CME 323: Distributed Algorithms and Optimization, Spring 2015
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matriod and Stanford.

Lecture 4, 4/6/2016. Scribed by Henry Neeb, Christopher Kurrus, Andreas Santucci.

Overview

Today we will continue covering divide and conquer algorithms. We will generalize divide and conquer algorithms and write down a general recipe for it. What's nice about these algorithms is that they are *timeless*; regardless of whether Spark or any other distributed platform ends up winning out in the next decade, these algorithms always provide a theoretical foundation for which we can build on. It's well worth our understanding.

- Parallel merge sort
- General recipe for divide and conquer algorithms
- Parallel selection
- Parallel quick sort (introduction only)

Parallel selection involves scanning an array for the kth largest element in linear time. We then take the core idea used in that algorithm and apply it to quick-sort.

Parallel Merge Sort

Recall the merge sort from the prior lecture. This algorithm sorts a list recursively by dividing the list into smaller pieces, sorting the smaller pieces during reassembly of the list. The algorithm is as follows:

Algorithm 1: MergeSort(A)

Input : Array A of length n
Output: Sorted A

```
if n is 1 then
    return A
end
else
    L ← mergeSort(A[0, ..., n/2))
    R ← mergeSort(A[n/2, ..., n])
    return Merge(L, R)
end
```

Last lecture, we described one way where we can take our traditional `merge` operation and translate it into a `parallelMerge` routine with work $O(n \log n)$ and depth $O(\log n)$. Recall that to do this, we assign each of n processors a single element in one sorted array A and perform a binary search on the other array B for where the index would belong; the rank of the element in the output array is simply the sum $\text{rank}_A(x) + \text{rank}_B(x)$. Unfortunately, this leads our MergeSort routine to require work $W(n) = O(n \log^2 n)$, albeit at a better depth of $D(n) = O(\log^2 n)$. This is bothersome since by Brent's theorem, we still have T_1 showing up in the upper bound, hence we won't want to increase work for our algorithms in general.

We also started to describe a way in which if we have two sorted arrays in memory, and the largest element of one array is less than the smallest element of the other, then we may simply append the arrays together to yield a sorted output. This led to Richard Cole's 1988 paper, which achieves $W(n) = O(n \log n)$ and $D(n) = O(\log n)$.[2] The way Cole was able to do this was by modifying his routine to return *more* information than just a sort alone. So, some of the work of the `Merge` routine was being performed in the `mergeSort` routine itself.

Parallel Algorithms and (Seemingly) Trivial Operations

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; if we wish to get an array of zeros, that requires $\Theta(n)$ work since we must ensure the integrity of each entry.

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.

General Divide and Conquer Technique

Many divide and conquer algorithms are actually quite formulaic. We go over the general method.

1. Come up with recursive formulation of the problem.
2. Count the following:

- Let a denote the number of recursive calls the function makes.
- Let b denote the integer such that the recursive calls accept input of size n/b.

3. Determine the amount of work $T_1 = w$ and depth $T_\infty = t$ is required by procedures *outside of* any recursive calls; i.e. what is required of our *combine* step?

4. Write down the recurrence relations for the whole algorithm, which may be solved using the Master Theorem.[1]

$$T_1 = W(n) = aW\left(\frac{n}{b}\right) + w$$
$$T_\infty = D(n) = D\left(\frac{n}{b}\right) + t$$

   Part of the beauty in the divide and conquer method is that with respect to depth, it doesn't matter how many recursive calls we end up making. Since the recursive calls are independent, when we add them to our computational DAG, we do not incur more than one unit of depth.

Generally speaking, coming up with the recursive formulation takes the most creativity. The rest of the steps are more formulaic. It's not trivial to realize that to sort a bunch of numbers, you need to sort the left hand side and the right hand side and then merge the results back together. Many parallel algorithms, such as finding the convex hull of a bunch of points in 2D, are best solved using divide and conquer. Realizing that you even should be using divide and conquer takes a little bit of creativity.

As far as this class is concerned the hardest part is coming up with a combine step which is "cheap", i.e. does not require much work/depth. We're going to practice this creativity as we think about the next problem: parallel quick selection.

Parallel Selection

Suppose we have a list of unsorted integers, which we know nothing about. We wish to find the k^{th} largest element of the integers. We can use *quick selection* to accomplish this task in linear time without having to sort the entire list.[1] Manuel Blum proved this result in 1971 and was awarded a Turing Award for it. Incidentally, he was actually trying to prove that you need to sort to be able to do arbitrary selection. Quick selection is a randomized algorithm, so all run times presented will be on an expected basis.

