CSE 347 Lecture 6

Span analysis of divide-and-conquer

1. Span: motivation and definition
2. Analysis of the span of divide-and-conquer algorithms
3. Parallelizing Merge Sort

Today, we'll see the first of many alternatives to our basic model of algorithms. This will be an idealized model of running time on a parallel computer, in which we are free to run as many computations simultaneously as we wish—think of having a cluster so large that there is always an idle processor at your disposal. The number of steps you have to wait for a parallel algorithm to terminate in this model is the span of the algorithm. (By contrast, the total number of steps over all of these processors, which is essentially the same as the running time of the algorithm if it were run on a single processor in our original model, is known as the work of the algorithm.) Although the span is just an idealization, since in practice we are almost always constrained by the number of cores/processors at our disposal, the

span is still important — if it is large (relative to the work), then this suggests that there is not much point to devoting more cores to a computation. Or, that in order to make use of such resources, we will need a different algorithm

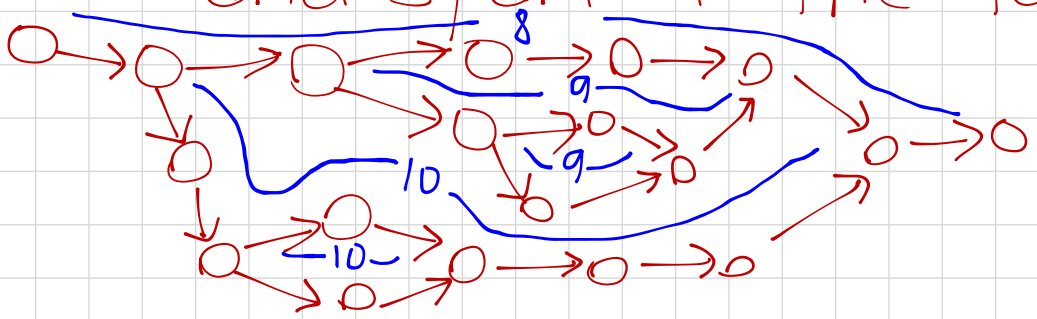
To describe our parallel algorithms, we will use the following terminology: the algorithm may, at any point, spawn a new thread that can run in parallel with the original algorithm. The algorithm may then, at any point, wait for the threads it spawned to terminate (and return a value) — we call this a sync.

Analyzing the span

To compute the span of an algorithm, it is convenient to consider a directed graph in which each node corresponds to a single step, and there is an edge from a step s to a step t if either i) t follows s, ii) t is spawned by s, or iii) t is a sync and s is the final step of a thread spawned by the syncing thread. These are the steps actually taken during the execution of an algorithm on a specific input, so the graph has no cycles. For example, a sequential (single-thread) algorithm corresponds to a chain of length equal to

the algorithm's running time: step 1 → step 2 → ... → step T

The span is now simply the longest directed path, and the work is the total # of nodes. So, what are the work and span of the following example?



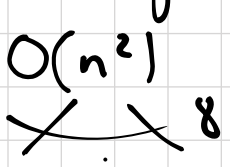
Work: # nodes = 19

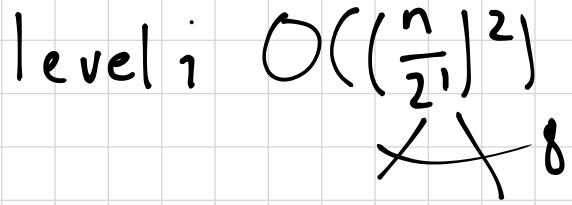
span: longest path = 10

For convenience, we might collapse chains down to a single node labeled by the # of steps it represents.

Divide-and-conquer algorithms can generally be parallelized by spawning the recursive calls, and then syncing before combining the results. There is then a close relationship between the recurrence trees that we draw and the computation DAG.

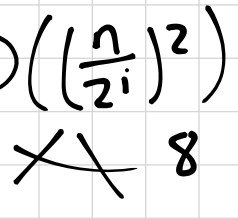
For example, suppose we take our first matrix multiplication divide-and-conquer algorithm. The tree was...

