Welcome!

SciNet Parallel Scientific Computing Course
Aug 31 - Sept 4, 2009

Introduction to Parallel Computing

Course Overview, and The 'Big Picture'

The Course

SciNet Parallel Scientific Computing Course

Aug 31 - Sept 4, 2009

Main Goal

- Students arriving with scientific computing background should be able to leave and immediately *start* parallelizing their codes.

Parallel Computing is Necessary

- As computing capacity goes up, bar rises for cutting edge simulation work (higher resolution, more physics, longer runs)
- Modern experiments or observations with bigger instruments - vastly more data to be processed
- Modern simulations or data processing requires parallel computation

Parallel Computing is Everywhere

- Parallel programming used to be needed only for the very largest computations or data sets
- Now, most laptops have two computing cores - independent (more or less) CPUs
- Modern simulations or data processing requires parallel computation

"Increasingly, we are discussing how to scale performance to core counts that we aren't yet shipping (but in some cases we've **hinted heavily** that we're heading in this direction). Dozens, hundreds, and even thousands of cores are not unusual design points around which the conversations meander."

--Anwar Ghuloum, June 30, 2008,

http://blogs.intel.com/research/2008/06/unwelcome_advice.php

Why multicore?

- Moore's Law didn't promise us clock speed.
- More transistors but getting hard to push clock speed up
- So more cores at fixed clock speed

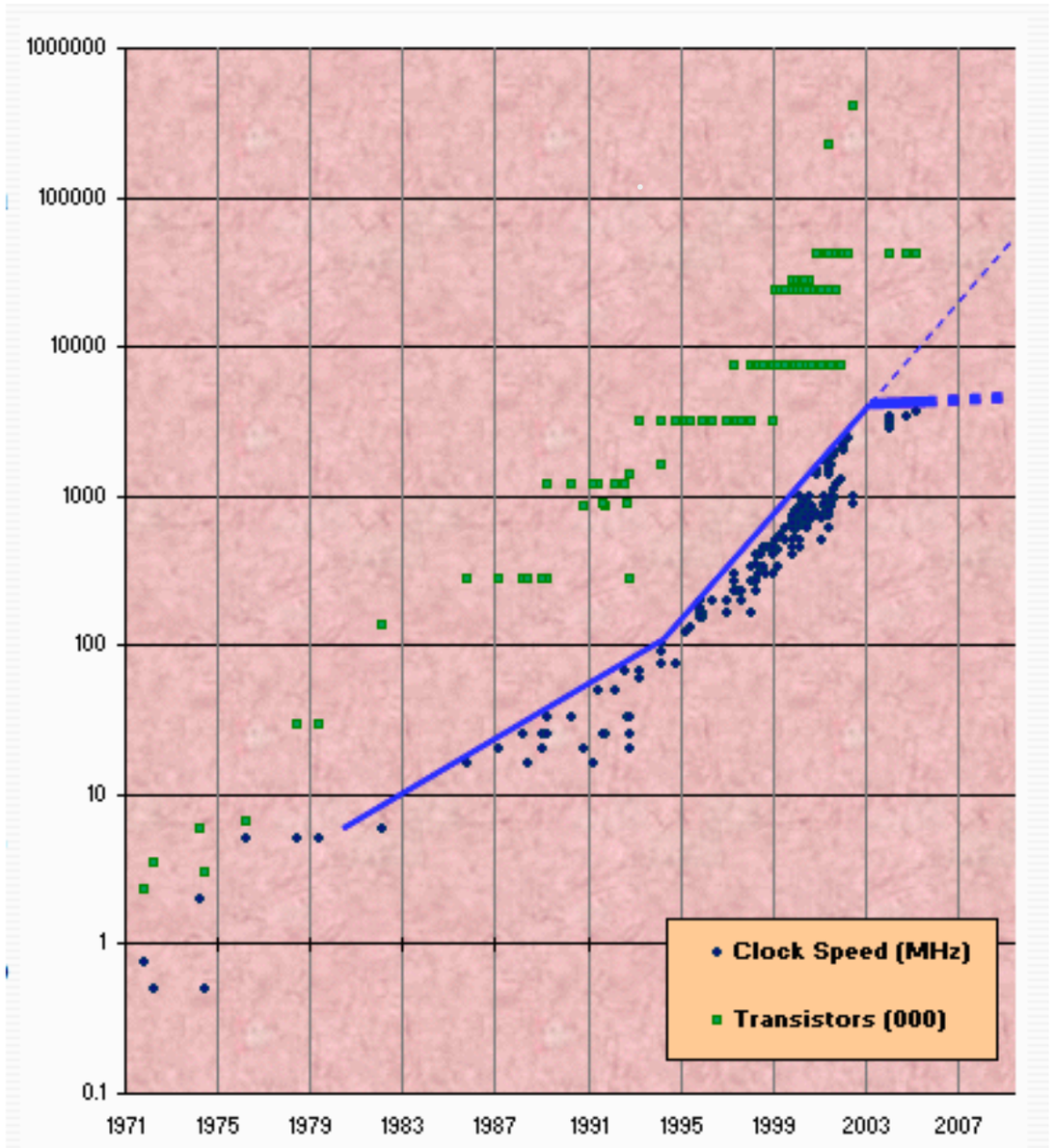
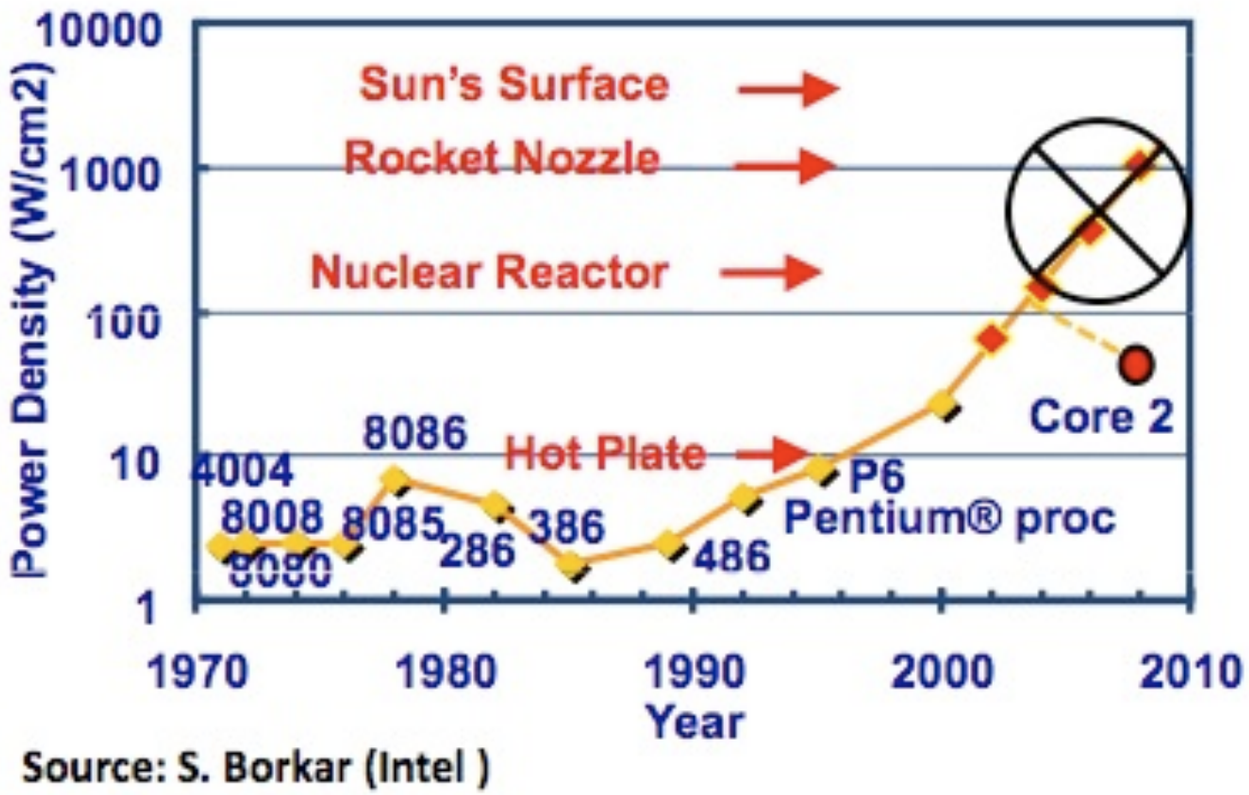


Figure 1: Intel CPU Introductions (sources: Intel, Wikipedia)

There are real engineering reasons why CPU speed has flatlined



Source: S. Borkar (Intel)

Parallel Computing:

- it's Necessary
- it's Everywhere
- it's Only Going to Get Worse

Schedule

Mon	Tues	Wed	Thurs	Fri
Intro to course	OpenMP 2	Map Making	NBody I	Other approaches
	Hands On			GPU
Lunch				
C, OpenMP I	MPI	Hydro	NBody 2	Resources
Hands On				

What will we be doing here

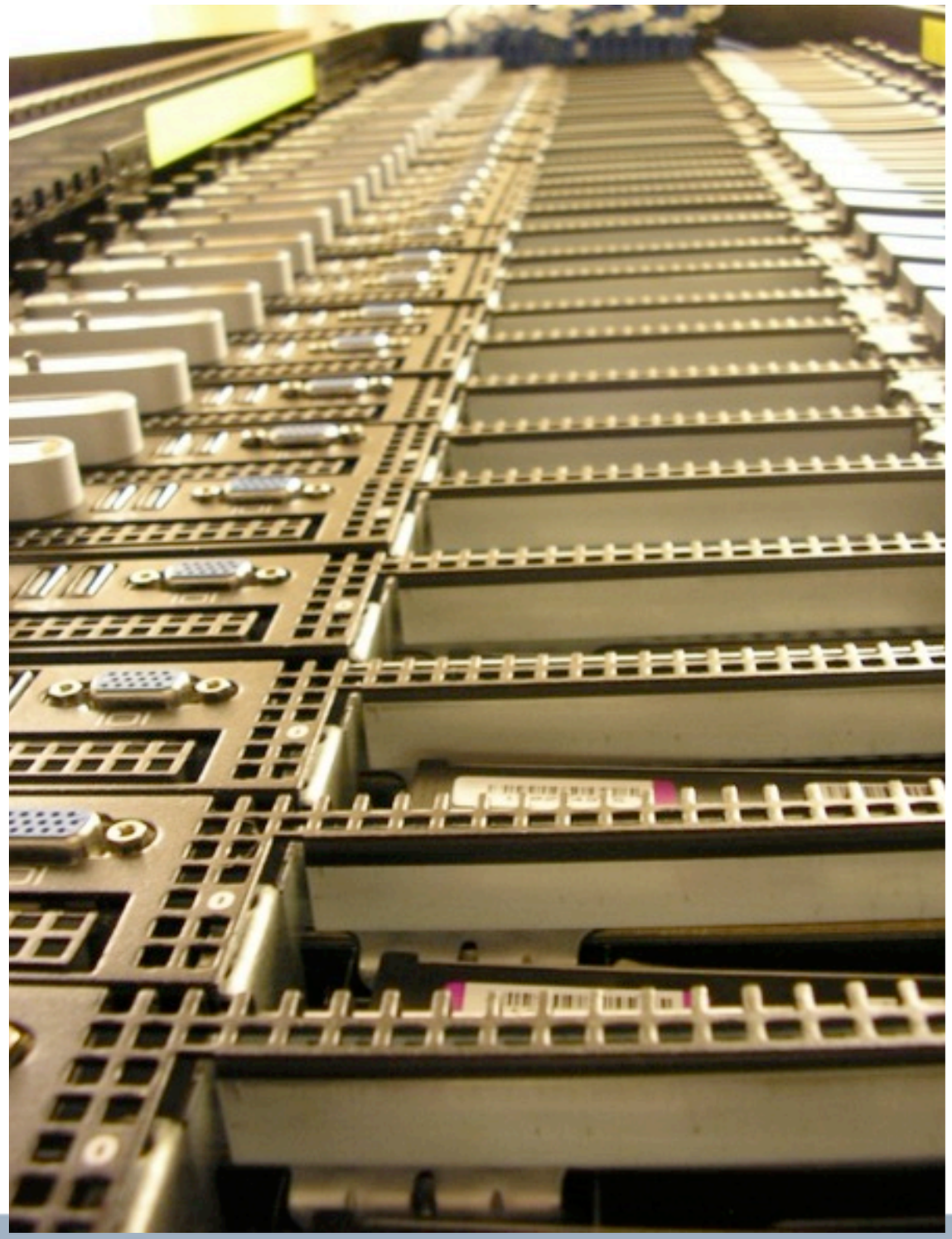
- This is a short course on parallel *programming*
- After (or during!) each lecture session there will be a hands on session where you will work on projects to help build skills with OpenMP, MPI.

Parallel Computing

I: Concurrency, Amdahl's Law, and Locality

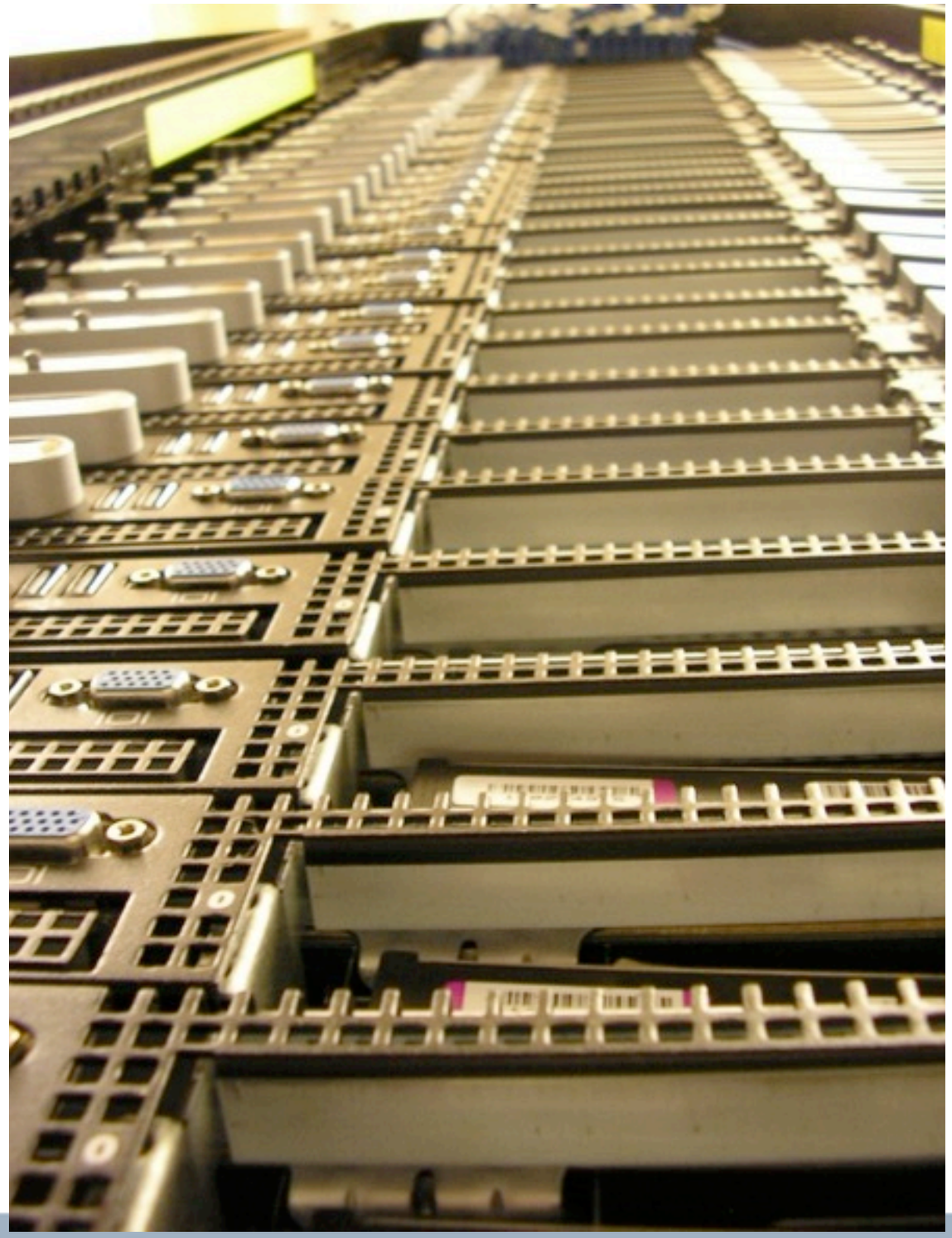
Why Parallel Computing?

- ***Faster***:
- At any given time, there is a limit as to **how fast** one computer can compute.
- So use more computers!



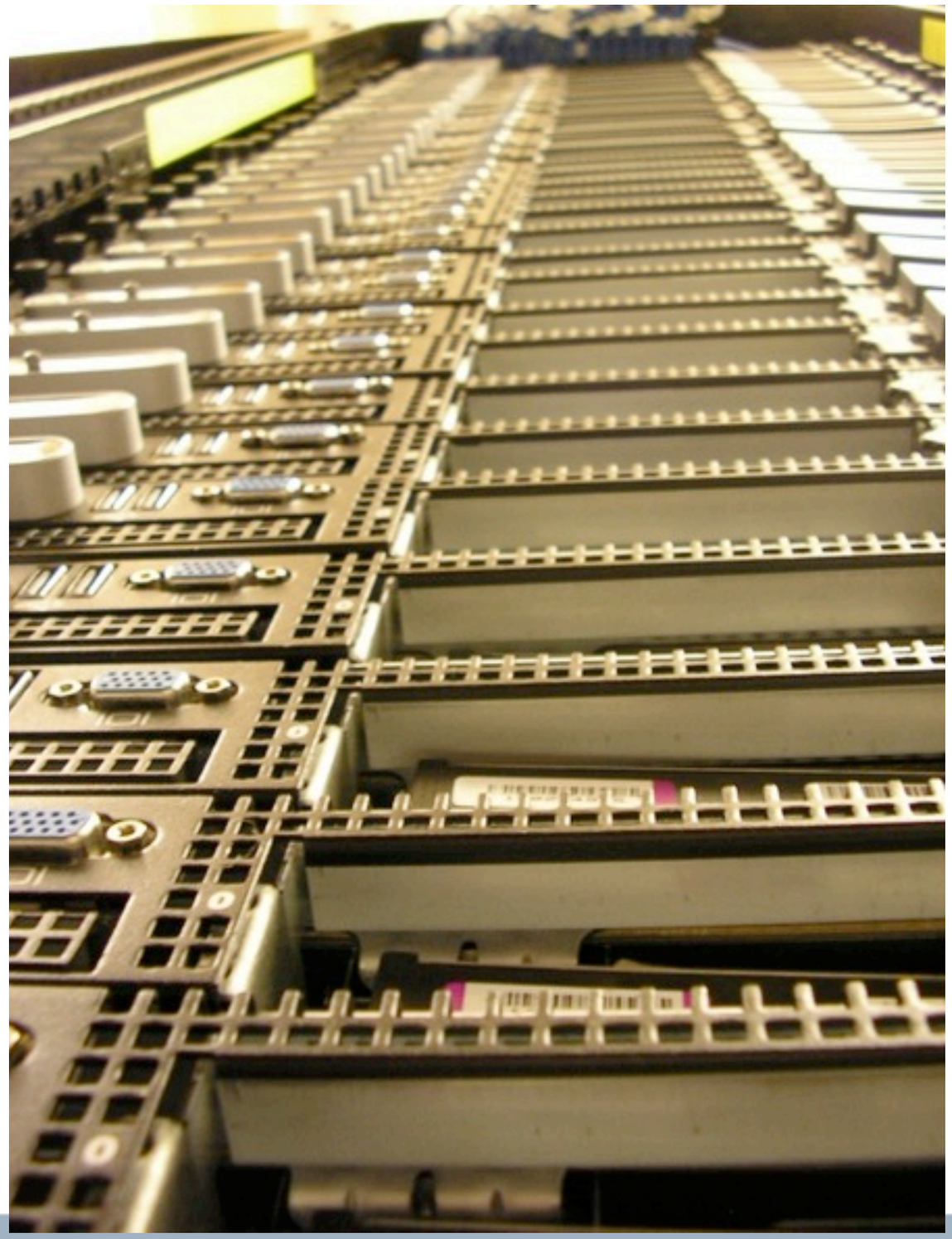
Why Parallel Computing?

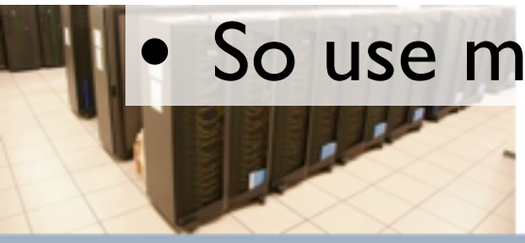
- ***Bigger***:
- At any given time, there is a limit as to **how much** memory, disk space, etc can be put on one computer.
- So use more computers!



Why Parallel Computing?

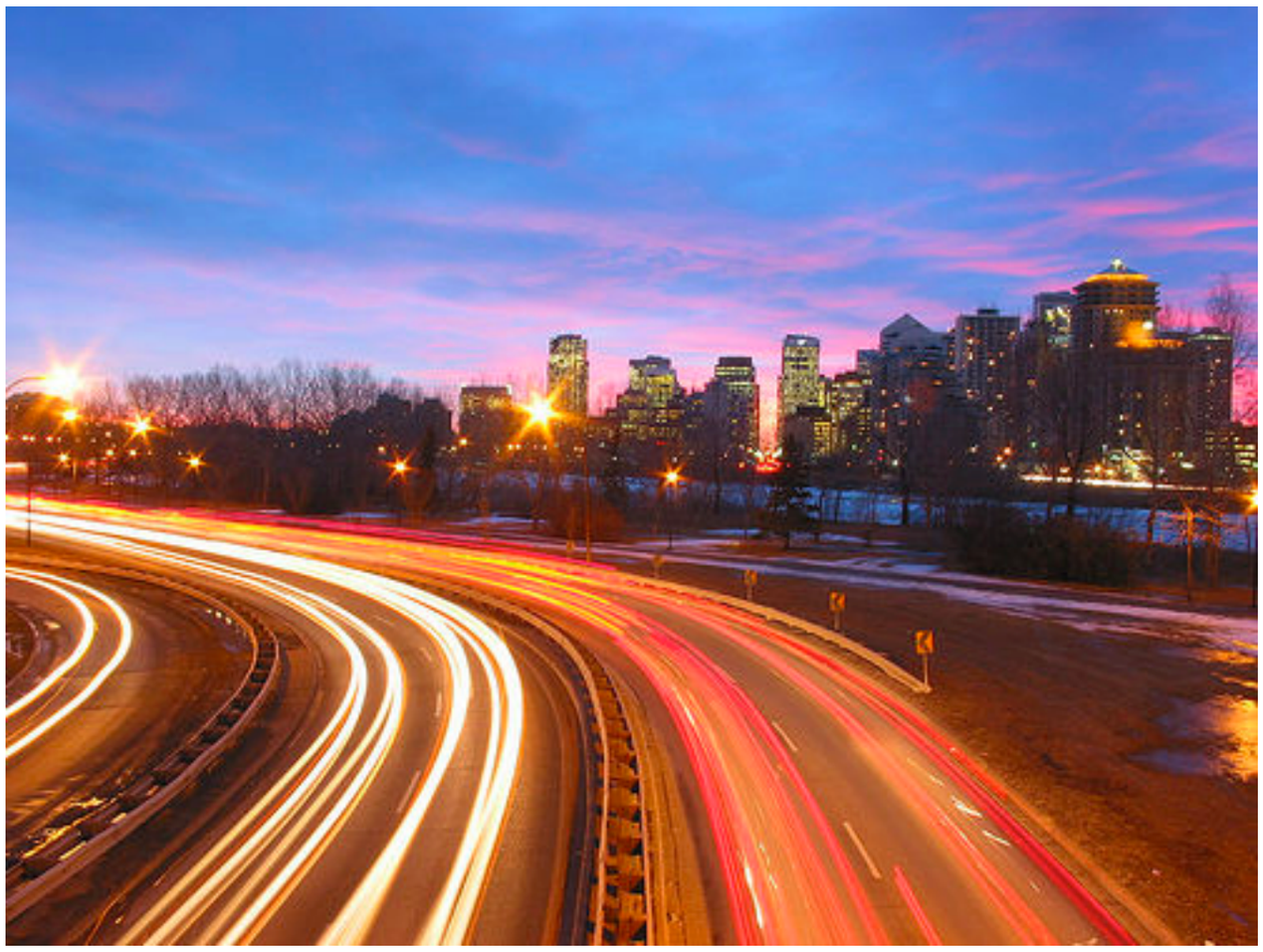
- ***More***:
- You have a program that runs in reasonable time one one processor but you want to run it **thousands of times**.
- So use more computers!





