CME 323: Distributed Algorithms and Optimization

Instructor: Reza Zadeh (rezab@stanford.edu)
TA: Alex Yang (yangzj@stanford.edu)

HW#1 – Solutions

1. The Karatsuba algorithm multiplies two integers x and y. Assuming each has n bits where n is a power of 2, it does this by splitting the bits of each integer into two halves, each of size $n/2$. For any integer x we will refer to the low order bits as x_l and the high order as x_h. The algorithm computes the result as follows:

```
function KM(x, y, n)
    if n = 1 then
        return x × y
    else
        a ← km(x_l, y_l, n/2)
        b ← km(x_h, y_h, n/2)
        c ← km(x_l + x_h, y_l + y_h, n/2)
        d ← c − a − b
        return (b · 2^n + d · 2^(n/2) + a)
    end if
end function
```

Note that multiplying by 2^k can be done just by shifting the bits over k positions.

(a.) Assuming addition, subtraction, and shifting take $O(n)$ work and $O(n)$ depth what is the work and depth of **km**?

Solution Each invocation of `km` with n-bit integers results in `km` being called three times (in parallel), each with input size $n/2$. We are told that $n = 2^k$ for some $k \in \mathbb{Z}^+$. Our recursion bottoms out when $n = 1$, in which case we perform a single-digit multiply in constant time. Notice that the number of single-digit multiplies, i.e. the number of times we bottom-out in our recursion and hit our base-case, is given by 3^k, where each multiply takes $O(1)$ *work*.

Let $W(n)$ define the total work of our algorithm. Since additions, subtractions, and bit-shifts are assumed to require $O(n)$ work, we may express

$$W(n) = 3W\left(\frac{n}{2}\right) + \alpha n$$

for some constant $\alpha \in \mathbb{R}^+$. Using the Master Theorem, [1] we see that $W(n) = \Theta(n^{\log_2 3})$.

With respect to Depth, let $D(n)$ denote the depth of our algorithm. We know that addition, subtraction, and shifting also require $O(n)$ depth. Notice that the recursive calls to `km` are made in parallel and therefore share no dependencies. Hence

$$D(n) = D(n/2) + \alpha n,$$

[1] Here, $a = 3, b = 2$, hence $\log_b a = \log_2 3$. Then, $f(n) = \alpha n = O(n)$. Thus $c = 1 < \log_2 \approx 1.6$. Case 1.

for some $\alpha \in \mathbb{R}$. Using the Master Theorem,[2] we see that $D(n) = \Theta(n)$.

(b.) Assuming addition, subtraction, and shifting take $O(n)$ work and $O(\log n)$ depth what is the work and depth of **km**?

Solution Work remains the same. But now our Depth is given by

$$D(n) = D(n/2) + O(\log n).$$

2. Suppose a square matrix is divided into blocks:

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

where all the blocks are the same size. The *Schur complement* of block D of M is $S = A - BD^{-1}C$. The inverse of the matrix M can then be expressed as:

$$M^{-1} = \begin{bmatrix} S^{-1} & S^{-1}BD^{-1} \\ -D^{-1}CS^{-1} & D^{-1} + D^{-1}CS^{-1}BD^{-1} \end{bmatrix}$$

This basically defines a recursive algorithm for inverting a matrix which makes two recursive calls (to calculate D^{-1} and S^{-1}), several calls to matrix multiply, and one each to elementwise add and subtrace two matrices. Assuming that matrix multiply has work $O(n^3)$ and depth $O(\log n)$ what is the work and depth of this inversion algorithm?

Solution Each iteration of the recursive matrix inversion algorithm involves two recursive calls, each on a square matrix whose side-length is half as large. The following steps in the algorithm require matrix multiplies which dominate the work and depth of elementwise operators.[3]

Therefore, we set up our recurrence for *work*,

$$W(n) = 2W\left(\frac{n}{2}\right) + \alpha n^3,$$

for some $\alpha \in \mathbb{R}^+$. Using the Master Theorem[4] , we conclude that $W(n) = O(n^3)$.

With regard to the depth of the algorithm, notice that D^{-1} required to compute S^{-1}, i.e. these operations may *not* be done in parallel. Hence

$$D(n) = 2D\left(\frac{n}{2}\right) + O(\log n).$$

We fall into case 1 of the Master Theorem, since $a = b = 2$ and $f(n) = O(\log n) = O(\sqrt{n})$,[5] Thus, $D(n) = O(n)$.

[2]Here, $a = 1$, $b = 2$, hence $\log_b a = \log_2 1 = 0$. So $f(n) = \alpha n = \Omega(n)$. This places us into Case 3. We check that $f(n/2) \leq kf(n)$ for some constant $k < 1$ – i.e. choose $1/2 < k < 1$, then $\alpha n/2 \leq k\alpha n$ satisfied.

[3]Specifically, the element-wise add or subtract requires n^2 independent additions. The work is clearly $O(n^2)$. Notice that depth $O(1)$, since with $p = n^2$ processors we can perform exactly one of the n^2 operations on each processor in constant time.

