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 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 programs 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 straighforward 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 j = 1 to n
5          c_ij = 0
6          for 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 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 in 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 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:

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

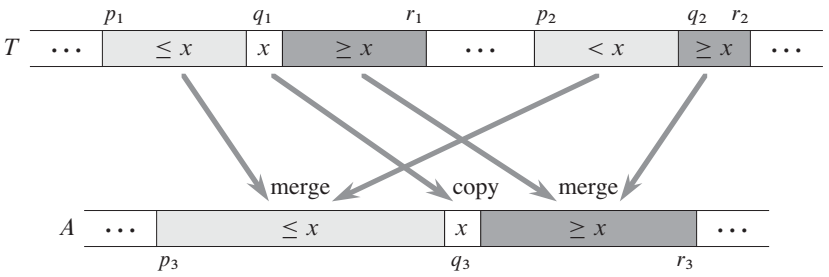

Figure 27.6 The idea behind the multithreaded merging of two sorted subarrays $T[p_1 \,..\, r_1]$ and $T[p_2 \,..\, r_2]$ into the subarray $A[p_3 \,..\, r_3]$. Letting $x = T[q_1]$ be the median of $T[p_1 \,..\, r_1]$ and q_2 be the place in $T[p_2 \,..\, r_2]$ such that x would fall between $T[q_2 - 1]$ 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 $T[q_2 \,..\, p_2]$ 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].

28 Matrix Operations

Because operations on matrices lie at the heart of scientific computing, efficient algorithms for working with matrices have many practical applications. This chapter focuses on how to multiply matrices and solve sets of simultaneous linear equations. Appendix D reviews the basics of matrices.

Section 28.1 shows how to solve a set of linear equations using LUP decompositions. Then, Section 28.2 explores the close relationship between multiplying and inverting matrices. Finally, Section 28.3 discusses the important class of symmetric positive-definite matrices and shows how we can use them to find a least-squares solution to an overdetermined set of linear equations.

One important issue that arises in practice is ***numerical stability***. Due to the limited precision of floating-point representations in actual computers, round-off errors in numerical computations may become amplified over the course of a computation, leading to incorrect results; we call such computations ***numerically unstable***. Although we shall briefly consider numerical stability on occasion, we do not focus on it in this chapter. We refer you to the excellent book by Golub and Van Loan [144] for a thorough discussion of stability issues.

28.1 Solving systems of linear equations

Numerous applications need to solve sets of simultaneous linear equations. We can formulate a linear system as a matrix equation in which each matrix or vector element belongs to a field, typically the real numbers $\mathbb{R}$. This section discusses how to solve a system of linear equations using a method called LUP decomposition.

We start with a set of linear equations in n unknowns $x_1, x_2, \ldots, x_n$:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 , \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 , \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n . \end{aligned} \tag{28.1}$$

A ***solution*** to the equations (28.1) is a set of values for $x_1, x_2, \ldots, x_n$ that satisfy all of the equations simultaneously. In this section, we treat only the case in which there are exactly n equations in n unknowns.

We can conveniently rewrite equations (28.1) as the matrix-vector equation

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

or, equivalently, letting $A = (a_{ij})$, $x = (x_i)$, and $b = (b_i)$, as

$$Ax = b . \tag{28.2}$$

If A is nonsingular, it possesses an inverse A^{-1}, and

$$x = A^{-1}b \tag{28.3}$$

is the solution vector. We can prove that x is the unique solution to equation (28.2) as follows. If there are two solutions, x and x', then $Ax = Ax' = b$ and, letting I denote an identity matrix,

$$\begin{aligned} x &= Ix \\ &= (A^{-1}A)x \\ &= A^{-1}(Ax) \\ &= A^{-1}(Ax') \\ &= (A^{-1}A)x' \\ &= x' . \end{aligned}$$

In this section, we shall be concerned predominantly with the case in which A is nonsingular or, equivalently (by Theorem D.1), the rank of A is equal to the number n of unknowns. There are other possibilities, however, which merit a brief discussion. If the number of equations is less than the number n of unknowns—or, more generally, if the rank of A is less than n—then the system is ***underdetermined***. An underdetermined system typically has infinitely many solutions, although it may have no solutions at all if the equations are inconsistent. If the number of equations exceeds the number n of unknowns, the system is ***overdetermined***, and there may not exist any solutions. Section 28.3 addresses the important

problem of finding good approximate solutions to overdetermined systems of linear equations.

