26 Parallel Algorithms

The vast majority of algorithms in this book are ***serial algorithms*** suitable for running on a uniprocessor computer that executes only one instruction at a time. This chapter extends our algorithmic model to encompass ***parallel algorithms***, where multiple instructions can execute simultaneously. Specifically, we'll explore the elegant model of task-parallel algorithms, which are amenable to algorithmic design and analysis. Our study focuses on fork-join parallel algorithms, the most basic and best understood kind of task-parallel algorithm. Fork-join parallel algorithms can be expressed cleanly using simple linguistic extensions to ordinary serial code. Moreover, they can be implemented efficiently in practice.

Parallel computers—computers with multiple processing units—are ubiquitous. Handheld, laptop, desktop, and cloud machines are all ***multicore computers***, or simply, ***multicores***, containing multiple processing "cores." Each processing core is a full-fledged processor that can directly access any location in a common ***shared memory***. Multicores can be aggregated into larger systems, such as clusters, by using a network to interconnect them. These multicore clusters usually have a ***distributed memory***, where one multicore's memory cannot be accessed directly by a processor in another multicore. Instead, the processor must explicitly send a message over the cluster network to a processor in the remote multicore to request any data it requires. The most powerful clusters are supercomputers, comprising many thousands of multicores. But since shared-memory programming tends to be conceptually easier than distributed-memory programming, and multicore machines are widely available, this chapter focuses on parallel algorithms for multicores.

One approach to programming multicores is ***thread parallelism***. This processor-centric parallel-programming model employs a software abstraction of "virtual processors," or ***threads*** that share a common memory. Each thread maintains its own program counter and can execute code independently of the other threads. The operating system loads a thread onto a processing core for execution and switches it out when another thread needs to run.

Unfortunately, programming a shared-memory parallel computer using threads tends to be difficult and error-prone. One reason is that it can be complicated to dynamically partition the work among the threads so that each thread receives approximately the same load. For any but the simplest of applications, the programmer must use complex communication protocols to implement a scheduler that load-balances the work.

Task-parallel programming

The difficulty of thread programming has led to the creation of ***task-parallel platforms***, which provide a layer of software on top of threads to coordinate, schedule, and manage the processors of a multicore. Some task-parallel platforms are built as runtime libraries, but others provide full-fledged parallel languages with compiler and runtime support.

Task-parallel programming allows parallelism to be specified in a "processor-oblivious" fashion, where the programmer identifies what computational tasks may run in parallel but does not indicate which thread or processor performs the task. Thus, the programmer is freed from worrying about communication protocols, load balancing, and other vagaries of thread programming. The task-parallel platform contains a scheduler, which automatically load-balances the tasks across the processors, thereby greatly simplifying the programmer's chore. ***Task-parallel algorithms*** provide a natural extension to ordinary serial algorithms, allowing performance to be reasoned about mathematically using "work/span analysis."

Fork-join parallelism

Although the functionality of task-parallel environments is still evolving and increasing, almost all support ***fork-join parallelism***, which is typically embodied in two linguistic features: ***spawning*** and ***parallel loops***. Spawning allows a subroutine to be "forked": executed like a subroutine call, except that the caller can continue to execute while the spawned subroutine computes its result. A parallel loop is like an ordinary **for** loop, except that multiple iterations of the loop can execute at the same time.

Fork-join parallel algorithms employ spawning and parallel loops to describe parallelism. A key aspect of this parallel model, inherited from the task-parallel model but different from the thread model, is that the programmer does not specify which tasks in a computation *must* run in parallel, only which tasks *may* run in parallel. The underlying runtime system uses threads to load-balance the tasks across the processors. This chapter investigates parallel algorithms described in the fork-join model, as well as how the underlying runtime system can schedule task-parallel computations (which include fork-join computations) efficiently.

Fork-join parallelism offers several important advantages:

- The fork-join programming model is a simple extension of the familiar serial programming model used in most of this book. To describe a fork-join parallel algorithm, the pseudocode in this book needs just three added keywords: **parallel**, **spawn**, and **sync**. Deleting these parallel keywords from the parallel pseudocode results in ordinary serial pseudocode for the same problem, which we call the "serial projection" of the parallel algorithm.
- The underlying task-parallel model provides a theoretically clean way to quantify parallelism based on the notions of "work" and "span."
- Spawning allows many divide-and-conquer algorithms to be parallelized naturally. Moreover, just as serial divide-and-conquer algorithms lend themselves to analysis using recurrences, so do parallel algorithms in the fork-join model.
- The fork-join programming model is faithful to how multicore programming has been evolving in practice. A growing number of multicore environments support one variant or another of fork-join parallel programming, including Cilk [290, 291, 383, 396], Habanero-Java [466], the Java Fork-Join Framework [279], OpenMP [81], Task Parallel Library [289], Threading Building Blocks [376], and X10 [82].

Section 26.1 introduces parallel pseudocode, shows how the execution of a task-parallel computation can be modeled as a directed acyclic graph, and presents the metrics of work, span, and parallelism, which you can use to analyze parallel algorithms. Section 26.2 investigates how to multiply matrices in parallel, and Section 26.3 tackles the tougher problem of designing an efficient parallel merge sort.

26.1 The basics of fork-join parallelism

Our exploration of parallel programming begins with the problem of computing Fibonacci numbers recursively in parallel. We'll look at a straightforward serial Fibonacci calculation, which, although inefficient, serves as a good illustration of how to express parallelism in pseudocode.

Recall that the Fibonacci numbers are defined by equation (3.31) on page 69:

$$F_i = \begin{cases} 0 & \text{if } i = 0\,, \\ 1 & \text{if } i = 1\,, \\ F_{i-1} + F_{i-2} & \text{if } i \geq 2\,. \end{cases}$$

To calculate the nth Fibonacci number recursively, you could use the ordinary serial algorithm in the procedure FIB on the facing page. You would not really want to

compute large Fibonacci numbers this way, because this computation does needless repeated work, but parallelizing it can be instructive.

```
FIB(n)
1  if n ≤ 1
2      return n
3  else x = FIB(n − 1)
4       y = FIB(n − 2)
5       return x + y
```

To analyze this algorithm, let $T(n)$ denote the running time of $\text{FIB}(n)$. Since $\text{FIB}(n)$ contains two recursive calls plus a constant amount of extra work, we obtain the recurrence

$$T(n) = T(n-1) + T(n-2) + \Theta(1) .$$

This recurrence has solution $T(n) = \Theta(F_n)$, which we can establish by using the substitution method (see Section 4.3). To show that $T(n) = O(F_n)$, we'll adopt the inductive hypothesis that $T(n) \le aF_n - b$, where $a > 1$ and $b > 0$ are constants. Substituting, we obtain

$$\begin{aligned} T(n) &\le (aF_{n-1} - b) + (aF_{n-2} - b) + \Theta(1) \\ &= a(F_{n-1} + F_{n-2}) - 2b + \Theta(1) \\ &\le aF_n - b , \end{aligned}$$

if we choose b large enough to dominate the upper-bound constant in the $\Theta(1)$ term. We can then choose a large enough to upper-bound the $\Theta(1)$ base case for small n. To show that $T(n) = \Omega(F_n)$, we use the inductive hypothesis $T(n) \ge aF_n - b$. Substituting and following reasoning similar to the asymptotic upper-bound argument, we establish this hypothesis by choosing b smaller than the lower-bound constant in the $\Theta(1)$ term and a small enough to lower-bound the $\Theta(1)$ base case for small n. Theorem 3.1 on page 56 then establishes that $T(n) = \Theta(F_n)$, as desired. Since $F_n = \Theta(\phi^n)$, where $\phi = (1 + \sqrt{5})/2$ is the golden ratio, by equation (3.34) on page 69, it follows that

$$T(n) = \Theta(\phi^n) . \tag{26.1}$$

Thus this procedure is a particularly slow way to compute Fibonacci numbers, since it runs in exponential time. (See Problem 31-3 on page 954 for faster ways.)

Let's see why the algorithm is inefficient. Figure 26.1 shows the tree of recursive procedure instances created when computing F_6 with the FIB procedure. The call to $\text{FIB}(6)$ recursively calls $\text{FIB}(5)$ and then $\text{FIB}(4)$. But, the call to $\text{FIB}(5)$ also

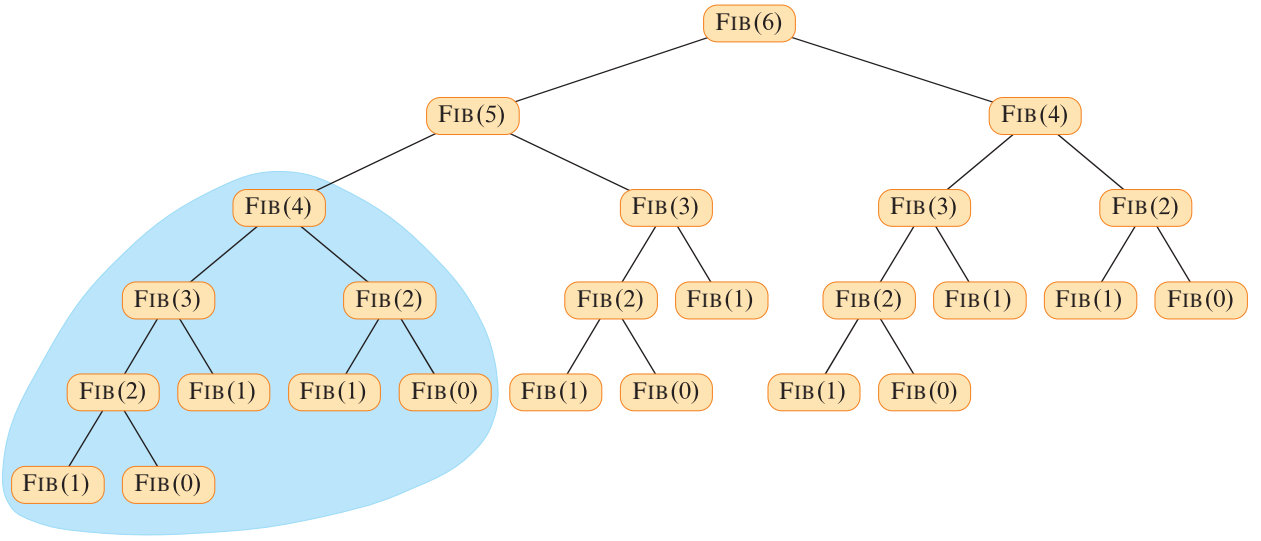


Figure 26.1 The invocation tree for FIB(6). Each node in the tree represents a procedure instance whose children are the procedure instances it calls during its execution. Since each instance of FIB with the same argument does the same work to produce the same result, the inefficiency of this algorithm for computing the Fibonacci numbers can be seen by the vast number of repeated calls to compute the same thing. The portion of the tree shaded blue appears in task-parallel form in Figure 26.2.

results in a call to FIB(4). Both instances of FIB(4) return the same result ($F_4 = 3$). Since the FIB procedure does not memoize (recall the definition of "memoize" from page 368), the second call to FIB(4) replicates the work that the first call performs, which is wasteful.

Although the FIB procedure is a poor way to compute Fibonacci numbers, it can help us warm up to parallelism concepts. Perhaps the most basic concept is to understand is that if two parallel tasks operate on entirely different data, then—absent other interference—they each produce the same outcomes when executed at the same time as when they run serially one after the other. Within FIB(n), for example, the two recursive calls in line 3 to FIB($n-1$) and in line 4 to FIB($n-2$) can safely execute in parallel because the computation performed by one in no way affects the other.

Parallel keywords

The P-FIB procedure on the next page computes Fibonacci numbers, but using the ***parallel keywords*** **spawn** and **sync** to indicate parallelism in the pseudocode.

If the keywords **spawn** and **sync** are deleted from P-FIB, the resulting pseudocode text is identical to FIB (other than renaming the procedure in the header

```
P-FIB(n)
1  if n ≤ 1
2      return n
3  else x = spawn P-FIB(n − 1)      // don't wait for subroutine to return
4       y = P-FIB(n − 2)            // in parallel with spawned subroutine
5       sync                        // wait for spawned subroutine to finish
6       return x + y
```

and in the two recursive calls). We define the ***serial projection***[1] of a parallel algorithm to be the serial algorithm that results from ignoring the parallel directives, which in this case can be done by omitting the keywords **spawn** and **sync**. For **parallel for** loops, which we'll see later on, we omit the keyword **parallel**. Indeed, our parallel pseudocode possesses the elegant property that its serial projection is always ordinary serial pseudocode to solve the same problem.