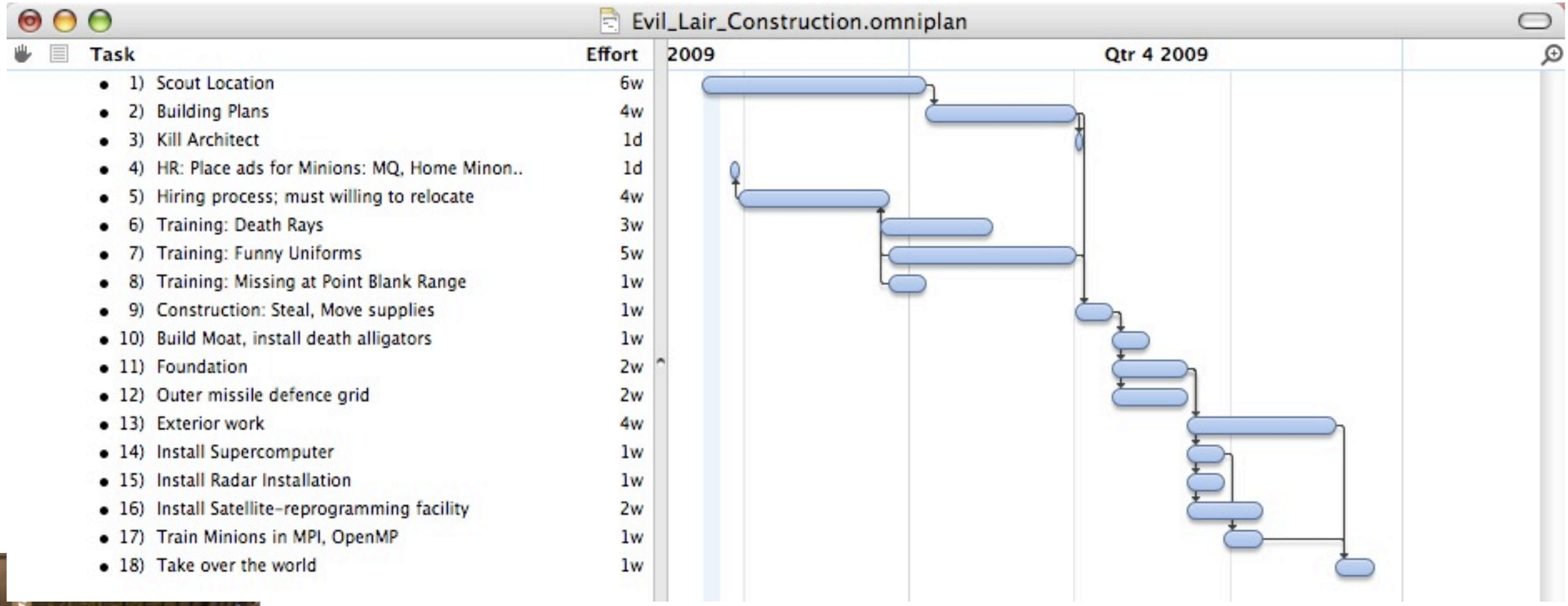
Concurrency

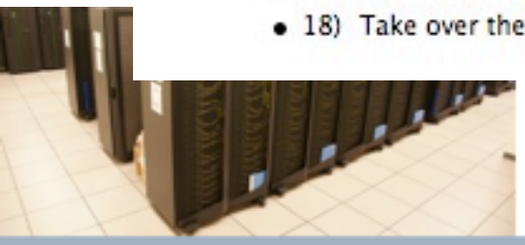
- Must be something for the 'more computers' to do.
- Must be able to find *concurrency* in your problems
  - Many Tasks
  - Order Unimportant



http://flickr.com/photos/splorp/

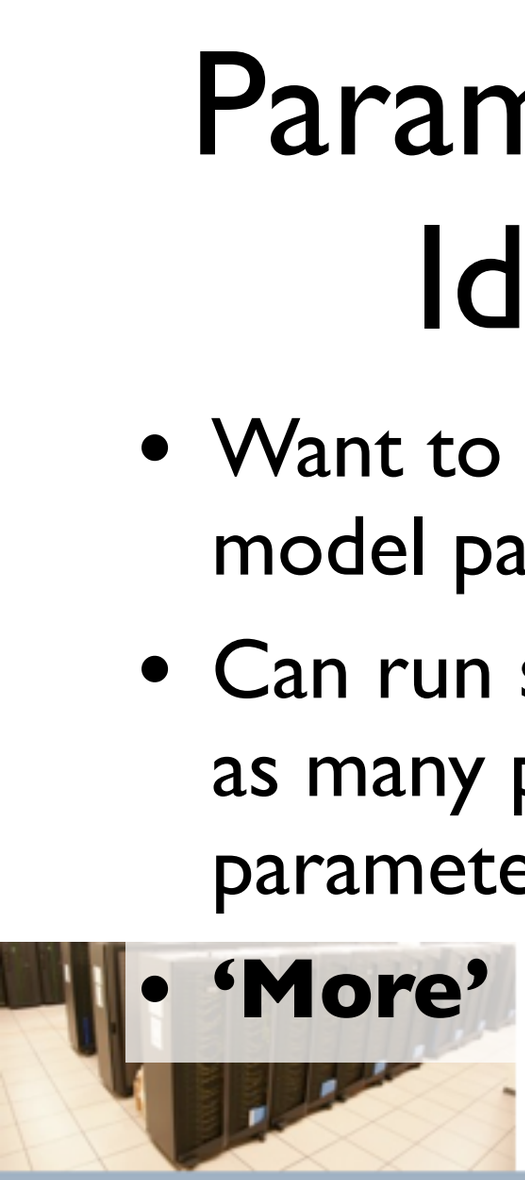
Data Dependancies Limit Concurrency

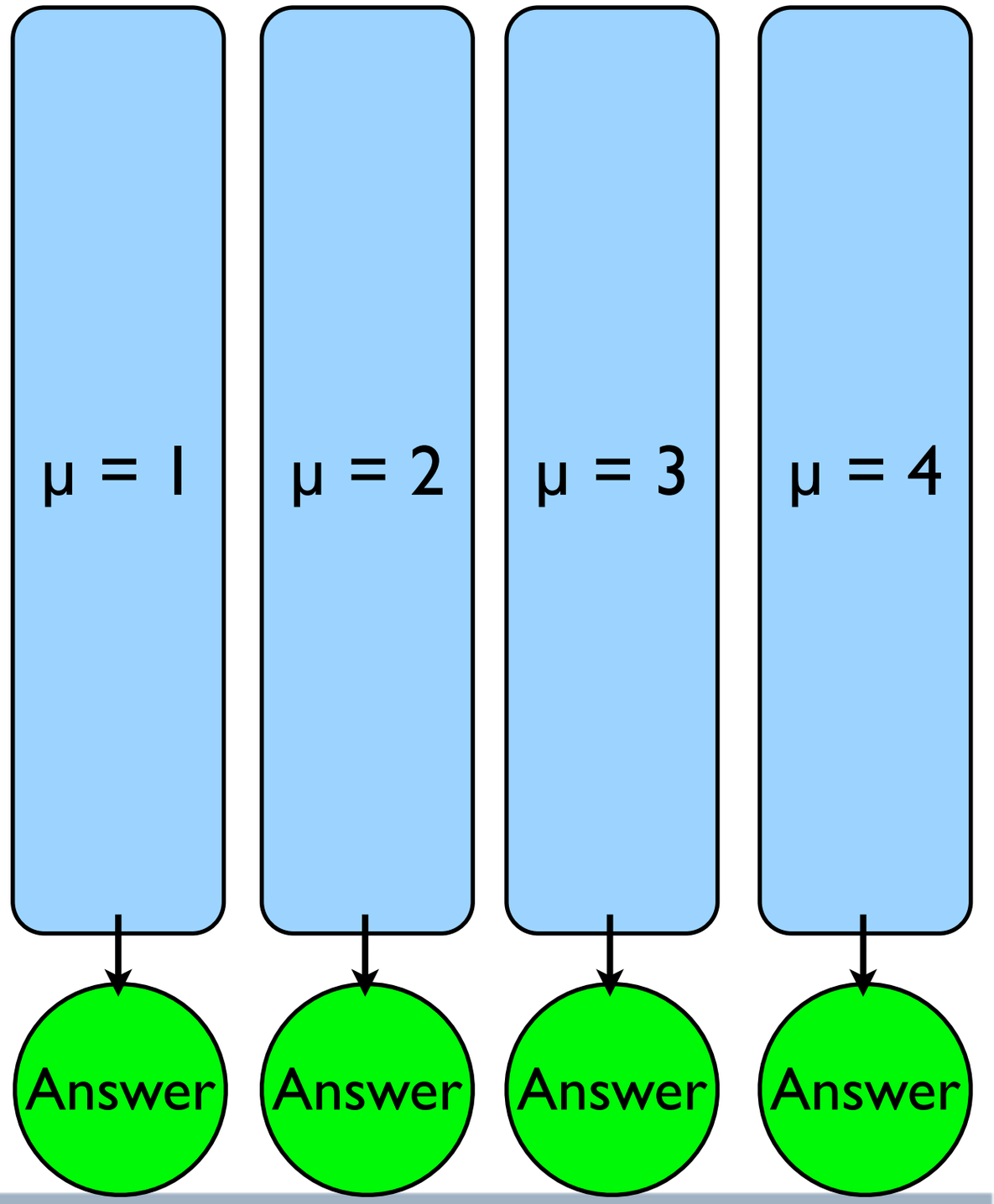




Parameter Study: Ideal case

- Want to know all results as model parameter varies
- Can run serial code on up to as many processors as parameter sets
- **'More'**



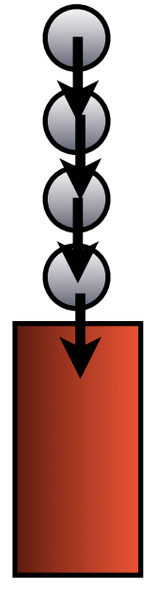


Throughput = Tasks/Time

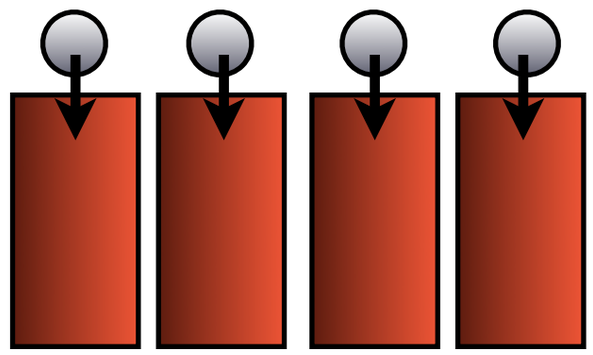
- How long it takes to process the N tasks you want done

$$\text{throughput} = \frac{N}{\text{time}}$$

- For completely independent tasks, P processors can increase throughput by factor P!

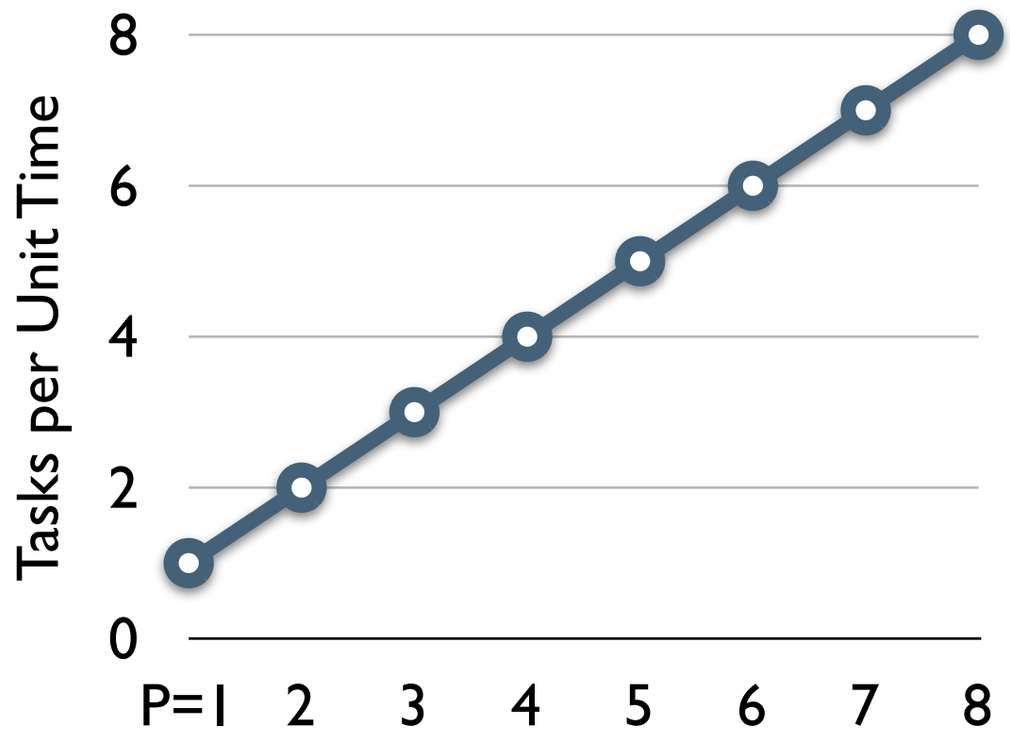


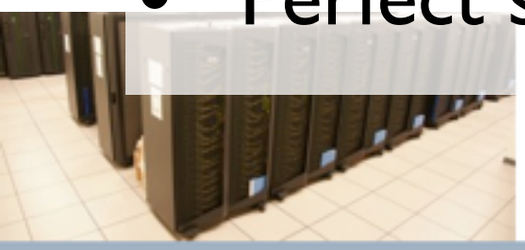
vs



Scaling with P

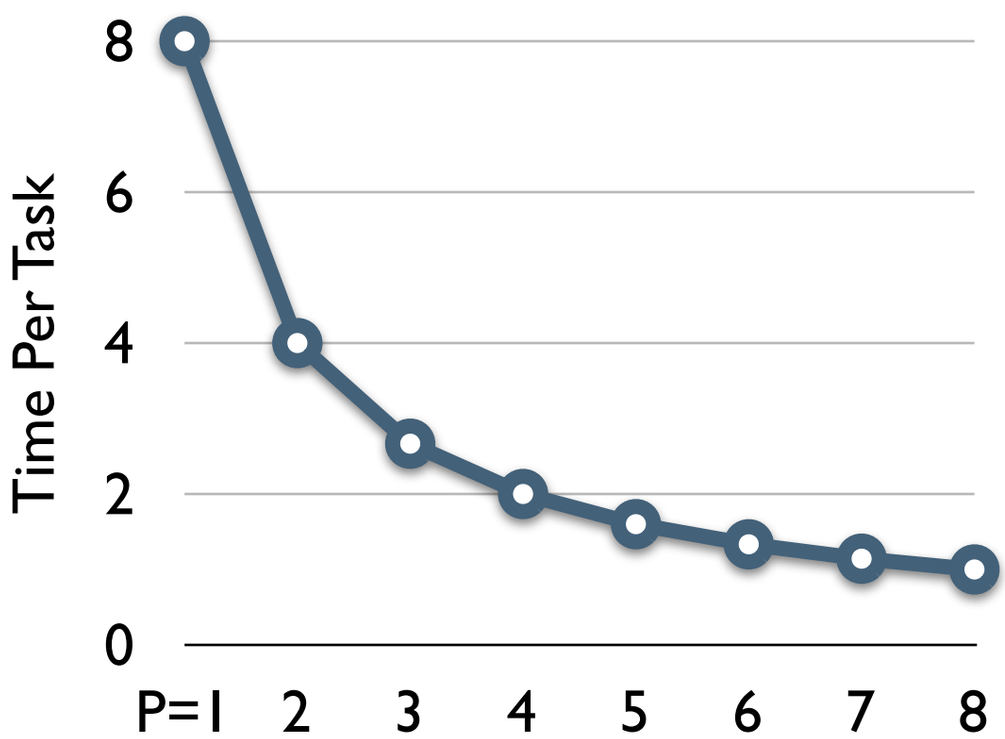
- How a problem **scales**: how throughput behaves as processor number increases
- In this case, the throughput scales linearly with the number of processors
- This is the best case
- `Perfect scaling'

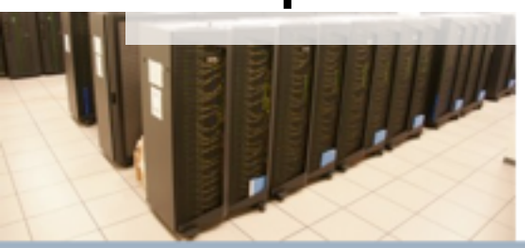




Scaling with P

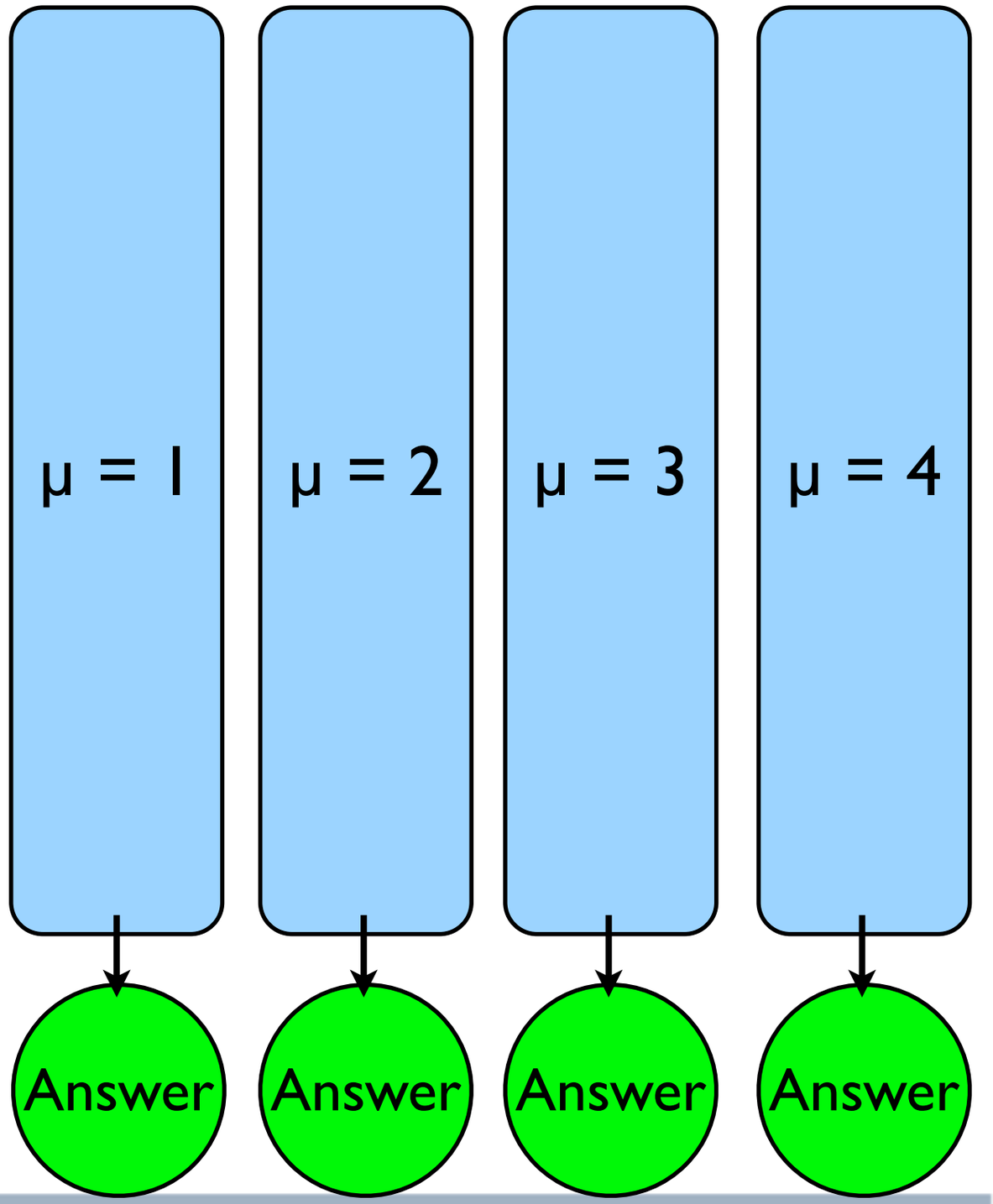
- Another way to look at it: time it takes to get some fixed amount of work done
- More usual (and more important!)
- Perfect scaling: time to completion ~ 1/P
- P procesors - P times faster





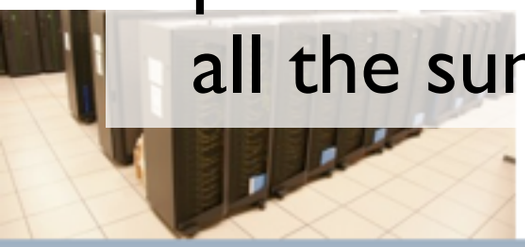
Parameter Study: 'Embarrassingly Parallel'

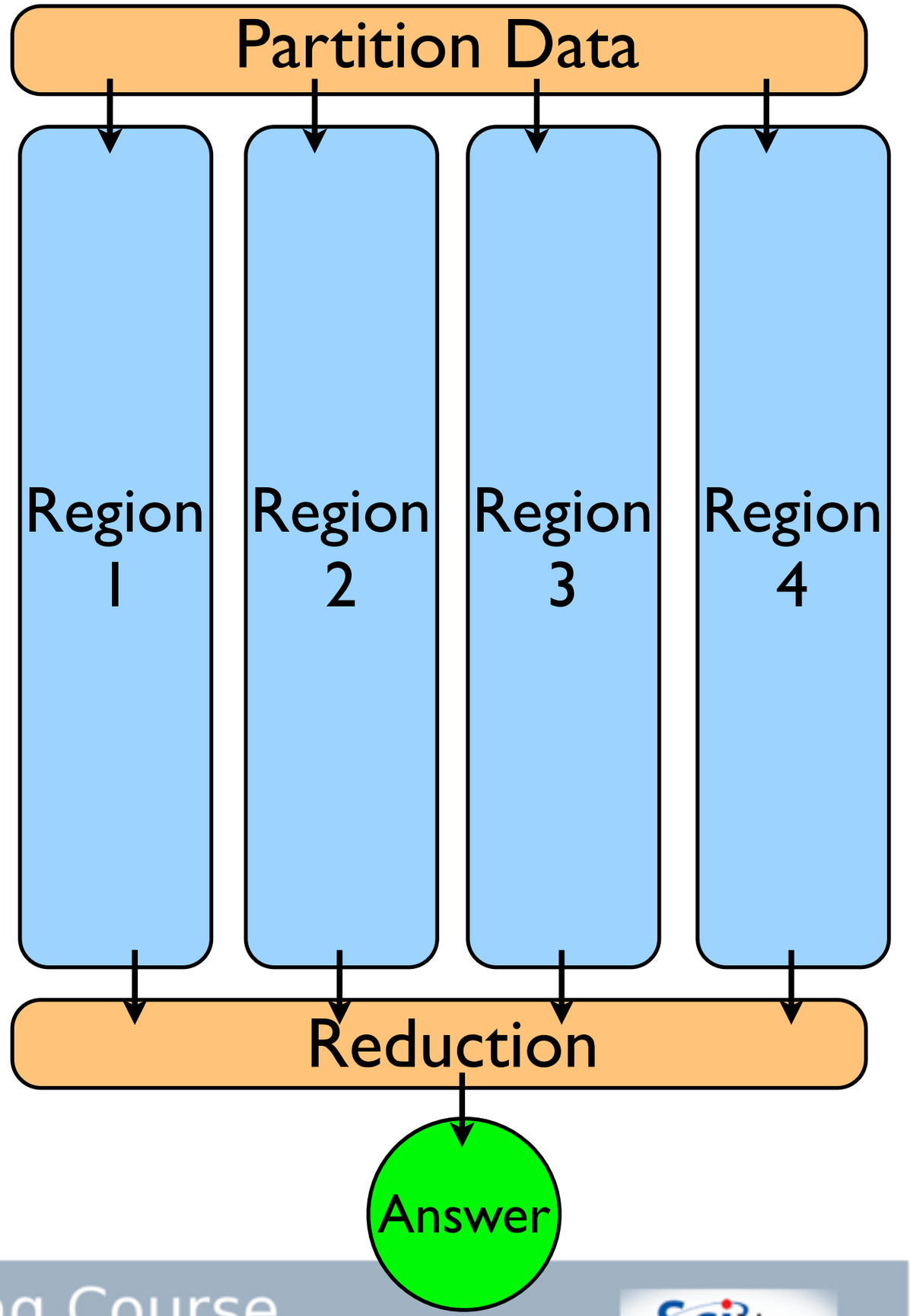
- **Scales** perfectly up to P=N
- Speedup = P: 'linear scaling', ideal case.



