Parallel Combinatorial Computing

Richard M. Karp

TR-91-006

January, 1991

Computer Science Division,[1,2,3]
University of California,
Berkeley, CA 94720
and
International Computer Science Institute,
Berkeley, CA

Abstract

In this article we suggest that the application of highly parallel computers to applications with a combinatorial or logical flavor will grow in importance. We briefly survey the work of theoretical computer scientists on the construction of efficient parallel algorithms for basic combinatorial problems. We then discuss a two-stage algorithm design methodology, in which an algorithm is first designed to run on a PRAM and then implemented for a distributed-memory machine. Finally, we propose the class of node expansion algorithms as a fruitful domain for the application of highly parallel computers.

---

[1]Research supported by NSF Grant No. CCR-9005448. karp@kazoo.Berkeley.EDU.

[2]To be published in the proceedings of the Conference on Very Large Scale Computations in the 21st Century, sponsored by Thinking Machines Corporation.

[3]I would like to thank Ottavia Bassetti of Thinking Machines Corporation for her invaluable help in the preparation of this paper.

1 Introduction

Over the past fifteen years theoretical computer scientists have been at work developing the foundations for the systematic study of parallel computers and parallel algorithms. In this article I would like to convey some of the insights of lasting value that have come out of this investigation of abstract models of parallel computation, to relate these studies to the practice of parallel computation, and, in particular, to draw attention to combinatorial search problems as a promising domain for the application of massively parallel computers.

To set the stage, let me briefly sketch some of the recent history of parallel computation. The earliest successes have come in the field of scientific computing. The reasons for this are fairly obvious. First of all, there are many urgent numerical problems in science and engineering that require vast computing resources. Secondly, it is often very easy to see that these problems can benefit from parallel execution: the quantities to be computed have a natural spatial or temporal relationship that immediately suggests a way of decomposing the problem into relatively independent parts that can be solved concurrently. Putting it another way, these problems are ripe for the "data parallel" approach that the Connection Machine is designed to exploit.

The problems in scientific computing that have yielded most easily to parallel computation often have computational kernels with a very simple and regular structure. Examples of such kernels are the Fast Fourier Transform, dense matrix multiplication, the solution of tridiagonal systems of linear equations and relaxation processes over fairly simple geometries. Algorithms for the solution of these kernel problems are very compactly expressible using the standard notation of algebra and can be implemented using simple data structures such as arrays. The computations tend to be oblivious rather than adaptive: i.e., the sequence of computation steps and data accesses to be performed is highly regular and predictable in advance, rather than being determined at run time according to the outcomes of conditional branches. The regularity and predictability of these computations simplifies the assignment of tasks to processors and data to memory modules, permits the steps in the computation to be scheduled at compile time rather than execute time, and makes it a relatively easy matter to rescale these algorithms as the number of processors and memory modules grow. Finally, the fact that a few basic kernels are useful in a wide range of applications makes it possible to build up a library of careful parallel implementations of kernel problems, and then to deal with more complex applications by composing library routines together.

The above description of parallel scientific computation is admittedly rather simplistic. It applies best to the earliest parallel numerical applications, in which the most evident opportunities for parallelism were being exploited. As the field has progressed the geometries being dealt with have become more complex, and the algorithms, more intricate and adaptive. Still, it is fair to say that, on the whole, parallel computation has been easier to exploit in the realm of scientific computing than in other application areas.

My main message in this article is that, in the coming decades, the applications of massively parallel computation will grow in diversity, and scientific computation will no longer be so dominant. Instead, a wide range of other applications having a combinatorial or logical flavor will come into prominence. I have in mind such areas as combinatorial optimization, theorem-proving, symbolic algebraic manipulation, query processing and inference in database systems and knowledge-based systems, DNA sequence analysis, image understanding, and natural language processing. The attributes of these combinatorial/logical problems are quite different from the simple regular properties of many numerical algorithms. They tend to involve the manipulation of discrete structures such as graphs, strings and symbolic expressions. In general, the array data structure is not adequate for these applications; instead, complex, pointer-based data structures are required. In these applications the "shape" of a computation - i.e., the pattern of steps to be executed and data to be accessed- tends to evolve dynamically and unpredictably. This means that decisions about the allocation of tasks to processors and data to memory modules have to be made at run time rather than compile time. The parallel architectures appropriate for this class of applications may well be very different from the parallel architectures that are suitable for numerical applications. For example, vector machines and systolic arrays, which are so well suited to many highly regular numerical tasks, may prove to be rather ineffective at solving combinatorial or logical problems. The data-parallel paradigm for decomposing a problem into loosely interacting, concurrently executable parts may be less appropriate in the combinatorial realm than it is in the numerical realm. And, because of the diversity of combinatorial applications, it is no easy matter to identify a small set of basic computational kernels out of which a wide range of useful parallel combinatorial algorithms can be constructed. It is also worth noting that, as the problems attacked in scientific computing become more complex and the algorithms more adaptive, numerical computations cease to follow simple, regular and predictable patterns, and take on some of the attributes associated with combinatorial applications.

