### Discharging an overflowing vertex

An overflowing vertex $u$ is ***discharged*** by pushing all of its excess flow through admissible edges to neighboring vertices, relabeling $u$ as necessary to cause edges leaving $u$ to become admissible. The pseudocode goes as follows.

```
DISCHARGE(u)
1  while u.e > 0
2      v = u.current
3      if v == NIL
4          RELABEL(u)
5          u.current = u.N.head
6      elseif c_f(u, v) > 0 and u.h == v.h + 1
7          PUSH(u, v)
8      else u.current = v.next-neighbor
```

Figure 26.9 steps through several iterations of the **while** loop of lines 1–8, which executes as long as vertex $u$ has positive excess. Each iteration performs exactly one of three actions, depending on the current vertex $v$ in the neighbor list $u.N$.

1. If $v$ is NIL, then we have run off the end of $u.N$. Line 4 relabels vertex $u$, and then line 5 resets the current neighbor of $u$ to be the first one in $u.N$. (Lemma 26.29 below states that the relabel operation applies in this situation.)
2. If $v$ is non-NIL and $(u, v)$ is an admissible edge (determined by the test in line 6), then line 7 pushes some (or possibly all) of $u$'s excess to vertex $v$.
3. If $v$ is non-NIL but $(u, v)$ is inadmissible, then line 8 advances $u.current$ one position further in the neighbor list $u.N$.

Observe that if DISCHARGE is called on an overflowing vertex $u$, then the last action performed by DISCHARGE must be a push from $u$. Why? The procedure terminates only when $u.e$ becomes zero, and neither the relabel operation nor advancing the pointer $u.current$ affects the value of $u.e$.

We must be sure that when PUSH or RELABEL is called by DISCHARGE, the operation applies. The next lemma proves this fact.

***Lemma 26.29***

If DISCHARGE calls PUSH$(u, v)$ in line 7, then a push operation applies to $(u, v)$. If DISCHARGE calls RELABEL$(u)$ in line 4, then a relabel operation applies to $u$.

***Proof*** The tests in lines 1 and 6 ensure that a push operation occurs only if the operation applies, which proves the first statement in the lemma.

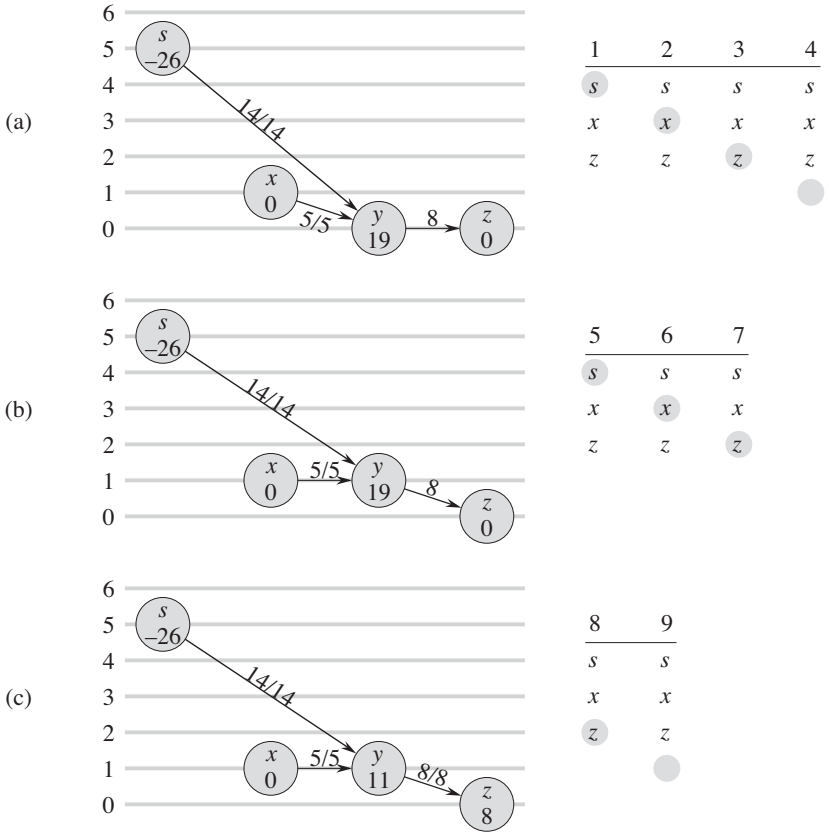



**Figure 26.9** Discharging a vertex $y$. It takes 15 iterations of the **while** loop of DISCHARGE to push all the excess flow from $y$. Only the neighbors of $y$ and edges of the flow network that enter or leave $y$ are shown. In each part of the figure, the number inside each vertex is its excess at the beginning of the first iteration shown in the part, and each vertex is shown at its height throughout the part. The neighbor list $y.N$ at the beginning of each iteration appears on the right, with the iteration number on top. The shaded neighbor is $y.current$. **(a)** Initially, there are 19 units of excess to push from $y$, and $y.current = s$. Iterations 1, 2, and 3 just advance $y.current$, since there are no admissible edges leaving $y$. In iteration 4, $y.current =$ NIL (shown by the shading being below the neighbor list), and so $y$ is relabeled and $y.current$ is reset to the head of the neighbor list. **(b)** After relabeling, vertex $y$ has height 1. In iterations 5 and 6, edges $(y, s)$ and $(y, x)$ are found to be inadmissible, but iteration 7 pushes 8 units of excess flow from $y$ to $z$. Because of the push, $y.current$ does not advance in this iteration. **(c)** Because the push in iteration 7 saturated edge $(y, z)$, it is found inadmissible in iteration 8. In iteration 9, $y.current =$ NIL, and so vertex $y$ is again relabeled and $y.current$ is reset.

