Experimental Evidence For The Power of Random Sampling in Practical Parallel Algorithms
(Preliminary Version)

Mujtaba Ghouse* [†]
Dept. of Computer Science, The Johns Hopkins Univ., Baltimore, MD 21218

Michael T. Goodrich[†] [‡]
Dept. of Computer Science, The Johns Hopkins Univ., Baltimore, MD 21218

Abstract

Recent results in parallel algorithm theory have shown random sampling to be a powerful technique for achieving efficient bounds on the expected asymptotic running time of parallel algorithms for a number of important problems. In this paper we show experimentally that randomization is also a powerful *practical* technique in the design and implementation of parallel algorithms. Specifically, we show that random sampling can be used to design parallel algorithms with fast expected run times, which meet or beat the run times of methods based on more conventional methods for a variety of benchmark tests. The constant factors of proportionality in the run times are small, and, most importantly, the expected work (and hence running time) avoids worst cases due to input distribution. We justify our approach through experimental results obtained on a Connection Machine CM-2 for a specific problem, namely, segment intersection reporting, and explore the effect of varying the parameters of our method.

1 Introduction

A fundamental goal of algorithmic research is to develop general paradigms for designing efficient algorithms. Ideally, such paradigms should be generic enough to apply to many different types of problems, yet powerful enough to result in methods that are significantly better than what could be achieved without these paradigms. One such paradigm that has recently emerged from research in algorithmic theory is the *random sampling* technique [5]. Simply put, the random sampling technique is based on the notion of selecting a relatively small random sample of the input, performing an inefficient (often "brute force") procedure on the sample, and using the information from that procedure to subdivide the input and recurse.

On an intuitive level, the primary reason the random sampling paradigm has lead to efficient algorithms is that it allows one to call a simple, inefficient method on a small subproblem, and then rely on a probablistic analysis to show that the solution to this subproblem divides the input into recursive calls whose expected size is also small. Moreover, since it is the algorithm that is making random choices, these expectations hold independent of any assumptions about the input distribution. The classic example is the randomized quicksort algorithm (e.g. see [9]), which runs in expected $O(n \log n)$ time regardless of input distribution, whereas the expected $O(n \log n)$ running time of the standard quicksort assumes a uniform input distribution. Indeed, the standard quicksort algorithm will run in $\Theta(n^2)$ time given inputs taken from a distribution of "almost sorted" lists, while randomized quicksort still runs in $O(n \log n)$ expected time for such distributions.

1.1 Previous Work

Random sampling has been used to achieve efficient bounds for the expected asymptotic running time for solving a host of different problems. It was first developed in the form we study in this paper by Clarkson [5], and has subsequently been used by a number of other researchers to design algorithms

[†]This research supported by the NSF and DARPA under Grant CCR-8908092, and by the NSF under Grant IRI-9116843.

[‡]This research is also supported by the NSF under Grants CCR-9003299 and CDA-9015667, and by The Bureau of the Census under Contract JSA-91-23.

with efficient asymptotic expected running times (e.g., see [2, 6, 14, 15, 16, 17, 7, 8, 11, 19, 21]). This approach has also been used to design parallel algorithms with efficient expected asymptotic running times and efficient expected asymptotic processor bounds. Examples of this include methods by Alon and Megiddo [2], Reif and Sen[19], Clarkson, Cole, and Tarjan [7], and the authors [11]. Indeed, optimal randomized parallel methods already exist for the specific problem we address: Clarkson, Cole and Tarjan [7] give an algorithm for segment intersection that runs in expected $O(\log n)$ time with $O(n \log n + k)$ expected work. Unfortunately, this methods is also quite complicated.

We know of no previous work showing the practical importance of this type of random sampling. There is therefore a natural question that we feel immediately follows from these recent advances in algorithmic theory:

Does random sampling also lead to efficient algorithms in practice?