We will solve this algorithm using Divide and Conquer. This will be the first randomized algorithm (and hence run time is expected) that we have seen in this class. We assume that our input array A has unique elements.[2]

[1] It's also worth mentioning that unrolling recurrences by hand is not that difficult. If you ever forget which case of the Master Theorem to use, you should be able to figure it out from scratch without much trouble.

[2] If the elements are not unique, then we can simply omit the duplicate elements from the recursive calls.

3.1 Idea

We summarize the algorithm in words. From the input list, A, pick a value at random called a pivot, p. For each item in A, put item into one of two sublists L and R such that:

$$x \in L \iff x < p$$
$$y \in R \iff y > p$$

We do not require L and R to be sorted. We can do this in linear time since it only requires passing through each element in A and performing a constant time check whether the element is greater or less than our pivot p.

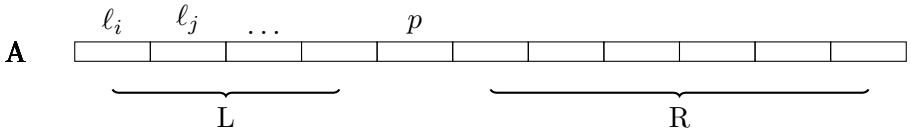

Figure 1: After re-arranging, all elements in L (such as ℓ_i, ℓ_j) are less than the pivot p. All elements in R are greater than the pivot p.

Crucial Realization Notice that the rank of p is *exactly* $|L|$. To find the k^{th} largest element in A, call it z, note the following:

$$\text{If } |L| < k, \text{then } z \notin L. \text{ We discard the values in L.}$$
$$\text{If } |L| > k, \text{then } z \notin R. \text{ We discard the values in R.}$$

We elaborate on why if $|L| < k$, then $z \notin L$. This is because for any element $l \in L$, its sorted order must be less than p's order. But p's order is equal to $|L| + 1$, so $z \in L$ would imply $k < |L|$, which is a contradiction. By similar logic we can argue if $|L| > k$, then $z \notin R$.

We then recursively perform the selection algorithm on the not discarded list, which reduces our problem size. Specifically, we prove that with probability 1/2, we reduce our problem size by at least 3/4. That will then be enough to guarantee expected linear time work.

Intuition for Analysis Since `Select` makes a chain of recursive calls, let us group multiple calls together into one "phase" of the algorithm. The sum of the work done by all calls is then equal to the sum of the work done by all phases of the algorithm. Concretely, let us define one "phase" of the algorithm to be when the algorithm decreases the size of the input array to 3/4 of the original size or less.

Why 3/4? Notice that if we can shrink the size of the array by a constant factor each iteration while only performing linear work per iteration, then using the Master Theorem we know that total work done would be linear. Further, picking any pivot in the "middle" half of the array means that after re-arranging elements into L and R, at least 1/4 of the array values are less than the pivot and at least 1/4 of the array values are larger than the pivot.

Algorithm 2: Select(A, k)

Input : Array A with n entries, integer $k \leq n$
Output: Value of the kth largest element in A

```
p ← a value selected uniformly at random from A     // This is our pivot.
L ← elements in A which are less than pivot p       // Requires Θ(n) work
R ← elements in A which are larger than pivot p
if |L| is k − 1 then
    return p                          // The pivot itself is the kth largest element
end
if |L| > k then
    return Select(L, k)               // kth largest element lives in L
end
else
    return Select(R, k-|L|-1)         // kth largest element lives in R
end
```

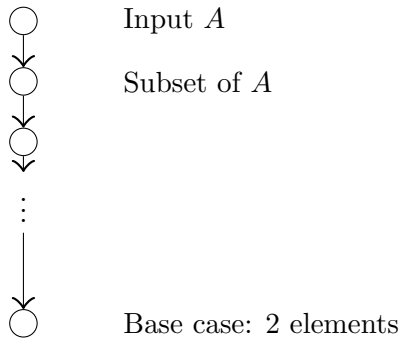

Figure 2: Observe the computational DAG for our `QuickSelect` algorithm, which is serial in nature.

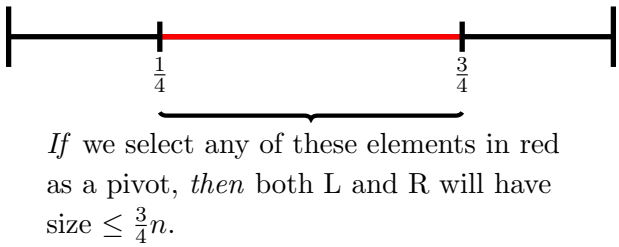

Figure 3: Figure illustrates the region of the data where picking a pivot will result in both L and R having sizes less than $\frac{3}{4}n$.

