Chapter 27

Michelle Bodnar, Andrew Lohr

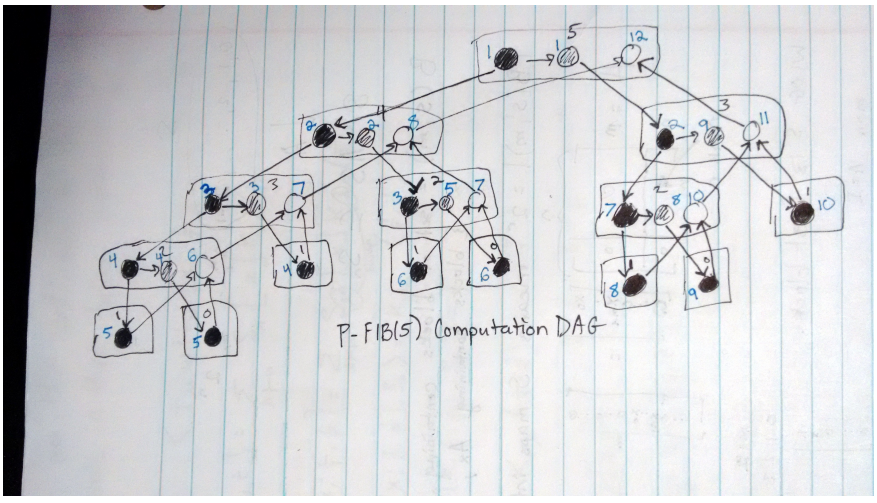
April 12, 2016

Exercise 27.1-1

This modification is not going to affect the asymptotic values of the span work or parallelism. All it will do is add an amount of overhead that wasn't there before. This is because as soon as the $FIB(n-2)$ is spawned the spawning thread just sits there and waits, it does not accomplish any work while it is waiting. It will be done waiting at the same time as it would of been before because the $FIB(n-2)$ call will take less time, so it will still be limited by the amount of time that the $FIN(n-1)$ call takes.

Exercise 27.1-2

The computation dag is given in the image below. The blue numbers by each strand indicate the time step in which it is executed. The work is 29, span is 10, and parallelism is 2.9.



Exercise 27.1-3

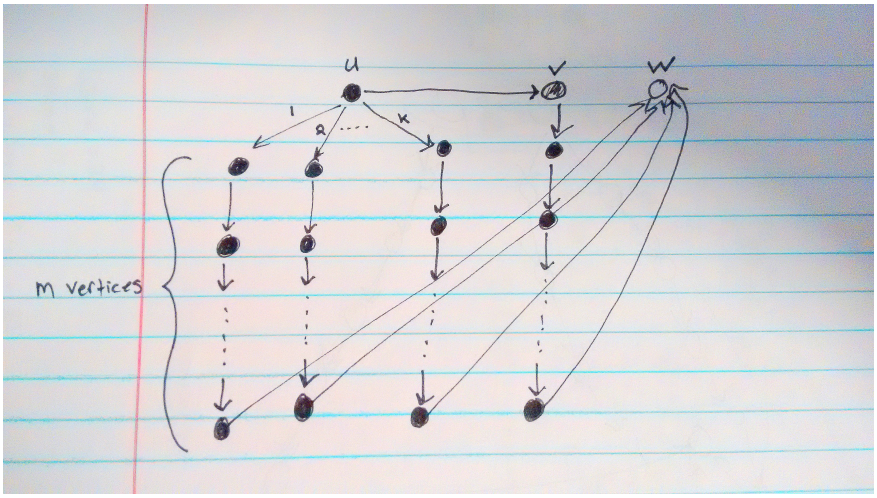
Suppose that there are x incomplete steps in a run of the program. Since each of these steps causes at least one unit of work to be done, we have that there is at most $(T_1 - x)$ units of work done in the complete steps. Then, we suppose by contradiction that the number of complete steps is strictly greater than $\lfloor (T_1 - x)/P \rfloor$. Then, we have that the total amount of work done during the complete steps is $P \cdot (\lfloor (T_1 - x)/P \rfloor + 1) = P\lfloor (T_1 - x)/P \rfloor + P = (T_1 - x) - ((T_1 - x) \mod P) + P > T_1 - x$. 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,

$$T_P \leq \lfloor (T_1 - x)/P \rfloor + x \leq \lfloor (T_1 - T_\infty)/P \rfloor + T_\infty$$

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_∞.

Exercise 27.1-4

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.



