Problem Solving With Large Clusters

What's the problem, and what resources do we have?

Game plan for today:

Structure of the course

Overview of parallel and distributed computing

Quick intro to distributed file systems

First steps with MapReduce & Hadoop

http://www.cslu.ogi.edu/~bedricks/courses/cs506-pslc/

Grading: Students will be graded as follows: 20% final project; 20% assignments; **60% participation** (including paper presentations). This will largely be a "seminar-style" course, and most sessions will revolve around student-led discussions of papers. Everybody should come to class having read the papers and prepared to participate in the discussion.

- Focus: Conventional vs. distributed algorithm
- Problem: What is the problem? Why is it important?
- Background: What are the conventional algorithms? You may ignore the specifics of the application area.
- Distributed Algorithm: Details, assumptions and advantages
- Evaluation: Experimental paradigm, corpus
- Results: Outcomes, analysis and discussion

Game plan for today:

Structure of the course

Overview of parallel and distributed computing

Quick intro to distributed file systems

First steps with MapReduce & Hadoop

The problem:

Many things we might want to do with computers take a long time.

Why?

Trivial answer: they require the computer to do a lot of work.

The problem:

This can be for two main reasons:

1. We are working with a lot of data
2. We have to do a lot of computations on each chunk of data.

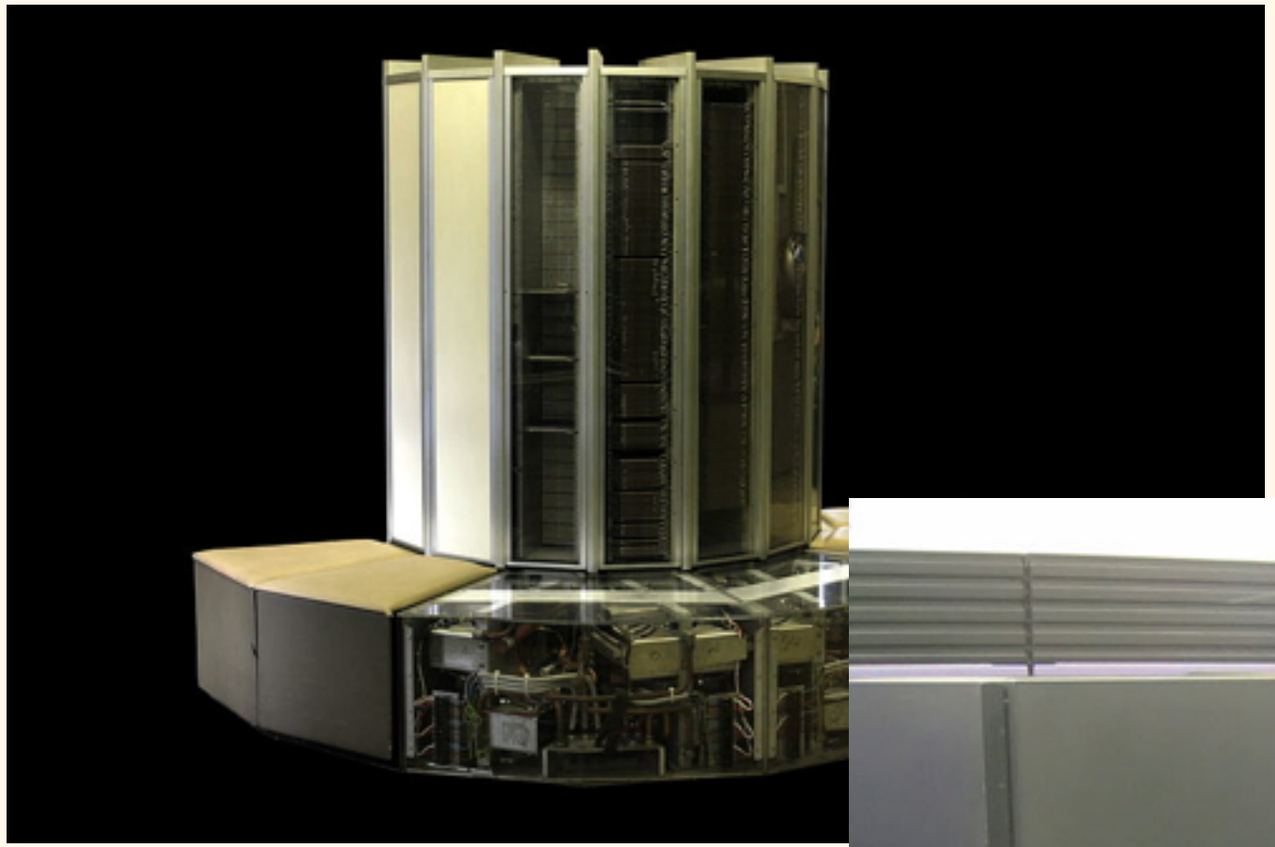
A common solution: split the work up!

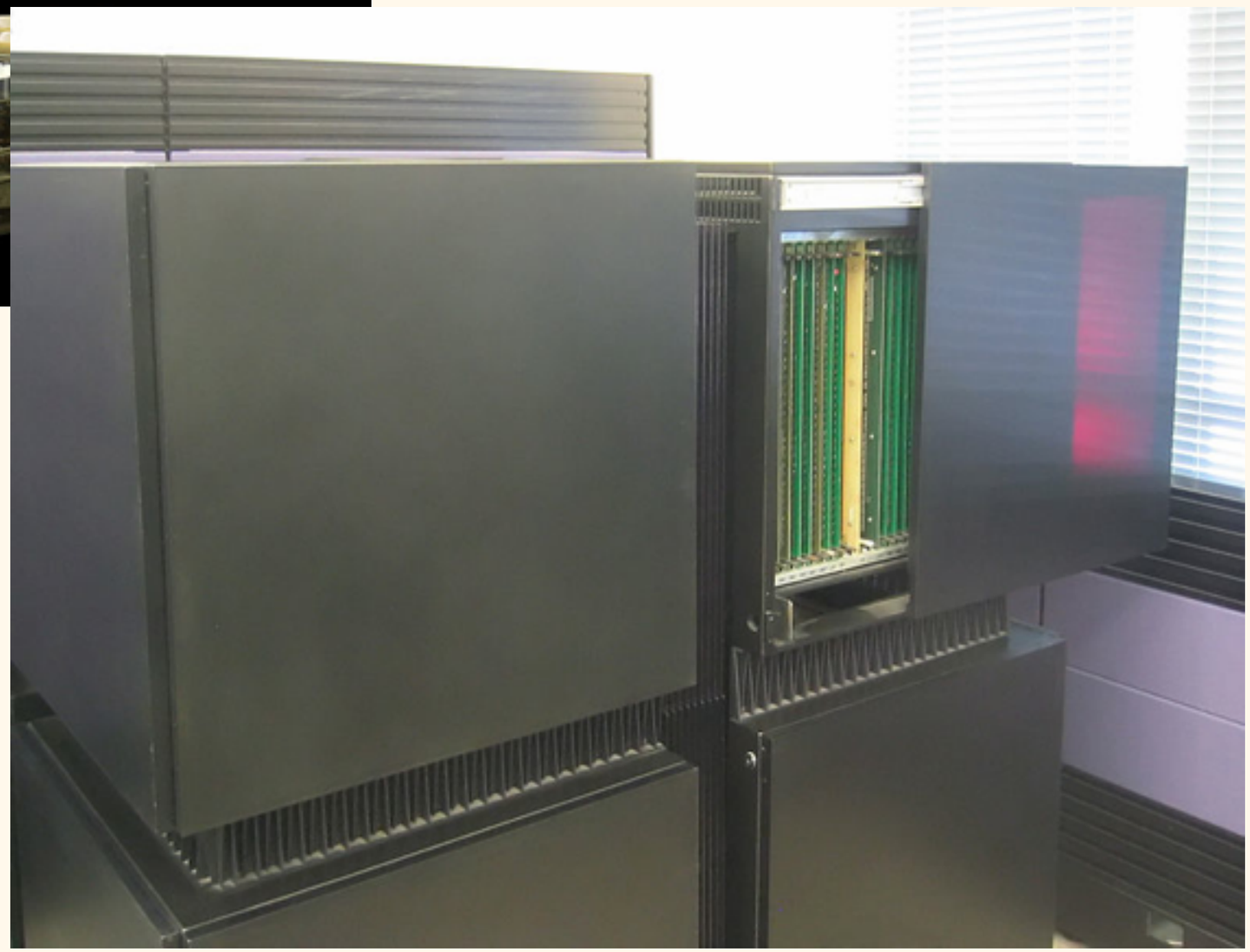
A common solution: split the work up!

1. Do parts of the computation in parallel (less work per processor)

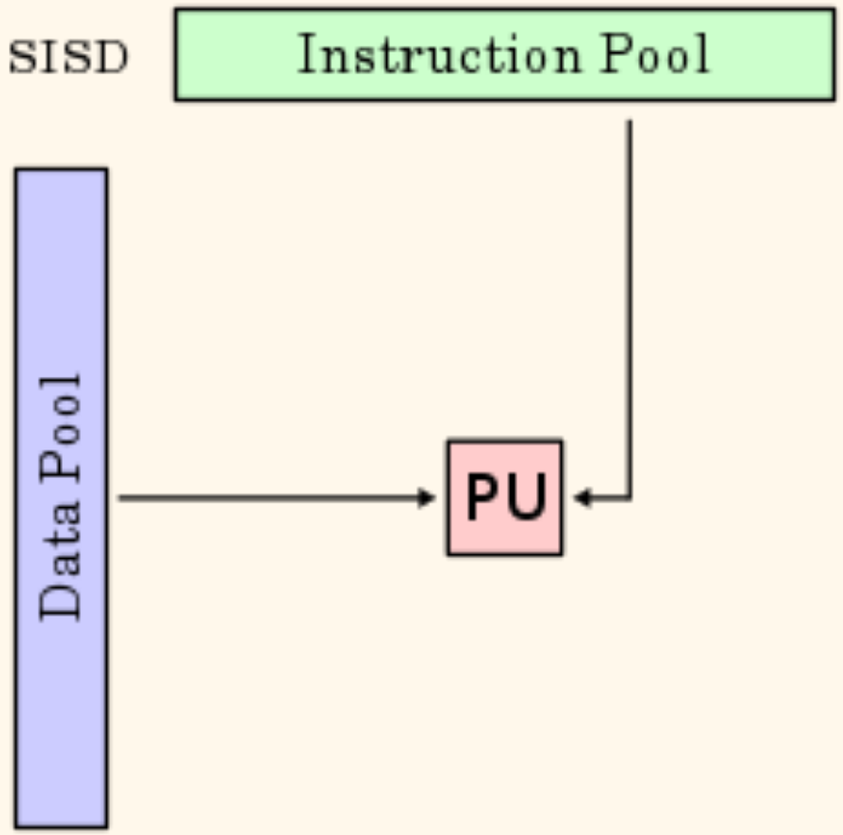
2. Split the data onto multiple computers (less data per processor)

Often, we (try to) do both!

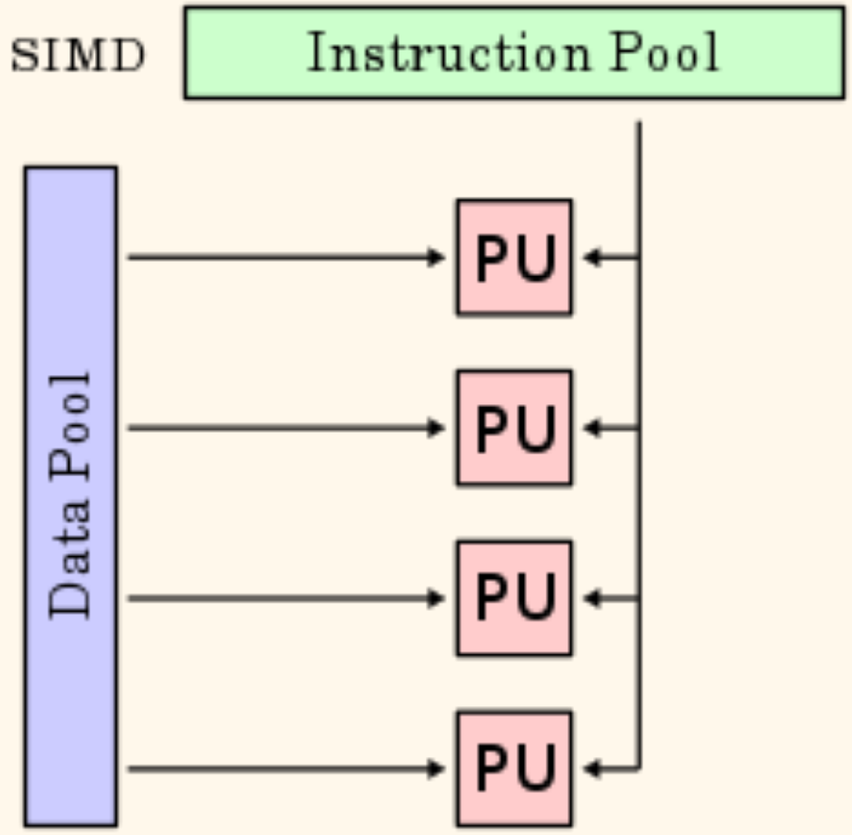




Ultimately, it all comes down to feeding instructions and data to processors:



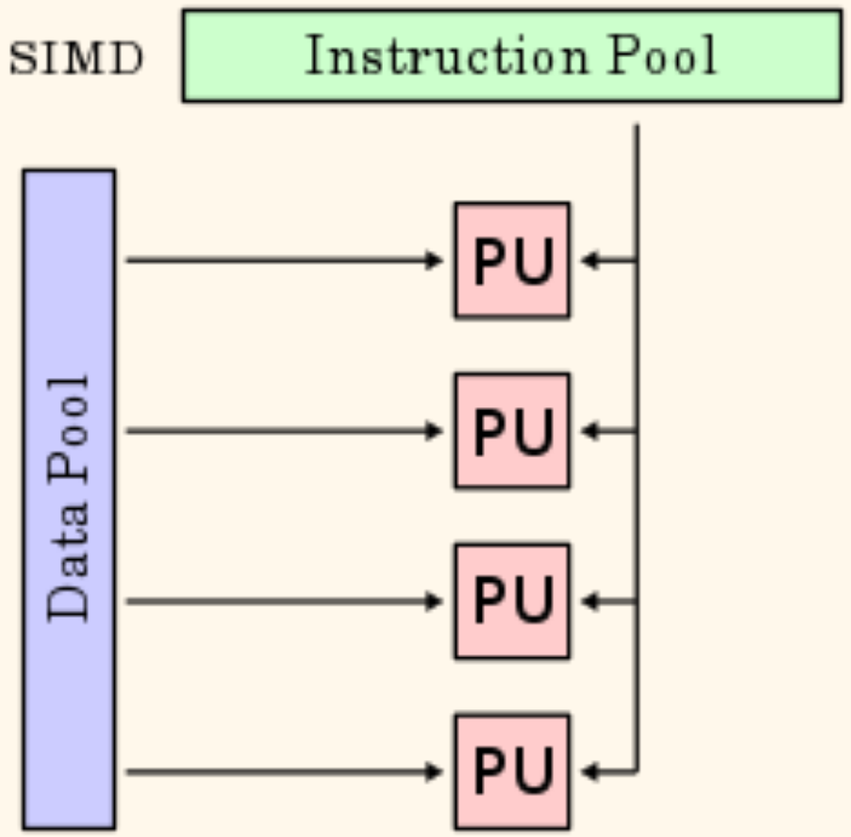
Single Instruction, Single Data



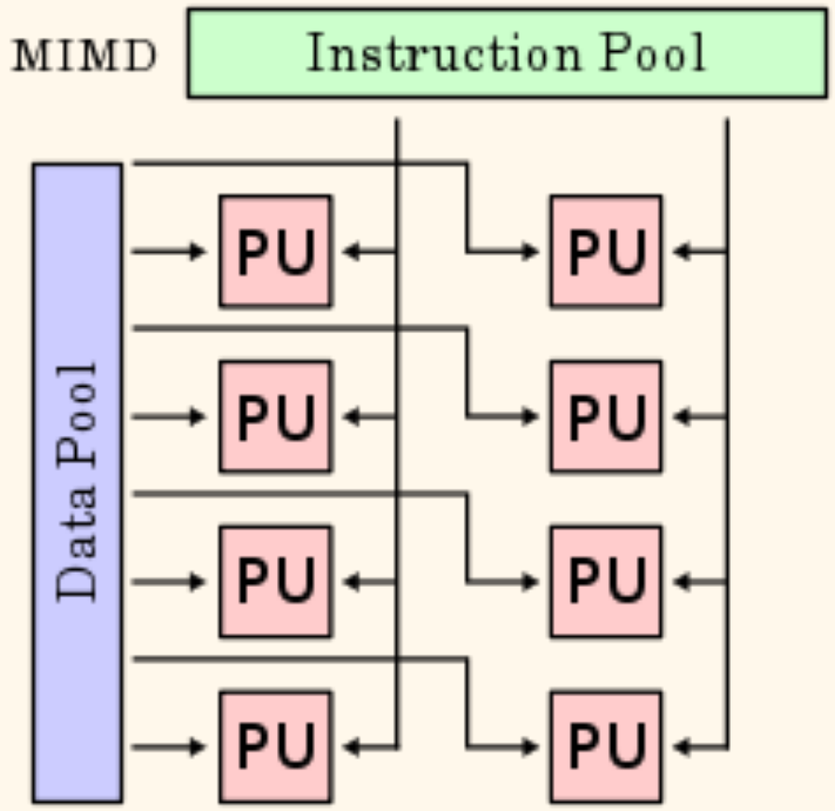
Single Instruction, Multiple Data

Most modern CPUs are SIMD (SSE3, etc.)...

Ultimately, it all comes down to feeding instructions and data to processors:



Single Instruction, Multiple Data

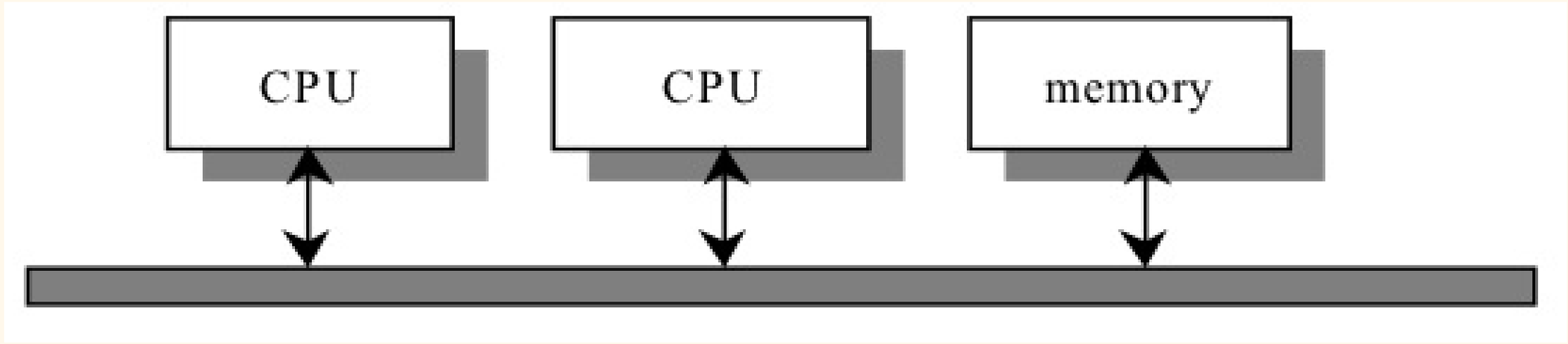


Multiple Instruction, Multiple Data

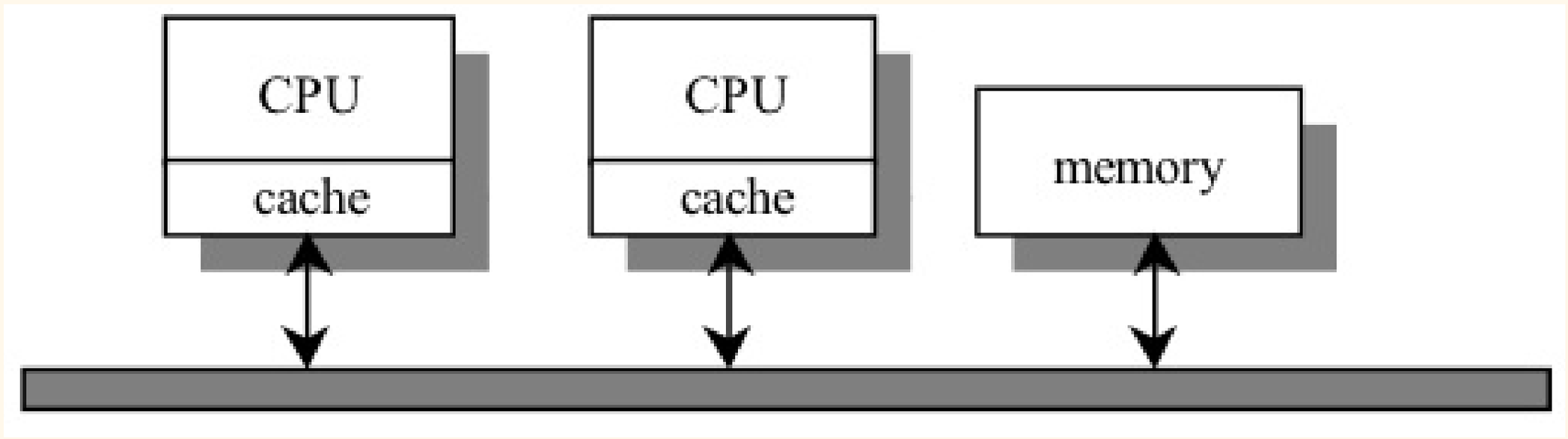
Some architectures are MIMD, e.g. Intel "Xeon Phi" and most modern parallel machines.

A single computer can have more than one CPU...

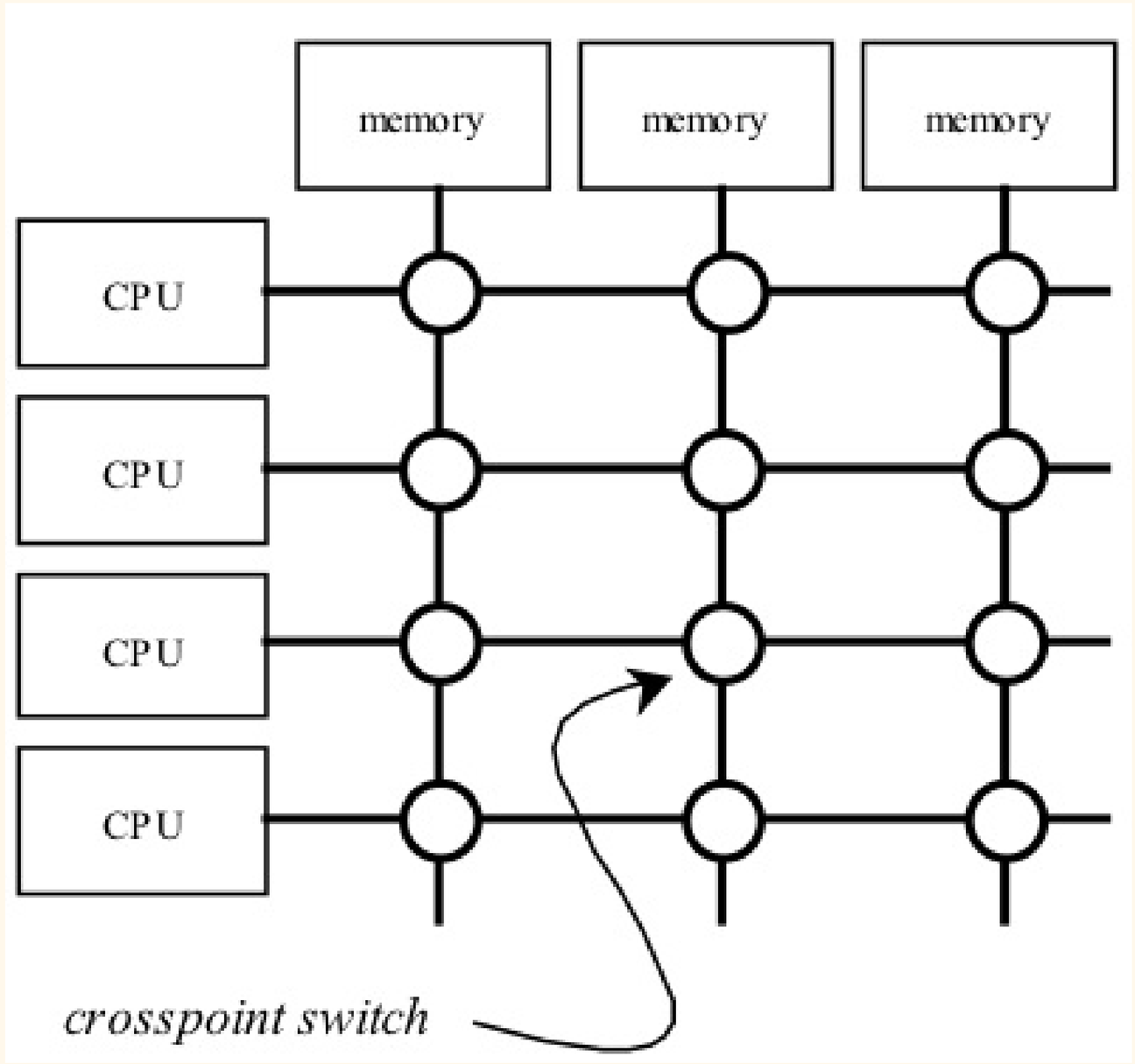
As a single bus:



As a single bus w/ cache:

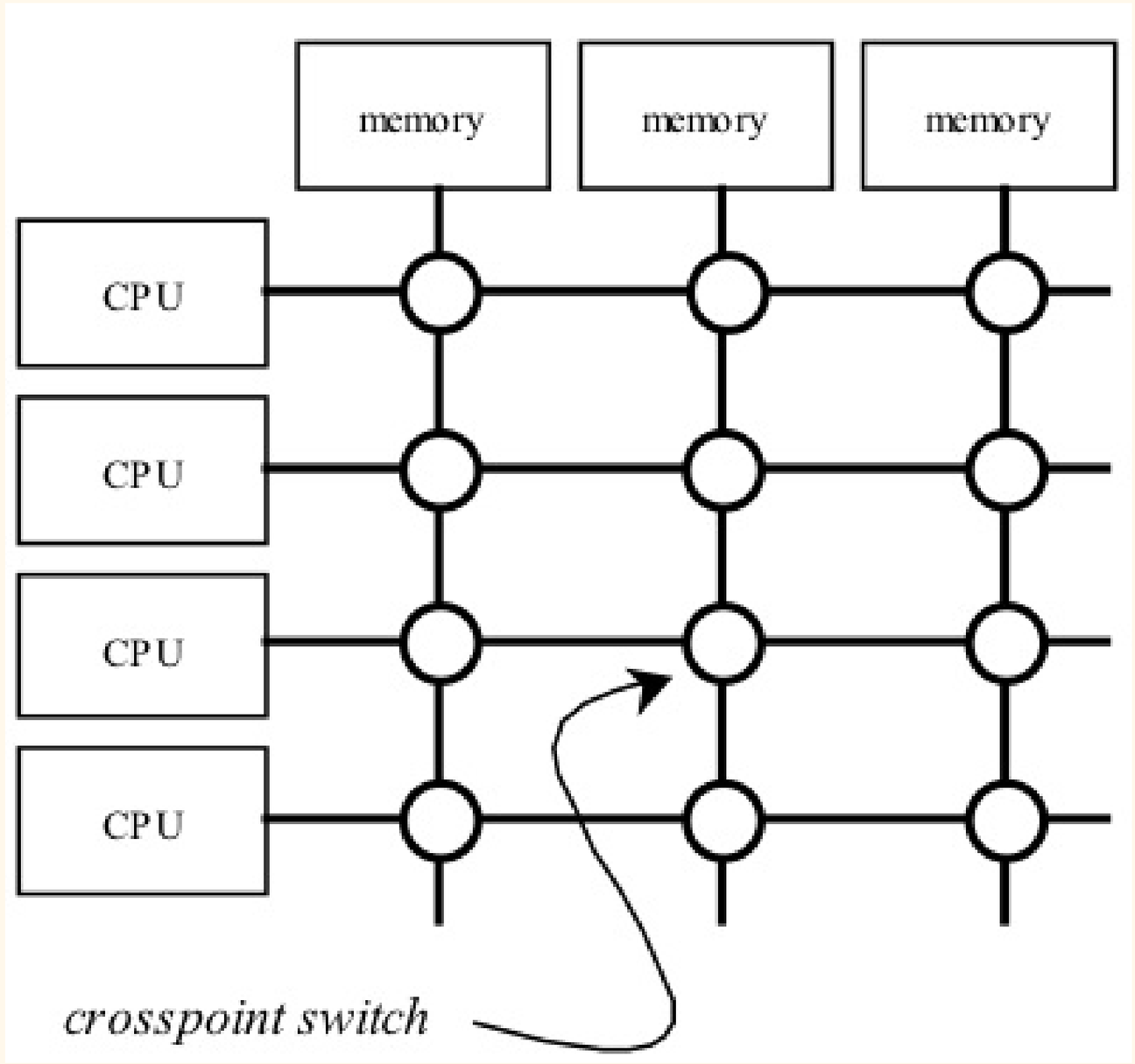


The question becomes: how to share memory across many CPUs?



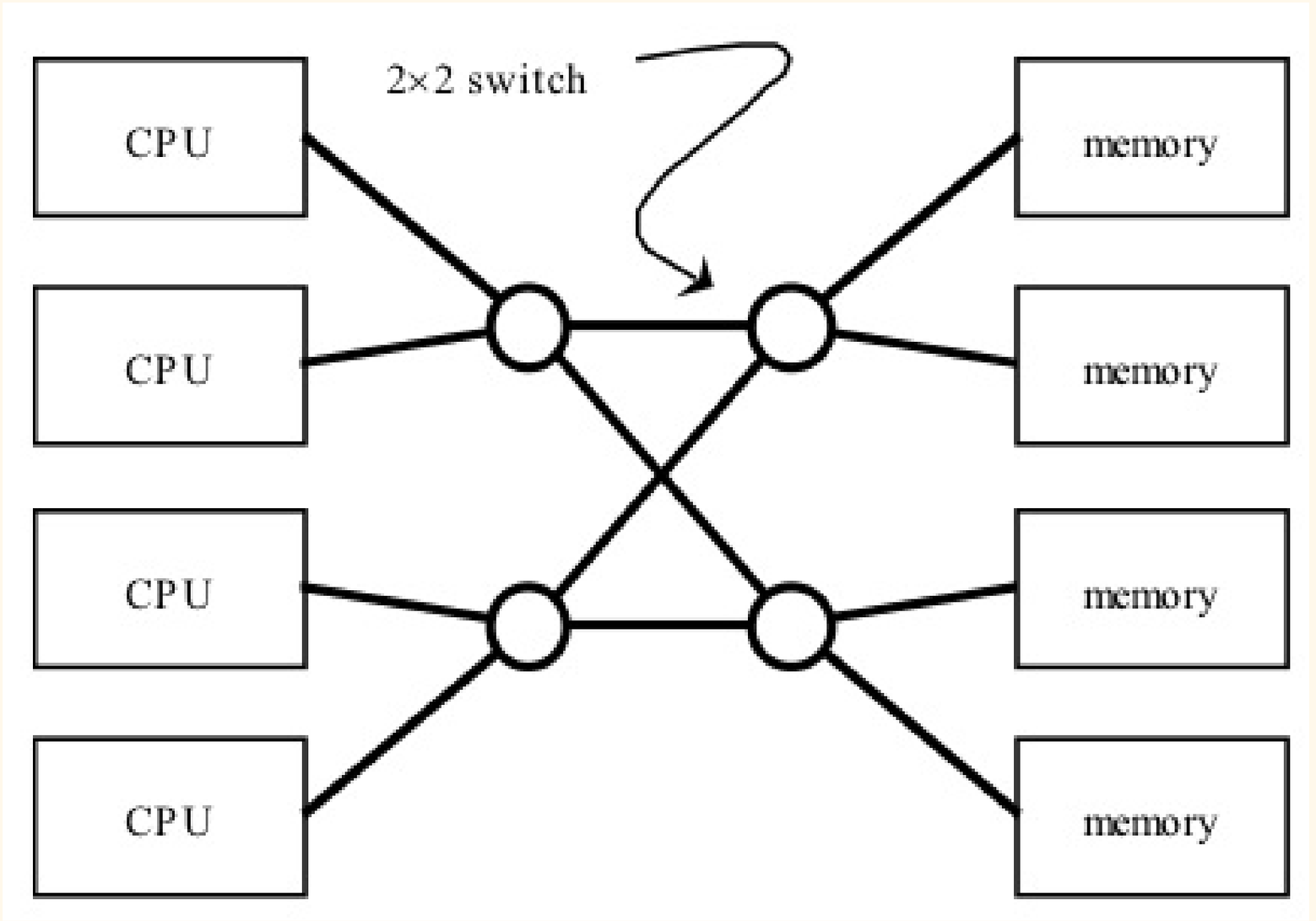
A crossbar topology is simple, but has many expensive* swtiches.

The question becomes: how to share memory across many CPUs?



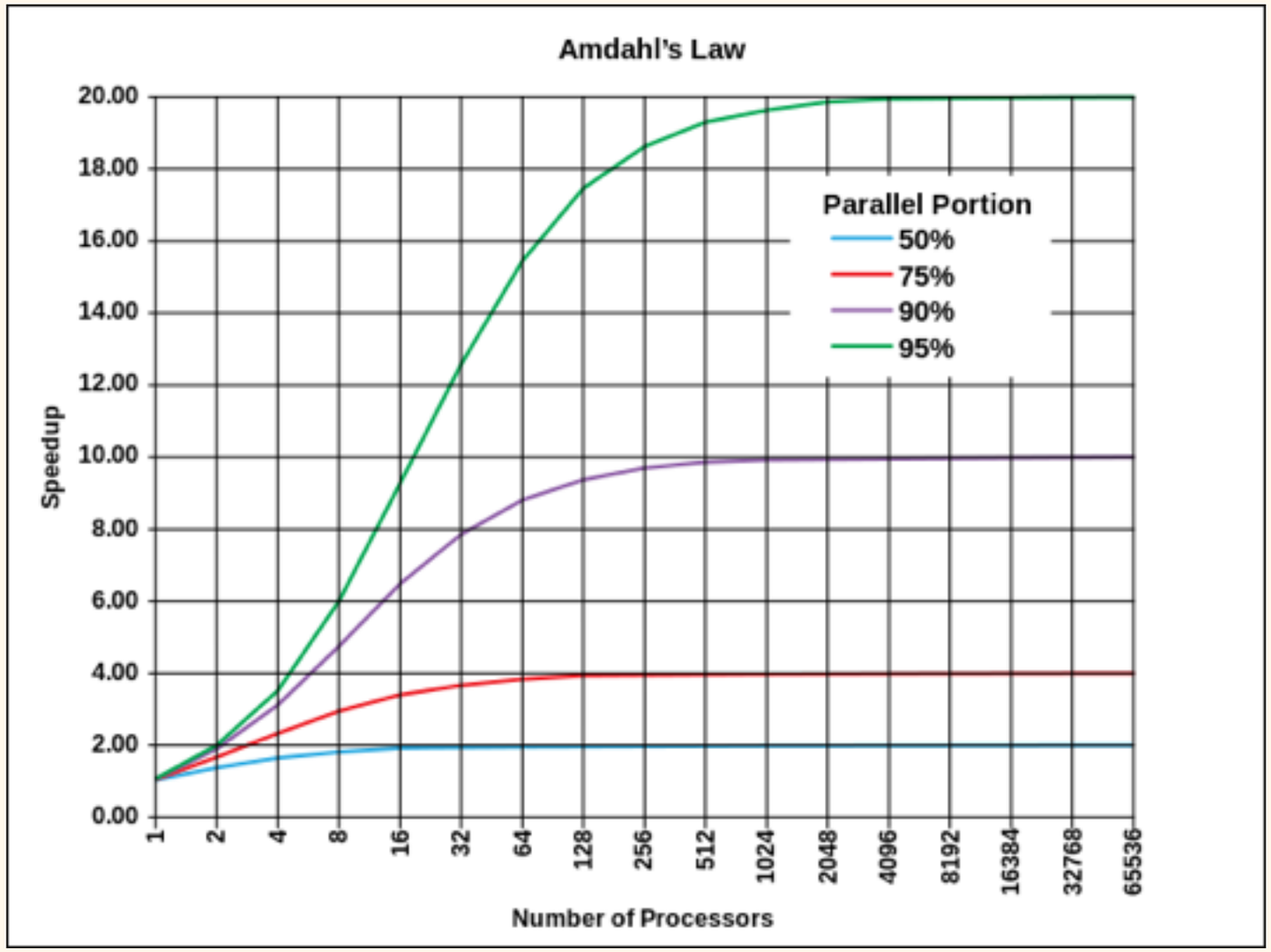
Expensive in terms of both time and silicon!

The question becomes: how to share memory across many CPUs?



Non-uniform memory access (NUMA) is a common compromise (Intel Nehalem, Westmere, etc.).

In the real world, there is never a linear speedup with an increase in CPUs.



$$T(n) = T(1)\left(B + \frac{1}{n}(1 - B)\right)$$

T: time taken

n: num. threads

B: proportion of algorithm that is strictly serial.

Amdahl's law states that the maximum speedup is related to the fraction of a program's work that is serial.