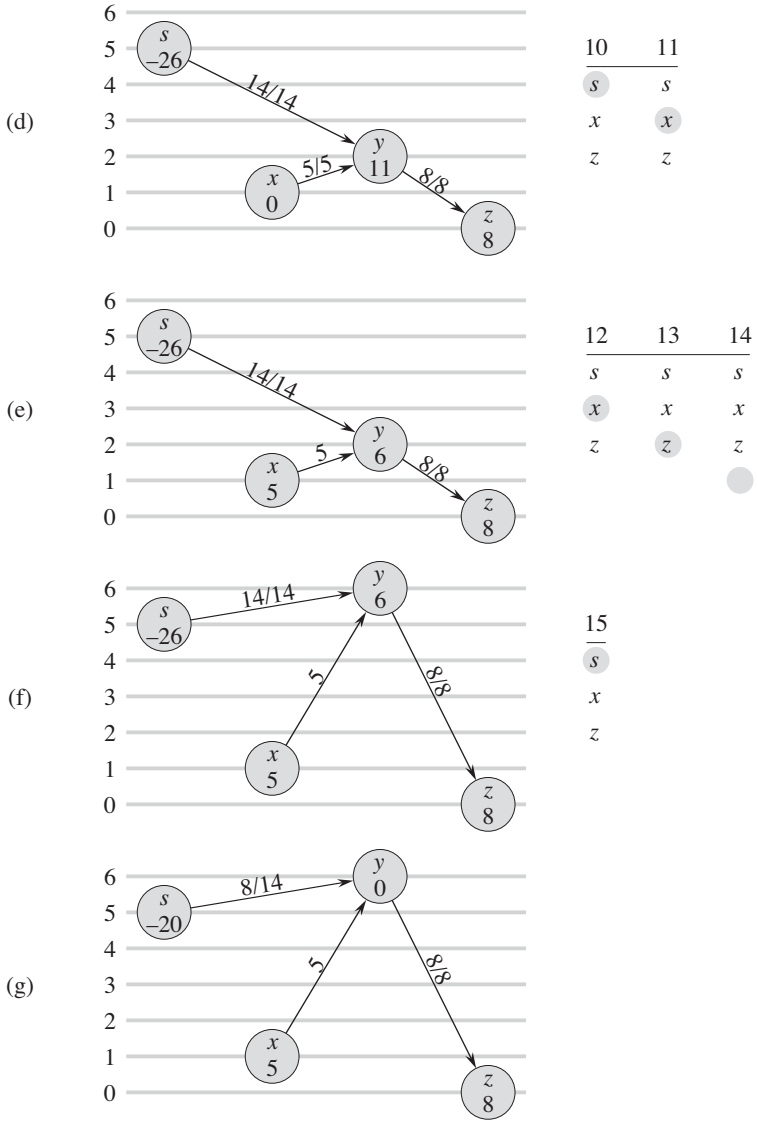



**Figure 26.9, continued** **(d)** In iteration 10, $(y, s)$ is inadmissible, but iteration 11 pushes 5 units of excess flow from $y$ to $x$. **(e)** Because $y.current$ did not advance in iteration 11, iteration 12 finds $(y, x)$ to be inadmissible. Iteration 13 finds $(y, z)$ inadmissible, and iteration 14 relabels vertex $y$ and resets $y.current$. **(f)** Iteration 15 pushes 6 units of excess flow from $y$ to $s$. **(g)** Vertex $y$ now has no excess flow, and DISCHARGE terminates. In this example, DISCHARGE both starts and finishes with the current pointer at the head of the neighbor list, but in general this need not be the case.

To prove the second statement, according to the test in line 1 and Lemma 26.28, we need only show that all edges leaving $u$ are inadmissible. If a call to DISCHARGE($u$) starts with the pointer $u.current$ at the head of $u$'s neighbor list and finishes with it off the end of the list, then all of $u$'s outgoing edges are inadmissible and a relabel operation applies. It is possible, however, that during a call to DISCHARGE($u$), the pointer $u.current$ traverses only part of the list before the procedure returns. Calls to DISCHARGE on other vertices may then occur, but $u.current$ will continue moving through the list during the next call to DISCHARGE($u$). We now consider what happens during a complete pass through the list, which begins at the head of $u.N$ and finishes with $u.current = \text{NIL}$. Once $u.current$ reaches the end of the list, the procedure relabels $u$ and begins a new pass. For the $u.current$ pointer to advance past a vertex $v \in u.N$ during a pass, the edge $(u, v)$ must be deemed inadmissible by the test in line 6. Thus, by the time the pass completes, every edge leaving $u$ has been determined to be inadmissible at some time during the pass. The key observation is that at the end of the pass, every edge leaving $u$ is still inadmissible. Why? By Lemma 26.27, pushes cannot create any admissible edges, regardless of which vertex the flow is pushed from. Thus, any admissible edge must be created by a relabel operation. But the vertex $u$ is not relabeled during the pass, and by Lemma 26.28, any other vertex $v$ that is relabeled during the pass (resulting from a call of DISCHARGE($v$)) has no entering admissible edges after relabeling. Thus, at the end of the pass, all edges leaving $u$ remain inadmissible, which completes the proof. ■

### The relabel-to-front algorithm

In the relabel-to-front algorithm, we maintain a linked list $L$ consisting of all vertices in $V - \{s, t\}$. A key property is that the vertices in $L$ are topologically sorted according to the admissible network, as we shall see in the loop invariant that follows. (Recall from Lemma 26.26 that the admissible network is a dag.)

The pseudocode for the relabel-to-front algorithm assumes that the neighbor lists $u.N$ have already been created for each vertex $u$. It also assumes that $u.next$ points to the vertex that follows $u$ in list $L$ and that, as usual, $u.next = \text{NIL}$ if $u$ is the last vertex in the list.

```
RELABEL-TO-FRONT(G, s, t)
 1  INITIALIZE-PREFLOW(G, s)
 2  L = G.V − {s, t}, in any order
 3  for each vertex u ∈ G.V − {s, t}
 4       u.current = u.N.head
 5  u = L.head
 6  while u ≠ NIL
 7       old-height = u.h
 8       DISCHARGE(u)
 9       if u.h > old-height
10            move u to the front of list L
11       u = u.next
```

The relabel-to-front algorithm works as follows. Line 1 initializes the preflow and heights to the same values as in the generic push-relabel algorithm. Line 2 initializes the list $L$ to contain all potentially overflowing vertices, in any order. Lines 3–4 initialize the *current* pointer of each vertex $u$ to the first vertex in $u$'s neighbor list.

As Figure 26.10 illustrates, the **while** loop of lines 6–11 runs through the list $L$, discharging vertices. Line 5 makes it start with the first vertex in the list. Each time through the loop, line 8 discharges a vertex $u$. If $u$ was relabeled by the DISCHARGE procedure, line 10 moves it to the front of list $L$. We can determine whether $u$ was relabeled by comparing its height before the discharge operation, saved into the variable *old-height* in line 7, with its height afterward, in line 9. Line 11 makes the next iteration of the **while** loop use the vertex following $u$ in list $L$. If line 10 moved $u$ to the front of the list, the vertex used in the next iteration is the one following $u$ in its new position in the list.