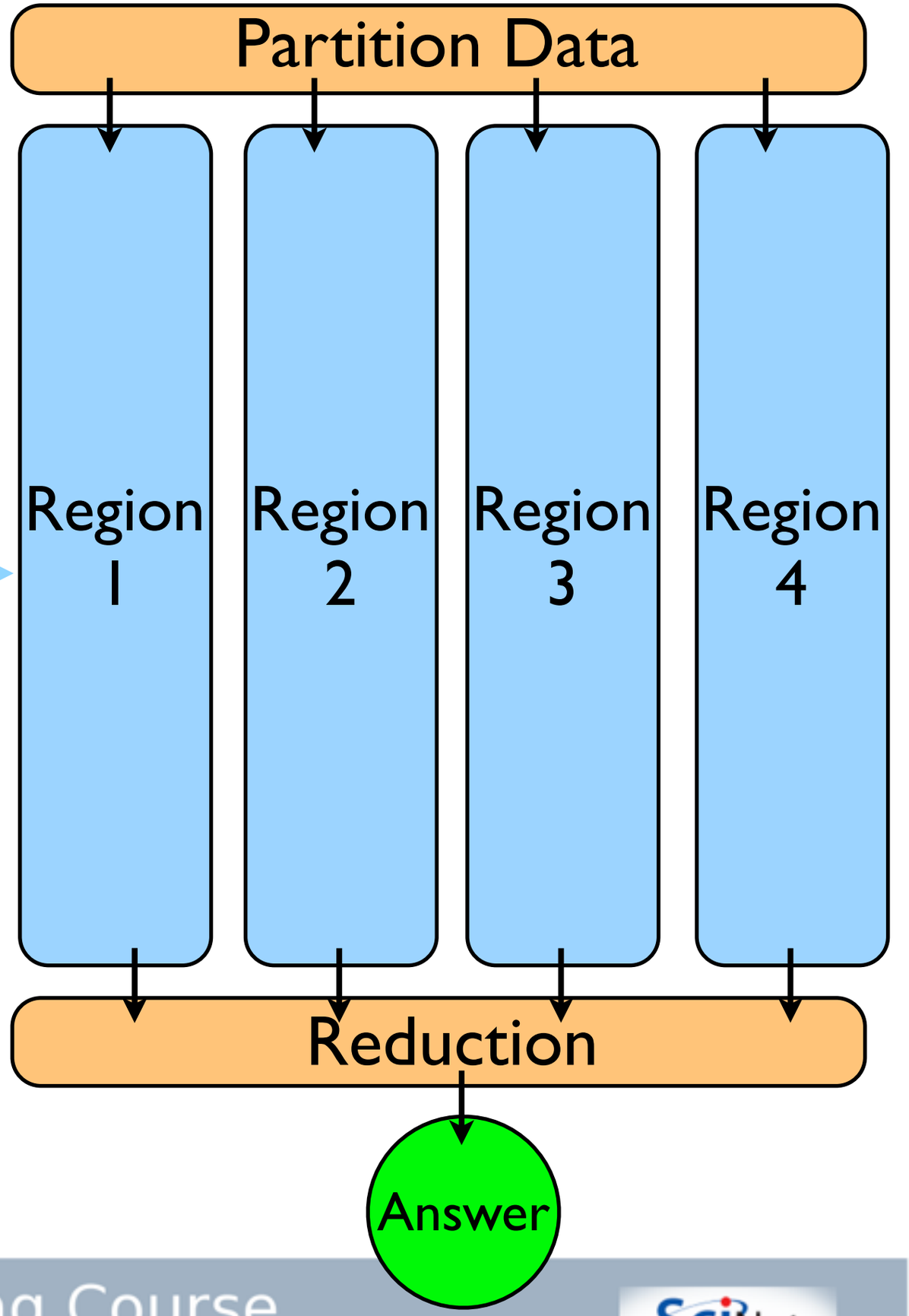
Problems Differ in amount of Concurrency

- Integrate (or some other simple processing) tabulated experimental data
- Integration of different regions can be summed by each processor
- But first need to get data to processor, then bring together all the sums





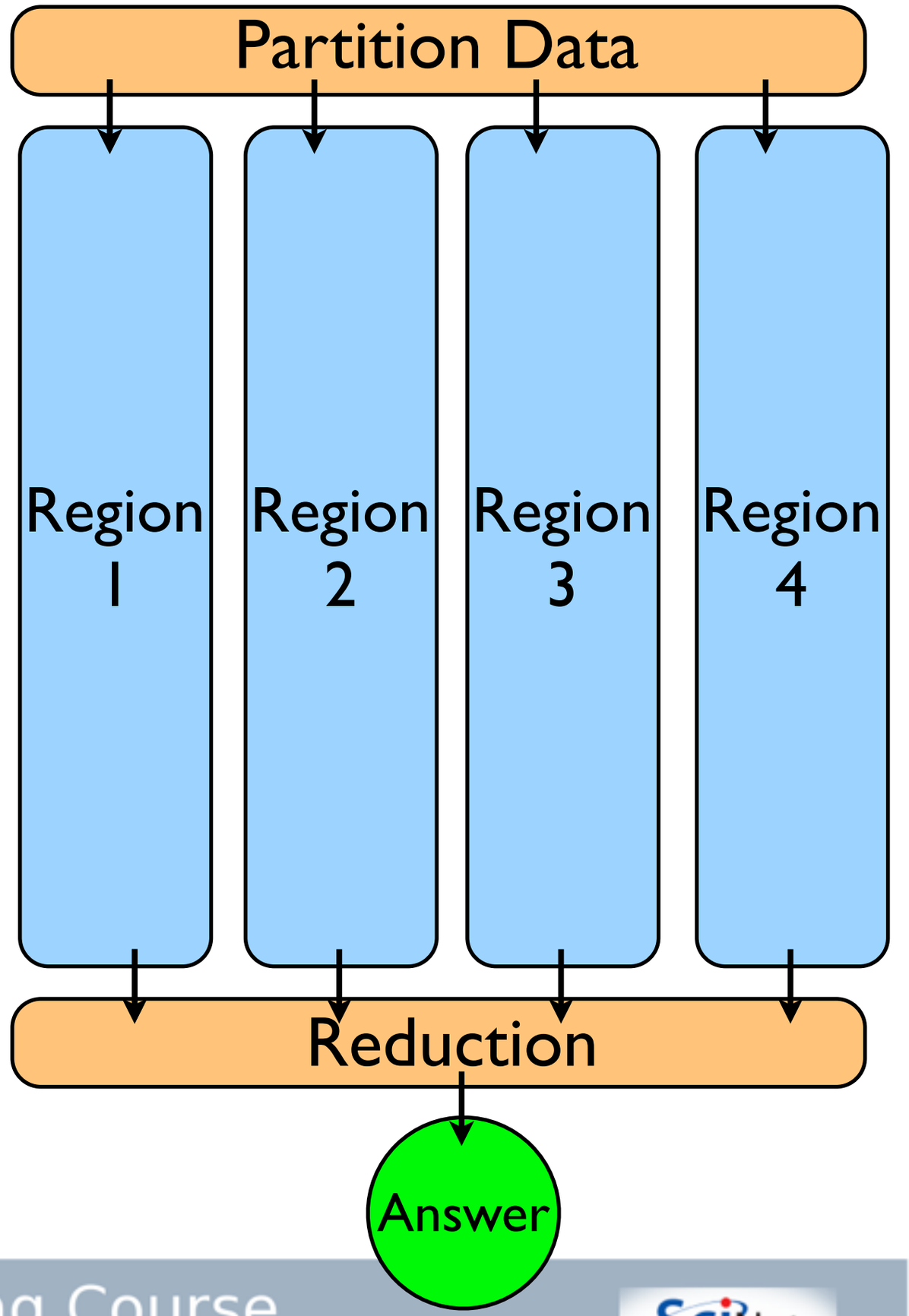
Parallel Portion:
Perfectly Parallel (as long as there is enough work)
$T \sim I/P$



Serial Portion:

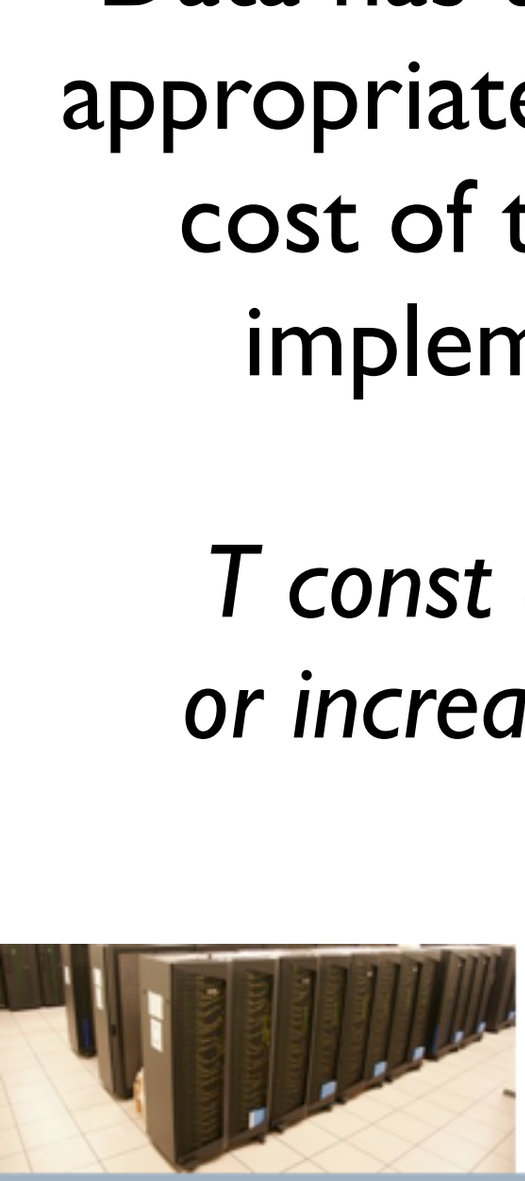
Sum has to be done; if done on one processor, just same as serial:

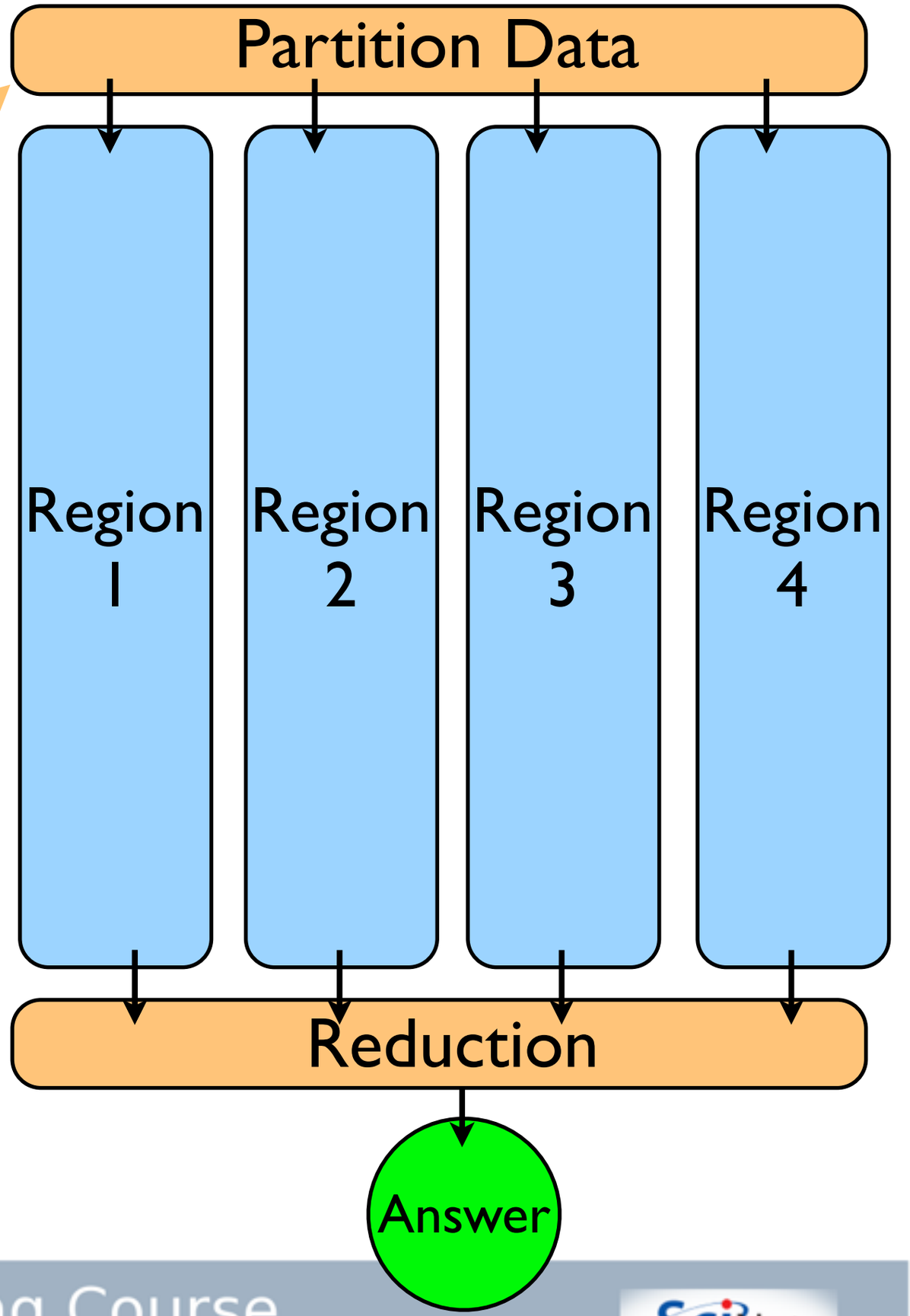
T ~ const



Parallel Overhead:
Data has to be sent to appropriate processor, a cost of the parallel implementation

T const (best case)
or increasing fn of P



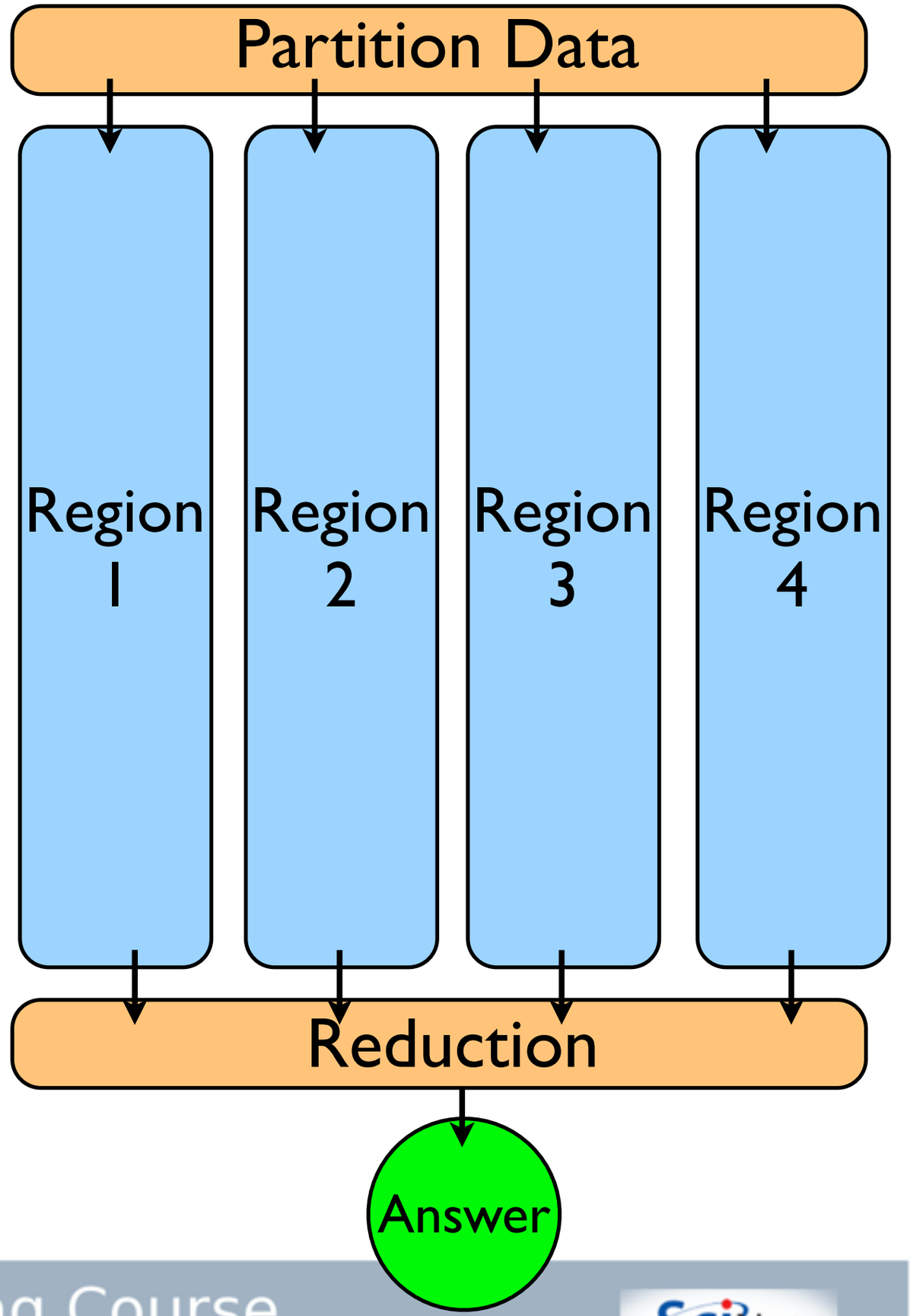


Total Time: Serial + Parallel

- Ignoring data-moving costs (for now):

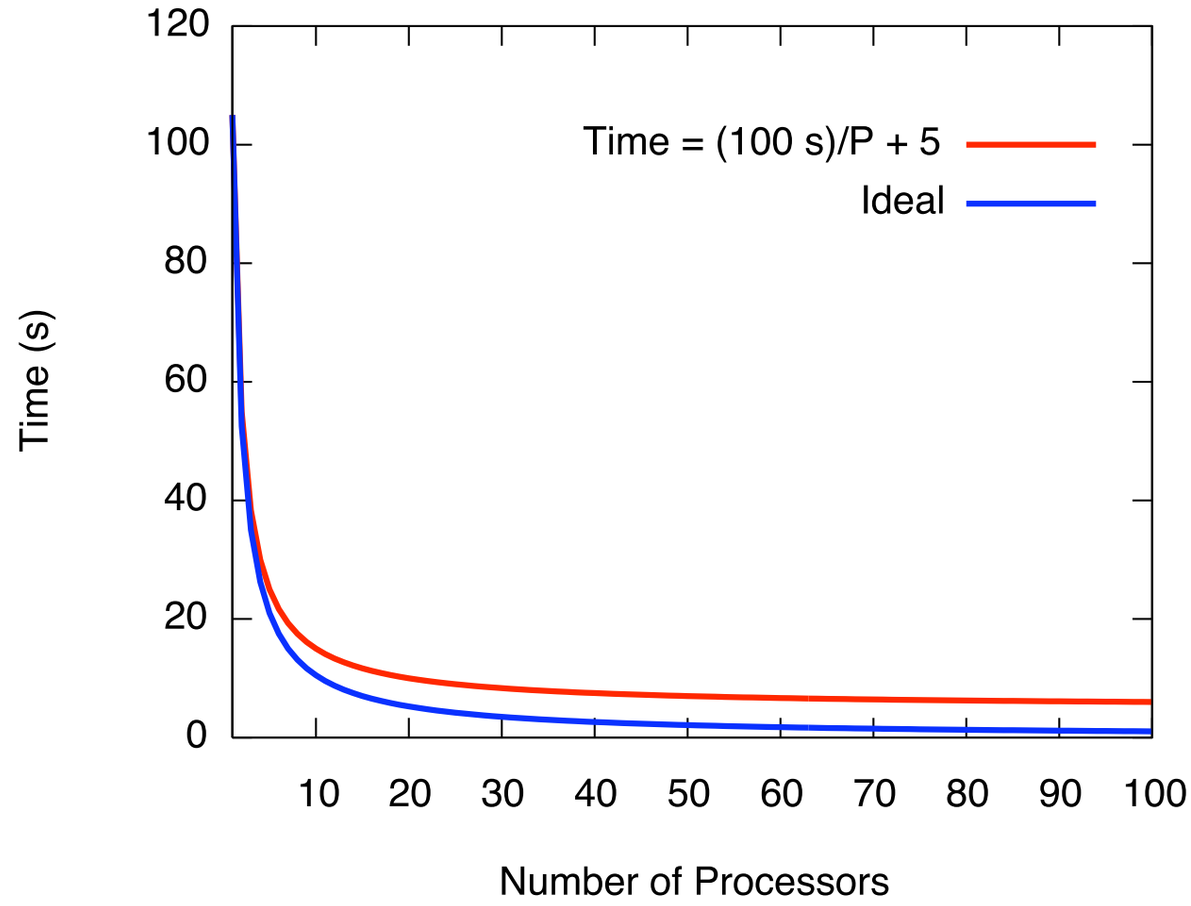
$$\text{time}(N,P) = \left\lfloor \frac{N}{P} \right\rfloor T_{\text{work}} + T_{\text{reduction}}(P)$$

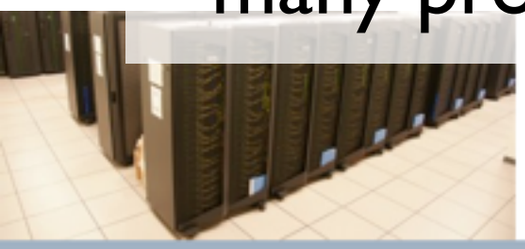
- Typically linear in P (sum)
- Eventually, as problem becomes increasingly scaled up, serial term dominates



Timing of simple case

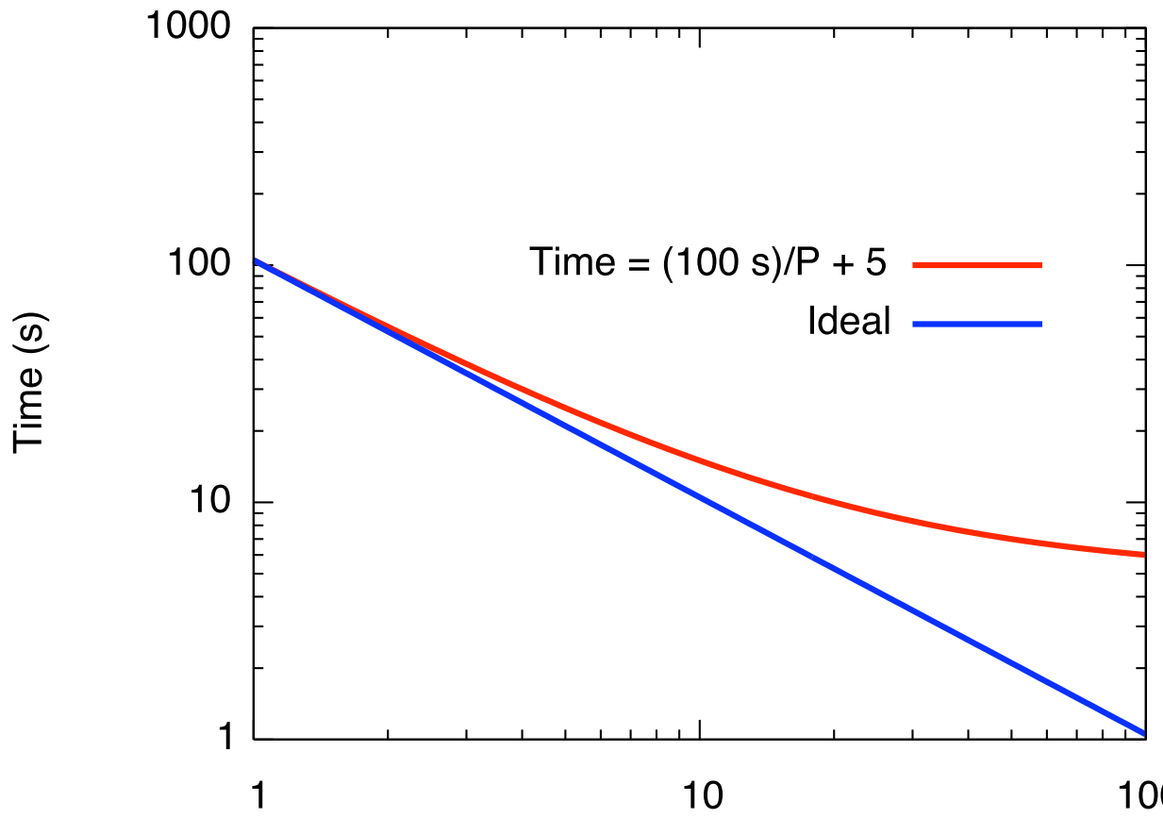
- Ignore data transfer costs; say:
  - 100 s in integration work
  - 5 s in assembling the parts
- How does this behave on many processors?





More processors per run don't always help

- Given timing data, how do we choose P to run on if we have N programs to run?
- Ideal case, timing goes down 1/P - doesn't matter
- Serial part (5%!) becomes a bottleneck
- Can improve **throughput** by running on *fewer* processors



Note: $t(50) = 7\text{s}$

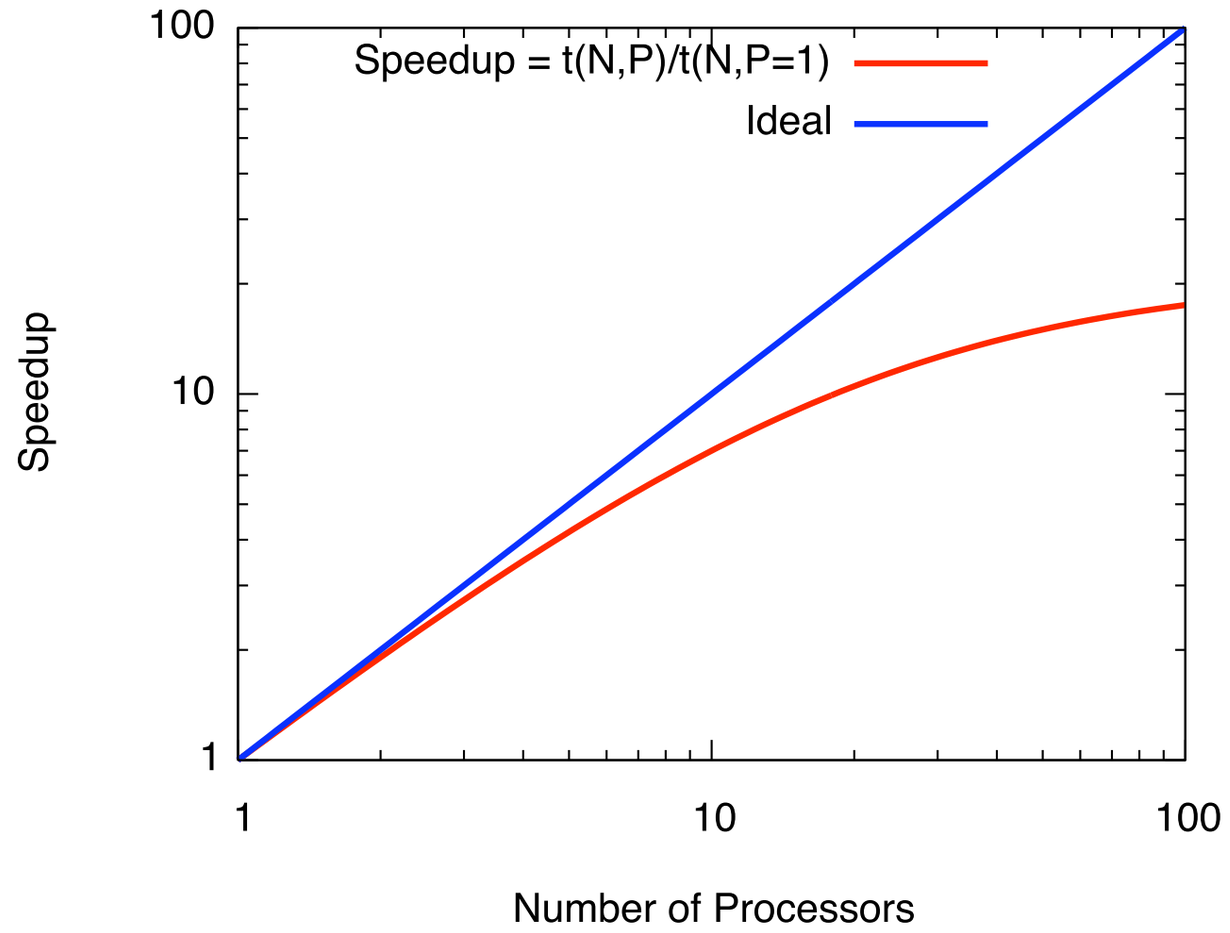
$t(25) = 9\text{s}$

Can run 2 jobs on 25 procs each in about same time as one on 50!

