10-22-1992

Building Segment Trees in Parallel

Peter Su
Dartmouth College

Scot Drysdale
Dartmouth College

BUILDING SEGMENT TREES IN PARALLEL

Peter Su
Scot Drysdale

Technical Report PCS-TR92-184

October,1992

Building Segment Trees in Parallel

Peter Su
Scot Drysdale
Dartmouth College

October 22, 1992

Abstract

The segment tree is a simple and important data structure in computational geometry [7, 11]. We present an experimental study of parallel algorithms for building segment trees. We analyze the algorithms in the context of both the PRAM (Parallel Random Access Machine) and hypercube architectures. In addition, we present performance data for implementations developed on the Connection Machine. We compare two different parallel algorithms, and we also compare our parallel algorithms to a good sequential algorithm for doing the same job. In this way, we evaluate the overall efficiency of our parallel methods.

Our performance results illustrate the problems involved in using popular machine models (PRAM) and analysis techniques (asymptotic efficiency) to predict the performance of parallel algorithms on real machines. We present two different analyses of our algorithms and show that neither is effective in predicting the actual performance numbers that we obtained.

1 The Segment Tree.

Let S be a finite set of edges in the plane. Set $n := |S|$, and let P be the set of endpoints of S. Assume we have sorted P by x-coordinate. Put infinitely long vertical lines down at each point in P. For each point $p \in P$, we define *strip*(p) to be the strip between the line at p and the first line to the left of p. If p is the leftmost point in P, then *strip*(p) is the half-plane to the left of the line at p.

To construct the segment tree, we first associate a tree node v with each point $p \in P$, and we define *strip*(v) to be *strip*(p). These nodes are the leaves of the segment tree. For each internal node u in the tree, we define *strip*(u) to be the union of the strips of the descendants of u.

We say that an edge $e \in S$ *covers* a strip in the plane if e intersects both the left and right boundaries of the strip. With each node v in the tree, we associate a set *cover*$(v) \subset S$. An edge $s \in S$ is in *cover*(v) if it covers *strip*(v) but not *strip*(*parent*(v)).

We store the edges of *cover*(v) in order by the "aboveness" relation. We say that an edge s is above another, t, with respect to a vertical line l if the intersection of s and l is above the intersection of t and l. We can order the edges in *cover*(v) with respect to any

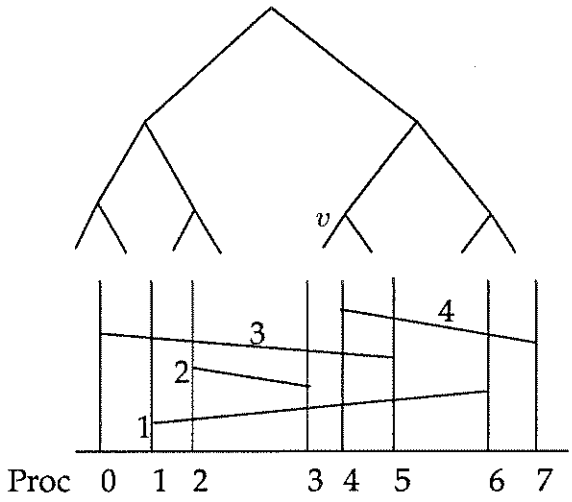

Figure 1: The segment tree.

vertical line between the boundaries of *strip*(v). If the edges do not intersect, this ordering will be consistent throughout *strip*(v), otherwise, it may not be.

We say that an edge *starts* in *strip*(v) if it has an endpoint in *strip*(v) but does not cover *strip*(v). We say than an edge e *crosses* the left or right boundary of *strip*(v) if the intersection of e, endpoints included, and the line defining the boundary is not empty. Define *left*(v) to be the set of edges which start in *strip*(v) and cross the its left boundary. Define *right*(v) to be the set of edges which start in *strip*(v) and cross its right boundary. The sets *left*(v) and *right*(v) can also be sorted by the aboveness relation, so we will assume that they are stored as sorted lists. For example, in figure 1, *left*(v) = $\{2\}$, *right*(v) = $\{4\}$, and *cover*(v) = $\{1,3\}$.

