CME 323: Distributed Algorithms and Optimization

Instructor: Reza Zadeh (rezab@stanford.edu)
TA: Anuj Nagpal (anujnag@stanford.edu)

HW#1 – Solutions

1. **(5 points)** 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 $\mathbf{km}(x, y, n)$:
   if $n = 1$ **then**
   return $x \times y$
   else
   $a \leftarrow \mathbf{km}(x_l, y_l)$
   $b \leftarrow \mathbf{km}(x_h, y_h)$
   $c \leftarrow \mathbf{km}(x_l + x_h, y_l + y_h)$
   $d \leftarrow c - a - b$
   return $(b2^n + d2^{n/2} + a)$
   end if

   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

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

2. **(5 points)** 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 subtract 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. **(8 points) Curly Brace Matching:** We want to make an algorithm that solves curly brace matching problem i.e. given a code snippet as a string with whitespaces removed, we want to identify whether each opening curly brace '{' in the string has a corresponding closing curly brace '}'. For example, "{{a = 2}{{b += 4}{c = a - b}}}" has valid curly brace matching but "{{{a = 4}{b = 2}}}}" or "{a*=2}{{b=3}}{c=a+b{}" does not.

   (a) Give a sequential algorithm to solve the curly brace matching problem using a stack. Your algorithm must take $O(n)$ work and $O(n)$ extra space.

   (b) Give a sequential algorithm to solve the problem without a stack such that your algorithm takes $O(n)$ work but $O(1)$ extra space.

   (c) Give a parallel algorithm to compute the total number of left curly braces "{" and the right parentheses "}", respectively (if these two numbers do not match, we directly know that the string does not have a valid curly brace matching). Your algorithm must run in $O(n)$ work and $O(\log n)$ depth. You can use $O(n)$ extra space.

   (d) Give a parallel algorithm to solve the curly brace matching problem. Your algorithm must take $O(n)$ work and $O(\log n)$ depth. You can use $O(n)$ extra space. Prove the cost of your algorithm.

Solution

(a) To check if a code snippet is a valid match, the basic idea is to scan the code string from left to right and cancel every left curly brace with the next right curly brace that comes after it. We can maintain a stack to do this such that whenever we see a left curly brace "{", we can push it in the stack. Likewise whenever a right curly brace is read, we can pop one "{" out from the stack. The algorithm returns true as long as there are enough "{"s in the stack to use for the right curly brace, and there are no "{" left in the stack at the end.

(b) Since there can only be "{" in the stack, we can just record the number of "{" by a counter instead of maintaining a stack. We can scan the string from left to right and maintain k to record the number of non-matched "{" in the stack so far. Whenever we see a "{", we increment this counter by 1 and whenever we see a "}", we decrement it by 1. As long as this counter stays non-negative during the whole process and its value is 0 at the end, the string has valid curly brace matching.

(c) To compute the total number of left curly braces, we can first look at the elements of input string in parallel and use a 0-1 indicator array to denote if the i-th element is a "{". Then using a parallel summation algorithm, we can get the total number of "{"s in the input with $O(n)$ work and $O(\log n)$ depth. Similar idea applies for counting "}"s.

(d) We can first create an array X of size n such that $X[i] = 1$ or -1 if the i-th element in the input string is "{" or "}" respectively, and 0 otherwise. Now we can run the parallel prefix sum algorithm on X and get array S. If $S[i] \geq 0$ for all i, and $S[n] = 0$, the input string has valid curly brace matching. The cost of the algorithm is bounded by the parallel prefix sum algorithm, which has $O(n)$ work and $O(\log n)$ depth.

4. **(7 points)** 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.

Algorithm 1: Parallel Merge

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

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 $O(1 + \log_2 n)$. 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 $D(n) = O(\log^2 n)$.

5. **(6 points)** Given the price of a stock at each day for n days, we want to determine the biggest profit we can make by buying one day and selling on a later day. For example, the following stock prices have a best profit of 5:

   $[12, 11, 10, 8, 5, 8, 9, 6, 7, 7, 10, 7, 4, 2]$

   since we can buy at 5 on day 5 and sell at 10 on day 11. This has a simple linear time serial solution. Give an algorithm to solve this problem that runs in $O(n)$ work and $O(\log n)$ depth. Give pseudocode as in the previous problem.