As the range of applications for very large scale parallel computing widens it becomes necessary to reexamine continually the choice of machine architectures, programming languages and programming environments, as well as the question of how parallel algorithms should be represented and evaluated. The research of the last fifteen years in theoretical computer science can provide some insight into these questions, particularly the ones related to the choice of parallel architectures and the representation and evaluation of algorithms. Because theoreticians work with abstract machines rather than real ones they have the advantage of being completely free to cast aside existing technology and define whatever abstract machines best capture the essential features of parallel computation; on the other hand, the freedom that theoreticians enjoy may tempt them to ignore crucial technological constraints and thus work with models of limited relevance. I believe that, on the whole, the theoretical work has been done with good taste, and that the abstract models we have created provide a great deal of insight into the kinds of parallel architectures and algorithms that will prevail by the beginning of the next century.

2 Abstract Distributed-Memory Machines

The theoretical work has converged on two classes of abstract machines: *distributed-memory machines* and *shared-memory machines.* A distributed-memory machine consists of a large, sparsely interconnected network of processors and memory modules; for simplicity, we will consider the case where the links in the network run between processors, and each memory module is the "local memory" of some processor. Each link in the network is a communication channel between two processors; it is capable of transmitting packets that typically represent data, requests for data access, or information about the status of a processor. In order for the processors to work in unison they must exchange data and status information via communication links. Since a processor may require status information from a remote processor or data from a remote memory module, the entire system must operate as a kind of post office, with each processor capable of forwarding packets addressed to other processors.

Each processor can be thought of as a simple sequential computer capable of executing arithmetic and logical instructions, issuing fetch and store instructions involving either its local memory or more remote memory modules, and forwarding packets addressed to other processors. In theoretical studies it is often assumed for convenience that each processor has its own program, which may be completely different from the programs of the other processors. In practice the creation of a completely separate program for each processor would be impractical, and instead it is necessary to resort to one of two approaches. In the MIMD (Multiple-Instruction Multiple-Data) approach each processor has its own program and program control unit, but all the programs are identical except for a few parameters, such as the processor's ID number. These parameters give each processor enough information to determine its individual role in the overall computation. In the SIMD (Single-Instruction Multiple-Data) approach there is a single program control unit which, at each instruction cycle, broadcasts the same instruction to all the processors; however, the instruction contains a number of fields which instruct each processor how to individuate its behavior according to the contents of its own registers. The distinction between the MIMD and SIMD approaches appears not to be crucially important, except that a SIMD machine is inherently synchronous, while a MIMD machine may be either synchronous or asynchronous.

An important factor in determining the performance of a distributed-memory machine is *latency*—the delay between the issuance of a request for data stored in a remote memory module and the actual receipt of that data. Latency occurs because of the time to route a read or write request from the requesting processor to the appropriate memory module, the time that the request waits in a queue while other requests to the same module are being processed and, in the case of a read request, the time to route data back to the requesting processor. In order to minimize the effects of latency, great care must be exercised in the assignment of computational tasks to processors and data to memory modules, so that no memory module is too heavily loaded and, as often as possible, processors can access needed data from their own local memories or from the memories of nearby processors.

The *topology*, or interconnection pattern, of a sparsely interconnected network of processors is of crucial importance in determining the delays that access requests will experience. When the number of processors is very small it is convenient to arrange them in a ring or along a linear bus. For certain special applications such as image processing, the natural geometry of the problem suggests an arrangement of the processors in a two-dimensional or three-dimensional mesh. For general-purpose applications involving a large number of processors, it seems best to choose an interconnection structure in which each processor has a small number of neighbors, and yet the diameter of the network is logarithmic in the number of processors. A typical example of such a structure is the Butterfly network. The number of processors in the Butterfly is of the form $n \times 2^n$, where n is a positive integer. Each processor is identified by a pair (i, x), where i is an integer between 0 and $n-1$ and x is an n-bit string. From each processor (i, x) there are links to $(i+1 \bmod n, x)$ and $(i+1 \bmod n, x')$, where x' is obtained by toggling the ith bit of x. It is easy to see that, for any two processors (i_1, x_1) and (i_2, x_2), there is a path of length at most $2n-1$ from (i_1, x_1) to (i_2, x_2). Further examples of low-diameter interconnection networks are discussed in the survey article [V90].