2 The Connection Machine.

The Connection Machine is a massively parallel fine grain SIMD architecture employing up to 64K simple processors, each with 64K bits of memory. Each processor in the Connection Machine has a one bit ALU, and all the processors are connected in a communications network that allows any processor to communicate with any other in the machine. In addition, every 32 processors share a floating point accelerator that can perform single precision calculations on behalf of the processors sharing it. The Connection Machine operates through a front end computer (in our case, a Sun-4) which sends instructions to it through a special interface. A microsequencer decodes the instructions, and then broadcasts the resulting instruction stream to the processors for execution [12].

One can program the Connection Machine at many levels, but the most widely used language is probably PARIS [10]. PARIS stands for "PARallel InStruction" set. The PARIS language is implemented as a library of subroutines which supplements a conventional language such as LISP or C. For our implementation work, we have been using LISP/PARIS.

PARIS allows the programmer to work with an abstract machine with a fixed number of "virtual processors." The system maps these virtual processors, or VPs onto the physical machine. The ratio of virtual processors to physical processors is called the VP ratio.

The abstract machine is called a "virtual processor set," or "VP set." One can work with multiple VP sets at once, so the programming model is relatively high level, and flexible.

Within a VP set, a programmer allocates parallel data structures using blocks of memory called *fields*. One copy of each field is allocated on every VP, so multiple copies may be allocated on each physical processor.

PARIS instructions resemble a high level assembly language, and specify operations to be performed by every processor in a VP set. There are instructions for arithmetic and logical operations, memory management, communication between processors, and communication between the Connection Machine and the front end computer. Communication between processors is supported by a hypercube network, and some sophisticated routing hardware and microcode. PARIS allows the programmer to specify either global communication operations, or specialized grid-like communication patterns. In addition, there are mechanisms in the instruction set for conditional execution, and local indirect addressing.

3 Primitive Operations.

We will describe our algorithms in terms of high level primitives which have efficient implementations on hypercube architectures in general and on the Connection Machine in particular [8, 13, 14, 17].

Elementwise Operations. The elementwise operators perform arithmetic and logical operations in each virtual processor of the machine in parallel. Such operations include unconditional and conditional assignments.

Permute. The permute operator takes a data field D and an index field π, and has each processor send the data it holds in D to the processor named by the value of π. This corresponds to an EREW PRAM write operation. In addition, we can define the equivalent of an EREW PRAM read operation by having each processor fetch data from the processor named in π. This takes two messages, so we'd like to try and avoid it.

Scan. The scan operator performs parallel prefix operations using several different operators. For example, we will use scan not only to calculate parallel prefix sums, but also do perform global broadcasts using the "copy" operator.

Pack. The pack operator takes a data field V and a flag field F and rearranges D so that all the data items that are marked "1" by F are moved to the lowest numbered processors in the VP set.

$$\begin{aligned} V &= [1\,2\,3\,4\,5\,6] \\ F &= [1\,0\,0\,0\,1\,1] \\ pack(V,F) &= [1\,5\,6] \end{aligned}$$

Bitonic Merge. Here we are given a field D which is arranged to be in bitonic order, that is, there is a block of processors holding data which is sorted, and then another block of processors holding data which is sorted in reverse order. After calling bitonic merge, the data in D is sorted [6].

Segmented Primitives. Each of our primitives can also work on "segmented" fields. A segmented field is a field that is logically partitioned into contiguous blocks of virtual processors. The segmented primitives work on each such block independently, in parallel.

For example:

$$\begin{aligned} V &= [[1\,2\,3][4\,5\,6]] \\ \textit{seg-scan-plus}(V) &= [[1\,3\,6][4\,9\,15]] \end{aligned}$$

The segmented primitives can all be implemented $O(1)$ calls to normal primitives [8].

4 The Algorithms.

We studied two algorithms for building segment trees in parallel. One, "Algorithm S" is based on sorting, while the other "Algorithm M" is based on merging.

4.1 Algorithm S. A straightforward parallel algorithm for building the segment tree was first presented by Aggarwal [2]. The algorithm works in three steps:

1. Sort the edge set by the x-coordinate of the endpoints of the edges. The sorted edge set forms the skeleton of the final tree.

