Computer are dumb

(And why you care.)

Outline

- Algorithms and scaling
- Heuristics in computation
- Some thoughts on hardware
- Are you right? (Or at least not wrong?)

Consider: finding SNPs.

Given: reference genome, sequence reads, mapping.

The mapping contains a list of reads, mapped locations within reference, and the location of differences.

How can we find all single-nucleotide variation?

Approach one: by genome

```
for location in genome:
  reference = genome[location]
  bases = get_overlapping(location)
  for base in bases:
     if base != reference:
          # count SNP
```

Approach two: by read

```
for read in mapped_reads:
  for differences in read:
     # count SNP
```

Approach one: by genome

```
for location in genome:
  reference = genome[location]
  bases = get_overlapping(location)
  for base in bases:
     if base != reference:
          # count SNP
```

How does this algorithm scale?

Imagine:

- increasing size of genome
- increasing number of reads

Approach two: by read

```
for read in mapped_reads:
  for differences in read:
     # count SNP
```

How does *this* algorithm scale?

Scaling and Big-O notation

- The first approach scales with both the size of the genome and the number of reads:

$$t \sim O(N * M)$$

- The second approach scales with just the number of reads:

$$t \sim O(M)$$

Scaling and Big-O notation

- The first approach scales with both the size of the genome and the number of reads:

$t \sim O(N * M)$ - why would you want this??

- The second approach scales with just the number of reads:

$t \sim O(M)$

What about a different problem?

- I am interested in locations X, Y, and Z.
- Give me all SNPs at or near those locations.

```
for location in list_of_locations:
  reference = genome[location]
  bases = get_overlapping(location)
  for base in bases:
     if base != reference:
          # count SNP
```

Important note

- *Algorithm* scaling is independent of the actual time it takes to run.

- Scaling tells you how time-to-run scales as the problem size changes, nothing more.

Easy-to-check vs easy-to-find

Given a number, factor it into only prime numbers.

This is hard.

Given a set of prime numbers, verify that they multiple to yield a particular number.

This is easy.

Easy-to-check vs easy-to-find, #2

Suppose:

50 dorm rooms, two students per room

100 students can be admitted, of 400 total

Dean has list of students that cannot be paired.

It is *easy to check* any particular list of student/room combinations for validity.

In general, it is *extremely hard* to quickly find a guaranteed solution.

Heuristics

- "Heuristics" are short cuts that usually work (but occasionally go horribly wrong).

- Not all problems are amenable.
  - Prime numbers? No good, *fast* short cut.
  - Housing? Sure – start with a random solution, eliminate one of each pair that conflicts, until you find a non-conflict..

- Heuristics rely on assumptions about the specific type of problem you're going to tackle, and don't always work.
  - If the Dean is evil, he can construct a list of incompatible roommates that breaks your process.
  - Or he can just gives you a really long list of incompatible roommates.

Example: BLAST

BLASTN filters sequences for exact matches between "words" of length 11:

```
GAGGGTATGACGATATGGCGATGGAC
||x|||||x|||||||||||x|x||x
GAcGGTATcACGATATGGCGgT-Gag
```

This results in a O(n log n) algorithm.