3 Abstract Shared-Memory Machines

A second abstract model, considerably more remote from reality, is the PRAM (Parallel Random Access Machine). The PRAM consists of a set of processors communicating through a single, monolithic shared memory. It is assumed that the time required to access a memory location is a constant, independent of the cell being accessed and the processor doing the accessing. A PRAM repeatedly executes a cycle in which the processors, operating synchronously, go through a Read step, an Execute step, and a Write step. In the Read step each processor reads into a register the contents of a cell in the shared memory. In the Execute step each processor executes a single arithmetic or logical instruction. In the Write step, each processor writes the contents of one of its registers into a cell in the shared memory. PRAM models differ according to how they handle *read conflicts*, in which two or more processors try to read the contents of the same cell at the same time, and *write conflicts*, in which two or more processors try to write into the same cell at the same time. An Exclusive-Read Model disallows concurrent reads, while a Concurrent-Read model permits them. An Exclusive-Write Model disallows concurrent writes, while a Concurrent-Write model permits concurrent writes and has some definite convention determining the value that gets stored when two or more processors try to write concurrently into the same cell. A program running in time T and using p processors on any one of these models can be simulated on the weakest model (Exclusive-Read Exclusive-Write) in time $T \log p$ using p processors. Thus the differences among the PRAM models are of limited importance and will be ignored in the present discussion.

We shall also gloss over the distinction between deterministic PRAMs and randomized ones. A randomized PRAM has access to a supply of independent, unbiased random bits, and thus may be viewed as "tossing coins" in the course of its execution. A randomized algorithm is not required to produce a correct result with certainty, but only with extremely high probability. There is a further distinction between *Las Vegas algorithms*, which never produce incorrect results but may on rare occasions fail to produce a result at all, and *Monte Carlo algorithms*, which may occasionally produce an incorrect result.

Theoreticians often assume for the sake of generality that each processor in a PRAM has its own completely distinct program, but, in practice, a shared-memory machine would be programmed using either the MIMD approach or the SIMD approach described above.

Because of its unrealistic assumption that the time required for a memory access is constant, the PRAM model cannot distinguish between accesses to a processor's local memory and remote accesses, and thus loses all ability to model issues related to the distribution of data among memory modules to minimize latency. Furthermore, the PRAM model assumes that, in every cycle, each processor accesses the shared memory both to read and to write. This encourages the construction of algorithms that are too fine-grained; i.e., algorithms in which communication between processors via the shared memory occurs too frequently. In practice, one would prefer coarser-grained algorithms, in which the great majority of a processor's memory accesses are to its own local memory, and interprocessor communication is relatively infrequent. Several attempts have been made to refine the PRAM model by making an explicit distinction between local and global memory, and by assuming that there is no global clock, so that processors are obliged to engage in a synchronization protocol before exchanging data via the global memory.

Despite its excessive simplicity the PRAM model has proved to be a very useful formalism for representing parallel algorithms and studying their computational complexity. It permits the algorithm designer to focus on the logical structure of a problem, leaving for later consideration the issues of communication, latency and synchronization. It is often useful to design an algorithm initially for a PRAM, and then to convert the algorithm to one that works efficiently on a particular distributed-memory machine.

4 Efficient PRAM Algorithms

In designing algorithms for the PRAM it is convenient to assume initially that the number of processors is unlimited. A parameter n represents the size of the input (often measured by the number of bits of input data), and the complexity of an algorithm is measured by its worst-case consumption of two resources: *time* and *work*. The computation time $T(n)$ gives the worst-case number of machine cycles as a function of the input size n. The work $W(n)$ gives the worst-case number of operations as a function of input size, taking into account not only the number of cycles but also the number of active processors in each cycle. The reason that it is attractive to design algorithms without worrying about how many processors are actually available, is that there is a general principle, attributed to Brent, which says that one can slow down a computation appropriately for the number of processors. Brent's principle states that, under certain weak hypotheses, if an algorithm consumes time $T(n)$ and work $W(n)$ when the number of processors is unlimited, then it can be executed on a p-processor PRAM in time $T(n)+\frac{W(n)}{p}$. The idea is to simulate the original algorithm cycle by cycle; a cycle in which, P processors are active is simulated using p processors in time $\lceil\frac{P}{p}\rceil$ by having each of the p simulating processors do the work of at most $\lceil\frac{P}{p}\rceil$ simulated processors.

In view of Brent's principle we shall evaluate PRAM algorithms according to two measures of difficulty: the time $T(n)$ and the work $W(n)$. We are particularly interested in parallel algorithms in which $W(n)$, the number of operations performed, is not much greater than $S(n)$, the number of steps executed by the most practical sequential algorithm known for the same problem. A convenient way to put this condition is to require that there exists a constant k, such that $W(n) \leq S(n)(\log(S(n))^k$. A parallel algorithm fulfilling this condition will be called *efficient.* We are interested in efficient algorithms that minimize the time $T(n)$. The number of processors that can be effectively exploited is then (up to logarithmic factors) $\frac{W(n)}{T(n)}$.

Somewhat surprisingly, efficient parallel algorithms for which $T(n)$ is at most $\log n$ have been devised for many of the key combinatorial problems that may be expected to find frequent use as subroutines. We describe a number of these problems, in each case giving $S(n)$, $W(n)$ and $T(n)$; in stating these bounds we omit constant multiplicative factors, which in general depend on fine details of the model of computation. Some of the parallel time bounds we cite can only be achieved if a CRCW PRAM is used, and some of them require randomized algorithms. Further details about these points can be found in the survey articles [EG88] and [KR90], and in the papers cited in those references.

