leader*, of each interval. In broadcasting, the leader of each ordered interval delivers a piece of data to all other processors in its interval. In reporting, all processors within each interval have data from B, and $\otimes$ is applied to these items, with the result ending up at the leader. Often broadcast and report involve only a single interval. Some computer architects have proposed special hardware to implement "op-and-broadcast," which is our broadcast with a single interval and "op" equal to $\otimes$.

5. *Concurrent Read/Write:* In concurrent read and concurrent write we assume that there are master records indexed by unique keys. In the concurrent read each processor specifies a key and ends up with the data in the master record indexed by that key, if such a record exists, or else a flag indicating that there is no such record. In the concurrent write each processor specifies a key and a value from B, and each master record is updated by applying $\otimes$ to all values sent to it. (Master records are generated for all keys written). These concurrent read and concurrent write operations are extensions of the operations of concurrent read and concurrent operations normally associated with parallel random access machines (PRAMs). They model a PRAM with associative memory and a powerful combining operation for concurrent writes. On most distributed memory machines the time to perform these more powerful operations is within a multiplicative constant of the time needed to simulate the usual concurrent read and concurrent write, and the use of the more powerful operations can result in significant algorithmic simplifications and speedups.

6. *Compression:* Compression moves data into a region of the machine where optimal interprocessor communication is possible. For example, compressing k items in a square mesh will move them to a $\sqrt{k} \times \sqrt{k}$ subsquare, while compressing them in a mesh-of-trees with at least k^2 base processors moves them to the diagonal of a $k \times k$ subsquare.

7. *Searching:* Given a set of n processors, suppose every processor P_i contains *searching item* $s_i \in S$ and *target item* $t_i \in T$. Further, suppose there exists a Boolean

relation $R(s,t)$, $s \in S$, $t \in T$. The *searching operation* requires each processor P_i to find the largest t_j such that $R(s_i,t_j)$ is true. This really should be viewed as a class of data movement operations since for any machine there are significant differences in the times searching takes, based on the properties of R. For our purposes we can make the strong assumption that the items and R are such that $R(s,t)$ is monotone in each variable, and that S and T are stored in sorted order. In this case the searching operation can be accomplished through merging and broadcasting within intervals (see [7]).

8. *Parallel Prefix:* If processor P_i initially contains value a_i from B, then the *parallel prefix* computation results in P_i containing $a_1 \otimes a_2 \otimes \cdots \otimes a_i$. In [2] this operation is called a *scan*. Note that the hardware feature known as "fetch-and-op" implements a variant of parallel prefix, where "op" is $\otimes$ and the ordering of the processors is not required to be deterministic.

9. *Reducing a Function:* Given sets Q and R, let g be a function mapping $Q \times R$ into B. The map f from Q into B defined by $f(q) = \otimes\{g(q,r) \mid r \in R\}$ is the *reduction of* g, and in the reducing a function operation each processor starting with an element q of Q ends up with $f(q)$. For example, if Q and R are sets of planar points, g is distance, B is the reals, and $\otimes$ is minimum, then $f(q)$ is the minimum distance from q to any point in R.

The reader might note that several of these operations can be easily obtained from others, sometimes as special cases. However, each of these has proven useful, and sometimes the special cases can be implemented significantly faster than the general operation.

3 Sample Algorithms

Our illustrative algorithms involve finding special points from a collection of planar points. Given a finite set S of planar points, a point $p = (p_x, p_y)$ in S is a *maximal point* of S if $p_x > q_x$ or $p_y > q_y$ for every point $q \neq p$ in S. The *maximal point problem* is to determine all maximal points of a given set. See Figure 1. A point $p \in S$ is an *extreme point* of S if it is not in the convex hull of $S - \{p\}$, or, equivalently, if it is a corner (vertex) of the smallest convex polygon containing S. The *extreme point problem* is to determine all extreme points of a given set. See Figure 2. Readers interested in serial algorithms for these problems, and in the numerous applications of maximal points and extreme points, might consult [9].