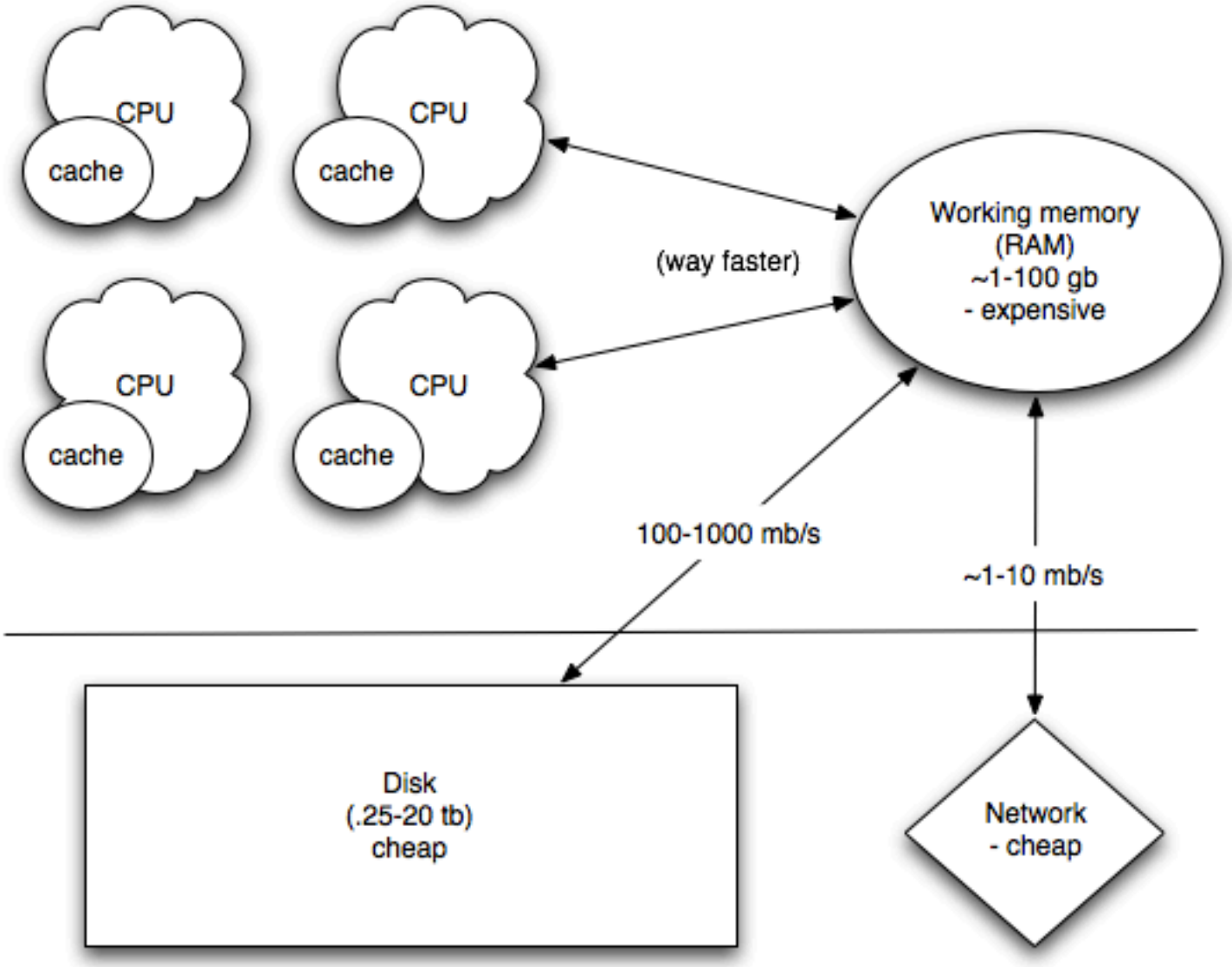
Example: BLAST

…but what about pathological situations?