Exercise 27.1-5

The information from T_{10} applied to equation (27.5) give us that

$$42 \leq T_1 - T_\infty 10 + T_\infty$$

which tell us that

$$420 \leq T_1 + 9T_\infty$$

Subtracting these two equations, we have that $100 \leq 8T_\infty$.

If we apply the span law to T_64, we have that $10 \geq T_\infty$. Applying the work law to our measurement for T_4 gets us that $320 \geq T_1$. Now, looking at the result of applying (27.5) to the value of T_10, we get that

$$420 \leq T_1 + 9T_\infty \leq 320 + 90 = 410$$

a contradiction. So, one of the three numbers for runtimes must be wrong. However, computers are complicated things, and its difficult to pin down what can affect runtime in practice. It is a bit harsh to judge professor Karan too poorly for something that may of been outside her control (maybe there was just a garbage collection happening during one of the measurements, throwing it off).

Exercise 27.1-6

We'll parallelize the for loop of lines 6-7 in a way which won't incur races. With the algorithm $P - PROD$ given below, it will be easy to rewrite the code. For notation, let a_i denote the i^{th} row of the matrix A.

Algorithm 1 P-PROD(a,x,j,j')

```
1: if j == j' then
2:     return a[j] · x[j]
3: end if
4: mid = ⌊(j+j')/2⌋
5: a' = spawn P-PROD(a,x,j,mid)
6: x' = P-PROD(a,x,mid+1,j')
7: sync
8: return a'+x'
```

Exercise 27.1-7

The work is unchanged from the serial programming case. Since it is flipping $\Theta(n^2)$ many entries, it does $\Theta(n^2)$ work. The span of it is $\Theta(\lg(n))$ this is because each of the parallel for loops can have its children spawned in time $\lg(n)$, so the total time to get all of the constant work tasks spawned is $2\lg(n) \in \Theta(\lg)$.

Algorithm 2 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 do
4:     y_i = 0
5: end
6: parallel for i = 1 to n do
7:     y_i = P-PROD(a_i, x, 1, n)
8: end
9: return y
```

Since the work of each task is $o(\lg(n))$, that doesn't affect the T_∞ runtime. The parallelism is equal to the work over the span, so it is $\Theta(n^2/\lg(n))$.

Exercise 27.1-8

The work is $\Theta(1 + \sum_{j=2}^{n} j - 1) = \Theta(n^2)$. The span is $\Theta(n)$ because in the worst case when $j = n$, the for-loop of line 3 will need to execute n times. The parallelism is $\Theta(n^2)/\Theta(n) = \Theta(n)$.

Exercise 27.1-9

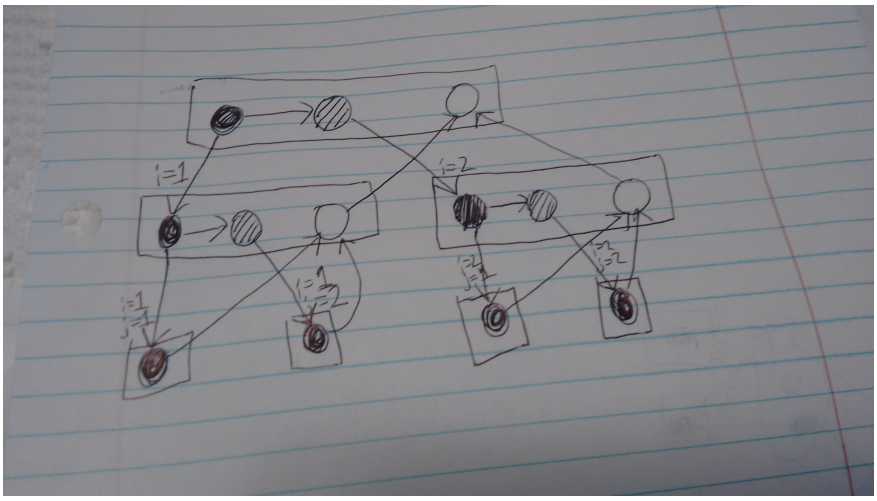
We solve for P in the following equation obtained by setting $T_P = T'_P$.

$$\begin{aligned}
\frac{T_1}{P} + T_\infty &= \frac{T'_1}{P} + T'_\infty \\
\frac{2048}{P} + 1 &= \frac{1024}{P} + 8 \\
\frac{1024}{P} &= 7 \\
\frac{1024}{7} &= P
\end{aligned}$$

So we get that there should be approximately 146 processors for them to have the same runtime.

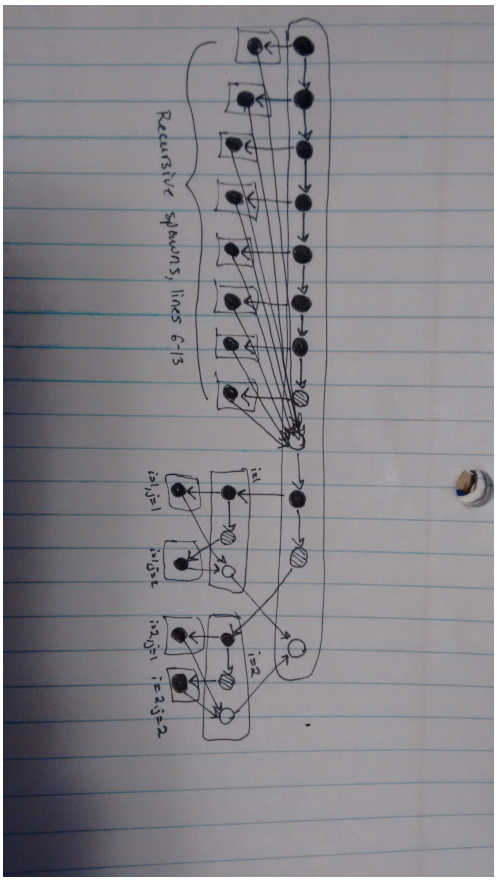
Exercise 27.2-1

See the computation dag in the image below. Assuming that each strand takes unnit time, the work is 13, the span is 6, and the parallelism is $\frac{13}{6}$



Exercise 27.2-2

See the computation dag in the image below. Assuming each strand takes unit time, the work is 30, the span is 16, and the parallelism is $\frac{15}{8}$.



Exercise 27.2-3

We perform a modification of the P-SQUARE-MATRIX-MULTIPLY algorithm. Basically, as hinted in the text, we will parallelize the innermost for loop in such a way that there aren't any data races formed. To do this, we will just define a parallelized dot product procedure. This means that lines 5-7 can be replaced by a single call to this procedure. P-DOT-PRODUCT computes the dot dot product of the two lists between the two bounds on indices.

Using this, we can use this to modify P-SQUARE-MATRIX-MULTIPLY

Since the runtime of the inner loop is $O(\lg(n))$, which is the depth of the recursion. Since the paralel for loops also take $O(\lg(n))$ time. So, since the runtimes are additive here, the total span of this procedure is $\Theta(\lg(n))$. The total work is still just $O(n^3)$ Since all the spawning and recursing call be replaced with the normal serial version once there aren't enough free processors to handle all of the spawned calls to P-DOT-PRODUCT.

Exercise 27.2-4

Algorithm 3 P-DOT-PROD(v,w,low,high)

