the problem size increases. Then, there are some problems for which we can invest increasing amounts of computation time in return for increasingly better approximate solutions. This chapter illustrates these possibilities with the vertex-cover problem (unweighted and weighted versions), an optimization version of 3-CNF satisfiability, the traveling-salesman problem, the set-covering problem, and the subset-sum problem.

27 Multithreaded Algorithms

The vast majority of algorithms in this book are ***serial algorithms*** suitable for running on a uniprocessor computer in which only one instruction executes at a time. In this chapter, we shall extend our algorithmic model to encompass ***parallel algorithms***, which can run on a multiprocessor computer that permits multiple instructions to execute concurrently. In particular, we shall explore the elegant model of dynamic multithreaded algorithms, which are amenable to algorithmic design and analysis, as well as to efficient implementation in practice.

Parallel computers—computers with multiple processing units—have become increasingly common, and they span a wide range of prices and performance. Relatively inexpensive desktop and laptop ***chip multiprocessors*** contain a single ***multicore*** integrated-circuit chip that houses multiple processing "cores," each of which is a full-fledged processor that can access a common memory. At an intermediate price/performance point are clusters built from individual computers—often simple PC-class machines—with a dedicated network interconnecting them. The highest-priced machines are supercomputers, which often use a combination of custom architectures and custom networks to deliver the highest performance in terms of instructions executed per second.

Multiprocessor computers have been around, in one form or another, for decades. Although the computing community settled on the random-access machine model for serial computing early on in the history of computer science, no single model for parallel computing has gained as wide acceptance. A major reason is that vendors have not agreed on a single architectural model for parallel computers. For example, some parallel computers feature ***shared memory***, where each processor can directly access any location of memory. Other parallel computers employ ***distributed memory***, where each processor's memory is private, and an explicit message must be sent between processors in order for one processor to access the memory of another. With the advent of multicore technology, however, every new laptop and desktop machine is now a shared-memory parallel computer,

and the trend appears to be toward shared-memory multiprocessing. Although time will tell, that is the approach we shall take in this chapter.

One common means of programming chip multiprocessors and other shared-memory parallel computers is by using ***static threading***, which provides a software abstraction of "virtual processors," or ***threads***, sharing a common memory. Each thread maintains an associated program counter and can execute code independently of the other threads. The operating system loads a thread onto a processor for execution and switches it out when another thread needs to run. Although the operating system allows programmers to create and destroy threads, these operations are comparatively slow. Thus, for most applications, threads persist for the duration of a computation, which is why we call them "static."

Unfortunately, programming a shared-memory parallel computer directly using static threads is difficult and error-prone. One reason is that dynamically partitioning the work among the threads so that each thread receives approximately the same load turns out to be a complicated undertaking. For any but the simplest of applications, the programmer must use complex communication protocols to implement a scheduler to load-balance the work. This state of affairs has led toward the creation of ***concurrency platforms***, which provide a layer of software that coordinates, schedules, and manages the parallel-computing resources. Some concurrency platforms are built as runtime libraries, but others provide full-fledged parallel languages with compiler and runtime support.

Dynamic multithreaded programming

One important class of concurrency platform is ***dynamic multithreading***, which is the model we shall adopt in this chapter. Dynamic multithreading allows programmers to specify parallelism in applications without worrying about communication protocols, load balancing, and other vagaries of static-thread programming. The concurrency platform contains a scheduler, which load-balances the computation automatically, thereby greatly simplifying the programmer's chore. Although the functionality of dynamic-multithreading environments is still evolving, almost all support two features: nested parallelism and parallel loops. Nested parallelism allows a subroutine to be "spawned," allowing the caller to proceed while the spawned subroutine is computing its result. A parallel loop is like an ordinary **for** loop, except that the iterations of the loop can execute concurrently.

These two features form the basis of the model for dynamic multithreading that we shall study in this chapter. A key aspect of this model is that the programmer needs to specify only the logical parallelism within a computation, and the threads within the underlying concurrency platform schedule and load-balance the computation among themselves. We shall investigate multithreaded algorithms written for

this model, as well how the underlying concurrency platform can schedule computations efficiently.

Our model for dynamic multithreading offers several important advantages:

- It is a simple extension of our serial programming model. We can describe a multithreaded algorithm by adding to our pseudocode just four "concurrency" keywords: **parallel**, **spawn**, **sync**, and **new**. Moreover, if we delete these concurrency keywords from the multithreaded pseudocode, the resulting text is serial pseudocode for the same problem, which we call the "serialization" of the multithreaded algorithm.
- It provides a theoretically clean way to quantify parallelism based on the notions of "work" and "span."
- Many multithreaded algorithms involving nested parallelism follow naturally from the divide-and-conquer paradigm. Moreover, just as serial divide-and-conquer algorithms lend themselves to analysis by solving recurrences, so do multithreaded algorithms.
- The model is faithful to how parallel-computing practice is evolving. A growing number of concurrency platforms support one variant or another of dynamic multithreading, including Cilk [51, 118], Cilk++ [71], OpenMP [59], Task Parallel Library [230], and Threading Building Blocks [292].

Section 27.1 introduces the dynamic multithreading model and presents the metrics of work, span, and parallelism, which we shall use to analyze multithreaded algorithms. Section 27.2 investigates how to multiply matrices with multithreading, and Section 27.3 tackles the tougher problem of multithreading merge sort.

27.1 The basics of dynamic multithreading

We shall begin our exploration of dynamic multithreading using the example of computing Fibonacci numbers recursively. Recall that the Fibonacci numbers are defined by recurrence (3.22):

$$\begin{aligned} F_0 &= 0 , \\ F_1 &= 1 , \\ F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2 . \end{aligned}$$

Here is a simple, recursive, serial algorithm to compute the nth Fibonacci number:

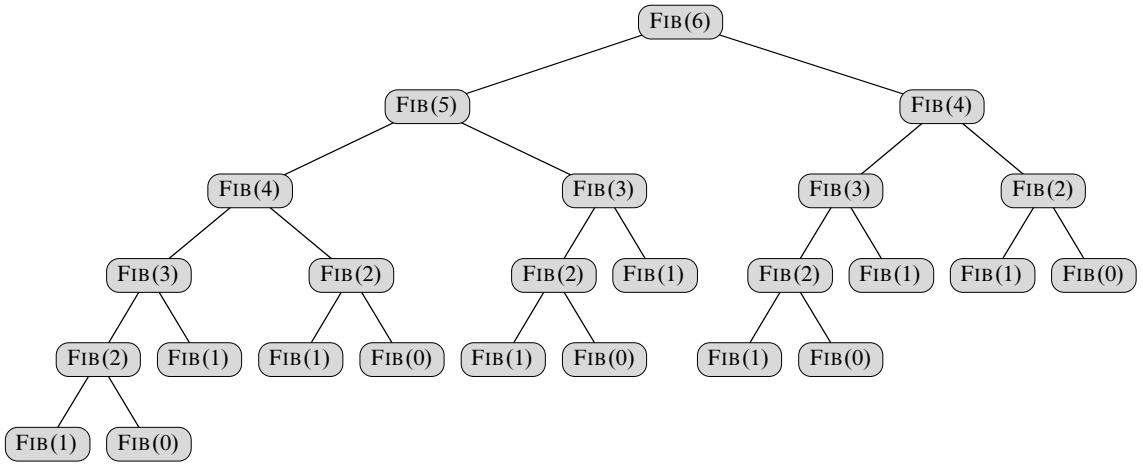


Figure 27.1 The tree of recursive procedure instances when computing FIB(6). Each instance of FIB with the same argument does the same work to produce the same result, providing an inefficient but interesting way to compute Fibonacci numbers.

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

You would not really want to compute large Fibonacci numbers this way, because this computation does much repeated work. Figure 27.1 shows the tree of recursive procedure instances that are created when computing F_6. For example, a call to FIB(6) recursively calls FIB(5) and then FIB(4). But, the call to FIB(5) also 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, the second call to FIB(4) replicates the work that the first call performs.

Let $T(n)$ denote the running time of FIB(n). Since 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 show using the substitution method. For an inductive hypothesis, assume that $T(n) \le a F_n - b$, where $a > 1$ and $b > 0$ are constants. Substituting, we obtain

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

if we choose b large enough to dominate the constant in the $\Theta(1)$. We can then choose a large enough to satisfy the initial condition. The analytical bound

$$T(n) = \Theta(\phi^n)\,, \tag{27.1}$$

where $\phi = (1 + \sqrt{5})/2$ is the golden ratio, now follows from equation (3.25). Since F_n grows exponentially in n, this procedure is a particularly slow way to compute Fibonacci numbers. (See Problem 31-3 for much faster ways.)