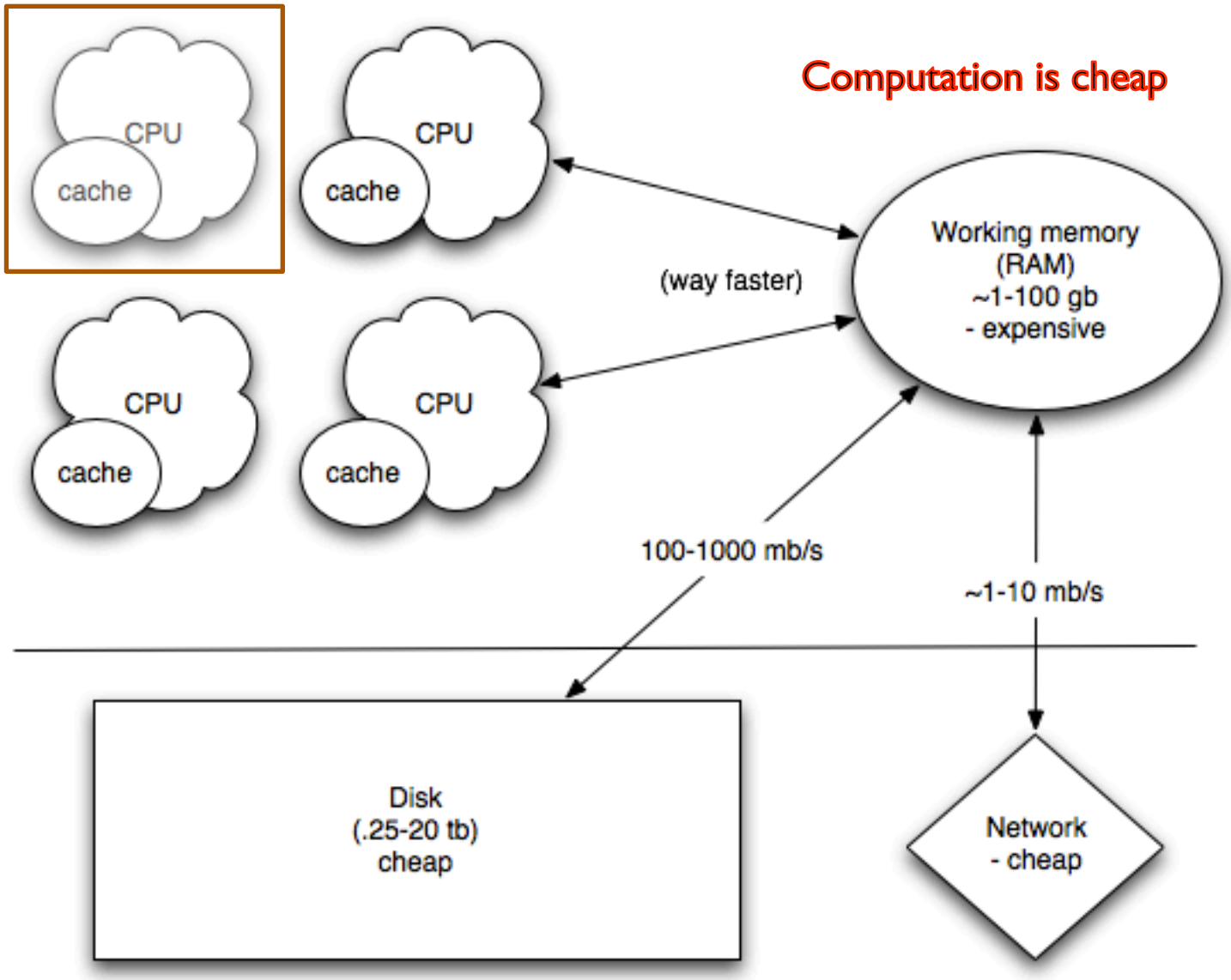
```
GAGGGTATGACGATATGGCGATGGAC
||x|||||x|||||x||||||x|x||x
GAcGGTATcACGATGTGGCGgT-Gag
```

This will not be scored as a match, because BLAST *only* scores matches with a core "seed" match of 11 bases.

