CME 323: Distributed Algorithms and Optimization, Spring 2020
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 5, 4/14/2020. Scribed by Andreas Santucci, Edited by Robin Brown.

5 Memory Management and (Seemingly) Trivial Operations

We have to be extremely careful when analyzing work and depth for parallel algorithms. Even the most trivial operations must be dealt with care. For example, even an operation as trivial as `concatenate`, i.e. in the case where we wish to concatenate two arrays together. Suppose we have two arrays A, B each of length n. If A and B are located in two different places in RAM then we might have a problem. The usual way to deal with these things is to allocate the output array which costs constant time regardless of input size. Then, if we wish to concatenate two arrays into our output, we simply give one thread half the array and the other thread the other half of the array memory. When both threads are done, it's as though we have already performed a concatenate operation for free.

If allowed each thread to return its own array in the usual manner, we can still concatenate the two with $D(n) = O(1)$. To do this, we simply assign each processor one element to be copied or moved in RAM and perform the n operations in parallel. Hence depth is constant. However, we still need to perform $O(n)$ work in this case.

We assume that we can allocate memory in constant time, as long as we don't ask for the memory to have special values in it. That is, we can request a large chunk of memory (filled with garbage bit sequences) in constant time. However, requesting an array of zeros already requires $\Theta(n)$ work since we must ensure the integrity of each entry[1]. In the context of sequential algorithms, this is not a concern since reading in an input of n bits, or outputting n bits already requires $\Theta(n)$ work, so zeroing out an array of size n does not dominate the operation count. However, in some parallel algorithms, no processor reads in the entire input, so naively zeroing out a large array can easily dominate the operation time of the algorithm.

6 QuickSort

With this in mind, we will now finish the analysis of QuickSort, taking a closer look at memory management dueing the algorithm. The algorithm is as follows.

6.1 Analysis on Memory Management

Recall that in Lecture 4, we designed an algorithm to construct L and R in $O(n)$ work and $O(\log n)$ depth. We will take this opportunity to highlight some of the intricacies of memory management during the construction of L and R.

[1] In practice, zeroing out an array is optimized by hardware and is not a concern.

Algorithm 1: QuickSort

Input: An array A

Output: Sorted A

1 $p \leftarrow$ element of A chosen uniformly at random

2 $L \leftarrow [a | a \in A \text{ s.t. } a < p]$ `// Implicitly:` $B_L \leftarrow \mathbb{1}\{a_i < p\}_{i=1}^n$`,` `prefixSum`$(B_l)$`,`

3 $R \leftarrow [a | a \in A \text{ s.t. } a > p]$ `// which requires` $\Theta(n)$ `work and` $O(\log n)$ `depth.`

4 **return** [QuickSort(L), p, QuickSort(R)]

Selecting a pivot uniformly at random We denote the size our input array A by n. To be precise, we can perform step 1 in $\Theta(\log n)$ work and $O(1)$ depth. That is, to generate a number uniformly from the set $\{1, 2, \ldots, n\}$ we can assign $\log n$ processors to independently flip a bit "on" with probability $1/2$. The resulting bit-sequence can be interpreted as a $\log_2$ representation of a number from $\{1, \ldots, n\}$.

Allocating storage for L and R Start by making a call to the OS to allocate an array of n elements; this requires $O(1)$ work and depth, since we do not require the elements to be initialized. We compare each element in the array with the pivot, p, and write a 1 to the corresponding element if the element belongs in L (i.e. it's smaller) and a 0 otherwise. This requires $\Theta(n)$ work but can be done in parallel, i.e. $O(1)$ depth. We are left with an array of 1's and 0's indicating whether an element belongs in L or not, call it $\mathbb{1}_L$,

$$\mathbb{1}_L = \mathbb{1}\{a \in A \text{ s.t. } a < p\}.$$

We then apply `PrefixSum` on the indicator array $\mathbb{1}_L$, which requires $O(n)$ work and $O(\log n)$ depth. Then, we may examine the value of the last element in the output array from `prefixSum` to learn the size of L. Looking up the last element in array $\mathbb{1}_L$ requires $O(1)$ work and depth. We can further allocate a new array for L in constant time and depth. Since we know $|L|$ and we know n, we also know $|R| = n - |L|$; computing $|R|$ and allocating corresponding storage requires $O(1)$ work and depth.

Thus, allocating space for L and R requires $O(n)$ work and $O(\log n)$ depth.