Speedup: How much faster with P procs?

- An important concept is the speedup of a given parallel implementation

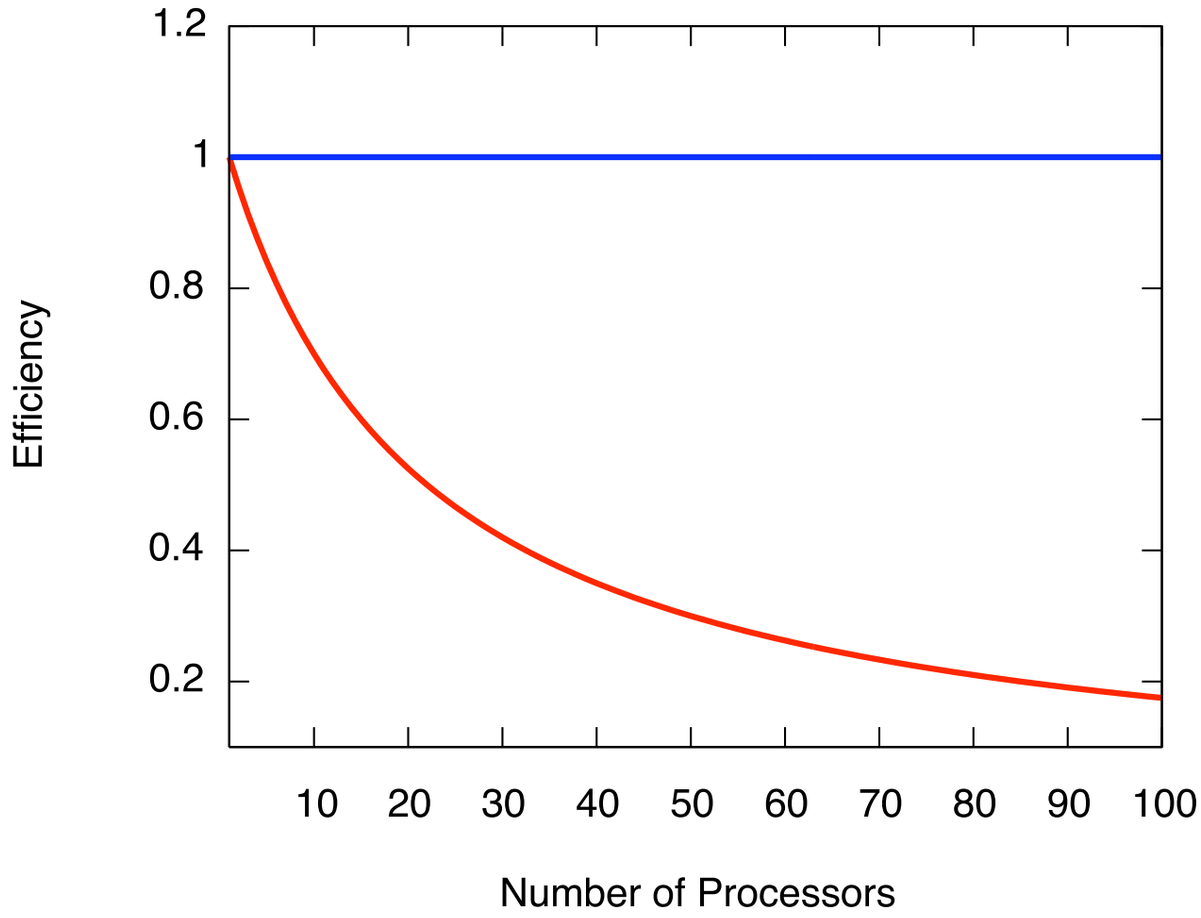
$$\text{speedup} = \frac{t(N,P)}{t(N,P=1)}$$



Efficiency: Speedup should be ~ P

- Related concept: Parallel Efficiency (compared to serial code)

$$\text{Efficiency} = \frac{t(N,P)}{Pt(N,P=1)}$$



Amdahl's Law

- **Any serial part of computation will eventually dominate**
- If serial fraction is f, even if parallel component goes to zero, speedup can only be 1/f

serial fraction

(perfectly) parallel fraction

$$\text{time}(N,P) \sim \left(f + \frac{1-f}{P}\right)$$

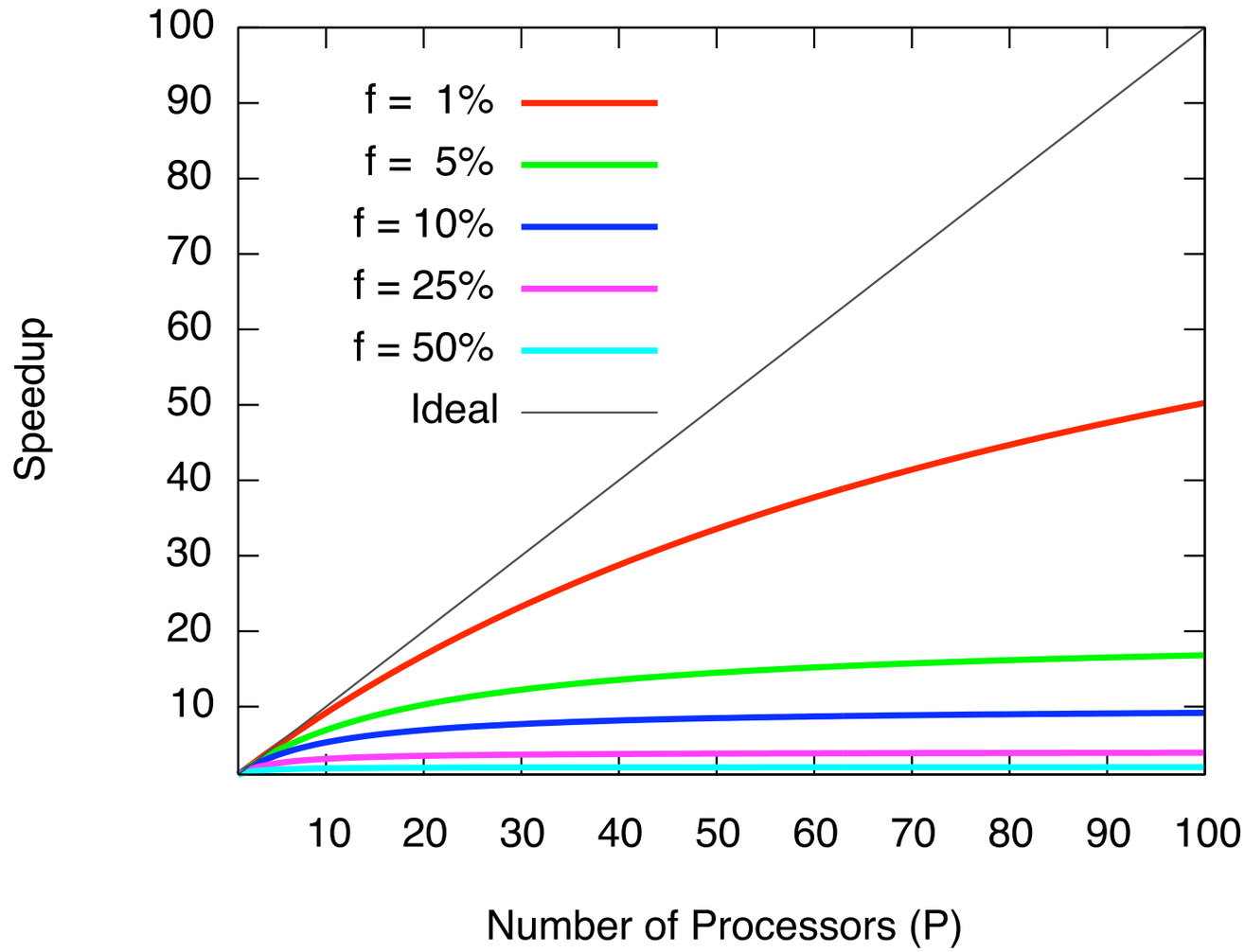
$$\text{Speedup} = \frac{1}{\left(f + \frac{1-f}{P}\right)}$$

$$\lim_{P\to\infty} \text{Speedup} = \frac{1}{f}$$

$$\lim_{P\to\infty} \text{Efficiency} = 0$$

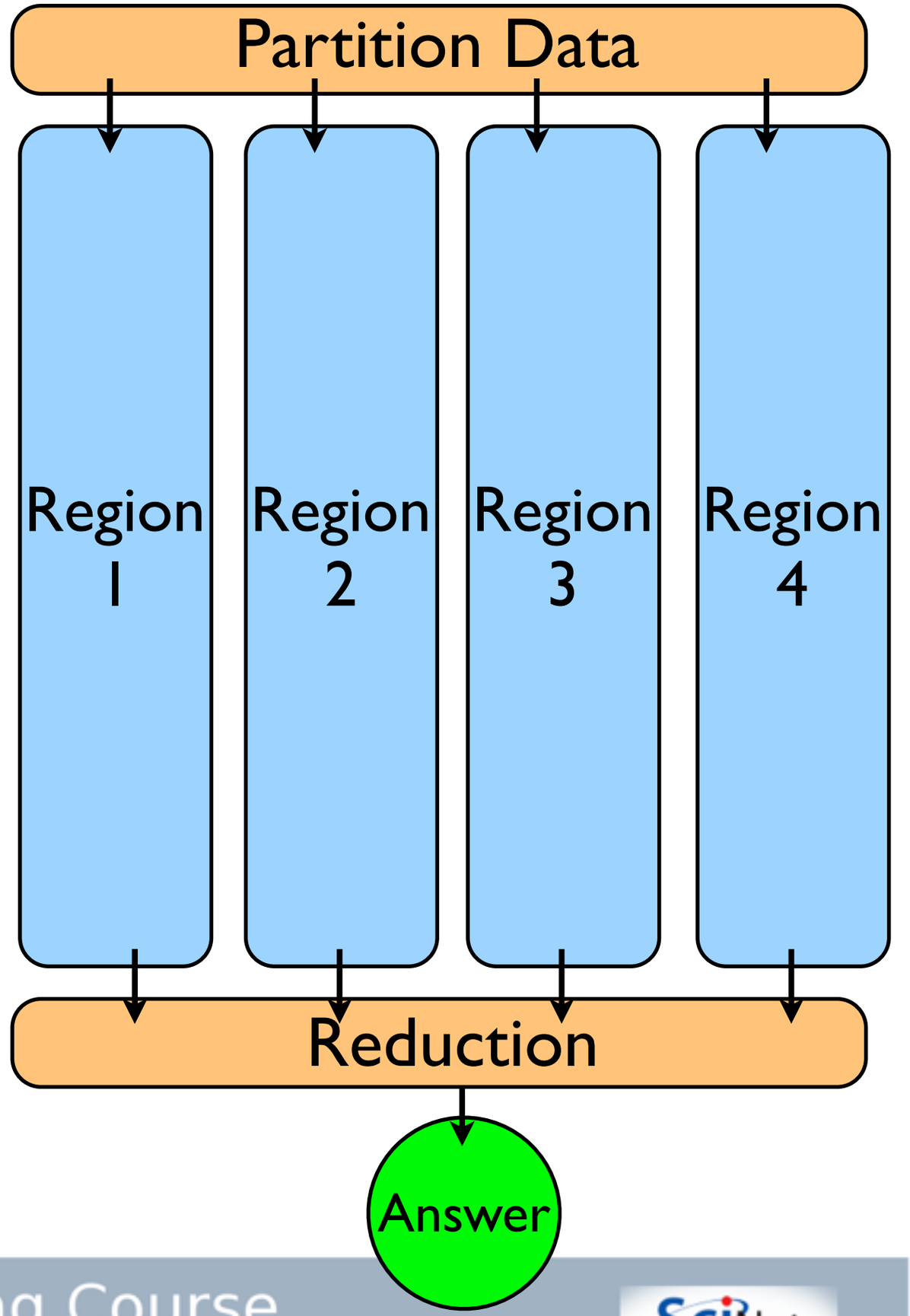
Amdahl's Law

- **Any serial part of computation will eventually dominate**
- If serial fraction is f, even if parallel component goes to zero, speedup can only be 1/f



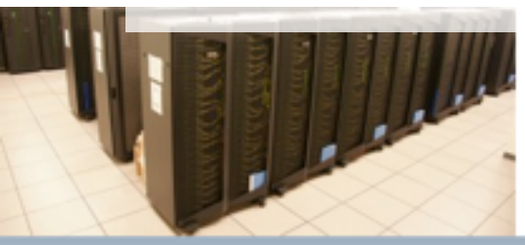
Avoiding Amdahl

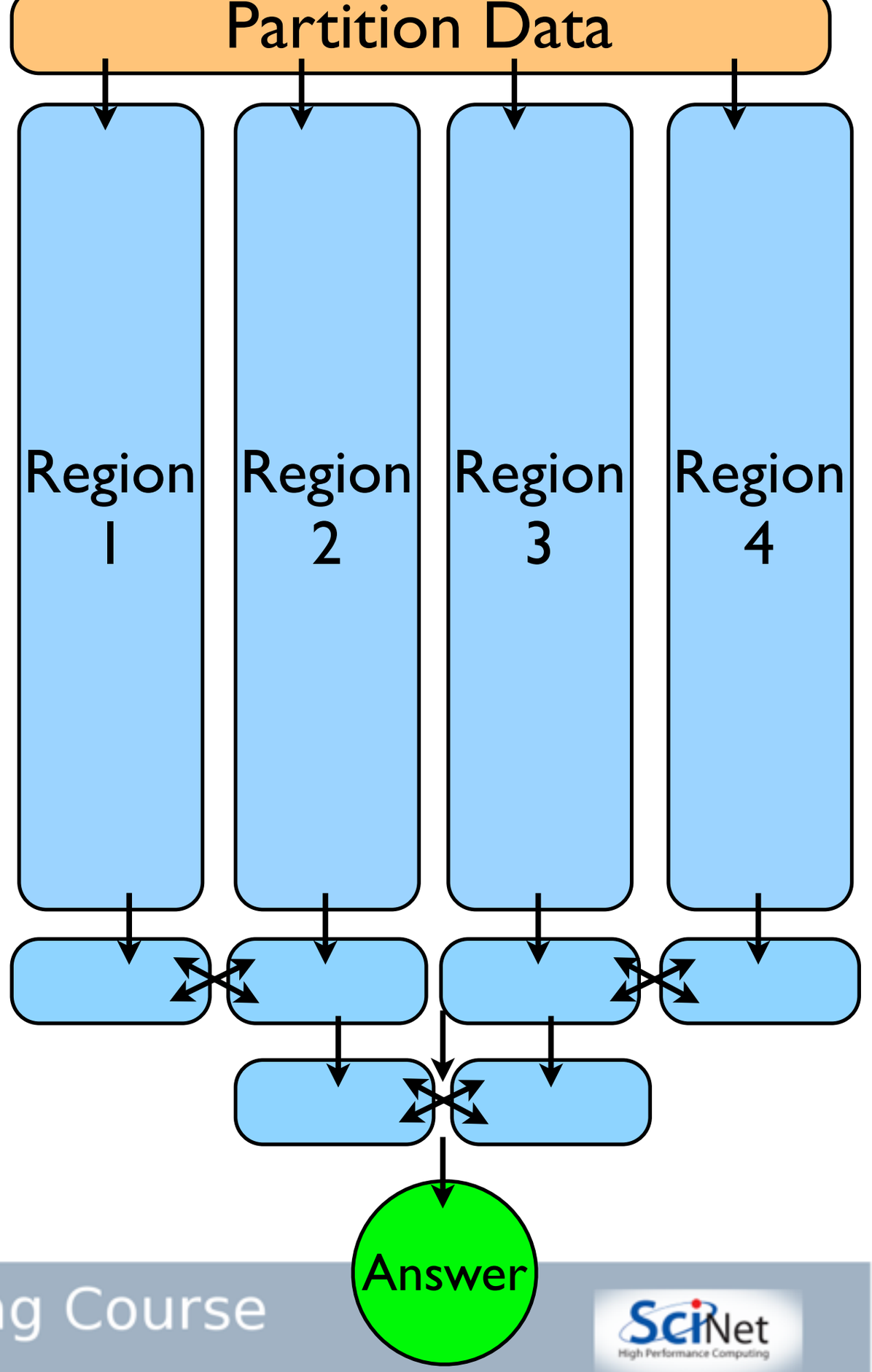
- In some cases, may not matter.
- If will run in reasonable time on some small number of processor, asymptotic arguments may not matter.



Trying to Beat Amdahl, #1

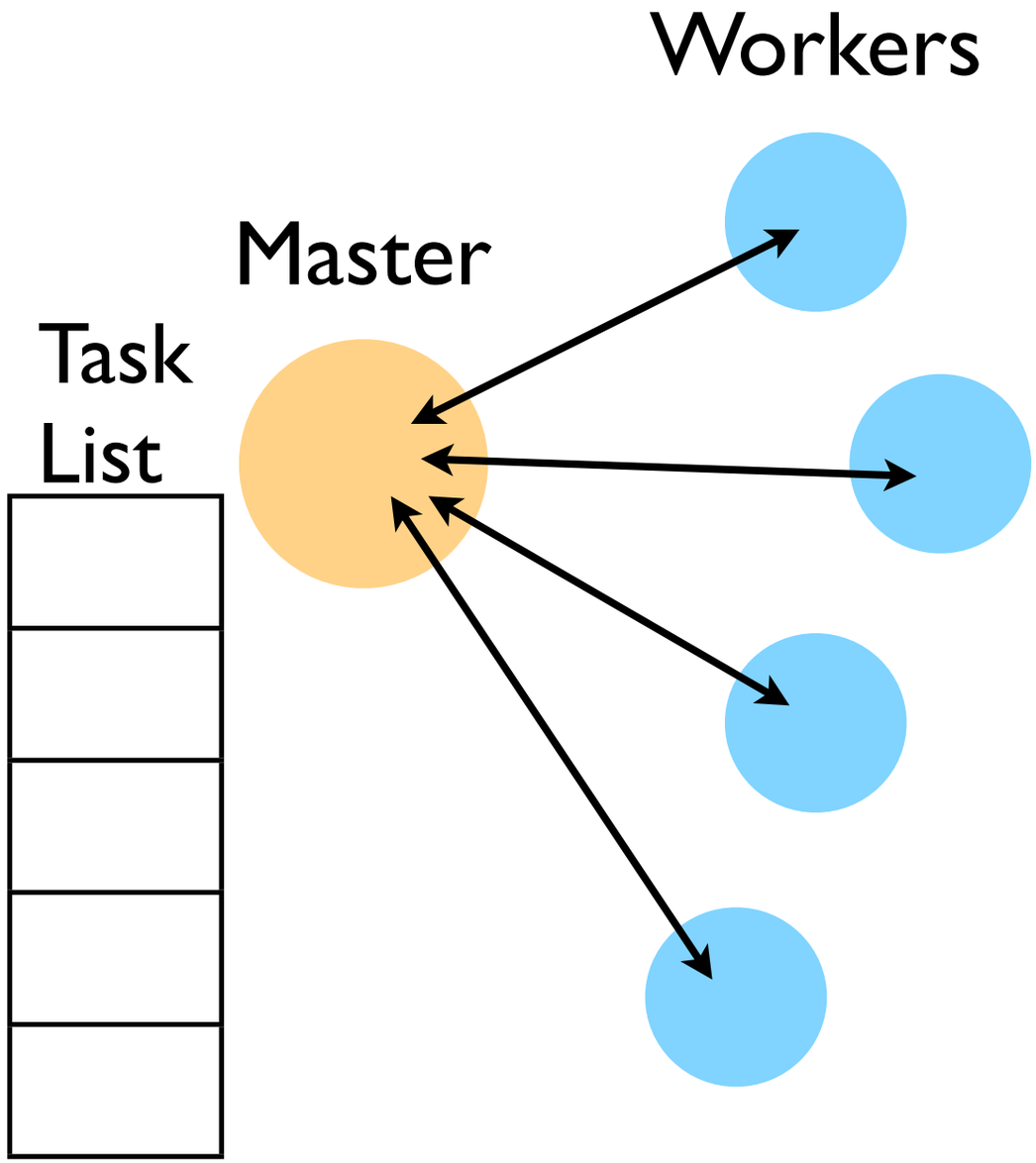
- Rewrite serial portions to take into account parallelism
- eg, many reductions can be done in parallel that will cost $\log_2(P)$ (not 1, but much better than serial = P...)





Trying to Beat Amdahl, #1

- Redo approach to avoid serial portions wherever possible.
- Means some models don't scale well - serial bottleneck
- Master task does disk I/O
- Master task assigns work to workers. (But SETI@Home?)

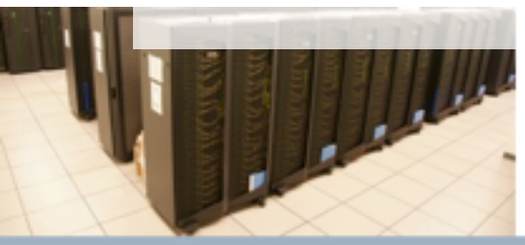


Big Lesson #1

Optimal **Serial** Algorithm for your problem may not be the P→ I limit of your optimal **Parallel** algorithm

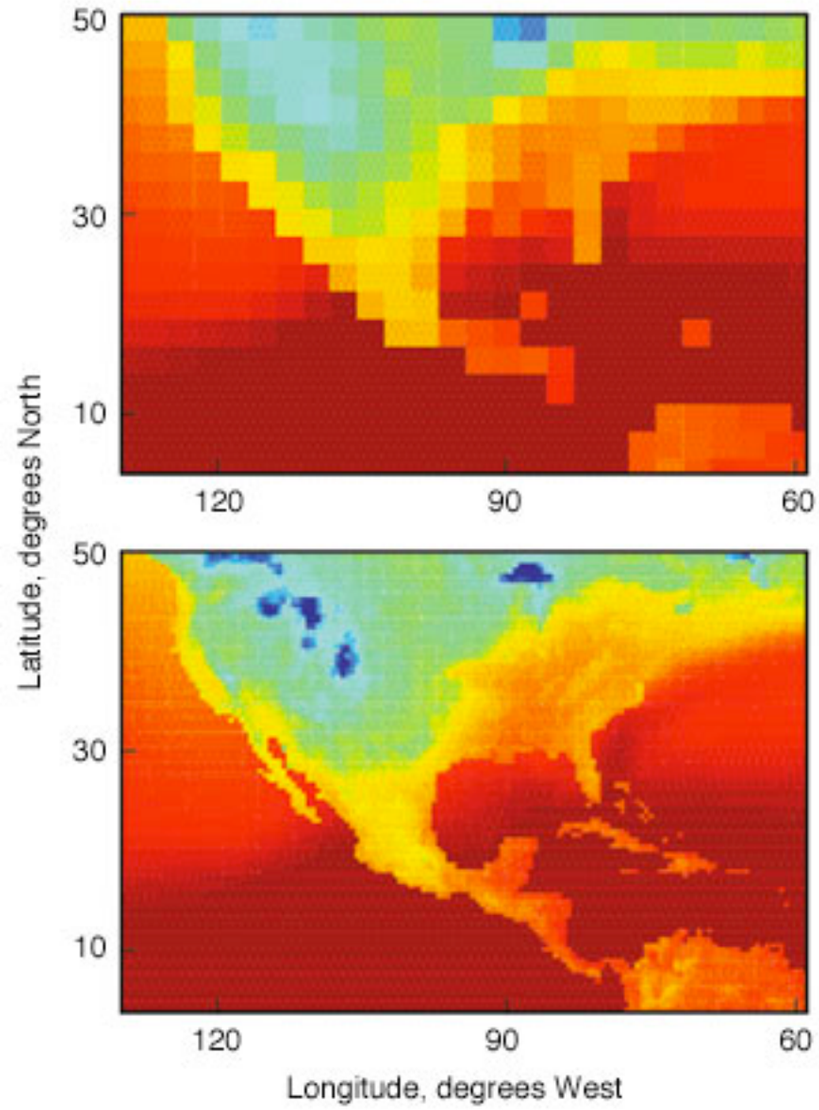
Consequences of BL#1

- P=1 parallel code *will* be slower than best serial code
- May be cheaper to re-compute values than send them (Time to send a float ~4-1000x time to multiply a float)
- As long as overhead is a small fraction of serial time for any reasonable N and doesn't depend on P, you're ok.
- (If cost ~ P, might as well be serial!)



