RANDOMIZATION IN PARALLEL ALGORITHMS AND ITS IMPACT ON COMPUTATIONAL GEOMETRY†

John H. Reif and Sandeep Sen
Computer Science Department
Duke University,
Durham, N.C. 27706,
U.S.A.

Abstract

Randomization offers elegant solutions to some problems in parallel computing. In addition to improved efficiency it often leads to simpler and practical algorithms. In this paper we discuss some of the characteristics of randomized algorithms and also give applications in computational geometry where use of randomization gives us significant advantage over the best known deterministic parallel algorithms.

Motivation

Designing parallel algorithms for various fundamental problems in computational geometry has received much attention in the last few years. After some early work by Anita Chow in her thesis, Aggarwal et al.[13] developed some general techniques for designing efficient parallel algorithms for a number of fundamental problems. These included convex hulls in two and three dimensions, voronoi diagram for planar point sites, triangulation and planar point-location among others. Although most of these problems have a sequential time-complexity of $\Theta(n \log n)$, the authors presented parallel algorithms which uses a linear number of processors and runs in $O(\log^k n)$ time (k being typically 2,3 or 4) in a PRAM model. Consequently, the problem of designing optimal (in the processor-time product sense) algorithms were left open. Since then a number of the open problems in the original list have been settled due to the work by Atallah, Cole and Goodrich[14] who were able to apply Cole's elegant techniques for parallel mergesort to a number of these problems.

We present techniques for obtaining optimal parallel algorithms for problems in computational geometry using randomization. As applications of our methods, we derive efficient parallel algorithms for planar-point location, convex-hull and trapezoidal decomposition. These algorithms run in time $T = O(\log n)$ using $O(n)$ processors for problem size n and terminate in the claimed time bound with probability $1 - n^{-c}$ for any integer c. These bounds are worst-case and do not depend on any input

† Supported in part by Air Force Contract AFSOR-87-0386, ONR contract N00014-87-K-0310, NSF grant CCR-8696134, DARPA/ARO contract DAAL03-88-K-0185, DARPA/ISTO contract N00014-88-K-0458.

Preprint of paper appearing in Optimal Algorithms;Lecture Notes in Computer Science, Vol. 401, 1989, pp. 1-8.

distribution. The main contribution of our work is a new random sampling technique called **Polling** which can be used for doing divide-and-conquer efficiently on various problems in computational geometry. Our techniques lead to algorithms that are considerably simpler than their methods and appear to have wider applications. For example we have derived an optimal O(logn) time n processors algorithm for constructing the convex hull of points in three dimensions (Reif and Sen[18]). Presently the best known deterministic algorithm for this problem takes $O(\log^2 n \log^* n)$ time using n processors (Dadoun and Kirkpatrick[17]).

Basics

Randomization was formally introduced by Rabin[6] and independently by Solovay & Strassen[8] as a tool for improving the efficiency of certain algorithms. In a nutshell, a randomized algorithm uses coin-flips to make decisions at different steps of the algorithm. Therefore a randomized algorithm is actually a family of algorithms where each member of this family corresponds to a fixed sequence of outcomes of the coin-flip. Two of the most commonly used forms of randomization in literature are the *Las Vegas* algorithms and *Monte Carlo* algorithms. The former kind ensures that the output of the algorithm is always correct - however only a fraction (usually greater than 1/2) of the family of algorithms halt within a certain time bound (as well as with respect to some other resources like space). In contrast, the Monte Carlo procedures always halt in a pre-determined time period; however the final output is correct with a certain probability (typically > 1/2). This lends itself very naturally to decision algorithms (Rabin's primality testing being a good example). For the purpose of this discussion we shall limit ourselves to the Las Vegas algorithms which have been more popular with the algorithm designers. For a general algorithm which produces more than just 'yes-no' output, the precise meaning of an incorrect output becomes subjective; for example we may need to know how close are we to the correct output in order to decide if the output is acceptable. Although, this is one of the reasons for bias towards Las Vegas algorithms, the use of either kind of algorithms depends on the particular application.

Complexity measures of randomized algorithms

Before we discuss the applications of these algorithms in parallel computing, it is important to review some of the performance measures used by these algorithms. This will enable us to compare the relative merits of different randomized algorithms. To begin, we must emphasize the distinctions between a randomized algorithm and probabilistic algorithm. By probabilistic algorithms, we imply those algorithms whose performance depend on the input distribution. For such algorithms, we are often interested in the average resources used over all inputs (assuming a fixed probability distribution of the input). A randomized algorithm does not necessarily depend on the input distribution. A randomized algorithm uses a certain amount of resources for the worst-case input with probability $1-\varepsilon$ ($0 < \varepsilon < 1$), i.e. the bound holds for any input (which is a stronger bound than the average bounds). This can be very well illustrated with the example of Hoare's *Quicksort* algorithm. In its original form, it is a probabilistic algorithm which performs very well on certain inputs and deteriorates sharply on some