So, we say that a *phase* in our algorithm ends as soon as we pick a pivot in the middle half of

our array, denoted by the red portion in the figure above. Recognize that in phase k, the array size is at most $n\left(\frac{3}{4}\right)^k$. The maximum number of *phases* before we can hit a base case is given by $\lceil \log_{4/3} n \rceil$, since in each phase we dampen the size of the data by $\frac{3}{4}$.

3.2 Analysis - Expected Work

The analysis of this algorithm is as follows. Let us define the number of calls made to `Select` when the size of the input array is of a particular size. That is, let X_k denote the number of times `Select` called with array of input size between

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

Total work is given by the sum of the work done for each X_k multiplied by the number of calls of that size. Realize that total work done during phase k given by

$$X_k \cdot cn\left(\frac{3}{4}\right)^k$$

for some constant $c \in \mathbb{R}^+$. Since in each phase, we reduce the input size by at least $3/4$, total number of phases given by $\lceil \log_{4/3} n \rceil$. Let W be a random variable describing the total work done. Then,

$$W \leq \sum_{k=0}^{\lceil \log_{4/3} n \rceil} \left(X_k \cdot cn\left(\frac{3}{4}\right)^k \right) = cn \sum_{k=0}^{\lceil \log_{4/3} n \rceil} \left(X_k \left(\frac{3}{4}\right)^k \right).$$

We're interested in the expected amount of total work. Hence, we apply the expectation operator to both sides (expectation is a monotone operator), and after applying linearity of expectation we see that,

$$\mathbb{E}[W] \leq cn \sum_{k=0}^{\lceil \log_{4/3} n \rceil} \mathbb{E}[X_k] \left(\frac{3}{4}\right)^k.$$

We now analyze $\mathbb{E}[X_k]$. Recall that

$$\mathbb{E}[X_k] = \sum_{i=0}^{\infty} i \cdot \Pr(X_k = i).$$

Claim: X_i is a Geometric Random Variable with parameter $p = \frac{1}{2}$, hence $\mathbb{E}[X_i] = 2$.

Recall that we defined a phase as lasting until we reduce the input size by at least $3/4$, Consider our choice of pivot - if the selected pivot is between the 1^{st} and 3^{rd} quartiles of the data, then both L and R will have size $\leq \frac{3}{4}n$. The probability of this occurrence is then $\frac{1}{2}$, so X_i must be a geometric with parameter $\frac{1}{2}$. Put another way, since all pivot choices are independent, $\Pr(X_k = i)$ equals the probability that the first $i-1$ pivot choices were in the outer quartiles *times* the probability that the ith pivot choice was in interquartile range, i.e.

$$\Pr(X_k = i) = \left(\frac{1}{2}\right)^{i-1} \cdot \frac{1}{2} = \frac{1}{2^i}.$$

Hence,

$$\mathbb{E}[X_k] = \sum_{i=0}^{\infty} i \Pr(X_k = i) = \sum_{i=0}^{\infty} \frac{i}{2^i} = 2.$$

That is, in expectation, we need to make two calls to `Select` before reducing our input size by at least 3/4. Ultimately, we see that

$$\mathbb{E}[W] \leq cn \sum_{k=0}^{\lceil \log_{4/3} n \rceil} \mathbb{E}[X_k] \left(\frac{3}{4}\right)^k \leq 2cn \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k = 2cn \cdot \frac{1}{1 - 3/4} = 8cn = O(n).$$

where we've used the fact that $\sum_k \left(\frac{3}{4}\right)^k$ is a geometric series with constant value (no dependence on n). We have derived expected total work to be $O(n)$. This is in the sequential world. We note that because our recursive calls are stochastic in this algorithm, we cannot use the Master Theorem.

So, by this analysis, in expectation,

$$T_1 = O(n).$$

Applying Markov's Inequality We may analyze our total work further by utilizing the Markov Inequality. We can compute a bound on the probability that our total work exceeds a multiple of our expected total work. For example, if we wanted to do analysis that our total work will exceed 5 times our expected work:

$$P(\text{Total Work} \geq 5 \times E[\text{Total Work}]) \leq \frac{E[\text{Total Work}]}{5 \times E[\text{Total Work}]} = \frac{1}{5}$$

This lets us say that we will only have do perform $5n$ work with probability at most 1/5. Call this probability $p = 1/5$. We may run this algorithm as many times as we want. We can run our algorithm $5c2n$ times. Each time, we have at probability at least 4/5 of finding our kth element in linear time. The total amount of work required is the sum over however many times we have to restart the process (potentially an infinite number of times) *times* the probability we have to restart that many times. Each time we fail, we do $5cn = O(n)$ work. Hence

$$\sum_{r=1}^{\infty} \left(\frac{4}{5}\right)^r O(n) = O(n) \cdot c = O(n).$$

That is, we can try an infinitude of times and yet still only expect to perform constant work.