[4]$a = 2, b = 2$, so $\log_b a = 1$; $f(n) = O(n^3)$ implies $c > \log_b a$, i.e. Case 3. We check $2f(n/2) \leq kf(n)$ for some $k \in (0, 1)$, i.e. let $k = 1/2$ then $2\alpha n^3/8 = \alpha n^3/4 \leq \alpha n^3/2$.

[5]To see why, note that $\log x < x$ for all $x > 0$. Then, $\log x = 2\log(\sqrt{x}) < 2\sqrt{x}$.

3. Describe a divide-and-conquer algorithm for merging two sorted arrays of lengths n into a sorted array of length $2n$. It needs to run in $O(n)$ work and $O(\log^2 n)$ depth. You can write the pseudocode for your algorithm so that it looks like your favorite sequential language (C, Java, Matlab, ...), but with an indication of which loops or function calls happen in parallel. For example, use `parallel for` for a parallel for loop, and something like:

```
parallel {
    foo(x,y)
    bar(x,y)
}
```

to indicate that `foo` and `bar` are called in parallel. You should prove correctness at the level expected in an algorithms class (e.g. CME305 or CS161).

Solution Notice that we may find the median of a sorted array in $O(1)$ time. Let $n = |A|$. If n odd, then the median element is uniquely determined by index $(n-1)/2$ (where we *index* starting from 0). If n even, there are two medians with *indices* at $n/2$ and $n/2 - 1$.

Below, $\preceq$ denotes the *element-wise inequality* operator. In our pseudo-code, we index an array just like arrays are sliced in python, e.g. `a[1:j]` means that we start at the *second* element and take all elements up to but *not including* index j.

function ParallelMerge(A, B)
Input: Sorted arrays A, B, where $|A| = n$ and $|B| = m$
Output: Merged and sorted array C of length $n + m$
  if $n \leq 1$ and $m \leq 1$ **then**
    return A and B in sorted order
  end if
  if $m \mod 2 == 1$ **then**
    $j \leftarrow (m-1)/2$
  else
    $j \leftarrow m/2$
  end if
  $i \leftarrow$ max index of corresponding element in A such that $A[0:i] \preceq B[j:m]$
  In Parallel, Do:
    $a \leftarrow \text{Merge}(A[0:i], B[0:j])$
    $b \leftarrow \text{Merge}(A[i:n], B[j:m])$
  End Parallel
  return concatenate(a, b)
end function

Notice that when j defined as above in our algorithm,

$$\max\{\texttt{a[0:i]}, \texttt{b[0:j]}\} \leq \min\{\texttt{a[i:n]}, \texttt{b[j:m]}\}$$

and that `a` and `b` are sorted. This is why after our recursive calls return `a` and `b`, we

claim that we may simply concatenate the result and maintain our sort-guarantee. We claim that this algorithm has work $O(n)$ and depth $O(\log^2 n)$.

In the recursion tree at depth d, there are 2^d calls made to the `merge` procedure, denoted by `Merge(`$A_{d,i}$`,` $B_{d,i}$`)` for $i = 1, 2, \ldots, 2^d$. Each of the $B_{d,i}$ are of size *exactly* $m/2^d$, and note that the size of $A_{d,i}$'s are such that $\sum_{i=1}^{2^d} |A_{d,i}| = n$.

Searching for the element in A such that $A[0:i] \preceq B[0:n/2]$ using a binary search on our sorted array A requires $O(\log n)$ work on a single processor. Therefore, we see that total work for taking two sorted arrays of size n is given by

$$W(n,n) = \sum_{d=1}^{\log_2 n} \left[\sum_{i=1}^{2^d} \log |A_{d,i}| + c \right] \qquad \text{for some constant } c$$

But note that since $\log x$ concave, so by Jensen's Inequality, This leads to the Arithmetic-Geometric Mean Inequality, with the consequence that for a set of n inputs

$$\frac{\sum_{i=1}^{n} \log x_i}{n} \leq \log\left(\frac{\sum_{i=1}^{n} x_i}{n} \right).$$

From our observations above regarding the size of $A_{d,i}$'s, and since $2^d \geq 1$ for all $d \in \mathbb{Z}^+$, we see that

$$\frac{\sum_{i=1}^{2^d} \log(|A_{d,i}|)}{2^d} \leq \log\left(\frac{n}{2^d}\right) \implies \sum_{i=1}^{2^d} \log(|A_{d,i}|) \leq 2^d \log\left(\frac{n}{2^d}\right)$$

Hence we see that

$$\begin{aligned} W(n,n) &\leq \sum_{d=1}^{\log n} \left(2^d \log\left(\frac{n}{2^d}\right) + c \right) \\ &\leq \sum_{d=1}^{\log n} 2^d(\log n - d) + c \log n = \log n \sum_{d=1}^{\log n} 2^d - \sum_{d=1}^{\log n} 2^d d + c \log n \\ &= (2(n-1)) \log n - 2(n \log n - n + 1) + c \log n \\ &= O(n). \end{aligned}$$

With regard to the *depth* of our algorithm, note that each recursion level has depth $O(\log n)$, due to our binary-search bottleneck; note as well that the recursion stops whenever the elements of $A_{d,i}$ become smaller than 2, which happens by recursion level $O(1 + \lceil \log_2 n \rceil)$. Hence we see that depth is $O(\log^2 n)$.

