Parallel Scheduling

1 Nested Parallelism and Brent's Theorem

In lecture 1 we presented the *Nested Parallelism Model*. This is an abstraction which allows one to analyze a parallel program. The model is for a hypothetical parallel machine which supports *fork/join parallelism*. The program can fork into multiple processes that run in parallel. The parent suspends at the fork point and resumes after all of its children have completed. Using this model one can analyze a program for its work and span.

The model is, of course, not a realistic one for actual computing – it assumes an infinite number of processors. Nontheless, it's a useful abstraction for two reasons:

1. It gives a programmer a measure of how well a program will perform without having to consider a plethora of details. Those details would include the particular hardware on which it is running, as well as other duties carried out by the operating system, such as the scheduling which tasks are to be done by which processors. (This is analogous to the way in which we use big-oh bounds when analyzing algorithms, as opposed to struggling to write a function which computes the running time down to the millisecond.)
2. The work and span bounds derived using the model can be used to a obtain viable estimates of the actual "wall clock" time that it will take for an algorithm to run on a specific parallel machine.

Point 2 is a consequence of a theorem that was mentioned in the very first lecture.

Theorem: Brent's Theorem

An algorithm with work W and span S can be scheduled on a P-processor machine to run in max $O(\max(\frac{W}{P}, S))$ time.

Matching this we have:

Theorem: Greedy Scheduling Theorem

There exists a greedy scheduling algorithm that achieves the bound of Brent's Theorem.

We will prove these theorems later. First we need to define the *computation graph* and the *pebble game*.

2 The Computation Graph

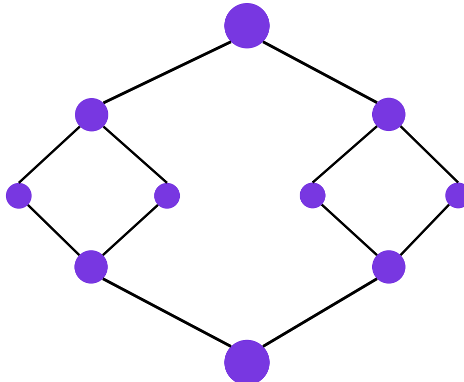
In order to prove the theorem we will represent a computation as a DAG (directed acyclic graph). We'll start with an example.

Algorithm: Parallel Sequence Sum

```
fun sum(S : sequence<int>) -> int:
  match length(S) with:
    case 0: return 0
    case 1: return S[0]
    case _:
      L, R = split_mid(S)
      Lsum, Rsum = parallel (sum(L), sum(R))
      return Lsum + Rsum
```

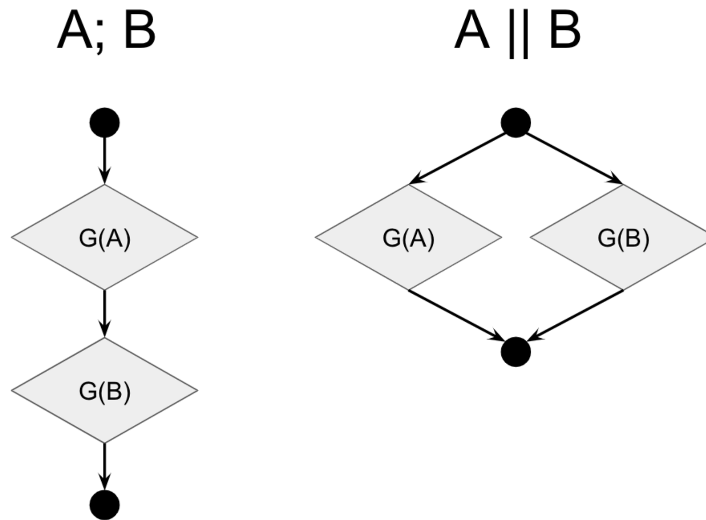
In this code the parent process does the base cases or the split. The fork is indicated by the call to `parallel(_,_)`. When that returns the parent process resumes, computes the sum, and returns it.

We an represent this by the following *computation graph*.



This illustrates what happens when the input is of size four. All the edges are directed from top to bottom. Each node represents a constant amount of work. In this case the top node represents the `match` and the `split_mid(S)`. The two four-node graphs on the left and right represent the recursive calls to `sum(L)` and `sum(R)`. Then the bottom node represents the computation of `return Lsum + Rsum`.

The computation DAG always has one starting node (at the top), and one ending node (at the bottom). And it is constructed recursively. The meaning of an edge in the graph from a node x to a node y is that computing y depends on having completed x. In this case we will say that x is a predecessor of y. If we had one processor then the computation can be carried out in any topological order of the DAG, which would respect the order requirements of the DAG.