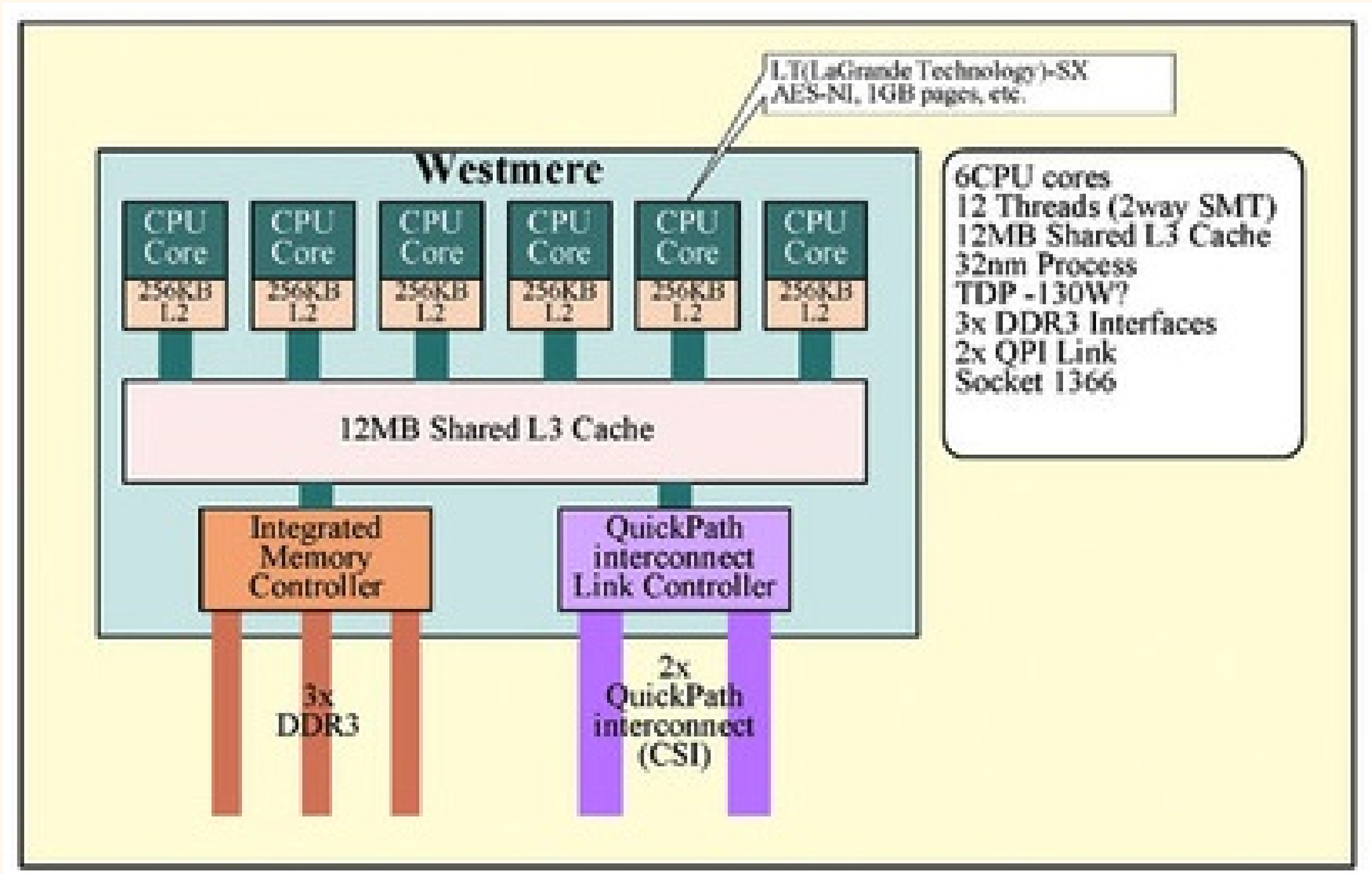
Other holdups include cache stalls, disk latency, etc. etc.

There are also the dreaded “fallacies of distributed computing” to keep in mind...

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

So, what do we have at CSLU?

The OHSU *bigbird* cluster uses Westmere CPUs; 2x per node.

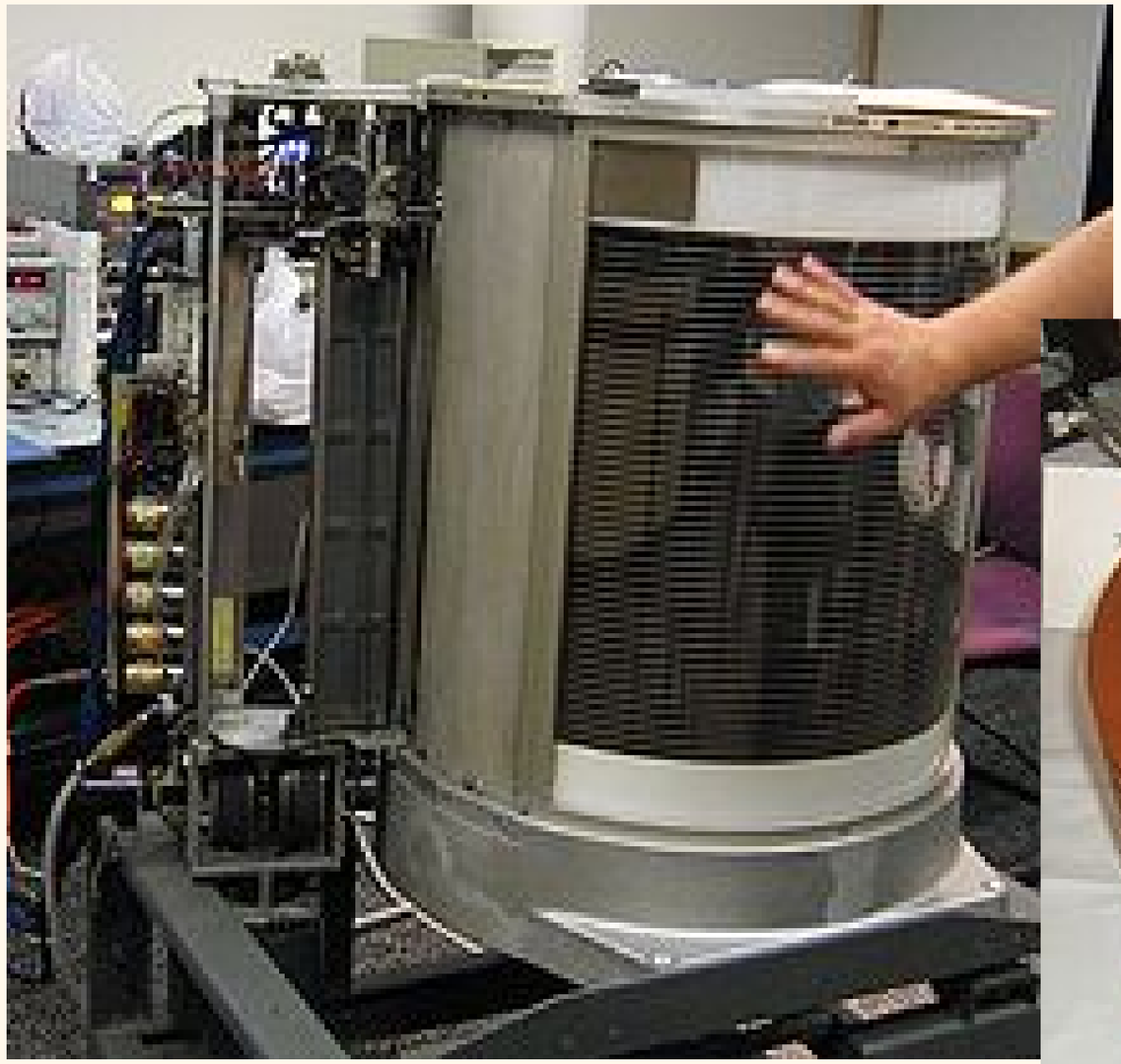


The bigbird cluster has 56 accessible nodes (`bigbirdXX.csee.ogi.edu`).

Each node uses Westmere CPUs; 2x per node (total of 24 logical cores).

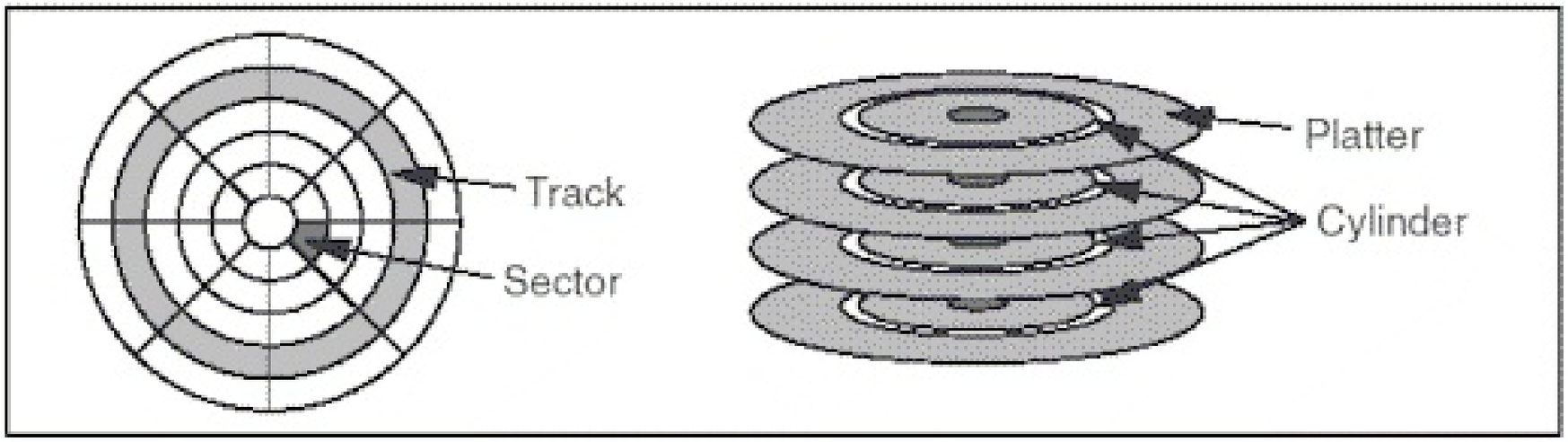
Each node has 48 GB of RAM, and all share a large distributed file system.

Let’s talk about storage.



Most hard drives are still mechanical.

- Hard disk has a number of disks
- Each disk segmented into tracks and sectors



- Disk speed: access time = seek time + latency time
  - Seek time: Time required to bring the head to the track
  - Latency time: Time required for the sector to reach the head
  - Platters spin about 7k to 15k rpm
  - Disk-to-buffer about 1Gbits/s, depends on track

Modern SSDs avoid these problems, but introduce others: Cost, limited life-span, etc.

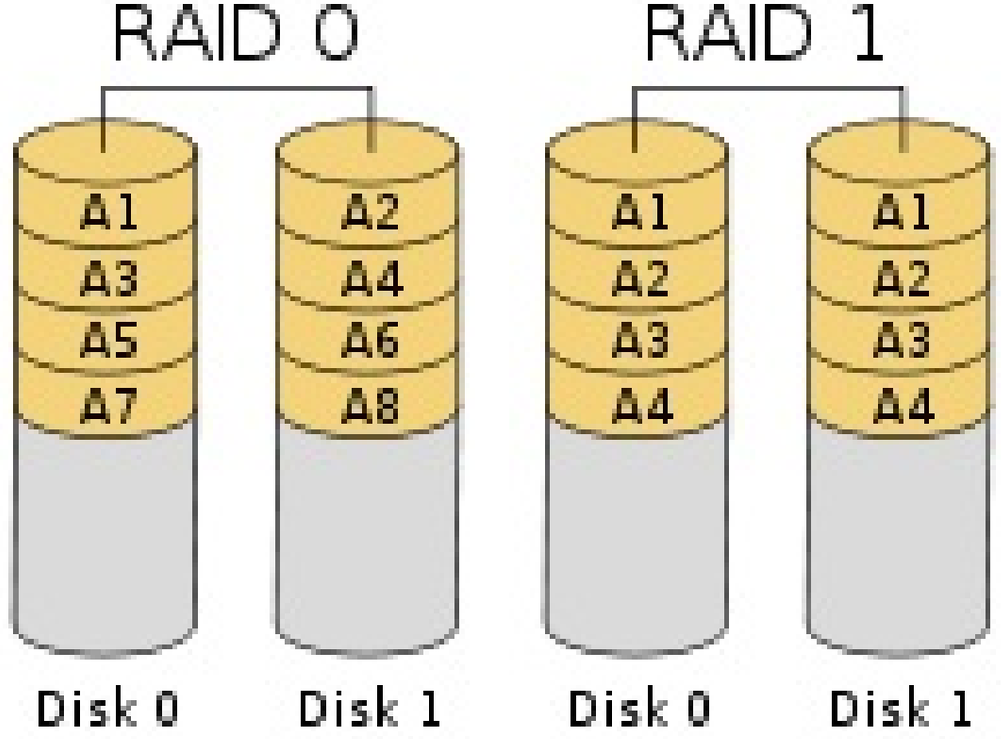
There are many consequences of the mechanical nature of hard disks:

Reading/writing a small number of large files is far faster than reading/writing a large number of small files.

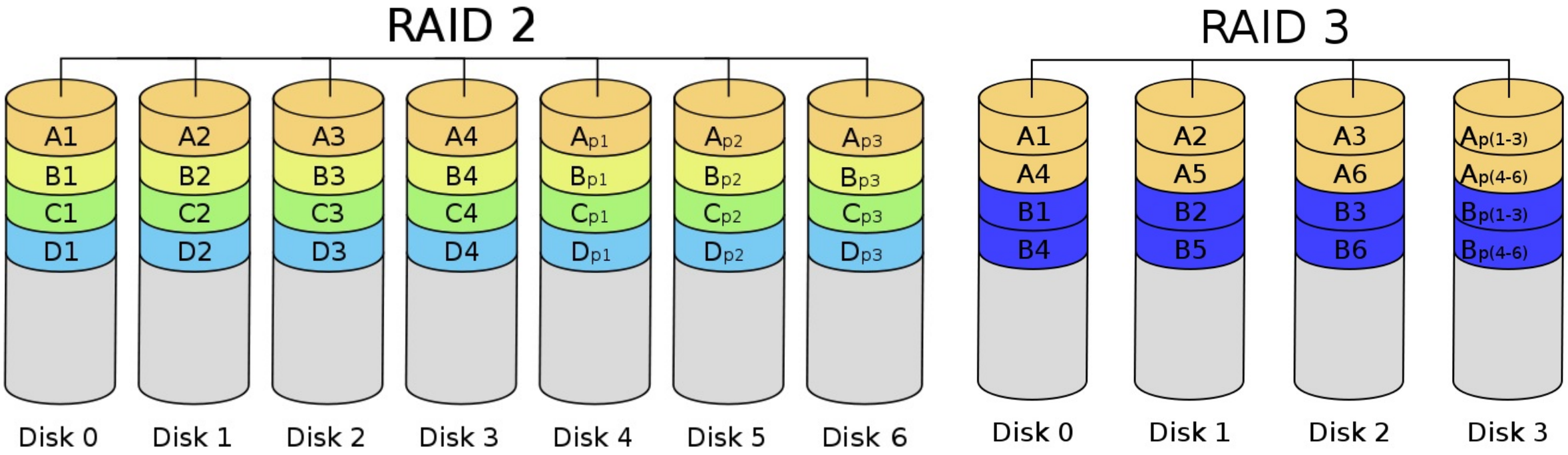
+Δ Moving parts -> +Δ things that can break.