Remark on Concatenate You may have wondered why we assume that `concatenate` takes constant time. Realize that if we were to naively appending the elements of one array to another, this would require $O(n)$ work, since we must copy or move each element from one address in memory to another. Notice, however, that each element may be moved independent of other elements, hence depth is $O(1)$.

But we can do much better thank this. We can bring work down to $O(1)$ while still maintaining unit depth. There are two ways to do this. The first way is to manage our memory directly so that the two input arrays are placed contiguously in our random access memory. The second is to simply use an `if` statement whenever accessing elements in our output array. This `if` statement costs unit work, and hence we can still maintain our guarantee of constant time access to any element in the output array.

Alternative Solution There is a way to find the median of the union of two sorted arrays in $\log n$ time on one-machine. Hence this sub-routine has $\log n$ work (and depth, as written). It can be proven that this can be done. After which, we can see that

$$W(n) = 2W\left(\frac{n}{2}\right) + O(\log n),$$
$$D(n) = D(n/2) + O(\log n).$$

Using the Master Theorem, the results show $W(n) = O(n)$ and $D(n) = O(\log^2 n)$.

4. Given a sequence of n real numbers $\boldsymbol{s} = (s_1, ..., s_n)$, the maximum contiguous subsequence sum problem is to find a contiguous subsequence of $\boldsymbol{s}$ such that its sum is maximal, i.e.

$$F(\boldsymbol{s}) = \max_{1 \le i \le j \le n} \sum_{k=i}^{j} s_k$$

   (a) Consider the following algorithm running in parallel manner, what is the work and depth of it? (Hint: be aware of the depth and work of max and sum)

```
function MaxContiguousSubsequenceSum(s[1...n])
    for 1 ≤ i ≤ j ≤ n do
        Compute SubSum(i, j) = Σ_{k=i}^{j} s_k
    end for
    return max({SubSum(i, j)}_{1≤i≤j≤n})
end function
```

   Solution The given algorithm can be divided into two dependent parts, 1. Compute the sum of every subsequence $\boldsymbol{s}[i, i+1, ..., j]$; 2. find the maximum of those subsum results. It's easy to see the total depth and work are the sum of the two parts from DAG. To move forward, we recall the work and depth of max and sum operation in parallel manner

$$W_{sum}(n) = O(n), \qquad W_{max}(n) = O(n)$$
$$D_{sum}(n) = O(\log n), \quad D_{max}(n) = O(\log n).$$

Clearly, the sum of subsequences $SubSum(i, j)$ can be computed in parallel, thus the depth for this part is just the maximum of each job $SubSum(i, j)$, which is

$$D_{part\,1} = \max_{1 \le i \le j \le n} D_{sum}(j - i) = O(\log n).$$

While the work is the sum of all $SubSum(i, j)$, we have

$$W_{part\,1} = \sum_{1 \le i \le j \le n} W_{sum}(j - i) = \sum_{i=1}^{n} \sum_{j=i}^{n} O(j - i) = O(n^3).$$

The depth and work for the second part is quite obvious. By using the fact that $\sum_{i=1}^{n} \sum_{j=i}^{n} = n * (n + 1)/2$, we get

$$W_{part\,2} = W_{max}(n(n + 1)/2) = O(n^2)$$
$$D_{part\,2} = D_{max}(n(n + 1)/2) = O(\log n)$$

Thus, the depth and work in total is

$$W = W_{part\,1} + W_{part\,2} = O(n^3)$$
$$D = D_{part\,1} + D_{part\,2} = O(\log n)$$

(b) Give an algorithm to solve this problem that runs in O(n log n) work and O($\log^2$n) depth. Give a short proof about its work and depth. (Hint: Divide and Conquer)

Solution We consider the following divide and conquer algorithm

function MCSSDivideAndConquer($\boldsymbol{s}[1 \ldots n]$)
  if n is 1 **then**
    return s[1]
  else
    $L, R \leftarrow s[1 \ldots n/2], s[n/2 + 1 \ldots n]$
    $m_L \leftarrow MCSSDivideAndConquer(L)$
    $m_R \leftarrow MCSSDivideAndConquer(R)$
    $\text{Suff}_L, \text{Pref}_R \leftarrow \text{SuffixSum}(L), \text{PrefixSum}(R)$
    $m_A = \max(\text{Suff}_L) + \max(\text{Pref}_R)$
    return $\max(m_L, m_A, m_R)$
  end if
end function

First, we prove the correctness of the above algorithm. For trivial case where $n = 1$, the algorithm produces correct result. For $n > 1$, L and R are both non-empty. Let $\boldsymbol{s}[i \ldots j]$ be the contiguous subsequence of $\boldsymbol{s}$ that has the largest sum. There are three cases to consider:

i. Both i and j belong to L.

ii. i is in L but j is in R.

iii. Both i and j belong to R.