Semantics of parallel keywords

Spawning occurs when the keyword **spawn** precedes a procedure call, as in line 3 of P-FIB. The semantics of a spawn differs from an ordinary procedure call in that the procedure instance that executes the spawn—the ***parent***—may continue to execute in parallel with the spawned subroutine—its ***child***—instead of waiting for the child to finish, as would happen in a serial execution. In this case, while the spawned child is computing P-FIB$(n-1)$, the parent may go on to compute P-FIB$(n-2)$ in line 4 in parallel with the spawned child. Since the P-FIB procedure is recursive, these two subroutine calls themselves create nested parallelism, as do their children, thereby creating a potentially vast tree of subcomputations, all executing in parallel.

The keyword **spawn** does not say, however, that a procedure *must* execute in parallel with its spawned children, only that it *may*. The parallel keywords express the ***logical parallelism*** of the computation, indicating which parts of the computation may proceed in parallel. At runtime, it is up to a ***scheduler*** to determine which subcomputations actually run in parallel by assigning them to available pro-

---

[1] In mathematics, a projection is an idempotent function, that is, a function f such that $f \circ f = f$. In this case, the function f maps the set $\mathcal{P}$ of fork-join programs to the set $\mathcal{P}_S \subset \mathcal{P}$ of serial programs, which are themselves fork-join programs with no parallelism. For a fork-join program $x \in \mathcal{P}$, since we have $f(f(x)) = f(x)$, the serial projection, as we have defined it, is indeed a mathematical projection.

cessors as the computation unfolds. We'll discuss the theory behind task-parallel schedulers shortly (on page 759).

A procedure cannot safely use the values returned by its spawned children until after it executes a **sync** statement, as in line 5. The keyword **sync** indicates that the procedure must wait as necessary for all its spawned children to finish before proceeding to the statement after the **sync**—the "join" of a fork-join parallel computation. The P-FIB procedure requires a **sync** before the **return** statement in line 6 to avoid the anomaly that would occur if x and y were summed before P-FIB$(n-1)$ had finished and its return value had been assigned to x. In addition to explicit join synchronization provided by the **sync** statement, it is convenient to assume that every procedure executes a **sync** implicitly before it returns, thus ensuring that all children finish before their parent finishes.

A graph model for parallel execution

It helps to view the execution of a parallel computation—the dynamic stream of runtime instructions executed by processors under the direction of a parallel program—as a directed acyclic graph $G = (V, E)$, called a ***(parallel) trace***.[2] Conceptually, the vertices in V are executed instructions, and the edges in E represent dependencies between instructions, where $(u, v) \in E$ means that the parallel program required instruction u to execute before instruction v.

It's sometimes inconvenient, especially if we want to focus on the parallel structure of a computation, for a vertex of a trace to represent only one executed instruction. Consequently, if a chain of instructions contains no parallel or procedural control (no **spawn**, **sync**, procedure call, or **return**—via either an explicit **return** statement or the return that happens implicitly upon reaching the end of a procedure), we group the entire chain into a single ***strand***. As an example, Figure 26.2 shows the trace that results from computing P-FIB(4) in the portion of Figure 26.1 shaded blue. Strands do not include instructions that involve parallel or procedural control. These control dependencies must be represented as edges in the trace.

When a parent procedure calls a child, the trace contains an edge (u, v) from the strand u in the parent that executes the call to the first strand v of the spawned child, as illustrated in Figure 26.2 by the edge from the orange strand in P-FIB(4) to the blue strand in P-FIB(2). When the last strand v' in the child returns, the trace contains an edge (v', u') to the strand u', where u' is the successor strand of u in the parent, as with the edge from the white strand in P-FIB(2) to the white strand in P-FIB(4).

---

[2] Also called a ***computation dag*** in the literature.

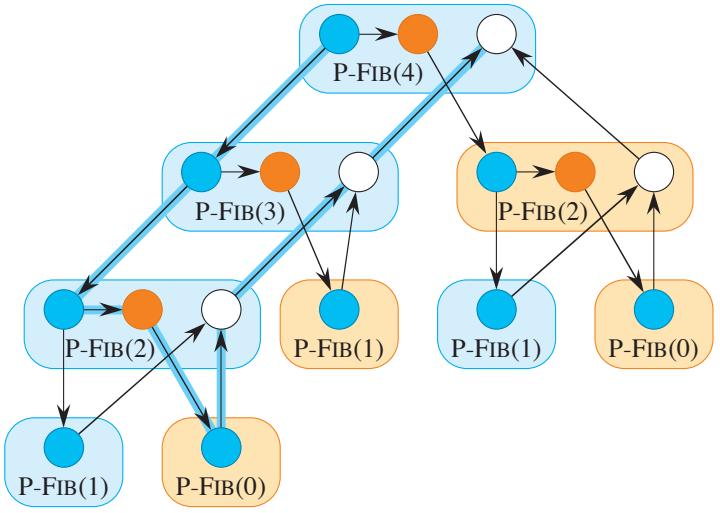


Figure 26.2 The trace of P-FIB(4) corresponding to the shaded portion of Figure 26.1. Each circle represents one strand, with blue circles representing any instructions executed in the part of the procedure (instance) up to the spawn of P-FIB($n-1$) in line 3; orange circles representing the instructions executed in the part of the procedure that calls P-FIB($n-2$) in line 4 up to the **sync** in line 5, where it suspends until the spawn of P-FIB($n-1$) returns; and white circles representing the instructions executed in the part of the procedure after the **sync**, where it sums x and y, up to the point where it returns the result. Strands belonging to the same procedure are grouped into a rounded rectangle, blue for spawned procedures and tan for called procedures. Assuming that each strand takes unit time, the work is 17 time units, since there are 17 strands, and the span is 8 time units, since the critical path—shown with blue edges—contains 8 strands.

When the parent spawns a child, however, the trace is a little different. The edge (u, v) goes from parent to child as with a call, such as the edge from the blue strand in P-FIB(4) to the blue strand in P-FIB(3), but the trace contains another edge (u, u') as well, indicating that u's successor strand u' can continue to execute while v is executing. The edge from the blue strand in P-FIB(4) to the orange strand in P-FIB(4) illustrates one such edge. As with a call, there is an edge from the last strand v' in the child, but with a spawn, it no longer goes to u's successor. Instead, the edge is (v', x), where x is the strand immediately following the **sync** in the parent that ensures that the child has finished, as with the edge from the white strand in P-FIB(3) to the white strand in P-FIB(4).

You can figure out what parallel control created a particular trace. If a strand has two successors, one of them must have been spawned, and if a strand has multiple predecessors, the predecessors joined because of a **sync** statement. Thus, in the general case, the set V forms the set of strands, and the set E of directed edges represents dependencies between strands induced by parallel and procedural

control. If G contains a directed path from strand u to strand v, we say that the two strands are ***(logically) in series***. If there is no path in G either from u to v or from v to u, the strands are ***(logically) in parallel***.

A fork-join parallel trace can be pictured as a dag of strands embedded in an ***invocation tree*** of procedure instances. For example, Figure 26.1 shows the invocation tree for FIB(6), which also serves as the invocation tree for P-FIB(6), the edges between procedure instances now representing either calls or spawns. Figure 26.2 zooms in on the subtree that is shaded blue, showing the strands that constitute each procedure instance in P-FIB(4). All directed edges connecting strands run either within a procedure or along undirected edges of the invocation tree in Figure 26.1. (More general task-parallel traces that are not fork-join traces may contain some directed edges that do not run along the undirected tree edges.)

Our analyses generally assume that parallel algorithms execute on an ***ideal parallel computer***, which consists of a set of processors and a ***sequentially consistent*** shared memory. To understand sequential consistency, you first need to know that memory is accessed by ***load instructions***, which copy data from a location in the memory to a register within a processor, and by ***store instructions***, which copy data from a processor register to a location in the memory. A single line of pseudocode can entail several such instructions. For example, the line $x = y + z$ could result in load instructions to fetch each of y and z from memory into a processor, an instruction to add them together inside the processor, and a store instruction to place the result x back into memory. In a parallel computer, several processors might need to load or store at the same time. Sequential consistency means that even if multiple processors attempt to access the memory simultaneously, the shared memory behaves as if exactly one instruction from one of the processors is executed at a time, even though the actual transfer of data may happen at the same time. It is as if the instructions were executed one at a time sequentially according to some global linear order among all the processors that preserves the individual orders in which each processor executes its own instructions.

For task-parallel computations, which are scheduled onto processors automatically by a runtime system, the sequentially consistent shared memory behaves as if a parallel computation's executed instructions were executed one by one in the order of a topological sort (see Section 20.4) of its trace. That is, you can reason about the execution by imagining that the individual instructions (not generally the strands, which may aggregate many instructions) are interleaved in some linear order that preserves the partial order of the trace. Depending on scheduling, the linear order could vary from one run of the program to the next, but the behavior of any execution is always as if the instructions executed serially in a linear order consistent with the dependencies within the trace.

In addition to making assumptions about semantics, the ideal parallel-computer model makes some performance assumptions. Specifically, it assumes that each

processor in the machine has equal computing power, and it ignores the cost of scheduling. Although this last assumption may sound optimistic, it turns out that for algorithms with sufficient "parallelism" (a term we'll define precisely a little later), the overhead of scheduling is generally minimal in practice.

Performance measures

We can gauge the theoretical efficiency of a task-parallel algorithm using ***work/span analysis***, which is based on two metrics: "work" and "span." The ***work*** of a task-parallel computation is the total time to execute the entire computation on one processor. In other words, the work is the sum of the times taken by each of the strands. If each strand takes unit time, the work is just the number of vertices in the trace. The ***span*** is the fastest possible time to execute the computation on an unlimited number of processors, which corresponds to the sum of the times taken by the strands along a longest path in the trace, where "longest" means that each strand is weighted by its execution time. Such a longest path is called the ***critical path*** of the trace, and thus the span is the weight of the longest (weighted) path in the trace. (Section 22.2, pages 617–619 shows how to find a critical path in a dag $G = (V, E)$ in $\Theta(V + E)$ time.) For a trace in which each strand takes unit time, the span equals the number of strands on the critical path. For example, the trace of Figure 26.2 has 17 vertices in all and 8 vertices on its critical path, so that if each strand takes unit time, its work is 17 time units and its span is 8 time units.

The actual running time of a task-parallel computation depends not only on its work and its span, but also on how many processors are available and how the scheduler allocates strands to processors. To denote the running time of a task-parallel computation on P processors, we subscript by P. For example, we might denote the running time of an algorithm on P processors by T_P. The work is the running time on a single processor, or T_1. The span is the running time if we could run each strand on its own processor—in other words, if we had an unlimited number of processors—and so we denote the span by T_∞.

The work and span provide lower bounds on the running time T_P of a task-parallel computation on P processors:

- In one step, an ideal parallel computer with P processors can do at most P units of work, and thus in T_P time, it can perform at most $P T_P$ work. Since the total work to do is T_1, we have $P T_P \geq T_1$. Dividing by P yields the ***work law***:

$$T_P \geq T_1/P. \tag{26.2}$$

- A P-processor ideal parallel computer cannot run any faster than a machine with an unlimited number of processors. Looked at another way, a machine

with an unlimited number of processors can emulate a P-processor machine by using just P of its processors. Thus, the ***span law*** follows:

$$T_P \geq T_\infty \,. \tag{26.3}$$

We define the ***speedup*** of a computation on P processors by the ratio T_1/T_P, which says how many times faster the computation runs on P processors than on one processor. By the work law, we have $T_P \geq T_1/P$, which implies that $T_1/T_P \leq P$. Thus, the speedup on a P-processor ideal parallel computer can be at most P. When the speedup is linear in the number of processors, that is, when $T_1/T_P = \Theta(P)$, the computation exhibits ***linear speedup***. ***Perfect linear speedup*** occurs when $T_1/T_P = P$.

The ratio T_1/T_∞ of the work to the span gives the ***parallelism*** of the parallel computation. We can view the parallelism from three perspectives. As a ratio, the parallelism denotes the average amount of work that can be performed in parallel for each step along the critical path. As an upper bound, the parallelism gives the maximum possible speedup that can be achieved on any number of processors. Perhaps most important, the parallelism provides a limit on the possibility of attaining perfect linear speedup. Specifically, once the number of processors exceeds the parallelism, the computation cannot possibly achieve perfect linear speedup. To see this last point, suppose that $P > T_1/T_\infty$, in which case the span law implies that the speedup satisfies $T_1/T_P \leq T_1/T_\infty < P$. Moreover, if the number P of processors in the ideal parallel computer greatly exceeds the parallelism—that is, if $P \gg T_1/T_\infty$—then $T_1/T_P \ll P$, so that the speedup is much less than the number of processors. In other words, if the number of processors exceeds the parallelism, adding even more processors makes the speedup less perfect.