- *Merging*
  Input Two increasing sequences of n integers.
  Output A single ordered sequence containing the elements of the two given sequences.
  $S(n) = n$, $W(n) = n$, $T(n) = \log\log n$

- *Maximum*
  Input An array of n integers.
  Output The largest element of the array.
  $S(n) = n$, $W(n) = n$, $T(n) = \log\log n$.

- *String Matching*
  Input Two strings: x, the *pattern* and y, the *text*, where the length of x is less than or equal to the length of y, and the sum of the lengths of the two strings is n.
  Output An indication of all places where x occurs as a consecutive block within y.
  $S(n) = n$, $W(n) = n$, $T(n) = \log\log n$.

- *Sorting*
  Input An array of n integers.
  Output An array in which the elements of the input array are arranged

in nondecreasing order.
$S(n) = n \log n$, $W(n) = n \log n$, $T(n) = \log n$.

- *Prefix Sums*
  Input An array $(x_1, x_2, \ldots, x_n)$ of elements drawn from a set on which an associative binary operation $\circ$ is defined; for example, $\circ$ might denote $+$, max or min.
  Output The array $(x_1, x_1 \circ x_2, \ldots, x_1 \circ x_2 \circ \ldots \circ x_n)$ of "partial sums."
  $S(n) = n$, $W(n) = n$, $T(n) = \log n$.

- *List Ranking* This is a variant of the prefix sums problem in which the elements are stored in a linked list rather than an array.
  Input A linked list containing the successive elements $x_1, x_2, \ldots, x_n$.
  Output The same linked list, but with element x_i replaced by $x_1 \circ x_2 \ldots \circ x_i$.
  $S(n) = n$, $W(n) = n$, $T(n) = \log n$.

  The following is a typical application of list ranking: given a linked list in which each element is marked "dead" or "live," construct an array consisting of the live elements, in the same order as they occur in the linked list.

- *Formula Evaluation*
  Input An arithmetic formula presented as a rooted binary tree in which the n leaves represent constants and the $n-1$ internal nodes are labeled with binary operators from the set $\{+, -, \times, \div\}$.
  Output The value of the formula.
  $S(n) = n$, $W(n) = n$, $T(n) = \log n$.
  Note: The result is not valid for arithmetic expressions represented as directed acyclic graphs rather than trees (the difference being that, in the case of directed acyclic graphs, common subexpressions may occur).

- *Maximal Independent Set* A set S of vertices in a graph is called a *maximal independent set* if no two vertices in S are adjacent and every vertex not in S is adjacent to a vertex in S.
  Input A graph with n edges.
  Output A maximal independent set.
  $S(n) = n$, $W(n) = n$, $T(n) = \log n$.

The maximal independent set problem typically arises in connection with the scheduling of concurrent activities. In such an application the vertices of the graph represent activities to be performed, and an edge between two vertices indicates that the corresponding activities interfere in some way, and thus cannot be performed concurrently. A maximal independent set then gives a maximal set of concurrently executable activities.

- *Connected Components*
  Input A graph with n edges.
  Output The connected components of the graph.
  $S(n) = n$, $W(n) = n$, $T(n) = \log n$.

- *Biconnected Components* A connected graph G is called *biconnected* if it cannot be disconnected by the removal of a single vertex. A maximal biconnected subgraph of G is called a *biconnected component.*
  Input A connected graph G with n edges.
  Output The biconnected components of G.
  $S(n) = n$, $W(n) = n$, $T(n) = \log n$.

  The standard sequential method of computing connected or biconnected components uses a technique called *depth-first search.* Interestingly, it is not known how to perform depth-first search efficiently in parallel, and an entirely different technique known as *ear decomposition* is used instead for the construction of efficient parallel algorithms to find connected and biconnected components.

- *Transitive Closure*
  Input A directed graph G with n vertices.
  Output A directed graph G^* with the same vertex set as G, such that, for any two vertices u and v, G^* has an edge from u to v if and only if G has a path from u to v.
  $S(n) = n^3$, $W(n) = n^3 \log n$, $T(n) = \log n$. (Sequential algorithms are known with asymptotic time bounds below n^3, but they are not suitable for practical use).

Certain important problems seem less amenable to parallelization than the ones listed above. The problem of solving n linear equations in n unknowns does not seem to be solvable efficiently in parallel unless $T(n)$ is allowed to grow at least linearly with n. Gaussian elimination requires sequential time n^3 (other algorithms with better asymptotic running times are known, but they are seldom used in practice). Since Gaussian elimination consists of n successive pivots, each requiring about n^2 arithmetic operations, we can easily achieve $W(n) = n^3$, $T(n) = n$. However, all the known parallel algorithms with $T(n)$ much smaller than n entail a blowup in the amount of work. For example, if we require that $T(n)$ be polylog in n (i.e., bounded by a fixed power of $\log n$), we seem to require $n^{3.5}$ processors. The basic processes of breadth-first search and depth-first search in directed graphs also pose difficulties; no methods are known for carrying out these processes efficiently in polylog parallel time. For the problem of computing a maximum flow in an n-node network, the best asymptotic sequential time bound known is n^3; in order to achieve a parallel algorithm whose work is within a polylog factor of this bound we seem to require $T(n) \geq n^2 \log n$.