```
if low == high then
    return v[low] = v[low]
end if
mid = ⌊(low+high)/2⌋
x = spawn P-DOT-PROD(v,w,low,mid)
y = P-DOT-PROD(v,w,mid+1,high)
sync
return x+y
```

Algorithm 4 MODIFIED-P-SQUARE-MATRIX-MULTIPLY

```
n = A.rows
let C be a new n × n matrix
parallel for i=1 to n do
    parallel for j=1 to n do
        c_{i,j} = P-DOT-PROD(A_{i,.}, B_{.,j}, 1, n)
    end
end
return C
```

Assume that the input is two matrices A and B to be multiplied. For this algorithm we use the function P-PROD defined in exercise 21.7-6. For notation, we let A_i denote the i^{th} row of A and A'_i denote the i^{th} column of A. Here, C is assumed to be a p by r matrix. The work of the algorithm is $\Theta(prq)$, since this is the runtime of the serialization. The span is $\Theta(\log(p)+\log(r)+\log(q)) = \Theta(\log(pqr))$. Thus, the parallelism is $\Theta(pqr/\log(pqr)$, which remains highly parallel even if any of p, q, or r are 1.

Algorithm 5 MATRIX-MULTIPLY(A,B,C,p,q,r)

```
1: parallel for i = 1 to p do
2:     parallel for j = 1 to r do
3:         C_ij = P-PROD(A_i, B'_j, 1, q)
4:     end
5: end
6: return C
```

Exercise 27.2-5

Split up the region into four sections. Then, this amounts to finding the transpose the upper left and lower right of the two submatrices. In addition to that, you also need to swap the elements in the upper right with their transpose position in the lower left. This dealing with the upper right swapping only takes

time $O(\lg(n^2)) = O(\lg(n))$. In addition, there are two subproblems, each of half the size. This gets us the recursion:

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

By the master theorem, we get that the total span of this procedure is $T_\infty \in O(\lg(n)$. The total work is still the usual $O(n^2)$.

Exercise 27.2-6

Since D^k cannot be computed without D^{k-1} we cannot parallelize the for loop of line 3 of Floyd-Warshall. However, the other two loops can be parallelized. The work is $\Theta(n^2)$, as in the serial case. The span is $\Theta(n \lg n)$. Thus, the parallelism is $\Theta(n/\lg n)$. The algorithm is as follows:

Algorithm 6 P-FLOYD-WARSHALL(W)