For cases 1 and 3, by induction, we must have $m_L \geq m_R$ ($m_R \geq m_L$), hence yielding the correct result. For case 2, we assert that $\boldsymbol{s}[i, \ldots, n/2]$ is the subsequence with the maximal suffix sum in $\boldsymbol{s}[1 \ldots n/2]$, while $\boldsymbol{s}[n/2+1 \ldots j]$ has the maximal prefix sum in $\boldsymbol{s}[n/2+1 \ldots n]$. Otherwise, $\sum_{k=i}^{j} s_k$ would not be maximal. Therefore,

$$\text{Suff}_L = \sum_{k=i}^{n/2} s_k,$$

$$\text{Pref}_R = \sum_{k=n/2+1}^{j} s_k.$$

Thus, $m_A = \sum_{k=i}^{j} s_k$ is the correct result.

Regarding the depth and work, we have

$$\begin{aligned} W(n) &= 2W(n/2) + W_{\text{PrefixSum}}(n/2) + W_{\text{SuffixSum}}(n/2) + 2W_{\max} \\ &= 2W(n/2) + O(n) \\ D(n) &= D(n/2) + \max(D_{\text{PrefixSum}} + D_{\text{SuffixSum}}) + D_{\max}(n) \\ &= D(n/2) + O(\log n) \end{aligned}$$

By the Master Theorem, $W(n) = O(n \log n)$ and $D(n) = O(\log^2 n)$.

5. In this problem, we'll look at how fast the maximum of a set of n elements can be computed when allowing for concurrent writes. In particular we allow the arbitrary write rule for "combining" (i.e. if there are a set of parallel writes to a location, one of them wins). Show that this can be done in $O(\log \log n)$ depth and $O(n)$ work.

(a.) Describe an algorithm for maximum that takes $O(n^2)$ work and $O(1)$ depth (using concurrent writes).

Solution For each element x_i, $1 \leq i \leq n$, we associate a bit initialized to have unit value. Notice that there are n bits to be initialized, hence work is $O(n)$ and depth is $O(1)$ since bits may be initialized independently.

For each pair of elements $x_i, x_j, i \leq j$, we make a comparison in parallel. Notice that the work is $\binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$, and the depth is 1, since each of the $\binom{n}{2}$ comparisons may be computed independently. We use $p = \binom{n}{2}$ processors, and for each comparison we attempt to *over-write* $b_i = 0$ if $x_i < x_j$ and $b_j = 0$ if $x_j < x_i$, i.e. if we can definitively say that an element is smaller than some other element in our input, its associated bit gets set to 0. This 0 encodes that the element is *not* a maximum.

Notice that we may end up with two processors writing to the same location in memory at the same time. However, notice that in our algorithm, we only at tempt to overwrite a bit if we turn it off. Hence we may allow arbitrary writes in the event of conflict, since all of the writes are trying to accomplish the same thing.

Notice that all bits whose associated value is 1 at the end of this process must have the same value, for if not we get a contradiction: fix attention to two such values; notice that since they have different values, they cannot be the same element, hence $i \neq j$, and thus at some-point in our algorithm they were considered. But if the element had different values, exactly one of them should have been turned off. Once a bit turns off, it never turns on again. Since we assume both associated bits have value 1, this is a contradiction.

Now, we need to *return* our result. Notice that if we were to naively loop through our bit sequence looking for a bit which is turned on, that this would take $O(n)$ time since there is no guarantee where the maximal element lies in the array. Instead, assign each of our n bits to a particular processor. In constant time, check whether the bit turned on. If it is, fetch the corresponding entry from the array in unit time and write it to output address in unit time. Although we do not have unit depth, we have a constant depth in our DAG which is *not* a function of n. Hence $T_\infty = O(1)$ for this algorithm.

Notice that by our argument in the previous paragraph, any writes which are concurrently attempted to output are all trying to write the same value, so again the arbitrary write rule causes no harm.

(b.) Use this to develop an algorithm with $O(n)$ work and $O(\log \log n)$ depth. Hint: use divide and conquer, but with a branching factor greater than 2.

Solution We use a Divide-And-Conquer algorithm with a branching factor of $n^{1/3}$. That is, we divide the array into $n^{1/3}$ blocks each of size $n^{2/3}$ elements, and recursively find their maximum. Notice that the recursion bottoms-out when $n \leq 2$. From the max elements of each of the $n^{1/3}$ blocks, compute the max using our brute-force parallel algorithm described in part (A), requiring $O(n^2)$ work. Then, we see that,

$$W(n) = n^{1/3}\, W(n^{2/3}) + O\left((n^{1/3})^2\right).$$

For depth, notice that the recursive calls from divide-and-conquer may be made in parallel, and the base-case (where we apply algorithm from part (a)) only requires $O(1)$ depth. So,

$$D(n) = D(n^{1/3}) + O(1).$$

Notice that we can *not* use the Master Theorem because in each recursive call we divide our input by $\sqrt[3]{n}$.

Solving recurrence relation for depth We first solve the recurrence relation for depth. We see that we have a constant amount of depth for each level of recursion, so we just need to solve for the number of recursion levels needed. We know $D(2) = O(1)$ (the recursion bottoms out when $n \leq 2$), so we use this fact to solve for the number of levels.

$$n^{1/3^k} = 2 \iff \frac{1}{3^k}\log(n) = \log(2) \iff \frac{\log(n)}{\log(2)} = \log_2(n) = 3^k \implies k = \log_3 \log_2(n)$$

We have constant depth for each level, so in total depth is $O(\log_3 \log_2(n)) = O(\log \log(n))$.