5 P-Complete Problems

The theory of parallel algorithms involves not only the design and analysis of algorithms but also the development of results indicating that certain problems are hard to parallelize. The theory of P-completeness is a powerful tool for developing such evidence; it plays a role analogous to the theory of NP-completeness in sequential computation. We shall give an informal sketch of the definitions underlying this theory. A computational problem will be specified as an input-output map. Since the inputs and outputs can be taken to be binary strings, a problem is formally a function from $\{0,1\}^*$ to $\{0,1\}^*$, where $\{0,1\}^*$ denotes the set of all finite strings of zeros and ones. The complexity class FP consists of all those problems solvable sequentially in polynomial time. The complexity class NC consists of all those functions computable on a PRAM in polylog time with polynomial-bounded work. Thus FP can be viewed as the collection of all problems solvable rapidly by sequential algorithms, and NC is a subset of FP which contains all those problems in FP which admit of efficient parallel algorithms that run in polylog time.

Problem $A : \{0,1\}^* \rightarrow \{0,1\}^*$ is said to be *NC-reducible* to problem $B : \{0,1\}^* \rightarrow \{0,1\}^*$ if there is a function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ in NC such that, for every input x, $A(x) = B(f(x))$ It follows that, if A is NC-reducible to B and B lies in NC, then A also lies in NC. Problem A is called P-*complete* if it lies in FP and every problem in FP is logspace-reducible to it. Thus the P-complete problems can be viewed as the problems in FP least likely to lie in NC; more precisely, a P-complete problem lies in NC only in the unlikely event that $FP = NC$; i.e., in the event that every problem solvable by a fast sequential algorithm is also solvable by a fast parallel algorithm whose work is polynomial-bounded. In practice, the P-completeness of a problem is taken as evidence that the problem is unlikely to lie in NC, and thus certainly unlikely to be solvable by an efficient parallel algorithm that runs in polylog time. Examples of P-complete problems are the max-flow problem, the linear programming problem and the problem of evaluating the outputs of an acyclic circuit composed of Boolean gates, given the values of the inputs.

6 A Two-Stage Method of Parallel Algorithm Design

Theoreticians tend to favor a two-stage method of parallel algorithm design. The first stage is to develop a PRAM algorithm for which the work $W(n)$ and the parallel time $T(n)$ are acceptably small, assuming that an unlimited number of processors is available. The second stage is then to implement this algorithm on a particular distributed-memory system. The primary goals in the implementation stage are to reduce the frequency of interprocessor communication and, when latency is unavoidable, to minimize its effects. Each processor used in the PRAM algorithm can be viewed as a *virtual processor* whose work has to be done by one of the limited number of physical processors of the distributed-memory machine. In addition to the mapping of virtual processors onto physical ones, each piece of data occurring in the problem must be assigned to the local memory of some processor. Finally, the routing of packets through the processor network must be managed.

Interesting theoretical work has been done showing that messages can be routed efficiently using randomization. Several authors (see the survey article [V90]) have considered *permutation routing* as a model problem. In this problem it is assumed that each processor in the network has exactly one packet to send and each processor will be the recipient of exactly one packet.

Thus the map from sources to destinations is a permutation. It is assumed that each link of the network can transmit one packet per unit time. The goal is to have all the packets reach their destinations in time proportional to the diameter of the network; and the basic difficulty is *congestion*, which occurs when many messages queue up, waiting to be transmitted along the same link. For several logarithmic-diameter networks, including the Hypercube and the Butterfly, it has been shown that a simple randomized scheme will succeed, with high probability, in routing any permutation in time bounded by a small constant times the diameter of the network. Roughly, the approach is as follows. Each processor chooses a random node in the network as the intermediate destination for its packet. Each message is routed along a shortest path from its source to its intermediate destination, and then along a shortest path from its intermediate destination to its final destination.

Another fundamental problem is sorting on distributed-memory networks. A randomized algorithm called Flashsort [RV87] is capable of sorting p items on a p-processor Butterfly network in time $O(\log p)$, assuming that each processor initially has one of the p items in its local memory.

Considerable effort has been devoted to the emulation of a PRAM on a distributed-memory machine. A major breakthrough is due to Ranade [R87], who gave a randomized algorithm for emulating a p-processor PRAM on a p-processor Butterfly, in such a way that, with high probability, the emulation of each cycle of the PRAM will be completed within time $O(\log p)$; i.e., in time proportional to the diameter of the Butterfly. In order to avoid memory contention, which occurs when read and write requests destined for the same memory module collide, Ranade's emulation randomizes the allocation of data to memory modules. Ranade's method also uses a novel method of combining access requests to the same memory cell whenever such requests collide in the process of being routed to their common destination.