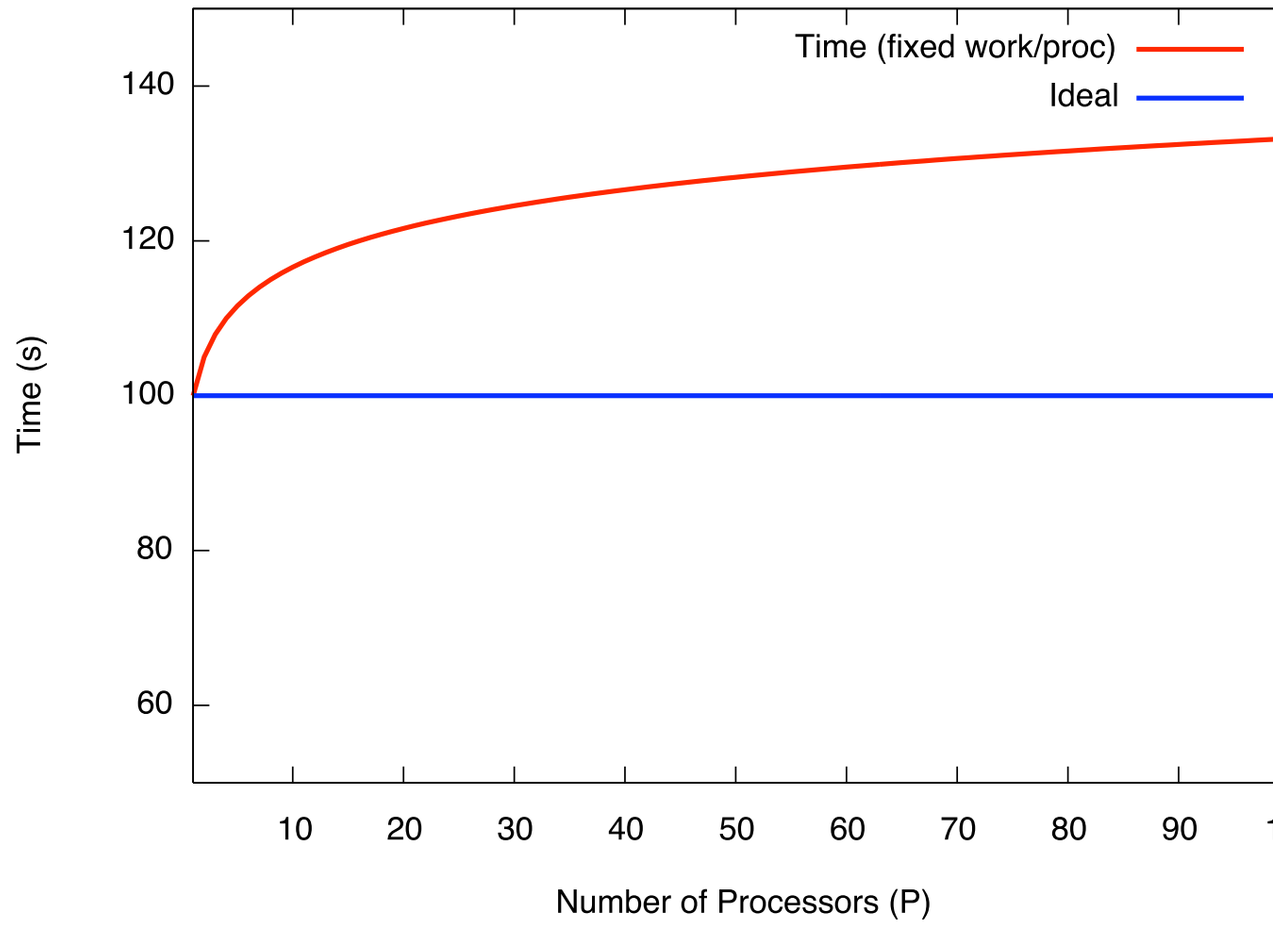
Trying to Beat Amdahl, #2 - Upsize

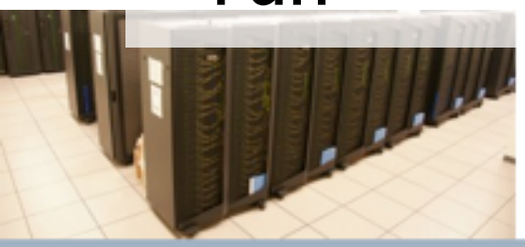
- Desktop problem isn't a supercomputer problem!
- Reason to run on big machines is size as well as speed
- Amdahl's law assumes constant size problem
- More work; f goes down.
- Gustafson's law: any sufficiently large problem can be efficiently parallelized.



Weak Scaling

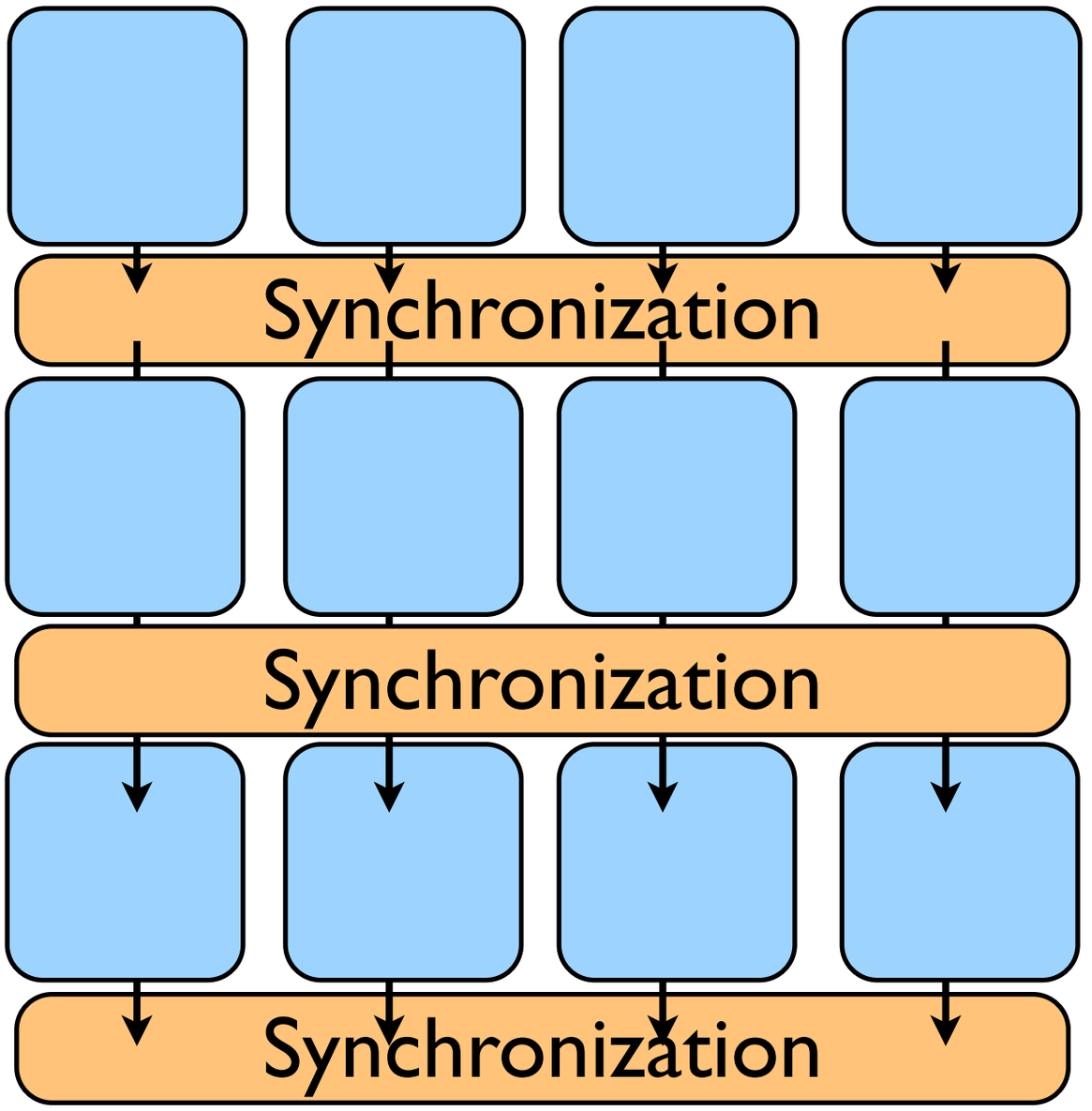
- How does problem behave if you expand problem size as number of processors?
- Strong Scaling - on how many processors can you efficiently run given problem
- Weak Scaling - how large a problem can you efficiently run





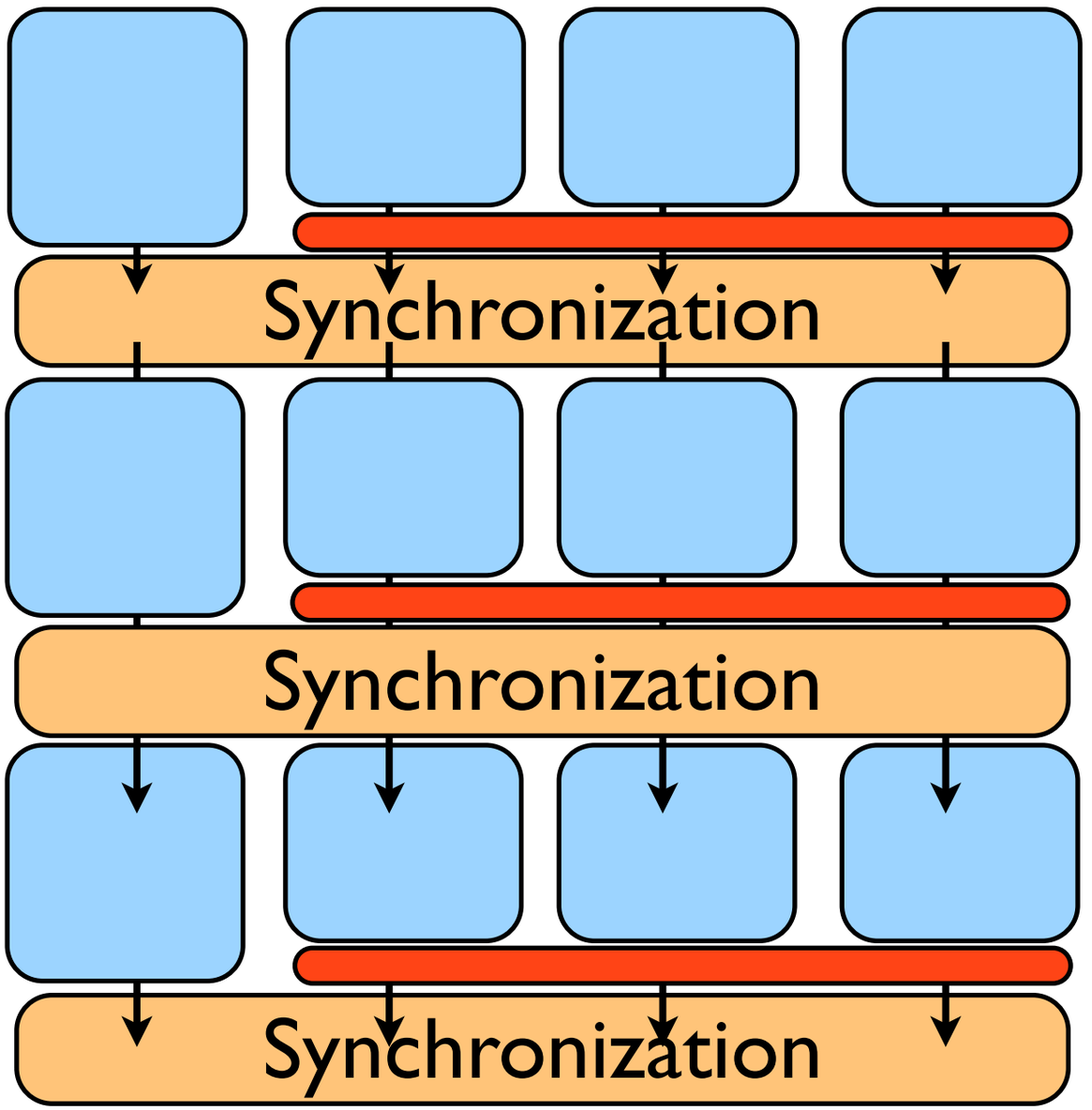
More on Concurrency

- Most problems are not pure concurrency
- Some level of synchronization, exchange of information needed between tasks
- This needs to be minimized
- Increases Amdahl's 'f'
- Are themselves costly



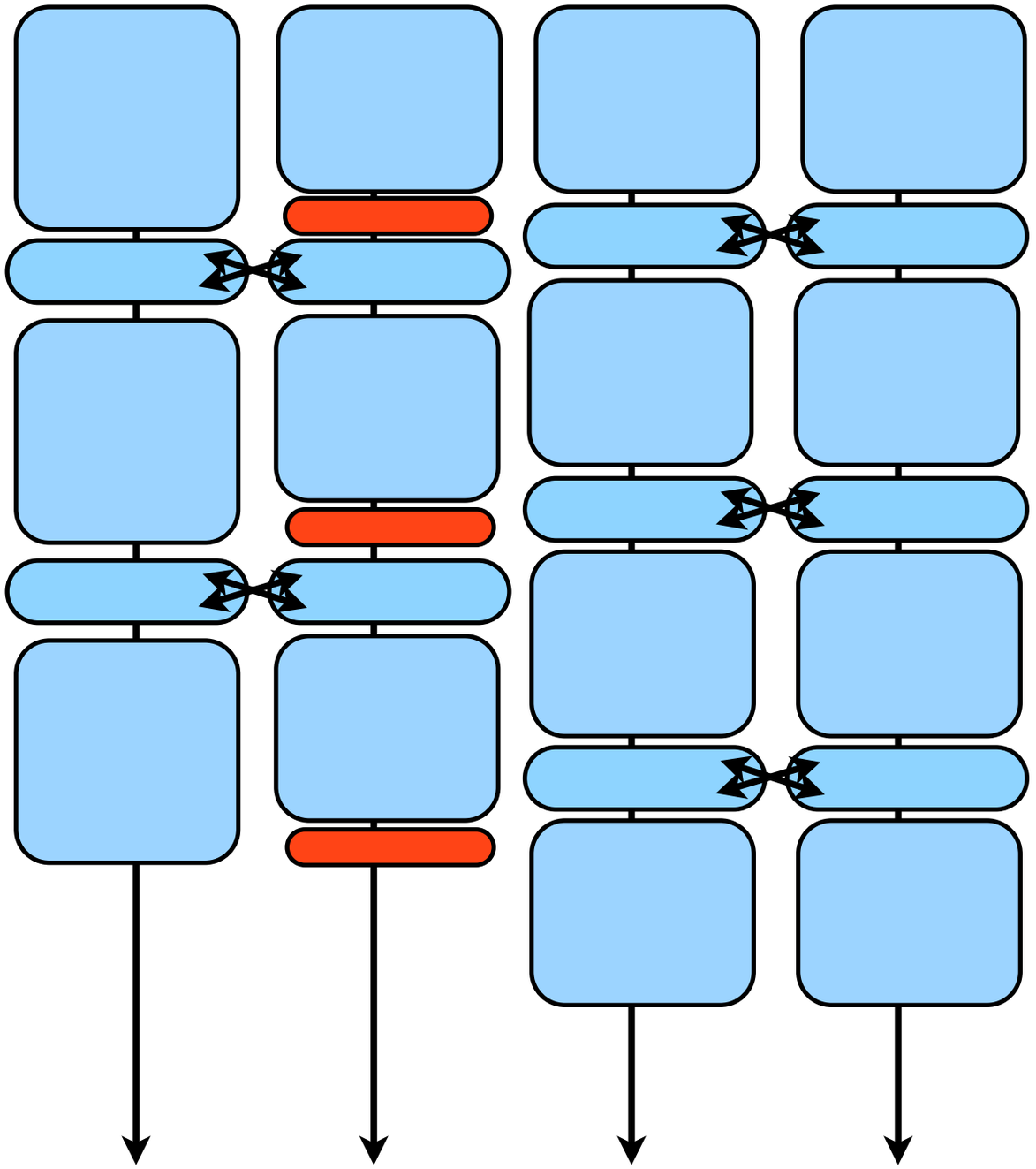
Concurrency

- Makes possible lots of wasted time ('load balancing', about which more later)



Locality

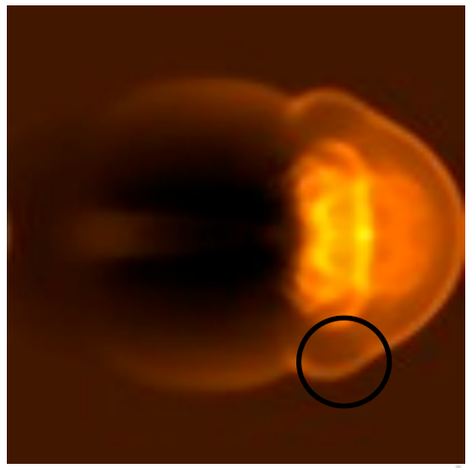
- Information needed by the task should be as local as possible.
- When tasks do need to interact, best that those interactions be as local as possible, and with as few others as possible
- Communications cost lower
- Fewer processes have are locked up during the necessary



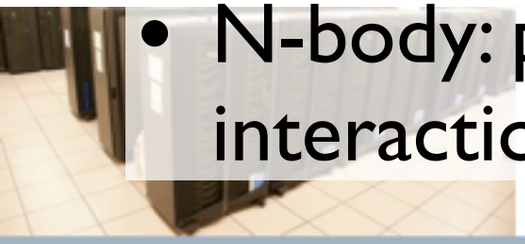
Big Lesson #2

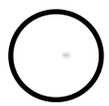
Parallel code design is about finding as much concurrency as possible, and arranging it in a way that maximizes locality.

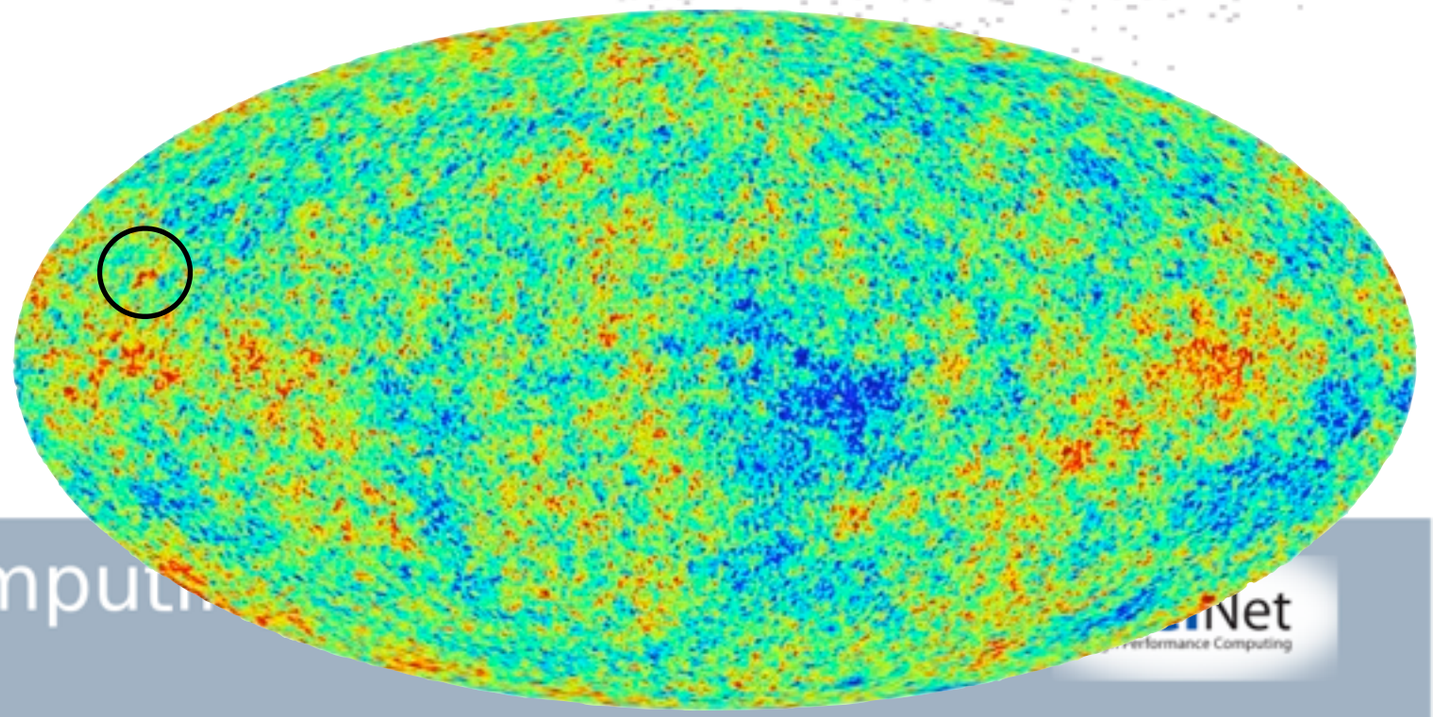
Finding Concurrency



- Identify tasks that can be done independently, order doesn't matter
- Our tasks - some options fairly clear
- Hydro: parts of domain
- Mapmaking: parts of map
- N-body: particles (or interactions)

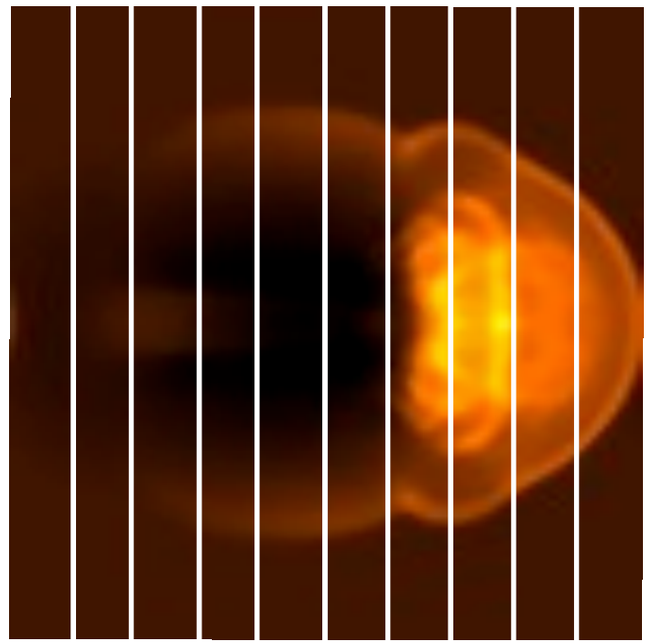




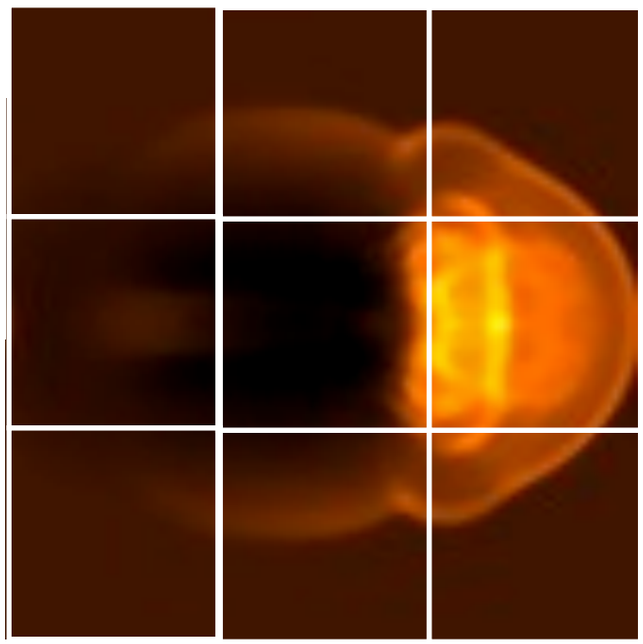


Maintaining Locality

- Now have to lump the concurrent bits into tasks
- Choosing that re-aggregation can greatly effect locality.



