Parallel Algorithms

Guy E. Blelloch and Bruce M. Maggs
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
guyb@cs.cmu.edu, bmm@cs.cmu.edu

Introduction

The subject of this chapter is the design and analysis of parallel algorithms. Most of today's algorithms are sequential, that is, they specify a sequence of steps in which each step consists of a single operation. These algorithms are well suited to today's computers, which basically perform operations in a sequential fashion. Although the speed at which sequential computers operate has been improving at an exponential rate for many years, the improvement is now coming at greater and greater cost. As a consequence, researchers have sought more cost-effective improvements by building "parallel" computers – computers that perform multiple operations in a single step. In order to solve a problem efficiently on a parallel machine, it is usually necessary to design an algorithm that specifies multiple operations on each step, i.e., a parallel algorithm.

As an example, consider the problem of computing the sum of a sequence A of n numbers. The standard algorithm computes the sum by making a single pass through the sequence, keeping a running sum of the numbers seen so far. It is not difficult however, to devise an algorithm for computing the sum that performs many operations in parallel. For example, suppose that, in parallel, each element of A with an even index is paired and summed with the next element of A, which has an odd index, i.e., $A[0]$ is paired with $A[1]$, $A[2]$ with $A[3]$, and so on. The result is a new sequence of $\lceil n/2 \rceil$ numbers that sum to the same value as the sum that we wish to compute. This pairing and summing step can be repeated until, after $\lceil \log_2 n \rceil$ steps, a sequence consisting of a single value is produced, and this value is equal to the final sum.

The parallelism in an algorithm can yield improved performance on many different kinds of computers. For example, on a parallel computer, the operations in a parallel algorithm can be performed simultaneously by different processors. Furthermore, even on a single-processor computer the parallelism in an algorithm can be exploited by using multiple functional units, pipelined functional units, or pipelined memory systems. Thus, it is important to make a distinction between the parallelism in an algorithm and the ability of any particular computer to perform multiple operations in parallel. Of course, in order for a parallel algorithm to run efficiently on any type of computer, the algorithm must contain at least as much parallelism as the computer, for otherwise resources would be left idle. Unfortunately, the converse does not always hold: some parallel computers cannot efficiently execute all algorithms, even if the algorithms contain a great deal of parallelism. Experience has shown that it is more difficult to build a general-purpose parallel machine than a general-purpose sequential machine.

The remainder of this chapter consists of nine sections. We begin in Section 1 with a discussion of how to model parallel computers. Next, in Section 2 we cover some general techniques that have proven useful in the design of parallel algorithms. Sections 3 through 7 present algorithms for solving problems from different domains. We conclude in Sections 8 through 10 with a discussion of current research topics, a collection of defining terms, and finally sources for further information.

Throughout this chapter, we assume that the reader has some familiarity with sequential algorithms and asymptotic analysis.

1 Modeling parallel computations

The designer of a sequential algorithm typically formulates the algorithm using an abstract model of computation called the *random-access machine* (RAM) [2, Chapter 1] model. In this model, the machine consists of a single processor connected to a memory system. Each basic CPU operation, including arithmetic operations, logical operations, and memory accesses, requires one time step. The designer's goal is to develop an algorithm with modest time and memory requirements. The random-access machine model allows the algorithm designer to ignore many of the details of the computer on which the algorithm will ultimately be executed, but captures enough detail that the designer can predict with reasonable accuracy how the algorithm will perform.

Modeling parallel computations is more complicated than modeling sequential computations because in practice parallel computers tend to vary more in organization than do sequential computers. As a consequence, a large portion of the research on parallel algorithms has gone into the question of modeling, and many debates have raged over what the "right" model is, or about how practical various models are. Although there has been no consensus on the right model, this research has yielded a better understanding of the relationship between the models. Any discussion of parallel algorithms requires some understanding of the various models and the relationship among them.

In this chapter we divide parallel models into two classes: multiprocessor models and work-depth models. In the remainder of this section we discuss these two classes and how they are related.

1.1 Multiprocessor models

A *multiprocessor model* is a generalization of the sequential RAM model in which there is more than one processor. Multiprocessor models can be classified into three basic types: local memory machine models, modular memory machine models, and parallel random-access machine (PRAM) models. Figure 1 illustrates the structure of these machine models. A local memory machine model consists of a set of n processors each with its own local memory. These processors are attached to a common communication network. A modular memory machine model consists of m memory modules and n processors all attached to a common network. An n-processor PRAM model consists of a set of n processors all connected to a common shared memory [32, 37, 38, 77].

The three types of multiprocessors differ in the way that memory can be accessed. In a local memory machine model, each processor can access its own local memory directly, but can access the memory in another processor only by sending a memory request through the network. As in the RAM model, all local operations, including local memory accesses, take unit time. The time taken to access the memory in another processor, however, will depend on both the capabilities of the communication network and the pattern of memory accesses made by other processors, since these other accesses could congest the network. In a modular memory machine model, a processor accesses the memory in a memory module by sending a memory request through the network. Typically the processors and memory modules are arranged so that the time for any processor to access any memory module is roughly uniform. As in a local memory machine model, the exact amount of time depends on the communication network and the memory access pattern. In a PRAM model, a processor can access any word of memory in a single step. Furthermore, these accesses can occur in parallel, i.e., in a single step, every processor can access the shared memory.

The PRAM models are controversial because no real machine lives up to its ideal of unit-time access to shared memory. It is worth noting, however, that the ultimate purpose of an abstract model is not to directly model a real machine, but to help the algorithm designer produce efficient algorithms. Thus, if an algorithm designed for a PRAM model (or any other model) can be translated to an algorithm that runs efficiently on a real computer, then the model has succeeded. In Section 1.4 we show how an algorithm designed for one parallel machine model can be translated so that it executes efficiently on another model.

The three types of multiprocessor models that we have defined are broad and allow for many variations. The local memory machine models and modular memory machine models may differ

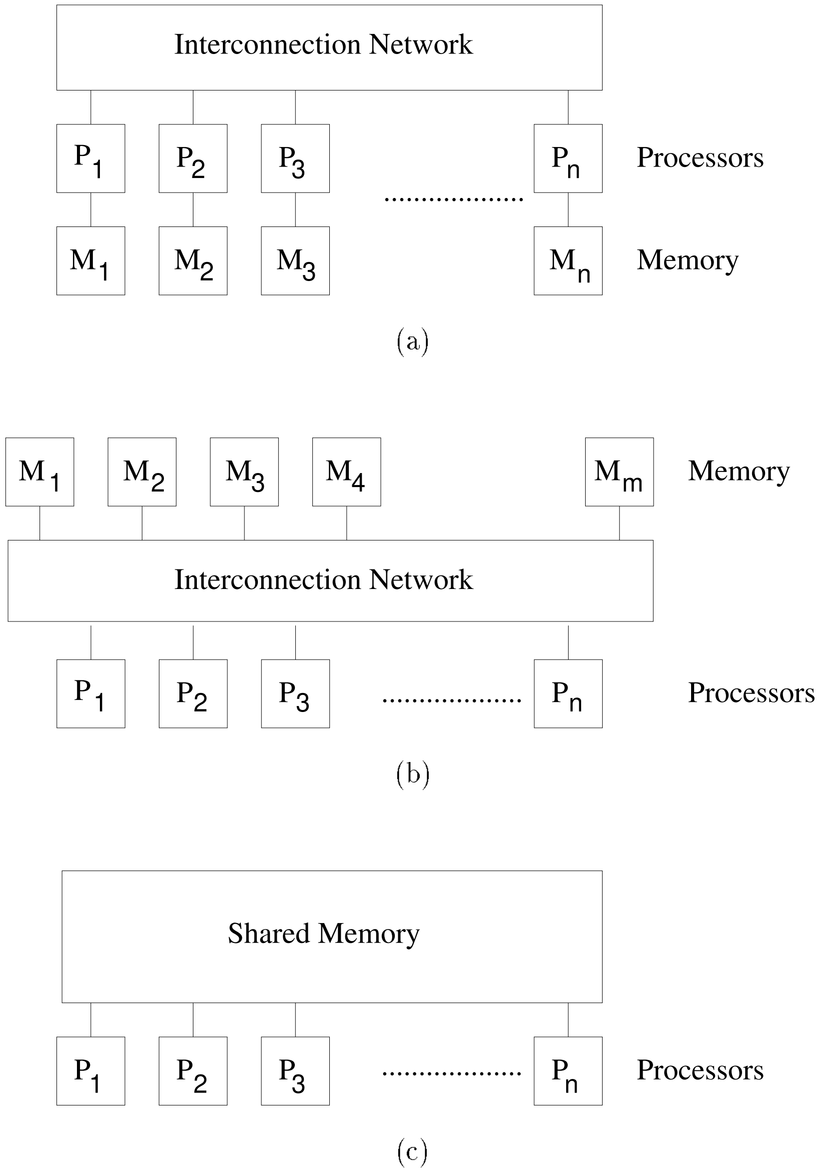


Figure 1: The three types of multiprocessor machine models. (a) A local memory machine model. (b) A modular memory machine model. (c) A parallel random-access machine (PRAM) model.

according to their network topologies. Furthermore, in all three types of models, there may be differences in the operations that the processors and networks are allowed to perform. In the remainder of this section we discuss some of the possibilities.

1.1.1 Network topology

A network is a collection of switches connected by communication channels. A processor or memory module has one or more communication ports that are connected to these switches by communication channels. The pattern of interconnection of the switches is called the network topology. The topology of a network has a large influence on the performance and also on the cost and difficulty of constructing the network. Figure 2 illustrates several different topologies.

The simplest network topology is a bus. This network can be used in both local memory machine models and modular memory machine models. In either case, all processors and memory modules are typically connected to a single bus. In each step, at most one piece of data can be written onto the bus. This data might be a request from a processor to read or write a memory value, or it might be the response from the processor or memory module that holds the value. In practice, the advantages of using a bus is that it is simple to build, and, because all processors and memory modules can observe the traffic on the bus, it is relatively easy to develop protocols that allow processors to cache memory values locally. The disadvantage of using a bus is that the processors have to take turns accessing the bus. Hence, as more processors are added to a bus, the average time to perform a memory access grows proportionately.

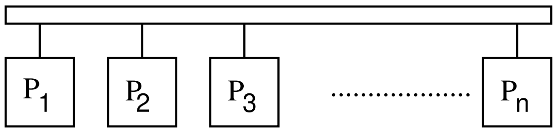
A 2-dimensional *mesh* is a network that can be laid out in a rectangular fashion. Each switch in a mesh has a distinct label (x, y) where $0 \leq x \leq X-1$ and $0 \leq y \leq Y-1$. The values X and Y determine the length of the sides of the mesh. The number of switches in a mesh is thus $X \cdot Y$. Every switch, except those on the sides of the mesh, is connected to four neighbors: one to the north, one to the south, one to the east, and one to the west. Thus, a switch labeled (x, y), where $0 < x < X-1$ and $0 < y < Y-1$ is connected to switches $(x, y+1)$, $(x, y-1)$, $(x+1, y)$, and $(x-1, y)$. This network typically appears in a local memory machine model, i.e., a processor along with its local memory is connected to each switch, and remote memory accesses are made by routing messages through the mesh. Figure 2(b) shows an example of an 8×8 mesh.

Several variations on meshes are also popular, including 3-dimensional meshes, toruses, and hypercubes. A *torus* is a mesh in which the switches on the sides have connections to the switches on the opposite sides. Thus, every switch (x, y) is connected to four other switches: $(x, y+1 \bmod Y)$, $(x, y-1 \bmod Y)$, $(x+1 \bmod X, y)$, and $(x-1 \bmod X, y)$. A hypercube is a network with 2^n switches in which each switch has a distinct n-bit label. Two switches are connected by a communication channel in a hypercube if and only if the labels of the switches differ in precisely one bit position. A hypercube with 16 switches is shown in Figure 2(c).

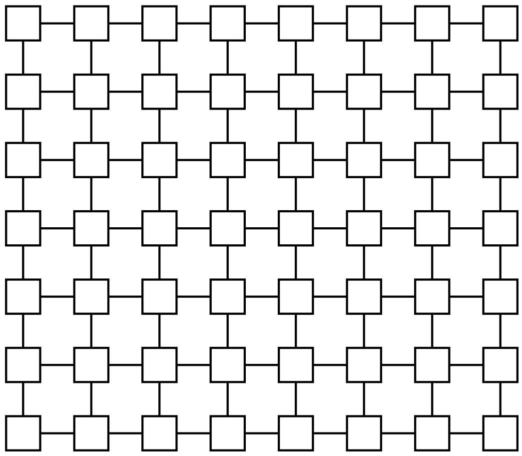
A *multistage network* is used to connect one set of switches called the *input switches* to another set called the *output switches* through a sequence of stages of switches. Such networks were originally designed for telephone networks [15]. The stages of a multistage network are numbered 1 through L, where L is the *depth* of the network. The switches on stage 1 are the input switches, and those on stage L are the output switches. In most multistage networks, it is possible to send a message from any input switch to any output switch along a path that traverses the stages of the network in order from 1 to L. Multistage networks are frequently used in modular memory computers; typically processors are attached to input switches, and memory modules to output switches. A processor accesses a word of memory by injecting a memory access request message into the network. This message then travels through the network to the appropriate memory module. If the request is to read a word of memory, then the memory module sends the data back through then network to the requesting processor. There are many different multistage network topologies. Figure 1.1.1(a), for example, shows a 2-stage network that connects 4 processors to 16 memory modules. Each switch in this network has two channels at the bottom and four channels at the top. The ratio of processors to memory modules in this example is chosen to reflect the fact that, in practice, a processor is capable of generating memory access requests faster than a memory module is capable of servicing them.

A *fat-tree* is a network structured like a tree [56]. Each edge of the tree, however, may represent many communication channels, and each node may represent many network switches (hence the name "fat"). Figure 1.1.1(b) shows a fat-tree with the overall structure of a binary tree. Typically the capacities of the edges near the root of the tree are much larger than the capacities near the leaves. For example, in this tree the two edges incident on the root represent 8 channels each, while the edges incident on the leaves represent only 1 channel each. A natural way to construct a local memory machine model is to connect a processor along with its local memory to each leaf of the fat-tree. In this scheme, a message from one processor to another first travels up the tree to the least common-ancestor of the two processors, and then down the tree.

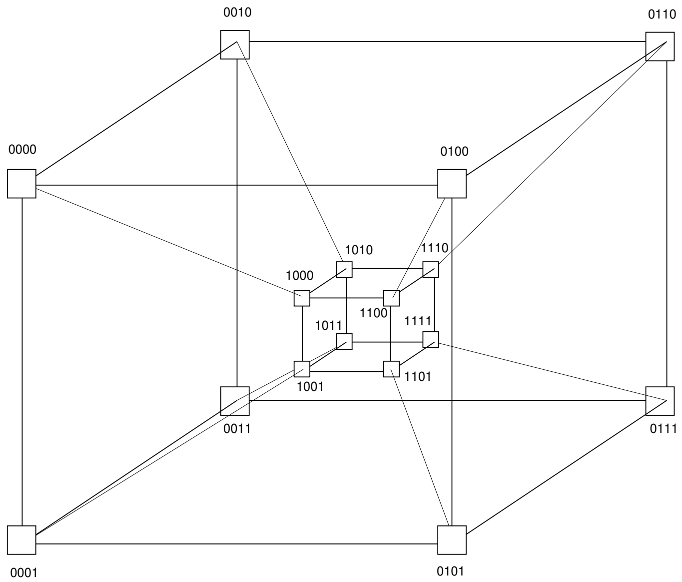
Many algorithms have been designed to run efficiently on particular network topologies such as the mesh or the hypercube. For an extensive treatment such algorithms, see [55, 67, 73, 80]. Although this approach can lead to very fine-tuned algorithms, it has some disadvantages. First, algorithms designed for one network may not perform well on other networks. Hence, in order to solve a problem on a new machine, it may be necessary to design a new algorithm from scratch. Second, algorithms that take advantage of a particular network tend to be more complicated than



(a) Bus

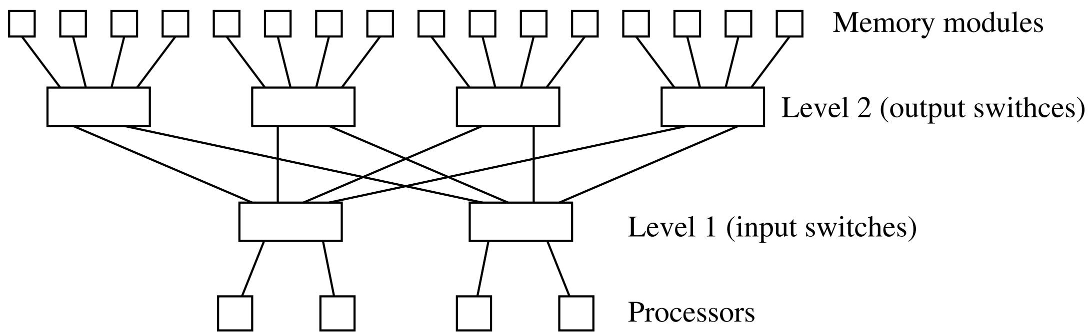


(b) 2-dimensional Mesh

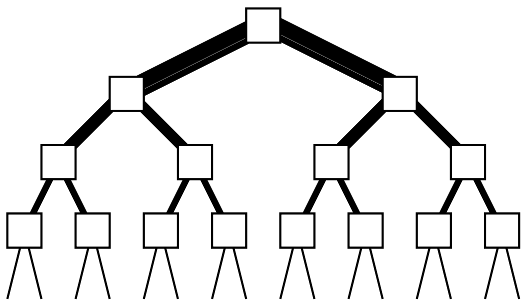


(c) Hypercube

Figure 2: Bus, mesh, and hypercube network topologies.