It is also known that a $p \log p$-processor PRAM operating in time T can be emulated (with high probability) by a p-processor Butterfly operating in time $O(T \log p)$.The emulation is based on the principle of pipelining. Each processor of the Butterfly is assigned the task of emulating $\log p$ processors of the PRAM. Because each physical processor has many virtual processors to emulate it can continue issuing read and write requests even while its previous requests are being processed. Thus, even though each individual request may experience latency proportional to $\log p$, the emulating processor remains active during the latency period. The principle of covering latency by having a physical processor time-share itself among many independent processes is fundamental in practice as well as in theory. Several current efforts in multiprocessor design focus on efficient implementation of the context switching that occurs when a processor shifts its attention from one process to another.

The emulation results we have been describing seem to support the view that useful parallel programs can be developed by a two-stage process in which one first obtains an algorithm for a PRAM with an unlimited number of processors and then uses a generic emulation technique to implement the algorithm on a distributed-memory machine. In particular, the second emulation result shows that, if a problem can be solved efficiently on a $p \log p$-processor PRAM , then it can be solved efficiently on a p-processor distributed-memory machine. This approach to parallel program design via generic PRAM simulation is viewed with great suspicion, particularly in the numerical analysis community. Some of the objections are technical in nature: the emulations require randomization, the estimation of time bounds neglects constant factors that may be all-important in practice, and the target machine of the emulation is an abstract distributed-memory machine rather than a real multiprocessor system. Beyond this, there is the intuition that good parallel algorithms must be hand-crafted in order to match the structure of a problem to the machine on which it is to be solved, and that generic emulation methods will inevitably be too crude. Time and considerable experimentation will be needed to determine the practical value of these emulation techniques.

7 Node Expansion Algorithms

In the rest of the paper I will focus on a class of combinatorial algorithms that possess a great deal of inherent parallelism and could provide an important new area for the application of parallel computers. I will call them *node expansion algorithms.* These algorithms are often based on tree-search or divide-and-conquer principles, and their distinguishing feature is the use of recursive procedure calls to construct a set of interrelated computational tasks which can be represented as the nodes of a rooted tree. Each task A is either executed directly or generates "children" $A_1, A_2, \ldots, A_d$ whose results may be needed for the completion of task A. The tree of tasks grows dynamically during the execution of the algorithm. Initially, only the root of the tree is given, representing the initial computational task. The primitive step is *node expansion*, in which a node is either processed directly or spawns a set of children. Node expansion algorithms arise in many different sorts of applications: examples are backtrack search, game tree search, branch-and-bound computation, theorem-proving, the execution of PROLOG programs, ray tracing in graphics, and the solution of partial differential equations by adaptive mesh refinement.

Backtrack search is perhaps the simplest application leading to node expansion algorithms. We illustrate backtrack search with a toy example, the "Eight Queens" problem, in which we want to enumerate all the ways to place eight queens on a chessboard so that no two of them attack each other. In this case, each node in the search tree represents a partial placement of mutually nonattacking queens on the board. The task associated with a node is to enumerate all the complete placements (i.e., placements of eight queens) that include the given partial placement. The root of the tree is the empty board. A node gets expanded by observing that the node represents a complete placement, by determining that the associated partial placement cannot be extended to a complete placement, or by spawning children corresponding to all the legal ways of placing one additional queen on the first unoccupied rank of the chessboard.

Game tree search is another well-known example, which comes up in connection with algorithms to play two-person games of perfect information such as chess or Go. Each node in the search tree represents a board position. The root represents the initial position in which a move is to be chosen. A node is expanded either by determining that it is a terminal node in the search tree or by generating a child for each new position created by making a legal move in the position represented by the node. Terminal nodes typically correspond to "quiescent" positions, which lend themselves to static evaluation without further search. The value of a nonterminal position represents its desirability from the point of view of one of the players (say, the first player). The value of a nonterminal position in which the first player is to move is the maximum of the values of the children; the value of a position in which the second player is to move is the minimum of the values of the chidren. As the tree is generated, values can be assigned to positions, and certain nodes can be pruned away because their values can be shown to be irrelevant to the evaluation of the root. The search terminates when the root gets evaluated and the correct move at the root is determined.

Certain backward-chaining theorem-proving methods for the propositional calculus are formally similar to game tree search. Each node represents a statement to be proved or disproved; a node gets expanded either by directly determining the truth value of the statement to be proven or by expressing that statement as a conjunction or disjunction of new statements, which become the children of the current node.

