Statistical Science
1993, Vol. 8, No. 1, 65–69

Randomization in Parallel Algorithms

Vijaya Ramachandran

Abstract. A randomized algorithm is one that uses random numbers or bits during the runtime of the algorithm. Such algorithms, when properly designed, can ensure a correct solution on every input with high probability. For many problems, randomized algorithms have been designed that are simpler or more efficient than the best deterministic algorithms known for the problems. In this article, we define a natural randomized parallel complexity class, RNC, and give a survey of randomized algorithms for problems in this class.

Key words and phrases: Analysis of algorithms and problem complexity, combinatorial probability, nonnumeric algorithms, parallel and distributed algorithms, parallel computation.

1. INTRODUCTION

In recent years, there has been an explosive growth in the availability and diversity of parallel processors for computation. For the purpose of parallel algorithm design, it is convenient to work with an abstract, simplified machine model, known as the *Parallel Random Access Machine* (*PRAM*). The PRAM incorporates the basic elements of a parallel machine and has the property that an algorithm designed for it will perform without significant degradation on most commonly available parallel machines, including shared-memory multiprocessors and fixed interconnection networks. For a survey of PRAM algorithms, both deterministic and randomized, see Karp and Ramachandran (1990).

The PRAM consists of a collection of independent sequential machines, each of which is called a processor, that communicate with one another through a global memory. This is a synchronous model, and a step of a PRAM consists of a read cycle in which each processor can read a global memory cell, a computing cycle in which each processor can perform a unit-time sequential computation and a write cycle in which each processor can write into a global memory cell. There are many variants of this model, differing in whether a read conflict or a write conflict is allowed and, in the latter case, differing by the method used to resolve a write conflict. Because efficient simulations of the various models are known, we shall not elaborate on these variants.

Parallel algorithms have been designed on the PRAM for a large number of important problems. This has been a rich and exciting area of research in recent years, and many new techniques and paradigms have been developed. However, in spite of impressive gains, some problems have proved to be resistant to attempts to design highly parallel algorithms for their solution. For some other problems, parallel algorithms with good performance have come at the expense of extremely complex and intricate algorithms. In this context, several researchers have turned to randomization in an attempt to obtain better algorithms for these problems.

A randomized parallel algorithm is an algorithm in which each processor has access to a random number generator. The goal is to use this capacity to generate random numbers to come up with an algorithm that solves a problem quickly and with high probability on every input. The requirement that the algorithm achieve its performance guarantee on every input is much stronger than that of demanding good average-case performance and is a highly desirable property in an algorithm that uses randomization. Correspondingly, such algorithms are more difficult to design. Fortunately, there have been a number of successes in the design of such algorithms for parallel machines.

2. QUALITY MEASURES FOR RANDOMIZED PARALLEL ALGORITHMS

For the purpose of this exposition, it is convenient to consider a problem as a binary relation $s(x, y)$, where x is the input to the problem and y is a possible output for x. The value of $s(x, y)$ will be true if and only if y is a solution on input x. An algorithm for the problem should, on input x, either output some y such that $s(x, y)$ is true or output the fact that no such y exists. A Monte Carlo (or one-sided error) randomized algorithm is one that returns a y such that $s(x, y)$ is true

Vijaya Ramachandran is Associate Professor, Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712.

with probability at least 1/2 if such a y exists; if no such y exists, the algorithm always reports failure. Other types of randomized algorithms can be defined, but Monte Carlo algorithms are the ones used most commonly.

Given a Monte Carlo algorithm for a problem, we can improve our confidence in the result supplied by the algorithm from 1/2 to $(1 - 1/2^k)$ by performing k independent runs of the algorithm. Hence, given any $\varepsilon > 0$, we can obtain a confidence level greater than $1 - \varepsilon$ by performing $\lceil \log(1/\varepsilon) \rceil$ independent runs of the algorithm.

The notion of a Monte Carlo algorithm applies to algorithms in any model—sequential, parallel or distributed. In the following sections, we will address the issue of obtaining Monte Carlo algorithms that execute quickly on a PRAM.