(a) 2-level multistage network



(b) Fat-tree

Figure 3: Multistage and Fat-tree network topologies.

algorithms designed for more abstract models like the PRAM models because they must incorporate some of the details of the network. Nevertheless, there are some operations that are performed so frequently by a parallel machine that it makes sense to design a fine-tuned network-specific algorithm. For example, the algorithm that routes messages or memory access requests through the network should exploit the network topology. Other examples include algorithms for broadcasting a message from one processor to many other processors, for collecting the results computed in many processors in a single processor, and for synchronizing processors.

An alternative to modeling the topology of a network is to summarize its routing capabilities in terms of two parameters, its latency and bandwidth. The latency, L, of a network is the time it takes for a message to traverse the network. In actual networks this will depend on the topology of the network, which particular ports the message is passing between, and the congestion of messages in the network. The latency, is often modeled by considering the worst-case time assuming that the network is not heavily congested. The bandwidth at each port of the network is the rate at which a processor can inject data into the network. In actual networks this will depend on the topology of the network, the bandwidths of the network's individual communication channels, and, again, the congestion of messages in the network. The bandwidth often can be usefully modeled as the maximum rate at which processors can inject messages into the network without causing it to become heavily congested, assuming a uniform distribution of message destinations. In this case, the bandwidth can be expressed as the minimum *gap* g between successive injections of messages into the network.

Three models that characterize a network in terms of its latency and bandwidth are the Postal model [14], the Bulk-Synchronous Parallel (BSP) model [85], and the LogP model [29]. In the Postal model, a network is described by a single parameter L, its latency. The Bulk-Synchronous Parallel model adds a second parameter g, the minimum ratio of computation steps to communication steps, i.e., the gap. The LogP model includes both of these parameters, and adds a third parameter o, the overhead, or wasted time, incurred by a processor upon sending or receiving a message.

1.1.2 Primitive operations

A machine model must also specify the types of operations that the processors and network are permitted to perform. We assume that all processors are allowed to perform the same local instructions as the single processor in the standard sequential RAM model. In addition, processors may have special instructions for issuing non-local memory requests, for sending messages to other processors, and for executing various global operations, such as synchronization. There may also be restrictions on when processors can simultaneously issue instructions involving non-local operations. For example a model might not allow two processors to write to the same memory location at the same time. These restrictions might make it impossible to execute an algorithm on a particular model, or make the cost of executing the algorithm prohibitively expensive. It is therefore important to understand what instructions are supported before one can design or analyze a parallel algorithm. In this section we consider three classes of instructions that perform non-local operations: (1) instructions that perform concurrent accesses to the same shared memory location, (2) instructions for synchronization, and (3) instructions that perform global operations on data.

When multiple processors simultaneously make a request to read or write to the same resource—such as a processor, memory module, or memory location—there are several possible outcomes. Some machine models simply forbid such operations, declaring that it is an error if more that one processor tries to access a resource simultaneously. In this case we say that the model allows only *exclusive* access to the resource. For example, a PRAM model might only allow exclusive read or write access to each memory location. A PRAM model of this type is called an exclusive-read exclusive-write (EREW) PRAM model. Other machine models may allow unlimited access to a shared resource. In this case we say that the model allows *concurrent* access to the resource. For example, a concurrent-read concurrent-write (CRCW) PRAM model allows both concurrent read and write access to memory locations, and a CREW PRAM model allows concurrent reads but only exclusive writes. When making a concurrent write to a resource such as a memory location there are many ways to resolve the conflict. The possibilities include choosing an arbitrary value from those written (arbitrary concurrent write), choosing the value from the processor with lowest index (priority concurrent write), and taking the *logical or* of the values written. A final choice is to allow for *queued* access. In this case concurrent access is permitted but the time for a step is proportional to the maximum number of accesses to any resource. A queue-read queue-write (QRQW) PRAM model allows for such accesses [36].

In addition to reads and writes to non-local memory or other processors, there are other important primitives that a model might supply. One class of such primitives support synchronization. There are a variety of different types of synchronization operations and the costs of these operations vary from model to model. In a PRAM model, for example, it is assumed that all processors operate in lock step, which provides implicit synchronization. In a local-memory machine model the cost of synchronization may be a function of the particular network topology. A related operation, broadcast, allows one processor to send a common message to all of the other processors. Some machine models supply more powerful primitives that combine arithmetic operations with communication. Such operations include the prefix and multiprefix operations, which are defined in Sections 3.2, and 3.3.

1.2 Work-depth models

Because there are so many different ways to organize parallel computers, and hence to model them, it is difficult to select one multiprocessor model that is appropriate for all machines. The alternative to focusing on the machine is to focus on the algorithm. In this section we present a class of models called work-depth models. In a *work-depth* model, the cost of an algorithm is determined by examining the total number of operations that it performs, and the dependencies among those operations. An algorithm's *work* W is the total number of operations that it performs; its *depth* D is the longest chain of dependencies among its operations. We call the ratio $\mathcal{P} = W/D$ the *parallelism* of the algorithm.

The work-depth models are more abstract than the multiprocessor models. As we shall see however, algorithms that are efficient in work-depth models can often be translated to algorithms that are efficient in the multiprocessor models, and from there to real parallel computers. The advantage of using a work-depth model is that there are no machine-dependent details to complicate the design and analysis of algorithms. Here we consider three classes of work-depth models: circuit models, vector machine models, and language-based models. We will be using a language-based model in this chapter, so we will return to these models in Section 1.5. The most abstract work-depth model is the *circuit model*. A circuit consists of nodes and directed arcs. A node represents a basic operation, such as adding two values. Each input value for an operation arrives at the corresponding node via an incoming arc. The result of the operation is then carried out of the node via one or more outgoing arcs. These outgoing arcs may provide inputs to other nodes. The number of incoming arcs to a node is referred to as the *fan-in* of the node and the number of outgoing arcs is referred to as the *fan-out*. There are two special classes of arcs. A set of *input arcs* provide input values to the circuit as a whole. These arcs do not originate at nodes. The *output arcs* return the final output values produced by the circuit. These arcs do not terminate at nodes. By definition, a circuit is not permitted to contain a directed cycle. In this model, an algorithm is modeled as a family of directed acyclic circuits. There is a circuit for each possible size of the input.

Figure 4 shows a circuit for adding 16 numbers. In this figure all arcs are directed towards the bottom. The input arcs are at the top of the figure. Each $+$ node adds the two values that arrive on its two incoming arcs, and places the result on its outgoing arc. The sum of all of the inputs to the circuit is returned on the single output arc at the bottom.

The work and depth of a circuit are measured as follows. The work is the total number of nodes. The work in Figure 4, for example, is 15. (The work is also called the *size* of the circuit.) The depth is the number of nodes on the longest directed path from an input arc and an output arc. In

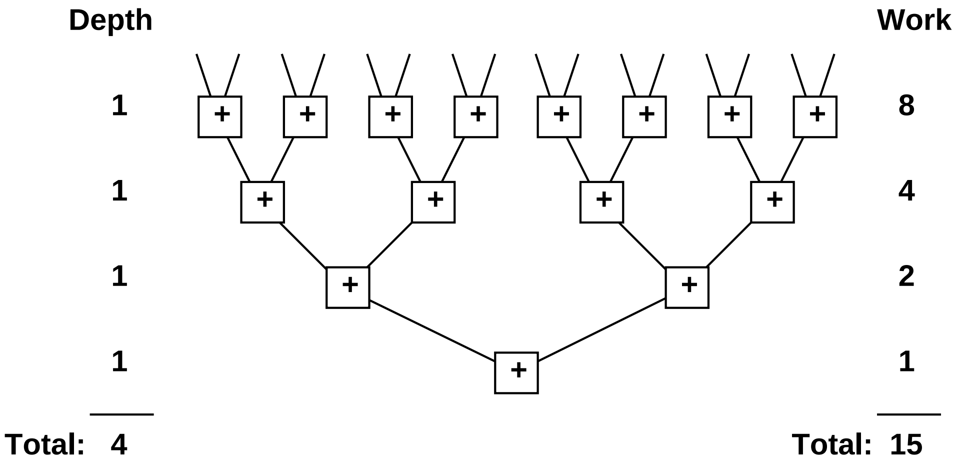


Figure 4: Summing 16 numbers on a tree. The total depth (longest chain of dependencies) is 4 and the total work (number of operations) is 15.

Figure 4, the depth is 4. For a family of circuits, the work and depth are typically parameterized in terms of the number of inputs. For example, the circuit in Figure 4 can be easily generalized to add n input values for any n that is a power of two. The work and depth for this family of circuits is $W(n) = n - 1$ and $D(n) = \log_2 n$.

Circuit models have been used for many years to study various theoretical aspects of parallelism, for example to prove that certain problems are difficult to solve in parallel. See [48] for an overview.

In a *vector model* an algorithm is expressed as a sequence of steps, each of which performs an operation on a vector (i.e., sequence) of input values, and produces a vector result [19, 69]. The work of each step is equal to the length of its input (or output) vector. The work of an algorithm is the sum of the work of its steps. The depth of an algorithm is the number of vector steps.

In a *language* model, a work-depth cost is associated with each programming language construct [20, 22]. For example, the work for calling two functions in parallel is equal to the sum of the work of the two calls. The depth, in this case, is equal to the maximum of the depth of the two calls.

1.3 Assigning costs to algorithms

In the work-depth models, the cost of an algorithm is determined by its work and by its depth. The notions of work and depth can also be defined for the multiprocessor models. The work W performed by an algorithm is equal to the number of processors multiplied by the time required for the algorithm to complete execution. The depth D is equal to the total time required to execute the algorithm.

The depth of an algorithm is important because there are some applications for which the time to perform a computation is crucial. For example, the results of a weather forecasting program are useful only if the program completes execution before the weather does!

Generally, however, the most important measure of the cost of an algorithm is the work. This can be argued as follows. The cost of a computer is roughly proportional to the number of processors in the computer. The cost for purchasing time on a computer is proportional to the cost of the computer multiplied by the amount of time used. The total cost of performing a computation, therefore, is roughly proportional to the number of processors in the computer multiplied by the amount of time, i.e., the work.

In many instances, the work (cost) required by a computation on a parallel computer may be slightly greater the the work required by the same computation on a sequential computer. If the time to completion is sufficiently improved, however, this extra work can often be justified. As we shall see, however, there is often a tradeoff between time-to-completion and total work performed. To quantify when parallel algorithms are efficient in terms of work, we say that a parallel algorithm is *work-efficient* if asymptotically (as the problem size grows) it requires at most a constant factor more work than the best sequential algorithm known.

1.4 Emulations among models

Although it may appear that a different algorithm must be designed for each of the many parallel models, there are often automatic and efficient techniques for translating algorithms designed for one model into algorithms designed for another. These translations are *work-preserving* in the sense that the work performed by both algorithms is the same, to within a constant factor. For example, the following theorem, known as Brent's Theorem [24], shows that an algorithm designed for the circuit model can be translated in a work-preserving fashion to a PRAM model algorithm.

Theorem 1.1 (Brent's Theorem) *Any algorithm that can be expressed as a circuit of size (i.e., work) W, depth D and with constant fan-in nodes in the circuit model can be executed in $O(W/P + D)$ steps in the CREW PRAM model.*

Proof: The basic idea is to have the PRAM emulate the computation specified by the circuit in a level-by-level fashion. The level of a node is defined as follows. A node is on level 1 if all of its inputs are also inputs to the circuit. Inductively, the level of any other node is one greater than the maximum of the level of the nodes that provide its inputs. Let l_i denote the number of nodes on level i. Then, by assigning $\lceil l_i/P \rceil$ operations to each of the P processors in the PRAM, the operations for level i can be performed in $O(\lceil l_i/P \rceil)$ steps. Concurrent reads might be required since many operations on one level might read the same result from a previous level. Summing the time over all D levels, we have

$$\begin{aligned} T_{PRAM}(W,D,P) &= O\left(\sum_{i=1}^{D}\left\lceil\frac{l_i}{P}\right\rceil\right) \\ &= O\left(\sum_{i=1}^{D}\left(\frac{l_i}{P}+1\right)\right) \\ &= O\left(\frac{1}{P}\left(\sum_{i=1}^{D} l_i\right)+D\right) \\ &= O\left(W/P+D\right). \end{aligned}$$

The last step is derived by observing that $W = \sum_{i=1}^{D} l_i$, i.e., that the work is equal to the total number of nodes on all of the levels of the circuit. □

The total work performed by the PRAM, i.e., the processor-time product, is $O(W+PD)$. This emulation is work-preserving to within a constant factor when the parallelism ($\mathcal{P} = W/D$) is at least as large as the number of processors P, for in this case the work is $O(W)$. The requirement that the parallelism exceed the number of processors is typical of work-preserving emulations.

Brent's theorem shows that an algorithm designed for one of the work-depth models can be translated in a work-preserving fashion to a multiprocessor model. Another important class of work-preserving translations are those that translate between different multiprocessor models. The translation we consider here is the work-preserving translation of algorithms written for the PRAM model to algorithms for a modular memory machine model that incorporates the feature of network topology. In particular we consider a *butterfly machine* model in which P processors are attached through a butterfly network of depth $\log P$ to P memory banks. We assume that, in constant time, a processor can hash a virtual memory address to a physical memory bank and an address within that bank using a sufficiently powerful hash function. This scheme was first proposed by Karlin and Upfal [47] for the EREW PRAM model. Ranade [72] later presented a more general approach that allowed the butterfly to efficiently emulate CRCW algorithms.

Theorem 1.2 *Any algorithm that takes time T on a P-processor PRAM model can be translated into an algorithm that takes time $O(T(P/P' + \log P'))$, with high probability, on a P'-processor butterfly machine model.*

Sketch of proof: Each of the P' processors in the butterfly emulates a set of P/P' PRAM processors. The butterfly emulates the PRAM in a step-by-step fashion. First, each butterfly processor emulates one step of each of its P/P' PRAM processors. Some of the PRAM processors may wish to perform memory accesses. For each memory access, the butterfly processor hashes the memory address to a physical memory bank and an address within the bank, and then routes a message through the network to that bank. These messages are pipelined so that a butterfly processor can have multiple outstanding requests. Ranade proved that if each processor in the P-processor butterfly sends at most P/P' messages, and if the destinations of the messages are determined by a sufficiently powerful random hash function, then the network can deliver all of the messages, along with responses, in $O(P/P' + \log P')$ time. The $\log P'$ term accounts for the latency of the network, and for the fact that there will be some congestion at memory banks, even if each processor sends only a single message. □

This theorem implies that the emulation is work preserving when $P \geq P' \log P'$, i.e., when the number of processors employed by the PRAM algorithm exceeds the number of processors in the butterfly by a factor of at least $\log P'$. When translating algorithms from one multiprocessor model (e.g., the PRAM model), which we call the *guest model*, to another multiprocessor model (e.g., the butterfly machine model), which we call the *host model*, it is not uncommon to require that the number of guest processors exceed the number of host processors by a factor proportional to the latency of the host. Indeed, the latency of the host can often be hidden by giving it a larger guest to emulate. If the bandwidth of the host is smaller than the bandwidth of a comparably sized guest, however, it is usually much more difficult for the host to perform a work-preserving emulation of the guest.

For more information on PRAM emulations, the reader is referred to [43, 86]

1.5 Model used in this chapter

Because there are so many work-preserving translations between different parallel models of computation, we have the luxury of choosing the model that we feel most clearly illustrates the basic ideas behind the algorithms, a work-depth language model. Here we define the model that we use in this chapter in terms of a set of language constructs and a set of rules for assigning costs to the constructs. The description here is somewhat informal, but should suffice for the purpose of this chapter. The language and costs can be properly formalized using a profiling semantics [22].

Most of the syntax that we use should be familiar to readers who have programmed in Algol-like languages, such as Pascal and C. The constructs for expressing parallelism, however, may be unfamiliar. We will be using two parallel constructs—a parallel *apply-to-each* construct and a *parallel-do* construct—and a small set of parallel primitives on sequences (one dimensional arrays). Our language constructs, syntax and cost rules are loosely based on the NESL language [20].

The *apply-to-each* construct is used to apply an expression over a sequence of values in parallel. It uses a set like notation. For example, the expression

$$\{a * a : a \in [3, -4, -9, 5]\}$$

squares each element of the sequence $[3, -4, -9, 5]$ returning the sequence $[9, 16, 81, 25]$. This can be read: "in parallel, for each a in the sequence $[3, -4, -9, 5]$, square a". The apply-to-each construct also provides the ability to subselect elements of a sequence based on a filter. For example

$$\{a * a \ : \ a \in [3, -4, -9, 5] \mid a > 0\}$$

can be read: "in parallel, for each a in the sequence $[3, -4, -9, 5]$ such that a is greater than 0, square a". It returns the sequence $[9, 25]$. The elements that remain maintain the relative order from the original sequence.

The *parallel-do* construct is used to evaluate multiple statements in parallel. It is expressed by listing the set of statements after the keywords **in parallel do**. For example, the following fragment of code calls FUNCTION1 on X and assigns the result to A and in parallel calls FUNCTION2 on Y and assigns the result to B.

in parallel do
  $A := \textsc{function1}(X)$
  $B := \textsc{function2}(Y)$

The parallel-do completes when all the parallel subcalls complete.