1.2 Our Results

In this paper we show that random sampling is in fact a powerful practical technique in the design and implementation of parallel algorithms. Specifically, we show that random sampling can be used to design parallel algorithms with fast expected run times, which meet or beat the run times of parallel algorithms based on more conventional methods in benchmark tests. The constant factors of proportionality in our run times are small, and, most importantly, the expected running times are independent of any assumptions about input distributions. Our work bounds are also quite small, but are asymptotically worse than that achieved by previous, more theoretical methods, such as those of Clarkson, Cole and Tarjan [7].

Our initial approach is based on two simple ingredients: (1) the existence of a simple, fast, processor-*inefficient* method, and (2) random sampling. The main idea is to take a random sample of size r (which we specify in the analysis of our method) and apply the fast method to this sample. Assuming that r is sufficiently small, we will be guaranteed to have a sufficient number of processors to perform this step efficiently. By then applying a probabilistic argument, one can then show that this subproblem solution often naturally divides the input into independent subproblems of expected size $O((n/r) \log n)$, where n is the total input size. This first step is the same as that used in previous randomized algorithms. Our technique differs in what follows this first step, however, in that in our algorithms we follow this first step *not* by recursing on each subproblem, but by calling a fast method on each subproblem. We also test a (theoretically superior) variation on this method, in which we take another random sample of size r' within each subproblem that contains more than n' points, where n' and r' is specified in the analysis. This effectively amounts to recursing exactly once.

We justify our approach through experimental results obtained on a Connection Machine CM-2 for segment intersection reporting, where one is given a collection of n segments in the plane and wishes a list of all intersecting pairs. This problem often arises from problems in computer graphics and solid modeling. For segment intersection reporting, the conventional practical approach involves dividing the segments into subgroups using an appropriately chosen *uniform grid* [1, 10, 20]. These algorithms have a fast expected running time *assuming* that the input comes from an appropriately-defined uniform distribution. Indeed, as we show in our analysis, these algorithms are quite sensitive to the specific definition of "uniform" that one uses. The random sampling approach needs no such assumptions. Another method that is simple to implement, and so might be used in preference to more intricate algorithms, is the "brute force" approach in which every segment is compared to every other segment. However, while the asymptotic work bound is worst-case optimal, and the constants involved in the implementation are very small, this method always performs $\Theta(n^2)$ work, even if there are no intersections, and so may perform much unnecessary work.

We have implemented our random sampling approach for the above problem on a Connection Machine CM-2 in C*, and and have run comparisons against the more traditional methods for solving these problems in parallel. Our tests are based upon selecting random inputs from distributions that would seem to significantly favor the conventional algorithms as well as from distributions that are still quite natural, but which would seem to be "bad" for the conventional approaches. As one would expect, the conventional methods run quite fast on the "kind" distributions, but are not nearly as efficient on the bad distributions. On the other hand, our methods, based on random sampling, run significantly faster on the bad distributions while still being competitive on the distributions that favor the conventional approaches.

In the sections that follow we specify the details to how we apply our random sampling approach to the our problem.

2 Segment Intersections

In the segment intersection problem, one is given a collection of n line segments in the plane and asked to report all intersecting pairs of segments. An interesting aspect of this problem, and one which makes the design of efficient algorithms a challenge, is that the size of the output can vary between 0 pairs and $\binom{n}{2}$ pairs. This problem was introduced by Bentley and Ottmann [3] who gave a simple plane sweep method running in $O((n+k)\log n)$ time. By a beautiful, but complicated, algorithm by Chazelle and Edelsbrunner [4], one can solve this problem in $O(n\log n + k)$ worst-case time, where k is the number of intersecting pairs. There is also a simpler randomized method, discovered independently by Clarkson [6] and Mulmuley [14] that runs in $O(n\log n + k)$ expected time (for any input distribution) and is based upon the paradigm of iteratively inserting the segments one-at-time in random order. All of these algorithms seem inherently sequential and do not seem to translate into efficient parallel methods. In parallel, as shown by Goodrich [12, 13], one can solve this problem deterministically in $O(\log n)$ time using either $O(n\log n + k)$ or $O(n^2/\log n)$ processors in the CREW PRAM parallel model*, while Clarkson, Cole and Tarjan gave a randomized parallel algorithm that runs in $O(\log n)$ time with an optimal expected $O(n\log n + k)$ work. These parallel methods establish parallel analogues to the efficient sequential methods, but could be criticized for their level of intricacy.