The picture above on the left shows how a graph is constructed for two computations A and B that are not done in parallel. The programmer presumably did not do them in parallel because something the B code depends on a result computed by A. The one on the right shows the parallel case. A parallel `for` loop would result in a graph with a potentially much higher branching factor.

Notice that the work, as we have used it throughout this course, corresponds to the number of nodes in the computation graph. Similarly, the span corresponds to the number of nodes on the *longest path* from the start node to the end node.

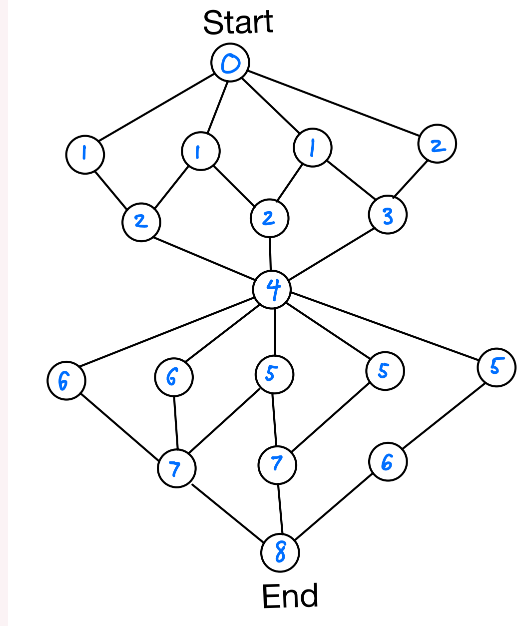
The pebble game defined in the next section will enable us to see how to carry out the computation represented by a computation graph using p processors. We will be able to use work, span, and the number of processors to get an estimate of the wall-clock-time a computation will take. This will enable us to prove Brent's Theorem, and the matching lower bound.

3 A Pebble Game

The pebble game is a way to visualize the process of scheduling a computation represented by a computation graph using p processors. Placing a pebble on a node corresponds to doing the computation represented by that node. The pebbles are placed onto the graph in rounds, as explained below:

> At the start of the game, there is a pebble just on the start node. That is the pebble of round 0. In each subsequent round up to p pebbles are simultaneously placed on nodes of the graph. Once a pebble is placed it never moves. A pebble may be placed on any node in round i as long as all of its predecessors were pebbled **in an earlier round**. Thus you can only pebble y in round i if the nodes on which it depends on were pebbled in some **previous round**.

Example: Example of Pebbling a Graph



The figure above shows how a graph might be pebbled when $p = 3$. The numbers in the nodes are the rounds in which that node is pebbled. In this example the greedy strategy is followed, in which each round pebbles as many nodes as possible, up to p.

This game is therefore defined by some value of p, and the computation graph that the game is being played on. The goal is to minimize the number of rounds that it takes to cover every node in the graph with pebbles. In the context of parallel computing, we can think of this as a program being run on a computer with p processors. Each round represents one unit of time, where each processor can compute one of the nodes that received a pebble.

It turns out that finding the minimum number of rounds needed to pebble the graph is an NP-complete problem[1]. However it also turns out that a simple greedy strategy can get within a factor of two of optimal, as we'll see below.

3.1 Greedy Strategies

For each round, the set of nodes that can receive a pebble are said to be **ready**. In a given round any pebbling algorithm is only allowed to pebble ready nodes. Let r be the number of ready nodes. If $r \leq p$ then it is logical to pebble those r nodes. If $r > p$ then there is more than one way to place p pebbles. A strategy for placing pebbles is said to be **greedy** if, on every round, it places $\min(r, p)$ pebbles. In other words, a strategy is greedy if it **places as many pebbles as possible** in each round.

[1] J. D. Ullman, "NP-complete scheduling problems," J. Comput. Syst. Sci., vol. 10, no. 3, pp. 384–393, 1975.

Theorem: You Might As Well Be Greedy

Let P be a pebbling process for a graph G with p processors that completes in R rounds. Then there is a greedy pebbling process that completes in at most R rounds.

Proof. Proof: We can convert P into a greedy algorithm which finishes in equal or fewer rounds. We run P and the modified process M in parallel. We preserve the invariant that at any time the vertices pebbled in P's graph is are a subset of (or possibly equal to) those pebbled in M's graph.

Whenever P is non-greedy and places fewer pebbles than greedy would, the modified process pebbles the same nodes that P did, and then a few more to be greedy. This is legal because the invariant insures that if a node is ready in P it's also ready in M. So the extra nodes only serve to preserve the invariant.