2. Have each edge perform a binary search through the skeleton tree to determine what nodes in the tree it covers.

3. Sort the resulting cover sets in parallel.

This algorithm has the advantage that it is simple to implement and optimize for the Connection Machine. Given the edge set S, we first define a VP set with $2|S|$ virtual processors. The edges are stored in a field e which is 160 bits long and stores five 32 bit floating point values:

x_1	y_1	x_2	y_2	*key*

We store two copies of each edge, one with $key = x_1$ and one with $key = x_2$, and sort e by the value in *key*. The sort step places all the edges in their correct positions at the leaves of the tree. An edge can cover either child of each node that lies on the path from its leaf node to the root of the tree. In order for the edges to find the nodes they cover, we do the following at each level of the tree:

1. Each node v with children x and y broadcasts the boundaries of *strip*(x) and *strip*(y) to the leaves of the subtrees rooted at x and y.

2. Each edge looks at the two sets of boundary information and checks to see if it belongs in *cover*(x) or *cover*(y). It cannot belong to both since if it did it would cover v. If it belongs to either, we record this information as a triple $(node, key, edge)$ where *node* is used to identify the tree node that an edge belongs in, *key* is used to order the edges by "aboveness", and *edge* it used to identify the edge so we don't have to copy that data into the node itself. In our implementation, we order the edges by where they cross the right boundary of a strip.

3. Now sort the triples that we recorded in the previous step using *node* as the primary key, and *key* as the secondary key.

We represent the tree itself as an array of segmented fields. Each element of the array corresponds to a level of the segment tree. The field representing level i of the tree is

partitioned into segments that are 2^i elements long. To perform the broadcast step, we just do a series of segmented scans which distribute the data into the correct blocks of processors.

Step 2 of the algorithm requires no communication, it is just a series of arithmetic and logical operations executed by each VP independently. The third step of the inner loop takes most of the time. We were able to get many different levels of performance from the algorithm by using different sorting algorithms. We used bitonic sort, parallel radix sort, and a variant of Reif and Valiant's *flashsort* called sample sort [16, 9]. We present a rough analysis of these algorithms below. For a more detailed look at sorting on the Connection Machine, the reader may refer to a recent paper by Blelloch, *et al* [9].

4.2 Analysis. Since the algorithm was originally designed in the PRAM model, we will analyze it for both the PRAM and the Connection Machine architecture. Comparing both sets of analysis with our actual performance numbers will illustrate some of the problems with designing algorithms in the PRAM model and using asymptotic analysis to estimate real performance. In the final paper, we will present a more detailed analysis for the Connection Machine which will better reflect the performance which we really obtained.

In the PRAM model, the above algorithm has an asymptotic run time of $O(\log^2 n)$, because the sort steps take $O(\log n)$ time each, and there are $O(\log n)$ of them. However, none of the $O(\log n)$ time PRAM sorting algorithms have reasonable constants. The most "practical" PRAM sorting algorithm is mergesort, which runs in $O(\log^2 n)$ time with a straightforward merge and $O(\log n \log\log n)$ time with an esoteric merge. Thus, we expect that the run time of algorithm S would be between $O(\log^2 n)$ and $(\log^3 n)$ depending on how good our sorting algorithm is.

To analyze algorithm S on the Connection Machine, we just have to analyze the three sorting algorithms that we used. Assume that we have a CM with p physical processors. We will analyze bitonic sort first. The CM implementation of bitonic sort uses special microcode that allows communication over all $\log p$ dimensions of the hypercube at the same time. This lets the CM implementation pipeline the merge steps to achieve an actual run time of $O((n/p) + \log p)$ for the "off chip" part of bitonic merge, that is, the part of the merge step that requires interprocessor communication. The rest of the merge step takes $O((n/p)\log n)$ time, so the whole bitonic sort algorithm uses $O((n/p)\log p + \log^2 p + (n/p)\log^2 n)$ time. Combining these expressions, the total run time for algorithm S using bitonic sort on the Connection Machine should be

$$O\left(\log n((n/p)\log p + \log^2 p + (n/p)\log^2 n)\right).$$

Parallel radix sort runs in $O((k/r)(n/p)\log p)$ time, where k is the length of the key, and we bucket the keys using blocks of r bits at a time. For more details on this algorithm, the reader is referred to the paper by Blelloch [9]. With this run time, algorithm S using radix sort should run in $O((k/r)(n/p)\log p \log n)$ time. Depending on the values of k and r, we might expect that algorithm S using radix sort would be about as fast as algorithm S using bitonic sort. However, note that using larger values of r the reduces amount of communication that radix sort must perform. Thus, on large data sets we can use larger values of r to improve the performance of the radix sort.