Work and depth are assigned to our language constructs as follows. The work and depth of a scalar primitive operation is one. For example, the work and depth for evaluating an expression such as $3 + 4$ is one. The work for applying a function to every element in a sequence is equal to the sum of the work for each of the individual applications of the function. For example, the work for evaluating the expression

$$\{a * a \ : \ a \in [0..n)\},$$

which creates an n-element sequence consisting of the squares of 0 through $n-1$, is n. The depth for applying a function to every element in a sequence is equal to the maximum of the depths of the individual applications of the function. Hence, the depth of the previous example is one. The work for a parallel-do construct is equal to the sum of the work for each of its statements. The depth is equal to the maximum depth of its statements. In all other cases, the work and depth for a sequence of operations is the sum of the work and depth for the individual operations.

In addition to the parallelism supplied by apply-to-each, we use four built-in functions on sequences, *distribute*, ++ (append), *flatten*, and $\leftarrow$ (write), each of which can be implemented in parallel. The function *distribute* creates a sequence of identical elements. For example, the expression

$$distribute(3, 5)$$

creates the sequence

$$[3, 3, 3, 3, 3].$$

The ++ function appends two sequences. For example $[2, 1]$++$[5, 0, 3]$ create the sequence $[2, 1, 5, 0, 3]$. The *flatten* function converts a nested sequence (a sequence for which each element is itself a sequence) into a flat sequence. For example,

$$\mathit{flatten}([[3, 5], [3, 2], [1, 5], [4, 6]])$$

creates the sequence

$$[3, 5, 3, 2, 1, 5, 4, 6].$$

The $\leftarrow$ function is used to write multiple elements into a sequence in parallel. It takes two arguments. The first argument is the sequence to modify and the second is a sequence of integer-value pairs that specify what to modify. For each pair (i, v) the value v is inserted into position i of the destination sequence. For example

$$[0, 0, 0, 0, 0, 0, 0, 0] \leftarrow [(4, -2), (2, 5), (5, 9)]$$

inserts the -2, 5 and 9 into the sequence at locations 4, 2 and 5, respectively, returning

$$[0, 0, 5, 0, -2, 9, 0, 0].$$

As a the PRAM model, the issue of concurrent writes arises if an index is repeated. Rather than choosing a single policy for resolving concurrent writes, we will explain the policy used for the individual algorithms. All of these functions have depth one and work n, where n is the size of the sequence(s) involved. In the case of $\leftarrow$, the work is proportional to the length of the sequence of integer-value pairs, not the modified sequence, which might be much longer. In the case of ++, the work is proportional to the length of the second sequence.

We will use a few shorthand notations for specifying sequences. The expression $[-2..1]$ specifies the same sequence as the expression $[-2, -1, 0, 1]$. Changing the left or right bracket surrounding a sequence to a parenthesis omits the first or last elements, i.e., $[-2..1)$ denotes the sequence $[-2, -1, 0]$. The notation $A[i..j]$ denotes the subsequence consisting of elements $A[i]$ through $A[j]$. Similarly, $A[i, j)$ denotes the subsequence $A[i]$ through $A[j - 1]$. We will assume that sequence indices are zero based, i.e., $A[0]$ extracts the first element of the sequence A.

Throughout this chapter our algorithms make use of random numbers. These numbers are generated using the functions *rand_bit()*, which returns a random bit, and *rand_int*(h), which returns a random integer in the range $[0, h-1]$.

2 Parallel algorithmic techniques

As in sequential algorithm design, in parallel algorithm design there are many general techniques that can be used across a variety of problem areas. Some of these are variants of standard sequential techniques, while others are new to parallel algorithms. In this section we introduce some of these techniques, including parallel divide-and-conquer, randomization, and parallel pointer manipulation. We will make use of these techniques in later sections.

2.1 Divide-and-conquer

A divide-and-conquer algorithm splits the problem to be solved into subproblems that are easier to solve than the original problem, solves the subproblems, and merges the solutions to the subproblems to construct a solution to the original problem.

The divide-and-conquer paradigm improves program modularity, and often leads to simple and efficient algorithms. It has therefore proven to be a powerful tool for sequential algorithm designers. Divide-and-conquer plays an even more prominent role in parallel algorithm design. Because the subproblems created in the first step are typically independent, they can be solved in parallel. Often the subproblems are solved recursively and thus the next divide step yields even more subproblems to be solved in parallel. As a consequence, even divide-and-conquer algorithms that were designed for sequential machines typically have some inherent parallelism. Note however, that in order for divide-and-conquer to yield a highly parallel algorithm, it is often necessary to parallelize the divide step and the merge step. It is also common in parallel algorithms to divide the original problem into as many subproblems as possible, so that they can all be solved in parallel.

As an example of parallel divide-and-conquer, consider the sequential mergesort algorithm. Mergesort takes a sequence of n keys as input and returns the keys in sorted order. It works by splitting the keys into two sequences of $n/2$ keys, recursively sorting each sequence, and then merging the two sorted sequences of $n/2$ keys into a sorted sequence of n keys. To analyze the sequential running time of mergesort we note that two sorted sequences of $n/2$ keys can be merged in $O(n)$ time. Hence the running time can be specified by the recurrence

$$T(n) = \begin{cases} 2T(n/2) + O(n) & n > 1 \\ O(1) & n = 1 \end{cases} \tag{1}$$

which has the solution $T(n) = O(n \log n)$. Although not designed as a parallel algorithm, mergesort has some inherent parallelism since the two recursive calls are independent, thus allowing them to be made in parallel. The parallel calls can be expressed as:

ALGORITHM: MERGESORT(A)

if $(|A| = 1)$ **then return** A
else
    in parallel do
        $L :=$ MERGESORT$(A[0..|A|/2))$
        $R :=$ MERGESORT$(A[|A|/2..|A|))$
    return MERGE(L, R)

Recall that in our work-depth model we can analyze the depth of an **in parallel do** by taking the maximum depth of the two calls, and the work by taking the sum of the work of the two calls. We assume that the merging remains sequential so that the work and depth to merge two sorted sequences of $n/2$ keys is $O(n)$. Thus for mergesort the work and depth are given by the recurrences:

$$
\begin{aligned}
W(n) &= 2W(n/2) + O(n) && (2)\\
D(n) &= \max(D(n/2), D(n/2)) + O(n) && (3)\\
&= D(n/2) + O(n) && (4)
\end{aligned}
$$

As expected, the solution for the work is $W(n) = O(n \log n)$, i.e., the same as the time for the sequential algorithm. For the depth, however, the solution is $D(n) = O(n)$, which is smaller than the work. Recall that we defined the parallelism of an algorithm as the ratio of the work to the depth. Hence, the parallelism of this algorithm is $O(\log n)$ (not very much). The problem here is that the merge step remains sequential, and is the bottleneck.

As mentioned earlier, the parallelism in a divide-and-conquer algorithm can often be enhanced by parallelizing the divide step and/or the merge step. Using a parallel merge [52] two sorted sequences of $n/2$ keys can be merged with work $O(n)$ and depth $O(\log \log n)$. Using this merge algorithm, the recurrence for the depth of mergesort becomes

$$D(n) = D(n/2) + O(\log \log n), \tag{5}$$

which has solution $D(n) = O(\log n \log \log n)$. Using a technique called *pipelined divide-and-conquer* the depth of mergesort can be further reduced to $O(\log n)$ [26]. The idea is to start the merge at the top level before the recursive calls complete.

Divide-and-conquer has proven to be one of the most powerful techniques for solving problems in parallel. In this chapter, we will use it to solve problems from computational geometry, sorting, and performing Fast Fourier Transforms. Other applications range from solving linear systems to factoring large numbers to performing n-body simulations.

2.2 Randomization

Random numbers are used in parallel algorithms to ensure that processors can make local decisions which, with high probability, add up to good global decisions. Here we consider three uses of randomness.

Sampling: One use of randomness is to select a representative sample from a set of elements. Often, a problem can be solved by selecting a sample, solving the problem on that sample, and then using the solution for the sample to guide the solution for the original set. For example, suppose we want to sort a collection of integer keys. This can be accomplished by partitioning the keys into buckets and then sorting within each bucket. For this to work well, the buckets must represent non-overlapping intervals of integer values, and each bucket must contain approximately the same number of keys. Random sampling 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.

Symmetry breaking: Another use of randomness is in symmetry breaking. For example, consider the problem of selecting a large independent set of vertices in a graph in parallel. (A set of vertices is *independent* if no two are neighbors.) Imagine that each vertex must decide, in parallel with all other vertices, whether to join the set or not. Hence, if one vertex chooses to join the set, then all of its neighbors must choose not to join the set. The choice is difficult to make simultaneously by each vertex if the local structure at each vertex is the same, for example if each vertex has the same number of neighbors. As it turns out, the impasse can be resolved by using randomness to break the symmetry between the vertices [58].

Load balancing: A third use of randomness is load balancing. One way to quickly partition a large number of data items into a collection of approximately evenly sized subsets is to randomly assign each element to a subset. This technique works best when the average size of a subset is at least logarithmic in the size of the original set.

2.3 Parallel pointer techniques

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 these techniques can often be replaced by parallel techniques with roughly the same power.

Pointer jumping. One of the oldest parallel pointer techniques is *pointer jumping* [88]. This technique can be applied to either lists or trees. In each pointer jumping step, each node in parallel replaces its pointer with that of its successor (or parent). 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. After at most $\lceil \log n \rceil$ steps, every node points to the same node, the end of the list (or root of the tree). This is described in more detail in Section 3.4.

Euler tour. An Euler tour of a directed graph is a path through the graph in which every edge is traversed exactly once. In an undirected graph each edge is typically replaced with two oppositely directed edges. The Euler tour of an undirected tree follows the perimeter of the tree visiting each edge twice, once on the way down and once on the way up. By keeping a linked structure that represents the Euler tour of a tree it is possible to compute many functions on the tree, such as the size of each subtree [83]. This technique uses linear work, and parallel depth that is independent of the depth of the tree. The Euler tour can often be used to replace a standard traversal of a tree, such as a depth-first traversal.

Graph contraction. Graph contraction is an operation in which a graph is reduced in size while maintaining some of its original structure. Typically, after performing a graph contraction operation, the problem is solved recursively on the contracted graph. The solution to the problem on the contracted graph is then used to form the final solution. For example, one way to partition a graph into its connected components is to first contract the graph by merging some of the vertices with neighboring vertices, then find the connected components of the contracted graph, and finally undo the contraction operation. Many problems can be solved by contracting trees [64, 65], in which case the technique is called *tree contraction.* More examples of graph contraction can be found in Section 4.

Ear decomposition. An ear decomposition of a graph is a partition of its edges into an ordered collection of paths. The first path is a cycle, and the others are called ears. The end-points of each ear are anchored on previous paths. Once an ear decomposition of a graph is found, it is not difficult to determine if two edges lie on a common cycle. This information can be used in algorithms for determining biconnectivity, triconnectivity, 4-connectivity, and planarity [60, 63]. An ear decomposition can be found in parallel using linear work and logarithmic depth, independent of the structure of the graph. Hence, this technique can be used to replace the standard sequential technique for solving these problems, depth-first search.

2.4 Other techniques

Many other techniques have proven to be useful in the design of parallel algorithms. Finding small graph separators is useful for partitioning data among processors to reduce communication [75, Chapter 14]. Hashing is useful for load balancing and mapping addresses to memory [47, 87]. Iterative techniques are useful as a replacement for direct methods for solving linear systems [18].

3 Basic operations on sequences, lists, and trees

We begin our presentation of parallel algorithms with a collection of algorithms for performing basic operations on sequences, lists, and trees. These operations will be used as subroutines in the algorithms that follow in later sections.

3.1 Sums

As explained near the beginning of this chapter, there is a simple recursive algorithm for computing the sum of the elements in an array.

ALGORITHM: SUM(A)
if $|A| = 1$ **then return** $A[0]$
else return SUM($\{A[2i] + A[2i+1] : i \in [0..|A|/2)\}$)

The work and depth for this algorithm are given by the recurrences

$$W(n) = W(n/2) + O(n) \quad (6)$$
$$D(n) = D(n/2) + O(1) \quad (7)$$

which have solutions $W(n) = O(n)$ and $D(n) = O(\log n)$. This algorithm can also be expressed without recursion (using a **while** loop), but the recursive version forshadows the recursive algorithm for the *scan* function.

As written, the algorithm only works on sequences that have lengths equal to powers of 2. Removing this restriction is not difficult by checking if the sequence is of odd length and separately adding the last element in if it is. This algorithm can also easily be modified to compute the "sum" using any other binary associative operator in place of +. For example the use of max would return the maximum value of in sequence.

3.2 Scans

The *plus-scan* operation (also called all-prefix-sums) takes a sequence of values and returns a sequence of equal length for which each element is the sum of all previous elements in the original sequence. For example, executing a plus-scan on the sequence $[3,5,3,1,6]$ returns $[0,3,8,11,12]$. An algorithm for performing the scan operation [81] is shown below.

ALGORITHM: SCAN(A)

1 **if** $|A| = 1$ **then return** $[0]$
2 **else**
3 $\quad S = \textsc{scan}(\{A[2i] + A[2i+1] : i \in [0..|A|/2)\})$
4 $\quad R = \{$**if** $(i \ mod\ 2) = 0$ **then** $S[i/2]$ **else** $S[(i-1)/2] + A[i-1] : i \in [0..|A|)\}$
5 **return** R

The algorithm works by element-wise adding the even indexed elements of A to the odd indexed elements of A, and then recursively solving the problem on the resulting sequence (Line 3). The result S of the recursive call gives the plus-scan values for the even positions in the output sequence R. The value for each of the odd positions in R is simply the value for the preceding even position in R plus the value of the preceding position from A.

The asymptotic work and depth costs of this algorithm are the same as for the SUM operation, $W(n) = O(n)$ and $D(n) = O(\log n)$. Also, as with the SUM operation, any binary associative operator can be used in place of the $+$. In fact the algorithm described can be used more generally to solve various recurrences, such as the first-order linear recurrences $x_i = (x_{i-1} \otimes a_i) \oplus b_i$, $0 \leq i \leq n$, where $\otimes$ and $\oplus$ are both binary associative operators [51].

Scans have proven so useful in the design of parallel algorithms that some parallel machines provide support for scan operations in hardware.

3.3 Multiprefix and fetch-and-add

The *multiprefix* operation is a generalization of the scan operation in which multiple independent scans are performed. The input to the multiprefix operation is a sequence A of n pairs (k, a), where k specifies a key and a specifies an integer data value. For each key value, the multiprefix operation performs an independent scan. The output is a sequence B of n integers containing the results of each of the scans such that if $A[i] = (k, a)$ then

$$B[i] = sum(\{b : (t, b) \in A[0..i) \mid t = k\}).$$

In other words, each position receives the sum of all previous elements that have the same key. As an example,

$$\textsc{multiprefix}([(1,5),(0,2),(0,3),(1,4),(0,1),(2,2)])$$

returns the sequence

$$[0, 0, 2, 5, 5, 0].$$

The *fetch-and-add* operation is a weaker version of the multiprefix operation, in which the order of the input elements for each scan is not necessarily the same as the order in the input sequence A. In this chapter we do not present an algorithm for the multiprefix operation, but it can be solved by a function that requires work $O(n)$ and depth $O(\log n)$ using concurrent writes [61].

3.4 Pointer jumping

Pointer jumping is a technique that can be applied to both linked lists and trees [88]. The basic pointer jumping operation is simple. Each node i replaces its pointer $P[i]$ with the pointer of the node that it points to, $P[P[i]]$. By repeating this operation, it is possible to compute, for each node in a list or tree, a pointer to the end of the list or root of the tree. Given a sequence P of pointers that represent a tree (i.e., pointers from children to parents), the following code will generate a pointer from each node to the root of the tree. We assume that the root points to itself.

ALGORITHM: POINT_TO_ROOT(P)

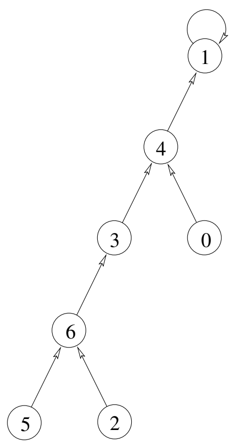
```
1  for j from 1 to ⌈log |P|⌉
2      P := {P[P[i]] : i ∈ [0..|P|)}
```

The idea behind this algorithm is that in each loop iteration the distance spanned by each pointer, with respect to the original tree, will double, until it points to the root. Since a tree constructed from $n = |P|$ pointers has depth at most $n - 1$, after $\lceil \log n \rceil$ iterations each pointer will point to the root. Because each iteration has constant depth and performs $\Theta(n)$ work, the algorithm has depth $\Theta(\log n)$ and work $\Theta(n \log n)$.

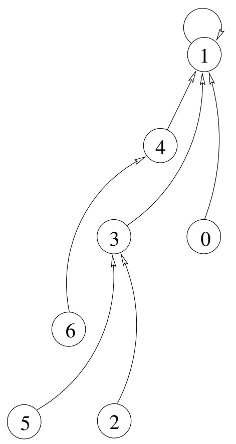
Figure 5 illustrates algorithm POINT_TO_ROOT applied to a tree consisting of seven nodes. The tree is shown before the algorithm begins and after one and two iterations of the algorithm. In each tree, every node is labeled with its index. Beneath each tree the sequence P representing the tree is shown.