Solution We will use a function called `min_scan`, which takes as input an array of numbers length n (call it A) and outputs an array of length n; in each index i of the output is the minimum number of $A[0 : i]$, i.e. the minimum number up to that point in the original array. We may implement this function as we do other scan functions, including all-prefix sum, to have $O(n)$ work and $O(\log n)$ depth.

Our entire algorithm for the stock-market problem can be described as follows.

Algorithm 2: Buy Low Sell High

Input : an array X of stock prices containing n elements
Output: The max-value we could achieve (non-negative)

```
mins ← min_scan(X)          // done in parallel
max_gains ← X − mins        // element-wise subtraction, in
    parallel
max_val ← max{max_gains}
if max_val > 0 then return max_val ;
else return 0;
```

Our algorithm does not involve any explicit recursive calls. Element-wise subtraction of n elements involves n independent computations, hence has work $O(n)$ and depth $O(1)$. Computing the maximal value of an n-array can be done with $O(n)$ work and $O(\log n)$ depth as seen in the following problem. The last step of our algorithm performs a constant time check before returning the output.

6. **(10 points)** 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.

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

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

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

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 $\sum_{i=0}^{k} x^i = \frac{1-x^{k+1}}{1-x}$.

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

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

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

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

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

$$= O(n)$$

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

$$\begin{aligned}
W(n) &= \sum_{j=1}^{\log_{3/2}\log_2(n)} \frac{O(n)}{O(n^{(2/3)^j/2})} \\
&= \frac{O(n)}{O(n^{(2/3)^k/2})} + \sum_{i=1}^{k-1} \frac{O(n)}{O(n^{(2/3)^i/2})} \\
&= \frac{O(n)}{O((n^{(2/3)^k})^{1/2})} + \sum_{i=1}^{k-1} \frac{O(n)}{O(n^{(2/3)^i/2})} \\
&= \frac{O(n)}{O((2)^{1/2})} + \sum_{i=1}^{k-1} \frac{O(n)}{O(n^{(2/3)^i/2})} \\
&\le O(n) + (k-1)\frac{O(n)}{O(n^{(2/3)^{(k-1)/2}})} \\
&= O(n^{1-k/3}) + O\left((\log\log n)n^{1-\delta}\right) \\
&= O(n)
\end{aligned}$$

7. **(8 points) Sorting Student IDs:** Consider the task of sorting a list of 8-digit Stanford Student IDs of the form $d_1 d_2 ... d_8$ where each d_i is a positive integer with $0 \le d_i \le 9$. This task can be done using standard sorting algorithms in $O(n \log n)$ time complexity but we can utilize the fact that all the numbers we want to sort are positive integers, whose each digit can be bucketed into 10 categories.

(a) Based on the bucketization idea, design a sequential algorithm that starts scanning the numbers from digit d_8 to d_1 and sorts them in $O(8n)$ time and $O(n)$ space complexity. Also provide pseudocode for same. (note that the idea behind the factor of 8 in time complexity is to hint that the algorithm can be extended to k-digit integers with $O(kn)$ time complexity).

(b) Using the same bucketization idea, design a parallel algorithm that starts scanning the numbers from digit d_1 instead and sorts them with $O(n \log_{10} n)$ work, $O(n)$ depth and $O(n)$ memory requirement (assume that this bucketization can't be done in parallel). Also provide pseudocode for same.

(c) Explain how can we improve the depth to $O(\log_{10} n)$ if we can also do the bucketization of elements in parallel. Assume you have sufficient number of processors.