2.1 The Uniform Grid Technique

A much more practical approach, as suggested by Akman *et al* [1], and Franklin *et al* [10], is based on the well-known *uniform grid* technique (e.g., see Samet [20]).

*Recall that the CREW PRAM model is the synchronous shared-memory parallel model that allows for concurrent reads from any memory cell but requires that each write to a memory cell be exclusive.

2.1.1 The Conventional Method

This approach is as follows:

1. Determine a bounding box for all the input segments and then determine appropriate dimensions for dividing this box into a grid. A natural choice is to use the average x- (resp., y-) interval covered by the segments as the length of the horizontal (vertical) side of each grid cell. This step can be easily implemented using two parallel summation steps.
2. For each segment s, determine the grid cells that s passes through, and create pairs of the form (c, s) for each cell c that contains some part of s. On an SIMD machine this step can be implemented by a simple parallel iteration, which repeatedly determines new cells crossed by each segment (for all segments in parallel) until every segment has determined all the cells it intersects.
3. Sort the (c, s) pairs lexicographically.
4. For each grid-cell c, consider all segments that intersect c, and apply a simple "brute force" procedure to find all their pairwise intersections that fall inside c.

2.1.2 Analysis of the Uniform Grid Technique

A straightforward analysis shows that this method is obviously correct and that it should run quite efficiently on sets of short segments uniformly distributed in the bounding box. Let g be the number of grid cells, let n_c be the number of segments that fall into a cell c, and let N_s be the number of cells that a segment s intersects. Suppose, for the sake of easy comparison, that this method is implemented in the CREW PRAM model. Then the running time of this method is $O(\log n + \max_s\{N_s\})$ and the total work† involved is $O(n + (\sum_c n_c)\log(\sum_c n_c) + \sum_c n_c^2)$. If

†Recall that the *work* performed by a parallel algorithm is the total number of "real" calculations made by the virtual processors used in the algorithm, and this parameter is the factor that determines the speed-up of any implementation on a massively parallel machine with a fixed number of processors.

the segments are short, then we would expect N_s to be a constant. If the segments are uniformly distributed, then we would expect n_c to be proportional to n/g. Thus, for this special case, the running time is expected to be $O(\log n)$ with a total work bound that is expected to be $O(n \log n + n^2/g)$. (Note that to achieve $O(\log n)$ time in this case, $n + n^2/(g \log n)$ processors would be needed.)

One of the motivations for solving the segment intersection problem is that it aids in the solution of problems such as hidden surface removal, rectangle union computation, and boundary evaluation of CSG trees, which are used to represent "real-life" objects in computational solid geometry. However, there is no guarantee that the segments resulting from these problems will be the large uniform set of small edges that the uniform grid technique favors. Indeed, the uniform grid method should not perform well on long segments uniformly distributed in the bounding box. Moreover, it is easy to generate distributions where it is no better than the brute-force compare-all-pairs algorithm, which requires total work proportional to $\binom{n}{2}$. Such a bad case would arise, for example, if most of the segments are short and fall into only a few different cells, and a few of the segments are long and determine a large bounding box. Note that the segments in this bad distribution need not intersect at all!

2.2 The Brute Force Technique

2.2.1 The Naive Method

This is the most naive approach, and also the simplest to implement. The method is as follows:

For each segment s, in turn, broadcast s to all the other segments, check for intersections, and then compact to collect all the intersections found.

2.2.2 Analysis of the Brute Force Technique

A straightforward analysis shows that this method is correct. Again, for ease of comparison, assume that this is implemented in the CREW PRAM model. For n segments, the running time of this method is $O(\log n)$, with $O(n^2)$ work. (These bounds would be achieved with $n^2/\log n$ processors.) Note that these bounds hold even if there are no intersections at all.

2.3 The Random Sample Approach

Our algorithm for solving the segment intersection problem is also practical, but avoids the difficulties that would come from bad distributions.