(Patterson, Gibson and Katz, 1987)

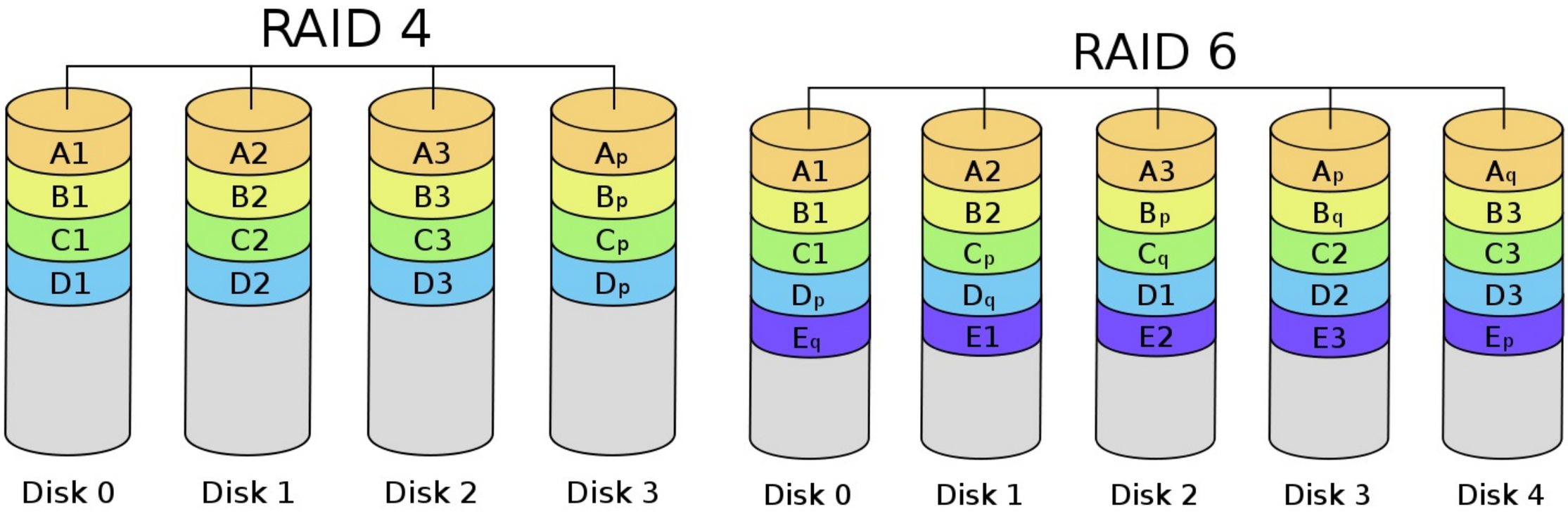
- Cost-effective to build capacity with many cheaper disks
- Divide the file into stripes, saved on independent disks
- Better performance by putting all the disks to work
- Compensate for higher failure rates with redundancy or parity
- RAID0: block level striping, zero redundancy, read nX
- RAID1: full mirroring, read nX, write 1x



- RAID2: bit-level, parity, sync-ed spindles
- RAID3: byte-level, parity, sync-ed spindles

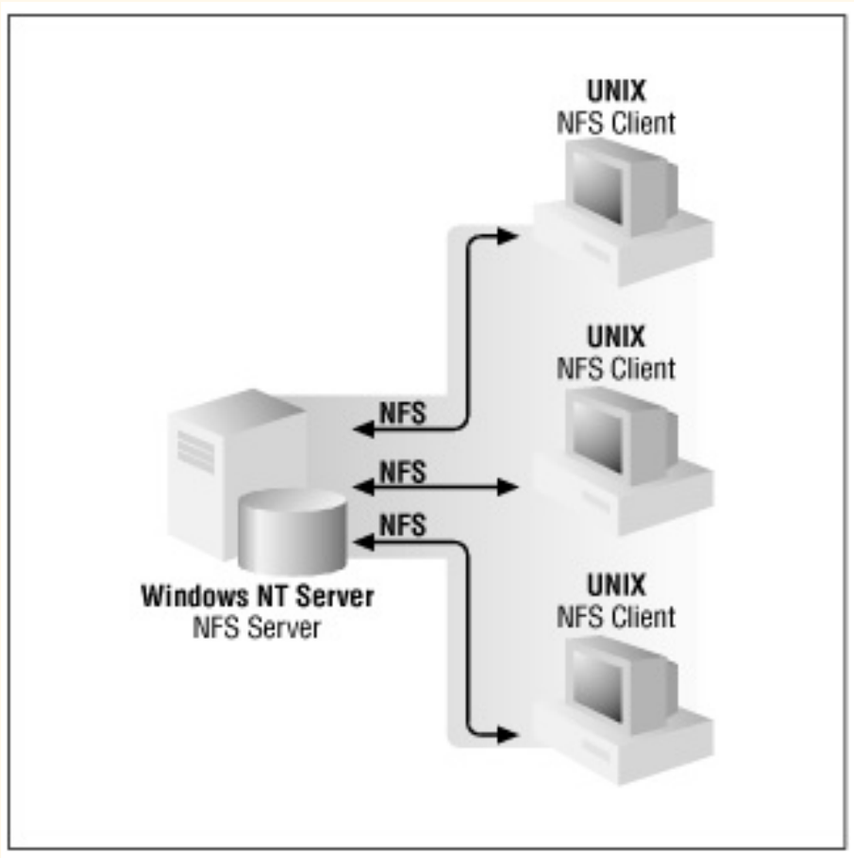


- RAID4: block-level, dedicated parity
- RAID6: block-level, doubly distributed parity



That's all well and good if you've only got one machine...

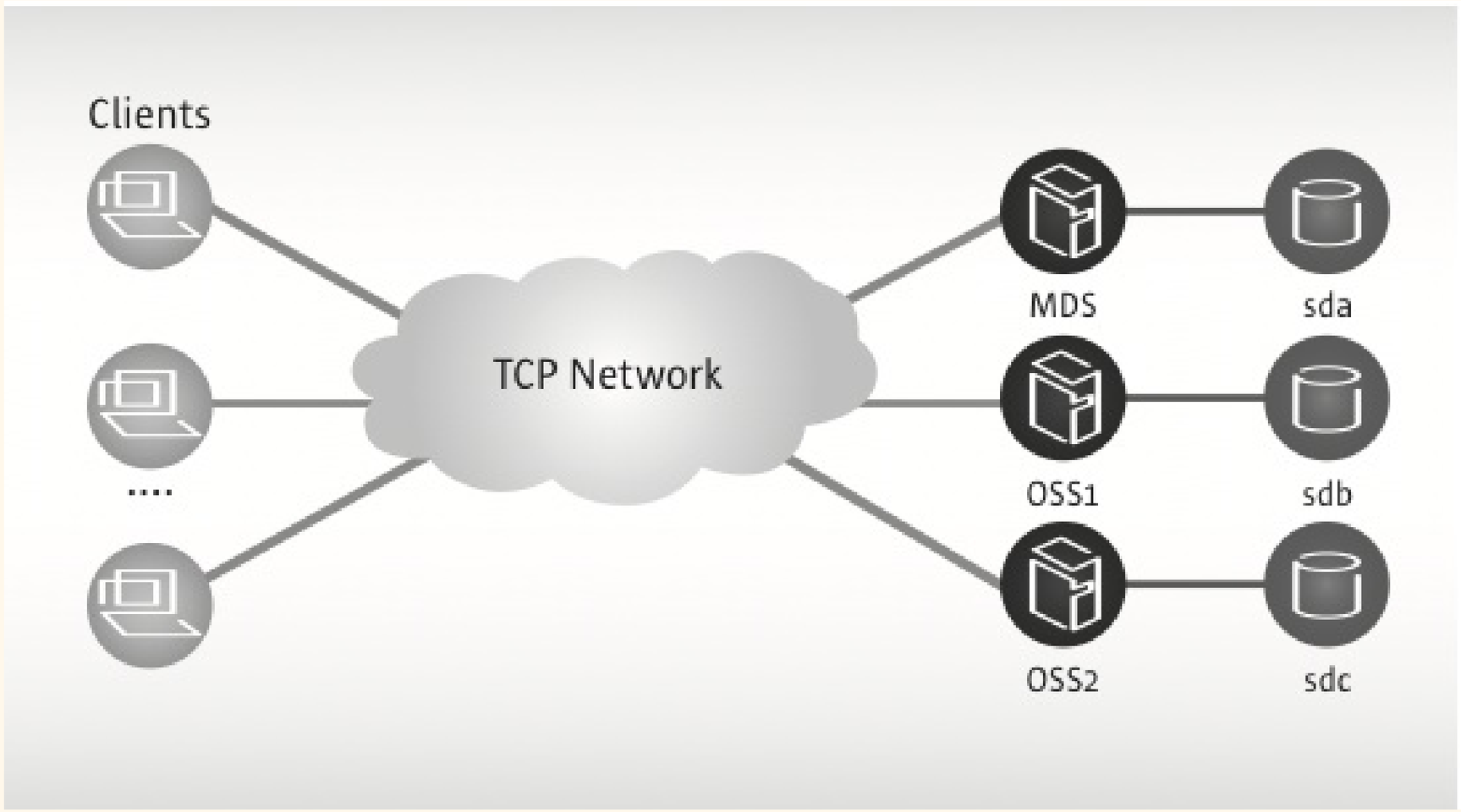
... but what if you need to share a disk array with more than one machine, over a network?

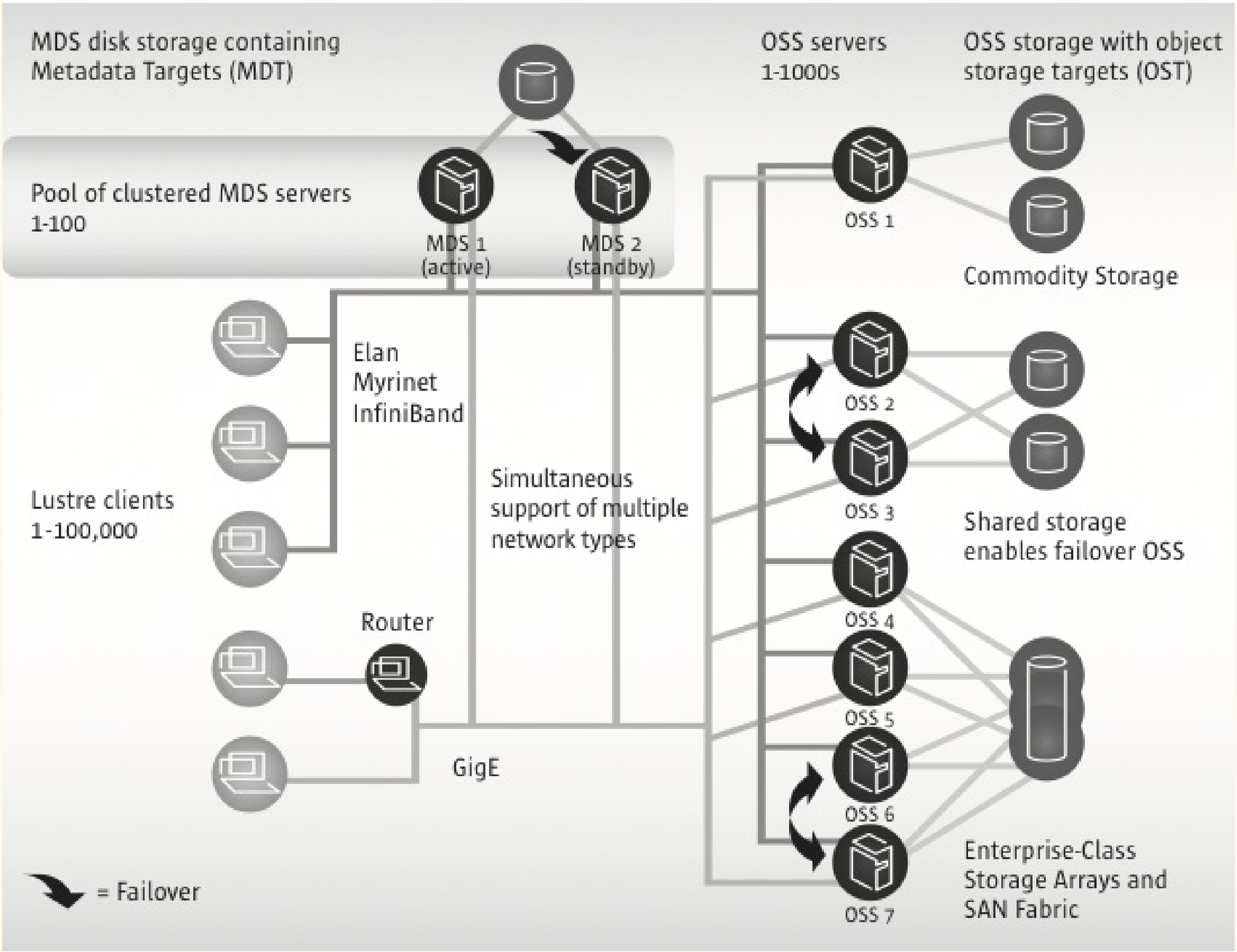


NFS gives file-level access, with server-side caching and coherency.