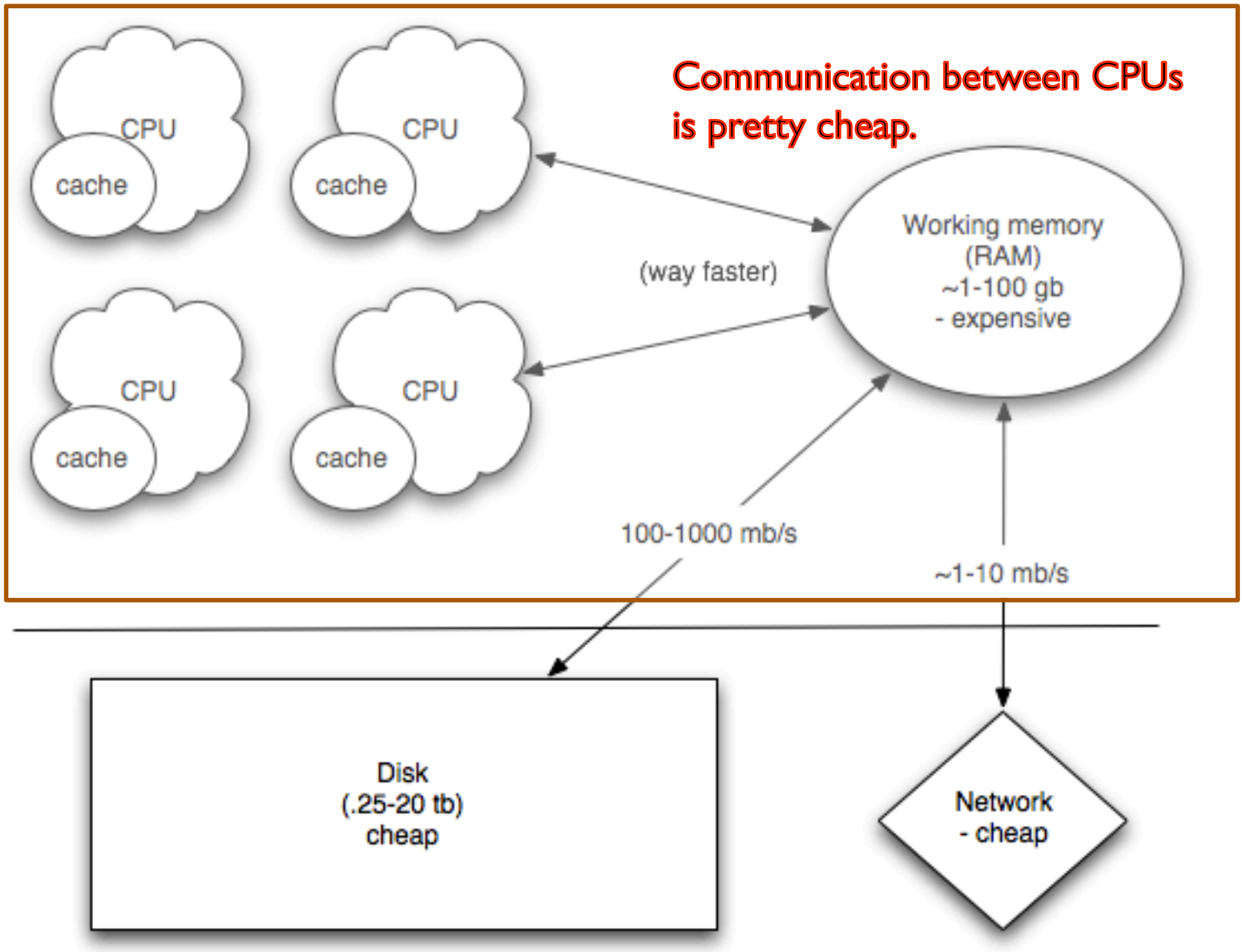
Computer architecture



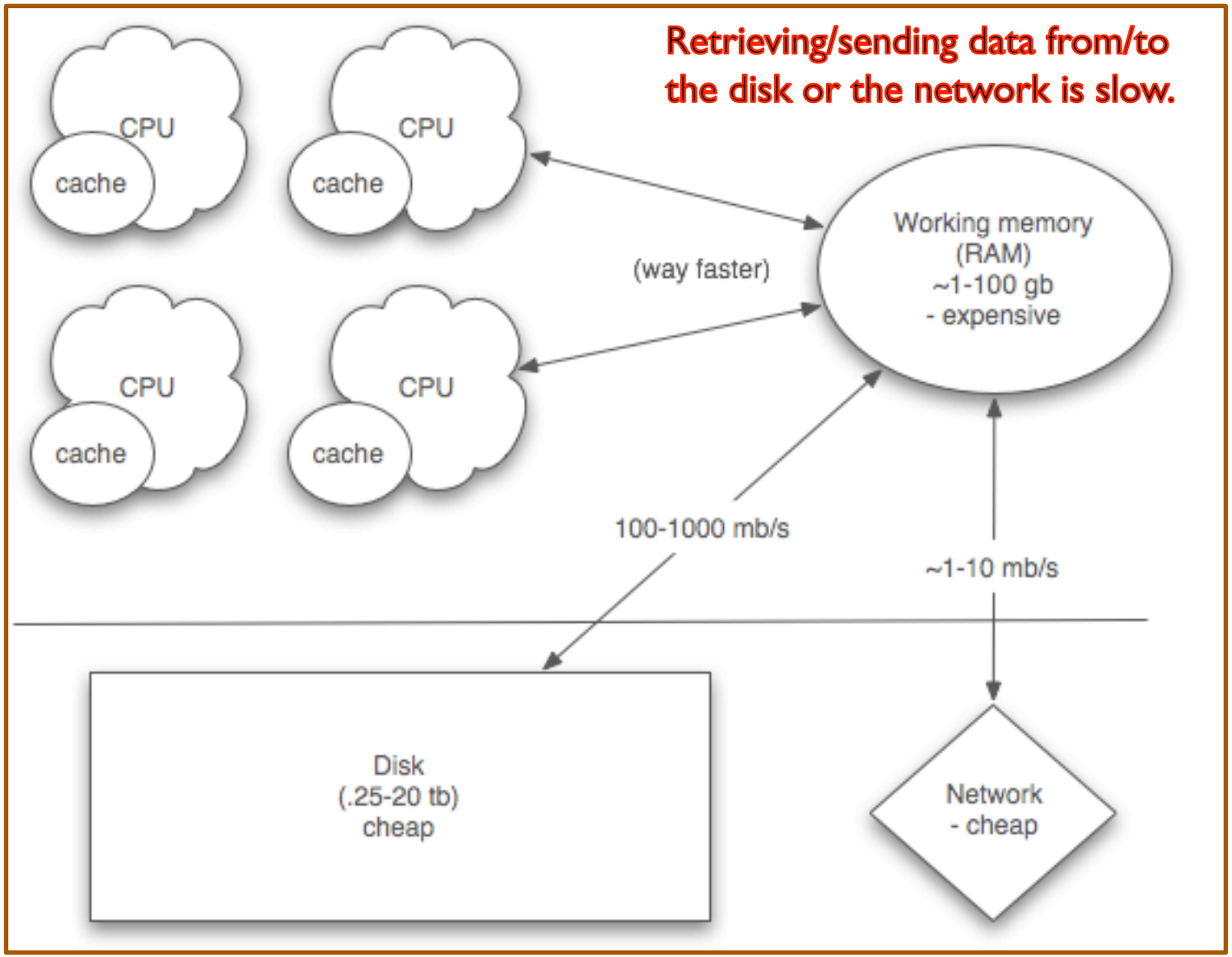
Computer architecture