other inputs. By assuming that all inputs are equally likely (known as random-input assumption), the algorithm performs very well on the average. By introducing randomization in the algorithm itself, it has been shown to perform very well on **all** inputs with high probability. This is certainly a more desirable property since a *malicious oracle* who could control the performance of the original algorithm by giving it worst case inputs, can no longer affect it. Of course, the onus of a successful run of the algorithm is now shifted to the outcome of the coin-flips. This depends on certain randomness properties of the random-number generator, which is a topic in itself. Also note that this discussion does not preclude designing randomized algorithms which are dependent on the input distribution but these algorithms are no different from their deterministic counterparts.

Until now we have characterized the randomized algorithms with a success probability of 1 - ε without specifying the possible forms of ε. It can be a fixed constant or a function (which takes values between (0,1)). It must be clear that ε should be minimized (compare this with deterministic algorithms where ε is 0). Intuitively we can expect a trade-off between ε and the amount of resource used. In other words, the failure probability ε must decrease with increasing amount of resources. Let us consider a concrete example. Suppose $T_A(n)$ is the *expected* running time of the randomized algorithm A for input size n. What can we say about ε? If we don't have any bounds other than the expectation we can only use Markov's inequality. From Markov's inequality, the probability that running time exceeds $kT_A(n)$ is less than 1/k. For example, if k=2, ε = 1/2. Compare this with an algorithm B for the same problem whose running time exceeds $k\alpha T_B(n)$ with probability less than $1/n^{\alpha}$ and suppose that for any given α, k is a constant independent of n. This implies that the probability of failure diminishes rapidly as n increases and vanishes asymptotically. We have characterized the failure probability ε as a decreasing function of the problem size, n and resources used by the algorithm. The reader will recognize that the faster ε decreases with these parameters the better is the algorithm. This makes algorithm B superior to algorithm A if $T_A(n)$ and $T_B(n)$ represent the same function. The basic idea is that depending on the application, the user chooses a certain value of ε and accordingly chooses k (given the value of n) with the objective of minimizing k. There is no reason to be pedantic about the kind of function ε should be except that a failure probability of the second form (that of algorithm B) has been very widely used in literature and such algorithms have been termed as having *high probability* of success. This kind of failure probability function is quite robust with respect to a polynomial number of procedures i.e. the union of a polynomial number of events, each with high probability of success, succeeds with high probability. It may be a non-trivial task to transform an algorithm like A to an algorithm like B (which succeeds with high probability). The reader must also appreciate that randomized algorithms like B which have such high probability of success should be competitive with deterministic algorithms for the same problem. According to Adleman & Manders[1], a randomized algorithm with success probability more than 1 - 2^{-k} (for some large fixed k) has a lower probability of failure than the hardware itself.

Parallel computation and randomization

Randomization has proven to be an extremely effective in parallel algorithm design. One of the earliest treatment of this topic can be found in Reif[12]. For a more recent and extensive survey the reader is encouraged to read the first two chapters of Rajasekaran[7]. A commonly accepted measure of efficiency of parallel algorithms is the *processor-time* product (in short $P \cdot T$). Although the primary objective of parallel algorithms is to minimize time complexity (the number of parallel time steps), in practice one also has to be careful about the number of processors needed to achieve this speed-up. The *efficiency* of a parallel algorithm is a measure of how expensive is the speed-up compared to the sequential algorithm. Clearly $P \cdot T$ product cannot be better than the sequential time complexity of the algorithm. Ideally one would like the speed-up to be linear with the number of processors used; however this is far from true in most cases. This also gives an abstract measure of how 'hard' it is to parallelize a particular problem. We say that a parallel algorithm is *efficient* if $P \cdot T \leq O(Seq(n) \cdot \log^k n)$ for some constant k where n is the input size. The class NC is defined to be the class of problems which admit poly-logarithmic time parallel algorithm using a polynomial number of processors. Note that while these algorithms admit fast parallel algorithms they may not be necessarily *efficient.*

Use of Random Sampling

Randomized sampling techniques have been used extensively in cases of divide-and-conquer algorithms (most parallel algorithms would fall under this category). The idea is to divide up the problem 'almost evenly' into smaller sub-problems using a randomly chosen subset of the input. This random subset is called splitters and because of the process of random selection, various probabilistic arguments can be used to bound the size of the sub-problems. For example, in parallel sorting, we can choose $\sqrt{n}$ keys randomly and partition the input into $\sqrt{n}$ subsets using the partitions induced by the random keys. Using simple probabilistic arguments, it is not difficult to bound the size of the subproblems to approximately $O(\sqrt{n \log n})$ with high probability. The main algorithm is then used recursively on each of the partitions.