3.5 List ranking

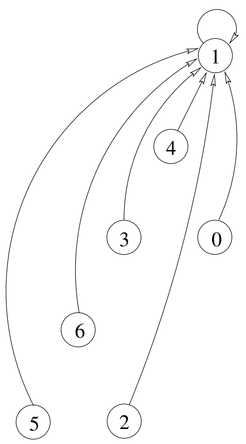
The problem of computing the distance from each node to the end of a linked list is called *list ranking*. Function POINT_TO_ROOT can be easily modified to compute these distances, as shown below.



(a) The input tree $P = [4, 1, 6, 4, 1, 6, 3]$.



(b) The tree $P = [1, 1, 3, 1, 1, 3, 4]$ after one iteration of the algorithm.



(c) The final tree $P = [1, 1, 1, 1, 1, 1, 1]$.

Figure 5: The effect of two iterations of algorithm POINT_TO_ROOT.

ALGORITHM: LIST_RANK(P)

1 $V = \{\textbf{if } P[i] = i \textbf{ then } 0 \textbf{ else } 1 : i \in [0..|P|)\}$
2 **for** j **from** 1 **to** $\lceil \log |P| \rceil$
3 $\quad V := \{V[i] + V[P[i]] : i \in [0..|P|)\}$
4 $\quad P := \{P[P[i]] : i \in [0..|P|)\}$
5 **return** V

In this function, $V[i]$ can be thought of as the distance spanned by pointer $P[i]$ with respect to the original list. Line 1 initializes V by setting $V[i]$ to 0 if i is the last node (i.e., points to itself), and 1 otherwise. In each iteration, Line 3 calculates the new length of $P[i]$. The function has depth $\Theta(\log n)$ and work $\Theta(n \log n)$.

It is worth noting that there are simple sequential algorithms that perform the same tasks as both functions POINT_TO_ROOT and LIST_RANK using only $O(n)$ work. For example, the list ranking problem can be solved by making two passes through the list. The goal of the first pass is simply to count the number of elements in the list. The elements can then be numbered with their positions from the end of the list in a second pass. Thus, neither function POINT_TO_ROOT nor LIST_RANK are work-efficient, since both require $\Theta(n \log n)$ work in the worst case. There are, however, several work-efficient parallel solutions to both of these problems.

The following parallel algorithm uses the technique of random sampling to construct a pointer from each node to the end of a list of n nodes in a work-efficient fashion [74]. The algorithm is easily generalized to solve the list-ranking problem.

1. Pick m list nodes at random and call them the *start* nodes.

2. From each start node u, follow the list until reaching the next start node v. Call the list nodes between u and v the *sublist* of u.

3. Form a shorter list consisting only of the start nodes and the final node on the list by making each start node point to the next start node on in the list.

4. Using pointer jumping on the shorter list, for each start node create a pointer to the last node in the list.

5. For each start node u, distribute the pointer to the end of the list to all of the nodes in the sublist of u.

The key to analyzing the work and depth of this algorithm is to bound the length of the longest sublist. Using elementary probability theory, it is not difficult to prove that the expected length of the longest sublist is at most $O((n \log m)/m)$. The work and depth for each step of the algorithm are thus computed as follows.

1. $W(n, m) = O(m)$ and $D(n, m) = O(1)$
2. $W(n, m) = O(n)$ and $D(n, m) = O((n \log m)/m)$
3. $W(n, m) = O(m)$ and $D(n, m) = O(1)$
4. $W(n, m) = O(m \log m)$ and $D(n, m) = O(\log m)$
5. $W(n, m) = O(n)$ and $D(n, m) = O((n \log m)/m)$

Thus, the work for the entire algorithm is $W(m, n) = O(n+m \log m)$, and the depth is $O((n \log m)/m)$. If we set $m = n/\log n$, these reduce to $W(n) = O(n)$ and $D(n) = O(\log^2 n)$.

Using a technique called *contraction*, it is possible to design a list ranking algorithm that runs in $O(n)$ work and $O(\log n)$ depth [8, 9]. This technique can also be applied to trees [64, 65].

3.6 Removing duplicates

This section presents two algorithms for removing the duplicate items that appear in a sequence. Thus, the input to each algorithm is a sequence, and the output is a new sequence containing exactly one copy of every item that appears in the input sequence. It is assumed that the order of the items in the output sequence does not matter. Such an algorithm is useful when a sequence is used to represent an unordered set of items. Two sets can be merged, for example, by first appending their corresponding sequences, and then removing the duplicate items.

3.6.1 Approach 1 : Using an array of flags

If the items are all non-negative integers drawn from a small range, we can use a technique similar to bucket sort to remove the duplicates. We begin by creating an array equal in size to the range, and initializing all of its elements to 0. Next, using concurrent writes we set a flag in the array for each number that appears in the input list. Finally, we extract those numbers with flags that have been set. This algorithm is expressed as follows.

ALGORITHM: $\textsc{rem\_duplicates}(V)$

$\textsc{range} := 1 + \textsc{max}(V)$
$\textsc{flags} := \mathit{distribute}(0, \textsc{range}) \leftarrow \{(i, 1) : i \in V\}$
return $\{j : j \in [0..\textsc{range}) \mid \textsc{flags}[j] = 1\}$

This algorithm has depth $O(1)$ and performs work $O(|V| + \textsc{max}(V))$. Its obvious disadvantage is that it explodes when given a large range of numbers, both in memory and in work.

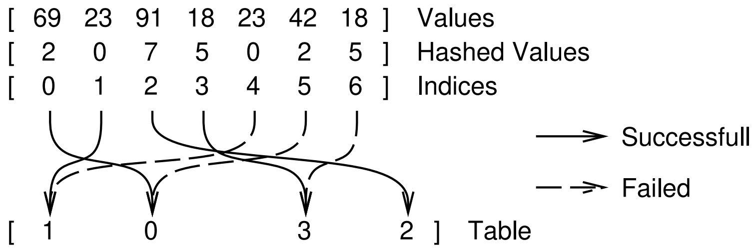


Figure 6: Each key attempts to write its index into a hash table entry.

3.6.2 Approach 2 : Hashing

A more general approach is to use a hash table. The algorithm has the following outline. The algorithm first creates a hash table that contains a prime number of entries, where the prime is approximately twice as large as the number of items in the set V. A prime size is best, because it makes designing a good hash function easier. The size must also be large enough that the chance of collisions in the hash table are not too great. Let m denote the size of the hash table. Next, the algorithm computes a hash value $hash(V[j], m)$ for each item $V[j] \in V$, and attempts to write the index j into the hash table entry $hash(V[j], m)$. For example, Figure 6 describes a particular hash function applied to the sequence [69, 23, 91, 18, 42, 23, 18]. We assume that if multiple values are simultaneously written into the same memory location, one of the values will be correctly written (the arbitrary concurrent write model). An index j is called a *winner* if the value $V[j]$ is successfully written into the hash table. In our example, the winners are $V[0]$, $V[1]$, $V[2]$, and $V[3]$, i.e., 69, 23, 91, and 18. The winners are added to the duplicate-free sequence that is being constructed, and then set aside. Among the losers, we must distinguish between two types of items, those that were defeated by an item with the same value, and those that were defeated by an item with a different value. In our example, $V[5]$ and $V[6]$ (23 and 18) were defeated by items with the same value, and $V[4]$ (42) was defeated by an item with a different value. Items of the first type are set aside because they are duplicates. Items of the second type are retained, and the algorithm repeats the entire process on them using a different hash function. In general, it may take several iterations before all of the items have been set aside, and in each iteration the algorithm must use a different hash function.

The code for removing duplicates using hashing is shown below.

ALGORITHM: REMOVE_DUPLICATES(V)

```
1   m := NEXT_PRIME(2 * |V|)
2   TABLE := distribute(−1, m)
3   i := 0
4   RESULT := {}
5   while |V| > 0
6      TABLE := TABLE ← {(hash(V[j], m, i), j) : j ∈ [0..|V|)}
7      WINNERS := {V[j] : j ∈ [0..|V|) | TABLE[hash(V[j], m, i)] = j}
8      RESULT := RESULT ++ WINNERS
9      TABLE := TABLE ← {(hash(k, m, i), k) : k ∈ WINNERS}
10     V := {k ∈ V | TABLE[hash(k, m, i)] ≠ k}
11     i := i + 1
12  return RESULT
```

The first four lines of function REMOVE_DUPLICATES initialize several variables. Line 1 finds first prime number larger than $2 * |V|$ using the built-in function NEXT_PRIME. Line 2 creates the hash table, and initializes its entries with an arbitrary value (-1). Line 3 initializes i, a variable that simply counts iterations of the **while** loop. Line 4 initializes the sequence RESULT to be empty. Ultimately, RESULT will contain a single copy of each distinct item from the sequence V.

The bulk of the work in function REMOVE_DUPLICATES is performed by the **while** loop. While there are items remaining to be processed, the code performs the following steps. In Line 6, each item $V[j]$ attempts to write its index j into the table entry given by the hash function $hash(V[j], m, i)$. Note that the hash function takes the iteration i as an argument, so that a different hash function is used in each iteration. Concurrent writes are used so that if several items attempt to write to the same entry, precisely one will win. Line 7 determines which items successfully wrote indices in Line 6, and stores the values of these items in an array called WINNERS. The winners are added to the RESULT in Line 8. The purpose of Lines 9 and 10 is to remove all of the items that are either winners or duplicates of winners. These lines reuse the hash table. In Line 9, each winner writes its value, rather than its index, into the hash table. In this step there are no concurrent writes. Finally, in Line 10, an item is retained only if it is not a winner, and the item that defeated it has a different value.

It is not difficult to prove that, provided that the hash values $hash(V[j], m, i)$ are random and sufficiently independent, both between iterations and within an iteration, each iteration reduces the number of items remaining by some constant fraction until the number of items remaining is small. As a consequence, $D(n) = O(\log n)$ and $W(n) = O(n)$.

4 Graphs

Graph problems are often difficult to parallelize since many standard sequential graph techniques, such as depth-first or priority-first search, do not parallelize well. For some problems, such as minimum-spanning tree and biconnected components, new techniques have been developed to generate efficient parallel algorithms. For other problems, such as single-source shortest paths, there are no known efficient parallel algorithms, at least not for the general case.

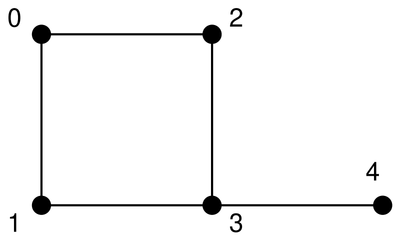
We have already outlined some of the parallel graph techniques in Section 2. In this section we describe algorithms for breadth-first-search, connected components and minimum-spanning-trees. These algorithms use some of the general techniques. In particular, randomization and graph contraction will play an important role in the algorithms. In this chapter we will limit ourselves to algorithms on sparse undirected graphs. We suggest the following sources for further information on parallel graph algorithms [75, Chapters 2-8], [45, Chapter 5], [35, Chapter 2].

4.1 Graphs and graph representations

A *graph* $G = (V, E)$ consists of a set of *vertices* V and a set of *edges* E in which each edge connects two vertices. In a *directed graph* each edge is directed from one vertex to another, while in an *undirected graph* each edge is symmetric, *i.e.*, goes in both directions. A *weighted graph* is a graph in which each edge $e \in E$ has a weight $w(e)$ associated with it. In this chapter we will use the convention that $n = |V|$ and $m = |E|$. Qualitatively, a graph is considered sparse if m is much less than n^2 and dense otherwise. The *diameter* of a graph, denoted $D(G)$, is the maximum, over all pairs of vertices (u, v), of the minimum number of edges that must be traversed to get from u to v.

There are three standard representations of graphs used in sequential algorithms: edge lists, adjacency lists, and adjacency matrices. An *edge list* consists of a list of edges, each of which is a pair of vertices. The list directly represents the set E. An *adjacency list* is an array of lists. Each array element corresponds to one vertex and contains a linked list of pointers to the neighboring vertices, *i.e.,* the linked list for a vertex v contains pointers to the vertices $\{u|(v, u) \in E\}$). An *adjacency matrix* is an $n \times n$ array A such that A_{ij} is 1 if $(i, j) \in E$ and 0 otherwise. The adjacency matrix representation is typically used only when the graph is dense since it requires $\Theta(n^2)$ space, as opposed to $\Theta(m)$ space for the other two representations. Each of these representations can be used to represent either directed or undirected graphs.

For parallel algorithms we use similar representations for graphs. The main change we make is to replace the linked lists with arrays. In particular the edge-list is represented as an array of edges and the adjacency-list is represented as an array of arrays. Using arrays instead of lists makes it



(a)

[(0,1), (0,2), (2,3), (3,4), (1,3), (1,0), (2,0), (3,2), (4,3), (3,1)]

(b)

[[1, 2], [0, 3], [0, 3], [1, 2, 4], [3]]

(c)

Figure 7: Representations of an undirected graph. (a) A graph G with 5 vertices and 5 edges. (b) An edge-list representation of G. (c) The adjacency-list representation of G. Values between square brackets are elements of an array, and values between parentheses are elements of a pair.

easier to process the graph in parallel. In particular, they make it easy to grab a set of elements in parallel, rather than having to follow a list. Figure 7 shows an example of our representations for an undirected graph. Note that for the edge-list representation of the undirected graph each edge appears twice, once in each direction (this property is important for some of the algorithms described in this chapter[1]). To represent a directed graph we simply only store the edge once in the desired direction. In the text we will refer to the left element of an edge pair as the *source vertex* and the right element as the *destination vertex*.

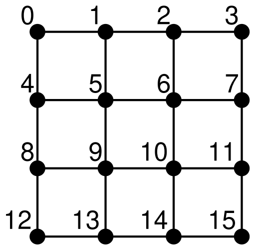
In designing algorithms, it is sometimes more efficient to use an edge list and sometimes more efficient to use an adjacency list. It is therefore important to be able to convert between the two representations. To convert from an adjacency list to an edge list (representation c to representation b in Figure 7) is straightforward. The following code will do it with linear work and constant depth

$$\mathit{flatten}(\{\{(j,i) : j \in G[i]\} : i \in [0..|G|\})$$

where G is the graph in the adjacency list representation. For each vertex i this code pairs up each of i's neighbors with i. The *flatten* is used since the nested apply-to-each will return a sequence of sequences which needs to be flattened into a single sequence.

To convert from an edge list to an adjacency list is somewhat more involved, but still requires only linear work. The basic idea is to sort the edges based on the source vertex. This places edges from a particular vertex in consecutive positions in the resulting array. This array can then be partitioned into blocks based on the source vertices. It turns out that since the sorting is on integers in the range $[0..|V|)$, a radix sort can be used (see Section 5.2), which requires linear work. The depth of the radix sort depends on the depth of the multiprefix operation (see Section 3.3.

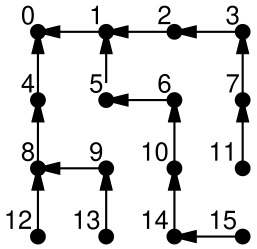
[1]If space is of serious concern, the algorithms can be easily modified to work with edges stored in just one direction.



(a)

Step	Frontier
0	[0]
1	[1, 4]
2	[2, 5, 8]
3	[3, 6, 9, 12]
5	[7, 10, 13]
6	[11, 14]
7	[15]

(b)



(c)

Figure 8: Example of Parallel Breadth First Search. (a) A graph G. (b) The frontier at each step of the BFS of G with $s = 0$. (c) A BFS tree.

4.2 Breadth first search

The first algorithm we consider is parallel breadth first search (BFS). BFS can be used to solve problems such as determining if a graph is connected or generating a spanning tree of a graph. Parallel BFS is similar to the sequential version, which starts with a source vertex s and visits levels of the graph one after the other using a queue to keep track of vertices that have not yet been visited. The main difference is that each level is going to be visited in parallel and no queue is required. As with the sequential algorithm each vertex will only be visited once and each edge at most twice, once in each direction. The work is therefore linear in the size of the graph, $O(n + m)$. For a graph with diameter D, the number of levels visited by the algorithm will be at least $D/2$ and at most D, depending on where the search is initiated. We will show that each level can be visited in constant depth, assuming a concurrent-write model, so that the total depth of parallel BFS is $O(D)$.

The main idea of parallel BFS is to maintain a set of frontier vertices, which represent the current level being visited, and to produce a new frontier on each step. The set of frontier vertices is initialized with the singleton s (the source vertex). A new frontier is generated by collecting all the neighbors of the current frontier vertices in parallel and removing any that have already been visited. This is not sufficient on its own, however, since multiple vertices might collect the same unvisited vertex. For example, consider the graph in Figure 8. On step 2 vertices 5 and 8 will both collect vertex 9. The vertex will therefore appear twice in the new frontier. If the duplicate vertices are not removed the algorithm can generate an exponential number of vertices in the frontier. This problem does not occur in the sequential BFS because vertices are visited one at a time. The parallel version therefore requires an extra step to remove duplicates.

The following function performs a parallel BFS. It takes as input a source vertex s and a graph G represented as an adjacency-array, and returns as its result a breadth-first-search tree of G. In a BFS tree each vertex visited at level i points to one of its neighbors visited at level $i-1$ (see Figure 8(c)). The source s is the root of the tree.

