27 Multithreaded Algorithms

27.1 The basics of dynamic multithreading

27.1-1

> Suppose that we spawn $\text{P-FIB}(n-2)$ 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?

(Removed)

27.1-2

> Draw the computation dag that results from executing $\text{P-FIB}(5)$. 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.

- Work: $T_1 = 29$.
- Span: $T_\infty = 10$.
- Parallelism: $T_1/T_\infty \approx 2.9$.

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

Suppose that there are x incomplete steps in a run of the program. Since each of these steps causes at least one unit of work to be done, we have that there is at most $(T_1 - x)$ units of work done in the complete steps. Then, we suppose by contradiction that the number of complete steps is strictly greater than $\lfloor (T_1 - x)/P \rfloor$. Then, we have that the total amount of work done during the complete steps is

$$P \cdot (\lfloor (T_1 - x)/P \rfloor + 1) = P\lfloor (T_1 - x)/P = (T_1 - x) - ((T_1 - x) \mod P) + P > T_1 - x.$$

This is a contradiction because there are only $(T_1 - x)$ units of work done during complete steps, which is less than the amount we would be doing. Notice that since T_∞ is abound on the total number of both kinds of steps, it is a bound on the number of incomplete steps, x, so,

$$T_P \le \lfloor (T_1 - x)/P \rfloor + x \le \lfloor (T_1 - T_\infty)/P \rfloor + T_\infty \,.$$

Where the second inequality comes by noting that the middle expression, as a function of x is monotonically increasing, and so is bounded by the largest value of x that is possible, namely T_∞ .

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.

The computation is given in the image below. Let vertex u have degree k, and assume that there are m vertices in each vertical chain. Assume that this is executed on k processors. In one execution, each strand from among the k on the left is executed concurrently, and then the m strands on the right are executed one at a time. If each strand takes unit time to execute, then the total computation takes $2m$ time. On the other hand, suppose that on each time step of the computation, $k - 1$ strands from the left (descendants of u) are executed, and one from the right (a descendant of v), is executed. If each strand take unit time to executed, the total computation takes $m + m/k$. Thus, the ratio of times is $2m/(m + m/k) = 2/(1 + 1/k)$. As k gets large, this approaches 2 as desired.

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.)

(Removed)

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.

(Removed)

27.1-7

> Consider the following multithreaded pseudocode for transposing an $n \times n$ matrix A in place:
>
> ```
> P-TRANSPOSE(A)
>     n = A.rows
>     parallel for j = 2 to n
>         parallel for i = 1 to j - 1
>             exchange a[i, j] with a[j, i]
> ```
>
> Analyze the work, span, and parallelism of this algorithm.

(Removed)

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.

(Removed)

27.1-9

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

(Removed)

27.2 Multithreaded matrix multiplication

27.2-1

> Draw the computation dag for computing $\text{P-SQUARE-MATRIX-MULTIPLY}$ 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.

(Omit!)

27.2-2

> Repeat Exercise 27.2-1 for $\text{P-MATRIX-MULTIPLY-RECURSIVE}$.

(Omit!)

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.

(Removed)

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.

(Removed)

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.

```
P-MATRIX-TRANSPOSE(A)
    n = A.rows
    if n == 1
        return
    partition A into n / 2 × n / 2 submatrices A11, A12, A21, A22
    spawn P-MATRIX-TRANSPOSE(A11)
    spawn P-MATRIX-TRANSPOSE(A12)
    spawn P-MATRIX-TRANSPOSE(A21)
    P-MATRIX-TRANSPOSE(A22)
    sync
    // exchange A12 with A21
    parallel for i = 1 to n / 2
        parallel for j = 1 + n / 2 to n
            exchange A[i, j] with A[i + n / 2, j - n / 2]
```

- span: $T(n) = T(n/2) + O(\lg n) = O(\lg^2 n)$.
- work: $T(n) = 4T(n/2) + O(n^2) = O(n^2 \lg n)$.

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.

