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

Lecture 1, 9/26/2015. Scribed by Suhas Suresha, Pin Pin, Andreas Santucci.

1 Caveats of Parallel Algorithms

Communication Cost Underlying *Brent's theorem* is the assumption that processors can communicate with each other instantaneously. That is, given an algorithm, we can use the work-depth model to estimate the computation time on one processor, T_1 the work, as well as for p processors, T_p, and an infinitude of processors, T_∞ the depth. We saw that

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty.$$

This is *not* applicable for distributed clusters, since in this scenario there is an inherent cost to communicating over a network with limited bandwidth.

Lower Bounds In the sequential world, it's possible to get nice lower bounds on the depth of the computational DAG. For example, reading an input of size n costs $\Omega(n)$ operations. In the parallel world, we no longer require that one processor perform $O(n)$ work. Instead, each processor may be assigned a constant amount of work in parallel, i.e. we could process $O(n)$ input in $O(1)$ time; the same can be said for output.[1] Hence, in parallel models, we defenestrate lower bounds we used to consider fundamental.

Given an input or output of size n,

$$\text{one processor must be assigned at least } n/p \text{ work:} \qquad T_1 \geq n \implies T_p \geq \frac{n}{p},$$

$$\text{at best, work evenly distributed between } p \text{ processors:} \qquad T_p \geq \frac{T_1}{p}.$$

However, we emphasize that $T_1 \geq n \not\Longrightarrow T_p \geq n$ for $p \geq 1$, as our examples above indicate. As a first example for some of these phenomena, we turn towards matrix multiplies.

2 Matrix Multiplies

We are given as input two $n \times n$ matrices, call them A and B. We wish to output 1 $n \times n$ matrix which represents the product $A^T B = C$.

[1] For example, some algorithms might have an input size of $O(n)$ and output size of $O(n^2)$. In sequential computing, it's trivial to say that the algorithm requires at *least* $\Omega(n^2)$ operations, since at the very least, if nothing else, the algorithm must write $\Omega(n^2)$ elements to output.

Sequential Algorithm By definition,

$$c_{ij} = \langle A_i, B_j \rangle = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Algorithm 1: Naive Matrix Multiply

Input : Two $n \times n$ matrices A, B
Output: $C = A^T B$

```
for i = 1, 2, ..., n do
    for j = 1, 2, ..., n do
        c_ij ← ⟨a_i, b_j⟩
    end
end
return C
```

There are exactly n^2 entries in C which we must compute. Each inner product takes $O(n)$ work since we have n multiplications and $n-1$ additions. Thus work is $O(n^3)$, i.e.

$$T_1 = O(n^3).$$

Parallel Algorithm How can we parallelize these computations? Trivially, we may compute each c_{ij} independent of one another. Set $p = n^2$, i.e. we have n^2 processors. Then we may execute the `for` loops in our naive matrix multiply in parallel.

Computing each c_{ij} still requires $O(n)$ work, and so $T_1 = O(n^3)$. But notice that the depth of our computation DAG is $O(n)$. Specifically, each c_{ij} still computed sequentially, i.e. in our inner product notation $\langle a_i, b_j \rangle$ there is a hidden `for` loop which is *not* parallel. Hence $T_\infty = O(n)$.

Improved Parallel Algorithm Recall that in the work-depth model, we have an infinitude of processors available. In the last lecture, we learned how we may parallelize a summation operation (or in general, any associative binary operator) such that it requires $O(\log n)$ depth. Thus if we parallelize each c_{ij} summation itself, we can bring depth down to $T_\infty = O(\log n)$. Notice that total work remains the same.

By Brent's theorem, we know that $T_P \leq \frac{n^3}{P} + \log(n)$. We note that Strassen's *sequential algorithm* for matrix multiplication has $T_1 = O(n^{2.81})$. We will revisit this in the next lecture.

3 The Master Theorem

The Master Theorem provides a nice solution for solving simple recurrences.

Unrolling the Simplest Case Let's consider a trivial case. Suppose the recurrence relation is of the form:

$$T(n) = b\,T\left(\frac{n}{b}\right) + n$$

where b some integer greater than 1. This corresponds to an algorithm which calls itself b times; when it calls itself, it calls with an input of size n/b; in doing the initial call, it additionally performs $O(n)$ work.

