Guy E. Blelloch and Bruce M. Maggs
Carnegie Mellon University

1 Parallel Algorithms

As more computers have incorporated some form of parallelism, the emphasis in algorithm design has shifted from sequential algorithms to parallel algorithms, i.e., algorithms in which multiple operations are performed simultaneously. As a consequence, our understanding of parallel algorithms has increased remarkably over the past ten years. The most important developments in the field have occurred in three broad areas: parallel models of computation, parallel algorithmic techniques, and parallel complexity theory. This chapter surveys these three areas. So many parallel algorithms have now been designed that a chapter of this length cannot cover even a small fraction of them. Hence, this chapter does not discuss individual algorithms in any detail. The interested reader should consult the following texts: [2, 3, 4, 5, 6, 7].

1.1 Parallel Models of Computation

Developing a standard parallel model of computation for analyzing algorithms has proven difficult because different parallel computers tend to vary significantly in their organizations. In spite of this difficulty, useful parallel models have emerged, along with a deeper understanding of the modeling process. In this section we describe three important principles that have emerged.

1. **Work-efficiency.** In designing a parallel algorithm, it is more important to make it efficient than to make it asymptotically fast. The efficiency of an algorithm is determined by the total number of operations, or *work* that it performs. On a sequential machine, an algorithm's work is the same as its time. On a parallel machine, the work is simply the processor-time product. Hence, an algorithm that takes time t on a P-processor machine performs work $W = Pt$. In either case, the work roughly captures the actual cost to perform the computation, assuming that the cost of a parallel machine is proportional to the number of processors in the machine. We call an algorithm *work-efficient* (or just efficient) if it performs the same amount of work, to within a constant factor, as the fastest known sequential algorithm. For example, a parallel algorithm that sorts n keys in $O(\sqrt{n}\log n)$ time using $\sqrt{n}$ processors is efficient since the work, $O(n \log n)$, is as good as any (comparison-based) sequential algorithm. However, a sorting algorithm that runs in $O(\log n)$ time using n^2 processors is not efficient. The first algorithm is better than the second – even though it is slower – because it's work, or cost, is smaller. Of course, given two parallel algorithms that perform the same amount of work, the faster one is generally better.

2. **Emulation.** The notion of work-efficiency leads to another important observation: a model can be useful without mimicking any real or even realizable machine. Instead, it

suffices that any algorithm that runs efficiently in the model can be translated into an algorithm that runs efficiently on real machines. As an example, consider the widely-used parallel random-access machine (PRAM) model. In the PRAM model, a set of processors share a single memory system. In a single unit of time, each processor can perform an arithmetic, logical, or memory access operation. This model has often been criticized as unrealistically powerful, primarily because no shared memory system can perform memory accesses as fast as processors can execute local arithmetic and logical operations. The important observation, however, is that for a model to be useful we only require that algorithms that are efficient in the model can be mapped to algorithms that are efficient on realistic machines, not that the model is realistic. In particular, any algorithm that runs efficiently in a P-processor PRAM model can be translated into an algorithm that runs efficiently on a P/L-processor machine with a latency L memory system[1], a much more realistic machine. In the translated algorithm, each of the P/L processors emulates L PRAM processors. The latency is "hidden" because a processor has useful work to perform while waiting for a memory access to complete. Although the translated algorithm is a factor of L slower than the PRAM algorithm, it uses a factor of L fewer processors, and hence is equally efficient.

3. **Modeling Communication.** To get the best performance out of a parallel machine, it is often helpful to model the communication capabilities of the machine, such as its latency, explicitly. The most important measure is the communication bandwidth. The bandwidth available to a processor is the maximum rate at which it can communicate with other processors or the memory system. Because it is more difficult to hide insufficient bandwidth than large latency, some measure of bandwidth is often included in parallel models. Sometimes the specific topology of the communication network is modeled as well. Although including this level of detail in the model often complicates the design of parallel algorithms, it's essential for designing the low-level communication primitives for the machine. In addition to modeling basic communication primitives, other operations supported by hardware, including synchronization and concurrent memory accesses, are often modeled, as well as operations that mix computation and communication, such as fetch-and-add and scans. A final consideration is whether the machine supports shared memory, or whether all communication relies on passing messages between the processors.

1.2 Algorithmic Techniques

A major advance in parallel algorithms has been the identification of fundamental algorithmic techniques. Some of these techniques are also used by sequential algorithms, but play a more prominent role in parallel algorithms, while others are unique to parallelism. Here we list some of these techniques with a brief description of each.

[1]The latency of a memory system is the time from when a processor makes a request to the memory system to when it receives the result.

1. **Divide-and-Conquer**. Divide-and-conquer is a natural paradigm for parallel algorithms. After dividing a problem into two or more subproblems, the subproblems can be solved in parallel. Typically the subproblems are solved recursively and thus the next divide step yields even more subproblems to be solved in parallel. For example, suppose we want to compute the convex-hull of a set of n points in the plane (i.e., compute the smallest convex polygon that encloses all of the points). This can be implemented by splitting the points into the leftmost $n/2$ and rightmost $n/2$, recursively finding the convex hull of each set in parallel, and then merging the two resulting hulls. Divide-and-conquer has proven to be one of the most powerful techniques for solving problems in parallel with applications ranging from linear systems to computer graphics and from factoring large numbers to n-body simulations.