2.3.1 The Randomized Method

Our method is as follows:

1. Take a random sample S of roughly r segments from the input, where the parameter r will be determined in the analysis. This step is implemented generating a random number x_s between 0 and n, for each segment s, and including s in the sample if $x_s \leq r$.

   Comment: Note that a sample generated in Step 1 is clearly free from duplicate entries and has an expected size of r. (See Figure 1a for an example input and Figure 1b for a sample taken from that input.)

2. Use a brute-force procedure to compute all the intersecting pairs of segments in S. Use another brute-force procedure to then determine for every segment endpoint or intersection point p the first segment (which may be a segment on the bounding box) that is hit by a vertical ray emanating upward and downward from p. This defines a *trapezoidal decomposition* T of the bounding box. This decomposition is called a "trapezoidal" decomposition because it decomposes the bounding box into cells that are essentially trapezoids. (See Figure 1c for the trapezoidal decomposition of the sample in Figure 1b, and see Preparata and Shamos [18] for more on this problem).

3. For each segment s, determine which cells in T it passes through, and for each segment s, create pairs of the form (t, s) for each cell t that contains some part of s. This step is implemented by broadcasting the trapezoids in T, one by one, so that each segment s can discover the trapezoids that it intersects.

4. Sort the (t, s) pairs lexicographically.

5. For each cell t, consider all segments that intersect t, and apply the brute force procedure, described in Section 2.2 above, to t to find all their pairwise intersections that fall inside t.

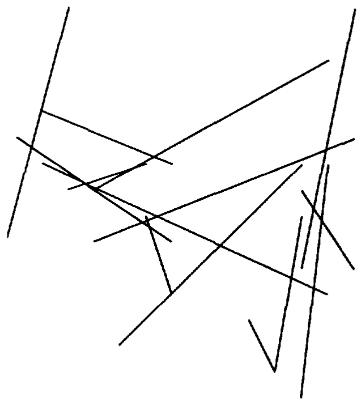

Figure 1: (1a) a set, S, of segments in the plane

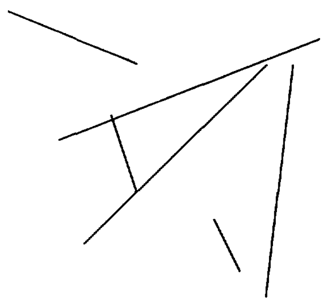

Figure 1: (1b) a random sample,R, of S

As mentioned above, the main contribution that random sampling makes is that it allows one to incorporate the structure of the input distribution into the decomposition of the bounding box into cells. That is, it is an *adaptive* technique.

2.3.2 Analysis of the Random Sample Method

Let g be the number of cells in the trapezoidal decomposition of the sample, and let n_t be the number of segments that fall into a cell t in the trapezoidal. Again, suppose for the sake of easy comparison, that this method is implemented in the CREW PRAM model. Then the running time of this method is $O(\log n + g)$ and the total work involved is $O(gn + r^2 + (\sum_t n_t)\log(\sum_t n_t) +$ (max $(n_t))^2$. Since our algorithm is randomized, we can analyze the expected work performed by our algorithm, where the expectation is taken over all possible choices our algorithm could have made. Clarkson [6] shows the following:

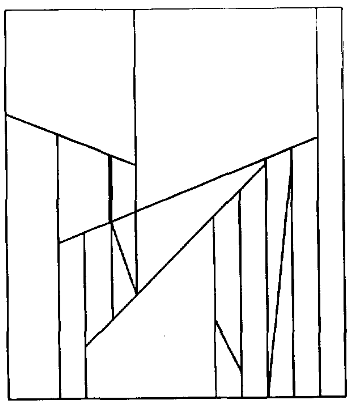

Figure 1: (1c) the grid from the trapezodial decomposition of R

Lemma 2.1: *Let A be a collection of n segments, which determine k pairs of intersections. Also, let S be a random sample of A of size r. Then the expected number of intersecting pairs of segments in S is $kr(r-1)/n(n-1)$. Moreover, if n_t denotes the number of segments intersecting a cell t in the trapezoidal map for S, then $\sum_t n_t$ is $O(n + kr/n)$ with probability at least $3/4$ and $\max_t\{n_t\}$ is $O((n/r)\log n)$ with probability at least $1 - 1/n^{10}$.*

This immediately implies that the expected value of g is $\Theta(r + kr^2/n^2)$, and the constants of proportionality are small. Moreover, it implies that the running time of our method is expected to be $O(\log n + r + kr^2/n + (n^2/r^2)\log^2 n)$.