Filling L and R Now, we use n processors, assigning each to exactly one element in our input array A, and in parallel we perform the following steps. Each processor $1, 2, \ldots, n$ is assigned to its corresponding entry in A. Suppose we fix attention to the kth processor, which is responsible for assigning the kth entry in A to its appropriate location in either L or R. We first examine $\mathbb{1}_L$`[k]` to determine whether the element belongs in L or R. In addition, examine the corresponding entry in `prefixSum` output, denote this value by $i =$ `prefixSum`$(\mathbb{1}_L)$`[k]`. If the kth entry of A belongs in L, then it may be written to the position i in L immediately, by definition of how what our `prefixSum` output on $\mathbb{1}_L$ means. If the kth entry instead belongs in R, then realize that index i tells us that exactly i entries "before" element k belong in L. Hence exactly $k - i$ elements belong

in array R before element $A[k]$. Hence we know exactly where to write the kth element to R if it belongs there.

Clearly, the process of filling L and R requires $O(n)$ work and $O(1)$ depth.

Work and Depth per Iteration Thus we may say that steps 1,2,3 of our algorithm require $O(n)$ work and $O(\log n)$ depth.[2] The last step of the algorithm requires recursive calls to `Quicksort` and a concatenation of several elements. Realize that if we are clever about how we allocate the memory for our arrays to begin with, this "concatenation" (or lack thereof) can be performed without extra work or depth.[3]

6.2 Total Expected Work

Now we analyze the total expected work of QuickSort.[4] We assume all our input elements are unique.[5] On a very high level, to compute the work, note that the work done summed across each level of our computational DAG is n. There are $\log_{4/3} n$ levels in our DAG, hence expected work given by

$$\mathbb{E}[T_1] = \mathbb{E}[\#\text{ levels}] \cdot \mathbb{E}[\text{work per level}] = O(n \log n)$$

Details We define the random indicator variable X_{ij} to be one if the algorithm *does compare* the ith *smallest* and the jth *smallest* elements of input array A during the course of its sorting routine, and zero otherwise. Let X denote the *total* number of comparisons made by our algorithm. Then, we have that

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

Number of Comparisons We take a moment to realize why we sum over $\binom{n}{2}$ elements instead of n^2: the only time in our `QuickSort` algorithm whereby we make a comparison of two elements is when we construct L and R. In doing so, we compare each element in the array to a fixed pivot of p, after which all elements in L less than p and all elements in R greater than p. Realize that pivot p is never compared to elements in L and R for the remainder of the algorithm. Hence each of the $\binom{n}{2}$ pairings are considered *at most* once by our `QuickSort` algorithm.

[2] It may be tempting to think that we can get away with *not* calculating a `prefixSum` on $\mathbb{1}_L$ in order to determine the sizes of L and R, and instead pass this indicator array $\mathbb{1}_L$ to our recursive call. We might think that we can then avoid incurring $\log n$ depth. However, realize that then when we pass $\mathbb{1}_L$ to our recursive call, it would still be of size n, hence we would not be decreasing the size of our problem with each recursive call. This would result in quite a poor algorithm.

[3] For details, see lecture 4 notes, specifically the section on "Parallel Algorithms and (Seemingly) Trivial Operations".

[4] We follow the analysis from CMU very closely.[1]

[5] If there are duplicate elements in our input A, then this only cuts down the amount of work required. As the algorithm is written, we look for strict inequality when constructing L and R. If we ever select one of the duplicated elements as a pivot, its duplicates values are not included in recursive calls hence the size of our sub-problems decreases even more than if all elements were unique.

Expected Number of Comparisons Realize that expectation operator $\mathbb{E}$ is monotone, hence

$$\mathbb{E}[X] \leq \mathbb{E}\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\mathbb{E}[X_{ij}],$$

where the equality follows from linearity of expectation. Since X_{ij} an indicator random variable,

$$\mathbb{E}[X_{ij}] = 0 \cdot \Pr(X_{ij} = 0) + 1 \cdot \Pr(X_{ij} = 1) = \Pr(X_{ij}).$$

Consider any particular X_{ij} for $i < j$ (i.e. one of interest). Denote the ith smallest element of input array A by $\text{rank}_A^{-1}(i)$ for $i = 1, 2, \ldots, n$. For any one call to `QuickSort`, there are exactly *three* cases to consider, depending on how the value of the pivot compares with $A[i]$ and $A[j]$.