Solving recurrence relation for work Now we solve the work recurrence relation. We may use a similar technique for finding the number of levels in the recursion tree, again using the fact that $W(2) = O(1)$, and we get $k = \log_{3/2} \log_2(n)$ as the number of levels. However, we do not have a constant amount of work at each level, so we need to *un-roll* the relation some.

$$\begin{aligned}
W(n) &= n^{1/3} W(n^{2/3}) + O(n^{(1/3)^2}) \\
&= n^{1/3}\left[(n^{2/3})^{1/3} W\left(n^{(2/3)^2}\right) + O\left(n^{(2/3)^2}\right)\right] + O(n^{(1/3)^2}) \\
&= n^{1/3}\left[(n^{2/3})^{1/3}\left[(n^{(2/3)^2})^{1/3} W(n^{(2/3)^3}) + O(n^{(2/3)^3})\right] + O(n^{(2/3)^2})\right] + O(n^{(1/3)^2}) \\
&= \underbrace{O(n^{2/3}) + n^{1/3}O(n^{(2/3)^2}) + n^{1/3}(n^{2/3})^{1/3}O(n^{(2/3)^3}) + n^{1/3}(n^{2/3})^{1/3}(n^{(2/3)^2})^{1/3}O(n^{(2/3)^4}) + \ldots}_{\log_{3/2}\log_2(n) \text{ terms}}
\end{aligned}$$

We now have a series where each summand a product of terms.[6] We recognize a pattern, and combine terms

$$\begin{aligned}
&O(n^{(2/3)^1}) + n^{1/3}O(n^{(2/3)^2}) + n^{1/3}(n^{(2/3)^1})^{1/3}O(n^{(2/3)^3}) + n^{1/3}(n^{2/3})^{1/3}(n^{(2/3)^2})^{1/3}O(n^{(2/3)^4}) + \ldots \\
&= \sum_{j=1}^{\log_{3/2}\log_2(n)} \left(\prod_{i=0}^{j-2}(n^{(2/3)^i})^{1/3}\right) O(n^{(2/3)^j})
\end{aligned}$$

Now, we simplify algebra.[7]

$$\left(n^{(2/3)^i}\right)^{1/3} = n^{\frac{1}{3}\left(\frac{2}{3}\right)^i} \implies \prod_{i=0}^{j-2}\left(n^{(2/3)^i}\right)^{1/3} = n^{\frac{1}{3}\sum_{i=0}^{j-2}\left(\frac{2}{3}\right)^i} = n^{\frac{1}{3}\frac{1-(\frac{2}{3})^{j-1}}{1-\frac{2}{3}}} = n^{1-(\frac{2}{3})^{j-1}}$$

So,

[6] We caveat our notation: $n^{(2/3)^i} = n^{2^i/3^i} \neq (n^{2/3})^i = n^{2i/3}$.

[7] Recall for any geometric series with $x \in \mathbb{R}$, that $\sum_{i=0}^{k} x^i = \frac{1-x^{k+1}}{1-x}$.

$$W(n) = \sum_{j=1}^{\log_{3/2}\log_2(n)} \underbrace{\left(\prod_{i=0}^{j-2}(n^{(2/3)^i})^{1/3}\right)}_{n^{1-(\frac{2}{3})^{j-1}}} O(n^{(2/3)^j}) = \sum_{j=1}^{\log_{3/2}\log_2(n)} O\left(n^{1-(\frac{2}{3})^{j-1}+(\frac{2}{3})^j}\right)$$

$$= \sum_{j=1}^{\log_{3/2}\log_2(n)} O\left(n^{1-(\frac{2}{3})^{j-1}\left(1-\frac{2}{3}\right)}\right)$$

$$= \sum_{j=1}^{\log_{3/2}\log_2(n)} O\left(n^{\frac{3^j-2^{j-1}}{3^j}}\right)$$

$$= \sum_{j=1}^{\log_{3/2}\log_2(n)} O\left(n^{1-(2/3)^j/2}\right)$$

$$= \sum_{j=1}^{\log_{3/2}\log_2(n)} \frac{O(n)}{O(n^{(2/3)^j/2})}$$

$$= O(n)$$

We claim that the last equality holds. To see this, for given n, let $k = \log_{3/2}\log_2(n)$. Let $\delta = (2/3)^{(k-1)/2} > 0$. We may re-write our equation for work as

$$W(n) = \sum_{j=1}^{\log_{3/2}\log_2(n)} \frac{O(n)}{O(n^{(2/3)^j/2})}$$

$$= \frac{O(n)}{O(n^{(2/3)^{k/2}})} + \sum_{i=1}^{k-1} \frac{O(n)}{O(n^{(2/3)^{i/2}})}$$

$$= \frac{O(n)}{O((n^{(2/3)^k})^{1/2})} + \sum_{i=1}^{k-1} \frac{O(n)}{O(n^{(2/3)^{i/2}})}$$

$$= \frac{O(n)}{O((2)^{1/2})} + \sum_{i=1}^{k-1} \frac{O(n)}{O(n^{(2/3)^{i/2}})}$$