3.3 Parallelizing our Select Algorithm

Note that there is only one recursive call to **Select**, so the recursive portion of this function is not divide and conquer in nature. How can we make this algorithm run in parallel? It's clear that each step of the algorithm runs in constant time except for creating L and R, hence this is the step we will focus on.

How to Construct L and R with Small Depth? We seek to construct L and R in a smart way. Naively, you may think that constructing an indicator list of which elements in A go into L and R is enough, but scanning and retrieving a solution from this construction will require $O(n)$ work. We can actually construct these lists using **PrefixSum**, which only takes $O(\log n)$ depth and $O(n)$ work still. We consider the following methodology to keep our depth low when constructing L and R (notice that constructing R follows the same steps as constructing L):

Algorithm 3: Constructing L (or R)

1 Allocate and empty array of size n, with all values initially 0.
2 Construct indicator list $B_L[0, ..., n-1]$, where $b_i = \mathbf{1}\{a_i < p\}$
3 Compute **PrefixSum** on B_L `// This requires` $O(\log n)$ `depth`
4 Create the array L of size $\textbf{PrefixSum}(B_L)[n-1]$
5 **for** $i = 1, 2, \ldots, n$ **do**
6 $\quad L\big[\textbf{PrefixSum}(B_L[i])\big] \leftarrow a[i]$ `// Can be done in parallel`
7 **end**

In the first and fourth steps, we request a chunk of $O(n)$ memory in constant time, i.e. we perform $O(n)$ work with unit depth. Also part of this first step, we use n processors and assign each to initialize one element of the array to zero. In our second step, we construct our indicator list B_L which indicates whether an entry belongs in L, i.e. whether the entry less than p. This requires $O(n)$ work but only $O(1)$ depth. Notice that only after completing step 3 do we know how much space to allocate for L (or R), since the value of the last entry in our prefix sum array tells us how many elements belong to L. This third step requires $O(n)$ work and $O(\log n)$ depth. Lastly, we fill in the entries of array L with $O(n)$ work and $O(1)$ depth.

Hence, our algorithm to construct L or R requires $O(n)$ work and $O(\log n)$ depth. So, if we use this (parallel) sub-routine within our `Select` algorithm, the recurrence for depth becomes

$$D(n) = D\left(\frac{n}{4/3}\right) + O(\log n) \implies D(n) = O(\log^2 n).$$

We remark that since our new method for constructing L and R did not require any additional work, our recurrence for work remains the same.

Example Suppose

$$A = \begin{bmatrix} 1 & 16 & 3 & 2 & 5 \end{bmatrix}$$

We randomly select a pivot, suppose we select $p \leftarrow 5$. This takes $O(1)$ work. We assign each element to a single processor, and if the element is strictly smaller than the pivot, we overwrite its corresponding bit in B_L with a 1,

$$B_L = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

This takes $\Theta(n)$ work, but since we perform operations in parallel it requires $O(1)$ depth. Now, we compute the prefix-sum on B_L, which gives us

$$\begin{bmatrix} 1 & 1 & 2 & 3 & 3 \end{bmatrix}$$

i.e. we now know that since B_L has bits turned on in the first, third, and fourth positions, and since the first, third, and fourth elements in the resulting prefix-sum output have values 1, 2, 3, this tells us where to place each element in L. Computing prefix-sum takes $O(n)$ work and $O(\log n)$ depth.

Parallel Quick Sort

Parallel quick sort is very similar to parallel selection. We randomly pick a pivot and sort the input list into two sublists based on their order relationship with the pivot. However, instead of discarding one sublist, we continue this process recursively until the entire list has been sorted. The pseudocode for this algorithm is as follows:

Algorithm 4: Quicksort(A)

Input : A
Output: sorted A

Let p be random uniformly selected from A
L = Elements of $A < p$
R = Elements of $A > p$
return [**QuickSort**(L), p, **QuickSort**(R)]

The analysis for this algorithm is performed in the next lecture.

References

[1] M. Blum, *Linear time bounds for median computations*, (1971).

[2] R. Cole, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.