2.3.3 The Recursive Randomized Method

This differs from the simple random sample method in that after step 3 above, all the segments in any cell t containing more than some threshold value n_2 are then subject to resampling, to take a new sample from t of size m_2.

2.4 Experimental Results

We should not be content with a nice asymptotic analysis on a theoretical model, however. In order to claim that our method yields fast practical methods we must experimentally compare it to the more conventional uniform grid approach.

2.4.1 Types of Distributions

We used the following input distributions for the segment intersection problem:

- **short uniform segments distributed at random in the interval $(0,n)(0,n)$.** The segments are of constant length. This distribution is tailor-made for the uniform grid technique, and one would expect that it would do better on this "best case" distribution than the randomized method.
- **axis-parallel rectangles of random size and location.** Several of the applications of segment intersection involve axis-parallel rectangles, so rectangles might give some insight into how the techniques would work in practice.
- **parallel diagonal line segments** The segments are of length n, and the ith segment is between points $(i,0)$ and $(2i,i)$, so all the segments will be included in at least one square of the grid method. Note that this could be avoided by using a different square size, but each fixed square size will have a similar worst case.

2.4.2 Test Results

All the segment intersection intersection algorithms were written in C*, and run on an 16384 processor CM2, at the University of Maryland Institute for Advanced Computer Science. The data points were generated on the CM2, using the internal pseudo-random number generator to conform to the distribution being tested. In accordance with the recommendations of the C* documentation, each run was repeated several times, to ensure that the time achieved is repeatable. For comparisons between methods, the random sample method used a sample size of $\sqrt{n}$. The timing was performed by the internal CM2 clock.

- **short uniform segments** The results confirmed that this is ideal for the uniform grid technique: the uniform grid program ran two orders of magnitude faster than for any other distribution. It showed no noticeable slowdown with increasing input size: all tests, with input sizes varying from 1000 to 10000, ran in under a second. The randomized technique showed a more noticable dependency on input-size. The randomized technique was somewhat faster than brute force, the difference becoming more noticeable at larger input sizes. (See Figure 2.)

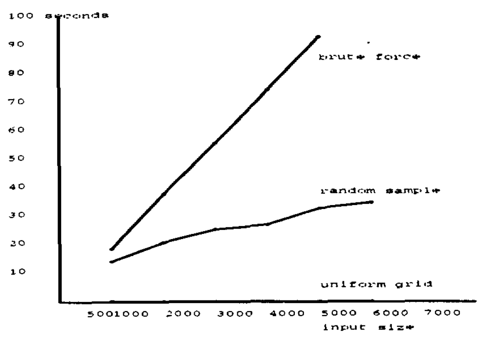

Figure 2: short uniform segments

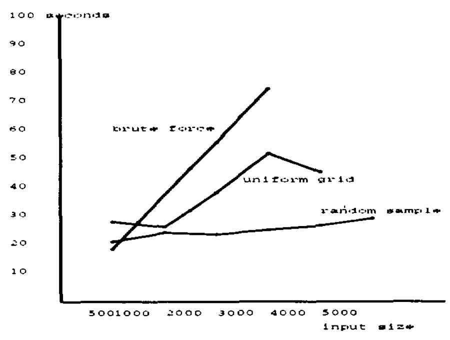

Figure 3: axis-parallel rectangles

- **axis-parallel rectangles** With this distribution, the uniform grid method took longer than the randomized technique, though not the brute force technique. Again, with the same comments as above, the brute force technique was slower than the randomized method. (See Figure 3)
- **parallel diagonal segments** With this, which was a worst case distribution for the grid method, the grid method was worse than both the randomized technique and brute force. (See Figure 4.) Note that this is what should be expected, as in such a "worst case," the grid computation amounts to that of the brute force technique, with some overhead.