As an example, consider the computation P-FIB(4) in Figure 26.2, and assume that each strand takes unit time. Since the work is $T_1 = 17$ and the span is $T_\infty = 8$, the parallelism is $T_1/T_\infty = 17/8 = 2.125$. Consequently, achieving much more than double the performance is impossible, no matter how many processors execute the computation. For larger input sizes, however, we'll see that P-FIB(n) exhibits substantial parallelism.

We define the ***(parallel) slackness*** of a task-parallel computation executed on an ideal parallel computer with P processors to be the ratio $(T_1/T_\infty)/P = T_1/(PT_\infty)$, which is the factor by which the parallelism of the computation exceeds the number of processors in the machine. Restating the bounds on speedup, if the slackness is less than 1, perfect linear speedup is impossible, because $T_1/(PT_\infty) < 1$ and the span law imply that $T_1/T_P \leq T_1/T_\infty < P$. Indeed, as the slackness decreases from 1 and approaches 0, the speedup of the computation diverges further and further from perfect linear speedup. If the slackness is less than 1, additional parallelism in an algorithm can have a great impact on its

execution efficiency. If the slackness is greater than 1, however, the work per processor is the limiting constraint. We'll see that as the slackness increases from 1, a good scheduler can achieve closer and closer to perfect linear speedup. But once the slackness is much greater than 1, the advantage of additional parallelism shows diminishing returns.

Scheduling

Good performance depends on more than just minimizing the work and span. The strands must also be scheduled efficiently onto the processors of the parallel machine. Our fork-join parallel-programming model provides no way for a programmer to specify which strands to execute on which processors. Instead, we rely on the runtime system's scheduler to map the dynamically unfolding computation to individual processors. In practice, the scheduler maps the strands to static threads, and the operating system schedules the threads on the processors themselves. But this extra level of indirection is unnecessary for our understanding of scheduling. We can just imagine that the scheduler maps strands to processors directly.

A task-parallel scheduler must schedule the computation without knowing in advance when procedures will be spawned or when they will finish—that is, it must operate ***online***. Moreover, a good scheduler operates in a distributed fashion, where the threads implementing the scheduler cooperate to load-balance the computation. Provably good online, distributed schedulers exist, but analyzing them is complicated. Instead, to keep our analysis simple, we'll consider an online ***centralized*** scheduler that knows the global state of the computation at any moment.

In particular, we'll analyze ***greedy schedulers***, which assign as many strands to processors as possible in each time step, never leaving a processor idle if there is work that can be done. We'll classify each step of a greedy scheduler as follows:

- ***Complete step***: At least P strands are ***ready*** to execute, meaning that all strands on which they depend have finished execution. A greedy scheduler assigns any P of the ready strands to the processors, completely utilizing all the processor resources.
- ***Incomplete step***: Fewer than P strands are ready to execute. A greedy scheduler assigns each ready strand to its own processor, leaving some processors idle for the step, but executing all the ready strands.

The work law tells us that the fastest running time T_P that we can hope for on P processors must be at least T_1/P. The span law tells us that the fastest possible running time must be at least T_∞. The following theorem shows that greedy scheduling is provably good in that it achieves the sum of these two lower bounds as an upper bound.

Theorem 26.1
On an ideal parallel computer with P processors, a greedy scheduler executes a task-parallel computation with work T_1 and span T_∞ in time

$$T_P \leq T_1/P + T_\infty \,. \tag{26.4}$$

Proof Without loss of generality, assume that each strand takes unit time. (If necessary, replace each longer strand by a chain of unit-time strands.) We'll consider complete and incomplete steps separately.

In each complete step, the P processors together perform a total of P work. Thus, if the number of complete steps is k, the total work executing all the complete steps is kP. Since the greedy scheduler doesn't execute any strand more than once and only T_1 work needs to be performed, it follows that $kP \leq T_1$, from which we can conclude that the number k of complete steps is at most T_1/P.

Now, let's consider an incomplete step. Let G be the trace for the entire computation, let G' be the subtrace of G that has yet to be executed at the start of the incomplete step, and let G'' be the subtrace remaining to be executed after the incomplete step. Consider the set R of strands that are ready at the beginning of the incomplete step, where $|R| < P$. By definition, if a strand is ready, all its predecessors in trace G have executed. Thus the predecessors of strands in R do not belong to G'. A longest path in G' must necessarily start at a strand in R, since every other strand in G' has a predecessor and thus could not start a longest path. Because the greedy scheduler executes all ready strands during the incomplete step, the strands of G'' are exactly those in G' minus the strands in R. Consequently, the length of a longest path in G'' must be 1 less than the length of a longest path in G'. In other words, every incomplete step decreases the span of the trace remaining to be executed by 1. Hence, the number of incomplete steps can be at most T_∞.

Since each step is either complete or incomplete, the theorem follows. ■

The following corollary shows that a greedy scheduler always performs well.

Corollary 26.2
The running time T_P of any task-parallel computation scheduled by a greedy scheduler on a P-processor ideal parallel computer is within a factor of 2 of optimal.

Proof Let T_P^* be the running time produced by an optimal scheduler on a machine with P processors, and let T_1 and T_∞ be the work and span of the computation, respectively. Since the work and span laws—inequalities (26.2) and (26.3)—give $T_P^* \geq \max\{T_1/P, T_\infty\}$, Theorem 26.1 implies that

$$\begin{aligned} T_P &\leq T_1/P + T_\infty \\ &\leq 2 \cdot \max\{T_1/P, T_\infty\} \\ &\leq 2T_P^* \,. \end{aligned}$$

■

The next corollary shows that, in fact, a greedy scheduler achieves near-perfect linear speedup on any task-parallel computation as the slackness grows.

Corollary 26.3
Let T_P be the running time of a task-parallel computation produced by a greedy scheduler on an ideal parallel computer with P processors, and let T_1 and T_∞ be the work and span of the computation, respectively. Then, if $P \ll T_1/T_\infty$, or equivalently, the parallel slackness is much greater than 1, we have $T_P \approx T_1/P$, a speedup of approximately P.

Proof If we suppose that $P \ll T_1/T_\infty$, then it follows that $T_\infty \ll T_1/P$, and hence Theorem 26.1 gives $T_P \leq T_1/P + T_\infty \approx T_1/P$. Since the work law (26.2) dictates that $T_P \geq T_1/P$, we conclude that $T_P \approx T_1/P$, which is a speedup of $T_1/T_P \approx P$. ■

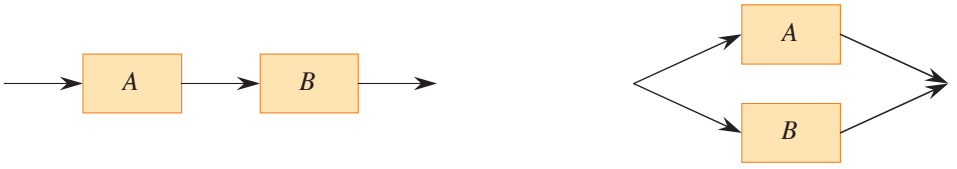
The $\ll$ symbol denotes "much less," but how much is "much less"? As a rule of thumb, a slackness of at least 10—that is, 10 times more parallelism than processors—generally suffices to achieve good speedup. Then, the span term in the greedy bound, inequality (26.4), is less than 10% of the work-per-processor term, which is good enough for most engineering situations. For example, if a computation runs on only 10 or 100 processors, it doesn't make sense to value parallelism of, say 1,000,000, over parallelism of 10,000, even with the factor of 100 difference. As Problem 26-2 shows, sometimes reducing extreme parallelism yields algorithms that are better with respect to other concerns and which still scale up well on reasonable numbers of processors.

Analyzing parallel algorithms

We now have all the tools we need to analyze parallel algorithms using work/span analysis, allowing us to bound an algorithm's running time on any number of processors. Analyzing the work is relatively straightforward, since it amounts to nothing more than analyzing the running time of an ordinary serial algorithm, namely, the serial projection of the parallel algorithm. You should already be familiar with analyzing work, since that is what most of this textbook is about! Analyzing the span is the new thing that parallelism engenders, but it's generally no harder once you get the hang of it. Let's investigate the basic ideas using the P-FIB program.

Analyzing the work $T_1(n)$ of P-FIB(n) poses no hurdles, because we've already done it. The serial projection of P-FIB is effectively the original FIB procedure, and hence, we have $T_1(n) = T(n) = \Theta(\phi^n)$ from equation (26.1).

Figure 26.3 illustrates how to analyze the span. If two traces are joined in series, their spans add to form the span of their composition, whereas if they are joined



Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

(a)

Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$

(b)

Figure 26.3 Series-parallel composition of parallel traces. **(a)** When two traces are joined in series, the work of the composition is the sum of their work, and the span of the composition is the sum of their spans. **(b)** When two traces are joined in parallel, the work of the composition remains the sum of their work, but the span of the composition is only the maximum of their spans.

in parallel, the span of their composition is the maximum of the spans of the two traces. As it turns out, the trace of any fork-join parallel computation can be built up from single strands by series-parallel composition.

Armed with an understanding of series-parallel composition, we can analyze the span of P-FIB(n). The spawned call to P-FIB$(n-1)$ in line 3 runs in parallel with the call to P-FIB$(n-2)$ in line 4. Hence, we can express the span of P-FIB(n) as the recurrence

$$\begin{aligned} T_\infty(n) &= \max\{T_\infty(n-1), T_\infty(n-2)\} + \Theta(1) \\ &= T_\infty(n-1) + \Theta(1) , \end{aligned}$$

which has solution $T_\infty(n) = \Theta(n)$. (The second equality above follows from the first because P-FIB$(n-1)$ uses P-FIB$(n-2)$ in its computation, so that the span of P-FIB$(n-1)$ must be at least as large as the span of P-FIB$(n-2)$.)

The parallelism of P-FIB(n) is $T_1(n)/T_\infty(n) = \Theta(\phi^n/n)$, which grows dramatically as n gets large. Thus, Corollary 26.3 tells us that on even the largest parallel computers, a modest value for n suffices to achieve near perfect linear speedup for P-FIB(n), because this procedure exhibits considerable parallel slackness.

Parallel loops

Many algorithms contain loops for which all the iterations can operate in parallel. Although the **spawn** and **sync** keywords can be used to parallelize such loops, it is more convenient to specify directly that the iterations of such loops can run in parallel. Our pseudocode provides this functionality via the **parallel** keyword, which precedes the **for** keyword in a **for** loop statement.

As an example, consider the problem of multiplying a square $n \times n$ matrix $A = (a_{ij})$ by an n-vector $x = (x_j)$. The resulting n-vector $y = (y_i)$ is given by the equation

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \ ,$$

for $i = 1, 2, \ldots, n$. The P-MAT-VEC procedure performs matrix-vector multiplication (actually, $y = y + Ax$) by computing all the entries of y in parallel. The **parallel for** keywords in line 1 of P-MAT-VEC indicate that the n iterations of the loop body, which includes a serial **for** loop, may be run in parallel. The initialization $y = 0$, if desired, should be performed before calling the procedure (and can be done with a **parallel for** loop).

```
P-MAT-VEC(A, x, y, n)
1  parallel for i = 1 to n          // parallel loop
2      for j = 1 to n               // serial loop
3          y_i = y_i + a_ij x_j
```

Compilers for fork-join parallel programs can implement **parallel for** loops in terms of **spawn** and **sync** by using recursive spawning. For example, for the **parallel for** loop in lines 1–3, a compiler can generate the auxiliary subroutine P-MAT-VEC-RECURSIVE and call P-MAT-VEC-RECURSIVE$(A, x, y, n, 1, n)$ in the place where the loop would be in the compiled code. As Figure 26.4 illustrates, this procedure recursively spawns the first half of the iterations of the loop to execute in parallel (line 5) with the second half of the iterations (line 6) and then executes a **sync** (line 7), thereby creating a binary tree of parallel execution. Each leaf represents a base case, which is the serial **for** loop of lines 2–3.

```
P-MAT-VEC-RECURSIVE(A, x, y, n, i, i')
1  if i == i'                       // just one iteration to do?
2      for j = 1 to n               // mimic P-MAT-VEC serial loop
3          y_i = y_i + a_ij x_j
4  else mid = ⌊(i + i')/2⌋          // parallel divide-and-conquer
5      spawn P-MAT-VEC-RECURSIVE(A, x, y, n, i, mid)
6      P-MAT-VEC-RECURSIVE(A, x, y, n, mid + 1, i')
7      sync
```

To calculate the work $T_1(n)$ of P-MAT-VEC on an $n \times n$ matrix, simply compute the running time of its serial projection, which comes from replacing the **parallel**