```
1: n = W.rows
2: D^(0) = W
3: for k = 1 to n do
4:     let D^(k) = (d_ij^(k)) be a new n × n matrix
5:     parallel for i = 1 to n do
6:         parallel for j = 1 to n do
7:             d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8:         end
9:     end
10: end for
11: return D^(n)
```

Exercise 27.3-1

To coarsen the base case of P-MERGE, just 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.

Exercise 27.3-2

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 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. The first 8 lines of code are identical to those in P-MERGE given on page 800, so

we omit them here.

Algorithm 7 P-MEDIAN-MERGE($T, p_1, r_1, p_2, r_2, A, p_3$)

```
1: Run lines 1 through 8 of P-MERGE
2: q = MEDIAN(T, p_1, r_1, p_2, r_2)
3: if q.arr == 1 then
4:     q_2 = BINARY-SEARCH(T[q.pos]), T, p_2, r_2)
5:     q_3 = p_3 + q.pos − p_1 + q_2 − p_2
6:     A[q_3] = T[q.pos]
7:     spawn P-MEDIAN-MERGE(T, p_1, q.pos − 1, p_2, q_2 − 1, A, p_3)
8:     P-MEDIAN-MERGE(T, q.pos + 1, r_1, q_2 + 1, r_2, A, p_3)
9:     sync
10: else
11:    q_2 = BINARY-SEARCH(T[q.pos], T, p_1, r_1)
12:    q_3 = p_3 + q.pos − p_2 + q_2 − p_1
13:    A[q_3] = T[q.pos]
14:    spawn P-MEDIAN-MERGE(T, p_1, q_2 − 1, p_2, q.pos − 1, A, p_3)
15:    P-MEDIAN-MERGE(T, q_2 + 1, r_1, q.pos + 1, r_2, A, p_3)
16:    sync
17: end if
```

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 $T_\infty(n) = O(\lg n) + O(\lg n/2) + T_\infty(n/2) = O(\lg n) + T_\infty(n/2) = \Theta(\lg^2 n)$, by exercise 4.6-2.

Exercise 27.3-3

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.

Algorithm 8 PPartition(L)

```
c = min{c, n}
pivot = L[n]
let Count be an array of length c
let r_1, ... r_{c+1} be roughly evenly spaced indices to L with r_1 = 1 and r_{c+1} = n
for i=1 ... c do
    Count[i] = spawn countnum(L[r_i, r_{i+1} − 1],pivot)
end for
sync
let T be a nearly complete binary tree whose leaves are the elements of Count
whose vertices have the attributes sum and lc
for all the leaves, let their sum value be the corresponding entry in Count
ComputeSums(T.root)
T.root.lc = 0
ComputeCousins(T.root)
Let Target be an array of length n that the elements will be copied into
for i=1 ... c do
    let cousin be the lc value of the node in T that corresponds to i
    spawn CopyElts(L,Target, cousin,r_i, r_{i+1} − 1)
end for
Target[n] = Target[T.root.sum]
Target[T.root.sum] = L[n]
return Target
```

Algorithm 9 CountNum(L,x)

```
ret = 0
for i=1 ... L.length do
    if L[i] < x then
        ret++
    end if
end for
return ret
```

Exercise 27.3-4

See the algorithm P-RECURSIVE-FFT. it parallelized over the two recursive calls, having a parallel for works because each of the iterations of the for loop

Algorithm 10 ComputeSums(v)

```
if v is an internal node then
    x = spawn ComputeSums(v.left)
    y = ComputeSums(v.right)
    sync
    v.sum = x+y
end if
return v.sum
```

Algorithm 11 ComputeCousins(v)

```
if v ≠ NIL then
    v.lc = v.p.lv
    if v = v.p.right then
        v.lc += c.p.left.sum
    end if
    spawn ComputeCousins(v.left)
    ComputeCousins(v.right)
    sync
end if
```

Algorithm 12 CopyElts(L1, L2, lc,lb,ub)