Although the FIB procedure is a poor way to compute Fibonacci numbers, it makes a good example for illustrating key concepts in the analysis of multithreaded algorithms. Observe that within FIB(n), the two recursive calls in lines 3 and 4 to FIB($n-1$) and FIB($n-2$), respectively, are independent of each other: they could be called in either order, and the computation performed by one in no way affects the other. Therefore, the two recursive calls can run in parallel.

We augment our pseudocode to indicate parallelism by adding the ***concurrency keywords*** **spawn** and **sync**. Here is how we can rewrite the FIB procedure to use dynamic multithreading:

```
P-FIB(n)
1  if n ≤ 1
2      return n
3  else x = spawn P-FIB(n − 1)
4      y = P-FIB(n − 2)
5      sync
6      return x + y
```

Notice that if we delete the concurrency keywords **spawn** and **sync** from P-FIB, the resulting pseudocode text is identical to FIB (other than renaming the procedure in the header and in the two recursive calls). We define the ***serialization*** of a multithreaded algorithm to be the serial algorithm that results from deleting the multithreaded keywords: **spawn**, **sync**, and when we examine parallel loops, **parallel** and **new**. Indeed, our multithreaded pseudocode has the nice property that a serialization is always ordinary serial pseudocode to solve the same problem.

Nested parallelism occurs when the keyword **spawn** precedes a procedure call, as in line 3. 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 complete, as would normally 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 concurrently with its spawned children, only that it *may*. The concurrency 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 concurrently by assigning them to available processors as the computation unfolds. We shall discuss the theory behind schedulers shortly.

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 complete before proceeding to the statement after the **sync**. In the P-FIB procedure, a **sync** is required before the **return** statement in line 6 to avoid the anomaly that would occur if x and y were summed before x was computed. In addition to explicit synchronization provided by the **sync** statement, every procedure executes a **sync** implicitly before it returns, thus ensuring that all its children terminate before it does.

A model for multithreaded execution

It helps to think of a ***multithreaded computation***—the set of runtime instructions executed by a processor on behalf of a multithreaded program—as a directed acyclic graph $G = (V, E)$, called a ***computation dag***. As an example, Figure 27.2 shows the computation dag that results from computing P-FIB(4). Conceptually, the vertices in V are instructions, and the edges in E represent dependencies between instructions, where $(u, v) \in E$ means that instruction u must execute before instruction v. For convenience, however, if a chain of instructions contains no parallel control (no **spawn**, **sync**, or **return** from a spawn—via either an explicit **return** statement or the return that happens implicitly upon reaching the end of a procedure), we may group them into a single ***strand***, each of which represents one or more instructions. Instructions involving parallel control are not included in strands, but are represented in the structure of the dag. For example, if a strand has two successors, one of them must have been spawned, and a strand with multiple predecessors indicates 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 control.

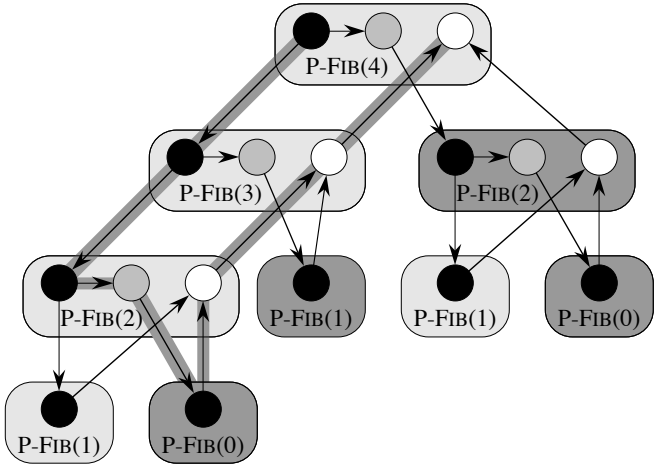


Figure 27.2 A directed acyclic graph representing the computation of P-FIB(4). Each circle represents one strand, with black circles representing either base cases or the part of the procedure (instance) up to the spawn of P-FIB($n-1$) in line 3, shaded circles representing the part of the procedure 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 part of the procedure after the **sync** where it sums x and y up to the point where it returns the result. Each group of strands belonging to the same procedure is surrounded by a rounded rectangle, lightly shaded for spawned procedures and heavily shaded for called procedures. Spawn edges and call edges point downward, continuation edges point horizontally to the right, and return edges point upward. Assuming that each strand takes unit time, the work equals 17 time units, since there are 17 strands, and the span is 8 time units, since the critical path—shown with shaded edges—contains 8 strands.

If G has a directed path from strand u to strand v, we say that the two strands are ***(logically) in series***. Otherwise, strands u and v are ***(logically) in parallel***.

We can picture a multithreaded computation as a dag of strands embedded in a tree of procedure instances. For example, Figure 27.1 shows the tree of procedure instances for P-FIB(6) without the detailed structure showing strands. Figure 27.2 zooms in on a section of that tree, showing the strands that constitute each procedure. All directed edges connecting strands run either within a procedure or along undirected edges in the procedure tree.

We can classify the edges of a computation dag to indicate the kind of dependencies between the various strands. A ***continuation edge*** (u, u'), drawn horizontally in Figure 27.2, connects a strand u to its successor u' within the same procedure instance. When a strand u spawns a strand v, the dag contains a ***spawn edge*** (u, v), which points downward in the figure. ***Call edges***, representing normal procedure calls, also point downward. Strand u spawning strand v differs from u calling v in that a spawn induces a horizontal continuation edge from u to the strand u' fol-

lowing u in its procedure, indicating that u' is free to execute at the same time as v, whereas a call induces no such edge. When a strand u returns to its calling procedure and x is the strand immediately following the next **sync** in the calling procedure, the computation dag contains ***return edge*** (u, x), which points upward. A computation starts with a single ***initial strand***—the black vertex in the procedure labeled P-FIB(4) in Figure 27.2—and ends with a single ***final strand***—the white vertex in the procedure labeled P-FIB(4).

We shall study the execution of multithreaded algorithms on an ***ideal parallel computer***, which consists of a set of processors and a ***sequentially consistent*** shared memory. Sequential consistency means that the shared memory, which may in reality be performing many loads and stores from the processors at the same time, produces the same results as if at each step, exactly one instruction from one of the processors is executed. That is, the memory behaves as if the instructions were executed sequentially according to some global linear order that preserves the individual orders in which each processor issues its own instructions. For dynamic multithreaded computations, which are scheduled onto processors automatically by the concurrency platform, the shared memory behaves as if the multithreaded computation's instructions were interleaved to produce a linear order that preserves the partial order of the computation dag. Depending on scheduling, the ordering could differ from one run of the program to another, but the behavior of any execution can be understood by assuming that the instructions are executed in some linear order consistent with the computation dag.

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 shall define precisely in a moment), the overhead of scheduling is generally minimal in practice.

Performance measures

We can gauge the theoretical efficiency of a multithreaded algorithm by using two metrics: "work" and "span." The ***work*** of a multithreaded 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. For a computation dag in which each strand takes unit time, the work is just the number of vertices in the dag. The ***span*** is the longest time to execute the strands along any path in the dag. Again, for a dag in which each strand takes unit time, the span equals the number of vertices on a longest or ***critical path*** in the dag. (Recall from Section 24.2 that we can find a critical path in a dag $G = (V, E)$ in $\Theta(V + E)$ time.) For example, the computation dag of Figure 27.2 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 multithreaded 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 multithreaded computation on P processors, we shall 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 multithreaded 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 PT_P work. Since the total work to do is T_1, we have $PT_P \geq T_1$. Dividing by P yields the ***work law***:

$$T_P \geq T_1/P \ . \tag{27.2}$$

- 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:

$$T_P \geq T_\infty \ . \tag{27.3}$$

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 is on P processors than on 1 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 P processors 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***, and when $T_1/T_P = P$, we have ***perfect linear speedup***.

The ratio T_1/T_∞ of the work to the span gives the ***parallelism*** of the multithreaded 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. Finally, and 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, the more processors we use beyond the parallelism, the less perfect the speedup.

As an example, consider the computation P-FIB(4) in Figure 27.2, 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 speedup is impossible, no matter how many processors we employ to execute the computation. For larger input sizes, however, we shall see that P-FIB(n) exhibits substantial parallelism.

We define the ***(parallel) slackness*** of a multithreaded 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. Thus, if the slackness is less than 1, we cannot hope to achieve perfect linear speedup, because $T_1/(PT_\infty) < 1$ and the span law imply that the speedup on P processors satisfies $T_1/T_P \leq T_1/T_\infty < P$. Indeed, as the slackness decreases from 1 toward 0, the speedup of the computation diverges further and further from perfect linear speedup. If the slackness is greater than 1, however, the work per processor is the limiting constraint. As we shall see, as the slackness increases from 1, a good scheduler can achieve closer and closer to perfect linear speedup.

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 multithreaded programming model provides no way to specify which strands to execute on which processors. Instead, we rely on the concurrency platform'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 concurrency platform's scheduler maps strands to processors directly.