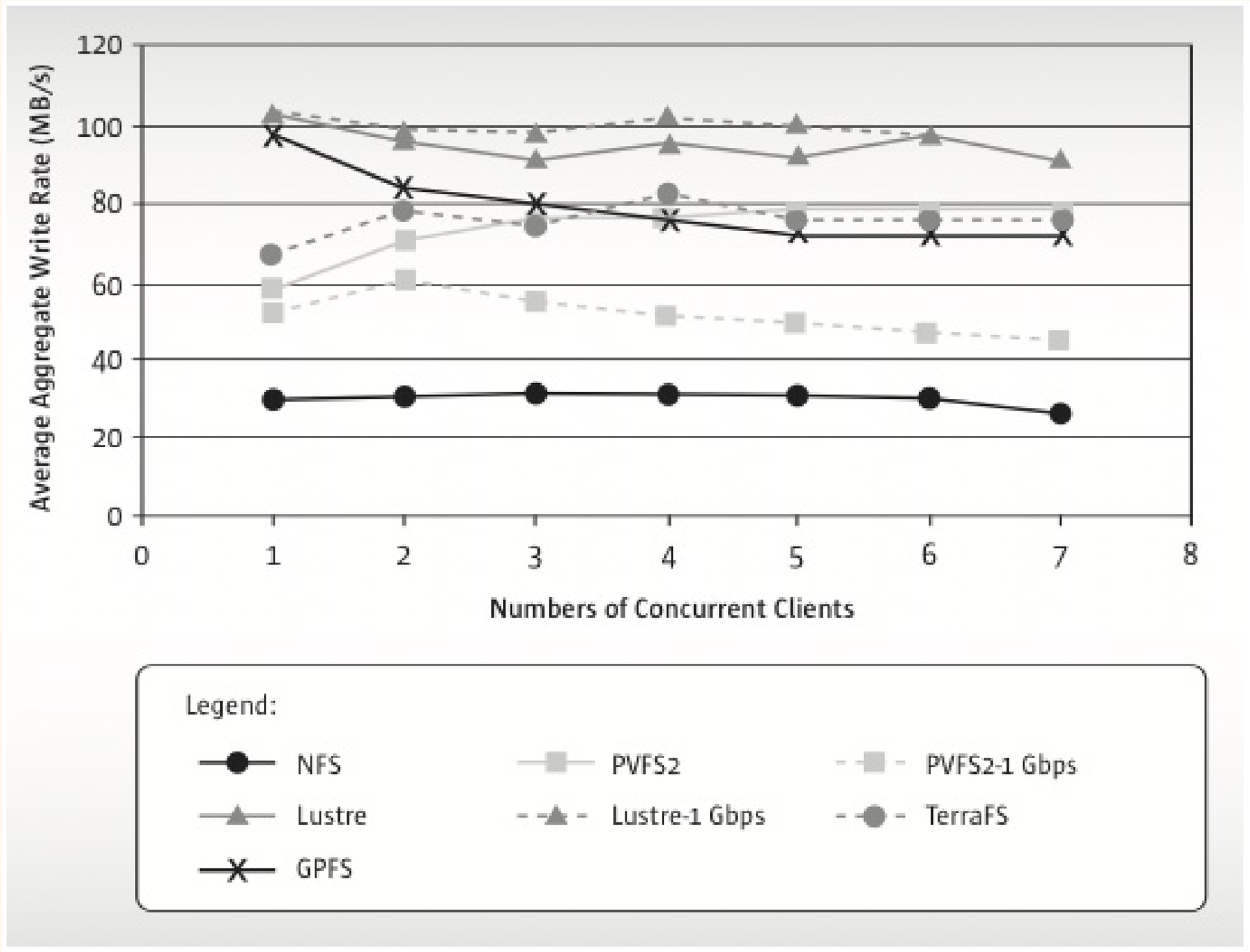
Other network file systems provide block-level access, etc.

Our cluster uses the Lustre distributed network file system:

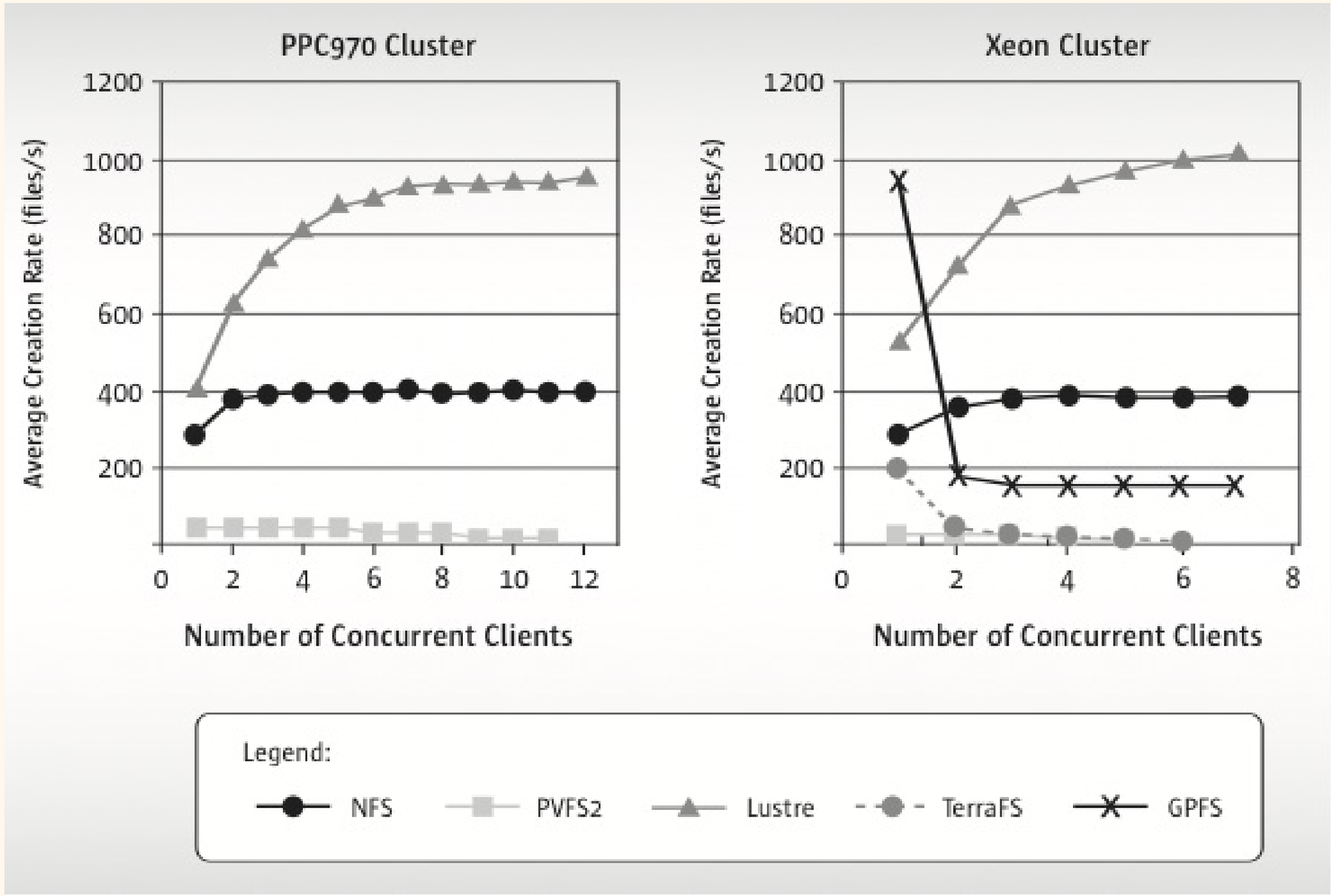



MDS disk storage containing
Metadata Targets (MDT)
Pool of clustered MDS servers
1-100
MDS 1
(active)
MDS 2
(standby)
OSS servers
1-1000s
OSS storage with object
storage targets (OST)
OSS 1
Commodity Storage
Elan
Myrinet
InfiniBand
Lustre clients
1-100,000
Simultaneous
support of multiple
network types
OSS 2
OSS 3
Shared storage
enables failover OSS
OSS 4
OSS 5
OSS 6
OSS 7
Router
GigE
Enterprise-Class
Storage Arrays and
SAN Fabric
= Failover

Lustre holds up well under concurrent load:



Lustre holds up well under concurrent load:



18 TB Lustre system (/l2/users/userid)

- MDS: 2 x 4-core CPUs @ 3GHz, 16 GB

- MDT: 15k rpm, 400GB x 15 (6 TB)

- ODS (5): 2 x 4-core CPUs @ 3GHz, 16 GB
- ODT: 7.2k rpm, 1 TB x 6
- Note: Limited backup

Future upgrades are planned!

We'll be talking more about distributed file systems/stores throughout the course.

Systems such as Lustre are extensions of traditional file systems...

... but for truly large data collections, the file system model can be inadequate.

File systems such as the Google File System (GFS) and the Hadoop File System (HDFS) can offer more scalability and reliability.

GFS was invented at Google to store their web search index:

- Fault-tolerance
- Implemented at user-level, provides location-awareness
- Assumptions: high sustained bandwidth > low latency
  - Large files are typical
  - Large streaming reads and small random reads
  - Large sequential writes and small random writes
- No file or directory aliases (hard or soft links)
- Clients can concurrently append to a file efficieintly

Google File System

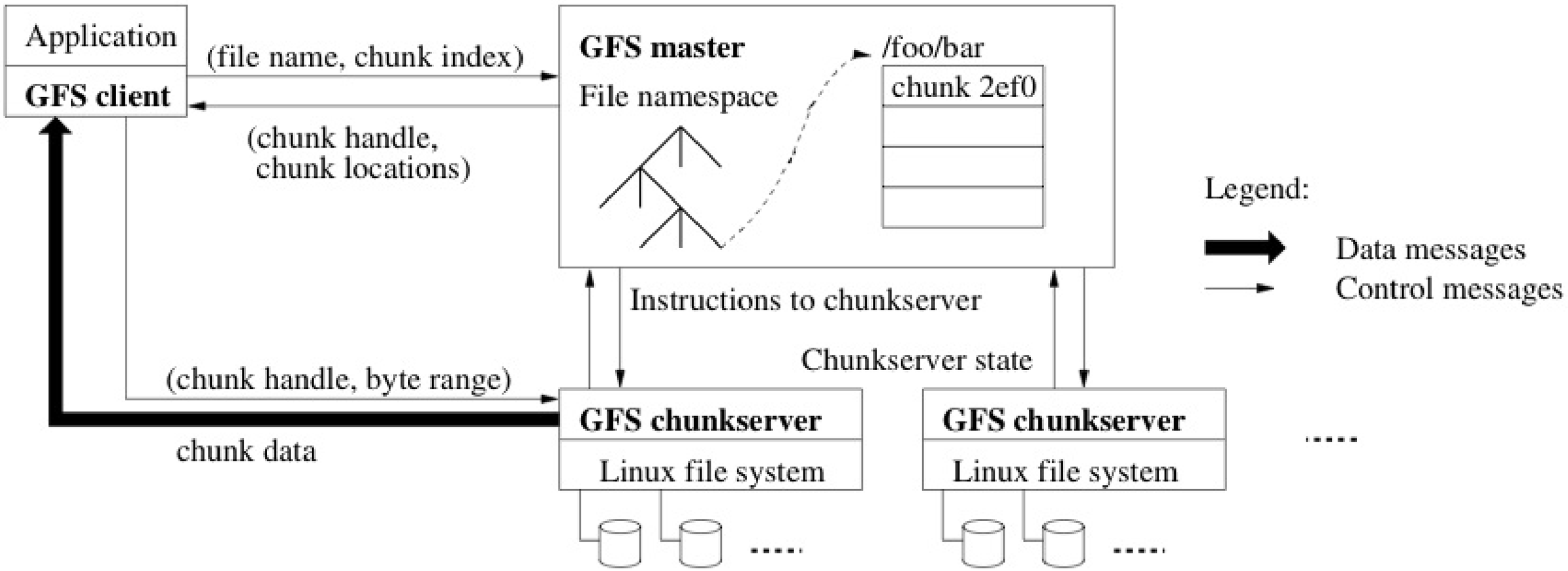


Figure 1: GFS Architecture

- Single master, multiple chunkservers
- Files are divided up into chunk, ID-ed by an addres
- Chunkservers manage chunks like local files
- Chunk data replicated for reliability