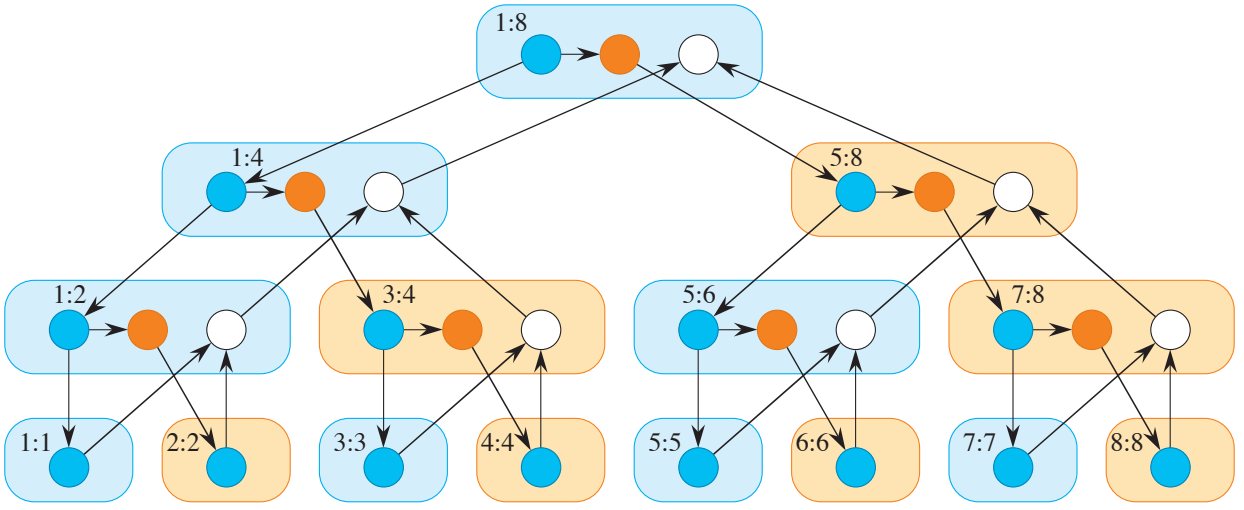


Figure 26.4 A trace for the computation of P-MAT-VEC-RECURSIVE$(A, x, y, 8, 1, 8)$. The two numbers within each rounded rectangle give the values of the last two parameters (i and i' in the procedure header) in the invocation (spawn, in blue, or call, in tan) of the procedure. The blue circles represent strands corresponding to the part of the procedure up to the spawn of P-MAT-VEC-RECURSIVE in line 5. The orange circles represent strands corresponding to the part of the procedure that calls P-MAT-VEC-RECURSIVE in line 6 up to the **sync** in line 7, where it suspends until the spawned subroutine in line 5 returns. The white circles represent strands corresponding to the (negligible) part of the procedure after the **sync** up to the point where it returns.

for loop in line 1 with an ordinary **for** loop. The running time of the resulting serial pseudocode is $\Theta(n^2)$, which means that $T_1(n) = \Theta(n^2)$. This analysis seems to ignore the overhead for recursive spawning in implementing the parallel loops, however. Indeed, the overhead of recursive spawning does increase the work of a parallel loop compared with that of its serial projection, but not asymptotically. To see why, observe that since the tree of recursive procedure instances is a full binary tree, the number of internal nodes is one less than the number of leaves (see Exercise B.5-3 on page 1175). Each internal node performs constant work to divide the iteration range, and each leaf corresponds to a base case, which takes at least constant time ($\Theta(n)$ time in this case). Thus, by amortizing the overhead of recursive spawning over the work of the iterations in the leaves, we see that the overall work increases by at most a constant factor.

To reduce the overhead of recursive spawning, task-parallel platforms sometimes ***coarsen*** the leaves of the recursion by executing several iterations in a single leaf, either automatically or under programmer control. This optimization comes at the expense of reducing the parallelism. If the computation has sufficient parallel slackness, however, near-perfect linear speedup won't be sacrificed.

Although recursive spawning doesn't affect the work of a parallel loop asymptotically, we must take it into account when analyzing the span. Consider a parallel loop with n iterations in which the ith iteration has span $iter_\infty(i)$. Since the depth of recursion is logarithmic in the number of iterations, the parallel loop's span is

$$T_\infty(n) = \Theta(\lg n) + \max\{iter_\infty(i) : 1 \le i \le n\} .$$

For example, let's compute the span of the doubly nested loops in lines 1–3 of P-MAT-VEC. The span for the **parallel for** loop control is $\Theta(\lg n)$. For each iteration of the outer parallel loop, the inner serial **for** loop contains n iterations of line 3. Since each iteration takes constant time, the total span for the inner serial **for** loop is $\Theta(n)$, no matter which iteration of the outer **parallel for** loop it's in. Thus, taking the maximum over all iterations of the outer loop and adding in the $\Theta(\lg n)$ for loop control yields an overall span of $T_\infty n = \Theta(n) + \Theta(\lg n) = \Theta(n)$ for the procedure. Since the work is $\Theta(n^2)$, the parallelism is $\Theta(n^2)/\Theta(n) = \Theta(n)$. (Exercise 26.1-7 asks you to provide an implementation with even more parallelism.)

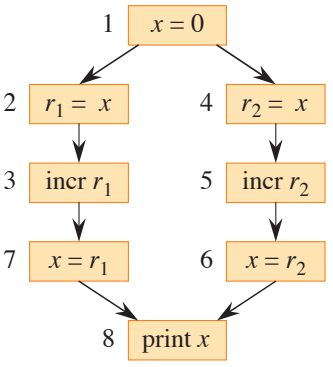
Race conditions

A parallel algorithm is ***deterministic*** if it always does the same thing on the same input, no matter how the instructions are scheduled on the multicore computer. It is ***nondeterministic*** if its behavior might vary from run to run when the input is the same. A parallel algorithm that is intended to be deterministic may nevertheless act nondeterministically, however, if it contains a difficult-to-diagnose bug called a "determinacy race."

Famous race bugs include the Therac-25 radiation therapy machine, which killed three people and injured several others, and the Northeast Blackout of 2003, which left over 50 million people in the United States without power. These pernicious bugs are notoriously hard to find. You can run tests in the lab for days without a failure, only to discover that your software sporadically crashes in the field, sometimes with dire consequences.

A ***determinacy race*** occurs when two logically parallel instructions access the same memory location and at least one of the instructions modifies the value stored in the location. The toy procedure RACE-EXAMPLE on the following page illustrates a determinacy race. After initializing x to 0 in line 1, RACE-EXAMPLE creates two parallel strands, each of which increments x in line 3. Although it might seem that a call of RACE-EXAMPLE should always print the value 2 (its serial projection certainly does), it could instead print the value 1. Let's see how this anomaly might occur.

When a processor increments x, the operation is not indivisible, but is composed of a sequence of instructions:



(a)

step	x	r_1	r_2
1	0	–	–
2	0	0	–
3	0	1	–
4	0	1	0
5	0	1	1
6	1	1	1
7	1	1	1

(b)

Figure 26.5 Illustration of the determinacy race in RACE-EXAMPLE. **(a)** A trace showing the dependencies among individual instructions. The processor registers are r_1 and r_2. Instructions unrelated to the race, such as the implementation of loop control, are omitted. **(b)** An execution sequence that elicits the bug, showing the values of x in memory and registers r_1 and r_2 for each step in the execution sequence.

```
RACE-EXAMPLE( )
1  x = 0
2  parallel for i = 1 to 2
3       x = x + 1          // determinacy race
4  print x
```

- Load x from memory into one of the processor's registers.
- Increment the value in the register.
- Store the value in the register back into x in memory.

Figure 26.5(a) illustrates a trace representing the execution of RACE-EXAMPLE, with the strands broken down to individual instructions. Recall that since an ideal parallel computer supports sequential consistency, you can view the parallel execution of a parallel algorithm as an interleaving of instructions that respects the dependencies in the trace. Part (b) of the figure shows the values in an execution of the computation that elicits the anomaly. The value x is kept in memory, and r_1 and r_2 are processor registers. In step 1, one of the processors sets x to 0. In steps 2 and 3, processor 1 loads x from memory into its register r_1 and increments it, producing the value 1 in r_1. At that point, processor 2 comes into the picture, executing instructions 4–6. Processor 2 loads x from memory into register r_2; increments it, producing the value 1 in r_2; and then stores this value into x, setting x to 1. Now, processor 1 resumes with step 7, storing the value 1 in r_1 into x, which

leaves the value of x unchanged. Therefore, step 8 prints the value 1, rather than the value 2 that the serial projection would print.

Let's recap what happened. By sequential consistency, the effect of the parallel execution is as if the executed instructions of the two processors are interleaved. If processor 1 executes all its instructions before processor 2, a trivial interleaving, the value 2 is printed. Conversely, if processor 2 executes all its instructions before processor 1, the value 2 is still printed. When the instructions of the two processors interleave nontrivially, however, it is possible, as in this example execution, that one of the updates to x is lost, resulting in the value 1 being printed.

Of course, many executions do not elicit the bug. That's the problem with determinacy races. Generally, most instruction orderings produce correct results, such as any where the instructions on the left branch execute before the instructions on the right branch, or vice versa. But some orderings generate improper results when the instructions interleave. Consequently, races can be extremely hard to test for. Your program may fail, but you may be unable to reliably reproduce the failure in subsequent tests, confounding your attempts to locate the bug in your code and fix it. Task-parallel programming environments often provide race-detection productivity tools to help you isolate race bugs.

Many parallel programs in the real world are intentionally nondeterministic. They contain determinacy races, but they mitigate the dangers of nondeterminism through the use of mutual-exclusion locks and other methods of synchronization. For our purposes, however, we'll insist on an absence of determinacy races in the algorithms we develop. Nondeterministic programs are indeed interesting, but nondeterministic programming is a more advanced topic and unnecessary for a wide swath of interesting parallel algorithms.

To ensure that algorithms are deterministic, any two strands that operate in parallel should be ***mutually noninterfering***: they only read, and do not modify, any memory locations accessed by both of them. Consequently, in a **parallel for** construct, such as the outer loop of P-MAT-VEC, we want all the iterations of the body, including any code an iteration executes in subroutines, to be mutually noninterfering. And between a **spawn** and its corresponding **sync**, we want the code executed by the spawned child and the code executed by the parent to be mutually noninterfering, once again including invoked subroutines.

As an example of how easy it is to write code with unintentional races, the P-MAT-VEC-WRONG procedure on the next page is a faulty parallel implementation of matrix-vector multiplication that achieves a span of $\Theta(\lg n)$ by parallelizing the inner **for** loop. This procedure is incorrect, unfortunately, due to determinacy races when updating y_i in line 3, which executes in parallel for all n values of j.

Index variables of **parallel for** loops, such as i in line 1 and j in line 2, do not cause races between iterations. Conceptually, each iteration of the loop creates an independent variable to hold the index of that iteration during that iteration's

```
P-MAT-VEC-WRONG(A, x, y, n)
1  parallel for i = 1 to n
2      parallel for j = 1 to n
3          y_i = y_i + a_ij x_j        // determinacy race
```

execution of the loop body. Even if two parallel iterations both access the same index variable, they really are accessing different variable instances—hence different memory locations—and no race occurs.

A parallel algorithm with races can sometimes be deterministic. As an example, two parallel threads might store the same value into a shared variable, and it wouldn't matter which stored the value first. For simplicity, however, we generally prefer code without determinacy races, even if the races are benign. And good parallel programmers frown on code with determinacy races that cause nondeterministic behavior, if deterministic code that performs comparably is an option.

But nondeterministic code does have its place. For example, you can't implement a parallel hash table, a highly practical data structure, without writing code containing determinacy races. Much research has centered around how to extend the fork-join model to incorporate limited "structured" nondeterminism while avoiding the full measure of complications that arise when nondeterminism is completely unrestricted.

A chess lesson

To illustrate the power of work/span analysis, this section closes with a true story that occurred during the development of one of the first world-class parallel chess-playing programs [106] many years ago. The timings below have been simplified for exposition.

The chess program was developed and tested on a 32-processor computer, but it was designed to run on a supercomputer with 512 processors. Since the supercomputer availability was limited and expensive, the developers ran benchmarks on the small computer and extrapolated performance to the large computer.

At one point, the developers incorporated an optimization into the program that reduced its running time on an important benchmark on the small machine from $T_{32} = 65$ seconds to $T'_{32} = 40$ seconds. Yet, the developers used the work and span performance measures to conclude that the optimized version, which was faster on 32 processors, would actually be slower than the original version on the 512 processors of the large machine. As a result, they abandoned the "optimization."

Here is their work/span analysis. The original version of the program had work $T_1 = 2048$ seconds and span $T_\infty = 1$ second. Let's treat inequality (26.4) on

page 760 as the equation $T_P = T_1/P + T_\infty$, which we can use as an approximation to the running time on P processors. Then indeed we have $T_{32} = 2048/32 + 1 = 65$. With the optimization, the work becomes $T_1' = 1024$ seconds, and the span becomes $T_\infty' = 8$ seconds. Our approximation gives $T_{32}' = 1024/32 + 8 = 40$.