ALGORITHM: BFS(s, G)

1 FRONT $:= [s]$
2 TREE $:= \textit{distribute}(-1, |G|)$
3 TREE$[s] := s$
4 **while** $(|\text{FRONT}| \neq 0)$
5 $\quad E := \textit{flatten}(\{\{(u,v) : u \in G[v]\} : v \in \text{FRONT}\})$
6 $\quad E' := \{(u,v) \in E \mid \text{TREE}[u] = -1\}$
7 $\quad$ TREE := TREE $\leftarrow E'$
8 $\quad$ FRONT $:= \{u : (u,v) \in E' \mid v = \text{TREE}[u]\}$
9 **return** TREE

In this code FRONT contains the set of frontier vertices, and TREE contains the current BFS tree, represented as an array of indices (pointers). The pointers (indices) in TREE are all initialized to -1, except for the source s which is initialized to point to itself. Each vertex in TREE is set to point to its parent in the BFS tree when it is visited. The algorithm assumes the arbitrary concurrent write model.

We now consider each iteration of the algorithm. The iterations terminate when there are no more vertices in the frontier (Line 4). The new frontier is generated by first collecting into an edge-array the set of edges from current frontier vertices to the neighbors of these vertices (Line 5). An edge from v to u is kept as the pair (u,v) (this is backwards from the standard edge representation and is used below to write from v to u). Next, the algorithm subselects the edges that lead to unvisited vertices (Line 6). Now for each remaining edge (u,v) the algorithm writes the source index v into into the destination vertex u (Line 7). In the case that more than one edge has the same destination, one of the source indices will be written arbitrarily—this is the only place that the algorithm uses a concurrent write. These indices become the parent pointers for the BFS tree, and are also used to remove duplicates for the next frontier set. In particular, the algorithm checks whether each edge succeeded in writing its source by reading back from the destination. If an edge reads the value it wrote, its destination is included in the new frontier (Line 8). Since only one edge that points to a given destination vertex will read back the same value, no duplicates will appear in the new frontier.

The algorithm requires only constant depth per iteration of the while loop. Since each vertex and its associated edges are visited only once, the total work is $O(m+n)$. An interesting aspect of this parallel BFS is that it can generate BFS trees that cannot be generated by a sequential BFS, even allowing for any order of visiting neighbors in sequential BFS. We leave the generation of an example as an exercise. We note, however, that if the algorithm used a priority concurrent write (see Section 1.5) on Line 7, then it would generate the same tree as a sequential BFS.

4.3 Connected components

We now consider the problem of labeling the connected components of an undirected graph. The problem is to label all the vertices in a graph G such that two vertices u and v have the same label if and only if there is a path between the two vertices. Sequentially the connected components of a graph can easily be labeled using either depth-first or breadth-first search. We have seen how to perform a breadth-first search, but the technique requires a depth proportional to the diameter of a graph. This is fine for graphs with small diameter, but does not work well in the general case. Unfortunately, in terms of work, even the most efficient polylogarithmic depth parallel algorithms for depth-first search and breadth-first search are very inefficient. Hence, the efficient algorithms for solving the connected components problem use different techniques.

The two algorithms we consider are based on *graph contraction*. Graph contraction works by contracting the vertices of a connected subgraph into a single vertex. The techniques we use allow the algorithms to make many such contractions in parallel across the graph. The algorithms therefore proceed in a sequence of steps, each of which contracts a set of subgraphs, and forms a smaller graph in which each subgraph has been converted into a vertex. If each such step of the algorithm contracts the size of the graph by a constant fraction, then each component will contract down to a single vertex in $O(\log n)$ steps. By running the contraction in reverse, the algorithms can label all the vertices in the components. The two algorithms we consider differ in how they select subgraphs for contraction. The first uses randomization and the second is deterministic. Neither algorithm is work efficient because they require $O((n+m)\log n)$ work for worst-case graphs, but we briefly discuss how they can be made work efficient in Section 4.3.3. Both algorithms require the concurrent write model.

4.3.1 Random mate graph contraction

The random mate technique for graph contraction is based on forming a set of star subgraphs and contracting the stars. A *star* is a tree of depth one—it consists of a root and an arbitrary number of children. The random mate algorithm finds a set of non-overlapping stars in a graph, and then

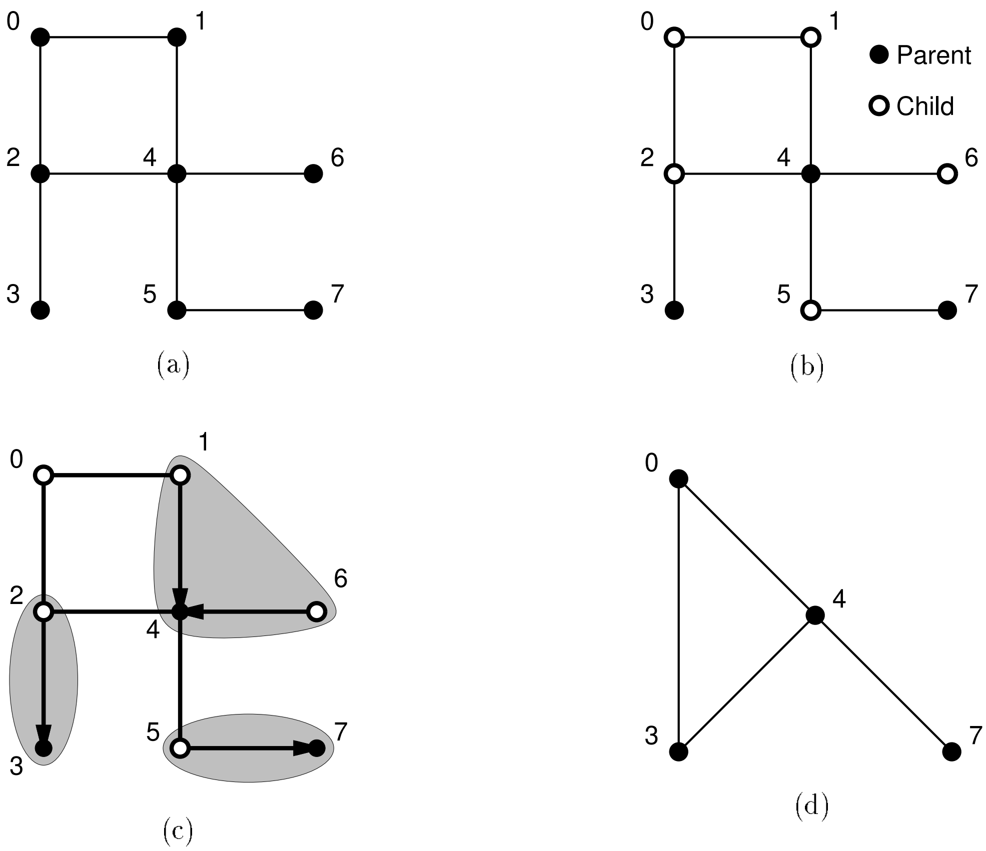


Figure 9: Example of one step of random mate graph contraction. (a) The original graph G. (b) G after selecting the parents randomly. (c) The stars formed after each child is assigned a neighboring parent as its root (each star is shaded). (d) The graph after contracting each star and relabeling the edges to point to the roots. Children with no neighboring parent remain in the graph.

contracts each star into a single vertex by merging each child into its parent. The technique used to form the stars is based on randomization. For each vertex the algorithm flips a coin to decide if that vertex is a parent or a child. We assume the coin is unbiased so that every vertex has a 50% probability of being a parent. Now for each child vertex the algorithm selects a neighboring parent vertex and makes that parent the child's root. If the child has no neighboring parent, it has no root and will be called an *orphan*. The parents are now the roots of a set of stars, each with zero or more children. The children are either orphans or belong to one of these stars. The algorithm now contracts the stars. When contracting, the algorithm updates any edge that points to a child of a star to point to the root. Figure 9 illustrates a full contraction step. This contraction step is repeated until all components are of size 1.

To analyze the number of contraction steps required to complete we need to know how many vertices the algorithm removes on each contraction step. First we note that a contraction step is only going to remove children, and only if they have a neighboring parent. The probability that a vertex will be removed is therefore the probability that a vertex is a child multiplied by the probability that at least one of its neighbors is a parent. The probability that it is a child is 1/2 and the probability that at least one neighbor is a parent is at least 1/2 (every vertex that is not fully contracted has one or more neighbors). The algorithm is therefore expected to remove at least 1/4 of the remaining vertices at each step, and since this is a constant fraction, it is expected to complete in $O(\log n)$ steps. The full probabilistic analysis is somewhat involved since it is possible to have a streak of bad flips, but it is not too hard to show that the algorithm is very unlikely to require more than $O(\log n)$ contraction steps.

The following algorithm uses random mate contraction for labeling the connected components of a graph. The algorithm works by contracting until each component is a single vertex and then re-expanding so that it can label all vertices in that component with the same label. The input to the algorithm is a graph G in the edge-list representation (note that this is a different representation than used in BFS), along with the labels of the vertices. The labels of the vertices are initialized to be unique indices in the range $0..(|V|-1)$. The output of the algorithm is a label for each vertex such that two vertices will have the same label if and only if they belong to the same component. In fact, the label of each vertex will be the original label of one of the vertices in the component.

ALGORITHM: CC_RANDOM_MATE(LABELS, E)
1 **if** $(|E| = 0)$ **then return** LABELS
2 **else**
3 CHILD := $\{$*rand_bit()* $: v \in [1..n]\}$
4 HOOKS := $\{(u, v) \in E \mid$ CHILD$[u]$ `and` $\neg$CHILD$[v]\}$
5 LABELS := LABELS $\leftarrow$ HOOKS
6 E' := $\{($LABELS$[u]$, LABELS$[v]) : (u, v) \in E \mid$ LABELS$[u] \neq$ LABELS$[v]\}$
7 LABELS$'$:= CC_RANDOM_MATE(LABELS, E')
8 LABELS$'$:= LABELS$'$ $\leftarrow \{(u,$ LABELS$'[v]) : (u, v) \in$ HOOKS$\}$
9 **return** LABELS$'$

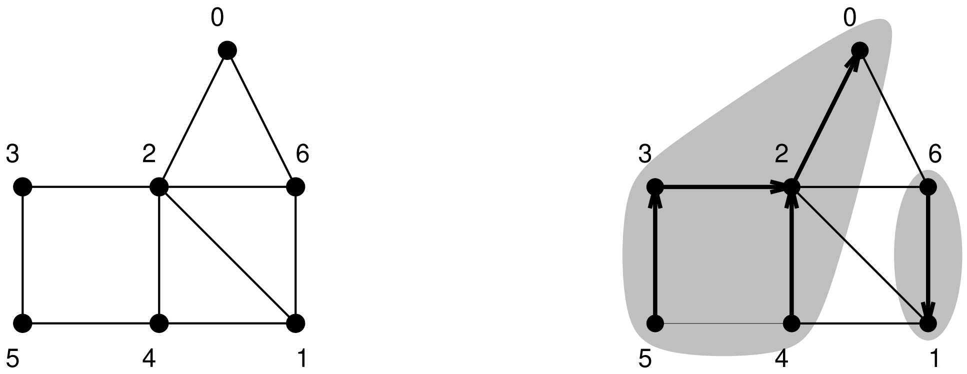
The algorithm works recursively by (a) executing one random-mate contraction step (b) recursively applying itself to the contracted graph, and (c) re-expanding the graph by passing the labels from each root of a contracted star (from step a) to its children. The graph is therefore contracted while going down the recursion and re-expanded while coming back up. The termination condition is when there are no remaining edges (Line 1). To form stars for the contraction step the algorithm flips a coin for each vertex (Line 3) and subselects all edges HOOKS that go from a child to a parent (Line 4). We call these edges *hook edges* and they represent a superset of the star edges (each child can have multiple hook edges, but only one parent in a star). For each hook edge the algorithm writes the parent's label into the child's label (Line 5). If a child has multiple neighboring parents, then one of the parent's labels is written arbitrarily—we assume an arbitrary concurrent write. At this point each child is labeled with one of its neighboring parents, if it has one. The algorithm now updates each edge by reading the labels from its two endpoints and using these as its new endpoints (Line 6). In the same step, the algorithm removes any edges that are within the same star. This gives a new sequence of edges E'. The algorithm has now completed the contraction step, and calls itself recursively on the contracted graph (Line 7). On returning from the recursive call, the LABELS$'$ that are returned are passed on to the children of the stars, effectively re-expanding the graph by one step (Line 8). The same hooks as were used for contraction can be used for this update.

Two things should be noted about this algorithm. First, the algorithm flips coins on all of the vertices on each step even though many have already been contracted (there are no more edges that point to them). It turns out that this will not affect our worst-case asymptotic work or depth bounds, but it is not difficult to flip coins only on active vertices by keeping track of them—just keep an array of the labels of the active vertices. Second, if there are cycles in the graph, then the algorithm will create redundant edges in the contracted subgraphs. Again, keeping these edges is not a problem for the correctness or cost bounds, but they could be removed using hashing as discussed in Section 3.6.

To analyze the full work and depth of the algorithm we note that each step only requires constant depth and $O(n+m)$ work. Since the number of steps is $O(\log n)$ with high probability, as mentioned earlier, the total depth is $O(\log n)$ and the work is $O((n+m)\log n)$, both with high probability. One might expect that the work would be linear since the algorithm reduces the number of vertices on each step by a constant fraction. We have no guarantee, however, that the number of edges is also going to contract geometrically, and in fact for certain graphs they will not. In Section 4.3.3 we discuss how to improve the algorithm so that it is work-efficient.

4.3.2 Deterministic graph contraction

Our second algorithm for graph contraction is deterministic [41]. It is based on forming a set of disjoint subgraphs, each of which is a tree, and then using the POINT_TO_ROOT routine (Section 3.4) to contract each subgraph to a single vertex. To generate the trees, the algorithm hooks each vertex into a neighbor with a smaller label (by *hooking* a into b we mean making b the parent of a). Vertices with no smaller-labeled neighbors are left unhooked. The result of the hooking is a set of disjoint trees—hooking only from larger to smaller guarantees there are no cycles. Figure 10 shows an example of a set of trees created by hooking. Since a vertex can have more than one neighbor with



(a) (b)

Figure 10: Tree-based graph contraction. (a) A graph G. (b) The hook edges induced by hooking larger to smaller vertices and the subgraphs induced by the trees.

a smaller label, a given graph can have many different hookings. For example, in Figure 10 vertex 2 could have hooked into vertex 1 instead of vertex 0.

The following algorithm uses this tree-based graph contraction for labeling the connected components of a graph. It uses the same input format as the random-mate contraction algorithm.

ALGORITHM: CC_TREE_CONTRACT(LABELS, E)

1 **if** $(|E| = 0)$
2 **then return** LABELS
3 **else**
4 HOOKS := $\{(u, v) \in E \mid u > v\}$
5 LABELS := LABELS $\leftarrow$ HOOKS
6 LABELS := POINT_TO_ROOT(LABELS)
7 $E' := \{(\text{LABELS}[u], \text{LABELS}[v]) : (u, v) \in E \mid \text{LABELS}[u] \neq \text{LABELS}[v]\}$
8 **return** CC_TREE_CONTRACT(LABELS, E')

The structure of the algorithm is similar to the random-mate graph contraction algorithm. The main differences are how the hooks are selected (Line 4), the pointer jumping step to contract the trees (Line 6), and the fact that no relabeling is required when returning from the recursive call. The hooking step simply selects edges that point from larger numbered vertices to smaller numbered vertices. This is called a *conditional hook*. The pointer jumping step uses the algorithm in Section 3.4. This labels every vertex in the tree with the root of the tree. The edge relabeling is the same as in the random-mate algorithm. The reason the contraction algorithm does not need to relabel the vertices after the recursive call is that the pointer jumping step will do the relabeling.

Although the basic algorithm we have described so far works well in practice, in the worst case it can take $n-1$ steps. Consider the graph in Figure 11(a). After hooking and contracting only one

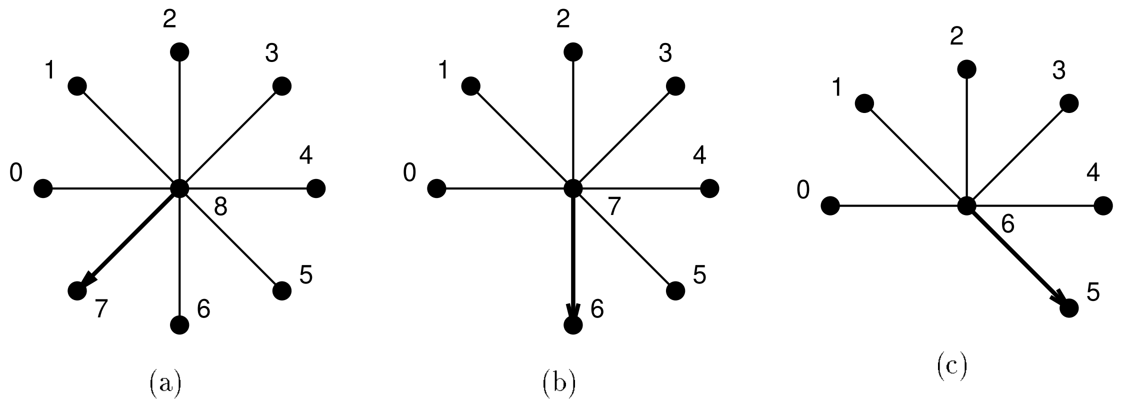


Figure 11: A worst case graph. (a) A star graph G with the maximum index at the root of the star. (b) G after one step of contraction. (c) G after two steps of contraction.

vertex has been removed. This could be repeated up to $n-1$ times. This worst case behavior can be avoided by trying to hook in both directions (from larger to smaller and from smaller to larger) and picking the hooking that hooks more vertices. We will make use of the following lemma.