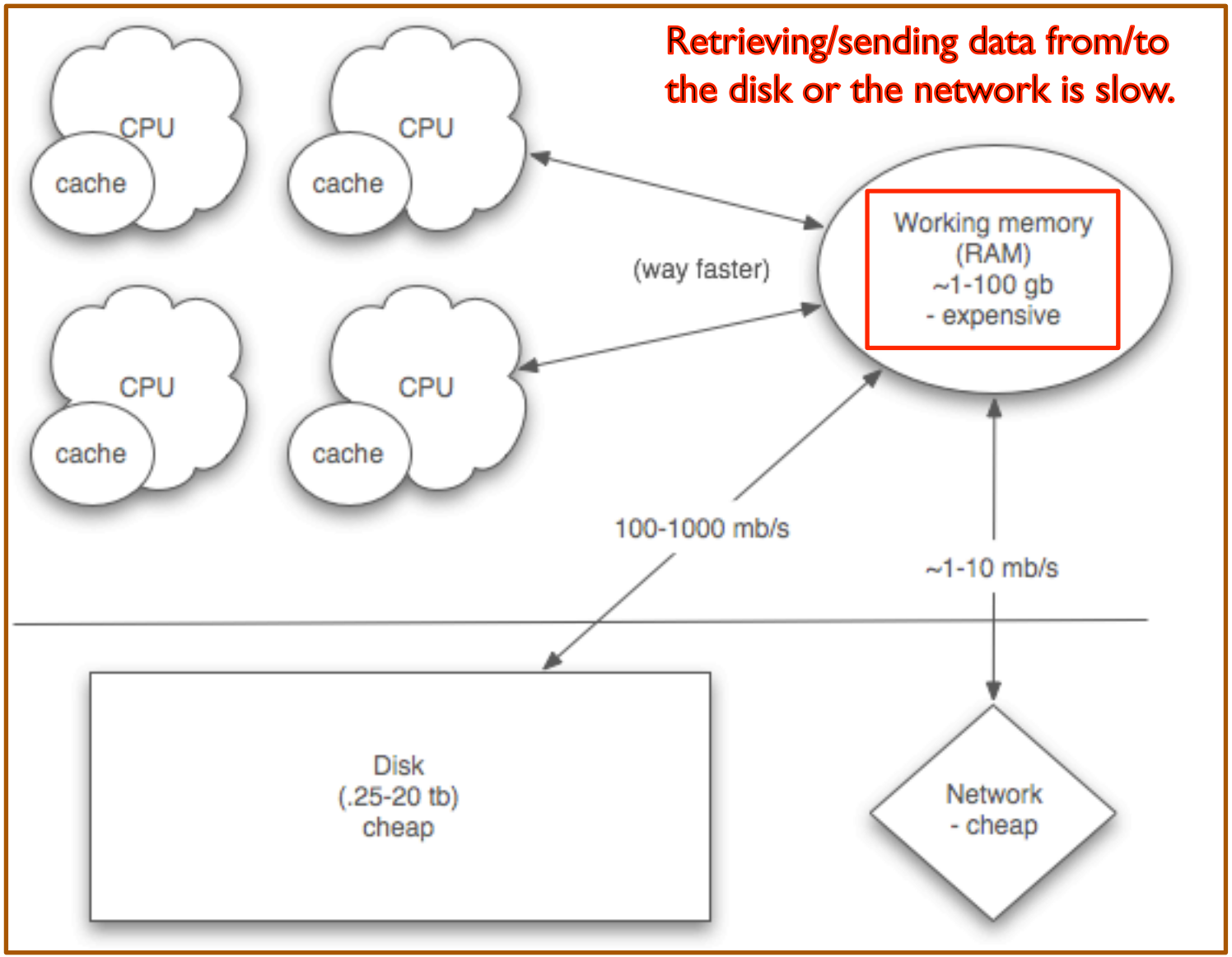
Computer architecture



Computer architecture



Computer architecture



Questions to ask