3. THE RANDOMIZED PARALLEL COMPLEXITY CLASS RNC

In the design and analysis of highly parallel algorithms for various problems, it has been observed that some problems have simple highly parallel algorithms using a relatively small number of processors, whereas others have resisted all attempts so far to obtain any algorithm with a significant amount of parallelism. In this context, researchers have identified a natural parallel complexity class NC. The class NC consists of those problems that have PRAM algorithms that run in polylog time, that is, time polynomial in the logarithm of the input size, with a polynomial number of processors. This class is robust in the sense that the collection of problems it contains does not vary with the machine model used, whether it is a shared-memory machine or a low-diameter interconnection network. It is also the smallest nontrivial class that is robust. Although several important problems have been shown to be in NC, many others have not. The latter problems are of two types (for our purposes): one type consists of those problems that are either provably not in NC by virtue of the fact that they are provably not in P (polynomial time) or are highly unlikely to be in NC by virtue of the fact that they are complete problems for a larger class (such as P); for these problems, we currently know of no technique, randomized or otherwise, to come up with algorithms that run in polylog time with a polynomial number of processors. The other type of problems are those for which no one has come up with either an algorithm to place the problem in NC or a completeness result to make it highly unlikely to be placed in NC. This latter type includes several important problems such as finding a maximum matching or a depth-first search tree (an important type of spanning tree) in a graph. For these problems, randomization has proved to be a valuable tool in coming up with fast parallel algorithms. For more on parallel complexity classes, see Cook (1981, 1985) and Karp and Ramachandran (1990).

The class RNC is the class of problems that can be solved by a Monte Carlo randomized algorithm that runs in polylog time with a polynomial number of processors. Thus, RNC is the randomized counterpart of NC.

4. SOME IMPORTANT PROBLEMS IN RNC

4.1 Testing If a Multivariate Polynomial Is Not Identically Zero

Let $Q(x_0, \ldots, x_{n-1})$ be a multivariate polynomial over a field. Consider the problem of determining if this polynomial is *not* identically zero. It is not known if this problem can be solved deterministically in polynomial time sequentially. However, if we allow randomization, we can come up with a simple Monte Carlo algorithm for the problem, as shown below.

The randomized algorithm is based on the following lemma, which is fairly straightforward to prove by induction on the number of variables n (note that the base case $n = 1$ follows from the fact that any single variable polynomial of degree n has at most n zeros).

LEMMA 1. *Let $Q(x_0, \ldots, x_{n-1})$ be a polynomial of degree d with coefficients over a field F. For any set $I \subseteq F$, the number of zeros of Q in $I^n \leq |I|^{n-1} \cdot d$.*

The above lemma gives us the following Monte Carlo algorithm to determine if Q is not identically zero (Schwartz, 1980):

1. Choose any set $J \subseteq F$ containing $2d$ elements.
2. Pick a random element $e = (e_0, \ldots, e_{n-1}) \in J^n$, and evaluate $Q(e)$.
3. If $Q(e) \neq 0$, then report $Q \not\equiv 0$, else report failure.

The probability of success in the case in which Q is not identically zero can be enhanced to $1 - 1/2^k$ by repeating steps 2 and 3 k times. This algorithm is easily implemented on a PRAM with n processors by having the ith processor generate e_i. Then, provided $Q(e)$ can be determined quickly given e, we have a fast algorithm to test if $Q(e) \not\equiv 0$. In particular, if evaluating $Q(e)$ is in NC, then determining if $Q(x_0, \ldots, x_{n-1}) \not\equiv 0$ is in RNC.