Let us return to our problem of solving the system $Ax = b$ of n equations in n unknowns. We could compute A^{-1} and then, using equation (28.3), multiply b by A^{-1}, yielding $x = A^{-1}b$. This approach suffers in practice from numerical instability. Fortunately, another approach—LUP decomposition—is numerically stable and has the further advantage of being faster in practice.

Overview of LUP decomposition

The idea behind LUP decomposition is to find three $n \times n$ matrices L, U, and P such that

$$PA = LU \ , \tag{28.4}$$

where

- L is a unit lower-triangular matrix,
- U is an upper-triangular matrix, and
- P is a permutation matrix.

We call matrices L, U, and P satisfying equation (28.4) an ***LUP decomposition*** of the matrix A. We shall show that every nonsingular matrix A possesses such a decomposition.

Computing an LUP decomposition for the matrix A has the advantage that we can more easily solve linear systems when they are triangular, as is the case for both matrices L and U. Once we have found an LUP decomposition for A, we can solve equation (28.2), $Ax = b$, by solving only triangular linear systems, as follows. Multiplying both sides of $Ax = b$ by P yields the equivalent equation $PAx = Pb$, which, by Exercise D.1-4, amounts to permuting the equations (28.1). Using our decomposition (28.4), we obtain

$$LUx = Pb \ .$$

We can now solve this equation by solving two triangular linear systems. Let us define $y = Ux$, where x is the desired solution vector. First, we solve the lower-triangular system

$$Ly = Pb \tag{28.5}$$

for the unknown vector y by a method called "forward substitution." Having solved for y, we then solve the upper-triangular system

$$Ux = y \tag{28.6}$$

for the unknown x by a method called "back substitution." Because the permutation matrix P is invertible (Exercise D.2-3), multiplying both sides of equation (28.4) by P^{-1} gives $P^{-1}PA = P^{-1}LU$, so that

$$A = P^{-1}LU \; . \tag{28.7}$$

Hence, the vector x is our solution to $Ax = b$:

$$\begin{aligned} Ax &= P^{-1}LUx && \text{(by equation (28.7))} \\ &= P^{-1}Ly && \text{(by equation (28.6))} \\ &= P^{-1}Pb && \text{(by equation (28.5))} \\ &= b \; . \end{aligned}$$

Our next step is to show how forward and back substitution work and then attack the problem of computing the LUP decomposition itself.

Forward and back substitution

Forward substitution can solve the lower-triangular system (28.5) in $\Theta(n^2)$ time, given L, P, and b. For convenience, we represent the permutation P compactly by an array $\pi[1..n]$. For $i = 1, 2, \ldots, n$, the entry $\pi[i]$ indicates that $P_{i,\pi[i]} = 1$ and $P_{ij} = 0$ for $j \neq \pi[i]$. Thus, PA has $a_{\pi[i],j}$ in row i and column j, and Pb has $b_{\pi[i]}$ as its ith element. Since L is unit lower-triangular, we can rewrite equation (28.5) as

$$\begin{aligned} y_1 &= b_{\pi[1]} \; , \\ l_{21}y_1 + y_2 &= b_{\pi[2]} \; , \\ l_{31}y_1 + l_{32}y_2 + y_3 &= b_{\pi[3]} \; , \\ &\vdots \\ l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \cdots + y_n &= b_{\pi[n]} \; . \end{aligned}$$

The first equation tells us that $y_1 = b_{\pi[1]}$. Knowing the value of y_1, we can substitute it into the second equation, yielding

$$y_2 = b_{\pi[2]} - l_{21}y_1 \; .$$

Now, we can substitute both y_1 and y_2 into the third equation, obtaining

$$y_3 = b_{\pi[3]} - (l_{31}y_1 + l_{32}y_2) \; .$$

In general, we substitute $y_1, y_2, \ldots, y_{i-1}$ "forward" into the ith equation to solve for y_i:

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij} y_j \ .$$

Having solved for y, we solve for x in equation (28.6) using ***back substitution***, which is similar to forward substitution. Here, we solve the nth equation first and work backward to the first equation. Like forward substitution, this process runs in $\Theta(n^2)$ time. Since U is upper-triangular, we can rewrite the system (28.6) as

$$\begin{aligned}
u_{11}x_1 + u_{12}x_2 + \cdots + u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n &= y_1 \ , \\
u_{22}x_2 + \cdots + u_{2,n-2}x_{n-2} + u_{2,n-1}x_{n-1} + u_{2n}x_n &= y_2 \ , \\
&\vdots \\
u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n &= y_{n-2} \ , \\
u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1} \ , \\
u_{n,n}x_n &= y_n \ .
\end{aligned}$$

Thus, we can solve for $x_n, x_{n-1}, \ldots, x_1$ successively as follows:

$$\begin{aligned}
x_n &= y_n/u_{n,n} \ , \\
x_{n-1} &= (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1} \ , \\
x_{n-2} &= (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n))/u_{n-2,n-2} \ , \\
&\vdots
\end{aligned}$$

or, in general,

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) / u_{ii} \ .$$

Given P, L, U, and b, the procedure LUP-SOLVE solves for x by combining forward and back substitution. The pseudocode assumes that the dimension n appears in the attribute $L.rows$ and that the permutation matrix P is represented by the array π.

```
LUP-SOLVE(L, U, π, b)
1  n = L.rows
2  let x be a new vector of length n
3  for i = 1 to n
4       y_i = b_{π[i]} − Σ_{j=1}^{i−1} l_ij y_j
5  for i = n downto 1
6       x_i = (y_i − Σ_{j=i+1}^{n} u_ij x_j) / u_ii
7  return x
```

Procedure LUP-SOLVE solves for y using forward substitution in lines 3–4, and then it solves for x using backward substitution in lines 5–6. Since the summation within each of the **for** loops includes an implicit loop, the running time is $\Theta(n^2)$.

As an example of these methods, consider the system of linear equations defined by

$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} x = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix},$$

where

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix},$$

$$b = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix},$$

and we wish to solve for the unknown x. The LUP decomposition is

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix},$$

$$U = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix},$$

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

(You might want to verify that $PA = LU$.) Using forward substitution, we solve $Ly = Pb$ for y:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 3 \\ 7 \end{pmatrix},$$

obtaining

$$y = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix}$$

by computing first y_1, then y_2, and finally y_3. Using back substitution, we solve $Ux = y$ for x:

$$\begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix},$$

thereby obtaining the desired answer

$$x = \begin{pmatrix} -1.4 \\ 2.2 \\ 0.6 \end{pmatrix}$$

by computing first x_3, then x_2, and finally x_1.

Computing an LU decomposition

We have now shown that if we can create an LUP decomposition for a nonsingular matrix A, then forward and back substitution can solve the system $Ax = b$ of linear equations. Now we show how to efficiently compute an LUP decomposition for A. We start with the case in which A is an $n \times n$ nonsingular matrix and P is absent (or, equivalently, $P = I_n$). In this case, we factor $A = LU$. We call the two matrices L and U an ***LU decomposition*** of A.

We use a process known as ***Gaussian elimination*** to create an LU decomposition. We start by subtracting multiples of the first equation from the other equations in order to remove the first variable from those equations. Then, we subtract multiples of the second equation from the third and subsequent equations so that now the first and second variables are removed from them. We continue this process until the system that remains has an upper-triangular form—in fact, it is the matrix U. The matrix L is made up of the row multipliers that cause variables to be eliminated.