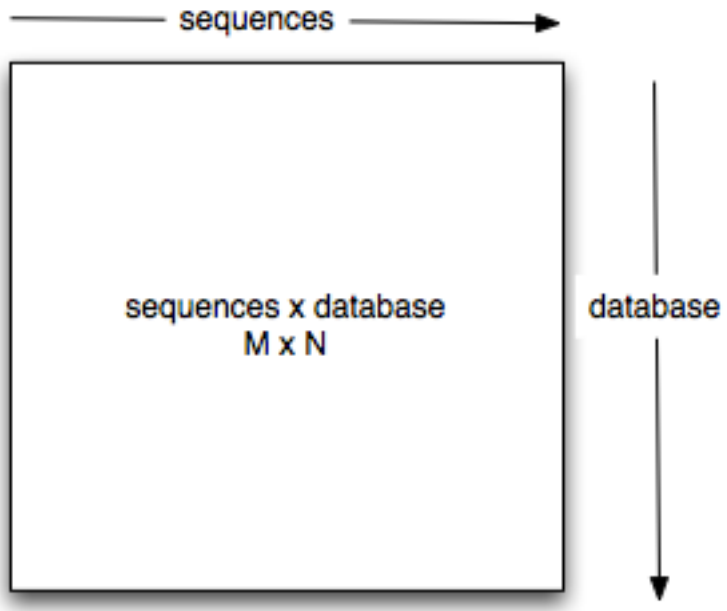
- Can I split my problem up into small chunks?

  (because, if so, I can use more than one computer effectively.)

