We will now formalize the DAG model. We will also see how parallel for loops are implemented and what are reductions.

1 The dag model for parallel computation

To analyze a parallel algorithm, it’s helpful to think of the execution of the program as a directed acyclic graph (dag), where nodes represent instructions, and edges represent dependencies between instructions. If there is an edge from node $u$ to node $v$, then node $u$ must be executed before node $v$. A serial computation is just a chain of instructions. A parallel computation is a dag. If there is no path from node $u$ to node $v$ and vice versa, then there is no prescribed ordering between $u$ and $v$, and they can be executed in any order, or in parallel with each other.

For convenience, we won’t represent each instruction as a node. Instead, we will group together a sequence of instructions that does not contain any parallel keywords (i.e., spawn, sync, or returning from a spawned function), referred to as a strand, into a single node, and the dag models dependencies between strands.

To make this model concrete, let’s take a look at how we model the dag for a parallel computation. Recall the Fibonacci code from last lecture:

```
Fib(n)
1   if n ≤ 1
2       then return 1
3   x ← spawn Fib(n - 1)
4   y ← Fib(n - 2)
5   sync
6   return x + y
```

The DAG representation of Fib(4) is shown below.
In this model, just like the RAM model, we will assume a basic set of instructions, and each instruction takes a constant amount of time, including access to memory. Unlike the RAM model, however, notice this Dag model is not really tied to the underlying machine but rather tied to the control structure of the program. As it turns out, there is a way to map the costs we derive using this model onto costs for running on parallel machines (machines with multiple processing units).

In terms of the dag representation of a parallel computation, let’s assume that we chop up the strands so that each node consists of only one instruction (i.e., each node has cost one). Then, the work is the total number of nodes in the graph, and the span is the number of nodes along the longest path in the dag.¹

2 Analyzing parallel_for loops

We said that parallel for loops are syntactic sugar for spawns and syncs. Let’s see how they are implemented. Say we have a simple parallel_for for copying elements of an array to another array:

```
1  parallel_for i ← 1 to n
2         do B[i] ← A[i]
```

The parallel_for is just syntactic sugar, and is actually implemented using spawn and sync. So let’s think about how we can do that. A parallel_for simply means that all iterations can potentially execute in parallel, so one simple way of implementing it is to spawn the iterations in a for loop:

```
1  for i ← 1 to n
2     do B[i] ← A[i]
```

¹If each node may cost different amount, then the span is the sum of the cost along the path in the dag that has the largest such sum.
COPYELEM(A, B, i)
1   B[i] ← A[i]
2   return
3   for i ← 1 to n
4     do spawn COPYELEM(A, B, i)

By spawning off each iteration, the code indicates that this iteration can execute in parallel with the rest of the for loop (the later iterations).

What is the work and span of this simple piece of code? It’s easy to see the why that’s the work and span by looking at the computation dag generated by this computation:

![Computation Dag]

The work is \( \theta(n) \), which is asymptotically the same as if we had removed the spawn keyword (albeit the spawning does increase the work by a constant amount compared to its serial counterpart, i.e., code without spawn). What about the span? The span is also \( \theta(n) \) — the longest path follows the entire chain of nodes that spawn off each iteration plus a node that executes an iteration. Thus, this code sadly has PUNY amount of parallelism — \( \theta(1) \)!

The fundamental problem here is that, this way of implementing parallel for causes the spawning of each iteration to depend on one another, and we are throwing away the parallelism among iterations. This may be O.K. if each iteration contains substantial amount of work, but that’s not the case here.

**Exercise 1** What are the work and span in the code above, if we had replace COPYELEM with some other function that has work and span of \( \theta(n) \)?

**Decreasing the span via coarsening**

Well, there isn’t much point to run a parallel program that has a parallelism of \( \theta(1) \) (you might as well execute it serially!), so let’s see if we can increase the parallelism by decreasing span.