```
counter1 = lc+1
counter2 = lb
for i=lb ... ub do
    if L1[i] < x then
        L2[counter1++] = L1[i]
    else
        L2[counter2++] = L1[i]
    end if
end for
```

touch independent sets of variables. The span of the procedure is only $\Theta(\lg(n))$ giving it a parallelization of $\Theta(n)$

Algorithm 13 P-RECURSIVE-FFT(a)

```
n = a.length
if n == 1 then
    return a
end if
ω_n = e^{2πi/n}
ω = 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, ..., n/2 − 1 do
    y_k = y_k^[0] + ω y_k^[1]
    y_{k+(n/2)} = y_k^[0] − ω y_k^[1]
    ω = ω ω_n
end
return y
```

Exercise 27.3-5

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 PPartition. 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 $O(\lg(n))$.

Exercise 27.3-6

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-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 $T_\infty(n) = \Theta(\lg n/5) + T_\infty(n/5) + \Theta(\lg n) + T_\infty(7n/10+6) \leq \Theta(\lg n) + T_\infty(n/5) + T_\infty(7n/10+6)$. Using the substitution method we can show that $T_\infty(n) = O(n^\varepsilon)$ for some $\varepsilon < 1$. In particular, $\varepsilon = .9$ works. This gives a parallelization of $\Omega(n^{.1})$.

Algorithm 14 P-SELECT(A,i)

```
1: if n == 1 then
2:     return A[1]
3: end if
4: Initialize a new array T of length ⌊n/5⌋
5: parallel for i = 0 to ⌊n/5⌋ − 1 do
6:     T[i + 1] = MEDIAN(A[i⌊n/5⌋..i⌊n/5⌋ + 4])
7: end
8: if n/5 is not an integer then
9:     T[⌊n/5⌋] = MEDIAN(A[5⌊n/5⌋..n)
10: end if
11: x = P-SELECT(T, ⌈n/5⌉)
12: k = P-PARTITION(A, x)
13: if k == i then
14:     return x
15: else if i < k then
16:     P-SELECT(A[1..k − 1], i)
17: else
18:     P-SELECT(A[k + 1..n], i − k)
19: end if
```

Problem 27-1

a. See the algorithm 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 $O(1)$.

Algorithm 15 Sum-Arrays(A,B,C)

```
n = ⌊A.length/2⌋
if n=0 then
    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])
    sync
end if
```

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 get

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

So, 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

a. Our algorithm P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C,A,B) multiplies A and B, and adds their product to the matrix C. It is assumed that C contains all zeros when the function is first called.

b. The work is the same as the serialization, which is $\Theta(n^3)$. It can also be found by solving the recurrence $T_1(n) = \Theta(n^2) + 8T(n/2)$ where $T_1(1) = 1$. By the mater theorem, $T_1(n) = \Theta(n^3)$. The span is $T_\infty(n) = \Theta(1) + T_\infty(n/2) + T_\infty(n/2)$ with $T_\infty(1) = \Theta(1)$. By the master theorem, $T_\infty(n) = \Theta(n)$.

c. The parallelism is $\Theta(n^2)$. Ignoring the constants in the Θ-notation, the parallelism of the algorithm on 1000×1000 matrices is 1,000,000. Using P-MATRIX-MULTIPLY-RECURSIVE, the parallelism is 10,000,000, which is only about 10 times larger.

Problem 27-3

Algorithm 16 P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C,A,B)

1: $n = A.rows$
2: **if** $n = 1$ **then**
3: $\quad c_1 1 = c_1 1 + a_1 1 b_1 1$
4: **else**
5: $\quad$ Partition A, B, and C into $n/2 \times n/2$ submatrices
6: $\quad$ **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{11}, A_{11}, B_{11})
7: $\quad$ **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{12}, A_{11}, B_{12})
8: $\quad$ **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{21}, A_{21}, B_{11})
9: $\quad$ **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{22}, A_{21}, B_{12})
10: $\quad$ **sync**
11: $\quad$ **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{11}, A_{12}, B_{21})
12: $\quad$ **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{12}, A_{12}, B_{22})
13: $\quad$ **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{21}, A_{22}, B_{21})
14: $\quad$ **spawn** P-MATRIX-MULTIPLY-RECURSIVE-SPACE(C_{22}, A_{22}, B_{22})
15: $\quad$ **sync**
16: **end if**

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

a. The algorithm below has $\Theta(n)$ work because its serialization satisfies the recurrence $T_1(n) = 2T(n/2) + \Theta(1)$ and $T(1) = \Theta(1)$. It has span $T_\infty(n) = \Theta(\lg n)$ because it satisfies the recurrence $T_\infty(n) = T_\infty(n/2) + \Theta(1)$ and $T_\infty(1) = \Theta(1)$.

Algorithm 17 P-REDUCE(x,i,j)