In the following algorithms, n will denote the number of points. To simplify discussion, we will assume that the number of processors is also n. Extensions to cases where there are a few points per processor, rather than a single point per processor, are quite straightforward. In particular, we note that Thinking Machine's Connection Machine can be programmed using more virtual processors than real processors, and one is encouraged to write algorithms assuming a single point per virtual processor.

3.1 Maximal Points

Our first sample algorithm determines all maximal points, and was apparently first noted by Atallah and Goodrich [1].

Maximal Point Algorithm

1. Sort the n planar points so as to order them in reverse order by x-coordinate, with ties broken by reverse order by y-coordinate. That is, after sorting the points, they will be ordered so that if $i < j$ then either the x-coordinate of the point in processor P_i is greater than the x-coordinate of the point in processor P_j, or else the x-coordinates are the same and the y-coordinate of the point in processor P_i is greater than the y-coordinate of the point in processor P_j. Let (x_i, y_i) denote the coordinates of the point ending up in processor P_i.

2. Use parallel prefix, with $\otimes$ representing maximum and y_i as the data, to have each processor determine the largest y-coordinate stored in any processor of smaller index. Let L_i denote the value determined by processor P_i.

3. The point (x_i, y_i) is an extreme point if and only if $y_i > L_i$.

The running time of this algorithm, $T(n)$, is given by

$$T(n) = Sort(n) + Prefix(n) + O(1) \ ,$$

where $Sort(n)$ is the time to sort n items and $Prefix(n)$ is the time to perform parallel prefix. On all massively parallel architectures known to the authors, $Prefix(n) = O(Sort(n))$, and hence on such machines the time of the algorithm is $\Theta(Sort(n))$. Further, it is known that, at least for serial algorithms, determining maximal points is as hard as sorting [4]. Thus it appears that this portable algorithm is within a multiplicative factor of being optimal for all known massively parallel architectures.

3.2 Extreme Points

The following algorithm is based on the well-known tactic of using divide-and-conquer. To simplify exposition we assume

that no two points have the same x-coordinate. This assumption can easily be removed by including a few extra special cases in the algorithm.

Extreme Point Algorithm

1. *Preprocessing:* Sort the n planar points of the set S so as to order them by x-coordinate.
2. If $n \leq 2$ then all points are extreme points. Otherwise, note that if S_1 denotes the points in processors $0 \ldots (n/2) - 1$, and S_2 denotes the points in processors $(n/2) \ldots (n-1)$, then all points in S_1 have x-coordinates less than those of S_2. (We assume that processors $0 \ldots (n/2)-1$, and processors $(n/2) \ldots (n-1)$, form subsystems similar to the original machine. For example, in a hypercube we want the subsystems to be subcubes. On machines such as two-dimensional meshes or pyramids, one would subdivide into 4 pieces to achieve the proper subsystems.)
3. Recursively identify the extreme points of S_1 and the extreme points of S_2, enumerating them in counterclockwise fashion. This is a recursive call to step 2, not to step 1.
4. Identify the upper and lower common tangent lines between the extreme points of S_1 and the exteme points of S_2 by performing a searching operation. See Figure 3. This operation is performed by comparing the slopes of hull edges. Specifically, suppose $\overline{p_i q_j}$, $p_i \in S_1$, $q_j \in S_2$, is the upper tangent line between convex sets S_1 and S_2, as in Figure 3. Then it can be shown [8] that the slope of $\overline{p_i q_j}$ is between
   (a) the slope of $\overline{p_i p_{i-1}}$ and the slope of $\overline{p_{i+1} p_i}$, and
   (b) the slope of $\overline{q_{j-1} q_j}$ and the slope of $\overline{q_j q_{j+1}}$.