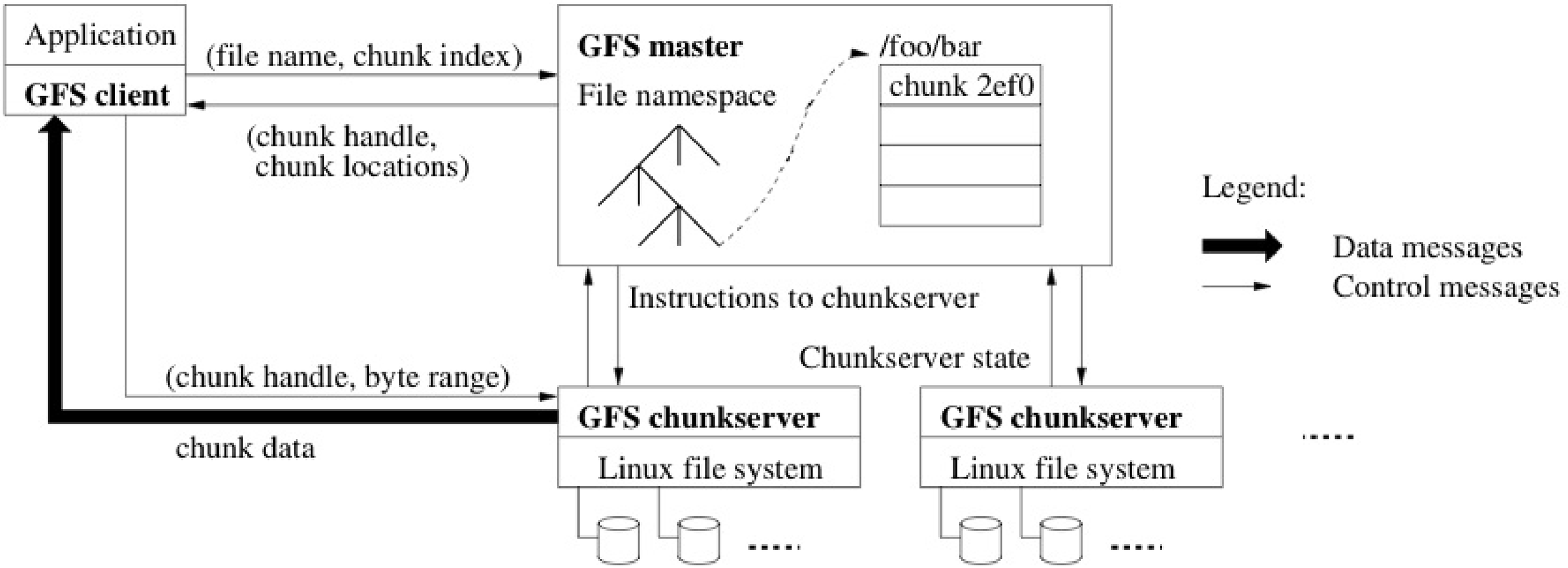


Figure 1: GFS Architecture

HDFS is essentially an open-source implementation of GFS.

We'll be talking more about distributed file systems/stores throughout the course!

Switching gears:

1. Do parts of the computation in parallel

2. Split the data onto multiple computers

There are many different ways to split up a problem to multiple computers...

One common paradigm is MapReduce.

MapReduce parallelizes serial programs by splitting them into two parts:

A “mapper”, which runs in parallel across an entire data set; and

A “reducer”, which operates on the result of the mapper.

An implementation of MapReduce (e.g., Hadoop) provides an environment for scheduling and running mappers and reducers.

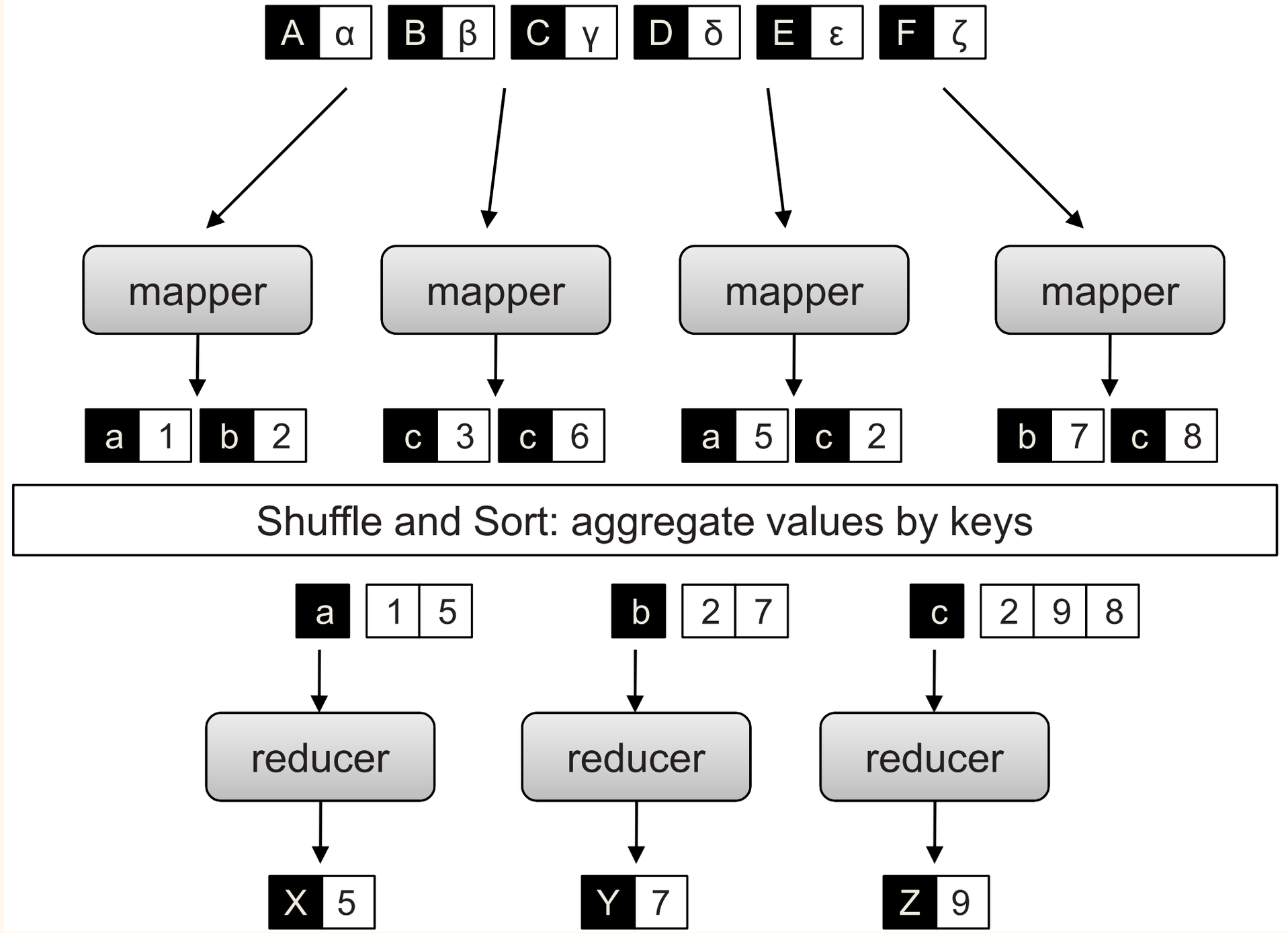
This includes not just job control, but also data flow management.

The basic unit of operation in a MapReduce program is a <*key,value*> tuple.

Mappers read tuples and produce new tuples...

... Reducers process the *aggregated* results of mappers, and produce new tuples.

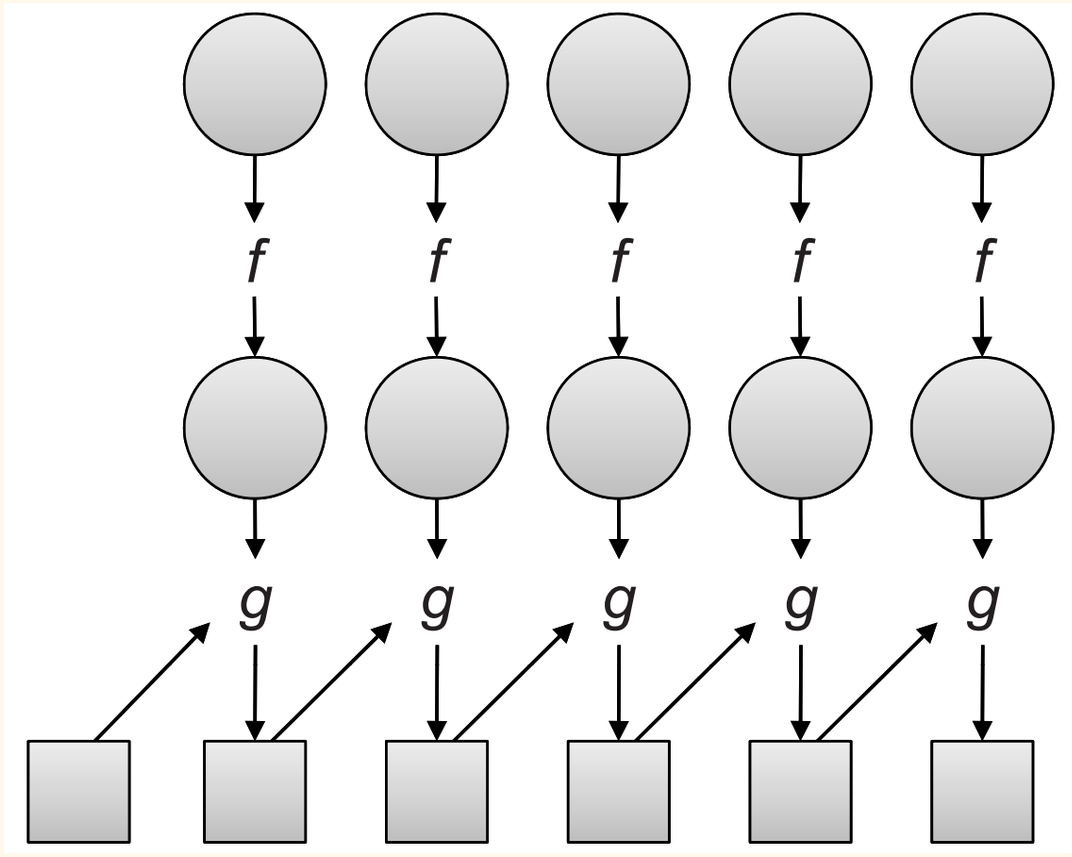
The MapReduce runtime takes care of managing which tuples get sent where.


A α
B β
C γ
D δ
E ε
F ζ
mapper
mapper
mapper
mapper
a 1
b 2
c 3
c 6
a 5
c 2
b 7
c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8
reducer
reducer
reducer
X 5
Y 7
Z 9

The MapReduce model makes several working assumptions:

1. Assume failures are common
2. Move processing to the data
3. Process data sequentially (avoid random access)
4. Hide system-level details
5. Seamless scalability

Quick aside: Where does the name come from?



Most functional programming languages define “map” and “reduce” (aka “fold”) operators.

Most functional programming languages define “map” and “reduce” (aka “fold”) operators.

“map” takes a function and applies it once to each item in a list:

```
map(f, [a, b, c, d]) =[f(a), f(b), f(c), f(d)]
```

```
def square(n):
    return n * n

nums = [1,2,3,4]

squares = map(square, nums) # [1,4,9,16]
```

Most functional programming languages define “map” and “reduce” (aka “fold”) operators.

“reduce” recursively applies a function to each item in a list.

```
reduce(f, [a, b, c, d]) = f(a, f(b, f(c, d)))

reduce(sum, [1, 2, 3, 4]) = sum(1, sum(2, sum(3, 4)))
```

```python
def r(a, b):
    return a + b

nums = [1, 2, 3, 4, 5, 6]

total = reduce(r, nums) # 21
```

In MapReduce land:

Mappers emit key-value pairs in parallel...

... which are then shuffled and sorted by key.

Tuples with the same key are passed to the same reducer...

... who then outputs its own list of tuples.

The MapReduce runtime makes sure that everything works the way it should.

"Hello World" in MapReduce: Word Counting

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)
```

```
1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

Problems with this approach:

1. Lots of key-value pairs flying around (one per word!)
2. Some reducers will have a lot more work to do than others (i.e., the one that has to add up “the”)

Partitioners and Combiners help avoid these problems by aggregating values at earlier steps.

Partitioners divide up the intermediate key space and assign keys to reducers...

... by default, by hashing the key and assigning modulo the number of reducers.