The branch-and-bound method of combinatorial optimization is an important source of node expansion algorithms. In general, a combinatorial optimization problem takes the form: minimize $f(x)$, where f is a given function called the *objective function* and x is constrained to lie in some discrete set called the *feasible set.* The elements of the feasible set are called *feasible solutions.* The minimizing x is called the *optimal solution* to the problem, and the corresponding function value is called the *optimal value* of the problem. One standard example is zero-one integer programming, in which the feasible solutions are the n-dimensional vectors of zeros and ones which satisfy a given set of linear inequalities, and the objective function is a linear function of n variables. A second standard example is the *asymmetric traveling-salesman problem*, in which we are given n cities, together with an $n \times n$ matrix (c_{ij}), in which entry c_{ij} represents the distance from city i to city j. The feasible solutions are the closed paths which visit each city exactly once, and the cost of such a closed path is the sum of the costs of its links. Thus the problem is to find a shortest tour through the n cities.

The branch-and-bound method generates a tree of interrelated combinatorial optimization problems, all of which have the same objective function but different feasible sets. The root represents the original problem to be solved. In general, a node (problem) A is expanded either by directly solving problem A (in which case A is a terminal node) or by creating new problems $A_1, A_2, \ldots, A_d$ (the children of A) such that the solution set of A is the union of the solution sets of the derived problems $A_1, A_2, \ldots, A_d$. This node expansion step is referred to as *branching.* It follows that, at a general step, after several node expansions have occurred, the optimal value of the original problem is the minimum of the optimal values of the *frontier nodes*, where a frontier node is one that has no children.

Another ingredient of the branch-and-bound method is the use of cost bounds. Whenever a problem is created, a lower bound on its optimal value is calculated. These lower bounds help guide the search for an optimal solution and permit any node to be pruned from the tree if its cost bound is greater than or equal to the cost of a known feasible solution.

Branch-and-bound methods are often the most practical way to solve NP-hard combinatorial optimization problems. As an example, we describe a simple branch-and-bound method for the $0-1$ integer programming problem. An instance of the problem is of the form:

$$\text{minimize } c \cdot x \text{ subject to:}$$

$$Ax \leq b, x \in \{0,1\}^n$$

where the data for the problem are the n-vector c, the m-vector b and the $m \times n$ matrix A. The object is to find a feasible $0-1$-vector $x = (x_1, x_2, \ldots, x_n)$ that minimizes the objective function. A subproblem created after some branching is identical to the original problem except that certain components of the vector x have been assigned fixed values from the set $\{0,1\}$. Thus, the generic form of a subproblem is:

$$\text{minimize } c \cdot x \text{ subject to:}$$

$$Ax \leq b, x \in \{0,1\}^n, x_{i_1} = a_{i_1}, x_{i_2} = a_{i_2}, \ldots, x_{i_t} = a_{i_t}$$

The cost bound associated with a subproblem is obtained by solving its *linear programming relaxation*, in which each variable that is not fixed is allowed to take on any value in the interval $[0,1]$. Thus, the linear programming relaxation of the generic subproblem is the form:

$$\text{minimize } c \cdot x \text{ subject to:}$$

$$Ax \leq b, 0 \leq x_i \leq 1, i = 1, 2, \ldots, n \; x_{i_1} = a_{i_1}, x_{i_2} = a_{i_2}, \ldots, x_{i_t} = a_{i_t}$$

Branching is accomplished by choosing some component x_i that is not fixed and creating two children which are identical to the parent, except that one of them has the additional constraint $x_i = 0$ and the other has the additional constraint $x_i = 1$.

8 Parallel Backtrack Search

I will close by describing some recent research on the parallel execution of node expansion algorithms. In this research the node expansion step is taken to be the unit of work, and it is assumed that, if a node (more precisely, the information needed to describe a node) is resident in the local memory of a processor, then the processor can expand that node in unit time, and the (descriptions of) the children will appear in the local memory of the expanding processor. The goal is to execute the entire node expansion process on a network of processors. In order to keep the processors busy it will be necessary to distribute nodes around the network. We assume that each link in the network can transmit one node per unit time.

Most node expansion algorithms exhibit the phenomenon of *pruning*, in which a node is determined to be irrelevant and thus need not be expanded. In game tree search, a node can be pruned if its value cannot affect the value of the root; pruned nodes correspond to game positions that, with optimal play by both sides, will never be reached. In a branch-and-bound computation, a node can be pruned away if its cost bound exceeds the cost of some known feasible solution. A desirable node expansion algorithm is one that keeps the processors busy expanding nodes, avoids expanding nodes that could have been pruned away, and does not transmit an excessive number of nodes along the links of the network.

Yanjun Zhang's Berkeley Ph.D. dissertation [Z90] presents original results on parallel node expansion algorithms for branch-and-bound computation and game-tree search and gives a survey of the literature on these topics. In this paper we confine ourselves to backtrack search. Node expansion algorithms for backtrack search are particularly simple, since pruning does not occur; the task is simply to generate the entire search tree by performing node expansions, starting from the root. However, the task is complicated by the fact that the pattern of growth of the search tree is completely unpredictable.