Sample sort consists of three stages [9]:

1. In the first stage, each processor chooses a random key as its "splitter." The splitters are sorted, and the splitter set is distributed throughout the machine. This takes $O(\log^2 p)$ time using the normal parallel radix sort.

2. In the second stage, each key does a binary search through the splitter set to figure out which processor it belongs in. The keys are then routed to their destination processors. This takes $O((n/p)\log p)$ time, assuming that it takes $O(\log p)$ steps to route p packets. If any processor runs out of memory, we have to repeat stages 1 and 2 until we achieve a good load balance.

3. Now each processor sorts the set of keys that it received. On average, this takes $O((n \log n)/p)$ time.

Thus, we expect that the sample sort algorithm runs in $O(\log^2 p + (n/p)\log p + (n \log n)/p)$ time. Given this, the version of algorithm S based on sample sort should have a run time of $O(\log n(\log^2 p + (n/p)\log p + (n \log n)/p))$

Our analysis shows that algorithm S, in all of its forms, is extremely inefficient. Given the fact that the best sequential algorithm for building a segment tree takes $O(n \log n)$ time, our parallel algorithm is a factor of $O(\log n)$ off what we should be able to achieve. Thus, we would expect that the algorithm would not perform as well as we would like when compared to a sequential algorithm for doing the same job.

4.3 Algorithm M. If the edge set S does not contain any intersecting edges, then we can use the work of Goodrich, Attallah and Cole to build the segment tree [4]. Our algorithm depends on the following fact.

FACT 4.1. *Given a node v with sibling w and children x and y. Let $sl(y)$ denote the set of edges which belong to $left(y)$ but do not cross the left boundary of $strip(x)$. Let $sr(x)$ denote the set of edges in $r(x)$ that don't cross the right boundary of $strip(y)$. Then*

$$\begin{aligned} left(v) &= (left(x) \cup left(y)) - sl(y) \\ right(v) &= (right(x) \cup right(y)) - sr(x) \end{aligned}$$

This fact shows us how to build the *left* and *right* sets bottom up. Using the formula above, we can build $left(v)$, $right(v)$, $left(w)$ and $right(w)$. Using this information, we can build the *cover* sets using the following formula:

$$cover(v) = \begin{cases} left(w) - right(v) & \text{if } v \text{ is a left child} \\ right(w) - left(v) & \text{otherwise.} \end{cases}$$

Using this formula, we can build an algorithm based on a merging step at each level of the tree rather than a sorting step at each level of the tree. The basic idea behind the algorithm is to "purge and merge." Starting with $left(x)$ and $left(y)$, we remove edges which are too short to belong to $left(v)$ and then perform a merge step to build $left(v)$. A symmetric process builds $right(v)$. Then, to build $cover(v)$, we examine either the *left* or *right* set of the sibling of v and remove any edges which do not belong. The algorithm that Goodrich, Cole and Attallah present achieves $O(\log n)$ run time on an $O(n)$ processor

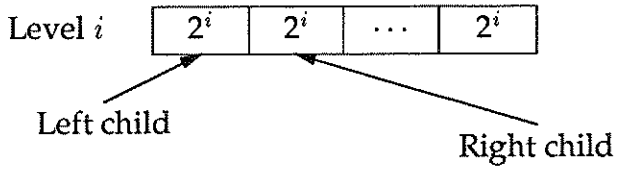

Figure 2: Representation of the tree.

PRAM. However, their algorithm is too complicated to implement. On the Connection Machine, we use bitonic merge as a primitive to construct our algorithm [6].

We again represent the tree as an array of segmented fields. At level i of the tree, each node is stored in a block that is 2^i virtual processors long. This representation now becomes important, since we have to have the data aligned this way to take advantage of bitonic merge. Also, note that in this representation, the even blocks of processors represent left children in the tree, while the odd blocks represent right children (see figure 2).