2. **Randomization**. The use of random numbers is ubiquitous in parallel algorithms. Intuitively, randomness is helpful because it allows processors to make local decisions which, with high probability, add up to good global decisions. For example, suppose we want to sort a collection of integer keys. This can be accomplished by partitioning the keys into buckets then sorting within each bucket. For this to work well, the buckets must represent non-overlapping intervals of integer values, and contain approximately the same number of keys. Randomization is used to determine the boundaries of the intervals. First each processor selects a random sample of its keys. Next all of the selected keys are sorted together. Finally these keys are used as the boundaries. Such random sampling is also used in many parallel computational geometry, graph, and string matching algorithms. Other uses of randomization include symmetry breaking, load balancing, and routing algorithms.

3. **Parallel Pointer Manipulations**. Many of the traditional sequential techniques for manipulating lists, trees, and graphs do not translate easily into parallel techniques. For example techniques such as traversing the elements of a linked list, visiting the nodes of a tree in postorder, or performing a depth-first traversal of a graph appear to be inherently sequential. Fortunately, each of these techniques can be replaced by efficient parallel techniques. These parallel techniques include pointer jumping, the Euler-tour technique, ear decomposition, and graph contraction. For example, one way to label each node of an n-node list (or tree) with the label of the last node (or root) is to use pointer jumping. In each pointer-jumping step each node in parallel replaces its pointer with that of its successor (or parent). After at most $\log n$ steps, every node points to the same node, the end of the list (or root of the tree).

4. **Others**. Other useful techniques include finding small graph separators for partitioning data among processors to reduce communication, hashing for balancing load across processors and mapping addresses to memory, and iterative techniques as a replacement for direct methods for solving linear systems.

These techniques have led to efficient parallel algorithms in most problem areas for which efficient sequential algorithms are known. In fact, some of the techniques originally developed for parallel algorithms have led to improvements in sequential algorithms.

1.3 Parallel Complexity Theory

Researchers have developed a theory of the parallel complexity of computational problems analogous to the theory of NP-completeness. A problem is said to belong to the class NC (Nick's Class) if it can be solved in time polylogarithmic in the size of the problem using at most a polynomial number of processors. The class NC in parallel complexity theory plays the role of P in sequential complexity, i.e., the problems in NC are thought to be tractable in parallel. Examples of problems in NC include sorting, finding minimum-cost spanning trees, and finding convex hulls. A problem is said to be P-complete if it can be solved in polynomial time and if its inclusion in NC would imply that $NC = P$. Hence, the notion of P-completeness plays the role of NP-completeness in sequential complexity. (And few believe that $NC = P$.) Examples of P-complete problems include finding a maximum flow and finding a lexicographically minimum independent set of nodes in a graph. Much early work in parallel algorithms aimed at showing that certain problems belonged to the class NC (without considering the issue of efficiency). This work tapered off, however, as the importance of work-efficiency became evident. Also, even if a problem is P-complete, there may be efficient (but not necessarily polylogarithmic time) parallel algorithms for solving it. For example, several efficient and highly parallel algorithms are known for solving the maximum flow problem, which is P-complete.

1.4 Current and Future Directions

Recently the emphasis of research on parallel algorithms has shifted to pragmatic issues. The theoretical work on algorithms has been complemented by extensive experimentation. This experimental work has yielded insights into how to build parallel machines [1], how to make parallel algorithms perform well in practice [8], how to model parallel machines more accurately, and how to express parallel algorithms in parallel programming languages.

Two effective parallel programming paradigms have emerged: control-parallel programming and data-parallel programming. In a control-parallel program, multiple independent processes or functions may execute simultaneously on different processors and communicate with each other. Some of the most successful control-parallel programming systems include Linda, MPI, and PVM. In each step of a data-parallel program an operation is performed in parallel across a set of data. Successful data-parallel programming languages include *Lisp, NESL, and HPF. Although the data-parallel programming paradigm might appear to be less general than the control-parallel paradigm, most parallel algorithms found in the literature can be expressed more naturally using data-parallel constructs.

There has also been a focus on solving problems from applied domains, including computational biology, astronomy, seismology, fluid dynamics, scientific visualization, computer-aided design, and database management. Interesting algorithmic problems arising from these domains include generating meshes for finite element analysis, solving sparse linear systems, solving n-body problems, pattern matching, ray tracing, and many others.

Commodity personal computers with multiple processors have begun to appear on the market. As this trend continues, we expect the use of parallel algorithms to increase dramatically.

References

[1] George Almasi and Allan Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings, Redwood City, CA, 1994.

[2] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[3] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley, Reading, Mass., 1992.

[4] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared memory machines. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science—Volume A: Algorithms and Complexity.* MIT Press, Cambridge, Mass., 1990.

[5] Vipin Kumar, Ananth Grama, Anshul Gupta and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings, Redwood City, CA, 1994.

[6] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes.* Morgan Kaufmann, San Mateo, CA, 1992.

[7] John H. Reif (editor). *Synthesis of Parallel Algorithms.* Morgan Kaufmann, San Mateo, California, 1993.

[8] Gary W. Sabot (editor). *High Performance Computing: Problem Solving with Parallel and Vector Architectures.* Addison Wesley, Reading, MA, 1995.