A multithreaded scheduler must schedule the computation with no advance knowledge of when strands will be spawned or when they will complete—it must operate ***on-line***. Moreover, a good scheduler operates in a distributed fashion, where the threads implementing the scheduler cooperate to load-balance the computation. Provably good on-line, distributed schedulers exist, but analyzing them is complicated.

Instead, to keep our analysis simple, we shall investigate an on-line ***centralized*** scheduler, which knows the global state of the computation at any given time. In particular, we shall analyze ***greedy schedulers***, which assign as many strands to processors as possible in each time step. If at least P strands are ready to execute during a time step, we say that the step is a ***complete step***, and a greedy scheduler assigns any P of the ready strands to processors. Otherwise, fewer than P strands are ready to execute, in which case we say that the step is an ***incomplete step***, and the scheduler assigns each ready strand to its own processor.

From the work law, the best running time we can hope for on P processors is $T_P = T_1/P$, and from the span law the best we can hope for is $T_P = T_\infty$. 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 27.1

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

$$T_P \le T_1/P + T_\infty \; . \tag{27.4}$$

Proof We start by considering the complete steps. In each complete step, the P processors together perform a total of P work. Suppose for the purpose of contradiction that the number of complete steps is strictly greater than $\lfloor T_1/P \rfloor$. Then, the total work of the complete steps is at least

$$\begin{aligned} P \cdot (\lfloor T_1/P \rfloor + 1) &= P \lfloor T_1/P \rfloor + P \\ &= T_1 - (T_1 \bmod P) + P \quad \text{(by equation (3.8))} \\ &> T_1 \quad \text{(by inequality (3.9))} \; . \end{aligned}$$

Thus, we obtain the contradiction that the P processors would perform more work than the computation requires, which allows us to conclude that the number of complete steps is at most $\lfloor T_1/P \rfloor$.

Now, consider an incomplete step. Let G be the dag representing the entire computation, and without loss of generality, assume that each strand takes unit time. (We can replace each longer strand by a chain of unit-time strands.) Let G' be the subgraph of G that has yet to be executed at the start of the incomplete step, and let G'' be the subgraph remaining to be executed after the incomplete step. A longest path in a dag must necessarily start at a vertex with in-degree 0. Since an incomplete step of a greedy scheduler executes all strands with in-degree 0 in G', the length of a longest path in G'' must be 1 less than the length of a longest path in G'. In other words, an incomplete step decreases the span of the unexecuted dag by 1. Hence, the number of incomplete steps is at most T_∞.

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

The following corollary to Theorem 27.1 shows that a greedy scheduler always performs well.

Corollary 27.2
The running time T_P of any multithreaded computation scheduled by a greedy scheduler on an ideal parallel computer with P processors 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 (27.2) and (27.3)—give us $T_P^* \geq \max(T_1/P, T_\infty)$, Theorem 27.1 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 multithreaded computation as the slackness grows.

Corollary 27.3
Let T_P be the running time of a multithreaded 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 $P \ll T_1/T_\infty$, we have $T_P \approx T_1/P$, or equivalently, a speedup of approximately P.

Proof If we suppose that $P \ll T_1/T_\infty$, then we also have $T_\infty \ll T_1/P$, and hence Theorem 27.1 gives us $T_P \leq T_1/P + T_\infty \approx T_1/P$. Since the work law (27.2) dictates that $T_P \geq T_1/P$, we conclude that $T_P \approx T_1/P$, or equivalently, that the speedup is $T_1/T_P \approx P$. ■

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 (27.4), 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 27-2 shows, sometimes by reducing extreme parallelism, we can obtain algorithms that are better with respect to other concerns and which still scale up well on reasonable numbers of processors.

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 27.3 The work and span of composed subcomputations. **(a)** When two subcomputations 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 subcomputations 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.

Analyzing multithreaded algorithms

We now have all the tools we need to analyze multithreaded algorithms and provide good bounds on their running times on various numbers 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 serialization of the multithreaded algorithm—which you should already be familiar with, since that is what most of this textbook is about! Analyzing the span is more interesting, but generally no harder once you get the hang of it. We shall 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 original FIB procedure is essentially the serialization of P-FIB, and hence $T_1(n) = T(n) = \Theta(\phi^n)$ from equation (27.1).

Figure 27.3 illustrates how to analyze the span. If two subcomputations are joined in series, their spans add to form the span of their composition, whereas if they are joined in parallel, the span of their composition is the maximum of the spans of the two subcomputations. For 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 parallelism of P-FIB(n) is $T_1(n)/T_\infty(n) = \Theta(\phi^n/n)$, which grows dramatically as n gets large. Thus, 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 all of whose iterations can operate in parallel. As we shall see, we can parallelize such loops using the **spawn** and **sync** keywords, but it is much more convenient to specify directly that the iterations of such loops can run concurrently. Our pseudocode provides this functionality via the **parallel** concurrency keyword, which precedes the **for** keyword in a **for** loop statement.

As an example, consider the problem of multiplying an $n \times n$ 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 $i = 1, 2, \ldots, n$. We can perform matrix-vector multiplication by computing all the entries of y in parallel as follows:

```
MAT-VEC(A, x)
1  n = A.rows
2  let y be a new vector of length n
3  parallel for i = 1 to n
4       y_i = 0
5  parallel for i = 1 to n
6       for new j = 1 to n
7            y_i = y_i + a_ij x_j
8  return y
```

In this code, the **parallel for** keywords in lines 3 and 5 indicate that the iterations of the respective loops may be run concurrently. The **new** keyword in line 6 indicates that a new variable j should be allocated for each iteration of i, rather than reusing the same variable, precluding different iterations of i from attempting to update the same variable j and causing a "race condition," which we shall examine in more detail starting on page 787.

A compiler can implement each **parallel for** loop as a divide-and-conquer subroutine using nested parallelism. For example, the **parallel for** loop in lines 5–7 can be implemented with the call MAT-VEC-MAIN-LOOP$(A, x, y, n, 1, n)$, where the compiler produces the auxiliary subroutine MAT-VEC-MAIN-LOOP as follows:

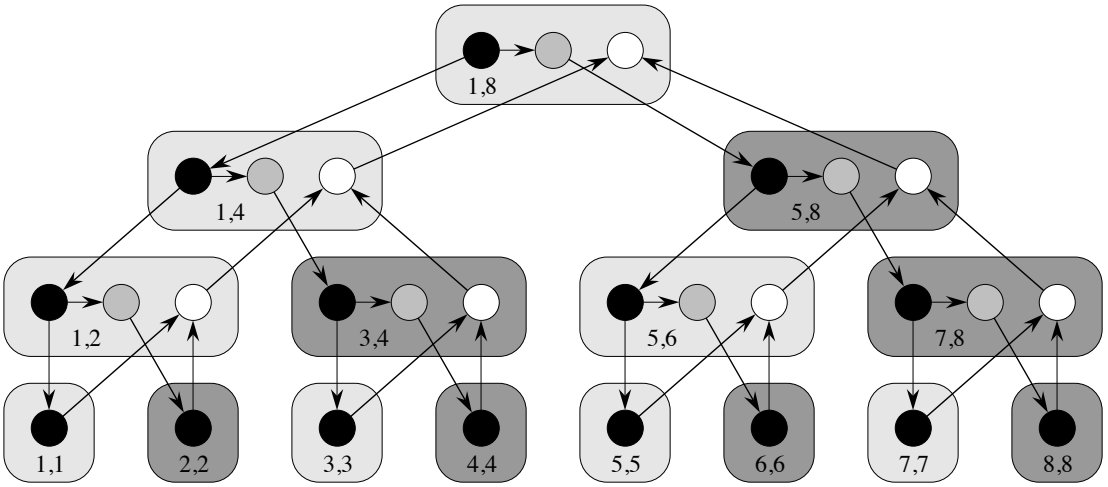


Figure 27.4 A dag representing the computation of MAT-VEC-MAIN-LOOP$(A, x, y, 8, 1, 8)$. The two numbers within each rounded rectangle give the values of the last two parameters (i and i' in the procedure header) in the invocation (spawn or call) of the procedure. The black circles represent strands corresponding to either the base case or the part of the procedure up to the spawn of MAT-VEC-MAIN-LOOP in line 5; the shaded circles represent strands corresponding to the part of the procedure that calls MAT-VEC-MAIN-LOOP in line 6 up to the **sync** in line 7, where it suspends until the spawned subroutine in line 5 returns; and the white circles represent strands corresponding to the (negligible) part of the procedure after the **sync** up to the point where it returns.

```
MAT-VEC-MAIN-LOOP(A, x, y, n, i, i')
1  if i == i'
2       for j = 1 to n
3            y_i = y_i + a_ij x_j
4  else mid = ⌊(i + i')/2⌋
5       spawn MAT-VEC-MAIN-LOOP(A, x, y, n, i, mid)
6       MAT-VEC-MAIN-LOOP(A, x, y, n, mid + 1, i')
7       sync
```

This code recursively spawns the first half of the iterations of the loop to execute in parallel with the second half of the iterations and then executes a **sync**, thereby creating a binary tree of execution where the leaves are individual loop iterations, as shown in Figure 27.4.