- **Case 1: $p \in \{\text{rank}_A^{-1}(i), \text{rank}_A^{-1}(j)\}$** In selecting a number from $\{1, 2, \ldots, n\}$ uniformly at random, we happen to select an element as pivot from A which is either the ith smallest element in A or the jth smallest element in A. In this case, $X_{ij} = 1$ by definition.
- **Case 2: $\text{rank}_A^{-1}(i) < p < \text{rank}_A^{-1}(j)$**: The value of the pivot element selected lies between the ith smallest element and the jth smallest element. Since $i < j$, element `i` placed in L and `j` placed in R and $X_{ij} = 0$, since the elements will never be compared.
- **Case 3: $p < \text{rank}_A^{-1}(i)$ or $p > \text{rank}_A^{-1}(j)$**: The value of the pivot is either less than the ith smallest element in A or greater than the jth smallest, in which case either both elements placed into R or both placed into L respectively. Hence X_{ij} may still be "flipped on" in another recursive call to `Quicksort`.

It's possible that on any given round of our algorithm, we end up falling into case 3. Ignore this for now. In doing so, we implicitly condition on falling into case 1 or 2, i.e. we condition on our rank p being chosen so that it lies in the set of values $\{\text{rank}_A^{-1}(i), \text{rank}_A^{-1}(i+1), \ldots, \text{rank}_A^{-1}(j)\}$. Then, the probability that $X_{ij} = 1$ (ignoring case 3) is exactly

$$2/(j - i + 1),$$

Thus,

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1} = \sum_{i=1}^{n-1} 2\sum_{j=i+1}^{n}\frac{1}{j-i+1} \\ &= \sum_{i=1}^{n-1} 2\left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-i+1}\right).\end{aligned}$$

Now, note that $H_n = \sum_{i=1}^{n}\frac{1}{i}$ is the Harmonic Number. We note that $\sum_{i=1}^{n}\frac{1}{i} < \int_1^n \frac{1}{x}\mathrm{d}x$, from Calculus,[6] and further that $\int_1^n \frac{1}{x}\mathrm{d}x = \ln n$. Hence we see that

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} 2\sum_{j=2}^{n-i+1}\frac{1}{j} < 2nH_n = O(n \log n).$$

[6]This can easily be seen by comparing a Lower Darboux Sum with a Riemann Integral.

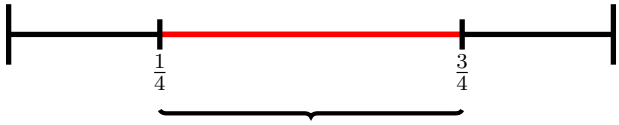
So, in expectation, the number of comparisons (i.e. the work performed by our algorithm) is $O(n \log n)$.

6.3 Total Expected Depth

Recall our analysis in Lecture 4 for `QuickSelect`. In a similar fashion, we may consider the number of *recursive calls* made to our algorithm when our input array is of size

$$\left(\frac{3}{4}\right)^{k+1} n < |A| < \left(\frac{3}{4}\right)^{k} n,$$

and denote this quantity by X_k. We saw that if we select a pivot in the right way, we can reduce input size by 3/4.



If we select any of these elements in red as a pivot, where we visualize the elements being in sorted order, *then* both L and R will have size $\leq \frac{3}{4}n$.

Figure 1: Ideal Pivot Selection

We saw that $\mathbb{E}[X_k] = 2$, since it's a geometric random variable (i.e., we continually make calls to `QuickSort` until we successfully reduce our input size by at least 3/4. In expectation, this takes two trials. Hence in expectation we require $2c \log_{4/3} n$ recursive calls to be made before we reach a base case.