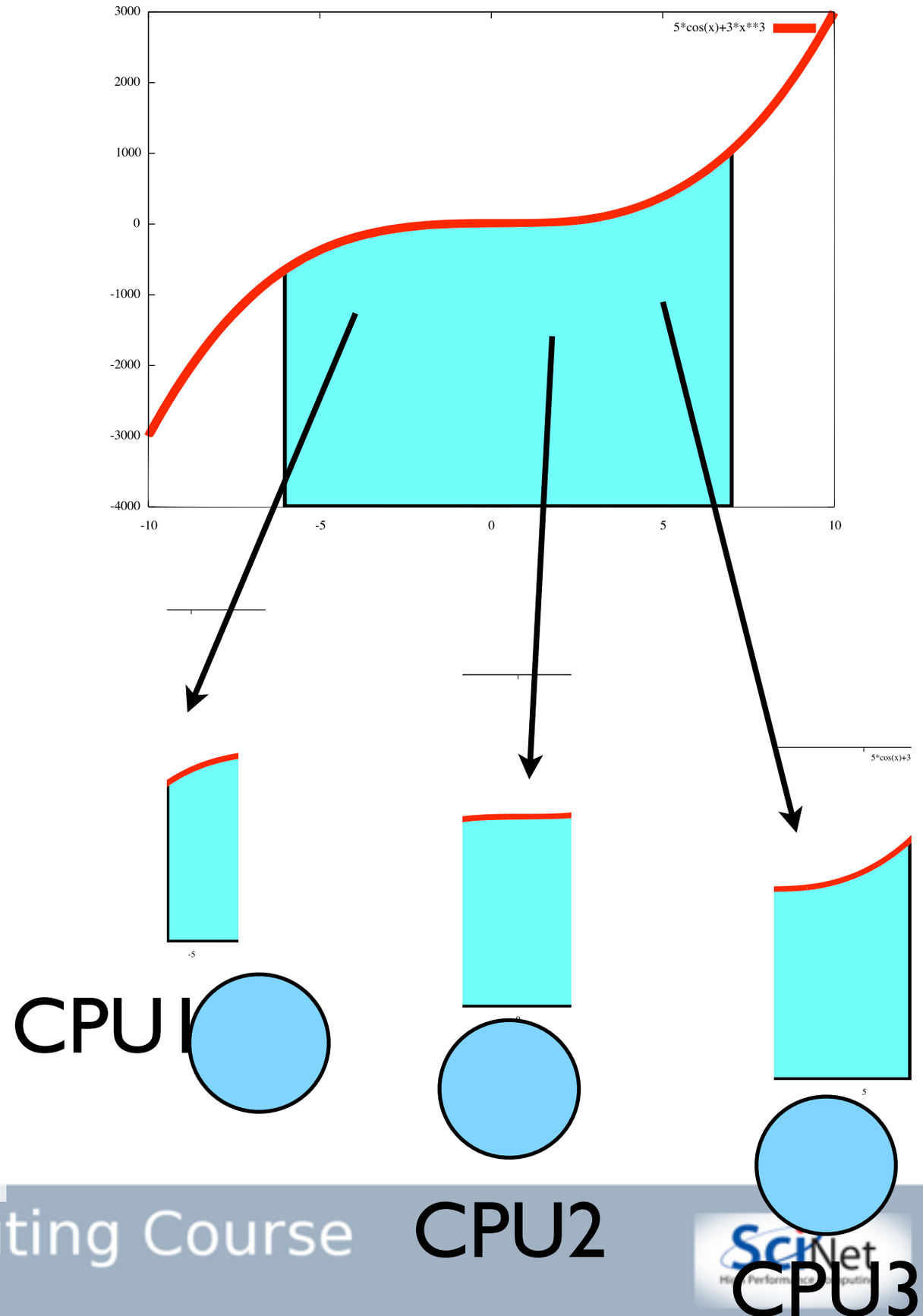
p = 9L



p = 4L

Example: 1d integration

- Integrate a 1d function with (say) Simpson's rule, with N points.
- Concurrency: can do each of the points independently, then sum.
- Locality: have each do a chunk

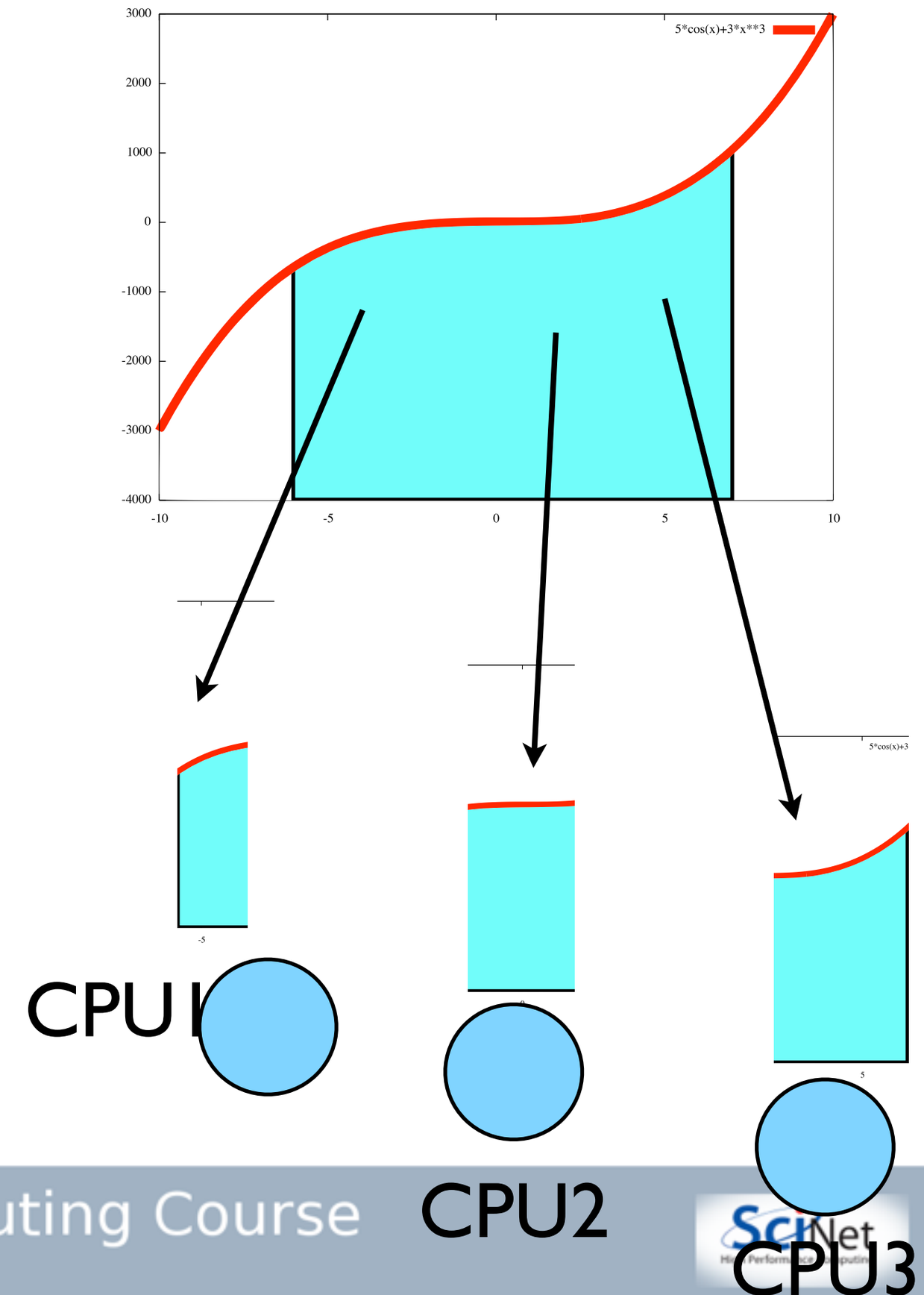


Example: 1d integration

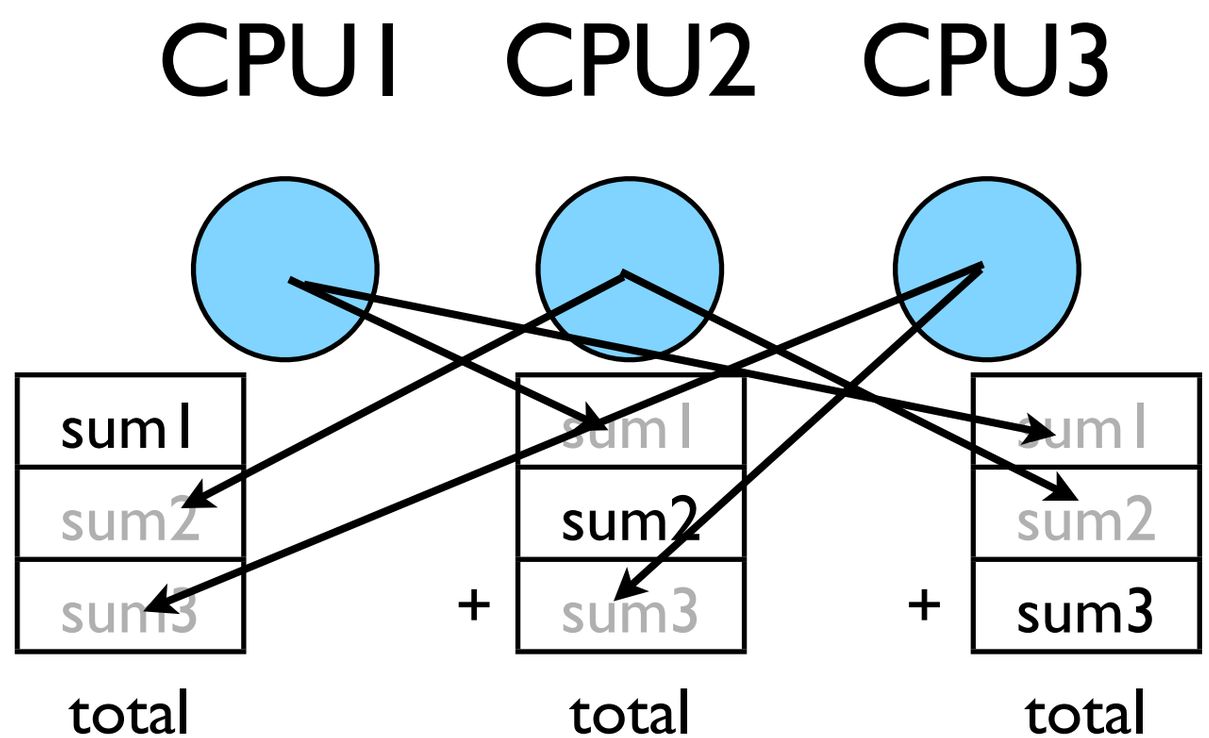
- Each processor gets N/P points to do
- Total compute time for one process:

$$T_{\text{comp}} = \left(\frac{N}{P}\right) N_{SR} C_{\text{comp}}$$

- Now how to do sums?



Example: 1d integration

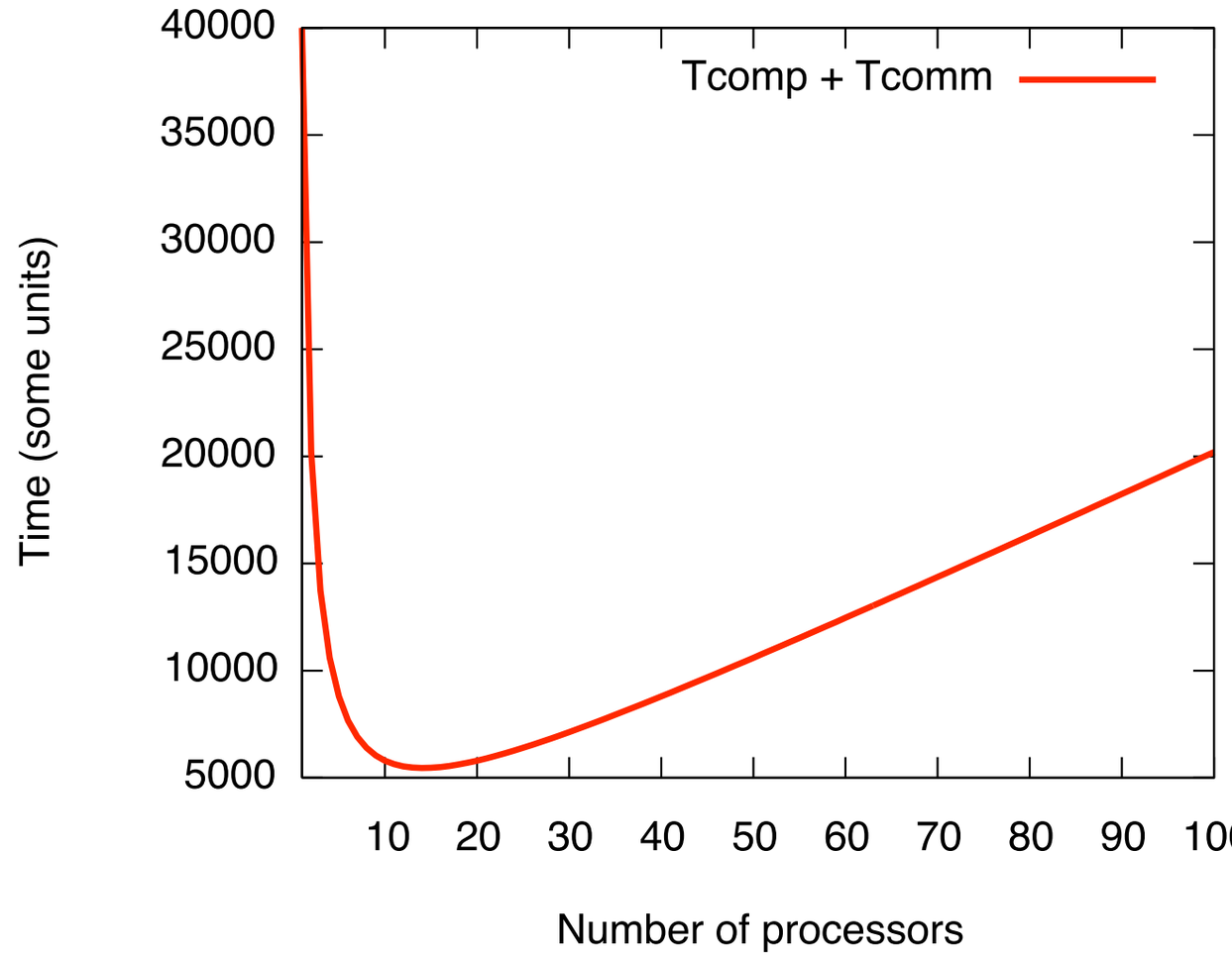


- Each processor sends partial sums to others, then all can do total
- Each processor sends its result (P-1) times and receives (P-1) results

$$T_{\text{comm}} = 2(P-1)C_{\text{comm}}$$

Integration **with** parallel costs:

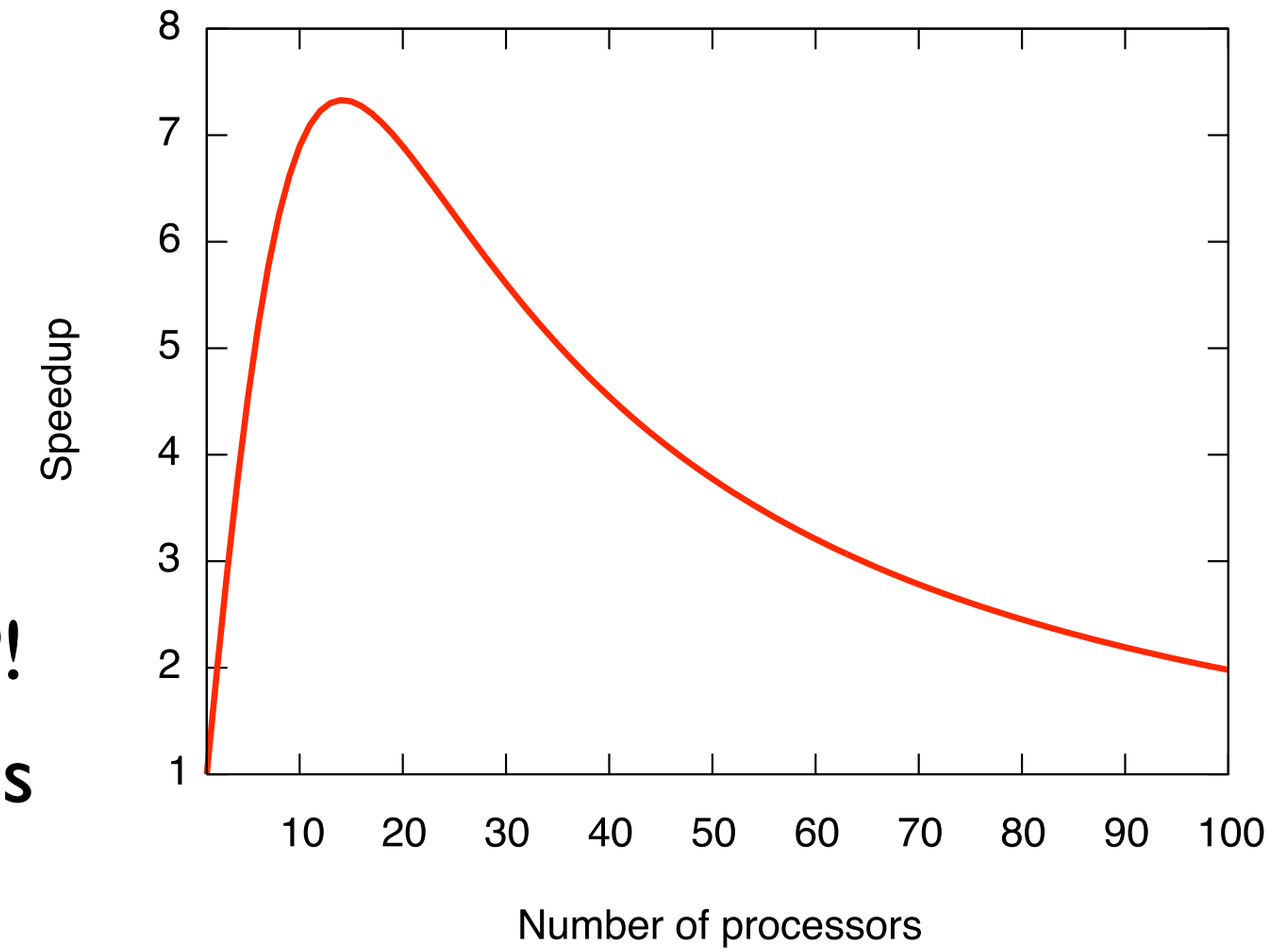
- Can actually get worse with P!
- Communication cost increases with P



$N = 10000$, $N_{sr}=4$,
$C_{comm}/C_{comp} = 100$

Integration **with** parallel costs:

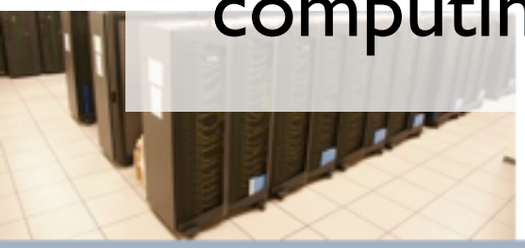
- Can actually get worse with P!
- Communication cost increases with P



$N = 10000$, $N_{sr}=4$,
$C_{comm}/C_{comp} = 100$

Communication -to- Computation ratio

- We want this to be (ideally) constant in P, or at least grow slowly; otherwise as we scale up, we spend more time sending messages than computing.



$$\frac{T_{\text{comm}}}{T_{\text{comp}}} = \frac{2(P-1)C_{\text{comm}}}{\frac{N}{P}N_{SR}C_{\text{comp}}}$$

$$= \frac{2P(P-1)}{N}\frac{1}{N_{\text{SR}}}\frac{C_{\text{comm}}}{C_{\text{comp}}}$$

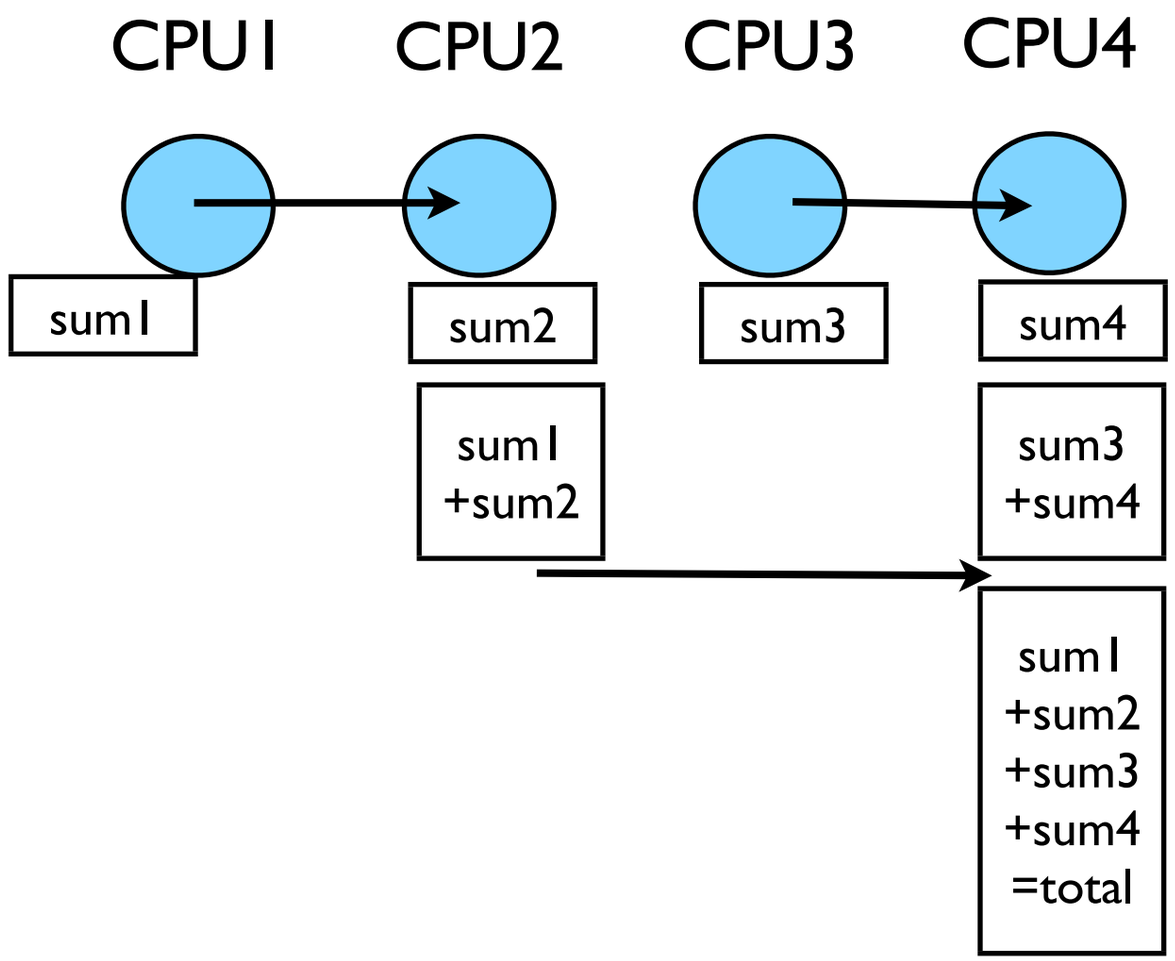
$$\sim P^2$$

If $N_{SR} \sim 4$, $C_{comm} \sim 1000\ C_{comp}$, N = 10000, then $T_{comm}/T_{comp} \sim 1.2$ for P=16

Better Summing

- Pairs of processors; send partial sums
- Total messages $\log_2(P)$
- Messages per proc; $\log_2(P)/P$
- Can repeat to send total back

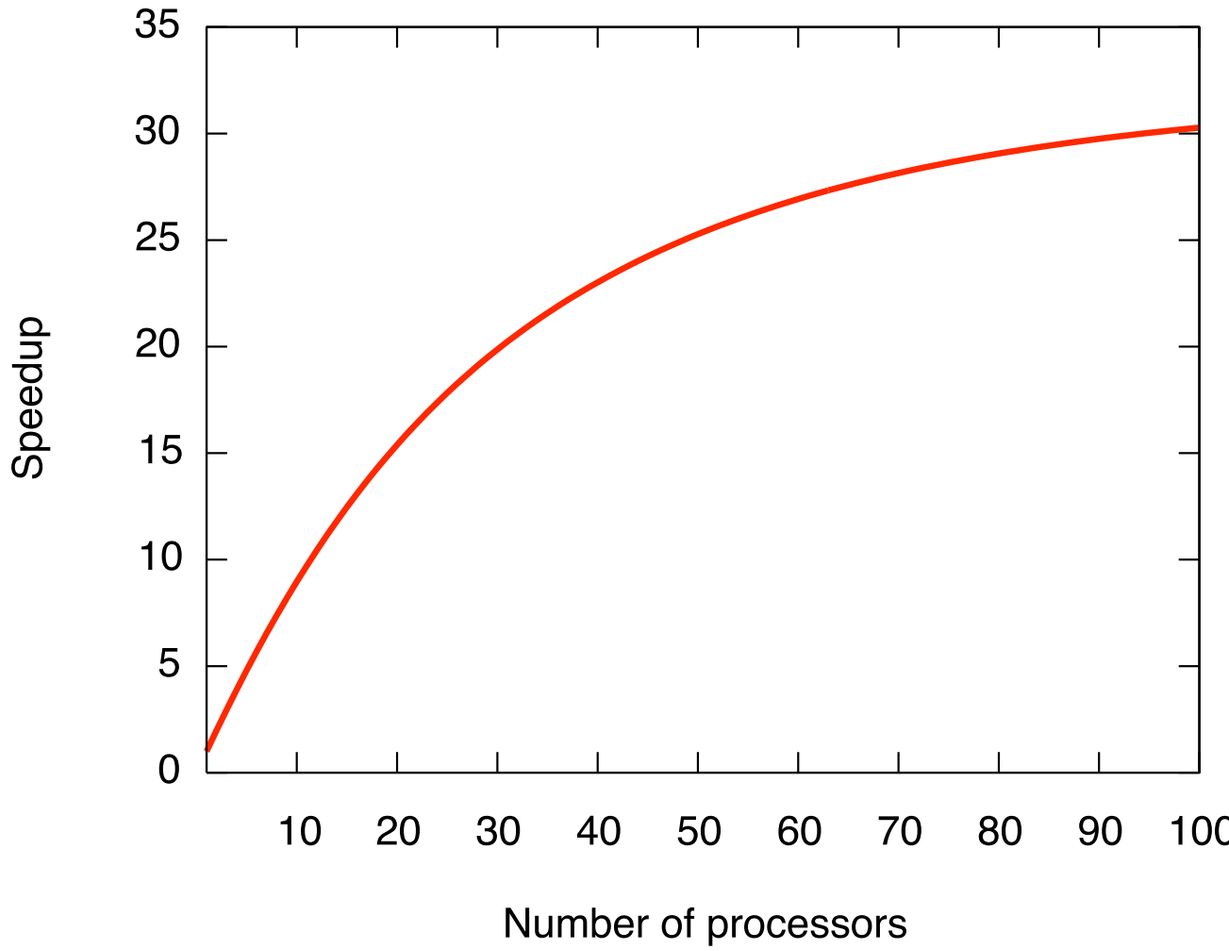
$$T_{\text{comm}} = 2\log_2(P)C_{\text{comm}}$$



Reduction; works for a variety of operators (+,*,min,max...)