To show that RELABEL-TO-FRONT computes a maximum flow, we shall show that it is an implementation of the generic push-relabel algorithm. First, observe that it performs push and relabel operations only when they apply, since Lemma 26.29 guarantees that DISCHARGE performs them only when they apply. It remains to show that when RELABEL-TO-FRONT terminates, no basic operations apply. The remainder of the correctness argument relies on the following loop invariant:

> At each test in line 6 of RELABEL-TO-FRONT, list $L$ is a topological sort of the vertices in the admissible network $G_{f,h} = (V, E_{f,h})$, and no vertex before $u$ in the list has excess flow.

**Initialization:** Immediately after INITIALIZE-PREFLOW has been run, $s.h = |V|$ and $v.h = 0$ for all $v \in V - \{s\}$. Since $|V| \geq 2$ (because $V$ contains at

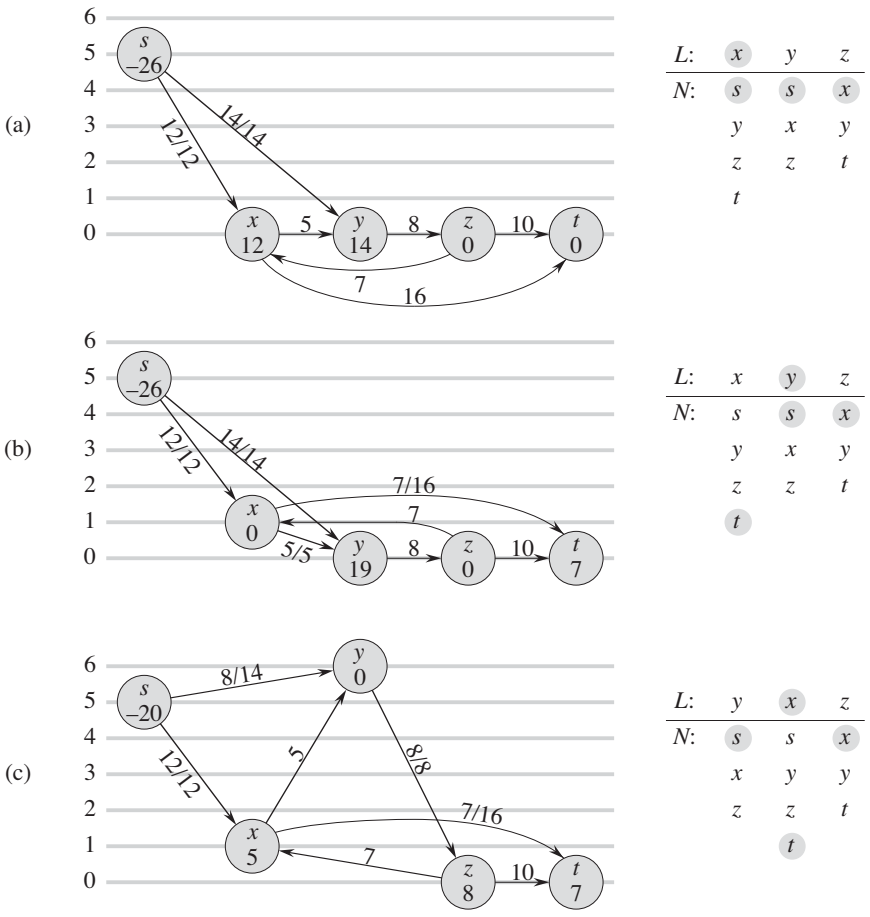



**Figure 26.10** The action of RELABEL-TO-FRONT. **(a)** A flow network just before the first iteration of the **while** loop. Initially, 26 units of flow leave source $s$. On the right is shown the initial list $L = \langle x, y, z \rangle$, where initially $u = x$. Under each vertex in list $L$ is its neighbor list, with the current neighbor shaded. Vertex $x$ is discharged. It is relabeled to height 1, 5 units of excess flow are pushed to $y$, and the 7 remaining units of excess are pushed to the sink $t$. Because $x$ is relabeled, it moves to the head of $L$, which in this case does not change the structure of $L$. **(b)** After $x$, the next vertex in $L$ that is discharged is $y$. Figure 26.9 shows the detailed action of discharging $y$ in this situation. Because $y$ is relabeled, it is moved to the head of $L$. **(c)** Vertex $x$ now follows $y$ in $L$, and so it is again discharged, pushing all 5 units of excess flow to $t$. Because vertex $x$ is not relabeled in this discharge operation, it remains in place in list $L$.

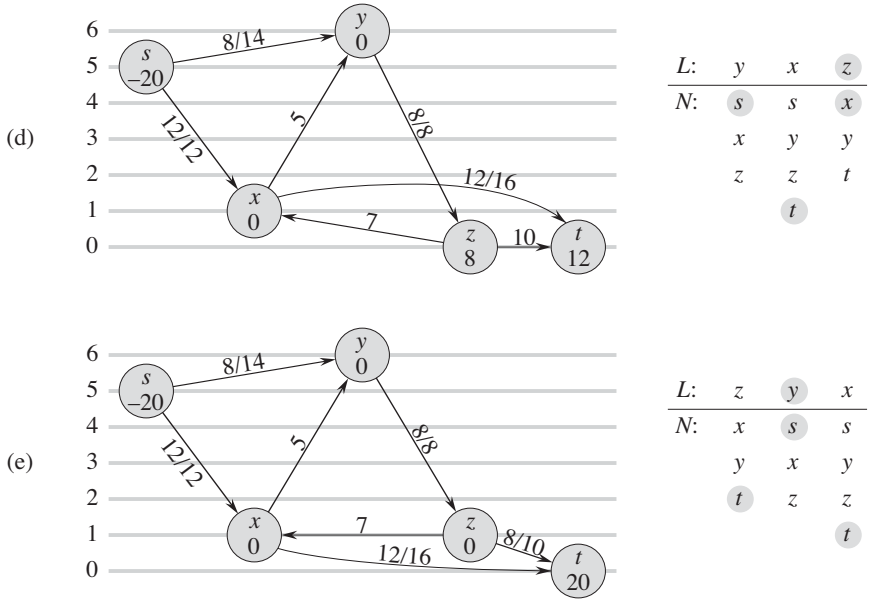



**Figure 26.10, continued** **(d)** Since vertex $z$ follows vertex $x$ in $L$, it is discharged. It is relabeled to height 1 and all 8 units of excess flow are pushed to $t$. Because $z$ is relabeled, it moves to the front of $L$. **(e)** Vertex $y$ now follows vertex $z$ in $L$ and is therefore discharged. But because $y$ has no excess, DISCHARGE immediately returns, and $y$ remains in place in $L$. Vertex $x$ is then discharged. Because it, too, has no excess, DISCHARGE again returns, and $x$ remains in place in $L$. RELABEL-TO-FRONT has reached the end of list $L$ and terminates. There are no overflowing vertices, and the preflow is a maximum flow.

least $s$ and $t$), no edge can be admissible. Thus, $E_{f,h} = \emptyset$, and any ordering of $V - \{s, t\}$ is a topological sort of $G_{f,h}$.