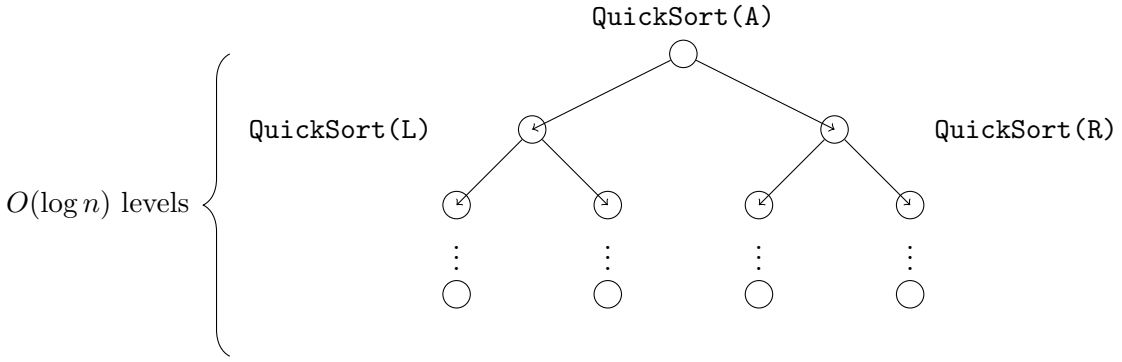


Figure 2: Computational DAG for `QuickSort`

Here, we see that in our computational DAG, we have $O(\log_{4/3} n)$ levels. At each level, our algorithm requires $O(\log n)$ depth since the bottleneck is in constructing L and R and specifically

in the call to Prefix Sum. Hence in expectation our total depth given by the expected number of levels times the expected depth per level: $\mathbb{E}[D(n)] = O(\log^2 n)$.

6.4 A Shortcut for Bounding Total Expected Work

Examine figure 6.3, the computational DAG for our QuickSort algorithm. We are initially handed an input array of size n elements. At each level of our DAG, the n elements are split into chunks across nodes in that level. Specifically, we're not sure exactly how many elements from A are allocated to L, nor do we know exactly how many elements from A are allocated to R. But we definitely know that there are always n elements we deal with on each level. The most work-intensive operation we perform in each call to QuickSort is in constructing L and R, which requires $\Theta(n)$ work. Hence, for a given level in our computational DAG, we perform exactly $\Theta(n)$ work total after summing across work in all nodes.

So, in this *very specialized analysis*, we have that expected work is

$$\mathbb{E}[W] = (\#\text{ nodes per level}) \times (\#\text{ levels}) = O(n \log n).$$

Since there are $O(\log n)$ levels in our tree in expectation, we perform $O(n \log n)$ work in expectation.

7 Matrix multiplication: Strassen's algorithm

We've all learned the naive way to perform matrix multiplies in $O(n^3)$ time.[7] In today's lecture, we review Strassen's sequential algorithm for matrix multiplication which requires $O(n^{\log_2 7}) = O(n^{2.81})$ operations; the algorithm is amenable to parallelization.[4]

A variant of Strassen's sequential algorithm was developed by Coppersmith and Winograd, they achieved a run time of $O(n^{2.375})$.[3] The current best algorithm for matrix multiplication $O(n^{2.373})$ was developed by Stanford's own Virginia Williams[5].

7.1 Idea - Block Matrix Multiplication

The idea behind Strassen's algorithm is in the formulation of matrix multiplication as a recursive problem. We first cover a variant of the naive algorithm, formulated in terms of block matrices, and then parallelize it. Assume $A, B \in \mathbb{R}^{n \times n}$ and $C = AB$, where n is a power of two.[8]

We write A and B as block matrices,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

[7] Refresher, to compute $C = AB$, we need to compute c_{ij}, of which there are n^2 entries. Each one may be computed via $c_{ij} = \langle a_i^T, b_j \rangle$ in $2n - 1 = \Theta(n)$ operations. Hence total work is $O(n^3)$.

[8] If n is *not* a power of two, then from a theoretical perspective we may simply pad the matrix with additional zeros. From a practical perspective, we would simply use un-equal size blocks.

where block matrices A_{ij} are of size $n/2 \times n/2$ (same with respect to block entries of B and C). Trivially, we may apply the definition of block-matrix multiplication to write down a formula for the block-entries of C, i.e.

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

7.2 Parallelizing the Algorithm

Realize that A_{ij} and $B_{k\ell}$ are smaller matrices, hence we have broken down our initial problem of multiplying two $n \times n$ matrices into a problem requiring 8 matrix multiplies between matrices of size $n/2 \times n/2$, as well as a total of 4 matrix additions. There is nothing fundamentally different between the matrix multiplies that we need to compute at this level relative to our original problem.

Further, realize that the four block entries of C may be computed independently from one another, hence we may come up with the following recurrence for *work:*

$$W(n) = 8W(n/2) + O(n^2)$$

By the Master Theorem,[9] $W(n) = O(n^{\log_2 8}) = O(n^3)$. So we have *not* made any progress (other than making our algorithm parallel). We already saw in lecture two that we can naively parallelize matrix-multiplies very simply to yield $O(n^3)$ work and $O(\log n)$ depth.