Lemma 4.1 *Let $G = (V, E)$ be an undirected graph in which each vertex has at least one neighbor, then either $|\{u \mid (u, v) \in E, u < v\}| \geq |V|/2$ or $|\{u \mid (u, v) \in E, u > v\}| > |V|/2$.*

Proof: Every vertex must either have a neighbor with a lesser index or a neighbor with a greater index. This means that if we consider the set of vertices with a lesser neighbor and the set of vertices with a greater neighbor, then one of those sets must consist of at least one half the vertices. □

This lemma will guarantee that if we try hooking in both directions and pick the better one we will remove at least 1/2 of the vertices on each step, so that the number of steps is bounded by $\lceil \log_2 n \rceil$.

We now consider the total cost of the algorithm. The hooking and relabeling of edges on each step takes $O(m)$ work and constant depth. The tree contraction using pointer jumping on each step requires $O(n \log n)$ work and $O(\log n)$ depth, in the worst case. Since there are $O(\log n)$ steps, in the worst case, the total work is $O((m + n \log n) \log n)$ and depth $O(\log^2 n)$. However, if we keep track of the active vertices (the roots), and only pointer jump on active vertices then the work is reduced to $O((m + n) \log n)$ since the number of vertices decreases geometrically in each step. This requires that the algorithm expands the graph on the way back up the recursion as done for the random-mate algorithm. The total work with this modification is the same work as the randomized technique, although the depth has increased.

4.3.3 Improved versions of connected components

There are many improvements to the two basic connected component algorithms we described. Here we mention some of them.

The deterministic algorithm can be improved to run in $O(\log n)$ depth with the same work bounds [13, 79]. The basic idea is to interleave the hooking steps with the pointer-jumping steps. By a pointer-jumping step we mean a single step of POINT_TO_ROOT. This means that each tree is only partially contracted when executing the next hooking step, and to avoid creating cycles the algorithm must always hook in the same direction (*i.e.*, from smaller to larger). The technique of alternating hooking directions (used to deal with graphs such as the one in Figure 11) therefore cannot be used. Instead each vertex checks if it belongs to any tree after hooking. If it does not then it can hook to any neighbor, even if it has a larger index. This is called an *unconditional hook*.

The randomized algorithm can be improved to run in optimal work, $O(n+m)$ [33]. The basic idea is to not use all of the edges for hooking on each step, and instead use a sample of the edges. This technique, first applied to parallel algorithms, has since been used to improve some sequential algorithms, such as deriving the first linear-work algorithm for finding a minimum spanning tree [46].

Another improvement is to use the EREW model instead of requiring concurrent reads and writes [42]. However this comes at the cost of greatly complicating the algorithm. The basic idea is to keep circular linked lists of the neighbors of each vertex, and then to splice these lists when merging vertices.

4.3.4 Extensions to spanning trees and minimum spanning trees

The connected component algorithms can be extended to finding a spanning tree of a graph or minimum spanning tree of a weighted graph. In both cases we assume the graphs are undirected.

A *spanning tree* of a connected graph $G = (V, E)$ is a connected graph $T = (V, E')$ such that $E' \subseteq E$ and $|E'| = |V| - 1$. Because of the bound on the number of edges, the graph T cannot have any cycles and therefore forms a tree. Any given graph can have many different spanning trees.

It is not hard to extend the connectivity algorithms to return the spanning tree. In particular, whenever components are hooked together the algorithm can keep track of which edges were used for hooking. Since each edge will hook together two components that are not connected yet, and only one edge will succeed in hooking the components, the collection of these edges across all steps will form a spanning tree (they will connect all vertices and have no cycles). To determine which edges were used for contraction, each edge checks if it successfully hooked after the attempted hook.

A *minimum spanning tree* of a connected weighted graph $G = (V, E)$ with weights $w(e)$ for

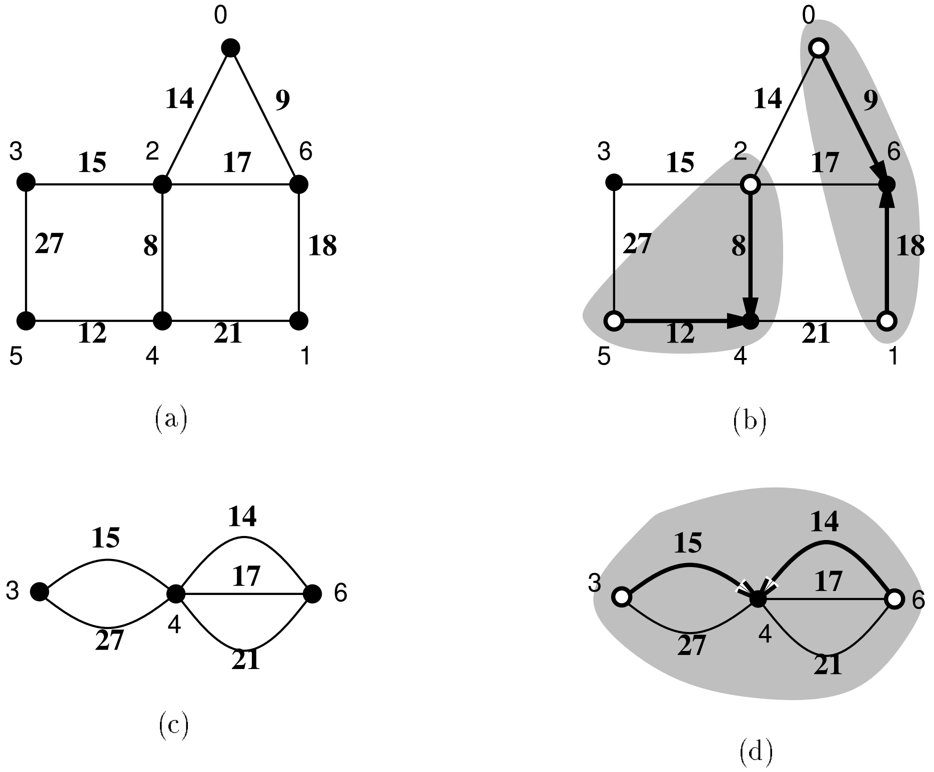


Figure 12: Example of the random-mate minimum-spanning-tree algorithm. (a) The original weighted graph G. (b) Each child (light) hooks across its minimum weighted edge to a parent (dark), if the edge is incident on a parent. (c) The graph after one step of contraction. (d) The second step in which children hook across minimum weighted edges to parents.

$e \in E$ is a spanning tree $T = (V, E')$ of G such that

$$w(T) = \sum_{e \in E'} w(e) \tag{8}$$

is minimized. The connected component algorithms can also be extended to find a minimum spanning tree. Here we will briefly consider an extension of the random-mate technique. Let us assume, without loss of generality, that all of the edge weights are distinct. If this is not the case, then lexicographical information can be added to the edges weights to break ties. It is well known that if the edge weights are distinct, then there is a unique minimum spanning tree. Furthermore, given any $W \subset V$, the minimum weight edge from W to $V - W$ must be in the minimum spanning tree. As a consequence, the minimum edge incident on a vertex will be in some minimum spanning tree. This will be true even after we contract subgraphs into vertices, since each subgraph is a subset of V.

For the minimum-spanning-tree algorithm, we modify the random mate technique so that each child u instead of picking an arbitrary parent to hook into, finds the incident edge (u, v) with minimum weight and hooks into v if it is a parent. If v is not a parent, then the child u does nothing (it is left as an orphan). Figure 12 illustrates the algorithm. As with the spanning-tree algorithm, we keep track of the hook edges and add them to a set E'. This new rule will still remove 1/4 of the vertices on each step on average since a vertex has 1/2 probability of being a child, and there is 1/2 probability that the vertex at the other end of the minimum edge is a parent. The one complication in this minimum spanning-tree algorithm is finding for each child the incident edge with minimum weight. Since we are keeping an edge list, this is not trivial to compute. If the algorithm used an adjacency list, then it would be easy, but since the algorithm needs to update the endpoints of the edges, it is not easy to maintain the adjacency list. One way to solve this problem is to use a priority concurrent write. In such a write, if multiple values are written to the same location, the one coming from the leftmost position will be written. With such a scheme the minimum edge can be found by presorting the edges by weight so the lowest weighted edge will always win when executing a concurrent write. Assuming a priority write, this minimum-spanning-tree algorithm has the same work and depth as the random-mate connected components algorithm.

There is also a linear-work logarithmic-depth randomized algorithm for finding a minimum-spanning tree [27], but it is somewhat more complicated than the linear-work algorithms for finding connected components.

5 Sorting

Sorting is a problem that admits a variety of parallel solutions. In this section we limit our discussion to two parallel sorting algorithms, QuickSort and radix sort. Both of these algorithms are easy to program, and both work well in practice. Many more sorting algorithms can be found in the literature. The interested reader is referred to [3, 45, 55] for more complete coverage.

5.1 QuickSort

We begin our discussion of sorting with a parallel version of QuickSort. This algorithm is one of the simplest to code.

ALGORITHM: QUICKSORT(A)

if $|A| = 1$ **then return** A
$i := rand\_int(|A|)$
$p := A[i]$
in parallel do
  $L :=$ QUICKSORT($\{a : a \in A \mid a < p\}$)
  $E := \{a : a \in A \mid a = p\}$
  $G :=$ QUICKSORT($\{a : a \in A \mid a > p\}$)
return L ++ E ++ G

We can make an optimistic estimate of the work and depth of this algorithm by assuming that each time a partition element p is selected, it divides the set A so that neither L nor G has more than half of the elements. In this case, the work and depth are given by the recurrences

$$W(n) = 2W(n/2) + O(n) \tag{9}$$

$$D(n) = D(n/2) + 1 \tag{10}$$

which have solutions $W(n) = O(n \log n)$ and $D(n) = O(\log n)$. A more sophisticated analysis [50] shows that the expected work and depth are indeed $W(n) = O(n \log n)$ and $D(n) = O(\log n)$, independent of the values in the input sequence A.

In practice, the performance of parallel QuickSort can be improved by selecting more than one partition element. In particular, on a machine with P processors, choosing $P - 1$ partition elements divides the keys into P sets, each of which can be sorted by a different processor using a fast sequential sorting algorithm. Since the algorithm does not finish until the last processor finishes, it is important to assign approximately the same number of keys to each processor. Simply choosing $p - 1$ partition elements at random is unlikely to yield a good partition. The partition can be improved, however, by choosing a larger number, sp, of candidate partition elements at random, sorting the candidates (perhaps using some other sorting algorithm), and then choosing the candidates with ranks $s, 2s, \ldots, (p-1)s$ to be the partition elements. The ratio s of candidates to partition elements is called the *oversampling ratio*. As s increases, the quality of the partition increases, but so does the time to sort the sp candidates. Hence there is an optimum value of s, typically larger than one, that minimizes the total time. The sorting algorithm that selects partition elements in this fashion is called *sample sort* [23, 89, 76].

5.2 Radix sort

Our next sorting algorithm is radix sort, an algorithm that performs well in practice. Unlike QuickSort, radix sort is not a *comparison sort*, meaning that it does not compare keys directly in order to determine the relative ordering of keys. Instead, it relies on the representation of keys as b-bit integers.

The basic radix sort algorithm (whether serial or parallel) examines the keys to be sorted one "digit" position at a time, starting with the least significant digit in each key. Of fundamental importance is that this intermediate sort on digits be *stable*: the output ordering must preserve the input order of any two keys with identical digit values in the position being examined.

The most common implementation of the intermediate sort is as a counting sort. A counting sort first counts to determine the *rank* of each key—its position in the output order—and then permutes the keys by moving each key to the location indicated by its rank. The following algorithm performs radix sort assuming one-bit digits.

ALGORITHM: RADIX_SORT(A, b)

```
1  for i from 0 to b − 1
2      FLAGS := {(a >> i) mod 2 : a ∈ A}
3      NOTFLAGS := {1 − b : b ∈ FLAGS}
4      R0 := SCAN(NOTFLAGS)
5      s0 := SUM(NOTFLAGS)
6      R1 := SCAN(FLAGS)
7      R := {if FLAGS[j] = 0 then R0[j] else R1[j] + s0 : j ∈ [0..|A|)}
8      A := A ← {(R[j], A[j]) : j ∈ [0..|A|)}
9  return A
```

For keys with b bits, the algorithm consists of b sequential iteration of a **for** loop, each iterations sorting according to one of the bits. Lines 2 and 3 compute the value and inverse value of the bit in the current position for each key. The notation $a >> i$ denotes the operation of shifting a to the right by i bit positions. Line 4 computes the rank of each key that has bit value 0. Computing the ranks of the keys with bit value 1 is a little more complicated, since these keys follow the keys with bit value 0. Line 5 computes the number of keys with bit value 0, which serves as the rank of the first key that has bit value 1. Line 6 computes the relative order of the keys with bit value 1. Line 7 merges the ranks of the even keys with those of the odd keys. Finally, Line 8 permutes the keys according to rank.

The work and depth of RADIX_SORT are computed as follows. There are b iterations of the **for** loop. In each iteration, the depths of Lines 2, 3, 7, 8, and 9 are constant, and the depths of Lines 4, 5, and 6 are $O(\log n)$. Hence the depth of the algorithm is $O(b \log n)$. The work performed by each of Lines 2 through 9 is $O(n)$. Hence, the work of the algorithm is $O(bn)$.

The radix sort algorithm can be generalized so that each b-bit key is viewed as b/r blocks of r bits each, rather than as b individual bits. In the generalized algorithm, there are b/r iterations of the

for loop, each of which invokes the SCAN function 2^r times. When r is large, a multiprefix operation can be used for generating the ranks instead of executing a SCAN for each possible value [23]. In this case, and assuming the multiprefix operation runs in linear work, it is not hard to show that as long as $b = O(\log n)$, the total work for the radix sort is $O(n)$, and the depth is the same order as the depth of the multiprefix operation.

Floating-point numbers can also be sorted using radix sort. With a few simple bit manipulations, floating point keys can be converted to integer keys with the same ordering and key size. For example, IEEE double-precision floating-point numbers can be sorted by first inverting the mantissa and exponent bits if the sign bit is 1, and then inverting the sign bit. The keys are then sorted as if they were integers.

6 Computational geometry

Problems in computational geometry involve calculating properties of sets of objects in a k-dimensional space. Some standard problems include finding the minimum distance between any two points in a set of points (closest-pair), finding the smallest convex region that encloses a set of points (convex-hull), and finding line or polygon intersections. Efficient parallel algorithms have been developed for most standard problems in computational geometry. Many of the sequential algorithms are based on divide-and-conquer and lead in a relatively straightforward manner to efficient parallel algorithms. Some others are based on a technique called plane sweeping, which does not parallelize well, but for which an analogous parallel technique, the *plane sweep tree* has been developed [1, 10]. In this section we describe parallel algorithms for two problems in two dimensions—closest pair and convex hull. For convex hull we describe two algorithms. These algorithms are good examples of how sequential algorithms can be parallelized in a straightforward manner.

We suggest the following sources for further information on parallel algorithms for computational geometry [6], [39], [45, Chapter 6], and [75, Chapters 9 and 11],

6.1 Closest pair

The *closest-pair problem* takes a set of points in k dimensions and returns the two points that are closest to each other. The distance is usually defined as Euclidean distance. Here we describe a closest pair algorithm for two dimensional space, also called the planar closest-pair problem. The algorithm is a parallel version of a standard sequential algorithm [17, 16], and for n points, it requires the same work as the sequential versions, $O(n \log n)$, and has depth $O(\log^2 n)$. The work is optimal.

The algorithm uses divide-and-conquer based on splitting the points along lines parallel to the y axis, and is expressed as follows.

ALGORITHM: CLOSEST_PAIR(P)

```
if (|P| < 2) then return (P, ∞)
x_m := MEDIAN({x : (x, y) ∈ P})
L := {(x, y) ∈ P | x < x_m}
R := {(x, y) ∈ P | x ≥ x_m}
in parallel do
   (L', δ_L) := CLOSEST_PAIR(L)
   (R', δ_R) := CLOSEST_PAIR(R)
P' := MERGE_BY_Y(L', R')
δ_P := BOUNDARY_MERGE(P', δ_L, δ_R, x_m)
return (P', δ_P)
```

This function takes a set of points P in the plane and returns both the original points sorted along the y axis, and the distance between the closest two points. The sorted points are needed to help merge the results from recursive calls, and can be thrown away at the end. It would not be difficult to modify the routine to return the closest pair of points in addition to the distance between them. The function works by dividing the points in half based on the median x value, recursively solving the problem on each half and then merging the results. The MERGE_BY_Y function merges L' and R' along the y axis and can use a standard parallel merge routine. The interesting aspect of the code is the BOUNDARY_MERGE routine, which works on the principle described by Bentley and Shamos [17, 16], and can be computed with $O(\log n)$ depth and $O(n)$ work. We first review the principle and then show how it can be performed in parallel.

The inputs to BOUNDARY_MERGE are the original points P sorted along the y axis, the closest distance within L and R, and the median point x_m. The closest distance in P must be either the distance δ_L, the distance δ_R, or a distance between a point in L and a point in R. For this distance to be less than δ_L or δ_R, the two points must lie within $\delta = \min(\delta_L, \delta_R)$ of the line $x = x_m$. Thus, the two vertical lines at $x_l = x_m - \delta$ and $x_r = x_m + \delta$ define the borders of a region M in which the points must lie (see Figure 13). If we could find the closest distance in M, call it δ_M, then the closest overall distance would be $\delta_P = \min(\delta_L, \delta_R, \delta_M)$.