We now unroll this recursion (by hand) to see its solution.

$$\begin{aligned}
T(n) &= b\,T\left(\frac{n}{b}\right) + n && \text{by our definition of } T(n) \\
&= b\left[bT\left(\frac{n}{b^2}\right) + \frac{n}{b}\right] + n && \text{unrolling to the second-level of our tree} \\
&= b^2 T\left(\frac{n}{b^2}\right) + b\cdot\frac{n}{b} + n && \text{re-arranging} \\
&= b^2\left[b\,T\left(\frac{n}{b^3}\right) + \frac{n}{b^2}\right] + b\cdot\frac{n}{b} + n && \text{unrolling to the third level of our tree} \\
&= b^3 T\left(\frac{n}{b^3}\right) + b^2\cdot\frac{n}{b^2} + b\cdot\frac{n}{b} + n && \text{re-arranging} \\
&= \ldots \\
&= n\log_b(n)
\end{aligned}$$

In arriving to the last equality, we need to know three things. First, most obviously, we need to have a base-case. Second, we need to know what is the depth of the tree resulting from the recursion? Clearly, by definition of logarithms, the depth is $\log_b n$, since each time we divide our input by size b. Lastly, we need to understand how much work is performed. Notice that in our initial call, we perform $O(n)$ work. At the second level of our tree, we have b children who each perform n/b work in addition to the $O(n)$ work of the parent (root) node. Hence for each level k of our tree we perform $k\cdot n$ work in total (inclusive of work done by parent nodes in the computational DAG). Therefore, the above recursive algorithm has a run-time of $O(n\log_b(n))$.

Had the recurrence been in a slightly different form, we may have gotten a slightly different result. The Master Theorem simply involves checking the form of a recurrence of interest against three cases to determine the asymptotic solution.

3.1 General form of the Master theorem

The general form of the Master theorem is as follows. If the recurrence relation is of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

where a is the number of sub-problems in the recursion, n/b is the size of each sub-problem and $f(n)$ is the cost of the work done outside the recursive calls.

Case 1 If $f(n) \in O(n^c)$ where $c < \log_b a$, then

$$T(n) \in \Theta(n^{\log_b a}).$$

Case 2 If it's true that, for some constant $k \geq 0$, $f(n) \in \Theta(n^c \log^k n)$ where $c = \log_b a$, then,

$$T(n) \in \Theta(n^c \log^{k+1} n).$$

Case 3 If $f(n) \in \Omega(n^c)$ where $c > \log_b a$, and if its *also* true that

$$af\left(\frac{n}{b}\right) \leq kf(n)$$

for some constant $k < 1$, and sufficiently large n, then

$$T(n) \in \Theta\left(f(n)\right)$$

Examples

Let us look at some examples now.

- $T(n) = T(n/b) + 1$. Here, $\log_b 1 = 0$ and $f(n) \in \Theta(n^0 \log^0 n)$, hence case 3. That is, $T(n) = \log_b(n)$. There are $\log_b n$ recursive calls made, each one involving unit work, hence total work is $1 \times \log_b n$.
- $T(n) = T(n/b) + n$. Here, $\log_b 1 = 0$ and $f(n) \in \Theta(n^1)$, i.e. Case 3. We check our regularity condition, using any $k \in (1/b, 1)$; we then satisfy $n/b \leq kn$.[2] Here, notice that with each recursive call, the work done outside the recursive call is being dampened by a constant factor b. Unrolling our recurrence, we realize a geometric series which converges to a real number $\alpha \in \mathbb{R}$; i.e. the total work is αn, hence $T(n) = \Theta(n)$.

These examples illustrate that we can get drastically different solutions to our recurrences based on which constants appear in specific places. Kleinberg and Tardos, CLRS all have fantastic references for this.

4 General recursive relations

For some recursive relations, we *cannot use* the Master theorem. For example, for any algorithm which involves recursive calls to itself where the input size is dampened by anything but a constant. I.e., if as opposed to n/b in our recurrence relation, we have something like $\sqrt{n}$, then this is *not covered* by the Master Theorem.

For such cases, we have to resort to unrolling the recurrence by hand to find a solution.

Example Consider the recurrence relation $T(n) = T(n^{1/2})+1$, with a base case given by $T(2) = 1$. We see that we cannot use the Master theorem for this case, since there is no constant $b > 1$ such that we may describe $T(n) = aT(n/b) + 1$. This leads to our sub-problems becoming smaller at a much faster rate.

So, we consider unrolling our recurrence by hand.

[2] Note that we assume $b > 1$.

$$
\begin{aligned}
T(n) &= T(n^{1/2}) + 1 && \text{our definition of } T(n) \\
&= \left(T(n^{1/4}) + 1\right) + 1 && \text{unrolling to second level} \\
&= \left(\left(T(n^{1/8}) + 1\right) + 1\right) + 1 && \text{unrolling to third level...}
\end{aligned}
$$

The question becomes, how many times we must make recursive calls before reaching our base case. Let k denote the number of recursive calls needed to reach our base case (specified by input size of 2):

$$
\begin{aligned}
n^{1/2^k} = 2 &\implies 1/2^k \log_2(n) = \underbrace{\log_2(2)}_{=1} \\
&\implies 2^k = \log_2(n) \\
&\implies k = \log_2\left(\log_2(n)\right)
\end{aligned}
$$

Hence the run-time of the above recursive algorithm is $O(\log_2 \log_2(n))$, since we end up adding a constant term $\log_2 \log_2 n$ times.[3]

Remark We remark that recursive algorithms are often trivial to parallelize, since the recursive calls naturally don't depend on each other, hence we may assign each to a processor. That is, when we make recursive calls, we often write our (sequential) algorithms in such a way that we don't care which one finishes first, since they all must finish before we may proceed with our algorithm.