Because $u$ is initially the head of the list $L$, there are no vertices before it and so there are none before it with excess flow.

**Maintenance:** To see that each iteration of the **while** loop maintains the topological sort, we start by observing that the admissible network is changed only by push and relabel operations. By Lemma 26.27, push operations do not cause edges to become admissible. Thus, only relabel operations can create admissible edges. After a vertex $u$ is relabeled, however, Lemma 26.28 states that there are no admissible edges entering $u$ but there may be admissible edges leaving $u$. Thus, by moving $u$ to the front of $L$, the algorithm ensures that any admissible edges leaving $u$ satisfy the topological sort ordering.

To see that no vertex preceding $u$ in $L$ has excess flow, we denote the vertex that will be $u$ in the next iteration by $u'$. The vertices that will precede $u'$ in the next iteration include the current $u$ (due to line 11) and either no other vertices (if $u$ is relabeled) or the same vertices as before (if $u$ is not relabeled). When $u$ is discharged, it has no excess flow afterward. Thus, if $u$ is relabeled during the discharge, no vertices preceding $u'$ have excess flow. If $u$ is not relabeled during the discharge, no vertices before it on the list acquired excess flow during this discharge, because $L$ remained topologically sorted at all times during the discharge (as just pointed out, admissible edges are created only by relabeling, not pushing), and so each push operation causes excess flow to move only to vertices further down the list (or to $s$ or $t$). Again, no vertices preceding $u'$ have excess flow.

**Termination:** When the loop terminates, $u$ is just past the end of $L$, and so the loop invariant ensures that the excess of every vertex is 0. Thus, no basic operations apply.

### Analysis

We shall now show that RELABEL-TO-FRONT runs in $O(V^3)$ time on any flow network $G = (V, E)$. Since the algorithm is an implementation of the generic push-relabel algorithm, we shall take advantage of Corollary 26.21, which provides an $O(V)$ bound on the number of relabel operations executed per vertex and an $O(V^2)$ bound on the total number of relabel operations overall. In addition, Exercise 26.4-3 provides an $O(VE)$ bound on the total time spent performing relabel operations, and Lemma 26.22 provides an $O(VE)$ bound on the total number of saturating push operations.

***Theorem 26.30***
The running time of RELABEL-TO-FRONT on any flow network $G = (V, E)$ is $O(V^3)$.

***Proof*** Let us consider a "phase" of the relabel-to-front algorithm to be the time between two consecutive relabel operations. There are $O(V^2)$ phases, since there are $O(V^2)$ relabel operations. Each phase consists of at most $|V|$ calls to DISCHARGE, which we can see as follows. If DISCHARGE does not perform a relabel operation, then the next call to DISCHARGE is further down the list $L$, and the length of $L$ is less than $|V|$. If DISCHARGE does perform a relabel, the next call to DISCHARGE belongs to a different phase. Since each phase contains at most $|V|$ calls to DISCHARGE and there are $O(V^2)$ phases, the number of times DISCHARGE is called in line 8 of RELABEL-TO-FRONT is $O(V^3)$. Thus, the total

work performed by the **while** loop in RELABEL-TO-FRONT, excluding the work performed within DISCHARGE, is at most $O(V^3)$.

We must now bound the work performed within DISCHARGE during the execution of the algorithm. Each iteration of the **while** loop within DISCHARGE performs one of three actions. We shall analyze the total amount of work involved in performing each of these actions.

We start with relabel operations (lines 4–5). Exercise 26.4-3 provides an $O(VE)$ time bound on all the $O(V^2)$ relabels that are performed.

Now, suppose that the action updates the $u.current$ pointer in line 8. This action occurs $O(\text{degree}(u))$ times each time a vertex $u$ is relabeled, and $O(V \cdot \text{degree}(u))$ times overall for the vertex. For all vertices, therefore, the total amount of work done in advancing pointers in neighbor lists is $O(VE)$ by the handshaking lemma (Exercise B.4-1).

The third type of action performed by DISCHARGE is a push operation (line 7). We already know that the total number of saturating push operations is $O(VE)$. Observe that if a nonsaturating push is executed, DISCHARGE immediately returns, since the push reduces the excess to 0. Thus, there can be at most one nonsaturating push per call to DISCHARGE. As we have observed, DISCHARGE is called $O(V^3)$ times, and thus the total time spent performing nonsaturating pushes is $O(V^3)$.

The running time of RELABEL-TO-FRONT is therefore $O(V^3 + VE)$, which is $O(V^3)$. ■

## Exercises

### *26.5-1*

Illustrate the execution of RELABEL-TO-FRONT in the manner of Figure 26.10 for the flow network in Figure 26.1(a). Assume that the initial ordering of vertices in $L$ is $\langle v_1, v_2, v_3, v_4 \rangle$ and that the neighbor lists are

$$\begin{aligned} v_1.N &= \langle s, v_2, v_3 \rangle , \\ v_2.N &= \langle s, v_1, v_3, v_4 \rangle , \\ v_3.N &= \langle v_1, v_2, v_4, t \rangle , \\ v_4.N &= \langle v_2, v_3, t \rangle . \end{aligned}$$

### *26.5-2* ⋆

We would like to implement a push-relabel algorithm in which we maintain a first-in, first-out queue of overflowing vertices. The algorithm repeatedly discharges the vertex at the head of the queue, and any vertices that were not overflowing before the discharge but are overflowing afterward are placed at the end of the queue. After the vertex at the head of the queue is discharged, it is removed. When the

queue is empty, the algorithm terminates. Show how to implement this algorithm to compute a maximum flow in $O(V^3)$ time.