$$\leq O(n) + (k-1)\frac{O(n)}{O(n^{(2/3)^{(k-1)/2}})}$$

$$= O(n^{1-k/3}) + O\left((\log\log n)n^{1-\delta}\right)$$

$$= O(n)$$

6. **Interval Scheduling Problem** Given a set of n tasks with start time and finish time for each task, $T = \{(s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n)\}$, we want to find the largest subset $S \subseteq \{1, 2, ..., n\}$ such that for any pair i, j in S, the intervals (s_i, f_i) and (s_j, f_j) do not overlap. Here we assume $s_i < f_i$ holds for any task, which means every task takes positive time to finish. Design a greedy algorithm to solve this problem, and prove

that the resulting schedule is optimal.

Solution First, we present the greedy algorithm for solving the interval scheduling problem:

```
function IntervalScheduling(T)
    T ← sorted(T, key = lambda x : x[1])
    S ← {}
    f_prev ← −∞
    for (s_i, f_i) in T do
        if s_i ≥ f_prev then
            S.add((s_i, f_i))
            f_prev ← f_i
        end if
    end for
    return S
end function
```

Optimality proof: To prove the algorithm gives optimal scheduling, we will prove that for any feasible scheduling $S_{arb} = \{(s_{k_1}, f_{k_1}), \dots (s_{k_m}, f_{k_m})\}$, it must hold that $|S_{arb}| \leq |S_G|$, where S_G is the result given by the above Greedy Algorithm.

Suppose for $S_G = \{(s_{G_1}, f_{G_1}), \dots (s_{G_l}, f_{G_l})\}$, there exists some feasible scheduling $S_{arb} = \{(s_{k_1}, f_{k_1}), \dots (s_{k_m}, f_{k_m})\}$ such that $|S_{arb}| > |S_G|$. Here, the tasks are non-overlapping and ordered increasingly, i.e.

$$s_{G_1} < f_{G_1} \leq s_{G_2} < f_{G_2} \leq \cdots \leq s_{G_l} < f_{G_l}$$
$$s_{k_1} < f_{k_1} \leq s_{k_2} < f_{k_2} \leq \cdots \leq s_{k_m} < f_{k_m}.$$

According to the greedy algorithm, (s_{G_1}, f_{G_1}) is the first element in the sorted task set T, implying $f_{G_1} \leq f_i$ for $\forall (s_i, f_i) \in T$. Therefore, we must have $f_{G_1} \leq f_{k_1}$.

Note that by assumption we have $|S_G| = l < m = |S_{arb}|$, there must exist some (s_{G_j}, f_{G_j}) which has the largest index such that $f_{G_i} \leq f_{k_i}$.

If (s_{G_j}, f_{G_j}) is the last element in S_G, i.e. $j = l$, then according to the greedy algorithm, there isn't any task simultaneously holds the two properties: 1. $s_i \geq f_{G_l}$; 2. $f_i \geq f_{G_l}$. Otherwise, there must exist some task comes later than (s_{G_l}, f_{G_l}) and has the property that $s_i \geq f_{prev} = f_{G_l}$, which implies it would be added to S_G, contradict to the statement (s_{G_j}, f_{G_j}) is the last element in S_G.

If (s_{G_j}, f_{G_j}) is not the last element in S_G, then we have $f_{G_j} \leq f_{k_j}$ and $f_{k_{j+1}} < f_{G_{j+1}}$. Therefore we have

$$f_{G_j} \leq f_{k_j} \leq s_{k_{j+1}} < f_{k_{j+1}} < f_{G_{j+1}},$$
$$f_{G_j} \leq s_{k_{j+1}}, \qquad f_{k_{j+1}} < f_{G_{j+1}}.$$

So after (s_{G_j}, f_{G_j}) added to S_G by greedy algorithm, $(s_{k_{j+1}}, f_{k_{j+1}})$ comes earlier than $(s_{G_{j+1}}, f_{G_{j+1}})$ and can be added to S_G if $f_{\text{prev}} = f_{G_j}$. The fact that there's no task between (s_{G_j}, f_{G_j}) and $(s_{G_{j+1}}, f_{G_{j+1}})$ implies $(s_{k_{j+1}}, f_{k_{j+1}})$ is not included into S_G, thus $f_{\text{prev}} \neq f_{G_j}$. This is only possible if there's been some other task added into S_G before $(s_{k_{j+1}}, f_{k_{j+1}})$, thus before $(s_{G_{j+1}}, f_{G_{j+1}})$. This contradicts to $(s_{G_{j+1}}, f_{G_{j+1}})$ is the next task added to S_G after (s_{G_j}, f_{G_j}).

Hence for any feasible scheduling $S_{arb} = \{(s_{k_1}, f_{k_1}), \ldots (s_{k_m}, f_{k_m})\}$, it must hold that $|S_{arb}| \leq |S_G|$. This shows the optimality of the greedy algorithm.