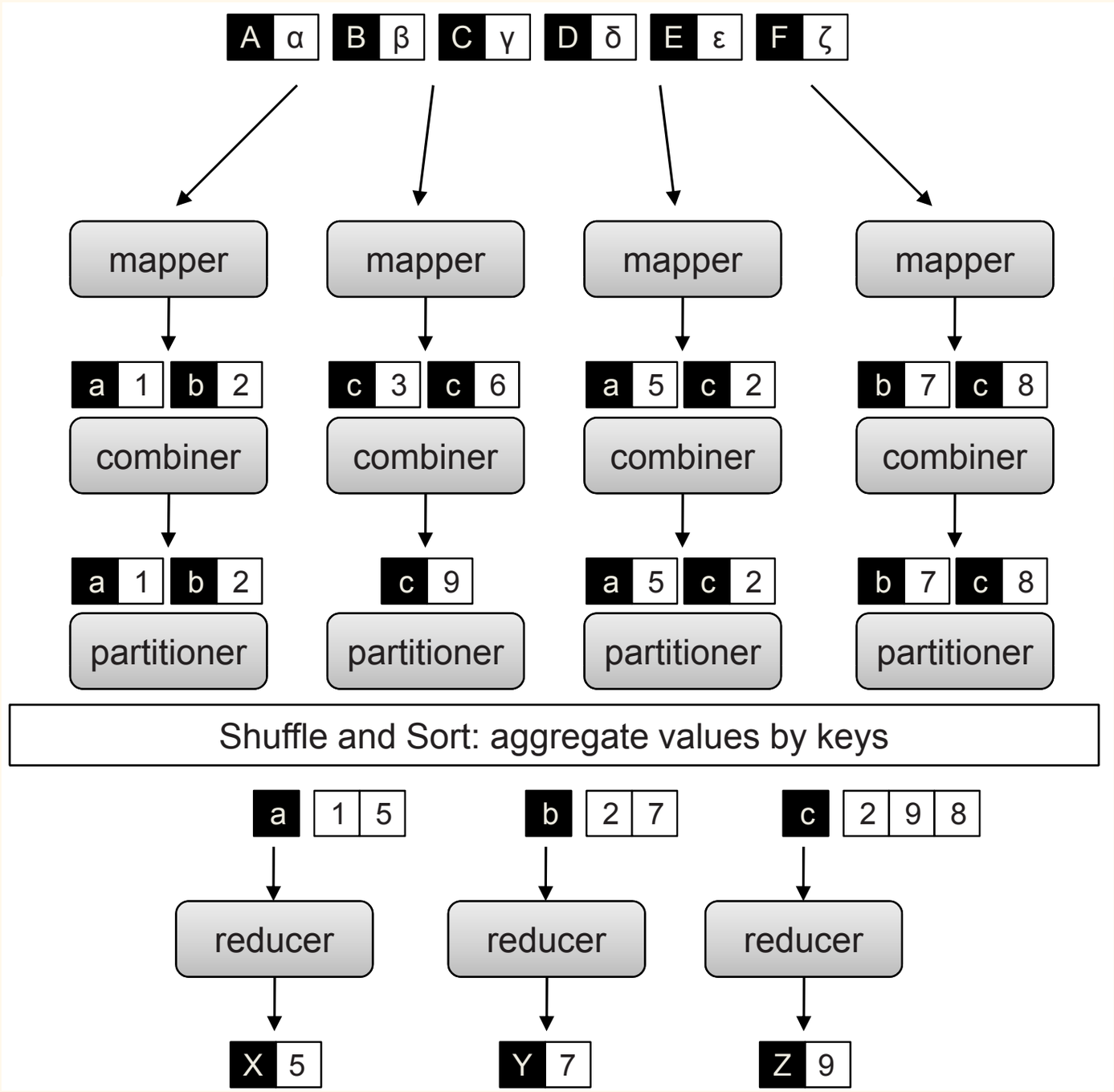
If needed, you can divide up the key space in other ways.

Partitioners and Combiners help avoid these problems by aggregating values at earlier steps.

Combiners are sometimes called “mini-reducers”, and operate on the output of individual mappers.

This pattern can result in major performance improvements!

Partitioners and Combiners help avoid these problems by aggregating values at earlier steps.



```
1: class Mapper
2:     method Map(docid a, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1                ▷ Tally counts for entire document
6:         for all term t ∈ H do
7:             Emit(term t, count H{t})
```

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1                ▷ Tally counts across documents
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

Things to keep in mind:

Tuples with the same key will be sent to the same reducer...

... but there is no way to specify, a priori, *which* specific reducer instance will get which key!

Combiners must therefore accept and emit data in the same format as the output of the mapper...

... and it is up to the runtime to decide how many times (or even if!) combiners will be run.

You have little/no control over:

On which node a mapper or reducer runs...

When a mapper or reducer starts/stops...

Which key-value pairs are processed by a specific mapper...

Which key-value pairs are processed by a specific reducer.

You *can* control:

What goes inside of keys and values...

Startup/shutdown code for mapper/reducer instances...

Preservation of state within a mapper/reducer instance across multiple input keys...

Sort order of intermediate key/value pairs, and therefore the order that a reducer encounters its data.

What is MapReduce “bad” at:

Anything that involves random access through an entire data set.

Thought experiment: How would you go about extending the word count program to compute maximum likelihood frequency estimates?

Most difficult MapReduce programming problems involve working around this limitation.

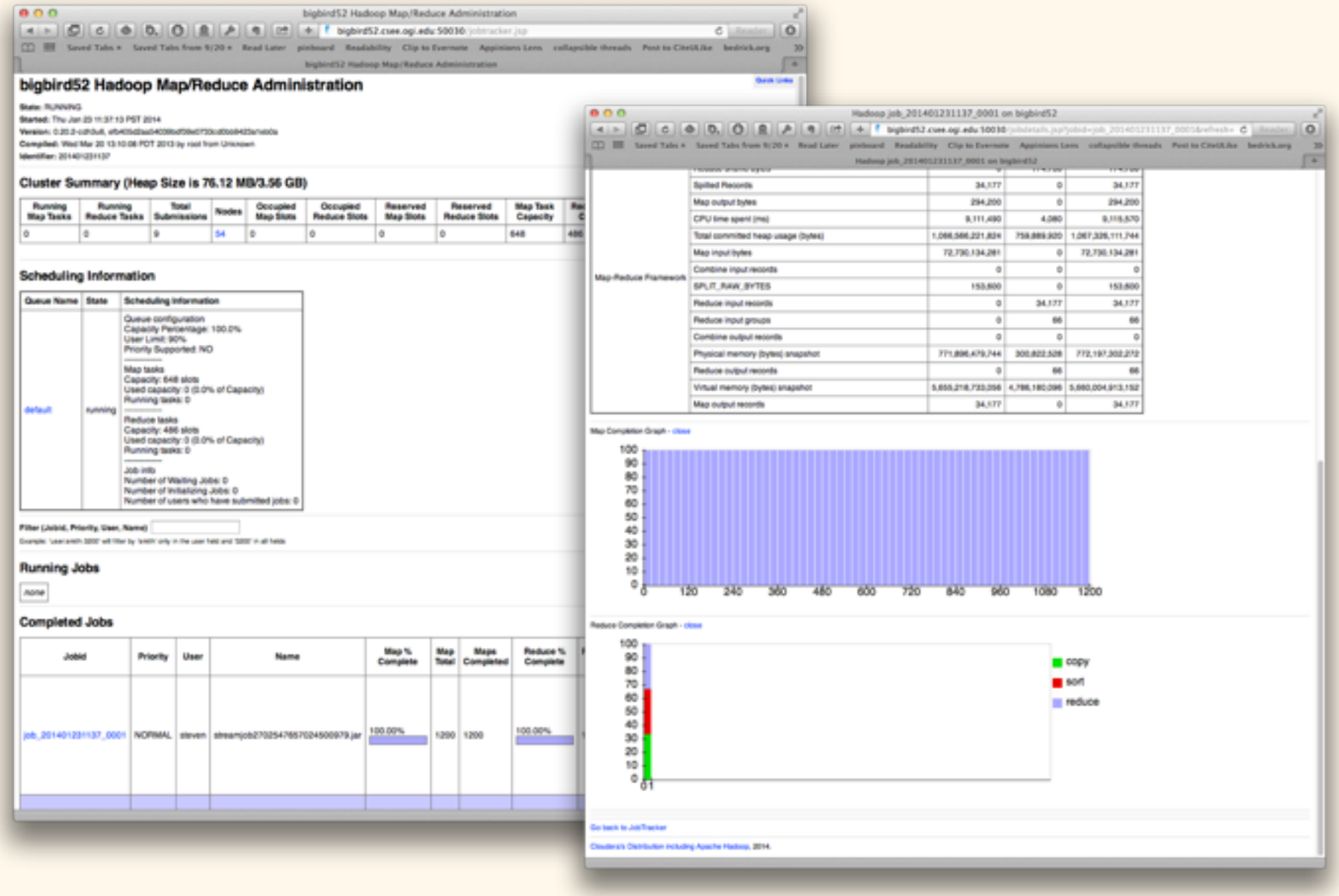
Our cluster runs the “Hadoop” open-source MapReduce implementation.

Hadoop has its own implementation of GFS (called HDFS); our cluster has ≈14 TB of HDFS storage (total, not available!).

Hadoop jobs can be run in a variety of ways; the two main ones are:

1. A native Java API, and
2. “Streaming” mode, in which mappers and reducers can be written in any language with STDIN/STDOUT.

Hadoop has a web-based control panel, in addition to command-line tools:



Let’s do a real example!

About the pubmed corpus:

66 years of MEDLINE (1946–2012) references.

Titles, abstracts, index terms, authors, etc.

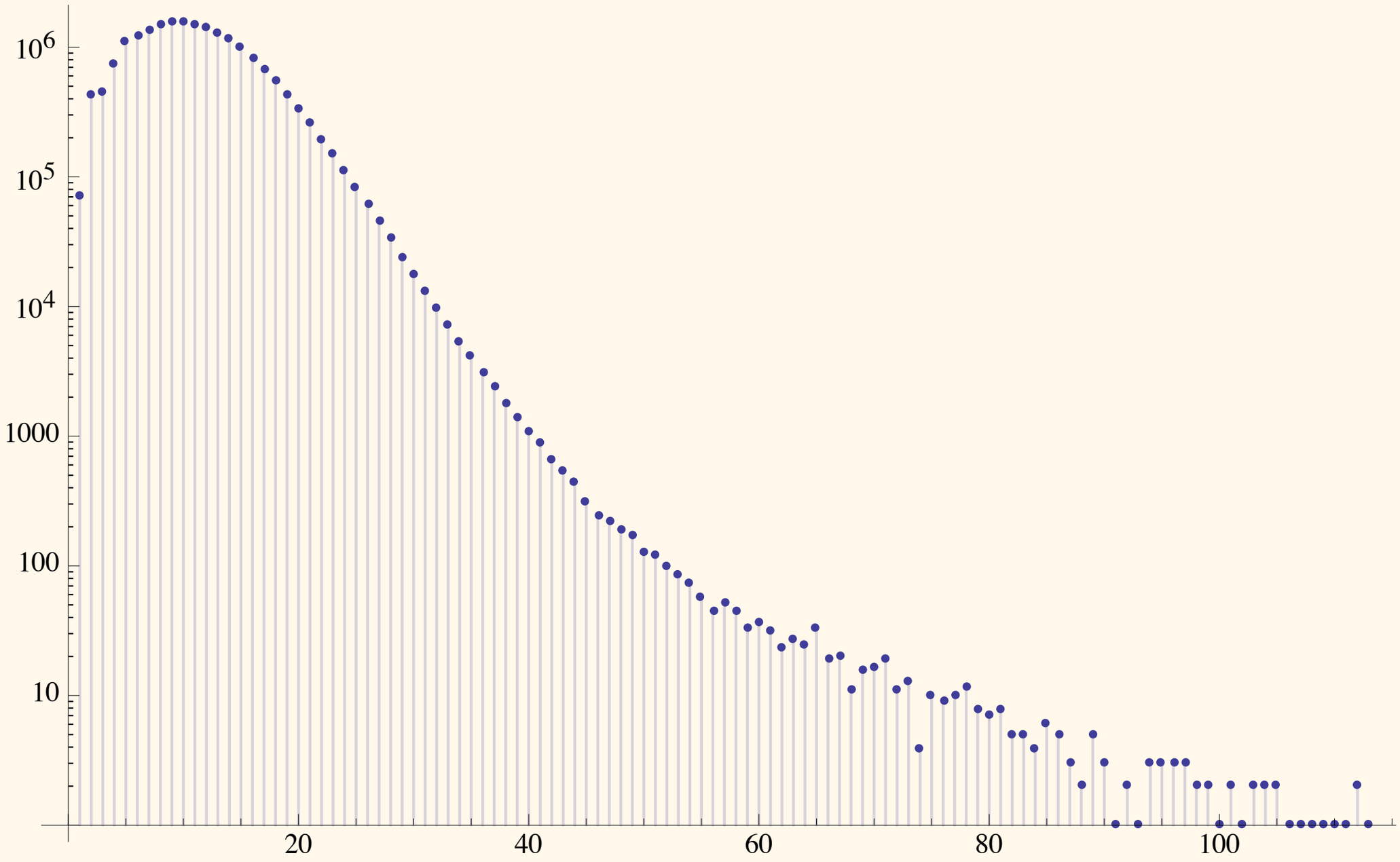
Stored in serialized JSON blobs.

Keys are PMIDs, values are JSON objects representing articles.

20.5 million articles.

Question: What's the distribution of article title length like?

Let's look at solving it using both the Java API and streaming mode.


10^6
10^5
10^4
1000
100
10
20
40
60
80
100