***26.5-3***
Show that the generic algorithm still works if RELABEL updates $u.h$ by simply computing $u.h = u.h + 1$. How would this change affect the analysis of RELABEL-TO-FRONT?

***26.5-4*** ⋆
Show that if we always discharge a highest overflowing vertex, we can make the push-relabel method run in $O(V^3)$ time.

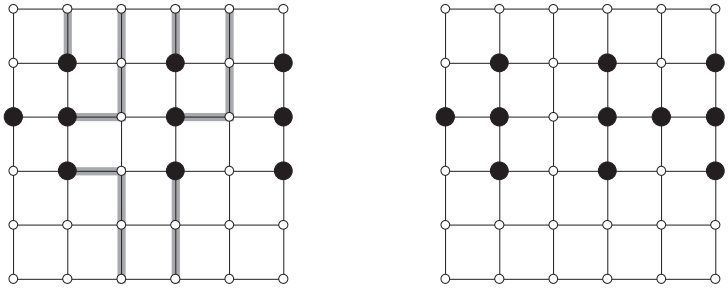
***26.5-5***
Suppose that at some point in the execution of a push-relabel algorithm, there exists an integer $0 < k \leq |V| - 1$ for which no vertex has $v.h = k$. Show that all vertices with $v.h > k$ are on the source side of a minimum cut. If such a $k$ exists, the ***gap heuristic*** updates every vertex $v \in V - \{s\}$ for which $v.h > k$, to set $v.h = \max(v.h, |V| + 1)$. Show that the resulting attribute $h$ is a height function. (The gap heuristic is crucial in making implementations of the push-relabel method perform well in practice.)

## Problems

***26-1 Escape problem***
An $n \times n$ ***grid*** is an undirected graph consisting of $n$ rows and $n$ columns of vertices, as shown in Figure 26.11. We denote the vertex in the $i$th row and the $j$th column by $(i, j)$. All vertices in a grid have exactly four neighbors, except for the boundary vertices, which are the points $(i, j)$ for which $i = 1$, $i = n$, $j = 1$, or $j = n$.

Given $m \leq n^2$ starting points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ in the grid, the ***escape problem*** is to determine whether or not there are $m$ vertex-disjoint paths from the starting points to any $m$ different points on the boundary. For example, the grid in Figure 26.11(a) has an escape, but the grid in Figure 26.11(b) does not.

***a.*** Consider a flow network in which vertices, as well as edges, have capacities. That is, the total positive flow entering any given vertex is subject to a capacity constraint. Show that determining the maximum flow in a network with edge and vertex capacities can be reduced to an ordinary maximum-flow problem on a flow network of comparable size.



(a) (b)

**Figure 26.11** Grids for the escape problem. Starting points are black, and other grid vertices are white. **(a)** A grid with an escape, shown by shaded paths. **(b)** A grid with no escape.

***b.*** Describe an efficient algorithm to solve the escape problem, and analyze its running time.

### *26-2 Minimum path cover*

A ***path cover*** of a directed graph $G = (V, E)$ is a set $P$ of vertex-disjoint paths such that every vertex in $V$ is included in exactly one path in $P$. Paths may start and end anywhere, and they may be of any length, including 0. A ***minimum path cover*** of $G$ is a path cover containing the fewest possible paths.

***a.*** Give an efficient algorithm to find a minimum path cover of a directed acyclic graph $G = (V, E)$. (*Hint:* Assuming that $V = \{1, 2, \ldots, n\}$, construct the graph $G' = (V', E')$, where

$$\begin{aligned} V' &= \{x_0, x_1, \ldots, x_n\} \cup \{y_0, y_1, \ldots, y_n\} \ , \\ E' &= \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\} \ , \end{aligned}$$

and run a maximum-flow algorithm.)

***b.*** Does your algorithm work for directed graphs that contain cycles? Explain.

### *26-3 Algorithmic consulting*

Professor Gore wants to open up an algorithmic consulting company. He has identified $n$ important subareas of algorithms (roughly corresponding to different portions of this textbook), which he represents by the set $A = \{A_1, A_2, \ldots, A_n\}$. In each subarea $A_k$, he can hire an expert in that area for $c_k$ dollars. The consulting company has lined up a set $J = \{J_1, J_2, \ldots, J_m\}$ of potential jobs. In order to perform job $J_i$, the company needs to have hired experts in a subset $R_i \subseteq A$ of

subareas. Each expert can work on multiple jobs simultaneously. If the company chooses to accept job $J_i$, it must have hired experts in all subareas in $R_i$, and it will take in revenue of $p_i$ dollars.

Professor Gore's job is to determine which subareas to hire experts in and which jobs to accept in order to maximize the net revenue, which is the total income from jobs accepted minus the total cost of employing the experts.

Consider the following flow network $G$. It contains a source vertex $s$, vertices $A_1, A_2, \ldots, A_n$, vertices $J_1, J_2, \ldots, J_m$, and a sink vertex $t$. For $k = 1, 2 \ldots, n$, the flow network contains an edge $(s, A_k)$ with capacity $c(s, A_k) = c_k$, and for $i = 1, 2, \ldots, m$, the flow network contains an edge $(J_i, t)$ with capacity $c(J_i, t) = p_i$. For $k = 1, 2, \ldots, n$ and $i = 1, 2, \ldots, m$, if $A_k \in R_i$, then $G$ contains an edge $(A_k, J_i)$ with capacity $c(A_k, J_i) = \infty$.

***a.*** Show that if $J_i \in T$ for a finite-capacity cut $(S, T)$ of $G$, then $A_k \in T$ for each $A_k \in R_i$.

***b.*** Show how to determine the maximum net revenue from the capacity of a minimum cut of $G$ and the given $p_i$ values.

***c.*** Give an efficient algorithm to determine which jobs to accept and which experts to hire. Analyze the running time of your algorithm in terms of $m$, $n$, and $r = \sum_{i=1}^{m} |R_i|$.

### *26-4 Updating maximum flow*

Let $G = (V, E)$ be a flow network with source $s$, sink $t$, and integer capacities. Suppose that we are given a maximum flow in $G$.

***a.*** Suppose that we increase the capacity of a single edge $(u, v) \in E$ by 1. Give an $O(V + E)$-time algorithm to update the maximum flow.

