RANDOMIZED PARALLEL ALGORITHMS†

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

Invited Paper

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 provide some concrete examples where use of randomization give us significant advantage over the best known deterministic parallel algorithms.

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 $1-\epsilon$ $(0 < \epsilon < 1)$, 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

† 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 Information Processing 89, G. Ritter, ed., Elsevier Science Publishers, North Holland, 1989, pp. 455-458.

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, $\epsilon = 1/2$. 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 $O(\sqrt{n}\,logn)$ with high probability. The main algorithm is then used recursively on each of the partitions.

Sorting and Routing in parallel computers

Perhaps nowhere is the impact of randomization felt as strongly as in the case of routing and sorting in fixed interconnection networks (like butterfly, hypercube etc.). Deterministic algorithms for these problems, which have the same asymptotic bounds as the best known randomized algorithms are impractical because of enormous multiplicative constants. Note that routing is an extremely important issue for efficient emulation of shared-memory (which are not feasible architectures) by small diameter inter-connection networks.

Since sorting is a fundamental problem in its own right and a powerful tool for the algorithm designer, let us briefly survey the impact of randomization on parallel sorting. The only deterministic sorting network, (*AKS* network) which sorts in O(logn) depth 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 models.

These models are very popular with algorithm designer 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.

Other applications

In case of various graph-related problems, use of randomization has yielded algorithms which are superior to the best known deterministic algorithms (for example graph-connectivity, integer sorting among others) Moreover, these algorithms are almost always simpler and more practical. As an extreme example, randomization has allowed certain problems to be effectively parallelizable (these are polylogarithmic time algorithms using a polynomial number of processors) for which no NC algorithms are known to this date. Depth-first search and graph matching are examples of problems for which no deterministic NC algorithms are known.

In a large number of cases, randomization has led the way to obtaining better bounds before efficient deterministic algorithms were shown to exist. They provide very powerful tools to tackle a problem and provide good insights to the problem which have often led to the development of deterministic algorithms. For example, Reif introduced a technique called *random-mate* which was exploited by Gazit[3] in his optimal graph-connectivity algorithm and by Miller and Reif[5] for Parallel Tree Contraction. Parallel tree contraction has numerous applications in parallel algorithms. This may give the impression that randomized algorithms are always easier to design; on the contrary, one of the common characteristics of randomized algorithms is that in spite of their simplicity, they present the algorithm designer with tough challenges for its analysis. It must be emphasized that the simplicity makes the job of the programmer easier.

Aesthetics of Randomization

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. Unfortunately, there exists a view that randomized algorithms are not as 'pure' as the deterministic algorithms. Hopefully the above discussion will convince the reader that there is a solid mathematical foundation underlying these techniques (unlike heuristics which are a collection of some empirical techniques). 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.

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.

References

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

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

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

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

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

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

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

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

[11] 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.

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