To calculate the work $T_1(n)$ of MAT-VEC on an $n \times n$ matrix, we simply compute the running time of its serialization, which we obtain by replacing the **parallel for** loops with ordinary **for** loops. Thus, we have $T_1(n) = \Theta(n^2)$, because the quadratic running time of the doubly nested loops in lines 5–7 dominates. This analysis

seems to ignore the overhead for recursive spawning in implementing the parallel loops, however. In fact, the overhead of recursive spawning does increase the work of a parallel loop compared with that of its serialization, 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 1 fewer than the number of leaves (see Exercise B.5-3). Each internal node performs constant work to divide the iteration range, and each leaf corresponds to an iteration of the loop, which takes at least constant time ($\Theta(n)$ time in this case). Thus, we can amortize the overhead of recursive spawning against the work of the iterations, contributing at most a constant factor to the overall work.

As a practical matter, dynamic-multithreading concurrency platforms sometimes ***coarsen*** the leaves of the recursion by executing several iterations in a single leaf, either automatically or under programmer control, thereby reducing the overhead of recursive spawning. This reduced overhead comes at the expense of also reducing the parallelism, however, but if the computation has sufficient parallel slackness, near-perfect linear speedup need not be sacrificed.

We must also account for the overhead of recursive spawning when analyzing the span of a parallel-loop construct. Since the depth of recursive calling is logarithmic in the number of iterations, for a parallel loop with n iterations in which the ith iteration has span $iter_\infty(i)$, the span is

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

For example, for MAT-VEC on an $n \times n$ matrix, the parallel initialization loop in lines 3–4 has span $\Theta(\lg n)$, because the recursive spawning dominates the constant-time work of each iteration. The span of the doubly nested loops in lines 5–7 is $\Theta(n)$, because each iteration of the outer **parallel for** loop contains n iterations of the inner (serial) **for** loop. The span of the remaining code in the procedure is constant, and thus the span is dominated by the doubly nested loops, yielding an overall span of $\Theta(n)$ for the whole procedure. Since the work is $\Theta(n^2)$, the parallelism is $\Theta(n^2)/\Theta(n) = \Theta(n)$. (Exercise 27.1-6 asks you to provide an implementation with even more parallelism.)

Race conditions

A multithreaded 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. Often, a multithreaded algorithm that is intended to be deterministic fails to be, because it contains a "determinacy race."

Race conditions are the bane of concurrency. Famous race bugs include the Therac-25 radiation therapy machine, which killed three people and injured sev-

eral others, and the North American Blackout of 2003, which left over 50 million people 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.

A ***determinacy race*** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write. The following procedure illustrates a race condition:

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

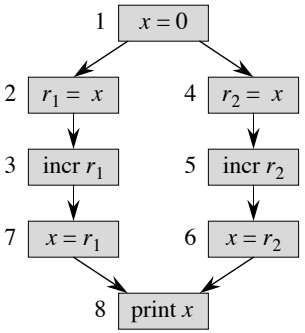
After initializing x to 0 in line 1, RACE-EXAMPLE creates two parallel strands, each of which increments x in line 3. Although it might seem that RACE-EXAMPLE should always print the value 2 (its serialization certainly does), it could instead print the value 1. Let's see how this anomaly might occur.

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

1. Read x from memory into one of the processor's registers.
2. Increment the value in the register.
3. Write the value in the register back into x in memory.

Figure 27.5(a) illustrates a computation dag representing the execution of RACE-EXAMPLE, with the strands broken down to individual instructions. Recall that since an ideal parallel computer supports sequential consistency, we can view the parallel execution of a multithreaded algorithm as an interleaving of instructions that respects the dependencies in the dag. Part (b) of the figure shows the values in an execution of the computation that elicits the anomaly. The value x is stored 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 reads 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 reads 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 2, as the serialization would print.

We can see what has happened. If the effect of the parallel execution were that processor 1 executed all its instructions before processor 2, the value 2 would be



(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 27.5 Illustration of the determinacy race in RACE-EXAMPLE. **(a)** A computation dag 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.

printed. Conversely, if the effect were that processor 2 executed all its instructions before processor 1, the value 2 would still be printed. When the instructions of the two processors execute at the same time, however, it is possible, as in this example execution, that one of the updates to x is lost.

Of course, many executions do not elicit the bug. For example, if the execution order were $\langle 1, 2, 3, 7, 4, 5, 6, 8\rangle$ or $\langle 1, 4, 5, 6, 2, 3, 7, 8\rangle$, we would get the correct result. That's the problem with determinacy races. Generally, most orderings produce correct results—such as any in which the instructions on the left execute before the instructions on the right, or vice versa. But some orderings generate improper results when the instructions interleave. Consequently, races can be extremely hard to test for. You can run tests for days and never see the bug, only to experience a catastrophic system crash in the field when the outcome is critical.

Although we can cope with races in a variety of ways, including using mutual-exclusion locks and other methods of synchronization, for our purposes, we shall simply ensure that strands that operate in parallel are ***independent***: they have no determinacy races among them. Thus, in a **parallel for** construct, all the iterations should be independent. Sometimes that means using the **new** keyword to ensure that different iterations do not operate on the same variable, as in MAT-VEC. The **new** keyword allows the same variable name to be used in multiple iterations while referring to different memory locations, which renders accesses to the variable in the different iterations independent. Between a **spawn** and the corresponding **sync**, the code of the spawned child should be independent of the code of the

parent, including code executed by additional spawned or called children. Note that arguments to a spawned child are evaluated in the parent before the actual spawn occurs, and thus the evaluation of arguments to a spawned subroutine is in series with any accesses to those arguments after the spawn.

As an example of how easy it is to generate code with races, here is a faulty implementation of multithreaded matrix-vector multiplication that achieves a span of $\Theta(\lg n)$ by parallelizing the inner **for** loop:

```
MAT-VEC-WRONG(A, x)
1  n = A.rows
2  let y be a new vector of length n
3  parallel for i = 1 to n
4       y_i = 0
5  parallel for i = 1 to n
6       parallel for new j = 1 to n
7            y_i = y_i + a_ij x_j
8  return y
```

This procedure is, unfortunately, incorrect due to races on updating y_i in line 7, which executes concurrently for all n values of j. Exercise 27.1-6 asks you to give a correct implementation with $\Theta(\lg n)$ span.

A multithreaded algorithm with races can sometimes be correct. 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. Generally, however, we shall consider code with races to be illegal.

A chess lesson

We close this section with a true story that occurred during the development of the world-class multithreaded chess-playing program ⋆Socrates [80], although the timings below have been simplified for exposition. The program was prototyped on a 32-processor computer but was ultimately to run on a supercomputer with 512 processors. At one point, the developers incorporated an optimization into the program that reduced its running time on an important benchmark on the 32-processor 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 512 processors. As a result, they abandoned the "optimization."

Here is their analysis. The original version of the program had work $T_1 = 2048$ seconds and span $T_\infty = 1$ second. If we treat inequality (27.4) as an equation, $T_P = T_1/P + T_\infty$, and use it as an approximation to the running time on P processors, we see that indeed $T_{32} = 2048/32 + 1 = 65$. With the optimization, the

work became $T_1' = 1024$ seconds and the span became $T_\infty' = 8$ seconds. Again using our approximation, we get $T_{32}' = 1024/32 + 8 = 40$.

The relative speeds of the two versions switch when we calculate the running times on 512 processors, however. In particular, we have $T_{512} = 2048/512+1 = 5$ seconds, and $T_{512}' = 1024/512 + 8 = 10$ seconds. The optimization that sped up the program on 32 processors would have made the program twice as slow on 512 processors! The optimized version's span of 8, which was not the dominant term in the running time on 32 processors, became the dominant term on 512 processors, nullifying the advantage from using more processors.

The moral of the story is that work and span can provide a better means of extrapolating performance than can measured running times.

Exercises

27.1-1

Suppose that we spawn P-FIB$(n-2)$ in line 4 of P-FIB, rather than calling it as is done in the code. What is the impact on the asymptotic work, span, and parallelism?

27.1-2

Draw the computation dag 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 dag on 3 processors using greedy scheduling by labeling each strand with the time step in which it is executed.

27.1-3

Prove that a greedy scheduler achieves the following time bound, which is slightly stronger than the bound proved in Theorem 27.1:

$$T_P \le \frac{T_1 - T_\infty}{P} + T_\infty \,. \tag{27.5}$$

27.1-4

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

27.1-5

Professor Karan measures her deterministic multithreaded 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 (27.2), the span law (27.3), and inequality (27.5) from Exercise 27.1-3.)