```
1: if i == j then
2:     return x[i]
3: else
4:     mid = ⌊(i + j)/2⌋
5:     x =spawn P-REDUCE(x, i, mid)
6:     y = P-REDUCE(x, mid + 1, j)
7:     sync
8:     return x ⊗ y
9: end if
```

b. The work of P-SCAN-1 is $T_1(n) = \Theta(n^2)$. The span is $T_\infty(n) = \Theta(n)$. The parallelism is $\Theta(n)$.

c. We'll prove correctness by induction on the number of recursive calls made to P-SCAN-2-AUX. If a single call is made then $n = 1$, and the algorithm sets $y[1] = x[1]$ which is correct. Now suppose we have an array which requires $n+1$ recursive calls. The elements in the first half of the array are accurately

computed since they require one fewer recursive calls. For the second half of the array,

$$y[i] = x[1]\otimes x[2]\otimes\ldots\otimes x[i] = (x[1]\otimes\ldots\otimes x[k])\otimes(x[k{+}1]\otimes\ldots\otimes x[i]) = y[k]\otimes(x[k{+}1]\otimes\ldots\otimes x[i]).$$

Since we have correctly computed the parenthesized term with P-SCAN-2-AUX, line 8 ensures that we have correctly computed $y[i]$.

The work is $T_1(n) = \Theta(n \lg n)$ by the master theorem. The span is $T_\infty(n) = \Theta(\lg^2 n)$ by exercise 4.6-2. The parallelism is $\Theta(n/\lg n)$.

d. Line 8 of P-SCAN-UP should be filled in by $right \otimes t[k]$. Lines 5 and 6 of P-SCAN-DOWN should be filled in by v and $v \otimes t[k]$ respectively. Now we prove correctness. First I claim that if line 5 is accessed after l recursive calls, then

$$t[k] = x[k] \otimes x[k-1] \otimes \ldots \otimes x[k - \lfloor n/2^l \rfloor + 1]$$

and

$$right = x[k+1] \otimes x[k+2] \otimes \ldots x[k + \lfloor n/2^l \rfloor].$$

If $n = 2$ we make a single call, but no recursive calls, so we start our base case at $n = 3$. In this case, we set $t[2] = x[2]$, and $2 - \lfloor 3/2 \rfloor + 1 = 2$. We also have $right = x[3] = x[2+1]$, so the claim holds. In general, on the l^{th} recursive call we set $t[k] = \text{P-SCAN-UP}(x, t, i, k)$, which is $t[\lfloor (i+k)/2 \rfloor] \otimes right$. By our induction hypothesis, $t[k] = x[(i+k)/2] \otimes x[(i+k)/2 - 1] \otimes \ldots \otimes x[(i+k)/2 - \lfloor n/2^{l+1} \rfloor + 1] \otimes x[(i+k)/2 + 1 \otimes \ldots \otimes x[(i+k)/2 + \lfloor n/2^{l+1} \rfloor]$. This is equivalent to our claim since $(k-i)/2 = \lfloor n/2^{l+1} \rfloor$. A similar proof shows the result for $right$.

With this in hand, we can verify that the value v passed to P-SCAN-DOWN(v, x, t, y, i, j) satisfies $v = x[1] \otimes x[2] \otimes \ldots \otimes x[i-1]$. For the base case, if a single recursive call is made then $i = j = 2$, and we have $v = x[1]$. In general, for the call on line 5 there is nothing to prove because i doesn't change. For the call on line 6, we replace v by $v \otimes t[k]$. By our induction hypothesis, $v = x[1] \otimes \ldots \otimes x[i-1]$. By the previous paragraph, if we are on the l^{th} recursive call, $t[k] = x[i] \otimes \ldots \otimes x[k - \lfloor n/2^l \rfloor + 1] = x[i]$ since on the l^{th} recursive call, k and i must differ by $\lfloor n/2^l \rfloor$. Thus, the claim holds. Since we set $y[i] = v \otimes x[i]$, the algorithm yields the correct result.

e. The work of P-SCAN-UP satisfies $T_1(n) = 2T(n/2) + \Theta(1) = \Theta(n)$. The work of P-SCAN-DOWN is the same. Thus, the work of P-SCAN-3 satisfies $T_1(n) = \Theta(n)$. The span of P-SCAN-UP is $T_\infty(n) = T_\infty(n/2) + O(1) = \Theta(\lg n)$, and similarly for P-SCAN-DOWN. Thus, the span of P-SCAN-3 is $T_\infty(n) = \Theta(\lg n)$. The parallelism is $\Theta(n/\lg n)$.

Problem 27-5

a. Note that in this algorithm, the first call will be SIMPLE-STENCIL(A,A), and when there are ranges indexed into a matrix, what is gotten back is a view of the original matrix, not a copy. That is, changed made to the view will show up in the original. We can set up a recurrence for the work, which

Algorithm 18 $SIMPLE - STENCIL(A, A_2)$

let $n_1 \times n_2$ be the size of A_2.
let $m_i = \left\lfloor \frac{n_i}{2} \right\rfloor$ for $i = 1, 2$.
if $m_1 == 0$ **then**
  if $m_2 == 0$ **then**
    compute the value for the only position in A_2 based on the current values in A.
  else
    $SIMPLE - STENCIL(A, A_2[1, 1 \ldots m_2])$