2.4.3 The Recursive Method in Practice

As can be seen from Figure 5, the overhead associated with the recursion appears to be greater than any benefit gained. As one might expect, the performance of the recursive method is essentially the

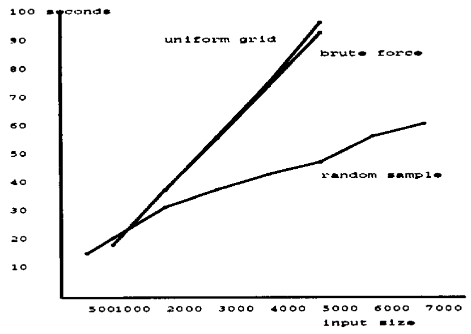

Figure 4: parallel diagonal segments

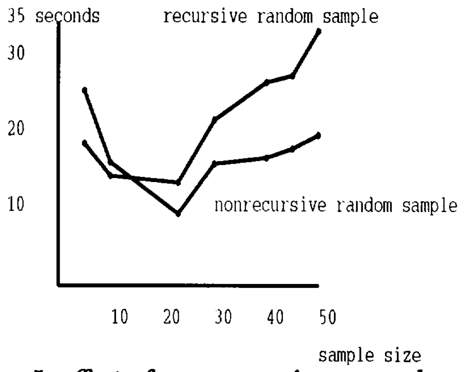

Figure 5: effect of more recursion on random sample method, for diagonal segments, n=500

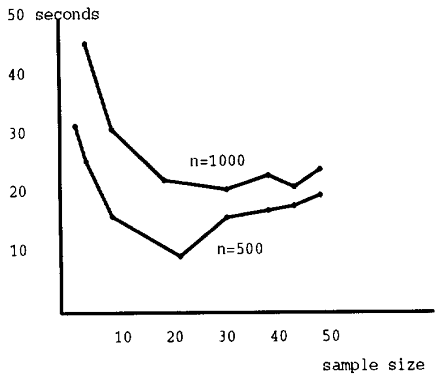

Figure 6: effect of sample size on run time for parallel diagonal segments

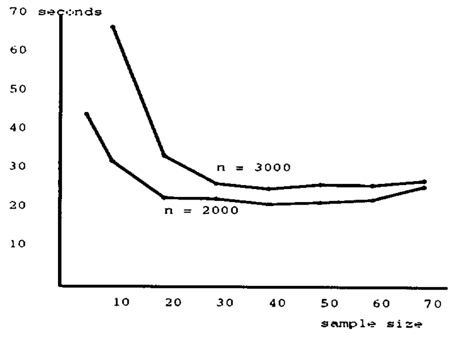

Figure 7: effect of sample size on run time for short uniform segments

same as that of the non-recursive random sample method given a larger sample, plus some overhead.

2.4.4 The Effect of Sample Size

Without a method of determining a "good" sample size in advance, the random sample method is not really useful, as it might otherwise have to be run several times in order to find a sample size with which the program runs efficiently. As can be seen from Figure 6, for this distribution there is a great range of sample sizes that have relatively little effect on the time taken, and in particular a size of exactly $\sqrt{n}$ is an adequate choice. This can also be seen for the short randomly distributed segments used for Figure 7. Note that the overall shape of the graphs is as one would expect from the discussion of lemma 2.1: a sample size that is very small results in unnecessarily large subproblems, and very long run times, while a large sample size also increases the overhead associated with solving the sample, and slightly increases the time taken overall.

3 Conclusion

We have shown how random sampling can be used to design efficient practical parallel algorithms that "randomize away" bad input distributions and run efficiently for any input distribution. Our methods have small constants of proportionality in the running times and are either competitive or significantly better than existing methods based on conventional techniques for designing practical parallel methods, which are quite sensitive to input distributions, as well as the naive "brute force" approach.