In order to implement the merge step, we assume we have three fields, l, r, and c that are holding the elements of the *left*, *right* and *cover* sets for level $i-1$. To build the new *cover* sets, we execute the following steps:

1. Broadcast the boundaries of the strips at level i of the tree using a segmented copy scan as in algorithm S.

2. Have each edge decide whether it belongs in l or r at level i. Each edge compares the boundary information with its left and right endpoints. We mark edges which do not belong for deletion. Here, we are computing the sets $sl(y)$ and $sr(x)$.

3. Delete any edges which do not belong. We can do this using *pack*, but it is faster to simply use another segmented copy scan to overwrite edges which do not belong with copies of edges that do belong. Goodrich calls this "changing the identity" of the deleted edges [4]. Note that we have to be careful about edges at segment boundaries. In the following example, the edge b has to be prevented from being copied into the high segment. Also, the edge a will not be overwritten because there is no edge before that is marked as "good." Therefore, the result of the copy-scan, R is not right.

$$\begin{aligned} E &= [a\,2\,b\,4\,|\,5\,3\,2\,1] \\ F &= [0\,1\,1\,0\,|\,0\,0\,0\,1] \\ R &= [a\,2\,b\,b\,|\,b\,b\,1\,1] \end{aligned}$$

In order to avoid these problems, it suffices to replace inactive boundary edges with dummy values that will not change any answers. For example:

$$\begin{aligned} E &= [a\,2\,b\,4\,|\,5\,3\,2\,1] \\ E' &= [-\infty\,2\,b\,4\,|\,\infty\,3\,2\,1] \\ F &= [0\,1\,1\,0\,|\,0\,0\,0\,1] \\ R &= [-\infty\,2\,b\,b\,|\,\infty\,\infty\,1\,1] \end{aligned}$$

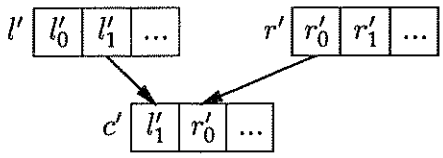

Figure 3: Copying edges to c'.

The choice of $-\infty$ versus ∞ is important because we have to preserve the bitonic ordering of the edges in order for our merge steps to work correctly.

4. Have each edge in l and r compute a new merge key based on the strip information. Recall that we merge the edges based on where they intersect a vertical line passing through each strip. We use the left boundary line of a strip for the *left* sets and the right boundary for the *right* sets.

5. Call bitonic merge to merge the even and odd blocks of 2^{i-1} processors in l and r. The result of this merge is the *left* and *right* sets for level i.

6. Next, we compute the result of the second formula in fact 1. Call the results of the merge operations l' and r'. First, we copy the odd segments of l' to the even segments of c', and the even segments of r' to the odd segments of c' (see figure 3).

   The copy step swaps the *left* and *right* sets of the left and right children. If v is a left child, then any edge in the *right* set of v's sibling which also crosses the left boundary of *strip*(v) is in *cover*(v). Similarly, if v is a right child, we check the *left* set of its sibling. We check these conditions directly by comparing the endpoints of the edges in the *left* and *right* sets with the strip boundaries. We then change the identity of edges which do not belong.

4.4 Analysis. In the PRAM model, this algorithm runs in $O(\log^2 n)$ time, since the merge steps each take $O(\log n)$ time to perform. With some improvements, one can reduce the run time to $O(\log n \log\log n)$ [5]. With even more sweat, one can obtain a run time of $O(\log n)$ on $O(n)$ processors [4].

On the Connection Machine, we used the merge steps from the bitonic sort code discussed earlier as a subroutine. Each merge step consists of a constant number of scans, permutes, and calls to bitonic merge. Thus, the run time of algorithm M on the CM is proportional to the run time of bitonic sort, namely $O((n/p)\log p + \log^2 p + (n/p)\log^2 n)$ steps.

Our analysis indicates that algorithm M should have an advantage over algorithm S since we are performing $O(\log n)$ merge steps rather than $O(\log n)$ sort steps.