***b.*** Suppose that we decrease the capacity of a single edge $(u, v) \in E$ by 1. Give an $O(V + E)$-time algorithm to update the maximum flow.

### *26-5 Maximum flow by scaling*

Let $G = (V, E)$ be a flow network with source $s$, sink $t$, and an integer capacity $c(u, v)$ on each edge $(u, v) \in E$. Let $C = \max_{(u,v) \in E} c(u, v)$.

***a.*** Argue that a minimum cut of $G$ has capacity at most $C\,|E|$.

***b.*** For a given number $K$, show how to find an augmenting path of capacity at least $K$ in $O(E)$ time, if such a path exists.

We can use the following modification of FORD-FULKERSON-METHOD to compute a maximum flow in $G$:

```
MAX-FLOW-BY-SCALING(G, s, t)
1  C = max_{(u,v)∈E} c(u, v)
2  initialize flow f to 0
3  K = 2^⌊lg C⌋
4  while K ≥ 1
5      while there exists an augmenting path p of capacity at least K
6          augment flow f along p
7      K = K/2
8  return f
```

***c.*** Argue that MAX-FLOW-BY-SCALING returns a maximum flow.

***d.*** Show that the capacity of a minimum cut of the residual network $G_f$ is at most $2K|E|$ each time line 4 is executed.

***e.*** Argue that the inner **while** loop of lines 5–6 executes $O(E)$ times for each value of $K$.

***f.*** Conclude that MAX-FLOW-BY-SCALING can be implemented so that it runs in $O(E^2 \lg C)$ time.

### *26-6 The Hopcroft-Karp bipartite matching algorithm*

In this problem, we describe a faster algorithm, due to Hopcroft and Karp, for finding a maximum matching in a bipartite graph. The algorithm runs in $O(\sqrt{V}E)$ time. Given an undirected, bipartite graph $G = (V, E)$, where $V = L \cup R$ and all edges have exactly one endpoint in $L$, let $M$ be a matching in $G$. We say that a simple path $P$ in $G$ is an ***augmenting path*** with respect to $M$ if it starts at an unmatched vertex in $L$, ends at an unmatched vertex in $R$, and its edges belong alternately to $M$ and $E - M$. (This definition of an augmenting path is related to, but different from, an augmenting path in a flow network.) In this problem, we treat a path as a sequence of edges, rather than as a sequence of vertices. A shortest augmenting path with respect to a matching $M$ is an augmenting path with a minimum number of edges.

Given two sets $A$ and $B$, the ***symmetric difference*** $A \oplus B$ is defined as $(A - B) \cup (B - A)$, that is, the elements that are in exactly one of the two sets.

***a.*** Show that if $M$ is a matching and $P$ is an augmenting path with respect to $M$, then the symmetric difference $M \oplus P$ is a matching and $|M \oplus P| = |M| + 1$. Show that if $P_1, P_2, \ldots, P_k$ are vertex-disjoint augmenting paths with respect to $M$, then the symmetric difference $M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$ is a matching with cardinality $|M| + k$.

The general structure of our algorithm is the following:

```
HOPCROFT-KARP(G)
1  M = ∅
2  repeat
3       let P = {P_1, P_2, ..., P_k} be a maximal set of vertex-disjoint
              shortest augmenting paths with respect to M
4       M = M ⊕ (P_1 ∪ P_2 ∪ ··· ∪ P_k)
5  until P == ∅
6  return M
```

The remainder of this problem asks you to analyze the number of iterations in the algorithm (that is, the number of iterations in the **repeat** loop) and to describe an implementation of line 3.

***b.*** Given two matchings $M$ and $M^*$ in $G$, show that every vertex in the graph $G' = (V, M \oplus M^*)$ has degree at most 2. Conclude that $G'$ is a disjoint union of simple paths or cycles. Argue that edges in each such simple path or cycle belong alternately to $M$ or $M^*$. Prove that if $|M| \leq |M^*|$, then $M \oplus M^*$ contains at least $|M^*| - |M|$ vertex-disjoint augmenting paths with respect to $M$.

Let $l$ be the length of a shortest augmenting path with respect to a matching $M$, and let $P_1, P_2, \ldots, P_k$ be a maximal set of vertex-disjoint augmenting paths of length $l$ with respect to $M$. Let $M' = M \oplus (P_1 \cup \cdots \cup P_k)$, and suppose that $P$ is a shortest augmenting path with respect to $M'$.

***c.*** Show that if $P$ is vertex-disjoint from $P_1, P_2, \ldots, P_k$, then $P$ has more than $l$ edges.

***d.*** Now suppose that $P$ is not vertex-disjoint from $P_1, P_2, \ldots, P_k$. Let $A$ be the set of edges $(M \oplus M') \oplus P$. Show that $A = (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P$ and that $|A| \geq (k + 1)l$. Conclude that $P$ has more than $l$ edges.

***e.*** Prove that if a shortest augmenting path with respect to $M$ has $l$ edges, the size of the maximum matching is at most $|M| + |V|/(l + 1)$.

***f.*** Show that the number of **repeat** loop iterations in the algorithm is at most $2\sqrt{|V|}$. (*Hint:* By how much can $M$ grow after iteration number $\sqrt{|V|}$?)

***g.*** Give an algorithm that runs in $O(E)$ time to find a maximal set of vertex-disjoint shortest augmenting paths $P_1, P_2, \ldots, P_k$ for a given matching $M$. Conclude that the total running time of HOPCROFT-KARP is $O(\sqrt{V}E)$.

## Chapter notes

Ahuja, Magnanti, and Orlin [7], Even [103], Lawler [224], Papadimitriou and Steiglitz [271], and Tarjan [330] are good references for network flow and related algorithms. Goldberg, Tardos, and Tarjan [139] also provide a nice survey of algorithms for network-flow problems, and Schrijver [304] has written an interesting review of historical developments in the field of network flows.