We have also shown that a recursive random sampling approach may not be worthwhile, due to the extra overhead incurred, and that the variation of time taken with sample size is as expected.

Acknowledgements

We would like to thank Lewis Stiller for several helpful discussions, and we would like to thank Dr. Larry Davis and the Maryland High Performance Computing Consortium for generously giving us access to a Connection Machine CM-2 supercomputer.

References

[1] V. Akman, W.R. Franklin, M. Kankanhalli, C. Narayanaswami, "Geometric computing and uniform grid technique," *Computer-Aided Design*, 1989, 410–420.

[2] N. Alon and N. Megiddo, "Parallel Linear Programming in Fixed Dimension Almost Surely in Constant Time", *Proc. 31st Annual IEEE Symp. on Foundations of Computer Science*, 1990, 574–582.

[3] J. L. Bentley and T.A. Ottmann, "Algorithms for reporting and counting geometric intersections," *IEE Trans. Comput.* **C-28** (1979), pp. 643-647.

[4] B. Chazelle and H. Edelsbrunner, "An optimal algorithm for intersecting line segments in the plane," *Proc. 29th Annual IEEE Symp. on Foundations of Computer Science*, 1988, 590–600.

[5] K.L. Clarkson, "Further Applications of Random Sampling to Computational Geometry," *Proc. 18th Annual ACM Symposium on Theory of Computing,* (1986) 414–423.

[6] K.L. Clarkson, "Applications of Random Sampling in Computational Geometry II," *Proc. 4th Annual ACM Symposium on Computational Geometry,* (1988) pp. 1-11.

[7] K.L. Clarkson, R. Cole, and R.E. Tarjan, "Randomized Parallel Algorithms for Trapezoidal Diagrams," *Proc. 7th ACM Symp. on Computational Geometry*, 1991, 152–161.

[8] K.L. Clarkson and P. Shor, "New applications of random sampling in computational geometry II," *Disc. & Comput. Geo.*, Vol. 4, 1989, 387–421.

[9] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, "Introduction to Algorithms," MIT Press, 1989.

[10]
W.R. Franklin, N. Chandrasekhar, M. Kankanhalli, M. Seshan, and V. Akman, "Efficiency of uniform grids for intersection detection on serial and parallel machines," *Proc. Computer Graphics International '88*, Springer-Verlag, 1988.

[11] M. Ghouse and M.T. Goodrich, "In-Place Techniques for Parallel Convex Hull Algorithms," *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, 1991, 192–203.

[12] M.T. Goodrich, "Intersecting Line Segments in Parallel with an Output-Sensitive Number of Processors," *SIAM Journal on Computing*, Vol. 20, No. 4, 1991, 737–755.

[13] M.T. Goodrich, "Constructing Arrangements Optimally in Parallel," *Discrete and Computational Geometry*, to appear. (a preliminary version appeared in *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, 1991, 169–179.)

[14] K. Mulmuley, "A fast planar partition algorithm, (I)," *Proc. 29th Ann. IEEE Symp. on Found. Comp. Sci.* (1988) pp. 580-589.

[15] K. Mulmuley, "A fast planar partition algorithm, (II)," *J. ACM* **38** (1991) pp. 74-103.

[16] K. Mulmuley, "Randomized multidimensional search trees: dynamic sampling," *Proc. 7th Annu. Symp. Comput. Geom.* (1991) pp. 121-131.

[17] K. Mulmuley, "Randomized multidimensional search trees: further results in dynamic sampling," *Proc. 32nd Annu. IEEE Symp. Found. Comput. Sci.* (1991) pp. 216-227.

[18] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, NY, 1985.

[19] J. H. Reif and S. Sen, "Polling: A New Randomized Sampling Technique for Computational Geometry," *Proc. 21st ACM Symposium on the Theory of Computing*, (1989) pp. 394-404.

[20] H. Samet, *Applications of Spatial Data Structures*, Addison-Wesley, 1990.

[21] R. Seidel, "Linear Programming and Convex Hulls Made Easy," *Proc. 6th ACM Symp. on Computational Geometry*, 1990, 211–215.