    $SIMPLE - STENCIL(A, A_2[1, m_2 + 1 \ldots n_3])$
  end if
else
  if $m_2 == 0$ **then**
    $SIMPLE - STENCIL(A, A_2[1 \ldots m_1, 1])$
    $SIMPLE - STENCIL(A, A_2[m_1 + 1 \ldots n_1, 1])$
  else
    $SIMPLE - STENCIL(A, A_2[1 \ldots m_1, 1 \ldots m_2])$
    spawn $SIMPLE - STENCIL(A, A_2[m_1 + 1 \ldots n_1, 1 \ldots m_2])$
    $SIMPLE - STENCIL(A, A_2[1 \ldots m_1, m_2 + 1 \ldots n_2])$
    sync
    $SIMPLE - STENCIL(A, A_2[m_1 + 1 \ldots n_1, m_2 + 1 \ldots n_2])$
  end if
end if

is just

$$W(n) = 4W(n/2) + \Theta(1)$$

which we can see by the master theorem has a solution which is $\Theta(n^2)$. For the span, the two middle subproblems are running at the same time, so,

$$S(n) = 3S(n/2) + \Theta(1)$$

Which has a solution that is $\Theta(n^{\lg(3)})$, also by the master theorem.

b. Just use the implementation for the third part with $b = 3$The work has the same solution of n^2 because it has the recurrence

$$W(n) = 9W(n/3) + \Theta(1)$$

The span has recurrence

$$S(n) = 5S(n/3) + \Theta(1)$$

Which has the solution $\Theta(n^{\log_3(5)})$

Algorithm 19 GEN-SIMPLE-STENCIL(A,A_2,b)

c. let $n \times m$ be the size of A_2.
if $(n \neq 0)\&\&(m \neq 0)$ **then**
if $(n == 1)\&\&(m == 1)$ **then**
compute the value at the only position in A_2
else
let $n_i = \lfloor \frac{in}{b} \rfloor$ for $i = 1, \ldots, b-1$
let $m_i = \lfloor \frac{im}{b} \rfloor$ for $i = 1, \ldots, b-1$
let $n_0 = m_0 = 1$
for k=2, ... b+1 **do**
for i=1, ... k-2 **do**
spawn $GEN{-}SIMPLE{-}STENCIL(A, A_2[n_{i-1} \ldots n_i, m_{k-i-1} \ldots m_{k-i}], b)$
end for
$GEN{-}SIMPLE{-}STENCIL(A, A_2[n_{i-1} \ldots n_i, m_{k-i-1} \ldots m_{k-i}], b)$
sync
end for
for k=b+2, ..., 2b **do**
for i=1,...,2b-k **do**
spawn $GEN{-}SIMPLE{-}STENCIL(A, A_2[n_{b-k+i-1} \ldots n_{b-k+i}, m_{b-i-1} \ldots m_{b-i}], b)$
end for
$GEN{-}SIMPLE{-}STENCIL(A, A_2[n_{3b-2k} \ldots n_{3b-2k+1} i, m_{2k-2b} \ldots m_{2k-2b+1}], b)$
sync
end for
end if
end if

The recurrences we get are

$$W(n) = b^2 W(n/b) + \Theta(1)$$

$$S(n) = (2b-1)W(n/b) + \Theta(1)$$

So, the work is $\Theta(n^2)$, and the span is $\Theta(n^{\lg_b(2b-1)})$. This means that the parallelization is $\Theta(n^{2-\lg_b(2b-1)})$. So, to show the desired claim, we only need to show that $2 - \log_b(2b-1) < 1$

$$\begin{aligned} 2 - \log_b(2b-1) &< 1 \\ \log_b(2b) - \log_b(2b-1) &< 1 \\ \log_b\left(\frac{2b}{2b-1}\right) &< 1 \\ \frac{2b}{2b-1} &< b \\ 2b &< 2b^2 - b \\ 0 &< 2b^2 - 3b \\ 0 &< (2b-3)b \end{aligned}$$

This is clearly true because b is an integer greater than 2 and this right hand side only has zeroes at 0 and $\frac{3}{2}$ and is positive for larger b.

Algorithm 20 BETTER-STENCIL(A)

d.
```
for k=2,..., n+1 do
    for i=1, ... k-2 do
        spawn compute and update the entry at A[i,k-i]
    end for
    compute and update the entry at A[k-1,1]
    sync
end for
for k=n+2,... 2n do
    for i=1, ... 2n-k do
        spawn compute and update the entries along the diagonal which have
indices summing to k
    end for
    sync
end for
```

This procedure has span only equal to the length of the longest diagonal with is $O(n)$ with a factor of $\lg(n)$ thrown in. So, the parallelism is $O(n^2/(n\lg(n))) = O(n/\lg(n))$.

Problem 27-6

a. The work law becomes $E[T_P] \geq E[T_1]/P$. The span law becomes $E[T_P] \geq E[T_\infty]$. The greedy scheduler bound becomes $E[T_P] \leq E[T_1]/P + E[T_\infty]$.

b. We'll compute each.

$$E[T_1]/E[T_P] = \frac{100 + 10^9 \cdot .99}{.01 + 10^9 \cdot .99} \approx 1$$
$$E[T_1/T_p] = 100 + .99 = 100.99.$$

Since the algorithm almost always runs in the same amount of time, regardless of the increase in number of processors, and the speedup tells us how many times faster something runs on P processors than on 1, the expected speedup should be approximately 1. Thus, $E[T_1]/E[T_P]$ is the better definition.

c. As $P \to \infty$ the speedup should approach the parallelism, so it makes sense to use a definition which agrees with part b in the limit.

d. Assume that PARTITION is implemented as described in exercise 27.3-3, with work $\Theta(n)$ and span $\Theta(\lg n)$. However, we will not modify anything else about RANDOMIZED-PARTITION.

Algorithm 21 P-RANDOMIZED-QUICKSORT(A,p,r)