The Ford-Fulkerson method is due to Ford and Fulkerson [109], who originated the formal study of many of the problems in the area of network flow, including the maximum-flow and bipartite-matching problems. Many early implementations of the Ford-Fulkerson method found augmenting paths using breadth-first search; Edmonds and Karp [102], and independently Dinic [89], proved that this strategy yields a polynomial-time algorithm. A related idea, that of using "blocking flows," was also first developed by Dinic [89]. Karzanov [202] first developed the idea of preflows. The push-relabel method is due to Goldberg [136] and Goldberg and Tarjan [140]. Goldberg and Tarjan gave an $O(V^3)$-time algorithm that uses a queue to maintain the set of overflowing vertices, as well as an algorithm that uses dynamic trees to achieve a running time of $O(VE \lg(V^2/E + 2))$. Several other researchers have developed push-relabel maximum-flow algorithms. Ahuja and Orlin [9] and Ahuja, Orlin, and Tarjan [10] gave algorithms that used scaling. Cheriyan and Maheshwari [62] proposed pushing flow from the overflowing vertex of maximum height. Cheriyan and Hagerup [61] suggested randomly permuting the neighbor lists, and several researchers [14, 204, 276] developed clever derandomizations of this idea, leading to a sequence of faster algorithms. The algorithm of King, Rao, and Tarjan [204] is the fastest such algorithm and runs in $O(VE \log_{E/(V \lg V)} V)$ time.

The asymptotically fastest algorithm to date for the maximum-flow problem, by Goldberg and Rao [138], runs in time $O(\min(V^{2/3}, E^{1/2})E \lg(V^2/E + 2) \lg C)$, where $C = \max_{(u,v)\in E} c(u, v)$. This algorithm does not use the push-relabel method but instead is based on finding blocking flows. All previous maximum-flow algorithms, including the ones in this chapter, use some notion of distance (the push-relabel algorithms use the analogous notion of height), with a length of 1

assigned implicitly to each edge. This new algorithm takes a different approach and assigns a length of 0 to high-capacity edges and a length of 1 to low-capacity edges. Informally, with respect to these lengths, shortest paths from the source to the sink tend have high capacity, which means that fewer iterations need be performed.

In practice, push-relabel algorithms currently dominate augmenting-path or linear-programming based algorithms for the maximum-flow problem. A study by Cherkassky and Goldberg [63] underscores the importance of using two heuristics when implementing a push-relabel algorithm. The first heuristic is to periodically perform a breadth-first search of the residual network in order to obtain more accurate height values. The second heuristic is the gap heuristic, described in Exercise 26.5-5. Cherkassky and Goldberg conclude that the best choice of push-relabel variants is the one that chooses to discharge the overflowing vertex with the maximum height.

The best algorithm to date for maximum bipartite matching, discovered by Hopcroft and Karp [176], runs in $O(\sqrt{V}E)$ time and is described in Problem 26-6. The book by Lovász and Plummer [239] is an excellent reference on matching problems.

# *VII Selected Topics*

## Introduction

This part contains a selection of algorithmic topics that extend and complement earlier material in this book. Some chapters introduce new models of computation such as circuits or parallel computers. Others cover specialized domains such as computational geometry or number theory. The last two chapters discuss some of the known limitations to the design of efficient algorithms and introduce techniques for coping with those limitations.

Chapter 27 presents an algorithmic model for parallel computing based on dynamic multithreading. The chapter introduces the basics of the model, showing how to quantify parallelism in terms of the measures of work and span. It then investigates several interesting multithreaded algorithms, including algorithms for matrix multiplication and merge sorting.

Chapter 28 studies efficient algorithms for operating on matrices. It presents two general methods—LU decomposition and LUP decomposition—for solving linear equations by Gaussian elimination in $O(n^3)$ time. It also shows that matrix inversion and matrix multiplication can be performed equally fast. The chapter concludes by showing how to compute a least-squares approximate solution when a set of linear equations has no exact solution.

Chapter 29 studies linear programming, in which we wish to maximize or minimize an objective, given limited resources and competing constraints. Linear programming arises in a variety of practical application areas. This chapter covers how to formulate and solve linear programs. The solution method covered is the simplex algorithm, which is the oldest algorithm for linear programming. In contrast to many algorithms in this book, the simplex algorithm does not run in polynomial time in the worst case, but it is fairly efficient and widely used in practice.

Chapter 30 studies operations on polynomials and shows how to use a well-known signal-processing technique—the fast Fourier transform (FFT)—to multiply two degree-$n$ polynomials in $O(n \lg n)$ time. It also investigates efficient implementations of the FFT, including a parallel circuit.

Chapter 31 presents number-theoretic algorithms. After reviewing elementary number theory, it presents Euclid's algorithm for computing greatest common divisors. Next, it studies algorithms for solving modular linear equations and for raising one number to a power modulo another number. Then, it explores an important application of number-theoretic algorithms: the RSA public-key cryptosystem. This cryptosystem can be used not only to encrypt messages so that an adversary cannot read them, but also to provide digital signatures. The chapter then presents the Miller-Rabin randomized primality test, with which we can find large primes efficiently—an essential requirement for the RSA system. Finally, the chapter covers Pollard's "rho" heuristic for factoring integers and discusses the state of the art of integer factorization.

Chapter 32 studies the problem of finding all occurrences of a given pattern string in a given text string, a problem that arises frequently in text-editing programs. After examining the naive approach, the chapter presents an elegant approach due to Rabin and Karp. Then, after showing an efficient solution based on finite automata, the chapter presents the Knuth-Morris-Pratt algorithm, which modifies the automaton-based algorithm to save space by cleverly preprocessing the pattern.

Chapter 33 considers a few problems in computational geometry. After discussing basic primitives of computational geometry, the chapter shows how to use a "sweeping" method to efficiently determine whether a set of line segments contains any intersections. Two clever algorithms for finding the convex hull of a set of points—Graham's scan and Jarvis's march—also illustrate the power of sweeping methods. The chapter closes with an efficient algorithm for finding the closest pair from among a given set of points in the plane.

Chapter 34 concerns NP-complete problems. Many interesting computational problems are NP-complete, but no polynomial-time algorithm is known for solving any of them. This chapter presents techniques for determining when a problem is NP-complete. Several classic problems are proved to be NP-complete: determining whether a graph has a hamiltonian cycle, determining whether a boolean formula is satisfiable, and determining whether a given set of numbers has a subset that adds up to a given target value. The chapter also proves that the famous traveling-salesman problem is NP-complete.