The relative speeds of the two versions switch when we estimate their running times on 512 processors, however. The first version has a running time of $T_{512} = 2048/512+1 = 5$ seconds, and the second version runs in $T_{512}' = 1024/512+8 = 10$ seconds. The optimization that speeds up the program on 32 processors makes the program run for twice as long on 512 processors! The optimized version's span of 8, which is not the dominant term in the running time on 32 processors, becomes the dominant term on 512 processors, nullifying the advantage from using more processors. The optimization does not scale up.

The moral of the story is that work/span analysis, and measurements of work and span, can be superior to measured running times alone in extrapolating an algorithm's scalability.

Exercises

26.1-1

What does a trace for the execution of a serial algorithm look like?

26.1-2

Suppose that line 4 of P-FIB spawns P-FIB$(n-2)$, rather than calling it as is done in the pseudocode. How would the trace of P-FIB(4) in Figure 26.2 change? What is the impact on the asymptotic work, span, and parallelism?

26.1-3

Draw the trace that results from executing P-FIB(5). Assuming that each strand in the computation takes unit time, what are the work, span, and parallelism of the computation? Show how to schedule the trace on 3 processors using greedy scheduling by labeling each strand with the time step in which it is executed.

26.1-4

Prove that a greedy scheduler achieves the following time bound, which is slightly stronger than the bound proved in Theorem 26.1:

$$T_P \le \frac{T_1 - T_\infty}{P} + T_\infty \,. \tag{26.5}$$

26.1-5

Construct a trace for which one execution by a greedy scheduler can take nearly twice the time of another execution by a greedy scheduler on the same number of processors. Describe how the two executions would proceed.

26.1-6

Professor Karan measures her deterministic task-parallel algorithm on 4, 10, and 64 processors of an ideal parallel computer using a greedy scheduler. She claims that the three runs yielded $T_4 = 80$ seconds, $T_{10} = 42$ seconds, and $T_{64} = 10$ seconds. Argue that the professor is either lying or incompetent. (*Hint:* Use the work law (26.2), the span law (26.3), and inequality (26.5) from Exercise 26.1-4.)

26.1-7

Give a parallel algorithm to multiply an $n \times n$ matrix by an n-vector that achieves $\Theta(n^2/\lg n)$ parallelism while maintaining $\Theta(n^2)$ work.

26.1-8

Analyze the work, span, and parallelism of the procedure P-TRANSPOSE, which transposes an $n \times n$ matrix A in place.

```
P-TRANSPOSE(A, n)
1  parallel for j = 2 to n
2      parallel for i = 1 to j − 1
3          exchange a_ij with a_ji
```

26.1-9

Suppose that instead of a **parallel for** loop in line 2, the P-TRANSPOSE procedure in Exercise 26.1-8 had an ordinary **for** loop. Analyze the work, span, and parallelism of the resulting algorithm.

26.1-10

For what number of processors do the two versions of the chess program run equally fast, assuming that $T_P = T_1/P + T_\infty$?

26.2 Parallel matrix multiplication

In this section, we'll explore how to parallelize the three matrix-multiplication algorithms from Sections 4.1 and 4.2. We'll see that each algorithm can be parallelized in a straightforward fashion using either parallel loops or recursive spawning. We'll analyze them using work/span analysis, and we'll see that each parallel algorithm attains the same performance on one processor as its corresponding serial algorithm, while scaling up to large numbers of processors.

A parallel algorithm for matrix multiplication using parallel loops

The first algorithm we'll study is P-MATRIX-MULTIPLY, which simply parallelizes the two outer loops in the procedure MATRIX-MULTIPLY on page 81.

```
P-MATRIX-MULTIPLY(A, B, C, n)
1  parallel for i = 1 to n                  // compute entries in each of n rows
2      parallel for j = 1 to n              // compute n entries in row i
3          for k = 1 to n
4              c_ij = c_ij + a_ik · b_kj    // add in another term of equation (4.1)
```

Let's analyze P-MATRIX-MULTIPLY. Since the serial projection of the algorithm is just MATRIX-MULTIPLY, the work is the same as the running time of MATRIX-MULTIPLY: $T_1(n) = \Theta(n^3)$. The span is $T_\infty(n) = \Theta(n)$, because it follows a path down the tree of recursion for the **parallel for** loop starting in line 1, then down the tree of recursion for the **parallel for** loop starting in line 2, and then executes all n iterations of the ordinary **for** loop starting in line 3, resulting in a total span of $\Theta(\lg n) + \Theta(\lg n) + \Theta(n) = \Theta(n)$. Thus the parallelism is $\Theta(n^3)/\Theta(n) = \Theta(n^2)$. (Exercise 26.2-3 asks you to parallelize the inner loop to obtain a parallelism of $\Theta(n^3/\lg n)$, which you cannot do straightforwardly using **parallel for**, because you would create races.)

A parallel divide-and-conquer algorithm for matrix multiplication

Section 4.1 shows how to multiply $n \times n$ matrices serially in $\Theta(n^3)$ time using a divide-and-conquer strategy. Let's see how to parallelize that algorithm using recursive spawning instead of calls.

The serial MATRIX-MULTIPLY-RECURSIVE procedure on page 83 takes as input three $n \times n$ matrices A, B, and C and performs the matrix calculation $C = C + A \cdot B$ by recursively performing eight multiplications of $n/2 \times n/2$ submatrices of A and B. The P-MATRIX-MULTIPLY-RECURSIVE procedure on the following page implements the same divide-and-conquer strategy, but it uses spawning to perform the eight multiplications in parallel. To avoid determinacy races in updating the elements of C, it creates a temporary matrix D to store four of the submatrix products. At the end, it adds C and D together to produce the final result. (Problem 26-2 asks you to eliminate the temporary matrix D at the expense of some parallelism.)

Lines 2–3 of P-MATRIX-MULTIPLY-RECURSIVE handle the base case of multiplying 1×1 matrices. The remainder of the procedure deals with the recursive case. Line 4 allocates a temporary matrix D, and lines 5–7 zero it. Line 8 partitions each of the four matrices A, B, C, and D into $n/2 \times n/2$ submatrices. (As

```
P-MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
 1  if n == 1                          // just one element in each matrix?
 2      c_11 = c_11 + a_11 · b_11
 3      return
 4  let D be a new n × n matrix        // temporary matrix
 5  parallel for i = 1 to n            // set D = 0
 6      parallel for j = 1 to n
 7          d_ij = 0
 8  partition A, B, C, and D into n/2 × n/2 submatrices
        A_11, A_12, A_21, A_22; B_11, B_12, B_21, B_22; C_11, C_12, C_21, C_22;
        and D_11, D_12, D_21, D_22; respectively
 9  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_11, B_11, C_11, n/2)
10  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_11, B_12, C_12, n/2)
11  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_21, B_11, C_21, n/2)
12  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_21, B_12, C_22, n/2)
13  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_12, B_21, D_11, n/2)
14  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_12, B_22, D_12, n/2)
15  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_22, B_21, D_21, n/2)
16  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_22, B_22, D_22, n/2)
17  sync                               // wait for spawned submatrix products
18  parallel for i = 1 to n            // update C = C + D
19      parallel for j = 1 to n
20          c_ij = c_ij + d_ij
```

with MATRIX-MULTIPLY-RECURSIVE on page 83, we're glossing over the subtle issue of how to use index calculations to represent submatrix sections of a matrix.) The spawned recursive call in line 9 sets $C_{11} = C_{11} + A_{11} \cdot B_{11}$, so that C_{11} accumulates the first of the two terms in equation (4.5) on page 82. Similarly, lines 10–12 cause each of C_{12}, C_{21}, and C_{22} in parallel to accumulate the first of the two terms in equations (4.6)–(4.8), respectively. Line 13 sets the submatrix D_{11} to the submatrix product $A_{12} \cdot B_{21}$, so that D_{11} equals the second of the two terms in equation (4.5). Lines 14–16 set each of D_{12}, D_{21}, and D_{22} in parallel to the second of the two terms in equations (4.6)–(4.8), respectively. The **sync** statement in line 17 ensures that all the spawned submatrix products in lines 9–16 have been computed, after which the doubly nested **parallel for** loops in lines 18–20 add the elements of D to the corresponding elements of C.

Let's analyze the P-MATRIX-MULTIPLY-RECURSIVE procedure. We start by analyzing the work $M_1(n)$, echoing the serial running-time analysis of its progenitor MATRIX-MULTIPLY-RECURSIVE. The recursive case allocates and zeros the

temporary matrix D in $\Theta(n^2)$ time, partitions in $\Theta(1)$ time, performs eight recursive multiplications of $n/2 \times n/2$ matrices, and finishes up with the $\Theta(n^2)$ work from adding two $n \times n$ matrices. Thus the work outside the spawned recursive calls is $\Theta(n^2)$, and the recurrence for the work $M_1(n)$ becomes

$$\begin{aligned} M_1(n) &= 8M_1(n/2) + \Theta(n^2) \\ &= \Theta(n^3) \end{aligned}$$

by case 1 of the master theorem (Theorem 4.1). Not surprisingly, the work of this parallel algorithm is asymptotically the same as the running time of the procedure MATRIX-MULTIPLY on page 81, with its triply nested loops.

Let's determine the span $M_\infty(n)$ of P-MATRIX-MULTIPLY-RECURSIVE. Because the eight parallel recursive spawns all execute on matrices of the same size, the maximum span for any recursive spawn is just the span of a single one of them, or $M_\infty(n/2)$. The span for the doubly nested **parallel for** loops in lines 5–7 is $\Theta(\lg n)$ because each loop control adds $\Theta(\lg n)$ to the constant span of line 7. Similarly, the doubly nested **parallel for** loops in lines 18–20 add another $\Theta(\lg n)$. Matrix partitioning by index calculation has $\Theta(1)$ span, which is dominated by the $\Theta(\lg n)$ span of the nested loops. We obtain the recurrence

$$M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) \,. \tag{26.6}$$

Since this recurrence falls under case 2 of the master theorem with $k = 1$, the solution is $M_\infty(n) = \Theta(\lg^2 n)$.

The parallelism of P-MATRIX-MULTIPLY-RECURSIVE is $M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$, which is huge. (Problem 26-2 asks you to simplify this parallel algorithm at the expense of just a little less parallelism.)

Parallelizing Strassen's method

To parallelize Strassen's algorithm, we can follow the same general outline as on pages 86–87, but use spawning. You may find it helpful to compare each step below with the corresponding step there. We'll analyze costs as we go along to develop recurrences $T_1(n)$ and $T_\infty(n)$ for the overall work and span, respectively.

1. If $n = 1$, the matrices each contain a single element. Perform a single scalar multiplication and a single scalar addition, and return. Otherwise, partition the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.2) on page 82. This step takes $\Theta(1)$ work and $\Theta(1)$ span by index calculation.
2. Create $n/2 \times n/2$ matrices $S_1, S_2, \ldots, S_{10}$, each of which is the sum or difference of two submatrices from step 1. Create and zero the entries of seven $n/2 \times n/2$ matrices $P_1, P_2, \ldots, P_7$ to hold seven $n/2 \times n/2$ matrix products. All

17 matrices can be created, and the P_i initialized, with doubly nested **parallel for** loops using $\Theta(n^2)$ work and $\Theta(\lg n)$ span.

3. Using the submatrices from step 1 and the matrices $S_1, S_2, \ldots, S_{10}$ created in step 2, recursively spawn computations of each of the seven $n/2 \times n/2$ matrix products $P_1, P_2, \ldots, P_7$, taking $7T_1(n/2)$ work and $T_\infty(n/2)$ span.
4. Update the four submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding or subtracting various P_i matrices. Using doubly nested **parallel for** loops, computing all four submatrices takes $\Theta(n^2)$ work and $\Theta(\lg n)$ span.

Let's analyze this algorithm. Since the serial projection is the same as the original serial algorithm, the work is just the running time of the serial projection, namely, $\Theta(n^{\lg 7})$. As we did with P-MATRIX-MULTIPLY-RECURSIVE, we can devise a recurrence for the span. In this case, seven recursive calls execute in parallel, but since they all operate on matrices of the same size, we obtain the same recurrence (26.6) as we did for P-MATRIX-MULTIPLY-RECURSIVE, with solution $\Theta(\lg^2 n)$. Thus the parallel version of Strassen's method has parallelism $\Theta(n^{\lg 7}/\lg^2 n)$, which is large. Although the parallelism is slightly less than that of P-MATRIX-MULTIPLY-RECURSIVE, that's just because the work is also less.