```
1: if p < r then
2:     q = RANDOMIZED-PARTITION(A, p, r)
3:     spawn P-RANDOMIZED-QUICKSORT(a, P, Q − 1)
4:     RANDOMIZED-QUICKSORT(A, q + 1, r)
5: end if
```

e. The work is just the runtime of the serialization, which we know to have expected time $O(n \lg n)$. For the span, our analysis will be similar to that of RANDOMIZED-SELECT from page 216. Let X_k be the indicator random variable which is equal to 1 if $A[p..q]$ has exactly k elements and 0 otherwise. Then we have

$$T_\infty(n) \leq \sum_{k=1}^{n} X_k \cdot T_\infty(\max(k-1, n-k)) + \Theta(\lg n).$$

By linearity of expectation this implies that

$$E[T_\infty(n)] \leq \sum_{k=1}^{n} \frac{1}{n} E[T_\infty(\max(k-1, n-k))] + \Theta(\lg n).$$

Since each even term appears twice we can write this as

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

We'll show that $E[T_\infty(n)] = O(n^{1-\varepsilon})$ for ε such that $\frac{2-2^{\varepsilon-1}}{2-\varepsilon} < 1$ by the substitution method. Suppose that $E[T_\infty(n)] \le c_1 n^{1-\varepsilon}$. Then we have

$$\begin{aligned}
E[T_\infty(n)] &\le \frac{2c_1}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} k^{1-\varepsilon} + \Theta(\lg n) \\
&\le \frac{2c_1}{n} \int_{k=\lfloor n/2 \rfloor}^{n} x^{1-\varepsilon} dx + \Theta(\lg n) \\
&= \frac{2c_1}{n} \left. \frac{x^{2-\varepsilon}}{2-\varepsilon} \right|_{k=\lfloor n/2 \rfloor}^{n} + \Theta(\lg n) \\
&= c_1 n^{1-\varepsilon} \left(\frac{2-2^{\varepsilon-1}}{2-\varepsilon} \right) + \Theta(\lg n).
\end{aligned}$$

Since the dominating term is strictly less than $c_1 n^{1-\varepsilon}$, we can overcome the $\Theta(\lg n)$ term. Thus, $E[T_\infty(n)] = O(n^{1-\varepsilon})$, so we achieve sublinear expected time. The expected parallelization $\Omega(n \lg n / n^{1-\varepsilon}) = \Omega(n^\varepsilon \lg n)$.