5 All Prefix Sum

Given a list of integers, we want to find the sum of all prefixes of the list, i.e. the running sum. We are given an input array A of size n elements long. Our output is of size $n + 1$ elements long, and its first entry is *always* zero. As an example, suppose $A = [3, 5, 3, 1, 6]$, then $R = \texttt{AllPrefixSum}(A) = [0, 3, 8, 11, 12, 18]$.

Algorithm Design In sequential world, this is trivial. How can we parallelize this problem? We need a mix of divide and conquer and our first parallel summation algorithm. We design the

[3] Realistically, in our world, $\log \log n$ is about 5.

following algorithm.

Algorithm 2: Prefix Sum

Input: All prefix sum for an array A

1 **if** *size of A is 1* **then**
2 | **return** *only element of A*
3 **end**
4 Let A' be the sum of adjacent pairs
5 Compute $R' = \text{AllPrefixSum}(A')$ `// Note: R' has every other element of R`
6 Fill in missing entries of R' using another $\frac{n}{2}$ processors

The general idea is that we first take the sums of adjacent pairs of A. So the size of A' is exactly half the size of A. Note that if the size of A not a power of 2, we simply pad it with zeros. Notice that R' has every other element of R, our desired output.

Specifically, every element in R' corresponds to an element with an index of even parity in A. It's as though as we did a running sum, but we only reported the running sum every two iterates. That is, we have an array A and R

$$A = \begin{bmatrix} a_1 & a_2 & a_3 & \dots & a_n \end{bmatrix}$$
$$R' = \begin{bmatrix} r_2 & r_4 & \dots & r_{n/2} \end{bmatrix}$$

That is, $r_2 = a_1 + a_2$, and $r_4 = a_1 + a_2 + a_3 + a_4$, and in general $r_k = \sum_{i=1}^{k} a_i$.

To compute the running sum for elements whose index is of odd parity in A, i.e. set

$$r_i = r_{i-1} + a_i$$

for $i = 1, 3, 5, \dots$, where we by convention let $r_0 = 0$.

Algorithm Analysis Notice that step 4, where we let A' be the sum of adjacent pairs, we must perform $n/2$ summations, hence work is $O(n)$. Realize that we may assign each processor a pair of numbers and perform the summations in parallel. Hence depth is $O(1)$.

Notice that step 6, filling in missing entries, we can assign each of the $n/2$ missing entries of R to a processor and compute its corresponding value in constant time. Hence step 6 has work is $n/2$, i.e. $O(n)$, and depth $O(1)$.

Now let's consider the work and depth for the entire algorithm. This is where recurrences come into play. Let $T_1 = W(n)$, and $T_\infty = D(n)$,

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

The expression for work follows since we make exactly one recursive call of exactly half the size, and in outside our recursive call we perform $O(n)$ work. By the Master Theorem, $W(n) = O(n)$.[4]

[4] Again, unrolling our recurrence we yield a geometric series scaled by n, hence work is $O(n)$.

With regard to depth, again realize that we make a recursive call on input size $n/2$, and outside the recursive call we only require constant depth. Again by the Master Theorem, we see that $D(n) = O(\log n)$.

For prefix-sum, this is pretty much the best we can hope for. We emphasize that recursion was critical for the parallelization of this algorithm.

6 Conclusion

We conclude with the remark that although recursive algorithms are amenable to parallelization, the algorithm designer must do some analytical work to make algorithms efficient in a parallel setting. Often times, we the combine step of a divide-and-conquer algorithm is sequential in nature, and can be the bottleneck of our analysis. To get around this, we must think carefully. We'll see more on this when we talk about MergeSort next lecture.

References

[1] J. Kleinberg, E. Tardos. *Algorithm Design.* Addison-Wesley Publishing Co.

[2] Volker Strassen. *Gaussian Elimination is not Optimal.* Numerische Mathematik, August 1969.

[3] T. Cormen, C. Stein,R. Rivest,C. Leiserson. *Introduction to Algorithms.* The MIT Press.

[4] G. Blelloch and B. Maggs. *Parallel Algorithmw.* Carnegie Mellon University.