5 Performance Results.

We implemented both algorithms on the Connection Machine in LISP/PARIS. In addition, we implemented a C version of algorithm M and ran tests on a DECstation 5000. We tested the algorithms on randomly generated sets of edges. For algorithm S, we just picked endpoints uniformly from the box $(0,1) \times (0,1)$. For algorithm M, we needed

sets of non-intersecting edges. We picked random horizontal edges by choosing three numbers x_1, y and x_2 uniformly from the interval $(0, 1)$. The edge was then defined to be the horizontal line segment between (x_1, y) and (x_2, y). Depending on the machine and the algorithm, we ran timings for problem sizes of between 4K and 512K endpoints. We averaged ten runs of each algorithm at each problem size to produce the data points shown in the graph.

5.1 Comparing the Parallel Algorithms. Figure 4 shows the performance of our parallel implementations. We ran our parallel timings on a Connection Machine 2 with 8K bit serial processors and 256 floating point units. The graphs show how the run time (in seconds) varies with the VP ratio. Thus, a VP ratio of one means that we ran with 8K endpoints while a VP ratio of 64 means we ran with 512K endpoints. The graphs show run time per virtual processor per physical processor. To compute the actual run time, multiply the value shown by the VP ratio. To run the large problems, we did not store the entire tree in memory at once, only one level of it. The strange looking data points in the figure 4 reflect the fact that the sample sort library routine does not actually use sample sort until the VP ratio is eight (8) or higher. Thus, at low VP ratios, this implementation runs slowly because it is using the standard library sorting routine but with extra overhead.

The graphs show that algorithm M is about two times faster than algorithm S using bitonic sort, and is slower than algorithm S on larger problems when we used radix and sample sorting! The reason for this is that both radix and sample sorting use local computation time to reduce the amount of communication that they must perform. For large problems, this gives them an advantage over bitonic sort and algorithm M, which both have high communication costs. We should note that there is one phase of algorithm M which we did not implement as well as is possible. In the final step of the merging process, we make two calls to the router which we could replace with direct hypercube communication. At VP ratio 64, this would save us approximately 8-10 seconds out of a total run time of 40 seconds, which is significant. However, even with this savings, algorithm M would not run much faster than algorithm S.

Figure 5 shows the time spent by algorithm M in each merge step at VP ratio 64. What the graph shows is that the iterations which must perform interprocessor communication are much more expensive than the iterations which do not. For large problem sizes, this communication cost will dominate the run time of the algorithm. Thus, the performance of algorithm M is limited by the global memory bandwidth of the machine.

This performance characteristic holds for any algorithm which uses the "split and merge" paradigm for solving problems by divide and conquer. For example, our performance numbers also show that bitonic sort, which uses split and merge, is much less efficient for large problems than sample sort, which does not.

The key point to note here is that the current style of asymptotic analysis was not exact enough to predict these problems. Thus, while our intuition and our analysis told us that an algorithm based on merging had to be faster than an algorithm based on sorting, this in fact was not the case. If we had analyzed the algorithms using a more accurate machine model, we may not have run into these problems.

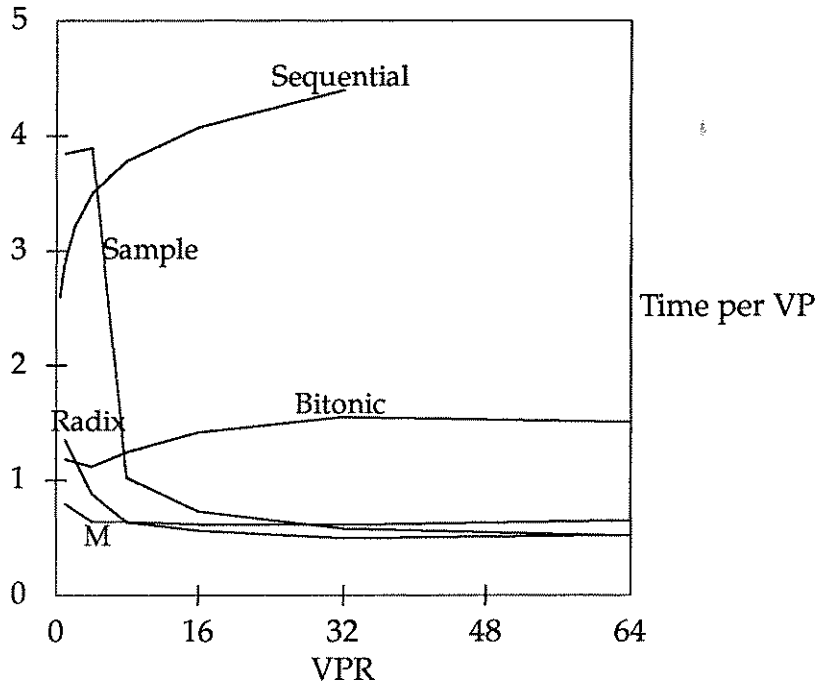

Figure 4: Run times for the various algorithms.