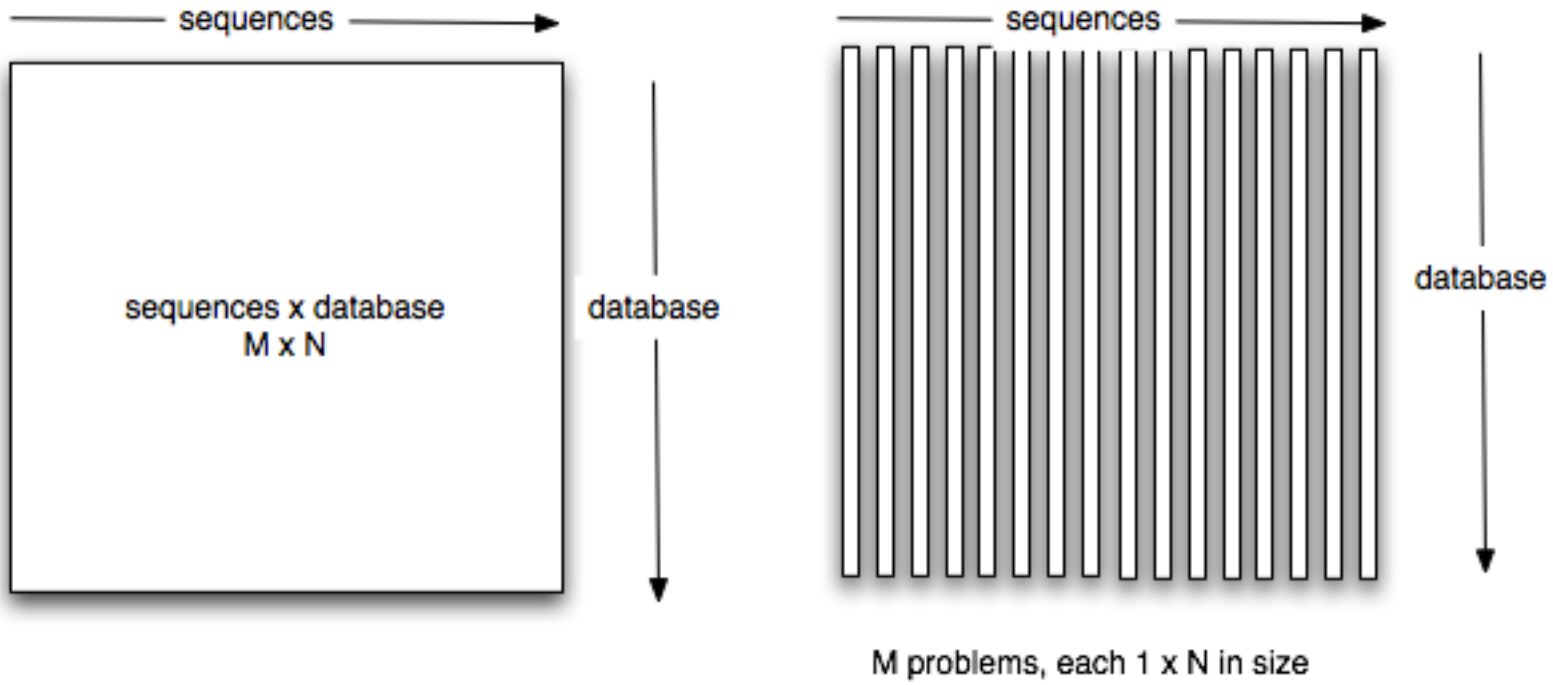
- How does my computation scale?

- How does my memory use scale?

Sequence comparison: n^2



…but "embarrassingly parallel"