7. **Solving Linear Systems**

Lower Triangular Systems Consider the task of solving the linear system $Ax = b$ where we assume A is lower triangular. A popular method for solving $Ax = b$ is *forward substitution*. The forward substitution algorithm can be represented as the following series of serial updates:

$x_1 \leftarrow b_1/a_{11}$
for $i = 2, \ldots, n$ **do**
  $x_i \leftarrow \left(b_i - \sum_{j=1}^{i-1} l_{ij} x_j\right) / a_{ii}$
end for

(a) What is the computation complexity of the forward substitution algorithm?

Solution At iteration k, the work is $O(k)$. So the total work is $O(\sum_{k=1}^{n} k) = O(n^2)$

The parallel forward substitution algorithm operates by parallelizing the serial forward substitution algorithm. Note that the y_j updates can all be executed in parallel.

$x_1 \leftarrow b_1/a_{11}$
for $i = 2, \ldots, n$ **do**
  $x_i \leftarrow (b_i - y_i)/a_{ii}$
  for $j = i + 1, \ldots, n$ **do**
    $y_j \leftarrow a_{j1}x_1 + \ldots + a_{ji}x_i$
  end for
end for

(b) Construct the DAG representing the parallel forward substitution algorithm. What is the depth of the DAG?

Solution At iterations k, updating y_j for $j > k$ can all be completed in parallel and thus have $O(1)$ depth. Updating x_k also has $O(1)$ depth so the total depth is $O(n)$

Tridiagonal Systems We now consider solving the system $Ax = b$ where A is tridiagonal. Explicitly, $a_{ij} = 0$ if $|i - j| \geq 2$. Note that this is equivalent to solving the

following system of linear equations:

$$g_1 x_1 + h_1 x_2 = b_1 \tag{1}$$
$$f_i x_{i-1} + g_i x_i + h_i x_{i+1} = b_i, \quad i = 2, \ldots, n-1 \tag{2}$$
$$f_n x_{n-1} + g_n x_n = b_n \tag{3}$$

where g_i are the diagonal elements of A, f_i the entries below the diagonal, and h_i the entries above the diagonal. The idea behind *even-odd reductions* is to recursively reduce the above system to one of half the size. Explicitly, if none of the diagonal entries are zero, we can solve for each x_i in terms of x_{i-1} and x_{i+1}. If we do this for all odd i, and substitute the expression back in, we obtain a system on just the even indexed variables.

(a) Using the above system of equations, derive a tridiagonal system of equations on just the even indexed variables.

Solution For simplicity, we let $x_j = 0$ for $j \leq 0$ and $j \geq n+1$ so we can account for the edge cases. Solving for x_i in (2) we obtain

$$x_i = \frac{1}{g_i}(b_i - f_i x_{i-1} - h_i x_{i+1})$$

. We plug this back into (2) to get

$$\frac{f_i}{g_{i-1}}(b_i - f_{i-1}x_{i-2} - h_{i-1}x_i) + g_i x_i + \frac{h_i}{g_{i+1}}(b_{i+1} - f_{i+1}x_i - h_{i+1}x_{i+2}) = b_i$$

This simplifies to

$$-\left(\frac{f_i f_{i-1}}{g_{i-1}}\right) x_{i-2} + \left(g_i - \frac{h_{i-1}f_i}{g_{i-1}} - \frac{h_i f_{i+1}}{g_{i+1}}\right) x_i - \left(\frac{h_i h_{i+1}}{g_{i+1}}\right) x_{i+2} = b_i - \frac{f_i}{g_{i-1}} b_{i-1} - \frac{h_i}{g_{i+1}} b_{i+1} \tag{4}$$

(b) What is the computational complexity of computing the coefficients of the reduced system?

Solution Computing the coefficients on the left hand side of (4) requires 8 multiplies and 2 sums. Computing the coefficient on the right hand side of (4) requires 2 multiplies and 2 sums. Each of the multiplies can be completed in parallel, while the sums require the results of the multiplications. This results in a total of 14 operations (work = $O(1)$ and depth = $O(1)$) to compute one instance of (4). $\frac{n}{2}$ of these equations make up the reduced tridiagonal system of equations (each of these equations can be computed in parallel). Thus, the reduction requires $O(n)$ work and $O(1)$ depth.

The above procedure can be recursively applied until the problem is reduced to a single equation. Then we work backwards to solve for the value of the eliminated variables.

(c) What is the computational complexity of solving for the eliminated variables?