7.3 Strassen's Algorithm

We now turn toward Strassen's algorithm, such that we will be able to reduce the number of sub-calls to matrix-multiplies to 7, using just a bit of algebra. In this way, we bring the work down to $O(n^{\log_2 7})$.

How do we do this? We use the following factoring scheme. We write down C_{ij}'s in terms of block matrices M_k's. Each M_k may be calculated simply from products and sums of sub-blocks of A and B. That is, we let

[9] Case 1: $f(n) = O(n^2)$, so $c = 2 < 3 = \log_2(8)$.

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22})\,(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

Crucially, each of the above factors can be evaluated using exactly *one* matrix multiplication. And yet, since each of the M_k's expands by the distributive property of matrix multiplication, they capture additional information. Also important, is that these matrices M_k may be computed independently of one another, i.e. this is where the parallelization of our algorithm occurs.

It can be verified that

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M3 + M_6
\end{aligned}
$$

Realize that our algorithm requires quite a few summations, however, this number is a constant independent of the size of our matrix multiples. Hence, the work is given by a recurrence of the form

$$W(n) = 7W(n/2) + O(n^2) \implies W(n) = O(n^{\log_2 7}).$$

What about the depth of this algorithm? Since all of our recursive matrix-multiplies may be computed in parallel, and since we can add matrices together in unit depth,[10] we see that depth is given by

$$D(n) = D(n/2) + O(1) \implies D(n) = O(\log n)$$

By Brent's theorem, $T_p \leq \frac{n^{2.81}}{p} + O(\log n)$. In the years since Strassen published his paper, people have been playing this game to bring down the work required marginally, but nobody has come up with a fundamentally different approach.

[10] We note that to perform matrix addition of two $n \times n$ matrices $X + Y = W$, we may calculate each of the n^2 entries $W_{ij} = X_{ij} + Y_{ij}$ in parallel using n^2 processors. Each entry requires only one fundamental unit of computation, hence the work for matrix addition is $O(n^2)$ and the depth is $O(1)$.

7.4 Drawbacks of Divide and Conquer

We now discuss some bottleneck's of Strassen's algorithm (and Divide and Conquer algorithms in general).

- We haven't considered communication bottlenecks; in real life communication is expensive.
- Disk/RAM differences are a bottleneck for recursive algorithms, and
- PRAM assumes perfect scheduling.

Caveat - Big $\mathcal{O}$ and Big Constants One last caveat specific to Strassen's Algorithm is that in practice, the $O(n^2)$ term requires $20 \cdot n^2$ operations, which is quite a large constant to hide. If our data is large enough that it must be distributed across machines in order to store it all, then really we can often only afford to pass through the entire data set one time. If each matrix-multiply requires twenty passes through the data, we're in big trouble. Big $\mathcal{O}$ notation is great to get you started, and tells us to throw away egregiously inefficient algorithms. But once we get down to comparing two reasonable algorithms, we often have to look at the algorithms more closely.

When is Strassen's worth it? If we're actually in the PRAM model, i.e. we have a shared memory cluster, then Strassen's algorithm tends to be advantageous only if $n \geq 1,000$, *assuming no communication costs*. Higher communication costs drive up the n at which Strassen's becomes useful very quickly. Even at $n = 1,000$, naive matrix-multiply requires $1e9$ operations; we can't really do much more than this with a single processor. Strassen's is mainly interesting as a theoretical idea. For more on Strassen in distributed models, see [2].

Disk Vs. RAM Trade-off What is the reason that we can only pass through our data once? There is a big trade-off between having data in ram and having it on disk. If we have tons of data, our data is stored on disk. We also have an additional constraint that with respect to *streaming data*, as the data are coming in they are being stored in memory, i.e. we have fast random access, but once we store the data to disk retrieving it again is expensive.

References

[1] *Probabilistic analysis and randomized quicksort*, Carnegie Mellon, 15451-s07, (2007).

[2] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, *Communication-optimal parallel algorithm for strassen's matrix multiplication*, CoRR, abs/1202.3173 (2012).

[3] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, J. Symbolic Computation, 9 (1990), pp. 251–280.

[4] V. Strassen, *Gaussian elimination is not optimal*, Numerische Mathematik, 13, pp. 354–356.

[5] V. V. Williams, *Multiplying matrices in $o(n^{2.373})$ time*, Stanford University, (2014).