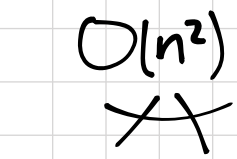


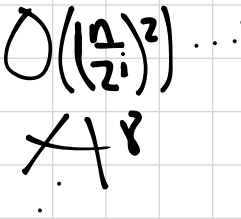


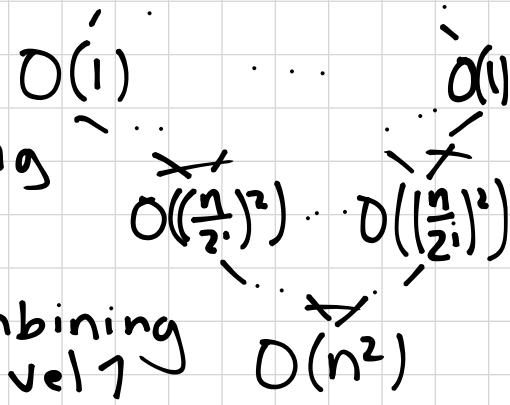
DAG:











span: $\sum_{i=1}^{\log_2 n} C \cdot (\frac{n}{2^i})^2 + \sum_{i=1}^{\log_2 n} C \cdot (\frac{n}{2^i})^2 = O(n^2)$

dividing combining

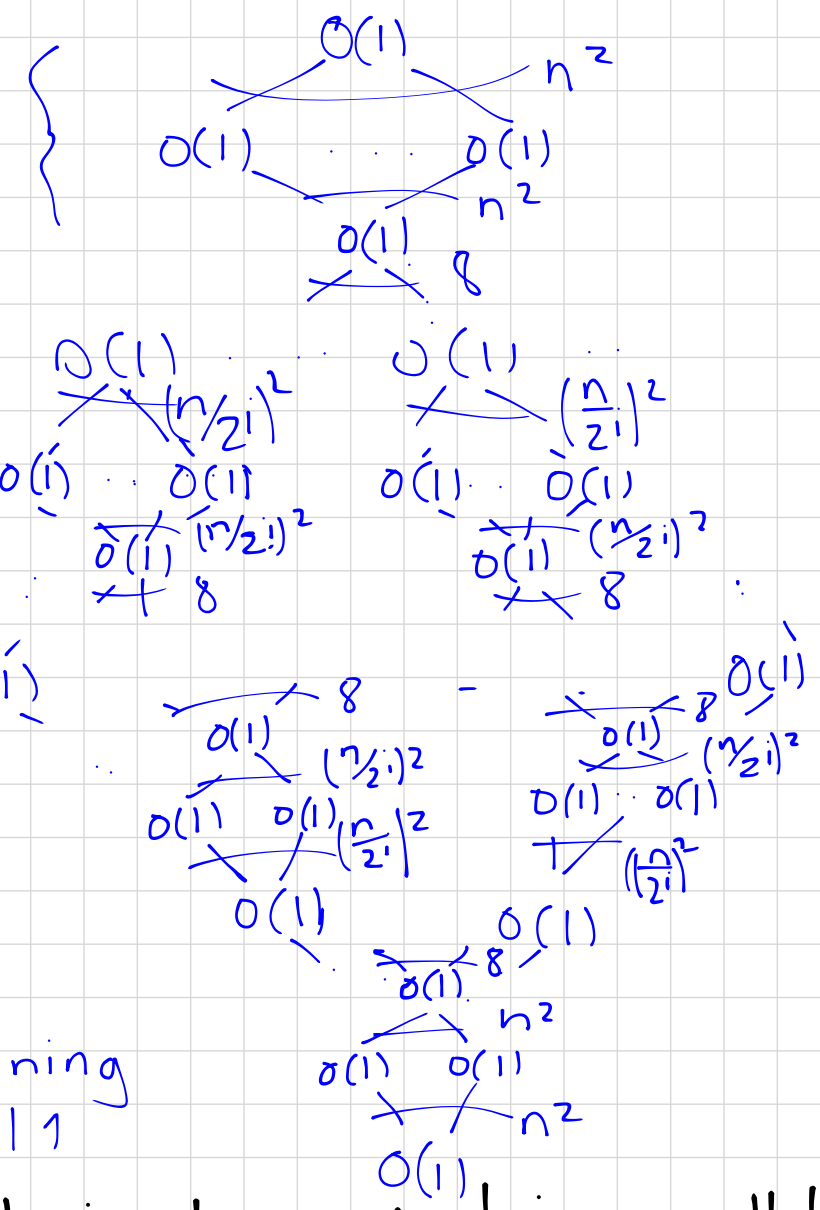
vs. $O(n^3)$ work...