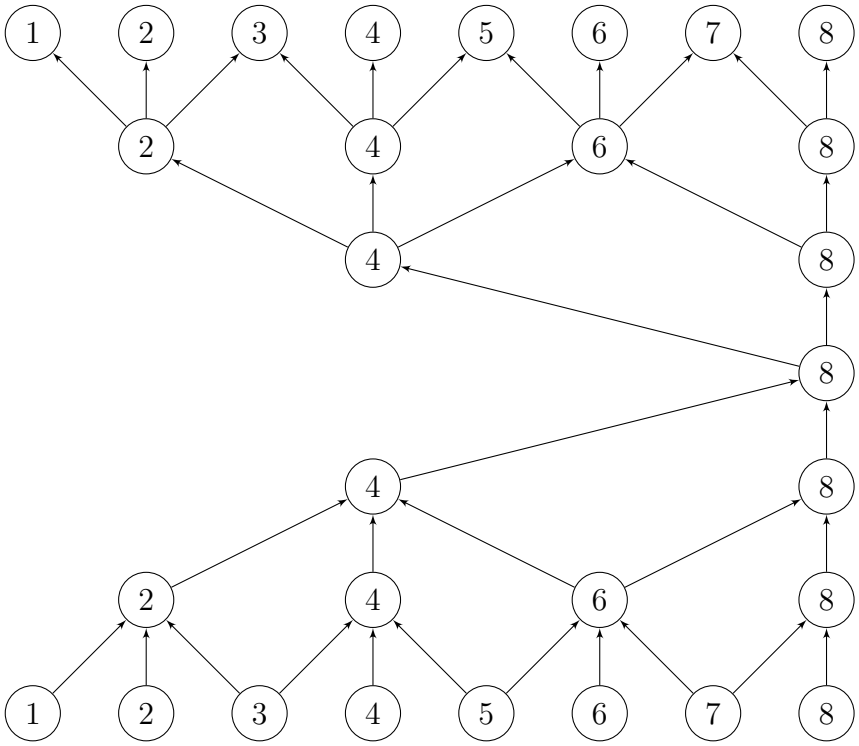
Solution If we cache the coefficients computed during the reduction phases, we do not need to recompute them during the steps where we back solve for the reduced variables. For the backsolve, we need to solve an equation of the form

$$c_{-1}x_{i-2^k} + c_0 x_i + c_1 x_{i+2^k} = \tilde{b}$$

for x_i where all variables and constants besides x_i are known. This can be executed in $O(1)$ depth and $O(1)$ work. There are $\frac{n}{2}$ such equations (to recover $\frac{n}{2}$ unknowns from $\frac{n}{2}$ knowns).

(d) Construct the DAG representing this algorithm.

Solution Figure depicts the even-odd reduction for n = 8. At the first stage, variables x_1, x_3, x_6, and x_8 are eliminated using equation (4). The algorithm recursively elliminates variables until only x_8 remains and is evaluated. The remaining variables are recursively evaluated until all variables have been solved for.



(e) What is the runtime of the even odd reduction algorithm on $\Theta(n)$ processors? From the previous parts, we conclude that the depth of this algorithm is $O(\log_2(n))$ so $T_\infty = O(\log_2(n))$. We now calculate T_1. $T_1 = O(\sum_{k=1}^{\log_2 n} \frac{n}{2^k})$.

$$\begin{aligned}
\sum_{k=1}^{\log_2 n} \frac{n}{2^k} &= n \sum_{k=1}^{\log_2 n} \frac{1}{2^k} \\
&\leq n \sum_{k=1}^{\infty} \frac{1}{2^k} \\
&= n
\end{aligned}$$

So $T_1 = n$. Consequently, we can apply Brent's theorem to get

$$O(\frac{n}{p}) \leq T_p \leq O(\frac{n}{p}) + O(\log_2(n))$$

Givens Rotations Givens Rotations are used to zero out the subdiagonal entries of the matrix A one at a time. Crucially, a Givens rotation only affects two rows of the matrix. We will use this fact to derive a parallel implementation of the Givens rotation algorithm. Specifically, if two successive Givens rotations affect disjoint sets of rows, then they can be computed in parallel.

(a) When n rows are available, what is the maximum number of Givens rotations we can apply simultaneously?

Solution Each Given's rotation affects two rows, so the maximum number we can apply simultaneously are $\lfloor \frac{n}{2} \rfloor$.

Implementing the Givens rotations in parallel ultimately comes down to deriving a schedule of the entries to eliminate at a particular step. We consider two functions $T(j,k)$ and $S(j,k)$ where $T(j,k)$ represents the iteration in which the jkth entry is eliminated, and j and $S(j,k)$ are the rows the Givens rotation operates on. To simulateneously implement the Givens rotations, we require that $T(j,k)$ and $S(j,k)$ satisfy:

- If $T(j,k) = T(j',k')$ and $(j,k) \neq (j',k')$ then $\{j, S(j,k)\} \cap \{j', S(j',k')\} = \emptyset$.
- If $T(j,k) = t$ and $S(j,k) = i$, then $T(j,l) < t$ and $T(i,l) < t$ for all $l < k$.

(b) Prove that the schedule given by

$$T(j,k) = n - j + 2k - 1$$
$$S(j,k) = j - 1$$

satisfies the above conditions.

Solution Suppose that $T(j,k) = T(j',k')$. Then $-j + 2k = -j' + 2k'$. If $j = j'$ then $k = k'$. If $j \neq j'$, then $j - j'$ is even and in particular, $|j - j'| \geq 2$. Therefore, the sets $\{j, j-1\}$ and $\{j', j'-1\}$ are disjoint and the first property is satisfied. For the second property,

$$T(S(j,k),l) = T(j-1,l) = T(j,l) + 1 = n - j + 2l < n - j + 2k - 1 = T(j,k).$$

(c) What is the maximum number of stages required by this schedule?

Solution We find the maximum number of stages by maximizing $T(j,k)$ over all j and k. This is attained by $j = n$ and $k = n - 1$, and so $T(j,k) = 2n - 3$.