27.1-6

Give a multithreaded 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.

27.1-7

Consider the following multithreaded pseudocode for transposing an $n \times n$ matrix A in place:

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

Analyze the work, span, and parallelism of this algorithm.

27.1-8

Suppose that we replace the **parallel for** loop in line 3 of P-TRANSPOSE (see Exercise 27.1-7) with an ordinary **for** loop. Analyze the work, span, and parallelism of the resulting algorithm.

27.1-9

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

27.2 Multithreaded matrix multiplication

In this section, we examine how to multithread matrix multiplication, a problem whose serial running time we studied in Section 4.2. We'll look at multithreaded algorithms based on the standard triply nested loop, as well as divide-and-conquer algorithms.

Multithreaded matrix multiplication

The first algorithm we study is the straightforward algorithm based on parallelizing the loops in the procedure SQUARE-MATRIX-MULTIPLY on page 75:

```
P-SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  parallel for i = 1 to n
4      parallel for new j = 1 to n
5          c_ij = 0
6          for new k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

To analyze this algorithm, observe that since the serialization of the algorithm is just SQUARE-MATRIX-MULTIPLY, the work is therefore simply $T_1(n) = \Theta(n^3)$, the same as the running time of SQUARE-MATRIX-MULTIPLY. 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 3, then down the tree of recursion for the **parallel for** loop starting in line 4, and then executes all n iterations of the ordinary **for** loop starting in line 6, 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 27.2-3 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 divide-and-conquer multithreaded algorithm for matrix multiplication

As we learned in Section 4.2, we can multiply $n \times n$ matrices serially in time $\Theta(n^{\lg 7}) = O(n^{2.81})$ using Strassen's divide-and-conquer strategy, which motivates us to look at multithreading such an algorithm. We begin, as we did in Section 4.2, with multithreading a simpler divide-and-conquer algorithm.

Recall from page 77 that the SQUARE-MATRIX-MULTIPLY-RECURSIVE procedure, which multiplies two $n \times n$ matrices A and B to produce the $n \times n$ matrix C, relies on partitioning each of the three matrices into four $n/2 \times n/2$ submatrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Then, we can write the matrix product as

$$\begin{aligned} \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\ &= \begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}. \end{aligned} \tag{27.6}$$

Thus, to multiply two $n \times n$ matrices, we perform eight multiplications of $n/2 \times n/2$ matrices and one addition of $n \times n$ matrices. The following pseudocode implements

this divide-and-conquer strategy using nested parallelism. Unlike the SQUARE-MATRIX-MULTIPLY-RECURSIVE procedure on which it is based, P-MATRIX-MULTIPLY-RECURSIVE takes the output matrix as a parameter to avoid allocating matrices unnecessarily.

```
P-MATRIX-MULTIPLY-RECURSIVE(C, A, B)
 1  n = A.rows
 2  if n == 1
 3      c_11 = a_11 b_11
 4  else let T be a new n × n matrix
 5      partition A, B, C, and T 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 T_11, T_12, T_21, T_22; respectively
 6      spawn P-MATRIX-MULTIPLY-RECURSIVE(C_11, A_11, B_11)
 7      spawn P-MATRIX-MULTIPLY-RECURSIVE(C_12, A_11, B_12)
 8      spawn P-MATRIX-MULTIPLY-RECURSIVE(C_21, A_21, B_11)
 9      spawn P-MATRIX-MULTIPLY-RECURSIVE(C_22, A_21, B_12)
10      spawn P-MATRIX-MULTIPLY-RECURSIVE(T_11, A_12, B_21)
11      spawn P-MATRIX-MULTIPLY-RECURSIVE(T_12, A_12, B_22)
12      spawn P-MATRIX-MULTIPLY-RECURSIVE(T_21, A_22, B_21)
13      P-MATRIX-MULTIPLY-RECURSIVE(T_22, A_22, B_22)
14      sync
15      parallel for i = 1 to n
16          parallel for new j = 1 to n
17              c_ij = c_ij + t_ij
```

Line 3 handles the base case, where we are multiplying 1×1 matrices. We handle the recursive case in lines 4–17. We allocate a temporary matrix T in line 4, and line 5 partitions each of the matrices A, B, C, and T into $n/2 \times n/2$ submatrices. (As with SQUARE-MATRIX-MULTIPLY-RECURSIVE on page 77, we gloss over the minor issue of how to use index calculations to represent submatrix sections of a matrix.) The recursive call in line 6 sets the submatrix C_{11} to the submatrix product $A_{11}B_{11}$, so that C_{11} equals the first of the two terms that form its sum in equation (27.6). Similarly, lines 7–9 set C_{12}, C_{21}, and C_{22} to the first of the two terms that equal their sums in equation (27.6). Line 10 sets the submatrix T_{11} to the submatrix product $A_{12}B_{21}$, so that T_{11} equals the second of the two terms that form C_{11}'s sum. Lines 11–13 set T_{12}, T_{21}, and T_{22} to the second of the two terms that form the sums of C_{12}, C_{21}, and C_{22}, respectively. The first seven recursive calls are spawned, and the last one runs in the main strand. The **sync** statement in line 14 ensures that all the submatrix products in lines 6–13 have been computed,

after which we add the products from T into C using the doubly nested **parallel for** loops in lines 15–17.

We first analyze the work $M_1(n)$ of the P-MATRIX-MULTIPLY-RECURSIVE procedure, echoing the serial running-time analysis of its progenitor SQUARE-MATRIX-MULTIPLY-RECURSIVE. In the recursive case, we partition in $\Theta(1)$ time, perform eight recursive multiplications of $n/2 \times n/2$ matrices, and finish up with the $\Theta(n^2)$ work from adding two $n \times n$ matrices. Thus, the recurrence for the work $M_1(n)$ is

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

by case 1 of the master theorem. In other words, the work of our multithreaded algorithm is asymptotically the same as the running time of the procedure SQUARE-MATRIX-MULTIPLY in Section 4.2, with its triply nested loops.

To determine the span $M_\infty(n)$ of P-MATRIX-MULTIPLY-RECURSIVE, we first observe that the span for partitioning is $\Theta(1)$, which is dominated by the $\Theta(\lg n)$ span of the doubly nested **parallel for** loops in lines 15–17. Because the eight parallel recursive calls all execute on matrices of the same size, the maximum span for any recursive call is just the span of any one. Hence, the recurrence for the span $M_\infty(n)$ of P-MATRIX-MULTIPLY-RECURSIVE is

$$M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) \; . \tag{27.7}$$

This recurrence does not fall under any of the cases of the master theorem, but it does meet the condition of Exercise 4.6-2. By Exercise 4.6-2, therefore, the solution to recurrence (27.7) is $M_\infty(n) = \Theta(\lg^2 n)$.

Now that we know the work and span of P-MATRIX-MULTIPLY-RECURSIVE, we can compute its parallelism as $M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$, which is very high.

Multithreading Strassen's method

To multithread Strassen's algorithm, we follow the same general outline as on page 79, only using nested parallelism:

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (27.6). This step takes $\Theta(1)$ work and span by index calculation.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices with $\Theta(n^2)$ work and $\Theta(\lg n)$ span by using doubly nested **parallel for** loops.

3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively spawn the computation of seven $n/2 \times n/2$ matrix products $P_1, P_2, \ldots, P_7$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices, once again using doubly nested **parallel for** loops. We can compute all four submatrices with $\Theta(n^2)$ work and $\Theta(\lg n)$ span.

To analyze this algorithm, we first observe that since the serialization is the same as the original serial algorithm, the work is just the running time of the serialization, namely, $\Theta(n^{\lg 7})$. As for 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 (27.7) as we did for P-MATRIX-MULTIPLY-RECURSIVE, which has solution $\Theta(\lg^2 n)$. Thus, the parallelism of multithreaded Strassen's method is $\Theta(n^{\lg 7}/\lg^2 n)$, which is high, though slightly less than the parallelism of P-MATRIX-MULTIPLY-RECURSIVE.

Exercises

27.2-1

Draw the computation dag for computing P-SQUARE-MATRIX-MULTIPLY on 2×2 matrices, labeling how the vertices in your diagram correspond to strands in the execution of the algorithm. Use the convention that spawn and call edges point downward, continuation edges point horizontally to the right, and return edges point upward. Assuming that each strand takes unit time, analyze the work, span, and parallelism of this computation.

27.2-2

Repeat Exercise 27.2-1 for P-MATRIX-MULTIPLY-RECURSIVE.

27.2-3

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

27.2-4

Give pseudocode for an efficient multithreaded 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 are 1. Analyze your algorithm.

27.2-5

Give pseudocode for an efficient multithreaded algorithm that transposes an $n \times n$ matrix in place by using divide-and-conquer and no **parallel for** loops to divide the matrix recursively into four $n/2 \times n/2$ submatrices. Analyze your algorithm.

27.2-6

Give pseudocode for an efficient multithreaded implementation of the Floyd-Warshall algorithm (see Section 25.2), which computes shortest paths between all pairs of vertices in an edge-weighted graph. Analyze your algorithm.

27.3 Multithreaded 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 paradigm, it seems like a terrific candidate for multithreading using nested parallelism. We can easily modify the pseudocode so that the first recursive call is spawned:

```
MERGE-SORT'(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      spawn MERGE-SORT'(A, p, q)
4      MERGE-SORT'(A, q + 1, r)
5      sync
6      MERGE(A, p, q, r)
```

Like its serial counterpart, MERGE-SORT′ sorts the subarray $A[p..r]$. After the two recursive subroutines in lines 3 and 4 have completed, which is ensured by the **sync** statement in line 5, MERGE-SORT′ calls the same MERGE procedure as on page 31.

Let us analyze MERGE-SORT′. To do so, we first need to analyze MERGE. Recall that its serial running time to merge n elements is $\Theta(n)$. Because MERGE is serial, both its work and its span are $\Theta(n)$. Thus, the following recurrence characterizes the work $MS'_1(n)$ of MERGE-SORT′ on n elements:

$$\begin{aligned} MS'_1(n) &= 2\,MS'_1(n/2) + \Theta(n) \\ &= \Theta(n \lg n)\,, \end{aligned}$$

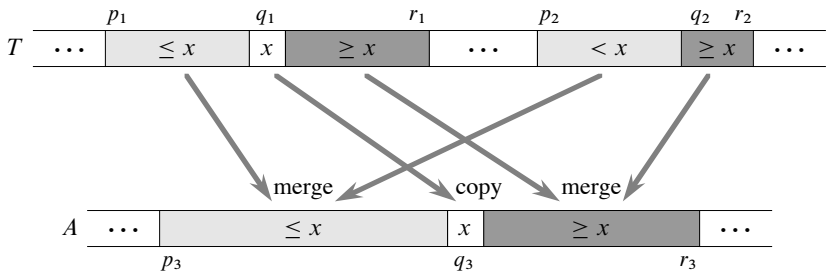


Figure 27.6 The idea behind the multithreaded merging of two sorted subarrays $T[p_1 \,..\, r_1]$ and $T[p_2 \,..\, r_2]$ into the subarray $A[p_3 \,..\, r_3]$. Letting $x = T[q_1]$ be the median of $T[p_1 \,..\, r_1]$ and q_2 be the place in $T[p_2 \,..\, r_2]$ such that x would fall between $T[q_2 - 1]$ and $T[q_2]$, every element in subarrays $T[p_1 \,..\, q_1 - 1]$ and $T[p_2 \,..\, q_2 - 1]$ (lightly shaded) is less than or equal to x, and every element in the subarrays $T[q_1 + 1 \,..\, r_1]$ and $T[q_2 + 1 \,..\, r_2]$ (heavily shaded) is at least x. To merge, we compute the index q_3 where x belongs in $A[p_3 \,..\, r_3]$, copy x into $A[q_3]$, and then recursively merge $T[p_1 \,..\, q_1 - 1]$ with $T[p_2 \,..\, q_2 - 1]$ into $A[p_3 \,..\, q_3 - 1]$ and $T[q_1 + 1 \,..\, r_1]$ with $T[q_2 \,..\, r_2]$ into $A[q_3 + 1 \,..\, r_3]$.

which is the same as the serial running time of merge sort. Since the two recursive calls of MERGE-SORT′ can run in parallel, the span MS'_∞ is given by the recurrence

$$\begin{aligned} MS'_\infty(n) &= MS'_\infty(n/2) + \Theta(n) \\ &= \Theta(n) \,. \end{aligned}$$

Thus, the parallelism of MERGE-SORT′ comes to $MS'_1(n)/MS'_\infty(n) = \Theta(\lg n)$, which is an unimpressive amount of parallelism. To sort 10 million elements, for example, it might achieve linear speedup on a few processors, but it would not scale up effectively to hundreds of processors.

You probably have already figured out where the parallelism bottleneck is in this multithreaded merge sort: the serial MERGE procedure. Although merging might initially seem to be inherently serial, we can, in fact, fashion a multithreaded version of it by using nested parallelism.

Our divide-and-conquer strategy for multithreaded merging, which is illustrated in Figure 27.6, operates on subarrays of an array T. Suppose that we are merging the two sorted subarrays $T[p_1 \,..\, r_1]$ of length $n_1 = r_1 - p_1 + 1$ and $T[p_2 \,..\, r_2]$ of length $n_2 = r_2 - p_2 + 1$ into another subarray $A[p_3 \,..\, r_3]$, of length $n_3 = r_3 - p_3 + 1 = n_1 + n_2$. Without loss of generality, we make the simplifying assumption that $n_1 \ge n_2$.

We first find the middle element $x = T[q_1]$ of the subarray $T[p_1 \,..\, r_1]$, where $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$. Because the subarray is sorted, x is a median of $T[p_1 \,..\, r_1]$: every element in $T[p_1 \,..\, q_1 - 1]$ is no more than x, and every element in $T[q_1 + 1 \,..\, r_1]$ is no less than x. We then use binary search to find the

index q_2 in the subarray $T[p_2 \,..\, r_2]$ so that the subarray would still be sorted if we inserted x between $T[q_2 - 1]$ and $T[q_2]$.

We next merge the original subarrays $T[p_1 \,..\, r_1]$ and $T[p_2 \,..\, r_2]$ into $A[p_3 \,..\, r_3]$ as follows:

1. Set $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$.
2. Copy x into $A[q_3]$.
3. Recursively merge $T[p_1 \,..\, q_1 - 1]$ with $T[p_2 \,..\, q_2 - 1]$, and place the result into the subarray $A[p_3 \,..\, q_3 - 1]$.
4. Recursively merge $T[q_1 + 1 \,..\, r_1]$ with $T[q_2 \,..\, r_2]$, and place the result into the subarray $A[q_3 + 1 \,..\, r_3]$.

When we compute q_3, the quantity $q_1 - p_1$ is the number of elements in the subarray $T[p_1 \,..\, q_1 - 1]$, and the quantity $q_2 - p_2$ is the number of elements in the subarray $T[p_2 \,..\, q_2 - 1]$. Thus, their sum is the number of elements that end up before x in the subarray $A[p_3 \,..\, r_3]$.

The base case occurs when $n_1 = n_2 = 0$, in which case we have no work to do to merge the two empty subarrays. Since we have assumed that the subarray $T[p_1 \,..\, r_1]$ is at least as long as $T[p_2 \,..\, r_2]$, that is, $n_1 \geq n_2$, we can check for the base case by just checking whether $n_1 = 0$. We must also ensure that the recursion properly handles the case when only one of the two subarrays is empty, which, by our assumption that $n_1 \geq n_2$, must be the subarray $T[p_2 \,..\, r_2]$.

Now, let's put these ideas into pseudocode. We start with the binary search, which we express serially. The procedure BINARY-SEARCH(x, T, p, r) takes a key x and a subarray $T[p \,..\, r]$, and it returns one of the following:

- If $T[p \,..\, r]$ is empty ($r < p$), then it returns the index p.
- If $x \leq T[p]$, and hence less than or equal to all the elements of $T[p \,..\, r]$, then it returns the index p.
- If $x > T[p]$, then it returns the largest index q in the range $p < q \leq r + 1$ such that $T[q - 1] < x$.

Here is the pseudocode:

```
BINARY-SEARCH(x, T, p, r)
1  low = p
2  high = max(p, r + 1)
3  while low < high
4      mid = ⌊(low + high)/2⌋
5      if x ≤ T[mid]
6          high = mid
7      else low = mid + 1
8  return high
```

The call BINARY-SEARCH(x, T, p, r) takes $\Theta(\lg n)$ serial time in the worst case, where $n = r - p + 1$ is the size of the subarray on which it runs. (See Exercise 2.3-5.) Since BINARY-SEARCH is a serial procedure, its worst-case work and span are both $\Theta(\lg n)$.

We are now prepared to write pseudocode for the multithreaded merging procedure itself. Like the MERGE procedure on page 31, the P-MERGE procedure assumes that the two subarrays to be merged lie within the same array. Unlike MERGE, however, P-MERGE does not assume that the two subarrays to be merged are adjacent within the array. (That is, P-MERGE does not require that $p_2 = r_1 + 1$.) Another difference between MERGE and P-MERGE is that P-MERGE takes as an argument an output subarray A into which the merged values should be stored. The call P-MERGE$(T, p_1, r_1, p_2, r_2, A, p_3)$ merges the sorted subarrays $T[p_1 .. r_1]$ and $T[p_2 .. r_2]$ into the subarray $A[p_3 .. r_3]$, where $r_3 = p_3 + (r_1 - p_1 + 1) + (r_2 - p_2 + 1) - 1 = p_3 + (r_1 - p_1) + (r_2 - p_2) + 1$ and is not provided as an input.