4.2 Finding a Maximum Matching in a Graph

A matching in an undirected graph is a subset of edges, no two of which share a vertex. A perfect matching is a matching that contains all vertices of the graph. An RNC algorithm for the problem of determining if a graph has a perfect matching can be obtained using a theorem of Tutte (1947) that shows that the determinant of a certain matrix of multivariate polynominals becomes identically zero if and only if the graph does not have a perfect matching. This matrix is easily

constructed from a description of the input graph. Because computing the determinant of an integer matrix is in NC, the problem of determining the existence of a perfect matching is in RNC by virtue of the above result on determining if a multivariate polynomial is not identically zero. This result can be extended to place the problem of determining the cardinality of a matching of maximum size and that of finding a matching with this cardinality in RNC (Karp, Upfal and Wigderson, 1986; Mulmuley, Vazirani and Vazirani, 1987).

The problem of finding a maximum matching in a graph is an important one and one that has received extensive attention in the context of sequential algorithm design. Although the problem is not known to be in NC, the above result allows us to find a maximum matching quickly and with high confidence. RNC algorithms for several other problems have been obtained in recent years; for example, Aggarwal and Anderson (1987), Aggarwal, Anderson and Kao (1989), Babai (1986), Gibbons et al. (1988), Karloff (1986) and Ramachandran (1988).

5. RANDOMIZATION LEADS TO SIMPLE PARALLEL ALGORITHMS

In practice, the number of processors available for solving a problem is typically much smaller than the problem size. Although massive parallelism may become a reality in the future, it is still of importance to address the issues that arise when the amount of parallelism available is small in comparison to the size of the input. In such a case the efficiency of a parallel algorithm becomes important. This refers to the speedup provided by the parallel algorithm using a fixed number of processors over the best currently known sequential algorithm. A parallel algorithm is considered efficient if the product of its running time and the number of processors it uses is within a polylog factor of the running time of the current best sequential algorithm; it is considered optimal if this product is within a constant factor of the running time of the best sequential algorithm. A parallel algorithm that is efficient (or optimal) when run using a certain number of processors will remain an efficient (or optimal) parallel algorithm when implemented on a smaller number of processors while running proportionately slower. Hence it is of interest to construct efficient parallel algorithms that run very fast, regardless of the number of processors available.

There are a few problems, such as computing a depth-first or breadth-first search tree in a dense graph or performing Gaussian elimination, for which parallel algorithms with very efficient speedups are known, even though NC algorithms are either not known or highly inefficient. However, most problems for which efficient parallel algorithms are known are in NC. Thus, placing a problem in NC is generally a first step to obtaining an efficient parallel algorithm for it.

Randomization has proved useful in the design of efficient and optimal parallel algorithms. It has also been useful in developing algorithms that are simpler than the best deterministic parallel algorithm known for the problem. Randomization has been applied to achieve these goals for a wide range of problems in graph theory (e.g., Alon, Babai and Itai, 1986; Gazit, 1986; Gibbons et al., 1988; Karp and Wigderson, 1985; and Luby, 1986), sorting (Hagerup, 1991; Rajasekharan and Reif, 1989; Reif and Valiant, 1987), list processing (Vishkin, 1984), computational geometry (Reif and Sen, 1989), string matching (Karp and Rabin, 1987), linear algebra (Borodin, von zur Gathen and Hopcroft, 1982; Borodin, Cook and Pippenger, 1983) and load balancing (Gil, 1991; Gil, Matias and Vishkin, 1991). The last is a subroutine used in many efficient parallel algorithms to ensure that all of the processors perform about the same amount of work. We illustrate this simplifying potential of randomization with a problem that has received much attention in this context, the maximal independent set problem (Alon, Babai and Itai, 1986; Karp and Wigderson, 1985; Luby, 1986).

The problem of finding a maximal independent set in a graph has been studied extensively in the context of parallel algorithm design. One reason for this is the fact that this problem arises quite frequently in the design of parallel graph algorithms. Another reason is that this problem is a very easy one to solve by a sequential algorithm, but designing a good parallel algorithm for it places quite a few challenges.