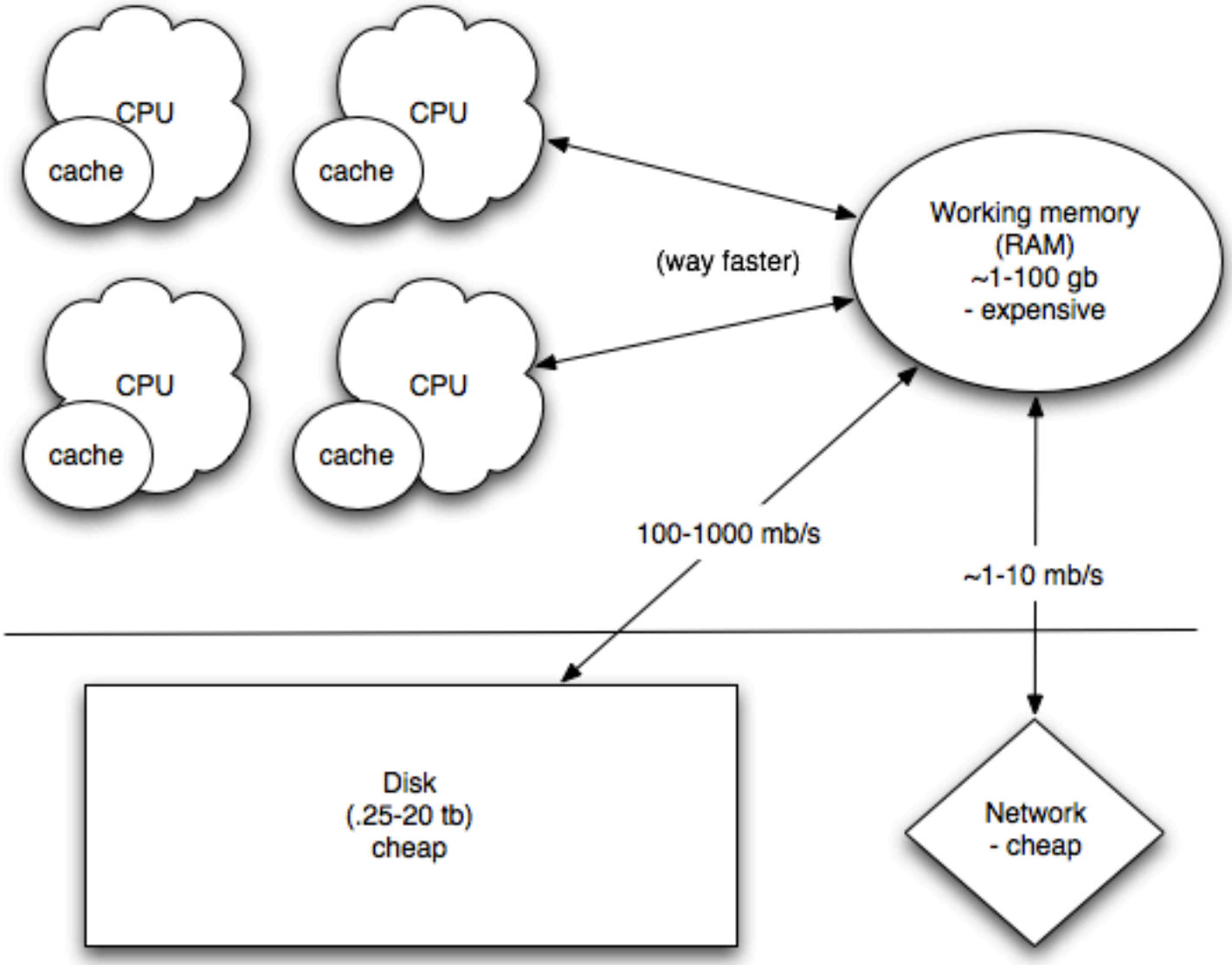
Mapping is embarrassingly parallel.

You need to calculate individual overlaps.

Assembly is not.

You need to calculate *all overlaps*.

Communication between CPUs is slow; this is main factor in splitting up tasks.



None of this is the #1 problem you will face with bioinformatics.

Here is the #1 problem:

How would you know if your answer was right or wrong?

Controls

- Just as with experiments, you can put negative and positive controls in your bioinformatics.

- e.g. with BLAST,
  - Do you see expected matches with the parameters and database you're using?

- Positive controls are often easier than negative, in "discovery-driven" science…

Internal controls

- Molecules and sequences for which you have expectations.

- "I know this gene comes up, based on qPCR. I expect to see it in my mRNAseq."

External controls

- Does the whole process work?
- "I can reproduce what this other person/ lab did, with their data, when I use my own software."
- This is much more rarely done…

Black box nature of algorithms

- When you listen to a computational biologist explain their clever algorithm…
- …it's a mistake to think that they necessarily know what's going on.
- Software is full of bugs and unintended consequences.

Tracking the process

- How do you know that your software *today* is doing the same thing it was doing last month? Or last year? Or in the hands of that other graduate student?

- There are some tools & techniques for dealing with change in software. We'll talk about them next week.

Concluding thoughts

- Every step of the process needs to be critically thought about and controlled.

- Choice of algorithms can be important, but depends on your problem: the convenient tool in your toolbox may not be well suited to your problem.