(Removed)

27.3 Multithreaded merge sort

27.3-1

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

Replace the condition on line 2 with a check that $n < k$ for some base case size k. And instead of just copying over the particular element of A to the right spot in B, you would call a serial sort on the remaining segment of A and copy the result of that over into the right spots in B.

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.

By a slight modification of exercise 9.3-8 we can find we can find the median of all elements in two sorted arrays of total length n in $O(\lg n)$ time. We'll modify P-MERGE to use this fact. Let $\text{MEDIAN}(T, p_1, r_1, p_2, r_2)$ be the function which returns a pair, q, where $q.pos$ is the position of the median of all the elements T which lie between positions p_1 and r_1, and between positions p_2 and r_2, and $q.arr$ is 1 if the position is between p_1 and r_1, and 2 otherwise.

```
P-MEDIAN-MERGE(T, p[1], r[1], p[2], r[2], A, p[3])
    n[1] = r[1] - p[1] + 1
    n[2] = r[2] - p[2] + 1
    if n[1] < n[2]              // ensure that n[1] ≥ n[2]
        exchange p[1] with p[2]
        exchange r[1] with r[2]
        exchange n[1] with n[2]
    if n[1] == 0              // both empty?
        return
    q = MEDIAN(T, p[1], r[1], p[2], r[2])
    if q.arr == 1
        q[2] = BINARY-SEARCH(T[q.pos], T, p[2], r[2])
        q[3] = p[3] + q.pos - p[1] + q[2] - p[2]
        A[q[3]] = T[q.pos]
        spawn P-MEDIAN-MERGE(T, p[1], q.pos - 1, p[2], q[2] - 1, A, p[3])
        P-MEDIAN-MERGE(T, q.pos + 1, r[1], q[2] + 1, r[2], A, p[3])
        sync
    else
        q[2] = BINARY-SEARCH(T[q.pos], T, p[1], r[1])
        q[3] = p[3] + q.pos - p[2] + q[2] - p[1]
        A[q[3]] = T[q.pos]
        spawn P-MEDIAN-MERGE(T, p[1], q[2] - 1, p[2], q.pos - 1, A, p[3])
        P-MEDIAN-MERGE(T, q[2] + 1, r[1], q.pos + 1, r[2], A, p[3])
        sync
```

The work is characterized by the recurrence $T_1(n) = O(\lg n) + 2T_1(n/2)$, whose solution tells us that $T_1(n) = O(n)$. The work is at least $\Omega(n)$ since we need to examine each element, so the work is $\Theta(n)$. The span satisfies the recurrence

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

by exercise 4.6-2.

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.)

Suppose that there are c different processors, and the array has length n and you are going to use its last element as a pivot. Then, look at each chunk of size $\lceil \frac{n}{c} \rceil$ of entries before the last element, give one to each processor. Then, each counts the number of elements that are less than the pivot. Then, we compute all the running sums of these values that are returned. This can be done easily by considering all of the subarrays placed along the leaves of a binary tree, and then summing up adjacent pairs. This computation can be done in time $\lg(\min\{c, n\})$ since it's the log of the number of leaves. From there, we can compute all the running sums for each of the subarrays also in logarithmic time. This is by keeping track of the sum of all more left cousins of each internal node, which is found by adding the left sibling's sum vale to the left cousin value of the parent, with the root's left cousin value initiated to 0. This also just takes time the depth of the tree, so is $\lg(\min\{c, n\})$. Once all of these values are computed at the root, it is the index that the subarray's elements less than the pivot should be put. To find the position where the subarray's elements larger than the root should be put, just put it at twice the sum value of the root minus the left cousin value for that subarray. Then, the time taken is just $O(\frac{n}{c})$. By doing this procedure, the total work is just $O(n)$, and the span is $O(\lg n)$, and so has parallelization of $O(\frac{n}{\lg n})$. This whole process is split across the several algoithms appearing here.

