All computers in 308 have Docker

- mounted COURSES
- put the cs348 folder
- open in VS Code

Fork/join style in coding

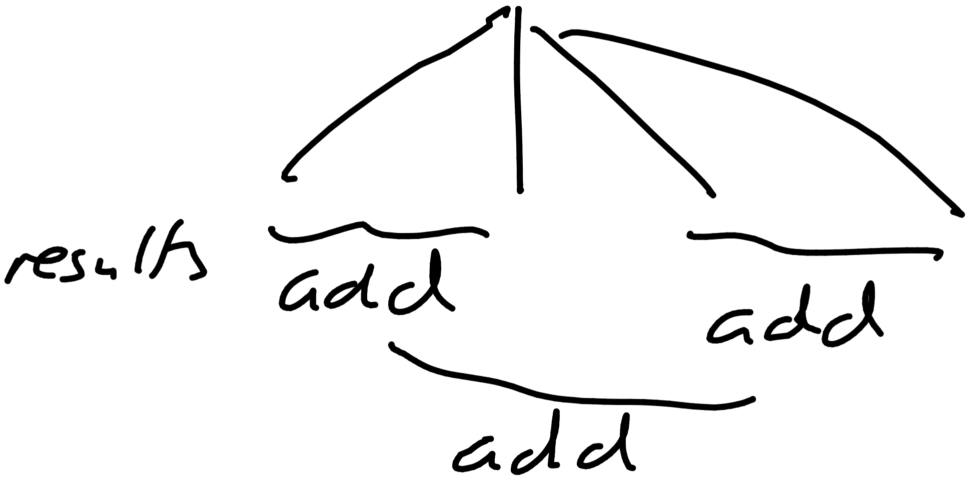
- and coding w/ Kotlin coroutines

Suppose you have a big, easily parallelizable job (e.g. sum all numbers), and you have 4 cores

Obvious? Split work into 4 threads, do addition in each, then add up 4 results.

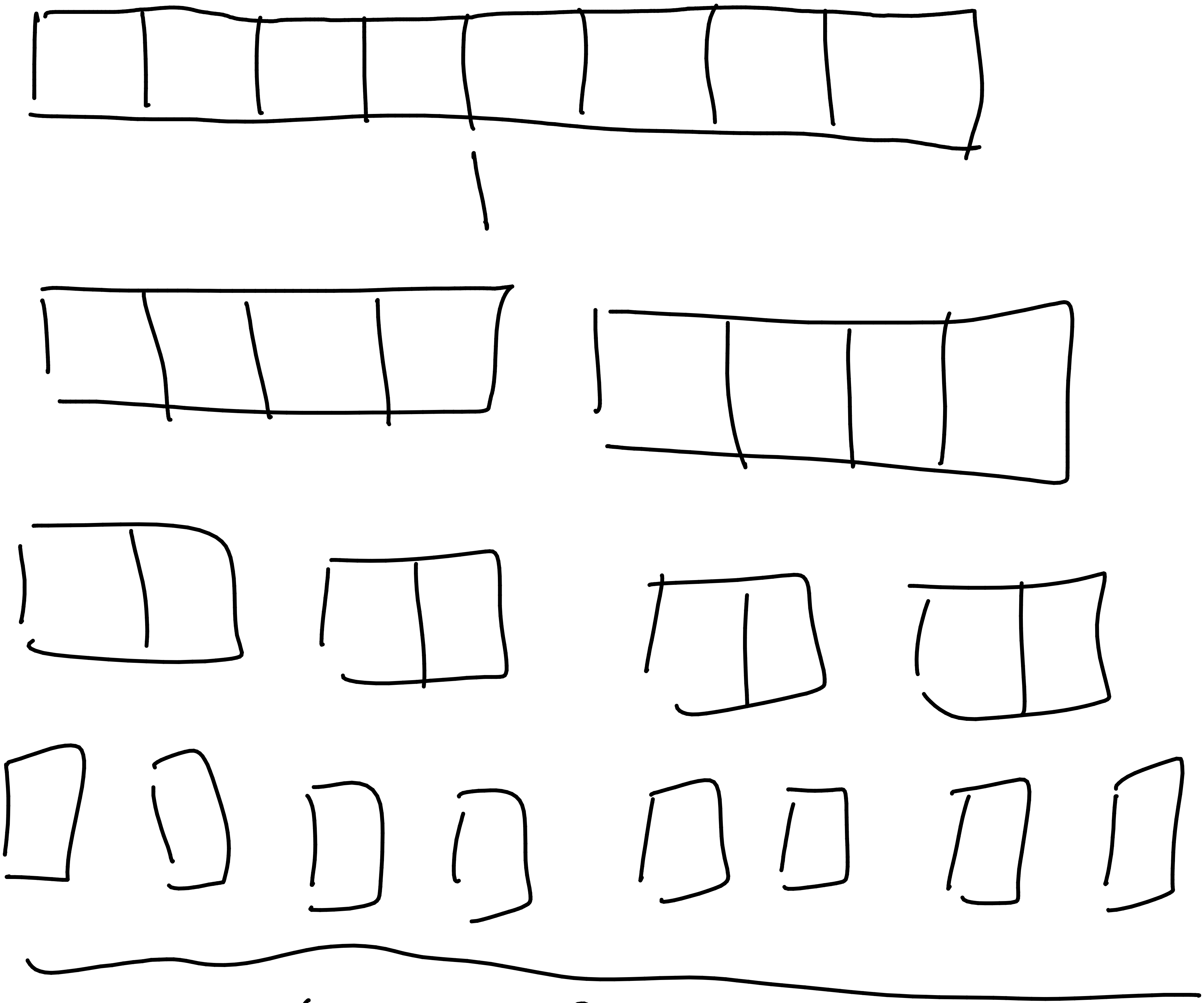
As a general principle, this could be problematic:

- assumes I can get equally to all cores at all times
- compute might take over a core and lots of work is left over
- splitting work <u>equally</u> may be hard (test if numbers are prime, and data is sorted)
- cores are now asymmetric - some are faster than others (slower <u>saves</u> energy)
- the combine work (add up all results) is <u>sequential</u>



in parallel!