Solution

(a) We can sort the IDs in k passes of the data, one for each digit position such that for j-th pass, we sort all the data based on the digit at position j from right. Since we only have to sort single digits, we can instead place each element in one of 10 buckets, with each bucket corresponding to a certain digit. This allows us to avoid the $O(\log n)$ barrier of comparison sorts. This algorithm is also known by the name of **radix sort**:

Algorithm 3: Sequential Radix Sort

```
function SequentialRadixSort(data):
    buckets = list of 10 buckets
    for i from 8 to 1 do
        for entry in data do
            d = d_i in entry
            Place entry in buckets[d]
        end
    end
    Replace data with the elements from buckets (in the same order)
    return data
end function
```

Since the algorithm goes through k passes of the data and performs constant work for each element, the total runtime is $O(kn)$. The space complexity is $O(n)$ since we maintain buckets as an auxiliary data structure for each pass.

(b) This time, we start sorting from the most significant digit instead. After the first pass through the data, we have 10 buckets just as previously, and can make recursive calls to sort each bucket. Furthermore, since the buckets do not depend on each other, these calls are independent, and can be parallelized:

Algorithm 4: Parallel Radix Sort

```
function ParallelRadixSort(data, i):
    buckets = list of 10 buckets
    if size(data)== 1 then
        return data
    end
    for entry in data do
        d = d_i in entry
        Place entry in buckets[d]
    end
    if i ≤ 8 then
        parallel for bucket in buckets do
            bucket = ParallelRadixSort(bucket, i+1)
        end
    end
    Replace data with the elements from buckets (in the same order)
    return data
```

The work and depth can be represented as follows:

$$W(n) = 10W(n/10) + O(n)$$

$$D(n) = D(n/10) + O(n)$$

which can be solved via the Master Theorem to give $W(n) = O(n \log_{10} n)$ and $D(n) = O(n)$.

(c) If the bucketization of elements can be done in parallel, we can assign each processor an equal chunk of the data to place in the buckets instead of scanning through the elements sequentially, since each element can be placed in a bucket independently. This gives a depth of:

$$D(n) = D(n/10) + O(1)$$

which can be solved to give $D(n) = O(\log_{10} n)$.

8. **(8 points) Scheduling to Minimize Lateness** Consider a problem where we have a single resource and a set of n requests to use the resource for an interval of time. Assume each request has a deadline d_i, and a length t_i required to complete the request. For each job we need to assign a start time $s(i)$ (its finish time is $f(i) = s(i) + t_i$). We say request i is late if $f(i) > d_i$, and the lateness of task i is given by $l_i = max\{0, f(i) - d_i\}$. Design a greedy algorithm that minimizes the maximum lateness across all jobs, and prove that the resulting schedule is optimal.

Solution We consider a greedy algorithm where we schedule the jobs in increasing order of their deadlines d_i. Without loss of generality, assume that the indices of the jobs correspond with their sorted deadlines (i.e., $d_1 \leq, \ldots, \leq d_n$). Our proof strategy will consist of converting any optimal solution step-by-step into our solution. We show that during each step, the maximum lateness does not increase. Consequently, our solution must have the same maximum lateness as the optimum.

Claim: There is an optimal schedule with no idle time. An idle time is a gap in a schedule, time when no job is being performed yet there are jobs left.

Proof: It is easy to see that there must exist an optimal schedule that has no idle time. We can transform an optimal schedule having idle times into one having none by scheduling jobs earlier during the gaps. This can only decrease their finishing time and hence lateness, never increase it.

Observation: Our solution has no idle time.

Let us say that a schedule A has an inversion if it schedules a job i before another job j but $d_i > d_j$.

Observation: Our solution has no inversion.

Claim: All schedules with no inversions and no idle time have the same maximum lateness.

Proof: If two different schedules have neither inversions nor idle times, they can only differ in the order in which jobs with identical deadlines are scheduled. Among all such jobs having the same deadline, the last one has the greatest lateness and this lateness does not depend on the order of the jobs.

Claim: There is an optimal schedule that has no inversions and no idle time.

Proof: We will prove this by using an exchange argument. Let OPT be an optimal schedule. If OPT has an inversion, then there is a pair i and j such that j is scheduled immediately after i and $d_j < d_i$. After swapping i and j, we get a schedule with one less inversion. Since job j finishes earlier now than in OPT, its lateness cannot increase. In the new schedule, job i finishes at time $f(j)$ – the finishing time of job j in OPT. Its lateness is $f(j)–d_i < f(j)–d_j$, which is less than the lateness of job j in OPT. Hence, swapping jobs i and j does not increase the maximum lateness of the schedule. Since there are at most $\binom{n}{2}$ inversions, hence after at most $\binom{n}{2}$ swaps, we get an optimal schedule with no inversions.