```
P-MERGE(T, p1, r1, p2, r2, A, p3)
 1  n1 = r1 - p1 + 1
 2  n2 = r2 - p2 + 1
 3  if n1 < n2                    // ensure that n1 ≥ n2
 4      exchange p1 with p2
 5      exchange r1 with r2
 6      exchange n1 with n2
 7  if n1 == 0                    // both empty?
 8      return
 9  else q1 = ⌊(p1 + r1)/2⌋
10      q2 = BINARY-SEARCH(T[q1], T, p2, r2)
11      q3 = p3 + (q1 - p1) + (q2 - p2)
12      A[q3] = T[q1]
13      spawn P-MERGE(T, p1, q1 - 1, p2, q2 - 1, A, p3)
14      P-MERGE(T, q1 + 1, r1, q2, r2, A, q3 + 1)
15      sync
```

The P-MERGE procedure works as follows. Lines 1–2 compute the lengths n_1 and n_2 of the subarrays $T[p_1 .. r_1]$ and $T[p_2 .. r_2]$, respectively. Lines 3–6 enforce the assumption that $n_1 \geq n_2$. Line 7 tests for the base case, where the subarray $T[p_1 .. r_1]$ is empty (and hence so is $T[p_2 .. r_2]$), in which case we simply return. Lines 9–15 implement the divide-and-conquer strategy. Line 9 computes the midpoint of $T[p_1 .. r_1]$, and line 10 finds the point q_2 in $T[p_2 .. r_2]$ such that all elements in $T[p_2 .. q_2 - 1]$ are less than $T[q_1]$ (which corresponds to x) and all the elements in $T[q_2 .. r_2]$ are at least as large as $T[q_1]$. Line 11 com-

putes the index q_3 of the element that divides the output subarray $A[p_3 \,..\, r_3]$ into $A[p_3 \,..\, q_3 - 1]$ and $A[q_3+1 \,..\, r_3]$, and then line 12 copies $T[q_1]$ directly into $A[q_3]$.

Then, we recurse using nested parallelism. Line 13 spawns the first subproblem, while line 14 calls the second subproblem in parallel. The **sync** statement in line 15 ensures that the subproblems have completed before the procedure returns. (Since every procedure implicitly executes a **sync** before returning, we could have omitted the **sync** statement in line 15, but including it is good coding practice.) There is some cleverness in the coding to ensure that when the subarray $T[p_2 \,..\, r_2]$ is empty, the code operates correctly. The way it works is that on each recursive call, a median element of $T[p_1 \,..\, r_1]$ is placed into the output subarray, until $T[p_1 \,..\, r_1]$ itself finally becomes empty, triggering the base case.

Analysis of multithreaded merging

We first derive a recurrence for the span $PM_\infty(n)$ of P-MERGE, where the two subarrays contain a total of $n = n_1+n_2$ elements. Because the spawn in line 13 and the call in line 14 operate logically in parallel, we need examine only the costlier of the two calls. The key is to understand that in the worst case, the maximum number of elements in either of the recursive calls can be at most $3n/4$, which we see as follows. Because lines 3–6 ensure that $n_2 \le n_1$, it follows that $n_2 = 2n_2/2 \le (n_1 + n_2)/2 = n/2$. In the worst case, one of the two recursive calls merges $\lfloor n_1/2 \rfloor$ elements of $T[p_1 \,..\, r_1]$ with all n_2 elements of $T[p_2 \,..\, r_2]$, and hence the number of elements involved in the call is

$$\begin{aligned} \lfloor n_1/2 \rfloor + n_2 &\le n_1/2 + n_2/2 + n_2/2 \\ &= (n_1 + n_2)/2 + n_2/2 \\ &\le n/2 + n/4 \\ &= 3n/4 \,. \end{aligned}$$

Adding in the $\Theta(\lg n)$ cost of the call to BINARY-SEARCH in line 10, we obtain the following recurrence for the worst-case span:

$$PM_\infty(n) = PM_\infty(3n/4) + \Theta(\lg n) \,. \tag{27.8}$$

(For the base case, the span is $\Theta(1)$, since lines 1–8 execute in constant time.) This recurrence does not fall under any of the cases of the master theorem, but it meets the condition of Exercise 4.6-2. Therefore, the solution to recurrence (27.8) is $PM_\infty(n) = \Theta(\lg^2 n)$.

We now analyze the work $PM_1(n)$ of P-MERGE on n elements, which turns out to be $\Theta(n)$. Since each of the n elements must be copied from array T to array A, we have $PM_1(n) = \Omega(n)$. Thus, it remains only to show that $PM_1(n) = O(n)$.

We shall first derive a recurrence for the worst-case work. The binary search in line 10 costs $\Theta(\lg n)$ in the worst case, which dominates the other work outside

of the recursive calls. For the recursive calls, observe that although the recursive calls in lines 13 and 14 might merge different numbers of elements, together the two recursive calls merge at most n elements (actually $n-1$ elements, since $T[q_1]$ does not participate in either recursive call). Moreover, as we saw in analyzing the span, a recursive call operates on at most $3n/4$ elements. We therefore obtain the recurrence

$$PM_1(n) = PM_1(\alpha n) + PM_1((1-\alpha)n) + O(\lg n)\,, \tag{27.9}$$

where α lies in the range $1/4 \le \alpha \le 3/4$, and where we understand that the actual value of α may vary for each level of recursion.

We prove that recurrence (27.9) has solution $PM_1 = O(n)$ via the substitution method. Assume that $PM_1(n) \le c_1 n - c_2 \lg n$ for some positive constants c_1 and c_2. Substituting gives us

$$\begin{aligned}
PM_1(n) &\le (c_1\alpha n - c_2\lg(\alpha n)) + (c_1(1-\alpha)n - c_2\lg((1-\alpha)n)) + \Theta(\lg n) \\
&= c_1(\alpha + (1-\alpha))n - c_2(\lg(\alpha n) + \lg((1-\alpha)n)) + \Theta(\lg n) \\
&= c_1 n - c_2(\lg\alpha + \lg n + \lg(1-\alpha) + \lg n) + \Theta(\lg n) \\
&= c_1 n - c_2\lg n - (c_2(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \\
&\le c_1 n - c_2\lg n\,,
\end{aligned}$$

since we can choose c_2 large enough that $c_2(\lg n + \lg(\alpha(1-\alpha)))$ dominates the $\Theta(\lg n)$ term. Furthermore, we can choose c_1 large enough to satisfy the base conditions of the recurrence. Since the work $PM_1(n)$ of P-MERGE is both $\Omega(n)$ and $O(n)$, we have $PM_1(n) = \Theta(n)$.

The parallelism of P-MERGE is $PM_1(n)/PM_\infty(n) = \Theta(n/\lg^2 n)$.

Multithreaded merge sort

Now that we have a nicely parallelized multithreaded merging procedure, we can incorporate it into a multithreaded merge sort. This version of merge sort is similar to the MERGE-SORT′ procedure we saw earlier, but unlike MERGE-SORT′, it takes as an argument an output subarray B, which will hold the sorted result. In particular, the call P-MERGE-SORT(A, p, r, B, s) sorts the elements in $A[p..r]$ and stores them in $B[s..s+r-p]$.