We can do much better, by also computing the four matrix sums in parallel. Suppose we now spawn a separate thread to compute each entry of our matrices before and after the recursive call. Now what is the span?

Each node now has $O(1)$ work. So, when we draw the tree, we find

height $O(\log n)$
$O(1)$ work per node $\Rightarrow$
Span $O(\log n)$

Note: this assumes I can spawn $O(n^2)$ threads in one step. If I can only spawn one thread, we increase the span by another $O(\log n)$ factor.



Strassen's algorithm may be similarly implemented in parallel to obtain a $O(n^{2.81})$ work, $O(\log n)$ span algorithm.

Parallelizing MergeSort

Naively parallelizing MergeSort leads to a similar problem in the combine step. Recall, the algorithm is

If the input list A has length $n=1$, return A.
$B \leftarrow$ MergeSort $(A[1 \cdots n/2])$
$C \leftarrow$ MergeSort $(A[n/2+1 \cdots n])$
Until B & C are both empty

Choose the smaller of the 1st elements of B & C, and add it to the end of D (initially empty)

Return D.

The problem is that the merge step at the end adds $\Theta(n)$ steps to the span. To address this, we'll introduce a divide-and-conquer subroutine for merging. As we'll see, it will be useful to solve a more general problem, in which the lists B & C may have different sizes, m and l; WLOG, suppose C is shorter (if not, swap B & C):

If B has length m=1,

  Return the correct concatenation of the one-element lists.

mid ← m/2

Use binary search to find the index s where B[mid] would appear in C.

Spawn D[1 ··· s+mid] ← Merge B[1 ··· mid] and C[1 ··· s]

D[s+mid+1 ··· m+l] ← Merge B[mid+1 ··· m] and C[s+1 ··· l]

sync

Return D ← so, no further work is needed here.

Note: we have given each call a range of indices in a global array, and merge the lists directly into place

Let's first calculate the work done by this algorithm. Let $k = mid + s$ be the size of the first recursive parallel call. Since $m \geq l$ and $mid = \frac{m}{2}$ we see that $k \geq \frac{m+l}{4}$. Similarly, $k = mid + s \leq \frac{m}{2} + l \leq \frac{m}{2} + \frac{l}{2} + \frac{m}{4} + \frac{l}{4} = \frac{3}{4}(m+l)$. The second recursive call has size $m+l-k$, which similarly lies within $[\frac{m+l}{4}, \frac{3}{4}(m+l)]$. The running time of

a single call is dominated by binary search, which takes time $O(\log(m+\ell))$. So we obtain the recurrence $W(n) = W(\alpha n) + W((1-\alpha)n) + \Theta(\log n)$ for $\alpha \in [\frac{1}{4}, \frac{3}{4}]$

What is $W(n)$? By induction, we can verify that if we choose $P > Q$ such that ① the work per-node is at most $Q \log n - 4$ for $n > n_0$ ② for $n \leq n_0$, the work is at most $P - Q$, then $W(n) \leq Pn - Q\log n$. Indeed, then $W(n+1)$ is at most $P\alpha(n+1) - Q\log\alpha(n+1) + P(1-\alpha)(n+1) - Q\log(1-\alpha)(n+1) + Q\log(n+1) - 4$

$$= P(n+1) + Q\log\left(\frac{1}{\alpha(1-\alpha)}\frac{1}{16}\frac{1}{n+1}\right) \leq P(n+1) + Q\log\left(\frac{16}{3}\cdot\frac{1}{16}\right) - Q\log(n+1)$$

$\log\left(\frac{16}{3}\cdot\frac{1}{16}\right) \leq 0$

The span is easier to calculate. Observe that both recursive calls have size at most $\frac{3}{4}(m+\ell)$, so the tree has depth $O(\log n)$. Since each call does $O(\log n)$ work, the overall span of merge is $O(\log^2 n)$.

Plugging in this new, parallel merge algorithm into Merge Sort, we obtain now that combining has span $O(\log^2 n)$ and the tree had depth $O(\log n)$, so the overall span is $O(\log^3 n)$. As we verified above, the parallel merge still only uses work $O(n)$, so the overall work of our parallelized MergeSort remains $O(n \log n)$. The correctness of the parallel merge algorithm is immediate by induction- $D[1 \cdots s+mid]$ only contains elements $\leq B[mid]$, and $D[s+mid+1 \cdots \ell+m]$ contains the elements $> B[mid]$, so $D[1 \cdots \ell+m]$ is a merged list in sorted order.