CSE 347 Analysis of Algorithms April 24, 2019

Lecture 25 Summary

Instructor: Brendan Juba

1. We consider an idealized model of parallel running time, the *span*.
   (a) We allow an algorithm to *spawn* a new thread at any point, that runs in parallel with the original thread. The original thread may then at any later point *sync*, waiting for all of the threads it has spawned to terminate and return values.
   (b) We suppose we have an unlimited number of cores available, so whenever a thread is spawned it can be assigned to a new core and run in parallel. The span is then the number of steps for the algorithm to terminate. The *work*, by contrast, is the total number of steps run over all cores during the computation. (This is the same as the running time if we had a single core.)
   (c) If the span is similar to the work, then we cannot achieve much speed-up by using more cores. So we analyze the span to diagnose bottlenecks in parallel computation.
   (d) Conceptually, for the execution of an algorithm on a fixed input, we draw the following directed acyclic graph: nodes correspond to steps of the computation, and we have an edge from node s to node t if either t follows s, s spawns t, or t is a sync and s was the final step of a spawned thread.
      i. Then: span is the length of the longest directed path in this graph
      ii. Work is the total number of nodes in the graph
      iii. A sequential computation is a chain of length equal to the running time
      iv. We will frequently consider these graphs in which chains are replaced by nodes labeled with the number of steps they represent.

2. Parallelizing divide-and-conquer: matrix multiplication
   (a) Divide-and-conquer algorithms parallelize naturally: we spawn the recursive subcomputations, and sync before the combine step. But a naïve application of this strategy often leaves bottlenecks that could be improved.
   (b) Recall our naïve $O(n^3)$ recursive matrix multiplication algorithm: $A = \left[\begin{array}{c|c} A^{(1,1)} & A^{(1,2)} \\ \hline A^{(2,1)} & A^{(2,2)} \end{array}\right]$, $B = \left[\begin{array}{c|c} B^{(1,1)} & B^{(1,2)} \\ \hline B^{(2,1)} & B^{(2,2)} \end{array}\right]$, $C = \left[\begin{array}{c|c} A^{(1,1)}B^{(1,1)} + A^{(1,2)}B^{(2,1)} & A^{(1,1)}B^{(1,2)} + A^{(1,2)}B^{(2,2)} \\ \hline A^{(2,1)}B^{(1,1)} + A^{(2,2)}B^{(2,1)} & A^{(2,1)}B^{(1,2)} + A^{(2,2)}B^{(2,2)} \end{array}\right]$, so we recursively solve these eight subproblems of size $n/2$.
   (c) The DAG we draw to analyze span closely mirrors the tree we draw to analyze the recurrence. The span is $O(n^2)$–it is dominated by computing the four sums of $n/2 \times n/2$ matrices during the combine step.
   (d) If we spawn a separate thread to compute each entry of these sums, then (if we can spawn all n^2 threads at once) the span drops to $\log n$. (If we more realistically can only spawn one thread at a time, then it takes $\log n$ steps to spawn n^2 threads, so we

would have a span of $\log^2 n$.) This is *much* better, and shows that matrix multiplication parallelizes well.

3. A more sophisticated combine step: MergeSort

   (a) Applying the simple parallelization strategy to MergeSort, we find that the merge step has span $O(n)$, which is terrible.

   (b) We can reduce the span using a divide-and-conquer strategy for the merge that allows the two lists to have different lengths

      i. We pick the midpoint of the larger list, and split the smaller list at the point where the larger list's midpoint would appear. We then recursively merge the prefixes of the two lists (up to these split points) and the suffixes, putting them directly in place in the merged list.

      ii. The subproblems are most imbalanced when the lists are about the same size and all/none of the smaller list ends up in a subproblem. Then we get a $\frac{1}{4}$–$\frac{3}{4}$ split.

      iii. Regardless, this shows that there are $O(\log n)$ levels of recursion where each is dominated by binary search (to find where the midpoint falls in the smaller list) so that the span of this divide-and-conquer merge is $O(\log^2 n)$.

      iv. We still want that this divide-and-conquer merge still has work $O(n)$ so that the overall work remains $O(n \log n)$. The difficulty is that the subproblems have some sizes αn and $(1-\alpha)n$ for some $\alpha \in [1/4, 3/4]$. Nevertheless, we can show by induction that regardless of the value of α in this range, the solution to the resulting generic recurrence is still $O(n)$.

   (c) Now, our parallelized MergeSort has $O(\log n)$ levels of span $O(\log^2 n)$, so its span is $O(\log^3 n)$ overall. Since our new merge routine still only has $O(n)$ work, the work remains $O(n \log n)$.