```
P-MERGE-SORT(A, p, r, B, s)
 1  n = r − p + 1
 2  if n == 1
 3       B[s] = A[p]
 4  else let T[1..n] be a new array
 5       q = ⌊(p + r)/2⌋
 6       q′ = q − p + 1
 7       spawn P-MERGE-SORT(A, p, q, T, 1)
 8       P-MERGE-SORT(A, q + 1, r, T, q′ + 1)
 9       sync
10       P-MERGE(T, 1, q′, q′ + 1, n, B, s)
```

After line 1 computes the number n of elements in the input subarray $A[p..r]$, lines 2–3 handle the base case when the array has only 1 element. Lines 4–6 set up for the recursive spawn in line 7 and call in line 8, which operate in parallel. In particular, line 4 allocates a temporary array T with n elements to store the results of the recursive merge sorting. Line 5 calculates the index q of $A[p..r]$ to divide the elements into the two subarrays $A[p..q]$ and $A[q+1..r]$ that will be sorted recursively, and line 6 goes on to compute the number q' of elements in the first subarray $A[p..q]$, which line 8 uses to determine the starting index in T of where to store the sorted result of $A[q+1..r]$. At that point, the spawn and recursive call are made, followed by the **sync** in line 9, which forces the procedure to wait until the spawned procedure is done. Finally, line 10 calls P-MERGE to merge the sorted subarrays, now in $T[1..q']$ and $T[q'+1..n]$, into the output subarray $B[s..s+r-p]$.

Analysis of multithreaded merge sort

We start by analyzing the work $PMS_1(n)$ of P-MERGE-SORT, which is considerably easier than analyzing the work of P-MERGE. Indeed, the work is given by the recurrence

$$\begin{aligned} PMS_1(n) &= 2\,PMS_1(n/2) + PM_1(n) \\ &= 2\,PMS_1(n/2) + \Theta(n)\,. \end{aligned}$$

This recurrence is the same as the recurrence (4.4) for ordinary MERGE-SORT from Section 2.3.1 and has solution $PMS_1(n) = \Theta(n \lg n)$ by case 2 of the master theorem.

We now derive and analyze a recurrence for the worst-case span $PMS_\infty(n)$. Because the two recursive calls to P-MERGE-SORT on lines 7 and 8 operate logically in parallel, we can ignore one of them, obtaining the recurrence

$$\begin{aligned} PMS_\infty(n) &= PMS_\infty(n/2) + PM_\infty(n) \\ &= PMS_\infty(n/2) + \Theta(\lg^2 n) \, . \end{aligned} \tag{27.10}$$

As for recurrence (27.8), the master theorem does not apply to recurrence (27.10), but Exercise 4.6-2 does. The solution is $PMS_\infty(n) = \Theta(\lg^3 n)$, and so the span of P-MERGE-SORT is $\Theta(\lg^3 n)$.

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

$$\begin{aligned} PMS_1(n)/PMS_\infty(n) &= \Theta(n \lg n)/\Theta(\lg^3 n) \\ &= \Theta(n/\lg^2 n) \, , \end{aligned}$$

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. The straightforward way to coarsen the base case is to switch to an ordinary serial sort, perhaps quicksort, when the size of the array is sufficiently small.

Exercises

27.3-1

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

27.3-2

Instead of finding a median element in the larger subarray, as P-MERGE does, consider a variant that finds a median element of all the elements in the two sorted subarrays using the result of Exercise 9.3-8. Give pseudocode for an efficient multithreaded merging procedure that uses this median-finding procedure. Analyze your algorithm.

27.3-3

Give an efficient multithreaded algorithm for partitioning an array around a pivot, as is done by the PARTITION procedure on page 171. You need not partition the array in place. Make your algorithm as parallel as possible. Analyze your algorithm. (*Hint:* You may need an auxiliary array and may need to make more than one pass over the input elements.)

27.3-4

Give a multithreaded version of RECURSIVE-FFT on page 911. Make your implementation as parallel as possible. Analyze your algorithm.

27.3-5 ⋆

Give a multithreaded version of RANDOMIZED-SELECT on page 216. Make your implementation as parallel as possible. Analyze your algorithm. (*Hint:* Use the partitioning algorithm from Exercise 27.3-3.)

27.3-6 ⋆

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

Problems

27-1 Implementing parallel loops using nested parallelism

Consider the following multithreaded algorithm 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)
1  parallel for i = 1 to A.length
2      C[i] = A[i] + B[i]
```

a. Rewrite the parallel loop in SUM-ARRAYS using nested parallelism (**spawn** and **sync**) in the manner of MAT-VEC-MAIN-LOOP. Analyze the parallelism of your implementation.

Consider the following alternative implementation of the parallel loop, which contains a value *grain-size* to be specified:

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

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

b. Suppose that we set *grain-size* $= 1$. What is the parallelism of this implementation?

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.

27-2 Saving temporary space in matrix multiplication

The P-MATRIX-MULTIPLY-RECURSIVE procedure has the disadvantage that it must allocate a temporary matrix T of size $n \times n$, which can adversely affect the constants hidden by the Θ-notation. The P-MATRIX-MULTIPLY-RECURSIVE procedure does have high parallelism, however. 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. Describe a recursive multithreaded algorithm that eliminates the need for the temporary matrix T at the cost of increasing the span to $\Theta(n)$. (*Hint:* Compute $C = C + AB$ following the general strategy of P-MATRIX-MULTIPLY-RECURSIVE, but initialize C in parallel and insert a **sync** in a judiciously chosen location.)

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.

27-3 Multithreaded matrix algorithms

a. Parallelize the LU-DECOMPOSITION procedure on page 821 by giving pseudocode for a multithreaded 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 824.

c. Do the same for LUP-SOLVE on page 817.

d. Do the same for a multithreaded algorithm based on equation (28.13) for inverting a symmetric positive-definite matrix.

27-4 Multithreading reductions and prefix computations

A ***⊗-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 following 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. Use nested parallelism to implement a multithreaded algorithm P-REDUCE, which performs the same function with $\Theta(n)$ work and $\Theta(\lg n)$ span. Analyze your algorithm.

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

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

Unfortunately, multithreading SCAN is not straightforward. For example, 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 following procedure P-SCAN-1 performs the $\otimes$-prefix computation in parallel, albeit inefficiently:

```
P-SCAN-1(x)
1  n = x.length
2  let y[1..n] be a new array
3  P-SCAN-1-AUX(x, y, 1, n)
4  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.

By using nested parallelism, we can obtain a more efficient $\otimes$-prefix computation:

```
P-SCAN-2(x)
1  n = x.length
2  let y[1..n] be a new array
3  P-SCAN-2-AUX(x, y, 1, n)
4  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.

We can improve on both P-SCAN-1 and P-SCAN-2 by performing the $\otimes$-prefix computation in two distinct passes over the data. On the first pass, we gather the terms for various contiguous subarrays of x into a temporary array t, and on the second pass we use the terms in t to compute the final result y. The following pseudocode implements this strategy, but certain expressions have been omitted:

```
P-SCAN-3(x)
1  n = x.length
2  let y[1..n] and t[1..n] be new arrays
3  y[1] = x[1]
4  if n > 1
5       P-SCAN-UP(x, t, 2, n)
6       P-SCAN-DOWN(x[1], x, t, y, 2, n)
7  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
```

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 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.

27-5 Multithreading 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 15.4 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 $c[i-1, j-1]$.

In this problem, we examine how to use nested parallelism to multithread a simple stencil calculation on an $n \times n$ array A in which, of the values in A, 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 we have filled in the entries upon which $A[i, j]$ depends, we can fill in $A[i, j]$ in $\Theta(1)$ time (as in the LCS-LENGTH procedure of Section 15.4).

We can partition the $n \times n$ array A into four $n/2 \times n/2$ subarrays as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}. \tag{27.11}$$

Observe now that we can fill in subarray A_{11} recursively, since it does not depend on the entries of the other three subarrays. Once A_{11} is complete, we can continue to 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, we can fill in A_{22} recursively.

a. Give multithreaded pseudocode that performs this simple stencil calculation using a divide-and-conquer algorithm SIMPLE-STENCIL based on the decomposition (27.11) 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 last 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 multithreaded algorithm for this simple stencil calculation that achieves $\Theta(n/\lg n)$ parallelism. Argue using notions of work and span that the problem, in fact, has $\Theta(n)$ inherent parallelism. As it turns out, the divide-and-conquer nature of our multithreaded pseudocode does not let us achieve this maximal parallelism.

27-6 Randomized multithreaded algorithms

Just as with ordinary serial algorithms, we sometimes want to implement randomized multithreaded algorithms. This problem explores how to adapt the various performance measures in order to handle the expected behavior of such algorithms. It also asks you to design and analyze a multithreaded algorithm for randomized quicksort.

a. Explain how to modify the work law (27.2), span law (27.3), and greedy scheduler bound (27.4) to work with expectations when T_P, T_1, and T_∞ are all random variables.

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

d. Multithread the RANDOMIZED-QUICKSORT algorithm on page 179 by using nested parallelism. (Do not parallelize RANDOMIZED-PARTITION.) Give the pseudocode for your P-RANDOMIZED-QUICKSORT algorithm.

e. Analyze your multithreaded algorithm for randomized quicksort. (*Hint:* Review the analysis of RANDOMIZED-SELECT on page 216.)

Chapter notes

Parallel computers, models for 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 [48, 168] is another popular algorithmic programming model, which features operations on vectors and matrices as primitives.

Graham [149] and Brent [55] showed that there exist schedulers achieving the bound of Theorem 27.1. Eager, Zahorjan, and Lazowska [98] 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 [47] developed an algorithmic programming model based on work and span (which he called the "depth" of the computation) for data-parallel programming. Blumofe and Leiserson [52] gave a distributed scheduling algorithm for dynamic multithreading 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 [19] and Blelloch, Gibbons, and Matias [49] also provided provably good algorithms for scheduling dynamic multithreaded computations.

The multithreaded pseudocode and programming model were heavily influenced by the Cilk [51, 118] project at MIT and the Cilk++ [71] extensions to C++ distributed by Cilk Arts, Inc. Many of the multithreaded algorithms in this chapter appeared in unpublished lecture notes by C. E. Leiserson and H. Prokop and have been implemented in Cilk or Cilk++. The multithreaded merge-sorting algorithm was inspired by an algorithm of Akl [12].

The notion of sequential consistency is due to Lamport [223].