Given an undirected graph, a set of vertices is independent if no pair is connected by an edge. The *maximal independent set problem* is the problem of finding an independent set in an input graph with the property that the set cannot be enlarged into a larger independent set that contains it. This problem has a simple sequential algorithm: fix an ordering of the vertices and examine them in order, adding the examined vertex to the independent set if it contains no edge to a vertex in the set and discarding the vertex otherwise. Unfortunately, this sequential algorithm does not seem to lend itself to parallelization. Although fast, efficient deterministic parallel algorithms have been developed for this problem, these algorithms tend to be fairly complicated. On the other hand, the following simple randomized algorithm gives a fast parallel algorithm for the problem and is more efficient than any of the deterministic NC algorithms known for it.

The randomized algorithm examines each vertex v independently in parallel and assigns the vertex to the independent set with probability $1/(2\delta(v))$, where $\delta(v)$ is the degree of v in the graph. Because this assignment is made simultaneously over all vertices, it is possible that the set constructed is not independent. The algo-

rithm corrects this by examining each edge in the graph, and if the edge has both endpoints in the set, it throws out the vertex with smaller degree (breaking ties randomly). The resulting set is clearly independent. The algorithm then deletes all neighbors of all vertices in the set from the graph and repeats the procedure with the new, smaller graph. It can be shown that the expected number of edges in the new graph is no more than 7/8 the original number of edges. Hence a logarithmic number of stages of this algorithm suffices to construct an independent set that is maximal in the original graph. This gives an efficient randomized algorithm that runs in $O(\log^2 n)$ time using a linear number of processors.

6. ELIMINATING RANDOMIZATION

In some applications, it is desirable or necessary to have a deterministic algorithm. It turns out that some randomized algorithms can be derandomized provided only a limited amount of independence is required. This strategy is applicable to the algorithm described above for the maximal independent set problem. An analysis of that algorithm shows that only pairwise independence is required in the random trials. Hence the algorithm can be made deterministic by searching the entire space in parallel using a quadratic number of processors (Luby, 1986).

An alternate strategy that preserves the number of processors is to compute conditional probabilities while fixing one bit of the output at a time. If k-wise independence suffices for the randomized algorithm, for some constant k, then this gives a deterministic strategy that uses the same number of processors but increases the parallel time by a logarithmic factor. This technique has proved to be a powerful one and has resulted in efficient deterministic NC algorithms for several problems using the technique of first constructing an efficient randomized parallel algorithm with limited independence and then transforming the randomized algorithm into a deterministic one (Berger and Rompel, 1989; Luby, 1988; Motwani, Naor and Naor, 1989).

7. CONCLUSION

This article has described some of the most significant applications of randomization in parallel algorithm design. Randomization has led to fast parallel algorithms and to algorithms that are simple and efficient. These parallel algorithms provide correct solutions with high probability on every input. For most applications, such a performance is almost as satisfactory as a foolproof guarantee. Hence it is not surprising that randomization is a major trend in parallel algorithm design and applications, and we expect the field of randomized parallel algorithms to continue to be a thriving and productive area of research.

ACKNOWLEDGMENT

This work was supported in part by NSF Grant CCR-89-10707.

REFERENCES

AGGARWAL, A. and ANDERSON, R. J. (1987). A random NC algorithm for depth-first search. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing* 325–334. ACM Press, New York.

AGGARWAL, A., ANDERSON, R. J. and KAO, M. (1989). Parallel depth-first search in general directed graphs. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing* 297–308. ACM Press, New York.

ALON, N., BABAI, L. and ITAI, A. (1986). A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms* 7 567–583.

BABAI, L. (1986). A Las Vegas-NC algorithm for isomorphism of graphs with bounded multiplicity of eigenvalues. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science* 303–312. IEEE Computer Society Press, Los Alamitos, CA.

BERGER, B. and ROMPEL, J. (1989). Simulating *logsupcn*-wise independence in NC. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science* 2–7. IEEE Computer Society Press, Los Alamitos, CA.

BORODIN, A., COOK, S. A. and PIPPENGER, N. (1983). Parallel computation for well-endowed rings and space-bounded probabilistic machines. *Inform. and Control* 58 113–136.