The abstract problem, then, is to enumerate by node expansion all the nodes of an initially unknown rooted, oriented, ordered tree; by "oriented" we mean that each edge is directed from a parent to a child, and by "ordered" we mean that a left-to-right ordering is defined on the set of children of any given node. We begin by deriving lower bounds on the time required by any parallel algorithm to generate all the nodes of such a tree by node expansion. Let n be the number of nodes in the search tree, and let h be the maximum number of nodes in a root-leaf path. Let p be the number of processors. Then $\frac{n}{p}$ is a lower bound, since a processor requires one unit of time to expand a node. Also, h is a lower bound, since the nodes along a root-leaf path must be expanded successively. We shall discuss randomized node expansion algorithms which, with high probability, expand all the nodes in time that is within a constant factor of the inherent lower bound $\max(\frac{n}{p}, h)$.

Karp and Zhang have given a parallel node expansion algorithm based on the assumption that any processor can send a node to any other processor in unit time. One interpretation of this assumption is that there is a direct physical link from any processor to any other processor; this is practical only when the number of processors is quite small. A second interpretation is that the physical interconnection structure is sparse, but the unit of time is chosen sufficiently large to permit any message to be routed from its source to its destination.

The Karp-Zhang algorithm is based on randomized load balancing. We shall describe the algorithm in the special case where the tree is binary; i.e., each node has either no children or two children. Central to the algorithm is the concept of a local frontier. At a general step in the algorithm, there will be a set of nodes that have been created but not yet expanded; this set is called the *frontier*. Each node of the frontier be owned by (i.e., resides in the local memory of) exactly one processor, and the set of frontier nodes owned by a particular processor is called its *local frontier*. Each nonempty local frontier F has the property that its nodes hang to the right off a single path. More precisely, there is a path P in the tree such that F consists of the last node in P, together with the right-siblings (if any) of the other nodes in P. In particular, each nonempty local frontier has a unique node whose distance from the root is least, and a unique leftmost node among those whose distance from the root is greatest; these will be called, respectively, the highest and lowest nodes in the local frontier.

Let us call a processor *idle* if its local frontier is empty, *lightly loaded* if its local frontier consists of a single node, and *heavily loaded* if its local frontier contains more than one node. The Karp-Zhang algorithm alternates between *node expansion steps*, in which each processor that is not idle expands the lowest node of its local frontier, and *donation steps*, in which heavily loaded processors get paired up with idle processors and, in each pair, the heavily loaded processor sends the idle processor the highest node of its local frontier. Ideally, the algorithm should find a perfect pairing; i.e., one in which the number of pairs is exactly equal to the number of heavily loaded processors or the number of idle processors, whichever is smaller. However, the construction of such a pairing requires global knowledge of the status of the processors, and thus is not suitable for a distributed implementation. Instead, the Karp-Zhang algorithm operates as follows. First, each idle processor proposes to a random processor. Then, each heavily loaded processor that has received a proposal pairs itself with an arbitrary one of the idle processors that have proposed to it. Even though this algorithm does not achieve a perfect pairing, it tends to achieve a pairing that is not too far from maximum whenever the number of idle processors is large. It is shown in [KaZh] that, with high probability, the execution time of this algorithm is within a constant factor of the inherent lower bound $\max(\frac{n}{p}, h)$.

The assumption that any processor can communicate with any other in unit time is unrealistic. Recently, Ranade has given a more sophisticated parallel backtrack search algorithm which runs on the Butterfly network and, with high probability, has an execution time within a small constant factor of the inherent lower bound.

References

[EG88] D. Eppstein and Z. Galil, *Parallel Algorithmic Techniques for Combinatorial Computation,* Columbia University Department of Computer Science Technical Report, New York, 1988.

[KR90] R.M. Karp and V. Ramachandran, "Parallel Algorithms for Shared-memory Machines", in Chapter 17, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity,* ed., J. van Leeuwen, Elsevier, Amsterdam and MIT Press, Cambridge, 869-941, 1990.

[R87] A.G. Ranade, "How to Emulate Shared Memory", in: *Proc. 28th Annual IEEE Symp. Foundations of Computer Science* 185-194, 1987.

[RV87] J.H. Reif and L.G. Valiant, "A Logarithmic Time Sort for Linear Size Networks", *J. ACM* Vol. 34, 60-76, 1987.

[V90] L.G. Valiant, "General Purpose Parallel Architectures", in Chapter 18, *Handbook of Theoretical Computer Science, Vol. A: Algorithms* *and Complexity,* ed., J. van Leeuwen, Elsevier, Amsterdam and MIT Press, Cambridge, 1990, 943-971.

[Z90] Y. Zhang, *Parallel Algorithms for Combinatorial Search Problems,* Ph.D. Thesis. Also, UC Berkeley Computer Science Division, Report No. UCB/CSD 543, 1990.