To find δ_M we take advantage of the fact that not many points can be packed close together within M since all points within L or R must be separated by at least δ. Figure 13 shows the tightest possible packing of points in a $2\delta \times \delta$ rectangle within M. This packing implies that if the points in M are sorted along the y axis, each point can determine the minimum distance to

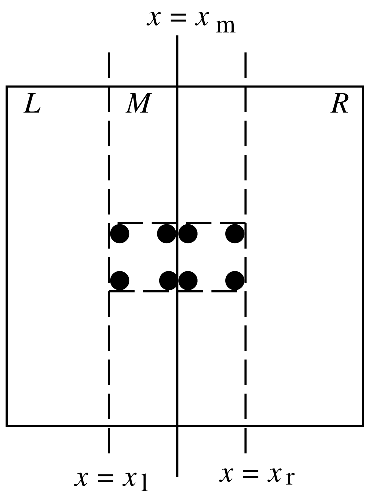


Figure 13: Merging two rectangles to determine the closest pair. Only 8 points can fit in the $2\delta \times \delta$ dashed rectangle.

another point in M by looking at a fixed number of neighbors in the sorted order, at most 7 in each direction. To see this consider one of the points along the top of the $2\delta \times \delta$ rectangle. To determine if there are any points below it that are closer than δ we need only to consider the points within the rectangle (points below the rectangle must be further than δ away). As the figure illustrates, there can be at most 7 other points within the rectangle. Given this property, the following function performs the border merge.

ALGORITHM: BOUNDARY_MERGE$(P, \delta_L, \delta_R, x_m)$

$\delta := \min(\delta_L, \delta_R)$
$M := \{(x, y) \in P \mid (x \geq x_m - \delta)$ `and` $(x \leq x_m + \delta)\}$
$\delta_M := \min(\{\min(\{distance(M[i], M[i+j]) : j \in [1..7]\})$
$\quad : i \in [0..|P| - 7)\}$
return $\min(\delta, \delta_M)$

For each point in M this function considers the seven points following it in the sorted order and determines the distance to each of these points. It then takes the minimum over all distances. Since the distance relationship is symmetric, there is no need to consider points appearing before a point in the sorted order.

The work of BOUNDARY_MERGE is $O(n)$ and the depth is dominated by the taking the minimum, which has $O(\log n)$ depth.[2] The work of the merge and median steps in CLOSEST_PAIR are also $O(n)$, and the depth of both are bounded by $O(\log n)$. The total work and depth of the algorithm

[2]The depth of finding the minimum or maximum of a set of numbers can actually be improved to $O(\log \log n)$ with concurrent reads [78].

can therefore be expressed by the recurrences

$$W(n) = 2W(n/2) + O(n) \tag{11}$$

$$D(n) = D(n/2) + O(\log n) \tag{12}$$

which have solutions $W(n) = O(n \log n)$ and $D(n) = O(\log^2 n)$.

6.2 Planar convex hull

The convex hull problem takes a set of points in k dimensions and returns the smallest convex region that contains all the points. In two dimensions the problem is called the planar convex hull problem and it returns the set of points that form the corners of the region. These points are a subset of the original points. We will describe two parallel algorithms for the planar convex hull problem. They are both based on divide-and-conquer, but one does most of the work before the divide step, and the other does most of the work after.

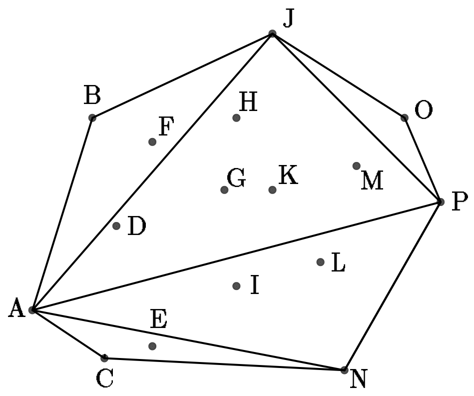
6.2.1 QuickHull

The parallel *QuickHull* algorithm is based on the sequential version [71], so named because of its similarity to the QuickSort algorithm. As with QuickSort, the strategy is to pick a "pivot" element, split the data based on the pivot, and recurse on each of the split sets. Also as with QuickSort, the pivot element is not guaranteed to split the data into equal-sized sets, and in the worst case the algorithm requires $O(n^2)$ work. In practice, however, the algorithm is often very efficient, probably the most practical of the convex hull algorithms. At the end of the section we briefly describe how the splits can be made so that the work is guaranteed to be bounded by $O(n \log n)$.

The QuickHull algorithm is based on the recursive function SUBHULL which is expressed as follows.

ALGORITHM: SUBHULL(P, p_1, p_2)

$P' := \{p \in P \mid \text{LEFT\_OF?}(p, (p_1, p_2))\}$
if $(|P'| < 2)$
then return $[p_1]$ ++ P'
else
    $i := \text{MAX\_INDEX}(\{\text{DISTANCE}(p, (p_1, p_2)) : p \in P'\})$
    $p_m := P'[i]$
    in parallel do
        $H_l := \text{SUBHULL}(P', p_1, p_m)$
        $H_r := \text{SUBHULL}(P', p_m, p_2)$
    return H_l ++ H_r



[A B C D E F G H I J K L M N O P]

A [B D F G H J K M O] P [C E I L N]

A [B F] J [O] P N [C E]

A B J O P N C

Figure 14: An example of the *QuickHull* algorithm.

This function takes a set of points P in the plane and two points p_1 and p_2 that are known to lie on the convex hull, and returns all the points that lie on the hull clockwise from p_1 to p_2, inclusive of p_1, but not of p_2. For example in Figure 14 SUBHULL$([A, B, C, ..., P], A, P)$ would return the sequence $[A, B, J, O]$.

The function SUBHULL works as follows. Line 1 removes all the elements that cannot be on the hull because they lie to the right of the line from p_1 to p_2. Determining which side of a line a point lies on can easily be calculated with a few arithmetic operations. If the remaining set P' is either empty or has just one element, the algorithm is done. Otherwise the algorithm finds the point p_m farthest from the line (p_1, p_2). The point p_m must be on the hull since as a line at infinity parallel to (p_1, p_2) moves toward (p_1, p_2), it must first hit p_m. In Line 5 the function MAX_INDEX returns the index of the maximum value of a sequence, which is then used to extract the point p_m. Once p_m is found, SUBHULL is called twice recursively to find the hulls from p_1 to p_m, and from p_m to p_2. When the recursive calls return, the results are appended.

The following function uses SUBHULL to find the full convex hull.

ALGORITHM: QUICKHULL(P)

$X := \{x : (x, y) \in P\}$

$x_{\min} := P[\mathit{min\_index}(X)]$

$x_{\max} := P[\mathit{max\_index}(X)]$

return SUBHULL$(P, x_{\min}, x_{\max})$ ++ SUBHULL$(P, x_{\max}, x_{\min})$

We now consider the cost of the parallel QuickHull, and in particular the SUBHULL routine, which does all the work. The call to MAX_INDEX uses $O(n)$ work and $O(\log n)$ depth. Hence, the cost of everything other than the recursive calls is $O(n)$ work and $O(\log n)$ depth. If the recursive calls are balanced so that neither recursive call gets much more than half of the data then the number of levels of recursion will be $O(\log n)$. This will lead to the algorithm running in $O(\log^2 n)$ depth. Since the sum of the sizes of the recursive calls can be less than n (*e.g.*, the points within the triangle AJP will be thrown out when making the recursive calls to find the hulls between A and J and between J and P), the work can be as little as $O(n)$, and often is in practice. As with QuickSort, however, when the recursive calls are badly partitioned the number of levels of recursion can be as bad as $O(n)$ with work $O(n^2)$. For example, consider the case when all the points lie on a circle and have the following unlikely distribution. x_{min} and x_{max} appear on opposite sides of the circle. There is one point that appears half way between x_{min} and x_{max} on the sphere and this point becomes the new x_{max}. The remaining points are defined recursively. That is, the points become arbitrarily close to x_{min} (see Figure 15).

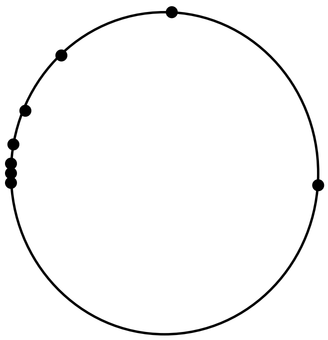


Figure 15: Contrived set of points for worst case QuickHull.

Kirkpatrick and Seidel [49] have shown that it is possible to modify QuickHull so that it makes provably good partitions. Although the technique is shown for a sequential algorithm, it is easy to parallelize. A simplification of the technique is given by Chan et al. [25]. Their algorithm admits even more parallelism and leads to an $O(\log^2 n)$-depth algorithm with $O(n \log h)$ work where h is the number of points on the convex hull.

6.2.2 MergeHull

The MergeHull algorithm [68] is another divide-and-conquer algorithm for solving the planar convex hull problem. Unlike QuickHull, however, it does most of its work after returning from the recursive calls. The function is expressed as follows.

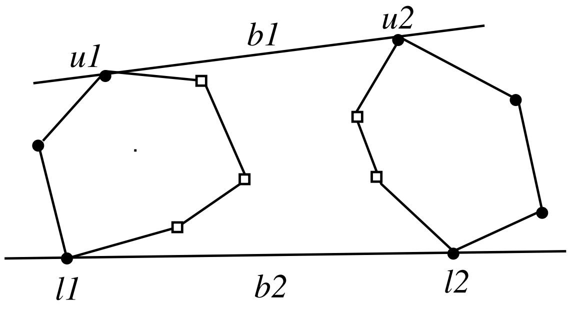


Figure 16: Merging two convex hulls.

ALGORITHM: MERGEHULL(P)

if $(|P| < 3)$ **then return** P
else
  in parallel do
    $H_L = \text{MERGEHULL}(P[0..|P|/2))$
    $H_R = \text{MERGEHULL}(P[|P|/2..|P|))$
  return JOIN_HULLS(H_L, H_R)

This function assumes the input P is presorted according to the x coordinates of the points. Since the points are presorted, H_L is a convex hull on the left and H_R is a convex hull on the right. The JOIN_HULLS routine is the interesting part of the algorithm. It takes the two hulls and merges them into one. To do this it needs to find lower and upper points l_1 and u_1 on H_L and l_2 and u_2 on H_R such that l_1, l_2 and u_1, u_2 are successive points on H (see Figure 16). The lines b_1 and b_2 joining these upper and lower points are called the upper and lower bridges, respectively. All the points between l_1 and u_1 and between u_2 and l_2 on the "outer" sides of H_L and H_R are on the final convex hull, while the points on the "inner" sides are not on the convex hull. Without loss of generality we only consider how to find the upper bridge b_1. Finding the lower bridge b_2 is analogous.

To find the upper bridge one might consider taking the points with the maximum y values on H_L and H_R. This approach does not work in general, however, since u_1 can lie as far down as the point with the minimum x or maximum x value (see Figure 17). Instead there is a nice solution due to Overmars [68] based on a dual binary search. Assume that the points on the convex hulls are given in order (e.g., clockwise). At each step the binary search algorithm will eliminate half of the remaining points from consideration in either H_L or H_R or both. After at most $\log|H_L| + \log|H_R|$ steps the search will be left with only one point in each hull, and these will be the desired points u_1 and u_2. Figure 18 illustrates the rules for eliminating part of H_L or H_R on each step.

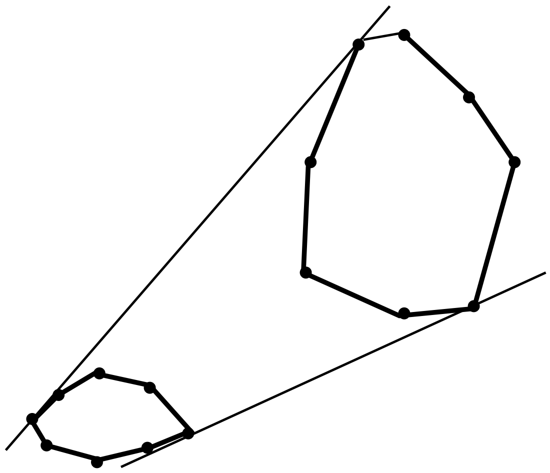


Figure 17: A bridge that is far from the top of the convex hull.

We now consider the cost of the algorithm. Each step of the binary search requires only constant work and depth since we need only to consider the two middle points M_1 and M_2, which can be found in constant time if the hull is kept sorted. The cost of the full binary search to find the upper bridge is therefore bounded by $D(n) = W(n) = O(\log n)$. Once we have found the upper and lower bridges we need to remove the points on H_L and H_R that are not on H and append the remaining convex hull points. This requires linear work and constant depth. The overall costs of MergeHull are therefore:

$$D(n) = D(n/2) + O(\log n) = O(\log^2 n) \tag{13}$$

$$W(n) = 2W(n/2) + O(n) = O(n \log n). \tag{14}$$

This algorithm can be improved to run in $O(\log n)$ depth using one of two techniques. The first involves modifying the search for the bridge points so that it runs in constant depth with linear work [12]. This involves sampling every $\sqrt{n}$th point on each hull and comparing all pairs of these two samples to narrow the search down to regions of size $\sqrt{n}$ in constant depth. The regions can then be finished in constant depth by comparing all pairs between the two regions. The second technique [1, 11] uses divide-and-conquer to separate the point set into $\sqrt{n}$ regions, solves the convex hull on each region recursively and then merges all pairs of these regions using the binary search method. Since there are $\sqrt{n}$ regions and each of the searches takes $O(\log n)$ work, the total work for merging is $O((\sqrt{n})^2 \log n) = O(n \log n)$ and the depth is $O(\log n)$. This leads to an overall algorithm that runs in $O(n \log n)$ work and $O(\log n)$ depth. The algorithms above require concurrent reads (CREW). The same bounds can be achieved with exclusive reads (EREW) [66].

7 Numerical algorithms

There has been an immense amount of work on parallel algorithms for numerical problems. Here we briefly discuss some of the problems and results. We suggest the following sources for further information on parallel numerical algorithms [75, Chapters 12-14], [45, Chapter 8], [53, Chapters 5, 10 and 11] and [18].

7.1 Matrix operations

Matrix operations form the core of many numerical algorithms and led to some of the earliest work on parallel algorithms. The most basic matrix operation is matrix multiplication. The standard triply nested loop for multiplying two dense matrices is highly parallel since each of the loops can be parallelized:

ALGORITHM: MATRIX_MULTIPLY(A, B)

$(l, m) := \mathit{dimensions}(A)$
$(m, n) := \mathit{dimensions}(B)$
in parallel for $i \in [0..l)$ **do**
  in parallel for $j \in [0..n)$ **do**
    $R_{ij} := \mathit{sum}(\{A_{ik} * B_{kj} : k \in [0..m)\})$
return R

If $l = m = n$, this routine does $O(n^3)$ work and has depth $O(\log n)$, due to the depth of the summation. This has much more parallelism than is typically needed, and most of the research on parallel matrix multiplication has concentrated on what subset of the parallelism to use so that communication costs can be minimized. Sequentially it is known that matrix multiplication can be performed using less than $O(n^3)$ work. Strassen's algorithm [82], for example, requires only $O(n^{2.81})$ work. Most of these more efficient algorithms are also easy to parallelize because they are recursive in nature (Strassen's algorithm has $O(\log n)$ depth using a simple parallelization).

Another basic matrix operation is to invert matrices. Inverting dense matrices has proven to be more difficult to parallelize than multiplying dense matrices, but the problem still supplies plenty of parallelism for most practical purposes. When using Gauss-Jordan elimination, two of the three nested loops can be parallelized, leading to an algorithm that runs with $O(n^3)$ work and $O(n)$ depth. A recursive block-based method using matrix multiplication leads to the same depth, although the work can be reduced by using a more efficient matrix multiplication algorithm. There are also more sophisticated, but less practical, work-efficient algorithms with depth $O(\log^2 n)$ [28, 70].

Parallel algorithms for many other matrix operations have been studied, and there has also been significant work on algorithms for various special forms of matrices, such as tridiagonal, triangular, and sparse matrices. Iterative methods for solving sparse linear systems has been an area of significant activity.

7.2 Fourier transform

Another problem for which there is a long history of parallel algorithms is the Discrete Fourier Transform (DFT). The Fast Fourier Transform (FFT) algorithm for solving the DFT is quite easy to parallelize and, as with matrix multiply, much of the research has gone into reducing communication costs. In fact the butterfly network topology is sometimes called the FFT network since the FFT has the same communication pattern as the network [55, Section 3.7]. A parallel FFT over complex numbers can be expressed as follows

ALGORITHM: FFT(A)

$n := |A|$
if $(n = 1)$ **then return** A
else
  in parallel do
    $\textsc{even} := \mathrm{FFT}(\{A[2i] : i \in [0..n/2)\})$
    $\textsc{odd} := \mathrm{FFT}(\{A[2i+1] : i \in [0..n/2)\})$
  return $\{\textsc{even}[j] + \textsc{odd}[j]e^{2\pi ij/n} : j \in [0..n/2)\}$++$\{\textsc{even}[j] - \textsc{odd}[j]e^{2\pi ij/n} : j \in [0..n/2)\}$

It simply calls itself recursively on the odd and even elements and then puts the results together. This algorithm does $O(n \log n)$ work, as does the sequential version, and has a depth of $O(\log n)$.

8 Research issues and summary

Recent work on parallel algorithms has focused on solving problems from domains such as pattern matching, data structures, sorting, computational geometry, combinatorial optimization, linear algebra, and linear and integer programming. For pointers to this work, see Section 10.