Smarter idea: split into as small subproblems as you can, each getting its own thread *(oops)



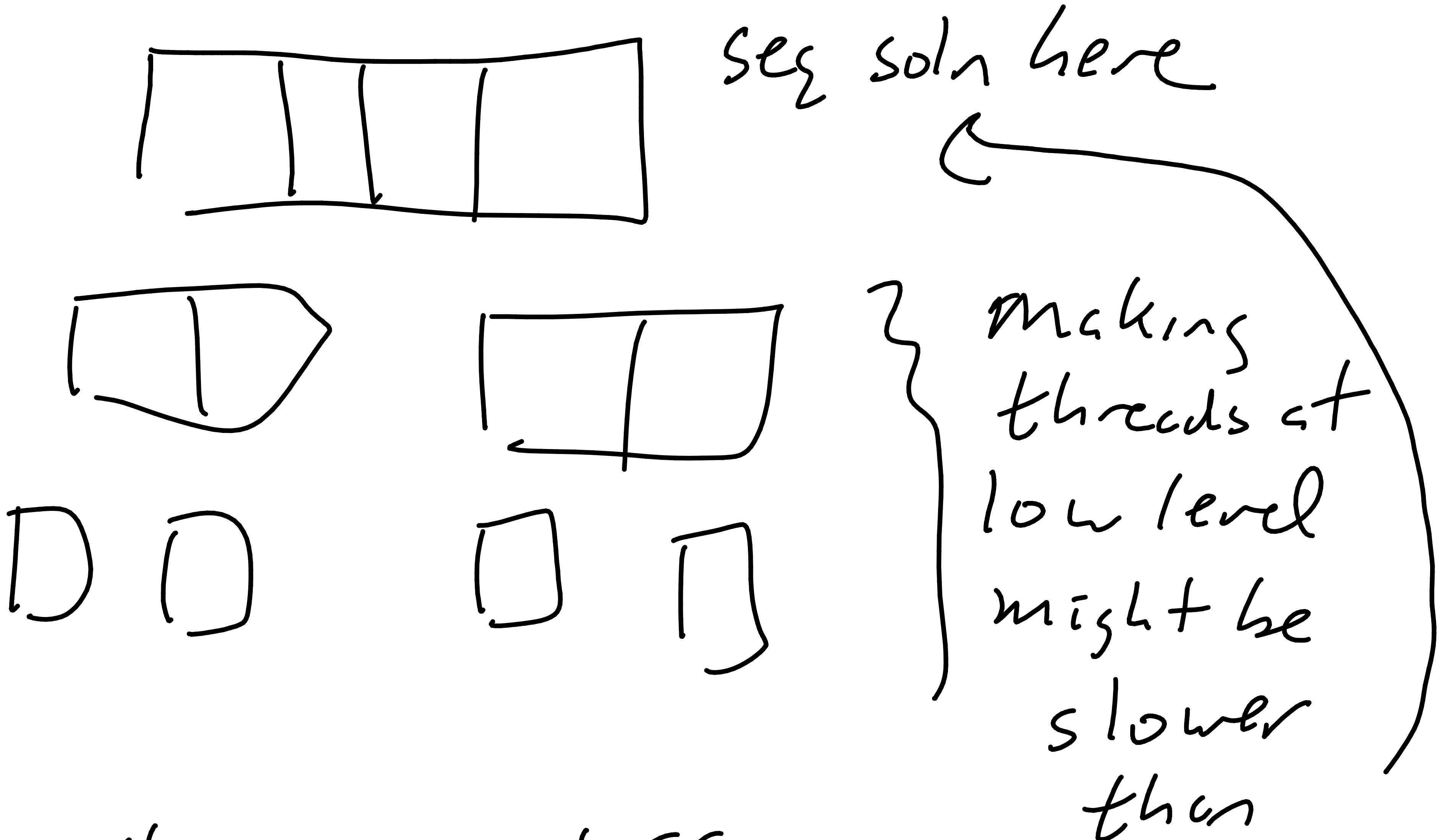
n threads for n data

Then aggregate up when done

The issues:

- generating threads takes time.

At some level of detail, you're just better off w/ a seq soln than overhead of creating more threads



Typically use a cutoff, determined experimentally and w/ wisdom to stop splitting

Because generated threads takes so much time, we get benefit from a thread pool.

a fixed number of threads that the PL (Kotlin) adds jobs to

Reading uses Java's "ForkJoin" framework to do this.

Instead, we'll use Kotlin coroutines, which also use thread pools.

scope

[As an aside, Kotlin coroutines also add structure to threads that vanilla threads don't have]

What is a coroutine?

[play on word subroutine]

Term has been around for decades, and often applies just for concurrent code

More recently, it might also include running in parallel

→ allows for concurrent (parallel?) work by aspects of program

- prog lang feature (whereas threads are an OS feature)
- PL retains control over it