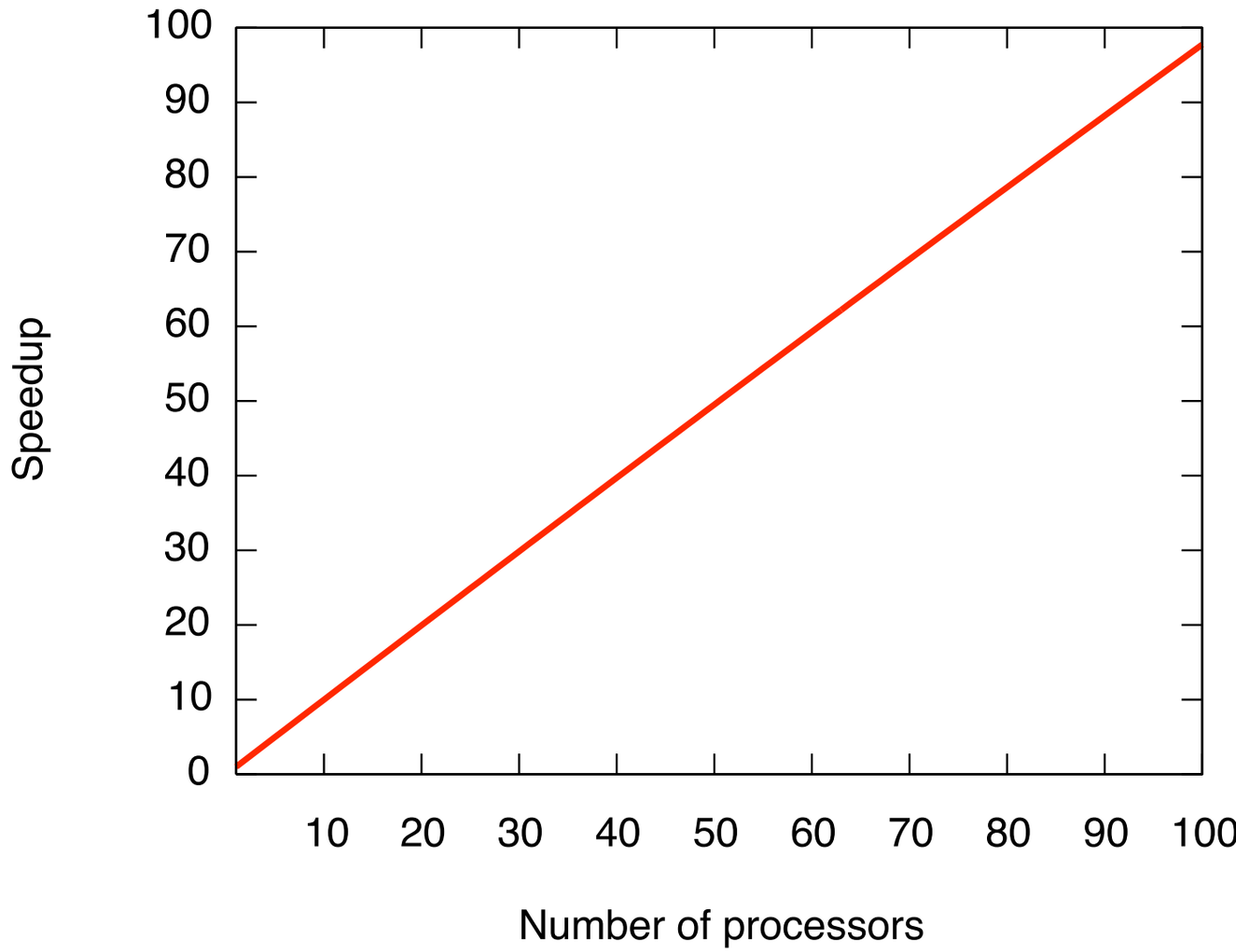
Speedup with reduction

- Very good! Efficiency still falling off past 20 or so processors
- (But integrating 10,000 numbers...)



Speedup with reduction

- with 1,000,000 numbers...



Communication -to- Computation ratio

$$\frac{T_{\text{comm}}}{T_{\text{comp}}} = \frac{2\log_2(P) C_{\text{comm}}}{\frac{N}{P} N_{SR} C_{\text{comp}}}$$

$$= \frac{2P\log_2(P)}{N} \frac{1}{N_{\text{SR}}} \frac{C_{\text{comm}}}{C_{\text{comp}}}$$

$$\sim P\log_2(P)$$

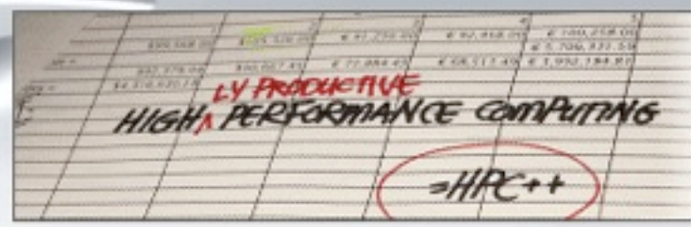
- Much better!
- As number of processors goes up, relative cost of communications goes up only logarithmically.

If $N_{SR} \sim 4$, $C_{comm} \sim 100\ C_{comp}$, N = 10000, then $T_{comm}/T_{comp} \sim 0.08$ for P=16

Parallel Computing

II: Parallel Computers

TOP500 SUPERCOMPUTER SITES



PROJECT | LISTS | STATISTICS | RESOURCES | NEWS

Home › Lists › June 2009

TOP500 List - June 2009 (1-100)

R_{max} and R_{peak} values are in TFlops. For more details about other fields, check the TOP500 description.

Power data in KW for entire system

next

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband / 2008 IBM	129600	1105.00	1456.70	2483.47
2	Oak Ridge National Laboratory United States	Jaguar - Cray XT5 QC 2.3 GHz / 2008 Cray Inc.	150152	1059.00	1381.40	6950.60
3	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution / 2009 IBM	294912	825.50	1002.70	2268.00

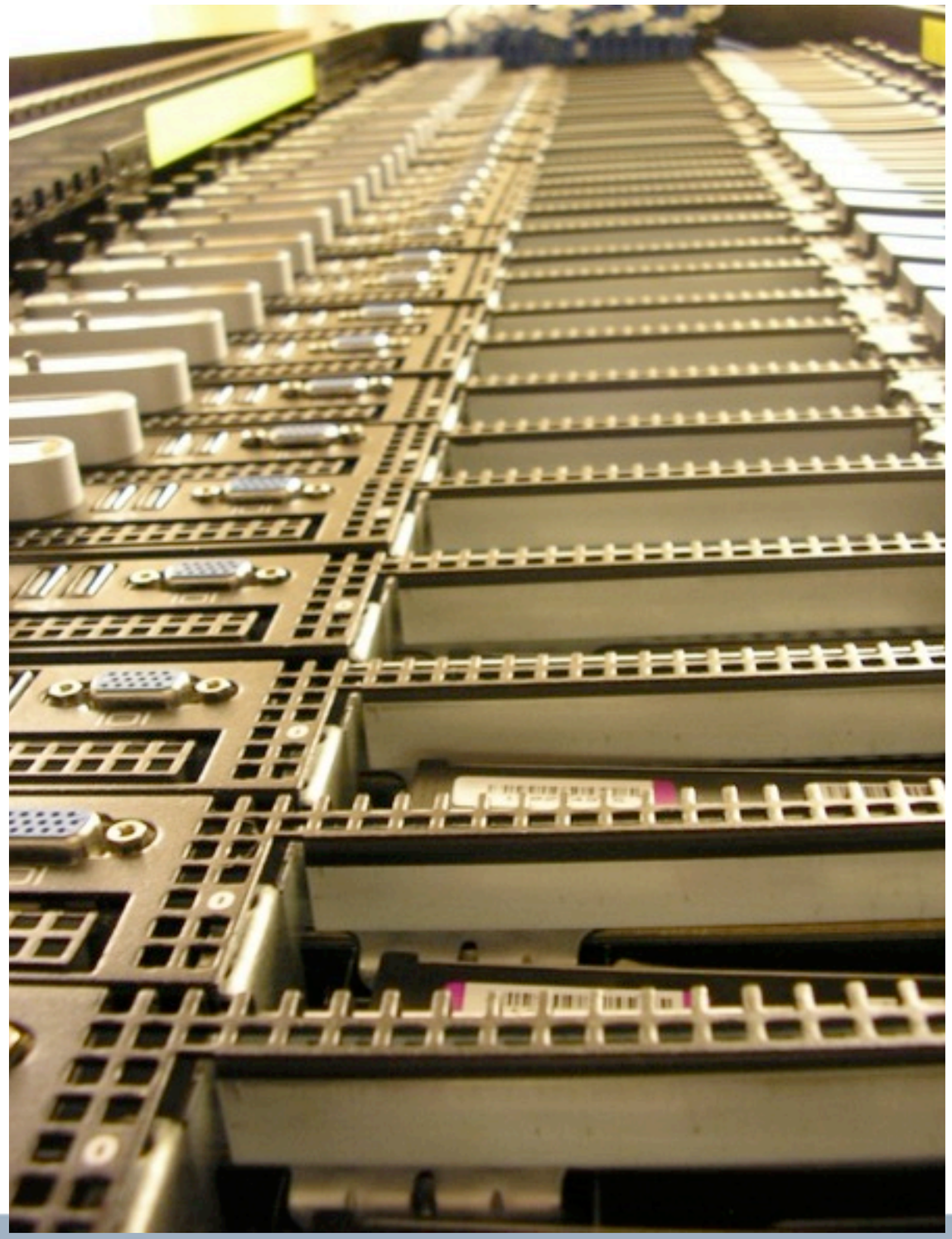
Top500.org:

List updated every 6 months of the worlds 500 largest supercomputers.

Info about architecture, ...

1 Petaflop (10^{15} flop/s); 126,600 cores

Computer Architectures



- How the computers work shape how best to progam them
- Shared Memory vs Distributed Memory.
- Vector computers...

Distributed Memory: Clusters

- Simplest type of parallel computer to build

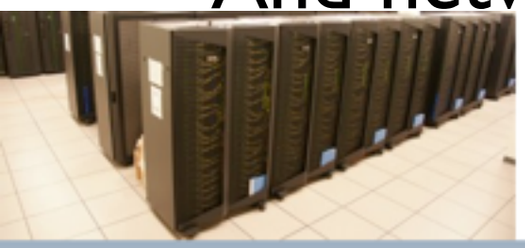
Distributed Memory: Clusters

http://flickr.com/photos/eurleif/

- Simplest type of parallel computer to build
- Take existing powerful standalone computers

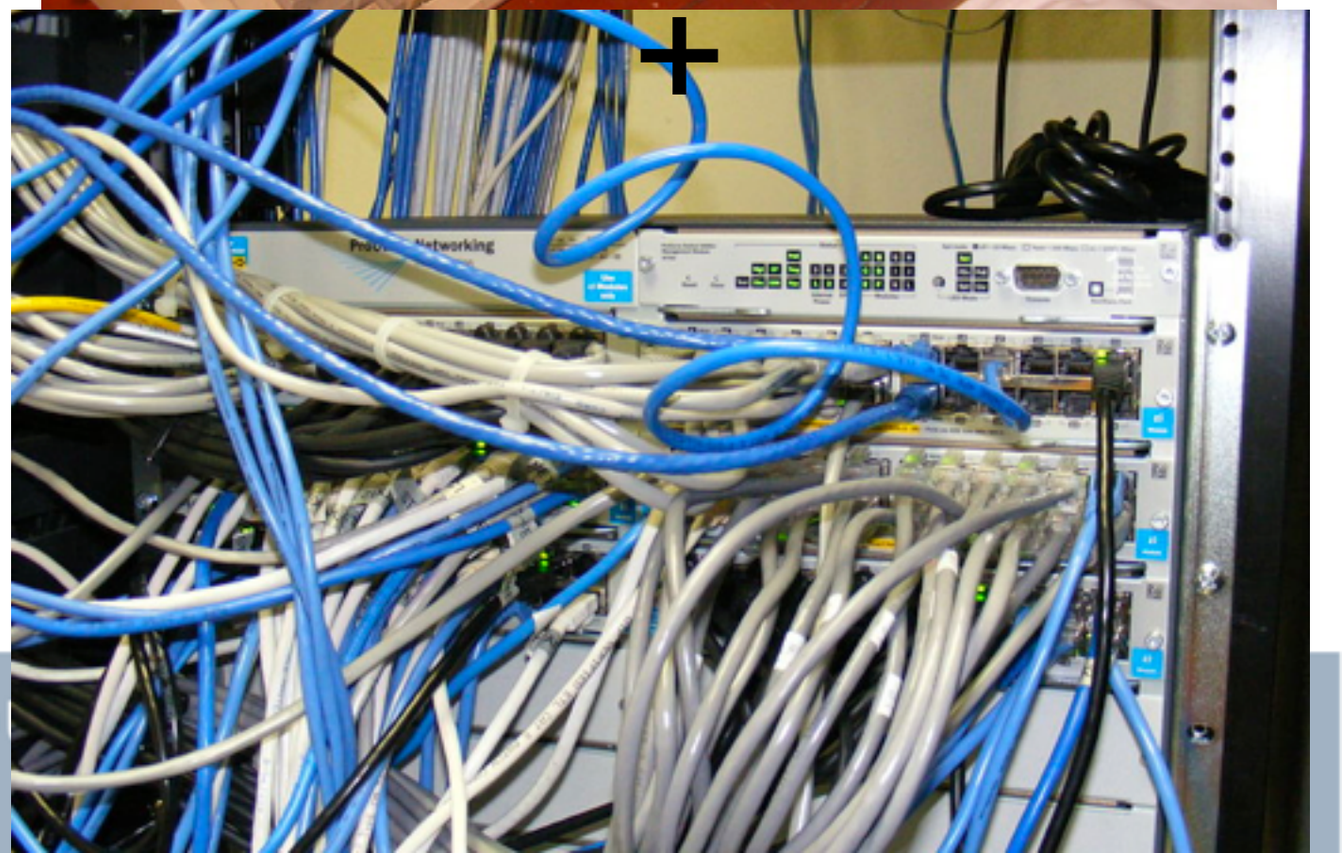
Distributed Memory: Clusters

- Simplest type of parallel computer to build
- Take existing powerful standalone computers
- And network them



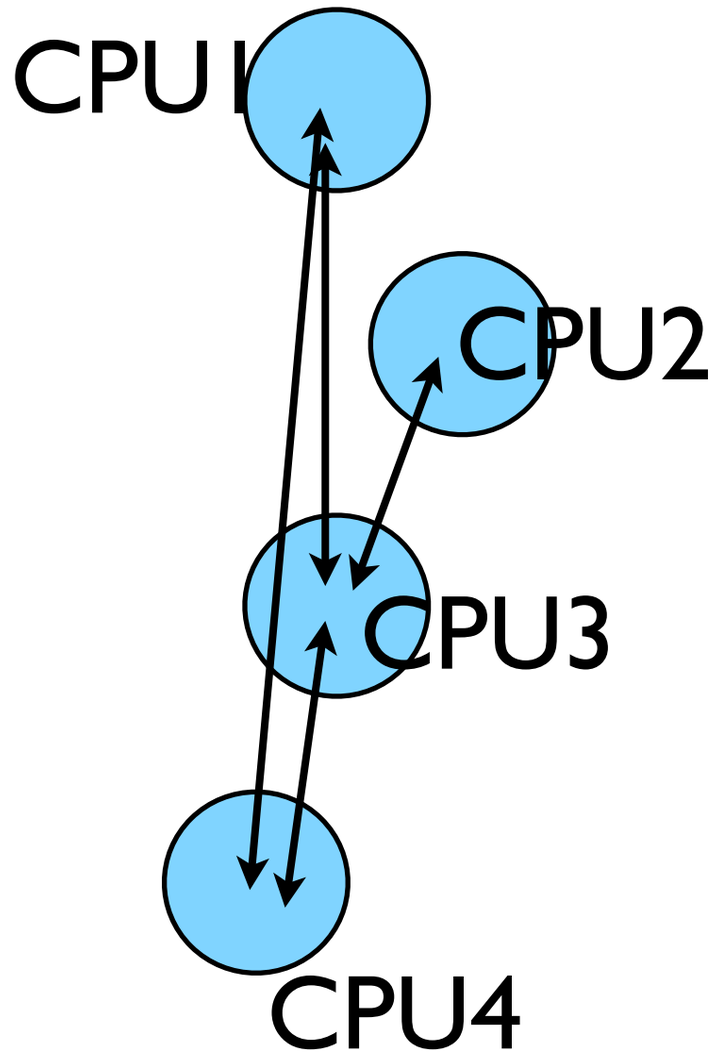
http://flickr.com/photos/eurleif/

+



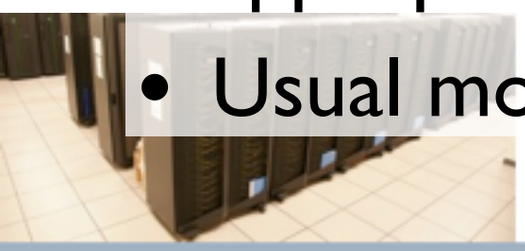
Each Node is Independent

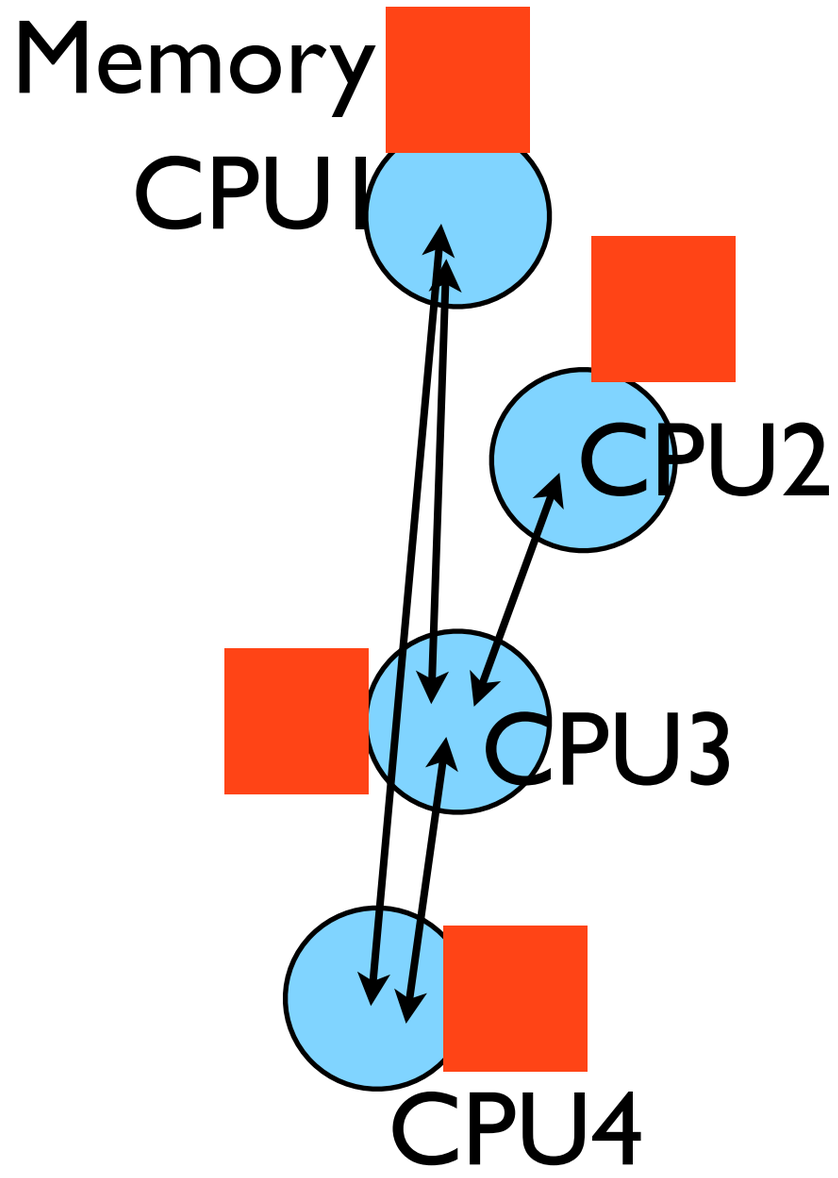
- Parallel code consists of programs running on separate computers, communicating with each other
- *Could* be entirely different programs



Each node has independent memory

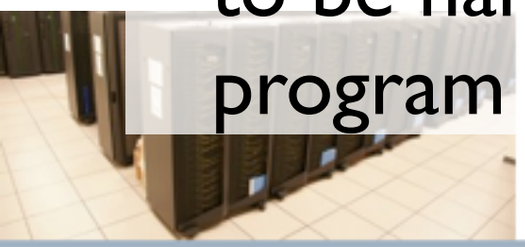
- Locally stores its own portion of problem
- Whenever it needs information from another region, requests it from appropriate CPU
- Usual model: 'message passing'

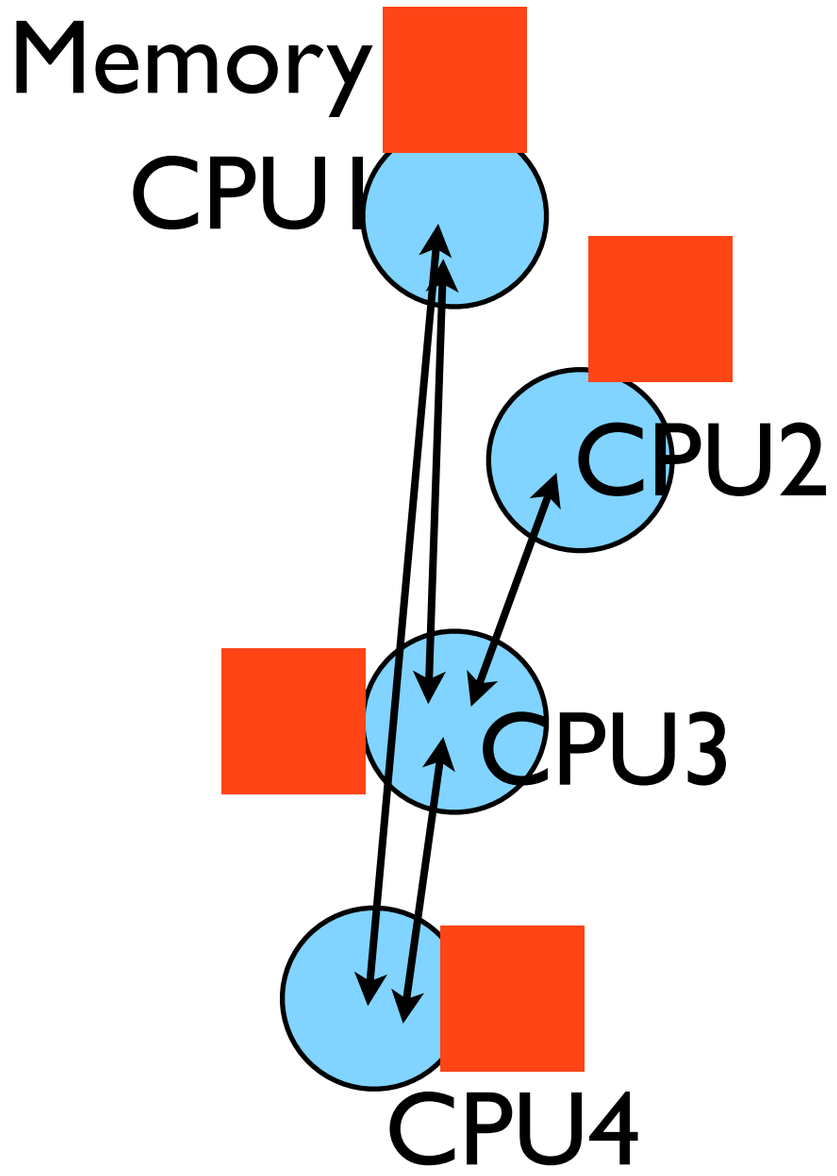




Clusters +Message Passing

- HW: Easy to build (harder to build *well*)
- HW: Can build larger and larger clusters relatively easily
- SW: Every communication has to be hand coded -- hard to program



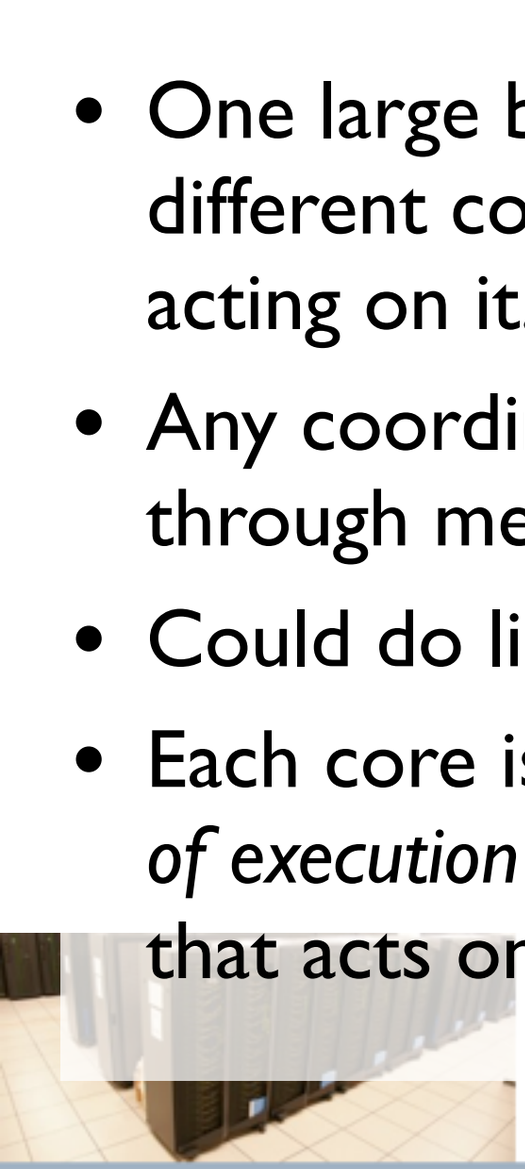


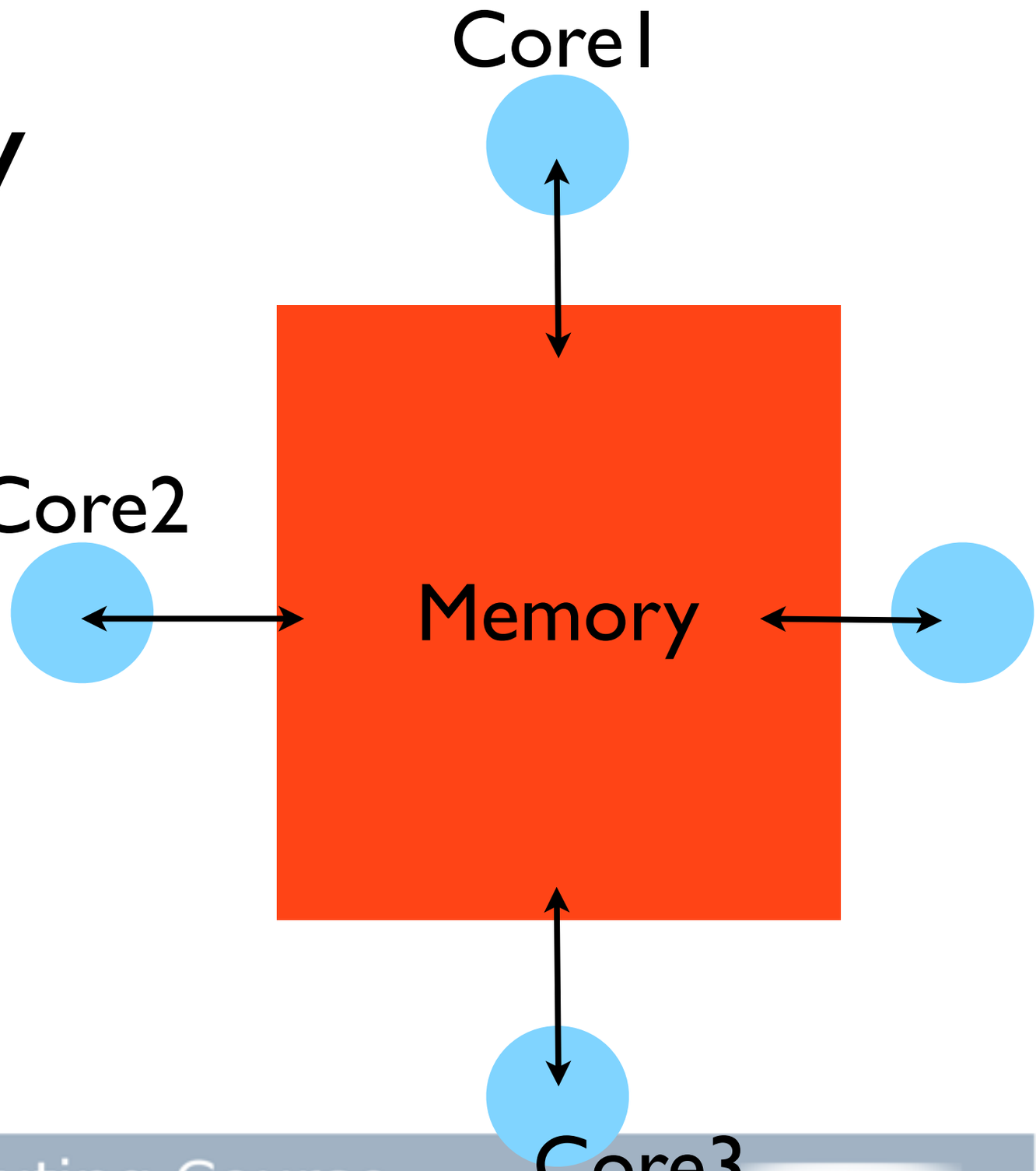
	Latency	Bandwidth
GigE	~10 μs	1 Gb/s (~60 ns/double)
Infiniband	~2 μs	2-10 Gb/s (~10 ns/double)

Processor speed: 1 FLOP ~ few ns or less

Shared Memory

- One large bank of memory, different computing cores acting on it. All 'see' same data
- Any coordination done through memory.
- Could do like before, but why?
- Each core is assigned a *thread of execution* of a single program that acts on the data





Thread Vs. Process

- Processes: Independant tasks with their own memory, resources
- Threads: Threads of execution within one process, 'seeing' the same memory, etc.