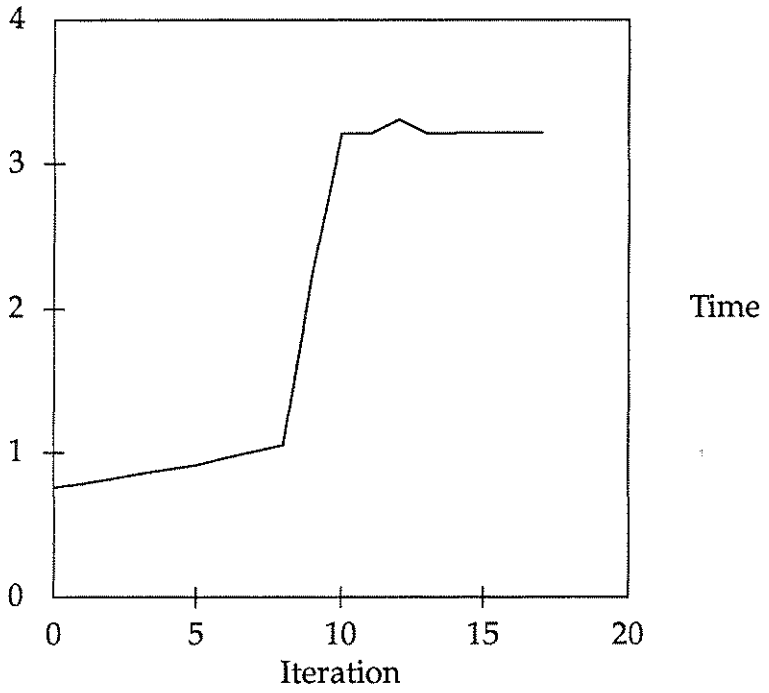

Figure 5: Iteration by iteration run times (secs) for algorithm M.

5.2 Killer Micros. In order to gain real perspective on how efficient our parallel algorithms are, we compared them to a good sequential implementation of algorithm M. Our theoretical analysis indicated that the parallel algorithms were doing more work than the sequential ones, and were thus less efficient. The run times we obtained support this conclusion.

Figure 4 also shows the run times of the C version of algorithm M running on a DECstation 5000. The axis labeled "VPR" represents what the VP ratio would have been if we had run the same size problem on the 8K Connection Machine. This makes the graphs easier to relate to each other. It is immediately clear that none of our algorithms are making effective use of the Connection Machine, since the fastest parallel algorithm is only eight times faster than a simple implementation in C. From a cost/performance standpoint, the parallel algorithms are dismal. In addition, the parallel code took much longer to develop and test than the C code, which is more understandable and maintainable.

6 Thoughts and Conclusions.

We have presented two practical algorithms for building segment trees in parallel on the Connection Machine. One algorithm is simple and based on sorting, the other is more complex and based on merging. The surprising fact is that the simple algorithm is just as fast as the complex one. Unfortunately, neither algorithm compares well (in terms of cost/performance) to a good sequential algorithm running on a conventional processor.

Our results lead us to many more questions than answers:

1. The most obvious question is whether or not we can find an even better algorithm for building segment trees. In particular, we would like to examine Reif's randomized algorithm for building the plane sweep tree [15]. This algorithm is interesting to study precisely because, like sample sort, it is not based on split and merge but uses parallel bucketing instead.

2. Can we show that, in practice, geometric algorithms based on bucketing rather than "split and merge" will work better? To do this, we need to develop the tools necessary to implement more sophisticated algorithms on the CM, and perhaps other architectures as well. These tools include compilers, libraries, profilers, debugging aids, and visualization tools.