   Therefore, each extreme point simply needs to find the edges of the other set with slopes just above and just below the slopes of the edges it is incident on. Since the extreme points are kept in sorted order, this can be accomplished by merging with respect to the slopes of the edges and then performing broadcasts within intervals.
5. Eliminate all extreme points between the common tangent lines (i.e., all extreme points of S_1 and S_2 that are inside the quadrilateral formed by the four endpoints representing the common tangent lines) and renumber the remaining extreme points. This is accomplished by broadcasting the information pertaining to the four endpoints to all processors maintaining a point of S, and then having each processor make a constant time decision as to whether or not it remains an extreme point, and if so, what its new number is.

The running time of the algorithm is given by

$$T(n) = T'(n) + Sort(n) \ ,$$

where $T'(n)$ is the time to perform all but the first step. $T'(n)$ satisfies the recurrence

$$T'(n) = T'(n/2) + Search(n) + Broad(n) + Elim(1) \ ,$$

where $T'(n/2)$ is the time for the recursive call, $Search(n)$ is the time to perform the grouping operation to determine the upper and lower common tangent lines, $Broad(n)$ is the time to perform a broadcast operation on a machine of size n, and $Elim(1)$ is the time required for each processor to make the final extreme point decision.

On a d-dimensional mesh or a d-dimensional pyramid, this gives a total running time of $\Theta(n^{1/d})$, which is easily seen to be optimal. On a hypercube the running time is $\Theta(\log^2 n)$, since the time for $T'(n)$ is $\Theta(\log^2 n)$ and sorting can be completed in the same time by using bitonic sort. It is not known if this is worst-case optimal, since it is an open question as to whether a hypercube can sort in $o(\log^2 n)$ worst-case time. While we do not have space to explain the details, we note that one can modify the above algorithm so that it subdivides the original set into n^c pieces at each stage [6], with $0 < c < 1$. On the hypercube the modified algorithm achieves $T' = \Theta(\log n)$, which gives a total worst-case running time of $\Theta(Sort(n))$. This modified version also runs in $\Theta(\log n)$ time on an EREW PRAM.

4 Final Remarks

Data movement operations should be thought of as the parallel computing analogue of data structures in serial computers. Both provide higher level constructs which help programmers organize their thoughts and programs, and both allow programmers to reuse carefully optimized implementations. Initial users of parallel computers were often willing to spend considerable programming time to achieve the performance available through parallel processing, but as parallel computers move from research into practice there will be resistance to significant reprogramming for each new massively parallel architecture. Systematic use of data movement operations seems to provide a means of achieving high performance on future architectures without unending reprogramming.

References

[1] M. Atallah and M. Goodrich, "Efficient plane sweeping in parallel", *ACM Symp. Comp. Geo.*, 1986, pp. 216–225.

[2] G. Blelloch, "Scans as primitive parallel operations", *Proc. 1987 Int'l. Conf. Parallel Proc.*, pp. 355–362.

[3] C. Kruskal, L. Rudolph, and M. Snir, "The power of parallel prefix", *Proc. 1985 Int'l. Conf. Parallel Proc.*, pp. 180–185.

[4] H.T. Kung, F. Luccio and F.P. Preparata, "On finding the maxima of a set of vectors", *JACM* **22**, pp. 469–476.

[5] R. Miller and Q.F. Stout, "Data movement techniques for the pyramid computer", *SIAM J. Computing* **16**, pp. 38–60.

[6] R. Miller and Q.F. Stout, "Efficient parallel convex hull algorithms", *IEEE Trans. Computers* **C-37** (1988), pp. 1605–1618.

[7] R. Miller and Q.F. Stout, *Parallel Algorithms for Regular Architectures: Meshes and Pyramids*, The MIT Press, 1996.

[8] F.P. Preparata and S.J. Hong, "Convex hulls of finite sets of points in two and three dimensions", *Comm.ACM* **2**, pp. 87–93.

[9] F.P. Preparata and M.I. Shamos, *Computational Geometry*, Springer-Verlag, 1985.