Exercises

26.2-1

Draw the trace for computing P-MATRIX-MULTIPLY on 2×2 matrices, labeling how the vertices in your diagram correspond to strands in the execution of the algorithm. Assuming that each strand executes in unit time, analyze the work, span, and parallelism of this computation.

26.2-2

Repeat Exercise 26.2-1 for P-MATRIX-MULTIPLY-RECURSIVE.

26.2-3

Give pseudocode for a parallel algorithm that multiplies two $n \times n$ matrices with work $\Theta(n^3)$ but span only $\Theta(\lg n)$. Analyze your algorithm.

26.2-4

Give pseudocode for an efficient parallel algorithm that multiplies a $p \times q$ matrix by a $q \times r$ matrix. Your algorithm should be highly parallel even if any of p, q, and r equal 1. Analyze your algorithm.

26.2-5
Give pseudocode for an efficient parallel version of the Floyd-Warshall algorithm (see Section 23.2), which computes shortest paths between all pairs of vertices in an edge-weighted graph. Analyze your algorithm.

26.3 Parallel merge sort

We first saw serial merge sort in Section 2.3.1, and in Section 2.3.2 we analyzed its running time and showed it to be $\Theta(n \lg n)$. Because merge sort already uses the divide-and-conquer method, it seems like a terrific candidate for implementing using fork-join parallelism.

The procedure P-MERGE-SORT modifies merge sort to spawn the first recursive call. Like its serial counterpart MERGE-SORT on page 39, the P-MERGE-SORT procedure sorts the subarray $A[p:r]$. After the **sync** statement in line 8 ensures that the two recursive spawns in lines 5 and 7 have finished, P-MERGE-SORT calls the P-MERGE procedure, a parallel merging algorithm, which is on page 779, but you don't need to bother looking at it right now.

```
P-MERGE-SORT(A, p, r)
 1  if p ≥ r                    // zero or one element?
 2      return
 3  q = ⌊(p + r)/2⌋             // midpoint of A[p : r]
 4  // Recursively sort A[p : q] in parallel.
 5  spawn P-MERGE-SORT(A, p, q)
 6  // Recursively sort A[q + 1 : r] in parallel.
 7  spawn P-MERGE-SORT(A, q + 1, r)
 8  sync                        // wait for spawns
 9  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
10  P-MERGE(A, p, q, r)
```

First, let's use work/span analysis to get some intuition for why we need a parallel merge procedure. After all, it may seem as though there should be plenty of parallelism just by parallelizing MERGE-SORT without worrying about parallelizing the merge. But what would happen if the call to P-MERGE in line 10 of P-MERGE-SORT were replaced by a call to the serial MERGE procedure on page 36? Let's call the pseudocode so modified P-NAIVE-MERGE-SORT.

Let $T_1(n)$ be the (worst-case) work of P-NAIVE-MERGE-SORT on an n-element subarray, where $n = r - p + 1$ is the number of elements in $A[p:r]$, and let $T_\infty(n)$

be the span. Because MERGE is serial with running time $\Theta(n)$, both its work and span are $\Theta(n)$. Since the serial projection of P-NAIVE-MERGE-SORT is exactly MERGE-SORT, its work is $T_1(n) = \Theta(n \lg n)$. The two recursive calls in lines 5 and 7 run in parallel, and so its span is given by the recurrence

$$\begin{aligned} T_\infty(n) &= T_\infty(n/2) + \Theta(n) \\ &= \Theta(n) , \end{aligned}$$

by case 1 of the master theorem. Thus the parallelism of P-NAIVE-MERGE-SORT is $T_1(n)/T_\infty(n) = \Theta(\lg n)$, which is an unimpressive amount of parallelism. To sort a million elements, for example, since $\lg 10^6 \approx 20$, it might achieve linear speedup on a few processors, but it would not scale up to dozens of processors.

The parallelism bottleneck in P-NAIVE-MERGE-SORT is plainly the MERGE procedure. If we asymptotically reduce the span of merging, the master theorem dictates that the span of parallel merge sort will also get smaller. When you look at the pseudocode for MERGE, it may seem that merging is inherently serial, but it's not. We can fashion a parallel merging algorithm. The goal is to reduce the span of parallel merging asymptotically, but if we want an efficient parallel algorithm, we must ensure that the $\Theta(n)$ bound on work doesn't increase.

Figure 26.6 depicts the divide-and-conquer strategy that we'll use in P-MERGE. The heart of the algorithm is a recursive auxiliary procedure P-MERGE-AUX that merges two sorted subarrays of an array A into a subarray of another array B in parallel. Specifically, P-MERGE-AUX merges $A[p_1 : r_1]$ and $A[p_2 : r_2]$ into subarray $B[p_3 : r_3]$, where $r_3 = p_3 + (r_1 - p_1 + 1) + (r_2 - p_2 + 1) - 1 = p_3 + (r_1 - p_1) + (r_2 - p_2) + 1$.

The key idea of the recursive merging algorithm in P-MERGE-AUX is to split each of the two sorted subarrays of A around a pivot x, such that all the elements in the lower part of each subarray are at most x and all the elements in the upper part of each subarray are at least x. The procedure can then recurse in parallel on two subtasks: merging the two lower parts, and merging the two upper parts. The trick is to find a pivot x so that the recursion is not too lopsided. We don't want a situation such as that in QUICKSORT on page 183, where bad partitioning elements lead to a dramatic loss of asymptotic efficiency. We could opt to partition around a random element, as RANDOMIZED-QUICKSORT on page 192 does, but because the input subarrays are sorted, P-MERGE-AUX can quickly determine a pivot that always works well.

Specifically, the recursive merging algorithm picks the pivot x as the middle element of the larger of the two input subarrays, which we can assume without loss of generality is $A[p_1 : r_1]$, since otherwise, the two subarrays can just switch roles. That is, $x = A[q_1]$, where $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$. Because $A[p_1 : r_1]$ is sorted, x is a median of the subarray elements: every element in $A[p_1 : q_1 - 1]$ is no more than x, and every element in $A[q_1 + 1 : r_1]$ is no less than x. Then the

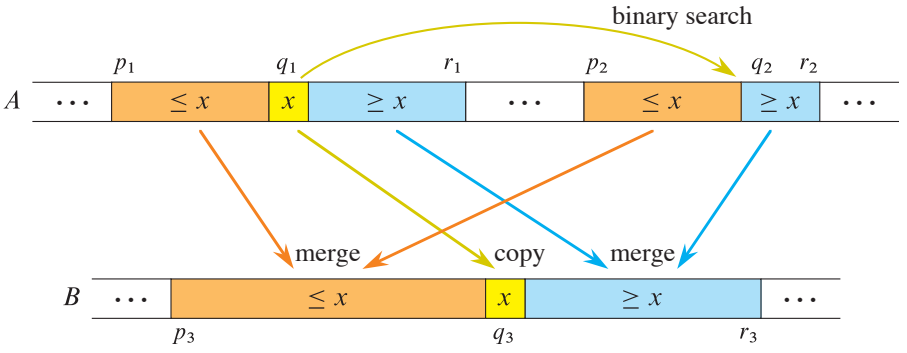


Figure 26.6 The idea behind P-MERGE-AUX, which merges two sorted subarrays $A[p_1 : r_1]$ and $A[p_2 : r_2]$ into the subarray $B[p_3 : r_3]$ in parallel. Letting $x = A[q_1]$ (shown in yellow) be a median of $A[p_1 : r_1]$ and q_2 be a place in $A[p_2 : r_2]$ such that x would fall between $A[q_2 - 1]$ and $A[q_2]$, every element in the subarrays $A[p_1 : q_1 - 1]$ and $A[p_2 : q_2 - 1]$ (shown in orange) is at most x, and every element in the subarrays $A[q_1 + 1 : r_1]$ and $A[q_2 + 1 : r_2]$ (shown in blue) is at least x. To merge, compute the index q_3 where x belongs in $B[p_3 : r_3]$, copy x into $B[q_3]$, and then recursively merge $A[p_1 : q_1 - 1]$ with $A[p_2 : q_2 - 1]$ into $B[p_3 : q_3 - 1]$ and $A[q_1 + 1 : r_1]$ with $A[q_2 : r_2]$ into $B[q_3 + 1 : r_3]$.

algorithm finds the "split point" q_2 in the smaller subarray $A[p_2 : r_2]$ such that all the elements in $A[p_2 : q_2 - 1]$ (if any) are at most x and all the elements in $A[q_2 : r_2]$ (if any) are at least x. Intuitively, the subarray $A[p_2 : r_2]$ would still be sorted if x were inserted between $A[q_2 - 1]$ and $A[q_2]$ (although the algorithm doesn't do that). Since $A[p_2 : r_2]$ is sorted, a minor variant of binary search (see Exercise 2.3-6) with x as the search key can find the split point q_2 in $\Theta(\lg n)$ time in the worst case. As we'll see when we get to the analysis, even if x splits $A[p_2 : r_2]$ badly—x is either smaller than all the subarray elements or larger—we'll still have at least $1/4$ of the elements in each of the two recursive merges. Thus the larger of the recursive merges operates on at most $3/4$ elements, and the recursion is guaranteed to bottom out after $\Theta(\lg n)$ recursive calls.

Now let's put these ideas into pseudocode. We start with the serial procedure FIND-SPLIT-POINT(A, p, r, x) on the next page, which takes as input a sorted subarray $A[p : r]$ and a key x. The procedure returns a split point of $A[p : r]$: an index q in the range $p \le q \le r + 1$ such that all the elements in $A[p : q - 1]$ (if any) are at most x and all the elements in $A[q : r]$ (if any) are at least x.

The FIND-SPLIT-POINT procedure uses binary search to find the split point. Lines 1 and 2 establish the range of indices for the search. Each time through the **while** loop, line 5 compares the middle element of the range with the search key x, and lines 6 and 7 narrow the search range to either the lower half or the upper half of the subarray, depending on the result of the test. In the end, after the range has been narrowed to a single index, line 8 returns that index as the split point.

```
FIND-SPLIT-POINT(A, p, r, x)
1  low = p                          // low end of search range
2  high = r + 1                     // high end of search range
3  while low < high                 // more than one element?
4      mid = ⌊(low + high)/2⌋       // midpoint of range
5      if x ≤ A[mid]                // is answer q ≤ mid?
6          high = mid               // narrow search to A[low : mid]
7      else low = mid + 1           // narrow search to A[mid + 1 : high]
8  return low
```

Because FIND-SPLIT-POINT contains no parallelism, its span is just its serial running time, which is also its work. On a subarray $A[p:r]$ of size $n = r - p + 1$, each iteration of the **while** loop halves the search range, which means that the loop terminates after $\Theta(\lg n)$ iterations. Since each iteration takes constant time, the algorithm runs in $\Theta(\lg n)$ (worst-case) time. Thus the procedure has work and span $\Theta(\lg n)$.

Let's now look at the pseudocode for the parallel merging procedure P-MERGE on the next page. Most of the pseudocode is devoted to the recursive procedure P-MERGE-AUX. The procedure P-MERGE itself is just a "wrapper" that sets up for P-MERGE-AUX. It allocates a new array $B[p:r]$ to hold the output of P-MERGE-AUX in line 1. It then calls P-MERGE-AUX in line 2, passing the indices of the two subarrays to be merged and providing B as the output destination of the merged result, starting at index p. After P-MERGE-AUX returns, lines 3–4 perform a parallel copy of the output $B[p:r]$ into the subarray $A[p:r]$, which is where P-MERGE-SORT expects it.

The P-MERGE-AUX procedure is the interesting part of the algorithm. Let's start by understanding the parameters of this recursive parallel procedure. The input array A and the four indices p_1, r_1, p_2, r_2 specify the subarrays $A[p_1:r_1]$ and $A[p_2:r_2]$ to be merged. The array B and the index p_3 indicate that the merged result should be stored into $B[p_3:r_3]$, where $r_3 = p_3 + (r_1 - p_1) + (r_2 - p_2) + 1$, as we saw earlier. The end index r_3 of the output subarray is not needed by the pseudocode, but it helps conceptually to name the end index, as in the comment in line 13.