Algorithms have also been designed specifically for the types of parallel computers that are available today. Particular attention has been paid to machines with limited communication bandwidth. For example, there is a growing library of software developed for the BSP model [40, 62, 85].

The parallel computer industry has been through a period of financial turbulence, with several manufacturers failing or discontinuing sales of parallel machines. In the past few years, however, a large number of inexpensive small-scale parallel machines have been sold. These machines typically consist of 4 to 8 commodity processors connected by a bus to a shared-memory system. As these machines reach the limit in size imposed by the bus architecture, manufacturers have reintroduced parallel machines based on the hypercube network topology (e.g., [54]).

9 Defining terms

CRCW. A shared memory model that allows for concurrent reads (CR) and concurrent writes (CW) to the memory.

CREW. This refers to a shared memory model that allows for Concurrent reads (CR) but only exclusive writes (EW) to the memory.

Depth. The longest chain of sequential dependences in a computation.

EREW A shared memory model that allows for only exclusive reads (ER) and exclusive writes (EW) to the memory.

Graph Contraction. Contracting a graph by removing a subset of the vertices.

List Contraction. Contracting a list by removing a subset of the nodes.

Multiprefix. A generalization of the scan (prefix sums) operation in which the partial sums are grouped by keys.

Multiprocessor Model. A model of parallel computation based on a set of communicating sequential processors.

Pipelined Divide-and-Conquer. A divide-and-conquer paradigm in which partial results from recursive calls can be used before the calls complete. The technique is often useful for reducing the depth of an algorithm.

Pointer Jumping. In a linked structure replacing a pointer with the pointer it points to. Used for various algorithms on lists and trees.

PRAM model. A multiprocessor model in which all processors can access a shared memory for reading or writing with uniform cost.

Prefix Sums. A parallel operation in which each element in an array or linked-list receives the sum of all the previous elements.

Random Sampling. Using a randomly selected sample of the data to help solve a problem on the whole data.

Recursive Doubling. The same as pointer jumping.

Scan. A parallel operation in which each element in an array receives the sum of all the previous elements.

Shortcutting. Same as pointer jumping.

Tree Contraction. Contracting a tree by removing a subset of the nodes.

Symmetry Breaking. A technique to break the symmetry in a structure such as a graph which can locally look the same to all the vertices. Usually implemented with randomization.

Work. The total number of operations taken by a computation.

Work-Depth Model. A model of parallel computation in which one keeps track of the total work and depth of a computation without worrying about how it maps onto a machine.

Work-Efficient. A parallel algorithm is work-efficient if asymptotically (as the problem size grows) it requires at most a constant factor more work than the best know sequential algorithm (or the optimal work).

Work-Preserving. A translation of an algorithm from one model to another is work-preserving if the work is the same in both models, to within a constant factor.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. O. Dúnlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(3):293–327, 1988.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[3] S. G. Akl. *Parallel Sorting Algorithms.* Academic Press, Orlando, FL, 1985.

[4] S. G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[5] S. G. Akl. *Parallel Computation: Models and Methods.* Prentice Hall, Englewood Cliffs, NJ, 1997.

[6] S. G. Akl and K. A. Lyons. *Parallel Computational Geometry.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[7] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings, Redwood City, CA, 1989.

[8] R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33(5):269–273, Jan. 1990.

[9] R. J. Anderson and G. L. Miller. Deterministic parallel list ranking. *Algorithmica*, 6(6):859–868, 1991.

[10] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM Journal of Computing*, 18(3):499–532, June 1989.

[11] M. J. Atallah and M. T. Goodrich. Efficient parallel solutions to some geometric problems. *Journal of Parallel and Distributed Computing*, 3(4):492–507, Dec. 1986.

[12] M. J. Atallah and M. T. Goodrich. Parallel algorithms for some functions of two convex polygons. *Algorithmica*, 3(4):535–548, 1988.

[13] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, C–36(10):1258–1263, Oct. 1987.

[14] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. *Mathematical Systems Theory*, 27(5):341–452, September/October 1994.

[15] V. E. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic.* Academic Press, New York, NY, 1965.

[16] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the Association for Computing Machinery*, 23(4):214–229, Apr. 1980.

[17] J. L. Bentley and M. I. Shamos. Divide-and-conquer in multidimensional space. In *Conference Record of the Eighth Annual ACM Symposium on Theory of Computing*, pages 220–230, May 1976.

[18] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[19] G. E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, Cambridge, MA, 1990.

[20] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, Mar. 1996.

[21] G. E. Blelloch, K. M. Chandy, and S. Jagannathan, editors. *Specification of Parallel Algorithms. Volume 18 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Providence, RI, 1994.

[22] G. E. Blelloch and J. Greiner. Parallelism in sequential functional languages. In *Proceedings of the Symposium on Functional Programming and Computer Architecture*, pages 226–237, June 1995.

[23] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31(2):135–167, March/April 1998.

[24] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the Association for Computing Machinery*, 21(2):201–206, Apr. 1974.

[25] T. M. Chan, J. Snoeyink, and C.-K. Yap. Output-sensitive construction of polytopes in four dimensions and clipped Voronoi diagrams in three. In *Proceedings of the 6th Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 282–291, Jan. 1995.

[26] R. Cole. Parallel merge sort. *SIAM Journal of Computing*, 17(4):770–785, Aug. 1988.

[27] R. Cole, P. N. Klein, and R. E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 243–250, June 1996.

[28] L. Csanky. Fast parallel matrix inversion algorithms. *SIAM Journal on Computing*, 5(4):618–623, Apr. 1976.

[29] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the Association for Computing Machinery*, 39(11):78–85, Nov. 1996.

[30] R. Cypher and J. L. C. Sanz. *The SIMD Model of Parallel Computation.* Springer-Verlag, New York, NY, 1994.

[31] D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Annual Review of Computer Science*, 3:233–83, 1988.

[32] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, May 1978.

[33] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM Journal on Computing*, 20(6):1046–1067, Dec. 1991.

[34] D. Gelernter, A. Nicolau, and D. Padua, editors. *Languages and Compilers for Parallel Computing. Research Monographs in Parallel and Distributed.* MIT Press, Cambridge, MA, 1990.

[35] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, Cambridge, England, 1988.

[36] P. B. Gibbons, Y. Matias, and V. Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. In *Proceedings of the 5th Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 638–648, Jan. 1994.

[37] L. M. Goldschlager. A unified approach to models of synchronous parallel machines. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 89–94, May 1978.

[38] L. M. Goldschlager. A universal interconnection pattern for parallel computers. *Journal of the Association for Computing Machinery*, 29(3):1073–1086, Oct. 1982.

[39] M. T. Goodrich. Parallel algorithms in geometry. In J. E. Goodman and J. O'Rourke, editors, *CRC Handbook of Discrete and Computational Geometry*, pages 669–682. CRC Press, 1997.

[40] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Towards efficiency and portability: Programming with the BSP model. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, June 1996.

[41] J. Greiner and G. E. Blelloch. Connected components algorithms. In G. W. Sabot, editor, *High Performance Computing: Problem Solving with Parallel and Vector Architectures.* Addison Wesley, Reading, MA, 1995.

[42] S. Halperin and U. Zwick. An optimal randomised logarithmic time connectivity algorithm for the EREW PRAM. *Journal of Computer and Systems Sciences*, 53(3):395–416, Dec. 1996.

[43] T. J. Harris. A survey of PRAM simulation techniques. *ACM Computing Surveys*, 26(2):187–206, June 1994.

[44] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Francisco, CA, second edition, 1996.

[45] J. JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley, Reading, MA, 1992.

[46] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the Association of Computing Machinery*, 42(2):321–328, Mar. 1995.

[47] A. R. Karlin and E. Upfal. Parallel hashing: an efficient implementation of shared memory. *Journal of the Association for Computing Machinery*, 35(5):876–892, Oct. 1988.

[48] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 869–941. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.

[49] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(1):287–299, Feb. 1986.

[50] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley Publishing Company, Reading, MA, 1973.

[51] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C–22(8):786–793, Aug. 1973.

[52] C. P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Trans. Comput.*, C–32(10):942–946, Oct. 1983.

[53] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings, Redwood City, CA, 1994.

[54] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

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

[56] C. E. Leiserson. Fat-Trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C–34(10):892–901, Oct. 1985.

[57] T. Lengauer. VLSI theory. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 837–868. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.

[58] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, Nov. 1986.

[59] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, San Francisco, CA, 1996.

[60] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (EDS) and st-numbering in graphs. *Theoretical Comput. Sci.*, 47:277–298, 1986.

[61] Y. Matias and U. Vishkin. On parallel hashing and integer sorting. *Journal of Algorithms*, 12(4):573–606, Dec. 1991.

[62] W. F. McColl. BSP programming. In G. E. Blelloch, K. M. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms. Volume 18 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 25–35. American Mathematical Society, Providence, RI, May 1994.

[63] G. L. Miller and V. Ramachandran. A new graph triconnectivity algorithm and its parallelization. *Combinatorica*, 12(1):53–76, 1992.

[64] G. L. Miller and J. H. Reif. Parallel tree contraction part 1: Fundamentals. In S. Micali, editor, *Randomness and Computation.* Volume 5 of *Advances in Computing Research*, pages 47–72. JAI Press, Greenwich, CT, 1989.

[65] G. L. Miller and J. H. Reif. Parallel tree contraction part 2: Further applications. *SIAM Journal of Computing*, 20(6):1128–1147, Dec. 1991.

[66] R. Miller and Q. F. Stout. Efficient parallel convex hull algorithms. *IEEE Transactions on Computers*, 37(12):1605–1619, Dec. 1988.

[67] R. Miller and Q. F. Stout. *Parallel Algorithms for Regular Architectures.* MIT Press, Cambridge, MA, 1996.

[68] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, Oct. 1981.

[69] V. R. Pratt and L. J. Stockmeyer. A characterization of the power of vector machines. *Journal of Computer and System Sciences*, 12(2):198–221, Apr. 1976.

[70] F. Preparata and D. Sarwate. An improved parallel processor bound in fast matrix inversion. *Information Processing Letters*, 7(3):148–150, Apr. 1978.

[71] F. P. Preparata and M. I. Shamos. *Computational Geometry — an Introduction.* Springer-Verlag, New York, NY, 1985.

[72] A. G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42(3):307–326, June 1991.

[73] S. Ranka and S. Sahni. *Hypercube algorithms: with applications to image processing and pattern recognition.* Springer-Verlag, New York, NY, 1990.

[74] M. Reid-Miller. List ranking and list scan on the Cray C90. *Journal of Computer and Systems Sciences*, 53(3):344–356, Dec. 1996.

[75] J. H. Reif, editor. *Synthesis of Parallel Algorithms.* Morgan Kaufmann, San Mateo, CA, 1993.

[76] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the Association for Computing Machinery*, 34(1):60–76, Jan. 1987.

[77] W. J. Savitch and M. Stimson. Time bounded random access machines with parallel processing. *Journal of the Association for Computing Machinery*, 26(1):103–118, Jan. 1979.

[78] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88–102, Mar. 1981.

[79] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, Mar. 1982.

[80] H. J. Siegel. *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies.* McGraw–Hill, New York, NY, second edition, 1990.

[81] H. S. Stone. Parallel tridiagonal equation solvers. *ACM Transactions on Mathematical Software*, 1(4):289–307, Dec. 1975.

[82] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[83] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing*, 14(4):862–874, Nov. 1985.

[84] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Rockville, MD, 1984.

[85] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

[86] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 943–971. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.

[87] U. Vishkin. Parallel-design distributed-implementation (PDDI) general purpose computer. *Theoretical Computer Science*, 32(1–2):157–172, July 1984.

[88] J. C. Wyllie. The complexity of parallel computations. Technical Report TR-79-387, Department of Computer Science, Cornell University, Ithaca, NY, Aug. 1979.

[89] M. C. K. Yang, J. S. Huang, and Y. Chow. Optimal parallel sorting scheme by order statistics. *SIAM Journal on Computing*, 16(6):990–1003, Dec. 1987.

10 Further information

In a chapter of this length, it is not possible to provide comprehensive coverage of the subject of parallel algorithms. Fortunately, there are several excellent textbooks and surveys on parallel algorithms including [4, 6, 5, 18, 30, 31, 35, 45, 48, 53, 55, 67, 75, 80].

There are many technical conferences devoted to the subjects of parallel computing and computer algorithms, so keeping abreast of the latest research developments is challenging. Some of the best work in parallel algorithms can be found in conferences such as the *ACM Symposium on Parallel Algorithms and Architectures*, the *IEEE International Parallel Processing Symposium*, the *IEEE Symposium on Parallel and Distributed Processing*, the *International Conference on Parallel Processing*, the *International Symposium on Parallel Architectures, Algorithms, and Networks*, the *ACM Symposium on the Theory of Computing*, the *IEEE Symposium on Foundations of Computer Science*, the *ACM–SIAM Symposium on Discrete Algorithms*, and the *ACM Symposium on Computational Geometry*.

In addition to parallel algorithms, this chapter has also touched on several related subjects, including the modeling of parallel computations, parallel computer architecture, and parallel programming languages. More information on these subjects can be found in [43, 86], [7, 44], and [21, 34], respectively. Other topics likely to interest the reader of this chapter include distributed algorithms [59] and VLSI layout theory and computation [57, 84].

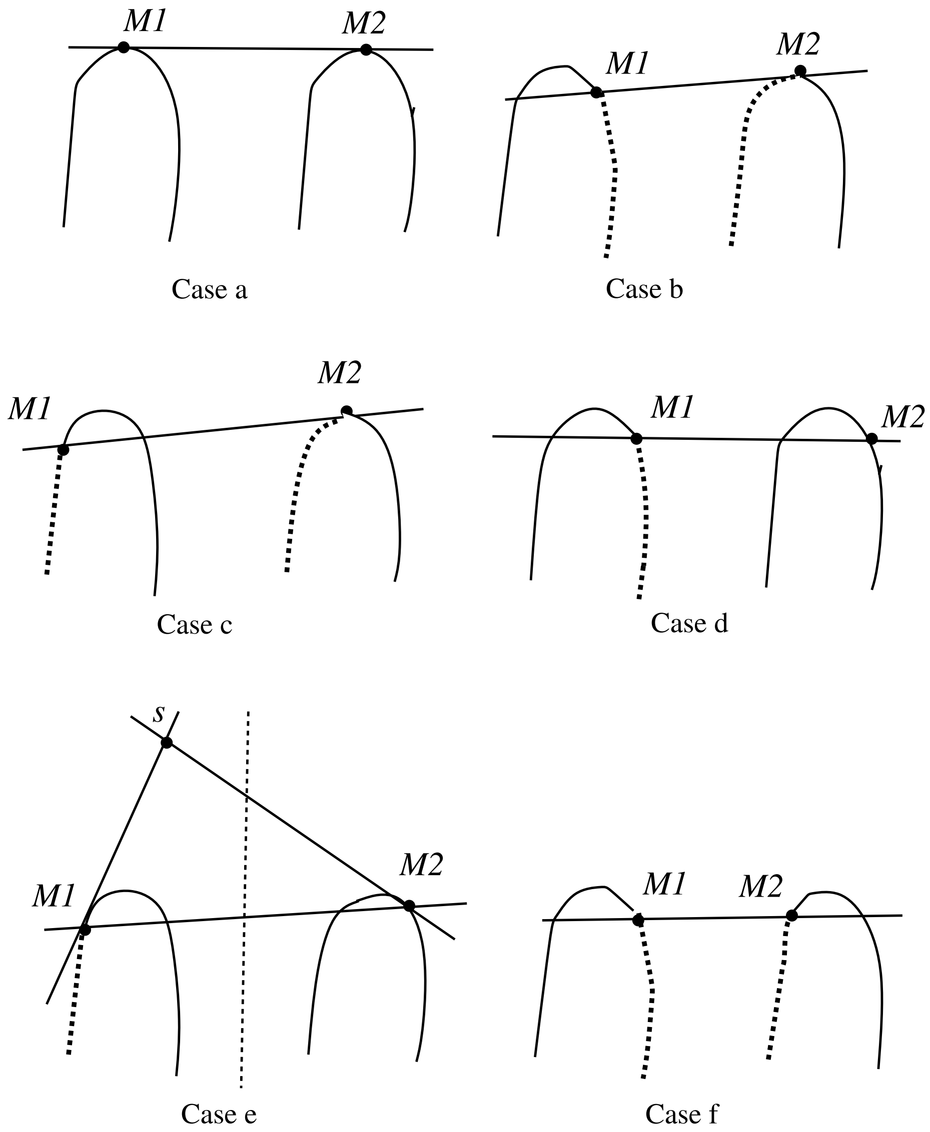


Figure 18: Cases used in the binary search for finding the upper bridge for the MergeHull. The points $M1$ and $M2$ mark the middle of the remaining hulls. In case (a), all of H_L and H_R lie below the line through $M1$ and $M2$. In this case, the line segment between $M1$ and $M2$ is the bridge. In the remaining cases, dotted lines represent the parts of the hulls H_L and H_R that can be eliminated from consideration. In cases (b) and (c), all of H_R lies below the line through $M1$ and $M2$, and either the left half of H_L or the right half of H_L lies below the line. In cases (d) through (f), neither H_L nor H_R lies entirely below the line. In the case (e), the region to eliminate depends on which side of a line separating H_L and H_R the intersection of the tangents appears. The mirror images of cases (b) through (e) are also used. Case (f) is actually an instance of case (d) and its mirror image.