From the fact that there is an optimal schedule having no inversions and no idle time combined with the facts that all schedules with no inversions and no idle time have the same maximum lateness and that our schedule has no inversions and no idle time, we can conclude that our schedule is optimal.

This proves the optimality of the greedy algorithm.

9. **(15 points) 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**
  $x_i \leftarrow \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j\right)/a_{ii}$
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 $O(\sum_{k=1}^{n} k) = O(n^2)$

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

What is the depth of the DAG representing the parallel forward substitution algorithm?

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:

$$g_1x_1 + h_1x_2 = b_1 \tag{1}$$
$$f_ix_{i-1} + g_ix_i + h_ix_{i+1} = b_i, \quad i = 2, \ldots, n-1 \tag{2}$$
$$f_nx_{n-1} + g_nx_n = b_n \tag{3}$$

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?

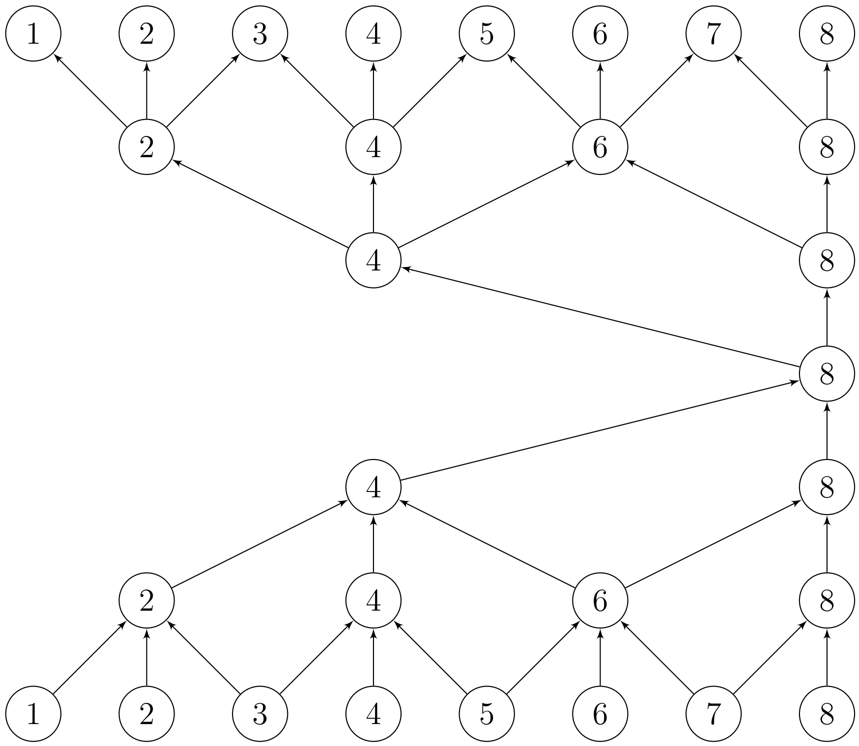
Solution If we cache the coefficients computed during the reduction phases, we do not need to recompute them during the steps where we back solve for the reduced variables. For the backsolve, we need to solve an equation of the form

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

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

$$O(\frac{n}{n}) \leq T_p \leq O(\frac{n}{n}) + O(\log n) \Rightarrow T_p \leq O(\log n)$$

10. **(8 points) 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 Givens rotation affects two rows, so the maximum number we can apply simultaneously are $\lfloor \frac{n}{2} \rfloor$.

(b) 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 jk-th 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 $l < k$.

Prove that the schedule given by

$$\begin{aligned} T(j,k) &= n - j + 2k - 1 \\ S(j,k) &= j - 1 \end{aligned}$$

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, we can verify that for all $l < k$

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

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

(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 that correspond to subdiagonal entries i.e. $k = j - 1$. This is attained by $j = n$ and $k = n - 1$, and so $T(j,k) = 2n - 3$.