Let’s coarsen the base case so that now each time we spawn, we execute \( G \) iterations (\( G \) for grainsize):
COPYMULITPLELEMTS(A, B, start, end)
1    for j ← start to end
3    return

for i ← 1 to n by G
5    do spawn COPYMULITPLELEMTS(A, B, i, MIN(i + G, n))

With this strategy, work is $nT_{iter} + (n/G)T_{spawn}$, which is still $\theta(n)$. Even though we didn’t asymptotically decrease the work, it is less overhead compared to no coarsening (which is $nT_{iter} + nT_{spawn}$).

Span has also decreased: $T_\infty = n/G + G$, and if we use $G = \sqrt{n}$, we get $T_\infty = \theta(\sqrt{n})$, which is asymptotically less than the span without coarsening.

In this case, since the work in each iteration is small, coarsening helps with parallelism, because we are removing the serial dependencies between each iteration. In general, coarsening tends to decrease the work but increase the span, and therefore decrease the parallelism. Depending on the computation, this can still help. In particular, if the computation already has plenty parallelism, it is a good idea to trade-off parallelism to decrease the work overhead due to spawn.

Binary splitting the iteration space

Besides coarsening, what else can we do to decrease the span? One strategy is to spawn-off parallel iterations in a binary tree:

COPYARRAY(A, B, start, end)
1    if start = end
2        then B[start] ← A[start]
3        return
4    mid ← [(start + end)/2]
5    spawn COPYARRAY(A, B, start, mid)
6    COPYARRAY(A, B, mid + 1, end)
7    return

Assuming $n = 8$, this computation generates the following dag:
The work of this computation is again $\theta(n)$, which you can analyze by replacing the `parallel_for` with just a regular `for`. It may seem that we are not accounting for the spawning overhead when we analyze the work this way. In fact as we have seen earlier, the spawning does increase the work, but not asymptotically, because in a binary tree, the number of internal nodes equals to the number of leaf nodes minus one, and assuming we are doing constant amount of work per internal node, we are not increasing the work asymptotically. Of course, if the work in each iteration is small, this is still substantial work overhead just to spawn off iterations.

Spawning off iterations in a binary tree indeed helps with the span. The span of this computation is just $\theta(lg n) + \theta(1)$ (which is just $\theta(lg n)$), because the part where the iterations are being spawned off has a recurrence of $T_\infty(n) = T_\infty(n/2) + \theta(1)$, which is $\theta(lg n)$, and the span of a leaf node is just $\theta(1)$.

**Exercise 2** How would you coarsen COPYARRAY? What would be the work, span, and parallelism of coarsened COPYARRAY? Does coarsening help?

General rule of thumb when writing parallel code:

- Try to make sure that the amount of work per spawn is large enough.
- If you have plenty of parallelism, try to trade some of it off to reduce work overhead from spawning.
- Use `cilk_for` (implemented using binary splitting) instead of a regular for loop with `cilk_spawn`.
- If you have nested loops that you can parallelize, parallelize the outer loops as opposed to the inner loops, if you are forced to make a choice.

\[^2\text{In Cilk Plus, there is a compiler pragma: \texttt{\#pragma cilk grainsize = G} that goes with \texttt{cilk_for}, which tells the compiler to generate code that uses base case of G iterations.}\]
3 Reduce

Note that we cannot use parallel for for summing all elements of an array due to the race. A reduce operation is designed for exactly that purpose, so let’s see how that works.

A reduce operation applies an associative function (such as sum, max, min, logical and, logical or ... etc) to a list or a sequence. We can compute a reduction in small work and span. For example, let’s use reduction to computes the sum of all the elements of an array:

\[
\text{SUM}(A, start, end) \\
1 \text{ if } start = end \\
2 \quad \text{then return } A[start] \\
3 \quad \text{mid} \leftarrow \left\lfloor \frac{start+end}{2} \right\rfloor \\
4 \quad s_1 \leftarrow \text{spawn SUM}(A, start, mid) \\
5 \quad s_2 \leftarrow \text{SUM}(A, mid + 1, end) \\
6 \quad \text{sync} \\
7 \text{return } s_1 + s_2
\]

The work of this operation is \(O(n)\) and the span is \(O(\lg n)\).