The procedure begins by checking the base case of the recursion and doing some bookkeeping to simplify the rest of the pseudocode. Lines 1 and 2 test whether the two subarrays are both empty, in which case the procedure returns. Line 3 checks whether the first subarray contains fewer elements than the second subarray. Since the number of elements in the first subarray is $r_1 - p_1 + 1$ and the number in the second subarray is $r_2 - p_2 + 1$, the test omits the two "+1's." If the first subarray

```
P-MERGE(A, p, q, r)
 1  let B[p : r] be a new array                    // allocate scratch array
 2  P-MERGE-AUX(A, p, q, q + 1, r, B, p)           // merge from A into B
 3  parallel for i = p to r                        // copy B back to A in parallel
 4      A[i] = B[i]

P-MERGE-AUX(A, p1, r1, p2, r2, B, p3)
 1  if p1 > r1 and p2 > r2                         // are both subarrays empty?
 2      return
 3  if r1 − p1 < r2 − p2                           // second subarray bigger?
 4      exchange p1 with p2                        // swap subarray roles
 5      exchange r1 with r2
 6  q1 = ⌊(p1 + r1)/2⌋                             // midpoint of A[p1 : r1]
 7  x = A[q1]                                      // median of A[p1 : r1] is pivot x
 8  q2 = FIND-SPLIT-POINT(A, p2, r2, x)            // split A[p2 : r2] around x
 9  q3 = p3 + (q1 − p1) + (q2 − p2)                // where x belongs in B ...
10  B[q3] = x                                      // ... put it there
11  // Recursively merge A[p1 : q1 − 1] and A[p2 : q2 − 1] into B[p3 : q3 − 1].
12  spawn P-MERGE-AUX(A, p1, q1 − 1, p2, q2 − 1, B, p3)
13  // Recursively merge A[q1 + 1 : r1] and A[q2 : r2] into B[q3 + 1 : r3].
14  spawn P-MERGE-AUX(A, q1 + 1, r1, q2, r2, B, q3 + 1)
15  sync                                           // wait for spawns
```

is the smaller of the two, lines 4 and 5 switch the roles of the subarrays so that $A[p_1, r_1]$ refers to the larger subarray for the balance of the procedure.

We're now at the crux of P-MERGE-AUX: implementing the parallel divide-and-conquer strategy. As we continue our pseudocode walk, you may find it helpful to refer again to Figure 26.6.

First the divide step. Line 6 computes the midpoint q_1 of $A[p_1 : r_1]$, which indexes a median $x = A[q_1]$ of this subarray to be used as the pivot, and line 7 determines x itself. Next, line 8 uses the FIND-SPLIT-POINT procedure to find the index q_2 in $A[p_2 : r_2]$ such that all elements in $A[p_2 : q_2 - 1]$ are at most x and all the elements in $A[q_2 : r_2]$ are at least x. Line 9 computes the index q_3 of the element that divides the output subarray $B[p_3 : r_3]$ into $B[p_3 : q_3 - 1]$ and $B[q_3 + 1 : r_3]$, and then line 10 puts x directly into $B[q_3]$, which is where it belongs in the output.

Next is the conquer step, which is where the parallel recursion occurs. Lines 12 and 14 each spawn P-MERGE-AUX to recursively merge from A into B, the first to merge the smaller elements and the second to merge the larger elements. The

sync statement in line 15 ensures that the subproblems finish before the procedure returns.

There is no combine step, as $B[p:r]$ already contains the correct sorted output.

Work/span analysis of parallel merging

Let's first analyze the worst-case span $T_\infty(n)$ of P-MERGE-AUX on input subarrays that together contain a total of n elements. The call to FIND-SPLIT-POINT in line 8 contributes $\Theta(\lg n)$ to the span in the worst case, and the procedure performs at most a constant amount of additional serial work outside of the two recursive spawns in lines 12 and 14.

Because the two recursive spawns operate logically in parallel, only one of them contributes to the overall worst-case span. We claimed earlier that neither recursive invocation ever operates on more than $3n/4$ elements. Let's see why. Let $n_1 = r_1 - p_1 + 1$ and $n_2 = r_2 - p_2 + 1$, where $n = n_1 + n_2$, be the sizes of the two subarrays when line 6 starts executing, that is, after we have established that $n_2 \le n_1$ by swapping the roles of the two subarrays, if necessary. Since the pivot x is a median of of $A[p_1:r_1]$, in the worst case, a recursive merge involves at most $n_1/2$ elements of $A[p_1:r_1]$, but it might involve all n_2 of the elements of $A[p_2:r_2]$. Thus we can bound the number of elements involved in a recursive invocation of P-MERGE-AUX by

$$\begin{aligned} n_1/2 + n_2 &= (2n_1 + 4n_2)/4 \\ &\le (3n_1 + 3n_2)/4 \quad (\text{since } n_2 \le n_1) \\ &= 3n/4\,, \end{aligned}$$

proving the claim.

The worst-case span of P-MERGE-AUX can therefore be described by the following recurrence:

$$T_\infty(n) = T_\infty(3n/4) + \Theta(\lg n)\,. \tag{26.7}$$

Because this recurrence falls under case 2 of the master theorem with $k = 1$, its solution is $T_\infty(n) = \Theta(\lg^2 n)$.

Now let's verify that the work $T_1(n)$ of P-MERGE-AUX on n elements is linear. A lower bound of $\Omega(n)$ is straightforward, since each of the n elements is copied from array A to array B. We'll show that $T_1(n) = O(n)$ by deriving a recurrence for the worst-case work. The binary search in line 8 costs $\Theta(\lg n)$ in the worst case, which dominates the other work outside of the recursive spawns. For the recursive spawns, observe that although lines 12 and 14 might merge different numbers of elements, the two recursive spawns together merge at most $n - 1$ elements (since $x = A[q]$ is not merged). Moreover, as we saw when analyzing the span, a recursive spawn operates on at most $3n/4$ elements. We therefore obtain the recurrence

$$T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)\,, \tag{26.8}$$

where α lies in the range $1/4 \le \alpha \le 3/4$. The value of α can vary from one recursive invocation to another.

We'll use the substitution method (see Section 4.3) to prove that the above recurrence (26.8) has solution $T_1(n) = O(n)$. (You could also use the Akra-Bazzi method from Section 4.7.) Assume that $T_1(n) \le c_1 n - c_2 \lg n$ for some positive constants c_1 and c_2. Using the properties of logarithms on pages 66–67—in particular, to deduce that $\lg\alpha + \lg(1-\alpha) = -\Theta(1)$—substitution yields

$$\begin{aligned}
T_1(n) &\le (c_1\alpha n - c_2\lg(\alpha n)) + (c_1(1-\alpha)n - c_2\lg((1-\alpha)n)) + \Theta(\lg n) \\
&= c_1(\alpha + (1-\alpha))n - c_2(\lg(\alpha n) + \lg((1-\alpha)n)) + \Theta(\lg n) \\
&= c_1 n - c_2(\lg\alpha + \lg n + \lg(1-\alpha) + \lg n) + \Theta(\lg n) \\
&= c_1 n - c_2\lg n - c_2(\lg n + \lg\alpha + \lg(1-\alpha)) + \Theta(\lg n) \\
&= c_1 n - c_2\lg n - c_2(\lg n - \Theta(1)) + \Theta(\lg n) \\
&\le c_1 n - c_2\lg n\,,
\end{aligned}$$

if we choose c_2 large enough that the $c_2(\lg n - \Theta(1))$ term dominates the $\Theta(\lg n)$ term for sufficiently large n. Furthermore, we can choose c_1 large enough to satisfy the implied $\Theta(1)$ base cases of the recurrence, completing the induction. The lower and upper bounds of $\Omega(n)$ and $O(n)$ give $T_1(n) = \Theta(n)$, asymptotically the same work as for serial merging.

The execution of the pseudocode in the P-MERGE procedure itself does not add asymptotically to the work and span of P-MERGE-AUX. The **parallel for** loop in lines 3–4 has $\Theta(\lg n)$ span due to the loop control, and each iteration runs in constant time. Thus the $\Theta(\lg^2 n)$ span of P-MERGE-AUX dominates, yielding $\Theta(\lg^2 n)$ span overall for P-MERGE. The **parallel for** loop contains $\Theta(n)$ work, matching the asymptotic work of P-MERGE-AUX and yielding $\Theta(n)$ work overall for P-MERGE.

Analysis of parallel merge sort

The "heavy lifting" is done. Now that we have determined the work and span of P-MERGE, we can analyze P-MERGE-SORT. Let $T_1(n)$ and $T_\infty(n)$ be the work and span, respectively, of P-MERGE-SORT on an array of n elements. The call to P-MERGE in line 10 of P-MERGE-SORT dominates the costs of lines 1–3, for both work and span. Thus we obtain the recurrence

$$T_1(n) = 2T_1(n/2) + \Theta(n)$$

for the work of P-MERGE-SORT, and we obtain the recurrence

$$T_\infty(n) = T_\infty(n/2) + \Theta(\lg^2 n)$$

for its span. The work recurrence has solution $T_1(n) = \Theta(n \lg n)$ by case 2 of the master theorem with $k = 0$. The span recurrence has solution $T_\infty(n) = \Theta(\lg^3 n)$, also by case 2 of the master theorem, but with $k = 2$.

Parallel merging gives P-MERGE-SORT a parallelism advantage over P-NAIVE-MERGE-SORT. The parallelism of P-NAIVE-MERGE-SORT, which calls the serial MERGE procedure, is only $\Theta(\lg n)$. For P-MERGE-SORT, the parallelism is

$$\begin{aligned} T_1(n)/T_\infty(n) &= \Theta(n \lg n)/\Theta(\lg^3 n) \\ &= \Theta(n/\lg^2 n) \,, \end{aligned}$$

which is much better, both in theory and in practice. A good implementation in practice would sacrifice some parallelism by coarsening the base case in order to reduce the constants hidden by the asymptotic notation. For example, you could switch to an efficient serial sort, perhaps quicksort, when the number of elements to be sorted is sufficiently small.

Exercises

26.3-1

Explain how to coarsen the base case of P-MERGE.

26.3-2

Instead of finding a median element in the larger subarray, as P-MERGE does, suppose that the merge procedure finds a median of all the elements in the two sorted subarrays using the result of Exercise 9.3-10. Give pseudocode for an efficient parallel merging procedure that uses this median-finding procedure. Analyze your algorithm.

26.3-3

Give an efficient parallel algorithm for partitioning an array around a pivot, as is done by the PARTITION procedure on page 184. You need not partition the array in place. Make your algorithm as parallel as possible. Analyze your algorithm. (*Hint:* You might need an auxiliary array and might need to make more than one pass over the input elements.)

26.3-4

Give a parallel version of FFT on page 890. Make your implementation as parallel as possible. Analyze your algorithm.

★ *26.3-5*

Show how to parallelize SELECT from Section 9.3. Make your implementation as parallel as possible. Analyze your algorithm.

Problems

26-1 Implementing parallel loops using recursive spawning

Consider the parallel procedure SUM-ARRAYS for performing pairwise addition on n-element arrays $A[1:n]$ and $B[1:n]$, storing the sums in $C[1:n]$.

```
SUM-ARRAYS(A, B, C, n)
1  parallel for i = 1 to n
2      C[i] = A[i] + B[i]
```

a. Rewrite the parallel loop in SUM-ARRAYS using recursive spawning in the manner of P-MAT-VEC-RECURSIVE. Analyze the parallelism.

Consider another implementation of the parallel loop in SUM-ARRAYS given by the procedure SUM-ARRAYS′, where the value *grain-size* must be specified.

```
SUM-ARRAYS'(A, B, C, n)
1  grain-size = ?                  // to be determined
2  r = ⌈n/grain-size⌉
3  for k = 0 to r − 1
4      spawn ADD-SUBARRAY(A, B, C, k · grain-size + 1,
                                   min{(k + 1) · grain-size, n})
5  sync

ADD-SUBARRAY(A, B, C, i, j)
1  for k = i to j
2      C[k] = A[k] + B[k]
```

b. Suppose that you set $grain\text{-}size = 1$. What is the resulting parallelism?

c. Give a formula for the span of SUM-ARRAYS′ in terms of n and *grain-size*. Derive the best value for *grain-size* to maximize parallelism.

26-2 Avoiding a temporary matrix in recursive matrix multiplication

The P-MATRIX-MULTIPLY-RECURSIVE procedure on page 772 must allocate a temporary matrix D of size $n \times n$, which can adversely affect the constants hidden by the Θ-notation. The procedure has high parallelism, however: $\Theta(n^3/\log^2 n)$.

For example, ignoring the constants in the Θ-notation, the parallelism for multiplying 1000×1000 matrices comes to approximately $1000^3/10^2 = 10^7$, since $\lg 1000 \approx 10$. Most parallel computers have far fewer than 10 million processors.

a. Parallelize MATRIX-MULTIPLY-RECURSIVE without using temporary matrices so that it retains its $\Theta(n^3)$ work. (*Hint:* Spawn the recursive calls, but insert a **sync** in a judicious location to avoid races.)

b. Give and solve recurrences for the work and span of your implementation.

c. Analyze the parallelism of your implementation. Ignoring the constants in the Θ-notation, estimate the parallelism on 1000×1000 matrices. Compare with the parallelism of P-MATRIX-MULTIPLY-RECURSIVE, and discuss whether the trade-off would be worthwhile.