BORODIN, A., VON ZUR GATHEN, J. and HOPCROFT, J. E. (1982). Fast parallel matrix and GCD computations. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science* 65–71. IEEE Computer Society Press, Los Alamitos, CA.

COOK, S. A. (1981). Towards a complexity theory of synchronous parallel computation. *Enseign. Math.* 27 99–124.

COOK, S. A. (1985). A taxonomy of problems with fast parallel algorithms. *Inform. and Control* 64 2–22.

GAZIT, H. (1986). An optimal randomized parallel algorithm for finding connected components in a graph. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science* 492–501. IEEE Computer Society Press, Los Alamitos, CA.

GIBBONS, P., KARP, R. M., MILLER, G. and SOROKER, D. (1988). Subtree isomorphism is in Random NC. In *Proceedings of the 3rd Aegean Workshop on Computing* 43–52. Springer, New York.

GIL, J. (1991). Fast load balancing on a PRAM. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing* 10–17. IEEE Computer Society Press, Los Alamitos, CA.

GIL, J., MATIAS, Y. and VISHKIN, U. (1991). Towards a theory of nearly constant time parallel algorithms. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science* 698–710. IEEE Computer Society Press, Los Alamitos, CA.

HAGERUP, T. (1991). Constant-time parallel integer sorting. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing* 299–306. ACM Press, New York.

KARLOFF, H. J. (1986). A Las Vegas RNC algorithm for maximum matching. *Combinatorica* 6 387–392.

KARP, R. M. and RABIN, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM J. Res. Develop.* 31 249–260.

KARP, R. M. and RAMACHANDRAN, V. (1990). Parallel algorithms for shared-memory machines. In *Handbook of Theoretical*

Computer Science, Vol A (J. van Leeuwen, ed.) 869–941. North-Holland, Amsterdam.

KARP, R. M., UPFAL, E. and WIGDERSON, A. (1986). Constructing a perfect matching is in random NC. *Combinatorica* **6** 35–48.

KARP, R. M. and WIGDERSON, A. (1985). A fast parallel algorithm for the maximal independent set problem. *J. Assoc. Comput. Mach.* **32** 762–773.

LUBY, M. (1986). A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.* **15** 1036–1053.

LUBY, M. (1988). Removing randomness in parallel computation without a processor penalty. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science* 162–173. IEEE Computer Society Press, Los Alamitos, CA.

MOTWANI, R., NAOR, J. and NAOR, M. (1989). The probabilistic method yields deterministic parallel algorithms. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science* 8–13. IEEE Computer Society Press, Los Alamitos, CA.

MULMULEY, K., VAZIRANI, U. V. and VAZIRANI, V. V. (1987). Matching is as easy as matrix inversion. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing* 345–354. ACM Press, New York.

RAJASEKHARAN, S. and REIF, J. H. (1989). Optimal and sublogarithmic time randomized parallel algorithms. *SIAM J. Comput.* **18** 594–607.

RAMACHANDRAN, V. (1988). Fast parallel algorithms for reducible flow graphs. In *Concurrent Computations: Algorithms, Architecture, and Technology* (S. K. Tewksbury, B. W. Dickinson and S. C. Schwartz, eds.) 117–138. Plenum Press, New York.

REIF, J. and SEN, S. (1989). Polling: A new randomized sampling technique for computational geometry. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing* 394–404. ACM Press, New York.

REIF, J. H. and VALIANT, L. G. (1987). A logarithmic time sort for linear size networks. *J. Assoc. Comput. Mach.* **34** 60–76.

SCHWARTZ, J. T. (1980). Fast probabilistic algorithms for verification of polynomial identities. *J. Assoc. Comput. Mach.* **27** 701–717.

TUTTE, W. T. (1947). The factorization of linear graphs. *J. London Math. Soc.* **22** 107–111.

VISHKIN, U. (1984). Randomized speed-ups in parallel computation. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing* 230–239. ACM Press, New York.