Our algorithm to implement this strategy is recursive. We wish to construct an LU decomposition for an $n \times n$ nonsingular matrix A. If $n = 1$, then we are done, since we can choose $L = I_1$ and $U = A$. For $n > 1$, we break A into four parts:

$$\begin{aligned} A &= \left(\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}\right) \\ &= \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix}, \end{aligned}$$

where v is a column $(n-1)$-vector, w^{T} is a row $(n-1)$-vector, and A' is an $(n-1) \times (n-1)$ matrix. Then, using matrix algebra (verify the equations by

simply multiplying through), we can factor A as

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} . \end{aligned} \tag{28.8}$$

The 0s in the first and second matrices of equation (28.8) are row and column $(n-1)$-vectors, respectively. The term vw^{T}/a_{11}, formed by taking the outer product of v and w and dividing each element of the result by a_{11}, is an $(n-1) \times (n-1)$ matrix, which conforms in size to the matrix A' from which it is subtracted. The resulting $(n-1) \times (n-1)$ matrix

$$A' - vw^{\mathrm{T}}/a_{11} \tag{28.9}$$

is called the ***Schur complement*** of A with respect to a_{11}.

We claim that if A is nonsingular, then the Schur complement is nonsingular, too. Why? Suppose that the Schur complement, which is $(n-1) \times (n-1)$, is singular. Then by Theorem D.1, it has row rank strictly less than $n-1$. Because the bottom $n-1$ entries in the first column of the matrix

$$\begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix}$$

are all 0, the bottom $n-1$ rows of this matrix must have row rank strictly less than $n-1$. The row rank of the entire matrix, therefore, is strictly less than n. Applying Exercise D.2-8 to equation (28.8), A has rank strictly less than n, and from Theorem D.1 we derive the contradiction that A is singular.

Because the Schur complement is nonsingular, we can now recursively find an LU decomposition for it. Let us say that

$$A' - vw^{\mathrm{T}}/a_{11} = L'U' ,$$

where L' is unit lower-triangular and U' is upper-triangular. Then, using matrix algebra, we have

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} \\ &= LU , \end{aligned}$$

thereby providing our LU decomposition. (Note that because L' is unit lower-triangular, so is L, and because U' is upper-triangular, so is U.)

Of course, if $a_{11} = 0$, this method doesn't work, because it divides by 0. It also doesn't work if the upper leftmost entry of the Schur complement $A' - vw^{\mathrm{T}}/a_{11}$ is 0, since we divide by it in the next step of the recursion. The elements by which we divide during LU decomposition are called ***pivots***, and they occupy the diagonal elements of the matrix U. The reason we include a permutation matrix P during LUP decomposition is that it allows us to avoid dividing by 0. When we use permutations to avoid division by 0 (or by small numbers, which would contribute to numerical instability), we are ***pivoting***.

An important class of matrices for which LU decomposition always works correctly is the class of symmetric positive-definite matrices. Such matrices require no pivoting, and thus we can employ the recursive strategy outlined above without fear of dividing by 0. We shall prove this result, as well as several others, in Section 28.3.

Our code for LU decomposition of a matrix A follows the recursive strategy, except that an iteration loop replaces the recursion. (This transformation is a standard optimization for a "tail-recursive" procedure—one whose last operation is a recursive call to itself. See Problem 7-4.) It assumes that the attribute $A.rows$ gives the dimension of A. We initialize the matrix U with 0s below the diagonal and matrix L with 1s on its diagonal and 0s above the diagonal.

LU-DECOMPOSITION(A)

```
 1  n = A.rows
 2  let L and U be new n × n matrices
 3  initialize U with 0s below the diagonal
 4  initialize L with 1s on the diagonal and 0s above the diagonal
 5  for k = 1 to n
 6      u_kk = a_kk
 7      for i = k + 1 to n
 8          l_ik = a_ik/u_kk          // l_ik holds v_i
 9          u_ki = a_ki               // u_ki holds w_i^T
10      for i = k + 1 to n
11          for j = k + 1 to n
12              a_ij = a_ij − l_ik u_kj
13  return L and U
```

The outer **for** loop beginning in line 5 iterates once for each recursive step. Within this loop, line 6 determines the pivot to be $u_{kk} = a_{kk}$. The **for** loop in lines 7–9 (which does not execute when $k = n$), uses the v and w^{T} vectors to update L and U. Line 8 determines the elements of the v vector, storing v_i in l_{ik}, and line 9 computes the elements of the w^{T} vector, storing w_i^{T} in u_{ki}. Finally, lines 10–12 compute the elements of the Schur complement and store them back into the ma-