OMP Threads

```
ljdursi@gpc
File  Edit  View  Terminal  Tabs  Help
top - 17:27:34 up 2 days,  1:40,  1 user,  load average: 1.81, 0.56, 0.20
Tasks: 142 total,   3 running, 139 sleeping,   0 stopped,   0 zombie
Cpu(s): 95.9%us,  3.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.1%hi,  1.0%si,  0.0%
Mem:  16411872k total,  2778368k used, 13633504k free,      256k buffers
Swap:        0k total,        0k used,        0k free,  2265652k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
18121 ljdursi   25   0 89536 1076  840 R 779.0  0.0   0:29.01 diffusion-omp
17193 root      15   0 35300 2580   60 S 15.0  0.0   0:01.57 pbs_mom
17192 root      15   0 35300 3216  696 R  6.0  0.0   0:00.48 pbs_mom
    1 root      15   0 10344  740  612 S  0.0  0.0   0:01.45 init
    2 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/0
    4 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/0
    5 root      RT  -5     0    0    0 S  0.0  0.0   0:00.01 migration/1
    6 root      34  19     0    0    0 S  0.0  0.0   0:00.01 ksoftirqd/1
    7 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/1
    8 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migration/2
    9 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/2
   10 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/2
   11 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migration/3
```

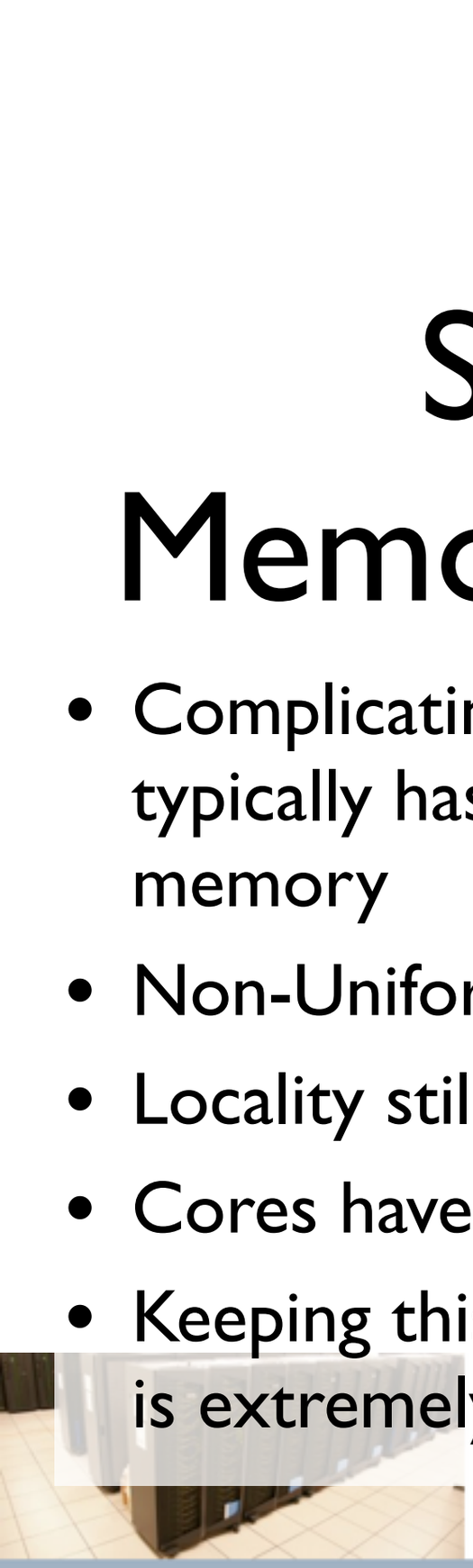
MPI Procs

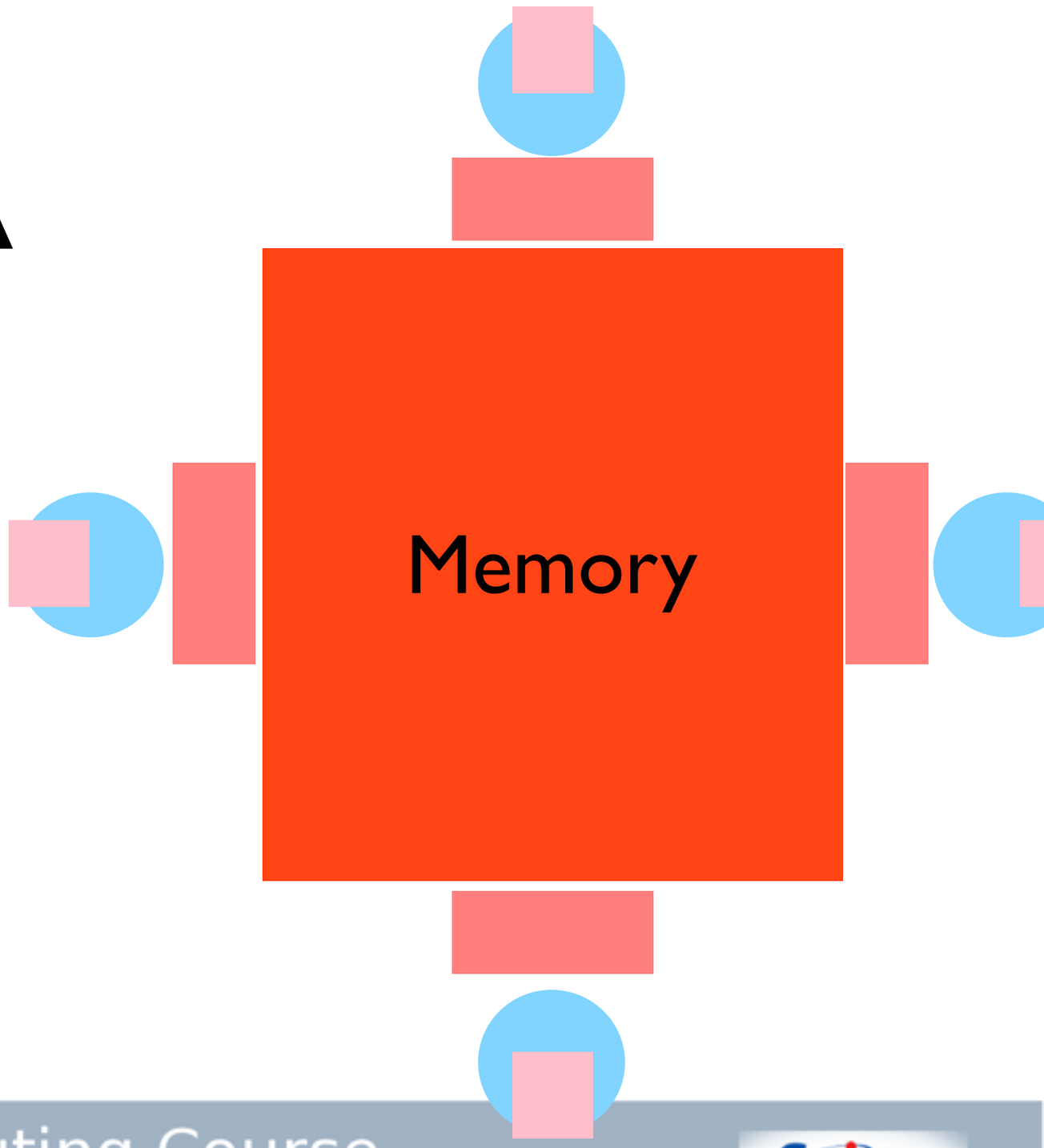
```
ljdursi@gp
File  Edit  View  Terminal  Tabs  Help
top - 17:33:58 up 2 days,  1:47,  1 user,  load average: 0.80, 0.31, 0.17
Tasks: 150 total,   9 running, 141 sleeping,   0 stopped,   0 zombie
Cpu(s):100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%
Mem:  16411872k total,  2801172k used, 13610700k free,      256k buffers
Swap:        0k total,        0k used,        0k free,  2268568k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
18393 ljdursi   25   0  187m 5504 3484 R 100.2  0.0   0:05.45 diffusion-mpi
18395 ljdursi   25   0  187m 5512 3492 R 100.2  0.0   0:05.46 diffusion-mpi
18397 ljdursi   25   0  187m 5508 3488 R 100.2  0.0   0:05.46 diffusion-mpi
18392 ljdursi   25   0  187m 5580 3556 R 99.9  0.0   0:05.40 diffusion-mpi
18394 ljdursi   25   0  187m 5504 3488 R 99.9  0.0   0:05.45 diffusion-mpi
18396 ljdursi   25   0  187m 5512 3492 R 99.9  0.0   0:05.45 diffusion-mpi
18398 ljdursi   25   0  187m 5500 3480 R 99.9  0.0   0:05.43 diffusion-mpi
18399 ljdursi   25   0  187m 5512 3492 R 99.9  0.0   0:05.46 diffusion-mpi
    1 root      15   0 10344  740  612 S  0.0  0.0   0:01.45 init
    2 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/0
    4 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/0
    5 root      RT  -5     0    0    0 S  0.0  0.0   0:00.01 migration/1
    6 root      34  19     0    0    0 S  0.0  0.0   0:00.01 ksoftirqd/1
```

Shared Memory:NUMA

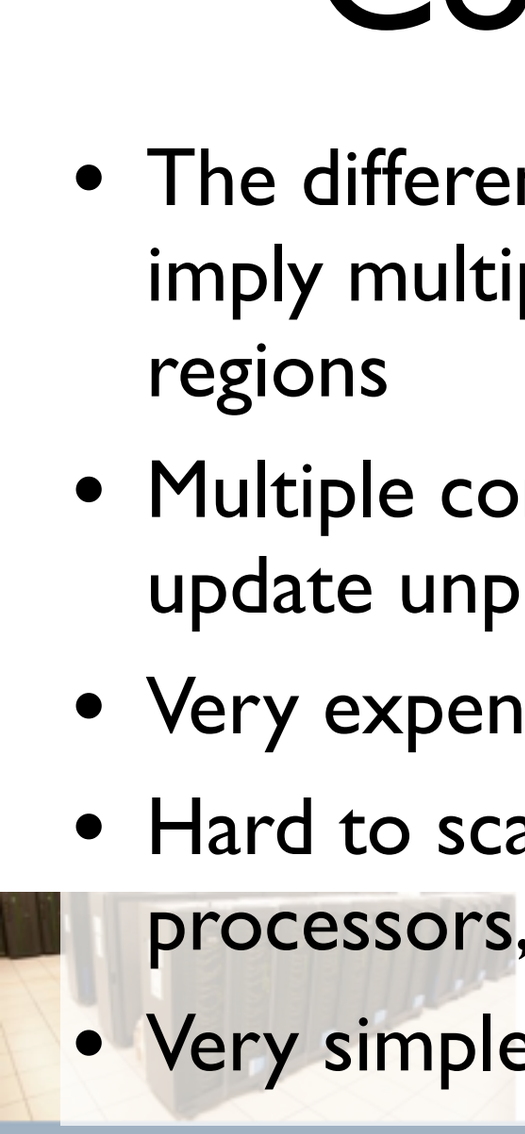
- Complicating things: each core typically has some of its own memory
- Non-Uniform Memory Access
- Locality still matters
- Cores have cache, too.
- Keeping this memory *coherent* is extremely challenging

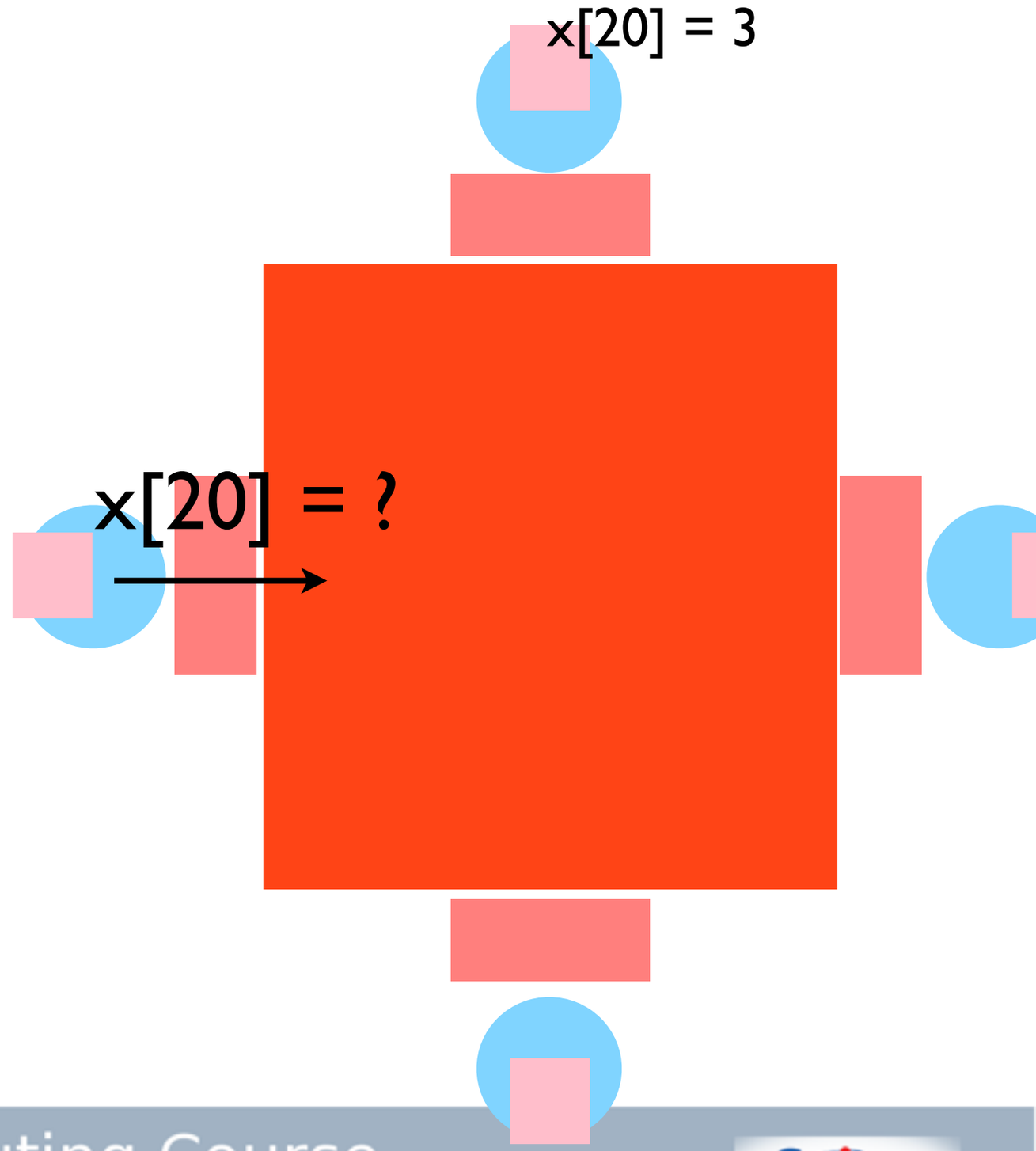




Coherency

- The different levels of memory imply multiple copies of some regions
- Multiple cores mean can update unpredictably
- Very expensive hardware
- Hard to scale up to lots of processors, very $$$
- Very simple to program!!





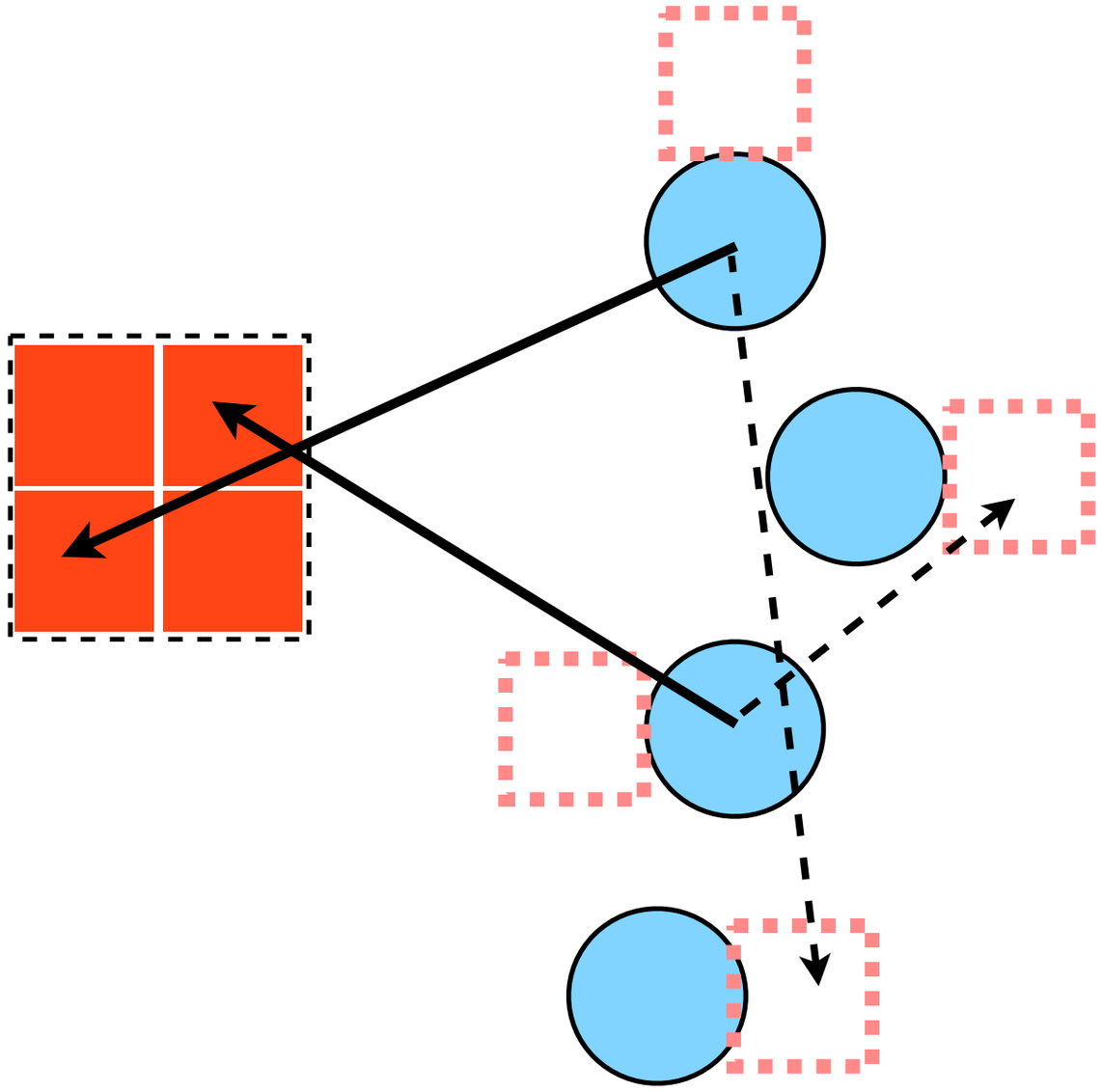
	Latency	Bandwidth
GigE	~10 μs	1 Gb/s (~60 ns/double)
Infiniband	~2 μs	2-10 Gb/s (~10 ns/double)
NUMA Shared Mem	~0.1 μs	10-20 Gb/s (~4 ns/double)

Processor speed: 1 FLOP ~ ns or less

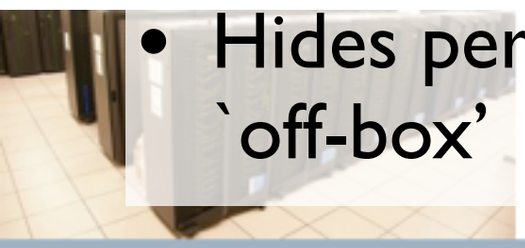
Big Lesson #3

The best approach to parallelizing your problem will depend on both details of your problem and of the hardware available.

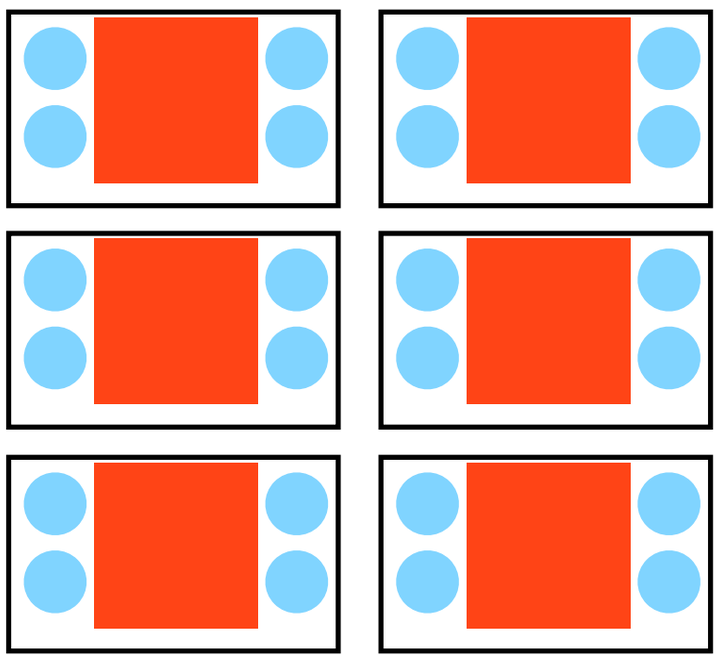
Distributed Shared Memory



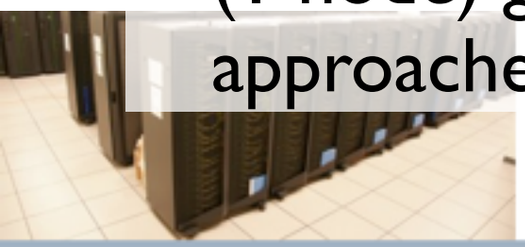
- Several attempts at making cluster memories look like a big shared memory
- Coherence much harder
- Large overhead
- Hides performance cost of going `off-box'



Hybrid Architectures



- Almost all of the biggest computers are now clusters of shared memory nodes
- Generally just use message passing across all cores, but as P (1 node) goes up, hybrid approaches start to make sense.



Hands On I

- Due by start of this afternoon
- 'Submit' by leaving files in a subdirectory '`hw1`' on the cluster

- `mkdir ~/hw1`
- Calculate the speedup as a function of P,N for the better summation example. Put in 'speedup. txt'
- Where would you expect performance to turn over on a modern machine using Infiniband? Shared memory? GigE?
- cd ~/pca/src/gettingstarted/ and make `omp_hello_world` and run it
- make `mpi_hello_world` and run it
- qsub -I -X into your reserved node and ensure this works
- Put all outputs in the `hw1` directory.