27.3-4

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

```
P-RECURSIVE-FFT(a)
    n = a.length
    if n == 1
        return a
    w[n] = e^{2 * π * i / n}
    w = 1
    a(0) = [a[0], a[2]..a[n - 2]]
    a(1) = [a[1], a[3]..a[n - 1]]
    y(0) = spawn P-RECURSIVE-FFT(a[0])
    y(1) = P-RECURSIVE-FFT(a[1])
    sync
    parallel for k = 0 to n / 2 - 1
        y[k] = y[k](0) + w * y[k](1)
        y[k + n / 2] = y[k](0) - w * y[k](1)
        w = w * w[n]
    return y
```

P-RECURSIVE-FFT parallelized over the two recursive calls, having a parallel for works because each of the iterations of the **for** loop touch independent sets of variables. The span of the procedure is only $\Theta(\lg n)$ giving

it a parallelization of $\Theta(n)$.

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.)

Randomly pick a pivot element, swap it with the last element, so that it is in the correct format for running the procedure described in 27.3-3. Run partition from problem 27.3-3. As an intermediate step, in that procedure, we compute the number of elements less than the pivot ($T.root.sum$), so keep track of that value after the end of partition. Then, if we have that it is less than k, recurse on the subarray that was greater than or equal to the pivot, decreasing the order statistic of the element to be selected by $T.root.sum$. If it is larger than the order statistic of the element to be selected, then leave it unchanged and recurse on the subarray that was formed to be less than the pivot. A lot of the analysis in section 9.2 still applies, except replacing the timer needed for partitioning with the runtime of the algorithm in problem 27.3-3. The work is unchanged from the serial case because when $c = 1$, the algorithm reduces to the serial algorithm for partitioning. For span, the $O(n)$ term in the equation half way down page 218 can be replaced with an $O(\lg n)$ term. It can be seen with the substitution method that the solution to this is logarithmic

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} C \lg k + O(\lg n) \leq O(\lg n).$$

So, the total span of this algorithm will still just be $O(\lg n)$.

27.3-6 ⋆

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

Let $MEDIAN(A)$ denote a brute force method which returns the median element of the array A. We will only use this to find the median of small arrays, in particular, those of size at most 5, so it will always run in constant time. We also let $A[i..j]$ denote the array whose elements are $A[i], A[i+1], \ldots, A[j]$. The function $P\text{-}PARTITION(A, x)$ is a multithreaded function which partitions A around the input element x and returns the number of elements in A which are less than or equal to x. Using a parallel **for** loop, its span is logarithmic in the number of elements in A. The work is the same as the serialization, which is $\Theta(n)$ according to section 9.3. The span satisfies the recurrence

$$\begin{aligned} T_\infty(n) &= \Theta(\lg n/5) + T_\infty(n/5) + \Theta(\lg n) + T_\infty(7n/10 + 6) \\ &\leq \Theta(\lg n) + T_\infty(n/5) + T_\infty(7n/10 + 6). \end{aligned}$$

Using the substitution method we can show that $T_\infty(n) = O(n^\epsilon)$ for some $\epsilon < 1$. In particular, $\epsilon = 0.9$ works. This gives a parallelization of $\Omega(n^0.1)$.

```
P-SELECT(A, i)
    if n == 1
```

```
        return A[1]
    let T[1..floor(n / 5)] be a new array
    parallel for i = 0 to floor(n / 5) - 1
        T[i + 1] = MEDIAN(A[i * floor(n / 5)..i * floor(n / 5) + 4])
    if n / 5 is not an integer
        T[floor(n / 5)] = MEDIAN(A[5 * floor(n / 5)..n])
    x = P-SELECT(T, ceil(n / 5))
    k = P-PARTITION(A, x)
    if k == i
        return x
    else if i < k
        P-SELECT(A[1..k - 1], i)
    else
        P-SELECT(A[k + 1..n], i - k)
```

Problem 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)
    parallel for i = 1 to A.length
        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)
    n = A.length
    grain-size = ?      // to be determined
    r = ceil(n / grain-size)
    for k = 0 to r - 1
        spawn ADD-SUBARRAY(A, B, C, k * grain-size + 1, min((k + 1) *
grain-size, n))
    sync
```

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

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

c. Give a formula for the span of $\text{SUM-ARRAYS}'$ in terms of n and grain-size. Derive the best value for grain-size to maximize parallelism.

a. See the algorithm $\text{SUM-ARRAYS}(A, B, C)$. The parallelism is $O(n)$ since it's work is $n \lg n$ and the span is $\lg n$.

b. If grainsize is 1, this means that each call of ADD-SUBARRAY just sums a single pair of numbers. This means that since the for loop on line 4 will run n times, both the span and work will be $O(n)$. So, the parallelism is just $O(1)$.