Random sampling in computational geometry was first introduced by Clarkson and since then he has published a series of results leading to improvements and simplification of a number of sequential algorithms in computational geometry. However his time bounds are *expected* in contrast to our *high-likelihood* bounds (these have success probabilities $1 - n^{-c}$ for any integer c) which aside from being weaker are of little use for obtaining parallel alorithms. The reason being that for sequential algorithms, he was able to use the linearity property of expectation (i.e. the expectation of the sum is the sum of expectations) and so it was enough to bound the expected running time of each individual step. For a parallel algorithm, one is looking for the maximum of the expectation from the expectation of a family of random variables and there exists no known method of obtaining this value. Among other techniques, we introduce a new random-sampling technique called 'Polling' which enables us to obtain *high probability* bounds for our algorithms and complements Clarkson's work to a large extent.

More specifically, it also allows us to use random sampling recursively without blowing up the problem size.

Resampling and Polling

The informal idea behind 'Polling' is the following: For a given problem of size n, we choose randomly a small (typically n^{ε} for $\varepsilon < 1/2$) subset of the given input and use it to divide the original problem. It is not difficult to show that the size of each sub-problem is no larger than $n^{1-\varepsilon}logn$ but the total size of the subproblems may be considerably larger than n. For example, if the input is line segments on the plane, a large number of the line segments may be broken up into smaller pieces during the divide step. This phenomenon is not witnessed in a problem like sorting where the total size of the subproblems is always exactly equal to the input size. Increase in the problem size at every recursive call would result in grossly inefficient algorithms. Clarkson[15] had shown that the total sum of the subproblems has *expected* value O(n). This implies that with probability atmost 1/2, this value would exceed $k_{total}n$ for some constant k_{total}. Consequently, if we choose independently O(logn) random subsets, at least one of them will be 'good' (i.e. sum of subproblems does not exceed k_{total}n) with high probability. If we resample O(logn) times, we may end up doing non-optimal number of operations. Instead we test the 'goodness' of a sample on only $\frac{n}{\log^r n}$ of the input. It can be proved that this gives estimates within a constant factor with very high probability (Reif and Sen[18]). Thus we are able to choose a random sample such that the size of the subproblems does not exceed more than a constant times the input size. We call this technique 'Polling' (as if we are polling a fraction of the input to test the 'goodness' of a sample).

Notice that even with Polling the sum of sub-problems may grow by a constant factor which could lead to a polylogarithmic factor increase over O(loglogn) levels of recursion. With some additional filtering methods (which can be efficiently applied since the size is still O(n)), we are able to bound the problem size by $k_{max}n$ at any level where k_{max} is a constant. For a parallel algorithm, we are able to get a recursion which is roughly of the form
$T(n) = T(n^{1-\varepsilon}) + O(logn)$. This has a solution T(n) = O(logn) and the processor bound is simultaneously O(n).

The overall algorithm can be summarized as following:

(i) Choose independently O(logn) random subsets each of size $O(n^{\varepsilon})$.

(ii) Use 'Polling' to identify a 'good' sample.

(iii) Partition the problem by the random sample.

(iv) Use 'Filtering' to control the sum of the sub-problems. (The implementation of this step is problem dependent. For example it is different in the case of

trapezoidal decomposition and convex hulls).

(v) If the largest sub-problem size is larger than a certain size apply algorithm recursively.

Randomization as a resource

Some recent efforts directed towards 'derandomization' of randomized algorithms have received attention. While such research has a lot of theoretical ramifications (in understanding relationship between the classes NC and Random NC), these methods, almost without exception lead to loss in efficiency of the algorithms. This may not be be very fruitful from a practical viewpoint. There are formal methods for proving lower bounds for randomized algorithms which gives a strong basis for 'optimality' in randomized algorithms. Moreover, it has been observed quite often that for a given problem, the lower-bounds for deterministic algorithms turn out to be identical to the lower bounds for randomized algorithms (within constant multiplicative factor).

Perhaps a more fruitful area of investigation could be directed towards reduction of the number of random bits used in an algorithm (without affecting the asymptotic bounds) since perfect 'randomness' has been recognized as an expensive resource. This has been demonstrated by some recent work due to Raghavan & Karloff[10] and the authors feel that further research in this direction could be very rewarding from a theoretical perspective as well as from a practical viewpoint. For all our algorithms we are able to bound the number of purely random bits to $O(\log^2 n)$.

Geometry on Interconnection networks

Note that the underlying model for all these algorithms is the parallel analogue of the sequential RAM (Random Access Memory) model, PRAM. This model has become the standard model (within minor variations) for theoretical work on design and analysis of parallel algorithms. The state of art of parallel geometric algorithms for feasible models such as butterfly or hypercubes is lagging far behind the PRAM models; the only known optimal O(logn) time n processor algorithm exists for 2-D convex hulls.