Finally, the modified algorithm terminates no later than P. □

Corollary

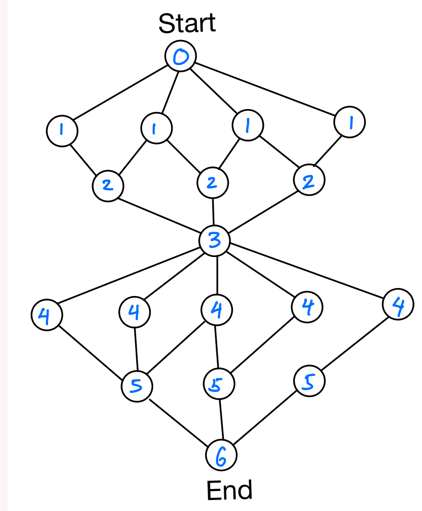
There exists an optimal greedy strategy.

Problem 1. Find an example for where a greedy strategy does not produce an optimal pebbling.

3.2 Lower and Upper Bounds on Pebbling

It will be useful in the analysis ahead to consider a specific case of the pebble game when the number of processors is unbounded. As usual round 0 is for the pebble on the start node. In general the nodes pebbled in round i are those nodes x for which the longest path from the start node to x is of length i. This partitions the graph into a sequence of layers. Call them $L_0, L_1, \ldots, L_d$. So L_i is the set of nodes not in layers $L_0, \ldots, L_{i-1}$ all of whose predecessors are in those layers.

Example: Layers



The example above shows the layers of a computation graph.

With this in hand we can prove a lower bound on the number of rounds necessary to pebble a graph.

Theorem: Lower Bound on Pebbling

At least $\max\left(\left\lceil \frac{n}{p} \right\rceil, d\right)$ rounds are necessary to pebble a graph with n nodes and d layers.

Proof. We'll show that both $\left\lceil \frac{n}{p} \right\rceil$ and d are lower bounds on the number of rounds.

For the first term, we consider that on each round, we are allowed to put at most p pebbles on the graph, and we need to put n pebbles down in total. Therefore, we will need at least $\left\lceil \frac{n}{p} \right\rceil$ rounds to have enough pebbles to cover the whole graph – any smaller number of rounds will not provide enough pebbles.

For the second term, note that there is a non-empty layer L_d. There exists a path from the start node to a node in L_d is of length d. Each of the d nodes on this path must be pebbled in a different round. Therefore the number of rounds is at least d. □

Now we are ready to prove that any greedy strategy is within a factor of two of optimal.

Theorem: Greedy is Good

Any greedy strategy will take at most $\frac{n}{p} + d$ rounds.

Proof. Let's divide the graph into levels, as before. We can then observe that on every round, at least one of the following events occur:

1. p pebbles are placed.
2. A level is finished, i.e. the last pebble on that level is placed.

To prove this, consider the situation at the beginning of a round. Let j be the lowest numbered level with an unpebbled node in it. Note that all the unpebbled nodes in level j are ready. Suppose that event 1 above does not happen, and fewer than p pebbles are put down in this round. The only way that a greedy algorithm will do this is if it exhausts all of the ready nodes. Therefore it must complete L_j in this round. So the event of type 2 occurs.

So every round is classified as type 1 or type 2, or both. The number classified as type 1 is at most $\left\lfloor \frac{n}{p} \right\rfloor$. The number classified as type 2 is at most d.

Therefore $\frac{n}{p} + d$ is an upper bound on the number of rounds. □

Here's an useful property of the max function which holds for all non-negative real numbers a, b:

$$\max(a, b) \;\leq\; a + b \;\leq\; 2\max(a, b).$$

This means that a greedy strategy's number of rounds is at most twice that of the optimal strategy.

4 Conclusions

The span of a computation S corresponds to d, and the work W corresponds to n. So the results of the previous section prove Brent's theorem, and the Greedy Scheduling Theorem.

Consider the two terms in this expression for how long a program will run:

$$\text{time} \;=\; \Theta\left(\frac{W}{p} + S\right)$$

Consider what happens to this expression as you add more processors. At what point does adding more processors begin to give a negligible improvement to this? The answer is that the improvement becomes negligible when the two quantities are approximately equal, namely:

$$p \;\simeq\; \frac{W}{S} \;=\; \text{“parallelism”}$$

The quantity on the right is therefore called the **parallelism** of the algorithm. Roughly speaking it's about how many processors can be usefully utilized to speed up the algorithm. Beyond that, the returns begin diminish.

Which is why in this course we strive to obtain algorithms with low span.