Wrapup coroutines

Speedup analysis

is a function. We are calling it

```
return coroutineScope {
    val left = async { countTask(arr, lo, mid, seqCutoff)}
    val right = countTask(arr, mid, hi, seqCutoff)
    left.await() + right
}
```

The parameter is ~~all the~~

all the code

inside the curly braces

~~incl~~ including the curly braces

return coroutineScope(

return

return coroutineScope(... function ...)

Kotlin has weird syntax where if the last parameter to a function is a function, you can (optionally) leave it out of the parens and then stick in curly braces

The code in the curly braces above is a function with no name, that we created on-the-fly. With fancy lingo, we call this a "lambda expression"

In a lambda expression, the value of the last statement executed is the return value for it

Hello friends, ignore me for I am writing.

So far so good? We'll see.

How thread pools actually work

Java has "ForkJoin" framework, which is very well documented on how it works

- Creates one <u>thread</u> per core

Each thread has a deque associated w/ it.

stack/queue combined
(add/remove from either end)

Whenever a thread generates more work, it adds it to the end of its deque

When a thread is bored, it gets the most recently added task to work on next

Why most recent and not oldest?
Seems like really old tasks would wait a long time.

in a recursive fork/join setup, the most recent tasks are the ones the others may depend on.
[just a heuristics]

- these deques are tracking work that <u>needs to be done</u>, not waiting/blocked tasks

---

Also implements "work stealing"
If a thread runs out of work (deque is empty), then it takes a task from a random other deque, and takes the oldest one from that (i.e. front of queue)

By taking from other end, it reduces contention over the queues

multiple actors trying to access at same time

by working on opposite ends, can use thread-safe deque designs to allow simultaneous access

(so to some degree, doesn't matter if you reversed whole thing)

Performance analysis

Quantify how good a parallel approach is

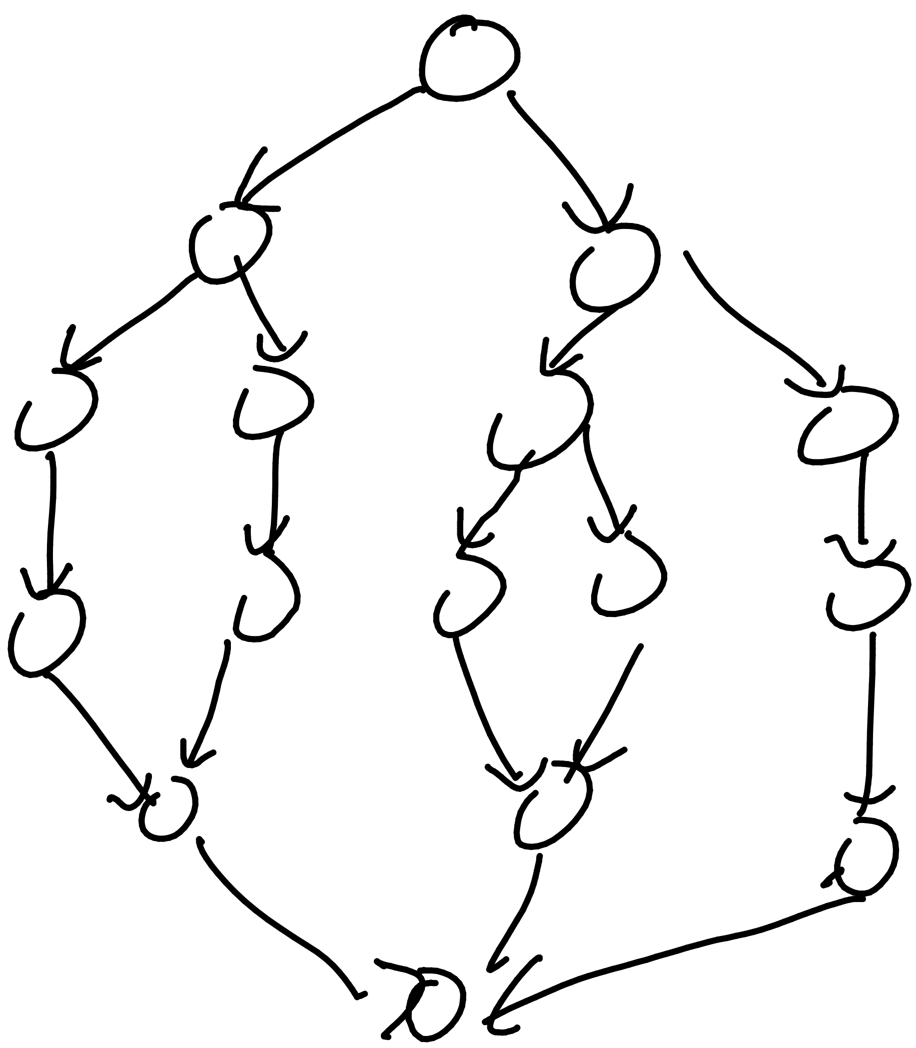
Imagine that our approach (algorithm) consists of a large number of $O(1)$ steps.

Define T_1 = amount of time to run on one processor (core)

If we have n steps, than

$T_1 = O(n)$

You can think of execution as a graph



sum by dividing

each level is multiple processors working at same time

time

T_1 is also thought of as all of the steps = $\boxed{\text{work}}$ done by algorithm

total # of steps that ~~need to~~ be done.

T1, as a runnning time, is the worse we can do. It's all sequential

What is the best that we can do? If we had infinite processors, how long would it take to run?
Would be 0 if nothing depended on each other

But if some tasks depend on others, then the best we can do is the length of the longest sequence of steps that have to be performed sequentially
= T_{infinity} = "span" of the algorithm

For sum example that we've done with fork/join, span is O(log n)

If we run on P processors, then the speedup is T1/TP.

If we consider the speedup if we had infinite processors

T1/T{inf} = speedup with infinite processors = "parallelism" of the algorithm