Chapter 35 shows how to find approximate solutions to NP-complete problems efficiently by using approximation algorithms. For some NP-complete problems, approximate solutions that are near optimal are quite easy to produce, but for others even the best approximation algorithms known work progressively more poorly as

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 three "concurrency" keywords: **parallel**, **spawn**, and **sync**. 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 $n$th Fibonacci number:

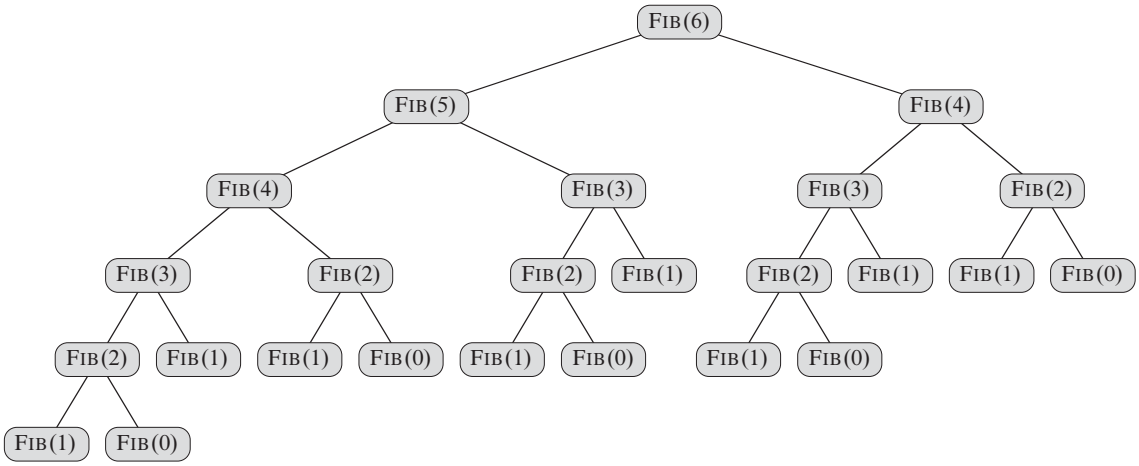



**Figure 27.1** The tree of recursive procedure instances when computing FIB(6). Each instance of FIB with the same argument does the same work to produce the same result, providing an inefficient but interesting way to compute Fibonacci numbers.

```
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**. 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.

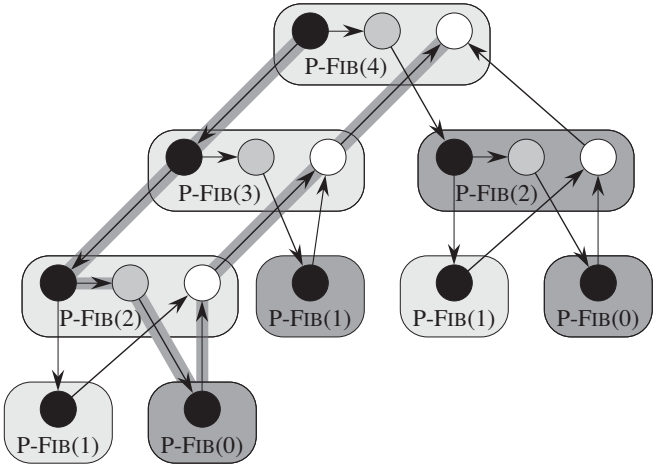



**Figure 27.2** A directed acyclic graph representing the computation of P-FIB(4). Each circle represents one strand, with black circles representing either base cases or the part of the procedure (instance) up to the spawn of P-FIB($n-1$) in line 3, shaded circles representing the part of the procedure 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 $T_\infty$.

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 $T_1/T_\infty$ 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 $T_1/T_P \le T_1/T_\infty < P$. 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 $T_1/T_P \le T_1/T_\infty < P$. 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 $T_\infty$ 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

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

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 $T_\infty$.

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 $T_\infty$ 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 $T_\infty$ 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 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. 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:

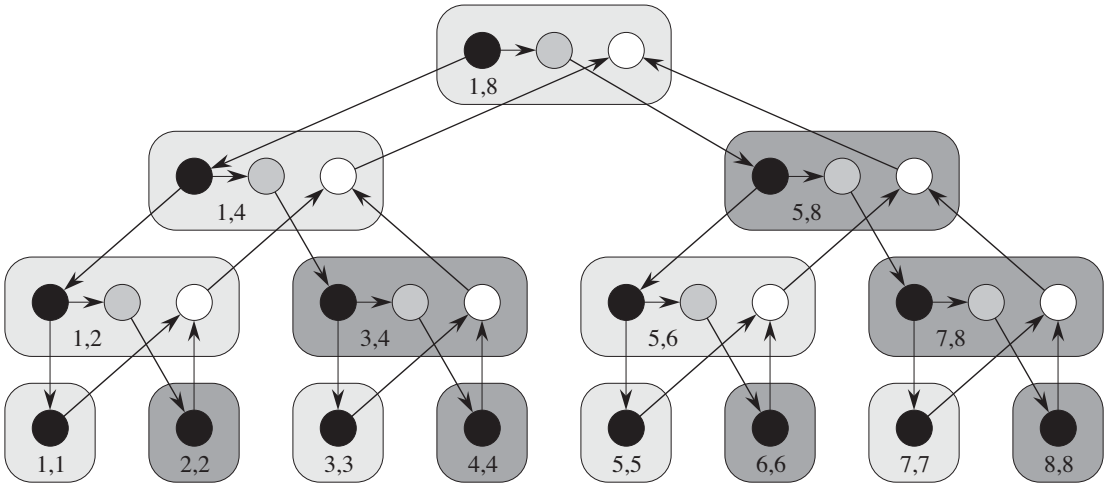



**Figure 27.4** A dag representing the computation of MAT-VEC-MAIN-LOOP$(A, x, y, 8, 1, 8)$. The two numbers within each rounded rectangle give the values of the last two parameters ($i$ and $i'$ in the procedure header) in the invocation (spawn or call) of the procedure. The black circles represent 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 $i$th 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
```

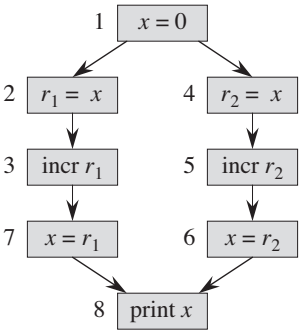
After initializing $x$ to 0 in line 1, RACE-EXAMPLE creates two parallel strands, each of which increments $x$ in line 3. Although it might seem that RACE-EXAMPLE should always print the value 2 (its serialization certainly does), it could instead print the value 1. Let's see how this anomaly might occur.

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. 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 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 processsors. 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 proven 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.)