One of our primary motivation for research in randomized parallel algorithms is the absence of optimal deterministic sorting algorithm on the commonly used interconnection networks. The only known optimal sorting network is the *AKS* network which sorts in O(logn) depth but has horrendous constants. Moreover, the algorithm is not suited for the widely used architectures like butterfly or hypercube, which are more suitable for general-purpose computations. In contrast, Flashsort[11], an O(logn) depth randomized sorting algorithm has much lower constants and runs on the standard architectures. The other optimal parallel mergesort algorithm of Cole, which has low constants and is very elegant is of little use in practice since it is tailor-made for PRAM model. This model is very popular among algorithm designers from a theoretical perspective because of its simplicity. However, for such algorithms to be of any use, there must be a general mechanism for mapping algorithms from PRAM on to

interconnection networks. These involve routing on interconnection networks and the best known routing algorithms (O(logn) time) involve use of randomization in some form. Moreover, such emulation usually lead to loss of efficiency by O(logn) multiplicative factor. Consequently, Cole's algorithm deteriorates to an $O(\log^2 n)$ algorithm on the interconnection networks. Unlike the cascading divide-and-conquer paradigm used in [14], we have reasons to believe that some of our methods can be extended to the fixed connection networks without loss of efficiency.

Conclusion

In summary, we note that randomized algorithms offer a very pragmatic alternative (to deterministic algorithms) in the area of parallel algorithms. Apart from being simpler than their deterministic counter-parts they usually have smaller constants. In addition, they provide a bridge between the algorithm-designer who designs algorithms for abstract models like PRAM and its realistic implementation on feasible architectures.

Bibliography

[1] L. Adleman and K. Manders, 'Reducibility, Randomness and Untractability,' Proc. 9th ACM STOC, 1977, pp. 151-163.

[2] A. Aggarwal and R. Anderson,'A Random NC Algorithm for Depth First Search,' Proc of the 19th ACM STOC, 1987, pp. 325-334.

[3] Aggarwal et al., 'Parallel Computational Geometry,' Proc. of the 26th Annual Symp on F.O.C.S., 1985, pp. 468-477. Also appears in ALGORITHMICA, Vol. 3, No. 3, 1988, pp. 293-327.

[4] Atallah, Cole and Goodrich, 'Cascading Divide-and-conquer: A technique for designing parallel algorithms, Proc. of the 28th Annual Symp. on F.O.C.S., 1987, pp. 151-160.

[5] Clarkson, 'Applications of random sampling in Computational Geometry II,' Proc. of the 4th Annual Symp. on Computational Geometry, June 1988, pp. 1-11.

[6] N. Dadoun and D. Kirkpatrick, 'Parallel Processing for efficient subdivision search,' Proc. of the 3rd Annual Symp. on Computational Geometry, pp. 205-214, 1987.

[7] H. Gazit, 'An optimal randomized parallel algorithm for finding connected components in a graph,' Proc of the IEEE FOCS, 1986, pp. 492-501.

[8[C.A.R. Hoare, 'Quicksort,' Computer Journal, 5(1), 1962, pp.10-15.

[9] H. Karloff and P. Raghavan, 'Randomized algorithms and Pseudorandom number generation,' Proc. of the 20th Annual STOC, 1988.

[10] G. Miller and J.H. Reif,'Parallel Tree contraction and its applications,' Proc of the IEEE FOCS, 1985, pp. 478-489.

[11] M.O. Rabin, 'Probabilistic Algorithms,' in: J.F. Traub, ed., Algorithms and Complexity, Academic Press, 1976, pp. 21-36.

[12] S. Rajasekaran, 'Randomized Parallel Computation,' Ph.D. Thesis, Aiken Computing Lab, Harvard University, 1988.

[13] J.H. Reif, 'On synchronous parallel computations with independent probabilistic choice,' SIAM J. Comput., Vol. 13, No. 1, Feb 1984, pp. 46-56.

[14] Reif and Sen, 'Optimal randomized parallel algorithms for computational geometry,' Proc. of the 16th Intl. Conf. on Parallel Processing, Aug 1987. Revised version available as Tech Rept CS-88-01, Computer Science Dept, Duke University.

[15] J. Reif and S. Sen, 'Polling: A new random sampling technique for Computational Geometry,' Proc. of the 21st STOC, 1989.

[16] J.H. Reif and L. Valiant, 'A Logarithmic Time Sort for Linear Size networks,' J. of ACM, Vol. 34, No.1, Jan '87, pp. 60-76.

[17] R. Solovay and V. Strassen,'A fast Monte-Carlo test for primality,' SIAM Journal of Computing, 1977, pp. 84-85.

[18] L.G. Valiant, 'A scheme for fast parallel communication,' SIAM Journal of Computing, vol. 11, no. 2, 1982, pp. 350-361.