26-3 Parallel matrix algorithms

Before attempting this problem, it may be helpful to read Chapter 28.

a. Parallelize the LU-DECOMPOSITION procedure on page 827 by giving pseudocode for a parallel version of this algorithm. Make your implementation as parallel as possible, and analyze its work, span, and parallelism.

b. Do the same for LUP-DECOMPOSITION on page 830.

c. Do the same for LUP-SOLVE on page 824.

d. Using equation (28.14) on page 835, write pseudocode for a parallel algorithm to invert a symmetric positive-definite matrix. Make your implementation as parallel as possible, and analyze its work, span, and parallelism.

26-4 Parallel reductions and scan (prefix) computations

A ***$\otimes$-reduction*** of an array $x[1:n]$, where $\otimes$ is an associative operator, is the value $y = x[1] \otimes x[2] \otimes \cdots \otimes x[n]$. The REDUCE procedure computes the $\otimes$-reduction of a subarray $x[i:j]$ serially.

```
REDUCE(x, i, j)
1  y = x[i]
2  for k = i + 1 to j
3       y = y ⊗ x[k]
4  return y
```

a. Design and analyze a parallel algorithm P-REDUCE that uses recursive spawning to perform the same function with $\Theta(n)$ work and $\Theta(\lg n)$ span.

A related problem is that of computing a ***$\otimes$-scan***, sometimes called a ***$\otimes$-prefix computation***, on an array $x[1:n]$, where $\otimes$ is once again an associative operator. The $\otimes$-scan, implemented by the serial procedure SCAN, produces the array $y[1:n]$ given by

$$\begin{aligned}
y[1] &= x[1] , \\
y[2] &= x[1] \otimes x[2] , \\
y[3] &= x[1] \otimes x[2] \otimes x[3] , \\
&\vdots \\
y[n] &= x[1] \otimes x[2] \otimes x[3] \otimes \cdots \otimes x[n] ,
\end{aligned}$$

that is, all prefixes of the array x "summed" using the $\otimes$ operator.

```
SCAN(x, n)
1  let y[1:n] be a new array
2  y[1] = x[1]
3  for i = 2 to n
4       y[i] = y[i − 1] ⊗ x[i]
5  return y
```

Parallelizing SCAN is not straightforward. For example, simply changing the **for** loop to a **parallel for** loop would create races, since each iteration of the loop body depends on the previous iteration. The procedures P-SCAN-1 and P-SCAN-1-AUX perform the $\otimes$-scan in parallel, albeit inefficiently.

```
P-SCAN-1(x, n)
1  let y[1:n] be a new array
2  P-SCAN-1-AUX(x, y, 1, n)
3  return y

P-SCAN-1-AUX(x, y, i, j)
1  parallel for l = i to j
2       y[l] = P-REDUCE(x, 1, l)
```

b. Analyze the work, span, and parallelism of P-SCAN-1.

The procedures P-SCAN-2 and P-SCAN-2-AUX use recursive spawning to perform a more efficient $\otimes$-scan.

```
P-SCAN-2(x, n)
1  let y[1 : n] be a new array
2  P-SCAN-2-AUX(x, y, 1, n)
3  return y

P-SCAN-2-AUX(x, y, i, j)
1  if i == j
2       y[i] = x[i]
3  else k = ⌊(i + j)/2⌋
4       spawn P-SCAN-2-AUX(x, y, i, k)
5       P-SCAN-2-AUX(x, y, k + 1, j)
6       sync
7       parallel for l = k + 1 to j
8            y[l] = y[k] ⊗ y[l]
```

c. Argue that P-SCAN-2 is correct, and analyze its work, span, and parallelism.

To improve on both P-SCAN-1 and P-SCAN-2, perform the $\otimes$-scan in two distinct passes over the data. The first pass gathers the terms for various contiguous subarrays of x into a temporary array t, and the second pass uses the terms in t to compute the final result y. The pseudocode in the procedures P-SCAN-3, P-SCAN-UP, and P-SCAN-DOWN on the facing page implements this strategy, but certain expressions have been omitted.

d. Fill in the three missing expressions in line 8 of P-SCAN-UP and lines 5 and 6 of P-SCAN-DOWN. Argue that with the expressions you supplied, P-SCAN-3 is correct. (*Hint:* Prove that the value v passed to P-SCAN-DOWN(v, x, t, y, i, j) satisfies $v = x[1] \otimes x[2] \otimes \cdots \otimes x[i-1]$.)

e. Analyze the work, span, and parallelism of P-SCAN-3.

f. Describe how to rewrite P-SCAN-3 so that it doesn't require the use of the temporary array t.

★ ***g.*** Give an algorithm P-SCAN-4(x, n) for a scan that operates in place. It should place its output in x and require only constant auxiliary storage.

h. Describe an efficient parallel algorithm that uses a $+$-scan to determine whether a string of parentheses is well formed. For example, the string (( ) ( )) ( )

is well formed, but the string (( ))) (( ) is not. (*Hint:* Interpret (as a 1 and) as a -1, and then perform a $+$-scan.)

```
P-SCAN-3(x, n)
1  let y[1 : n] and t[1 : n] be new arrays
2  y[1] = x[1]
3  if n > 1
4       P-SCAN-UP(x, t, 2, n)
5       P-SCAN-DOWN(x[1], x, t, y, 2, n)
6  return y

P-SCAN-UP(x, t, i, j)
1  if i == j
2       return x[i]
3  else
4       k = ⌊(i + j)/2⌋
5       t[k] = spawn P-SCAN-UP(x, t, i, k)
6       right = P-SCAN-UP(x, t, k + 1, j)
7       sync
8       return _____          // fill in the blank

P-SCAN-DOWN(v, x, t, y, i, j)
1  if i == j
2       y[i] = v ⊗ x[i]
3  else
4       k = ⌊(i + j)/2⌋
5       spawn P-SCAN-DOWN(_____, x, t, y, i, k)     // fill in the blank
6       P-SCAN-DOWN(_____, x, t, y, k + 1, j)       // fill in the blank
7       sync
```

26-5 Parallelizing a simple stencil calculation

Computational science is replete with algorithms that require the entries of an array to be filled in with values that depend on the values of certain already computed neighboring entries, along with other information that does not change over the course of the computation. The pattern of neighboring entries does not change during the computation and is called a ***stencil***. For example, Section 14.4 presents a stencil algorithm to compute a longest common subsequence, where the value in entry $c[i, j]$ depends only on the values in $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$,

as well as the elements x_i and y_j within the two sequences given as inputs. The input sequences are fixed, but the algorithm fills in the two-dimensional array c so that it computes entry $c[i, j]$ after computing all three entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$.

This problem examines how to use recursive spawning to parallelize a simple stencil calculation on an $n \times n$ array A in which the value placed into entry $A[i, j]$ depends only on values in $A[i', j']$, where $i' \le i$ and $j' \le j$ (and of course, $i' \ne i$ or $j' \ne j$). In other words, the value in an entry depends only on values in entries that are above it and/or to its left, along with static information outside of the array. Furthermore, we assume throughout this problem that once the entries upon which $A[i, j]$ depends have been filled in, the entry $A[i, j]$ can be computed in $\Theta(1)$ time (as in the LCS-LENGTH procedure of Section 14.4).

Partition the $n \times n$ array A into four $n/2 \times n/2$ subarrays as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}. \tag{26.9}$$

You can immediately fill in subarray A_{11} recursively, since it does not depend on the entries in the other three subarrays. Once the computation of A_{11} finishes, you can fill in A_{12} and A_{21} recursively in parallel, because although they both depend on A_{11}, they do not depend on each other. Finally, you can fill in A_{22} recursively.

a. Give parallel pseudocode that performs this simple stencil calculation using a divide-and-conquer algorithm SIMPLE-STENCIL based on the decomposition (26.9) and the discussion above. (Don't worry about the details of the base case, which depends on the specific stencil.) Give and solve recurrences for the work and span of this algorithm in terms of n. What is the parallelism?

b. Modify your solution to part (a) to divide an $n \times n$ array into nine $n/3 \times n/3$ subarrays, again recursing with as much parallelism as possible. Analyze this algorithm. How much more or less parallelism does this algorithm have compared with the algorithm from part (a)?

c. Generalize your solutions to parts (a) and (b) as follows. Choose an integer $b \ge 2$. Divide an $n \times n$ array into b^2 subarrays, each of size $n/b \times n/b$, recursing with as much parallelism as possible. In terms of n and b, what are the work, span, and parallelism of your algorithm? Argue that, using this approach, the parallelism must be $o(n)$ for any choice of $b \ge 2$. (*Hint:* For this argument, show that the exponent of n in the parallelism is strictly less than 1 for any choice of $b \ge 2$.)

d. Give pseudocode for a parallel algorithm for this simple stencil calculation that achieves $\Theta(n/\lg n)$ parallelism. Argue using notions of work and span that

the problem has $\Theta(n)$ inherent parallelism. Unfortunately, simple fork-join parallelism does not let you achieve this maximal parallelism.

26-6 Randomized parallel algorithms

Like serial algorithms, parallel algorithms can employ random-number generators. This problem explores how to adapt the measures of work, span, and parallelism to handle the expected behavior of randomized task-parallel algorithms. It also asks you to design and analyze a parallel algorithm for randomized quicksort.

a. Explain how to modify the work law (26.2), span law (26.3), and greedy scheduler bound (26.4) to work with expectations when T_P, T_1, and T_∞ are all random variables.

b. Consider a randomized parallel algorithm for which 1% of the time, $T_1 = 10^4$ and $T_{10,000} = 1$, but for the remaining 99% of the time, $T_1 = T_{10,000} = 10^9$. Argue that the ***speedup*** of a randomized parallel algorithm should be defined as $\mathrm{E}[T_1]/\mathrm{E}[T_P]$, rather than $\mathrm{E}[T_1/T_P]$.

c. Argue that the ***parallelism*** of a randomized task-parallel algorithm should be defined as the ratio $\mathrm{E}[T_1]/\mathrm{E}[T_\infty]$.

d. Parallelize the RANDOMIZED-QUICKSORT algorithm on page 192 by using recursive spawning to produce P-RANDOMIZED-QUICKSORT. (Do not parallelize RANDOMIZED-PARTITION.)

e. Analyze your parallel algorithm for randomized quicksort. (*Hint:* Review the analysis of RANDOMIZED-SELECT on page 230.)

f. Parallelize RANDOMIZED-SELECT on page 230. Make your implementation as parallel as possible. Analyze your algorithm. (*Hint:* Use the partitioning algorithm from Exercise 26.3-3.)

Chapter notes

Parallel computers and algorithmic models for parallel programming have been around in various forms for years. Prior editions of this book included material on sorting networks and the PRAM (Parallel Random-Access Machine) model. The data-parallel model [58, 217] is another popular algorithmic programming model, which features operations on vectors and matrices as primitives. The notion of sequential consistency is due to Lamport [275].

Graham [197] and Brent [71] showed that there exist schedulers achieving the bound of Theorem 26.1. Eager, Zahorjan, and Lazowska [129] showed that

any greedy scheduler achieves this bound and proposed the methodology of using work and span (although not by those names) to analyze parallel algorithms. Blelloch [57] developed an algorithmic programming model based on work and span (which he called "depth") for data-parallel programming. Blumofe and Leiserson [63] gave a distributed scheduling algorithm for task-parallel computations based on randomized "work-stealing" and showed that it achieves the bound $\mathrm{E}[T_P] \leq T_1/P + O(T_\infty)$. Arora, Blumofe, and Plaxton [20] and Blelloch, Gibbons, and Matias [61] also provided provably good algorithms for scheduling task-parallel computations. The recent literature contains many algorithms and strategies for scheduling parallel programs.

The parallel pseudocode and programming model were influenced by Cilk [290, 291, 383, 396]. The open-source project OpenCilk (www.opencilk.org) provides Cilk programming as an extension to the C and C++ programming languages. All of the parallel algorithms in this chapter can be coded straightforwardly in Cilk.

Concerns about nondeterministic parallel programs were expressed by Lee [281] and Bocchino, Adve, Adve, and Snir [64]. The algorithms literature contains many algorithmic strategies (see, for example, [60, 85, 118, 140, 160, 282, 283, 412, 461]) for detecting races and extending the fork-join model to avoid or safely embrace various kinds of nondeterminism. Blelloch, Fineman, Gibbons, and Shun [59] showed that deterministic parallel algorithms can often be as fast as, or even faster than, their nondeterministic counterparts.

Several of the parallel algorithms in this chapter appeared in unpublished lecture notes by C. E. Leiserson and H. Prokop and were originally implemented in Cilk. The parallel merge-sorting algorithm was inspired by an algorithm due to Akl [12].