```
SUM-ARRAYS(A, B, C)
    n = floor(A.length / 2)
    if n == 0
        C[1] = A[1] + B[1]
    else
        spawn SUM-ARRAYS(A[1..n], B[1..n], C[1..n])
        SUM-ARRAYS(A[n + 1..A.length], B[n + 1..A..length], C[n +
1..A.length])
```

c. Let g be the grainsize. The runtime of the function that spawns all the other functions is $\left\lceil \frac{n}{g} \right\rceil$. The runtime of any particular spawned task is g. So, we want to minimize

$$\frac{n}{g} + g.$$

To do this we pull out our freshman calculus hat and take a derivative, we have

$$0 = 1 - \frac{n}{g^2}.$$

To solve this, we set $g = \sqrt{n}$. This minimizes the quantity and makes the span $O(n/g + g) = O(\sqrt{n})$. Resulting in a parallelism of $O(\sqrt{n})$.

Problem 27-2 Saving temporary space in matrix multiplication

The $\text{P-MATRIX-MULTIPLY-RECURSIVE}$ 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.

(Removed)

Problem 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.

a. For the algorithm LU-DECOMPOSITION(A) on page 821, the inner **for** loops can be parallelized, since they never update values that are read on later runs of those loops. However, the outermost **for** loop cannot be parallelized because across iterations of it the changes to the matrices from previous runs are used to affect the next. This means that the span will be $\Theta(n \lg n)$, work will still be $\Theta(n^3)$ and, so, the parallelization will be $\Theta(\frac{n^3}{n \lg n}) = \Theta(\frac{n^2}{\lg n})$.

b. The **for** loop on lines 7-10 is taking the max of a set of things, while recording the index that that max occurs. This **for** loop can therefor be replaced with a $\lg n$ span parallelized procedure in which we arrange the n elements into the leaves of an almost balanced binary tree, and we let each internal node be the max of its two children. Then, the span will just be the depth of this tree. This procedure can gracefully scale with the number of processors to make the span be linear, though even if it is $\Theta(n \lg n)$ it will be less than the $\Theta(n^2)$ work later. The **for** loop on line 14-15 and the implicit **for** loop on line 15 have no concurrent editing, and so, can be made parallel to have a span of $\lg n$. While the **for** loop on lines 18-19 can be made parallel, the one containing it cannot without creating data races. Therefore, the total span of the naive parallelized algorithm

will be $\Theta(n^2 \lg n)$, with a work of $\Theta(n^3)$. So, the parallelization will be $\Theta(\frac{n}{\lg n})$. Not as parallized as part (a), but still a significant improvement.

c. We can parallelize the computing of the sums on lines 4 and 6, but cannot also parallize the **for** loops containing them without creating an issue of concurrently modifying data that we are reading. This means that the span will be $\Theta(n \lg n)$, work will still be $\Theta(n^2)$, and so the parallelization will be $\Theta(\frac{n}{\lg n})$.

d. The recurrence governing the amount of work of implementing this procedure is given by

$$I(n) \leq 2I(n/2) + 4M(n/2) + O(n^2).$$

However, the two inversions that we need to do are independent, and the span of parallelized matrix multiply is just $O(\lg n)$. Also, the n^2 work of having to take a transpose and subtract and add matrices has a span of only $O(\lg n)$. Therefore, the span satisfies the recurrence

$$I_\infty(n) \leq I_\infty(n/2) + O(\lg n).$$

This recurrence has the solution $I_\infty(n) \in \Theta(\lg^2 n)$ by exercise 4.6-2. Therefore, the span of the inversion algorithm obtained by looking at the procedure detailed on page 830. This makes the parallelization of it equal to $\Theta(M(n)/\lg^2 n)$ where $M(n)$ is the time to compute matrix products.

Problem 27-4 Multithreading reductions and prefix computations

A $\otimes$-***reduction*** of an array $x[1 \ldots n]$, 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 $x[i \ldots j]$ serially.

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

a. Use nested parallelism to implement a multithreaded algorithm $P\text{-}REDUCE$, 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 $\otimes$***-prefix computation***, sometimes called a $\otimes$***-scan***, on an array $x[1 \ldots n]$, where $\otimes$ is once again an associative operator. The $\otimes$-scan produces the array $y[1 \ldots n]$ 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 $\otimes$ operator. The following serial procedure SCAN performs a $\otimes$-prefix computation:

```
SCAN(x)
    n = x.length
    let y[1..n] be a new array
    y[1] = x[1]
    for i = 2 to n
        y[i] = y[i - 1] ⊗ x[i]
    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)
    n = x.length
    let y[1..n] be a new array
    P-SCAN-1-AUX(x, y, 1, n)
    return y
```

```
P-SCAN-1-AUX(x, y, i, j)
    parallel for l = i to j
        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)
    n = x.length
    let y[1..n] be a new array
    P-SCAN-2-AUX(x, y, 1, n)
    return y
```

```
P-SCAN-2-AUX(x, y, i, j)
    if i == j
        y[i] = x[i]
    else k = floor((i + j) / 2)
        spawn P-SCAN-2-AUX(x, y, i, k)
        P-SCAN-2-AUX(x, y, k + 1, j)
        sync
        parallel for l = k + 1 to j
            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)
    n = x.length
    let y[1..n] and t[1..n] be new arrays
    y[1] = x[1]
    if n > 1
        P-SCAN-UP(x, t, 2, n)
        P-SCAN-DOWN(x[1], x, t, y, 2, n)
    return y
```

```
P-SCAN-UP(x, t, i, j)
    if i == j
        return x[i]
    else
        k = floor((i + j) / 2)
        t[k] = spawn P-SCAN-UP(x, t, i, k)
        right = P-SCAN-UP(x, t, k + 1, j)
        sync
        return ____           // fill in the blank
```

```
P-SCAN-DOWN(v, x, t, y, i, j)
    if i == j
        y[i] = v ⊗ x[i]
    else
```

```
        k = floor((i + j) / 2)
        spawn P-SCAN-DOWN(____, x, t, y, i, k)  // fill in the blank
        P-SCAN-DOWN(____, x, t, y, k + 1, j)    // fill in the blank
        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 $\text{P-SCAN-DOWN}(v, x, t, y, i, j)$ satisfies $v = x[1] \otimes x[2] \otimes \cdots \otimes x[i-1]$.)

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

(Removed)

Problem 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 $b \geq 2$. 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 $b \geq 2$. (Hint: For this last argument, show that the exponent of n in the parallelism is strictly less than 1 for any choice of $b \geq 2$.)

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.

(Removed)

Problem 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 $E[T_1]/E[T_P]$, rather than $E[T_1/T_P]$.

c. Argue that the ***parallelism*** of a randomized multithreaded algorithm should be defined as the ratio $E[T_1]/E[T_\infty]$.

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.)

a.

$$E[T_P] \geq E[T_1]/P$$
$$E[T_P] \geq E[T_\infty]$$
$$E[T_P] \leq E[T_1]/P + E[T_\infty].$$

b.

$$E[T_1] \approx E[T_{10,000}] \approx 9.9 \times 10^8, E[T_1]/E[T_P] = 1.$$

$$E[T_1/T_{10,000}] = 10^4 * 0.01 + 0.99 = 100.99.$$

c. Same as the above.

d.

```
RANDOMIZED-QUICKSORT(A, p, r)
    if p < r
        q = RANDOM-PARTITION(A, p, r)
    spawn RANDOMIZED-QUICKSORT(A, p, q - 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
    sync
```

e.

$$E[T_1] = O(n \lg n)$$
$$E[T_\infty] = O(\lg n)$$
$$E[T_1]/E[T_\infty] = O(n).$$