3. What is the right way to model large scale machines? In his paper on sorting, Blelloch uses a machine model that closely matches the Connection Machine [9]. This makes his analysis more exact, but it may be difficult to relate his results to other architectures, especially those that are not hypercubes. On sequential computers, our machine models are fairly accurate even though they do not use many architecture-dependent parameters. Is it possible to develop such models for parallel machines?

   In addition, can we develop models of computation that capture the fact that global data movement is an expensive operation on parallel computers? Some recent work on modeling communication and hierarchical memory systems is a start in this direction [1, 3].

In general, more work needs to be done in examining the interaction between the algorithms we design, the models we use to design them, and the systems (architectures, compilers, run time systems) we try to implement them on.

7 Acknowledgements

None of this would have been possible without the use of the Connection Machine Network Service, made available to us by Thinking Machines Corporation through the Connection Machine Network Server Pilot Facility, supported under terms of DARPA contract number DACA76-88-C-0012. In addition, the code never would have worked without the help of the technical staff at Thinking Machines, who handled my never ending stream of problems and questions. In particular, Steve Smith provided the code for all the sorting algorithms that I used, and explained how it all worked.

In addition, I would like to thank the Institute for Advanced Computer Studies Parallel Processing Lab at the University of Maryland in College Park, and its director, Dr. Larry S. Davis. They provided us with access to an additional machine which we used to take the timings presented in this paper.

Lastly, Mike Goodrich showed me the right way to implement the merge step of algorithm M. His insight sped the code up by a factor of two.

References

[1] A. Aggarwal, A. Chandra, and M. Snir. Communication Complexity of PRAMs. *Theoretical Computer Science*, 17:3–28, 1990.

[2] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel Computational Geometry. *Algorithmica*, 3:293–327, 1988.

[3] B. Alpern, L. Carter, and E. Geig. Uniform Memory Hierarchies. *ACM Symposium On Theory Of Computing*, pages 600–608, 1990.

[4] M. J. Attallah, R. Cole, and M. T. Goodrich. Cascading Divide and Conquer: A Technique for Designing Parallel Algorithms. *SIAM Journal On Computing*, 18(3):499–532, 1989.

[5] M. J. Attallah and M. T. Goodrich. Efficient Plane Sweeping in Parallel. *ACM Conference on Computational Geometry*, pages 216–225, 1986.

[6] K. Batcher. Sorting Networks and their Applications. *Proc. AFIPS SJCC*, 32:307–314, 1968.

[7] J. Bentley and M. Shamos. A Problem in Multivariate Statistics: Algorithms, Data Structure, and Applications. *Annual Allerton Conference on Communication, Control, and Computing*, pages 193–201, 1977.

[8] G. Blelloch. Scans as Primitive Parallel Operators. *IEEE Transactions On Computers*, 38(11):1526–1538, 1989.

[9] G. Blelloch, C. Leiserson, B. Maggs, G. Plaxton, S. Smith, and M. Zagha. A Comparison of Sorting Algorithms for the Connection Machine. *Annual ACM Symposium on Parallel Algorithms and Architectures*, 1991.

[10] Thinking Machines Corporation. Paris Reference Manual. Technical report, 1989.

[11] F. Dehne and A. Rau-Chaplin. Implementing Data Structures on a Hypercube Multiprocessor, and Applications in Parallel Computational Geometry. *Journal Of Parallel And Distributed Computing*, 8:367–375, 1990.

[12] D. Douglas, B. Kahle, and A. Vasilevsky. The Architecture of the CM-2 Data Processor. Technical report, 1988.

[13] R. Miller and Q. F. Stout. Portable Parallel Algorithms for Geometric Problems. *Second Symposium on the Frontiers of Massively Parallel Computation*, pages 195–198, 1988.

[14] D. Nassimi and S. Sahni. Bitonic Sort on a Mesh-Connected Computer. *IEEE Transactions On Computers*, 27(1):1–7, 1979.

[15] J. H. Reif and S. Sen. Optimal Randomized Parallel Algorithms for Computational Geometry. *International Conference On Parallel Processing*, pages 270–277, 1989.

[16] J. H. Reif and L. G. Valiant. A Logarithmic Time Sort for Linear Size Networks. *Journal of the Association For Computing Machinery*, 34(1):60–76, 1987.

[17] I. Stojmenovic. Computational Geometry on a Hypercube. *International Conference On Parallel Processing*, pages 100–103, 1986.