

Introduction to Practical Parallel Programming

Course Overview, and The 'Big Picture'

Today's Main Goal

- Students arriving with scientific computing background should be able to leave and immediately *start* parallelizing codes, understand concepts involved

Schedule

9am-10:30	Basic Concepts
10:30-10:45	Break
10:45-12:45	Intro to OpenMP
12:45-1:45	Lunch
1:45-3:30	Intro to MPI
3:30-3:45	Break
3:45-5:00	Intro to MPI 2

What will we be doing here

- This is a short course on parallel programming
- You will be doing a lot of typing and programming to help build skills with OpenMP, MPI.

```
return 0;
int mpi_bc(domain_t *d, int dirn, int neigh) {
    /* first, ugly version which copies into a buffer

    double *out, *in;
    int count;
    int nx, ny, ng;
    int i, j, ioff, joff;
    int ierr;
    int bufsize;
    MPI_Status status;

    nx = d->Nx; ny = d->Ny; ng = d->Nguard;
    switch (dirn) {
        case XRIGHT_DIR: otherdirn = XLEFT_DIR;
        case XLEFT_DIR: otherdirn = XRIGHT_DIR;
        case YRIGHT_DIR: otherdirn = YLEFT_DIR;
        case YLEFT_DIR: otherdirn = YRIGHT_DIR;
    }

    if (dirn == XRIGHT_DIR || dirn == XLEFT_DIR) {
        bufsize = ng*(ny+2*ng)*NVAR;
        out = (double *)malloc(bufsize*sizeof(double));
        in = (double *)malloc(bufsize*sizeof(double));
        if (out == NULL || in == NULL) {
```

Parallel Computing

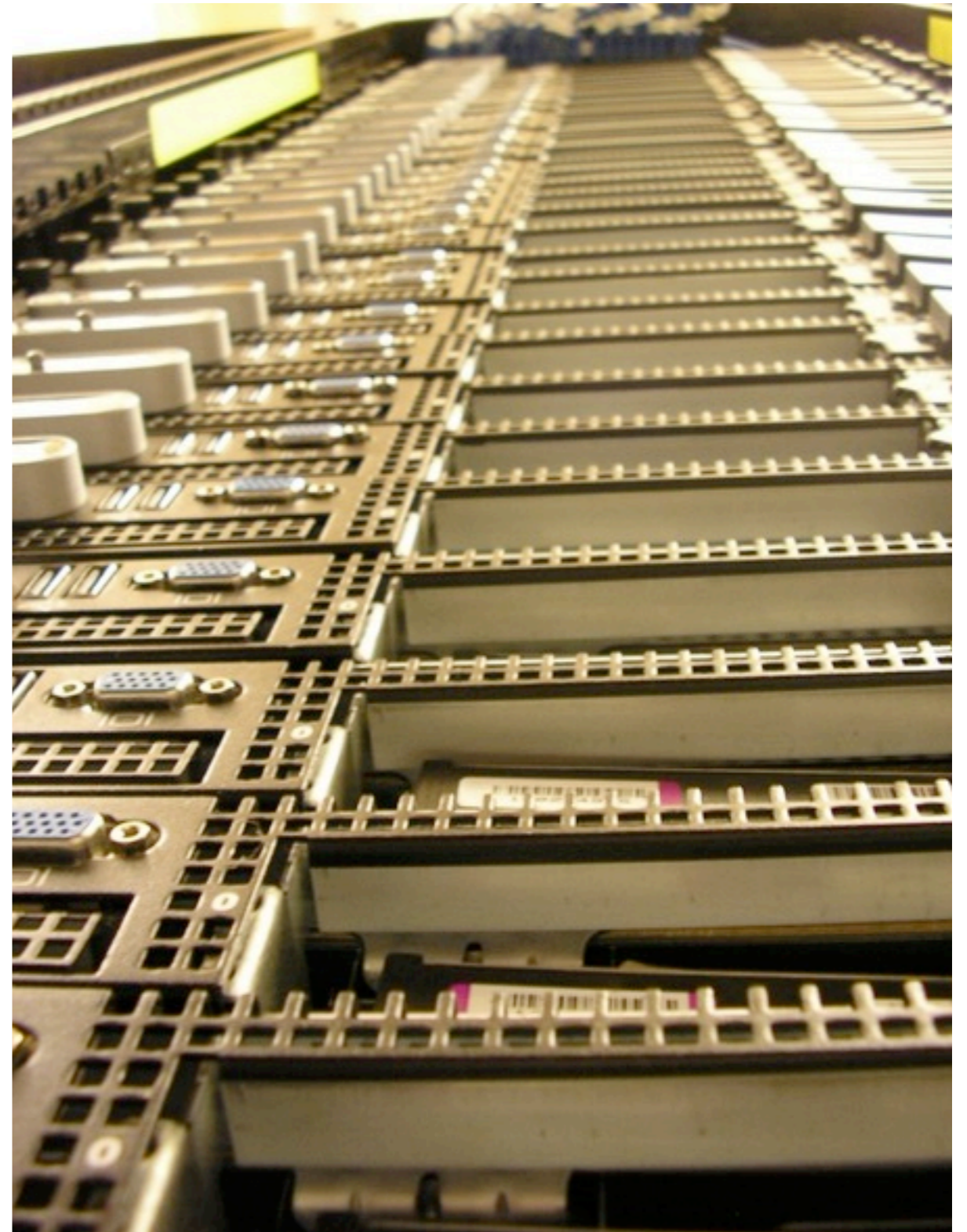
I: Concurrency, Amdahl's Law, and Locality

Why Parallel Computing?

Faster:

At any given time, there is a limit as to **how fast** one computer can compute.

So use more computers!

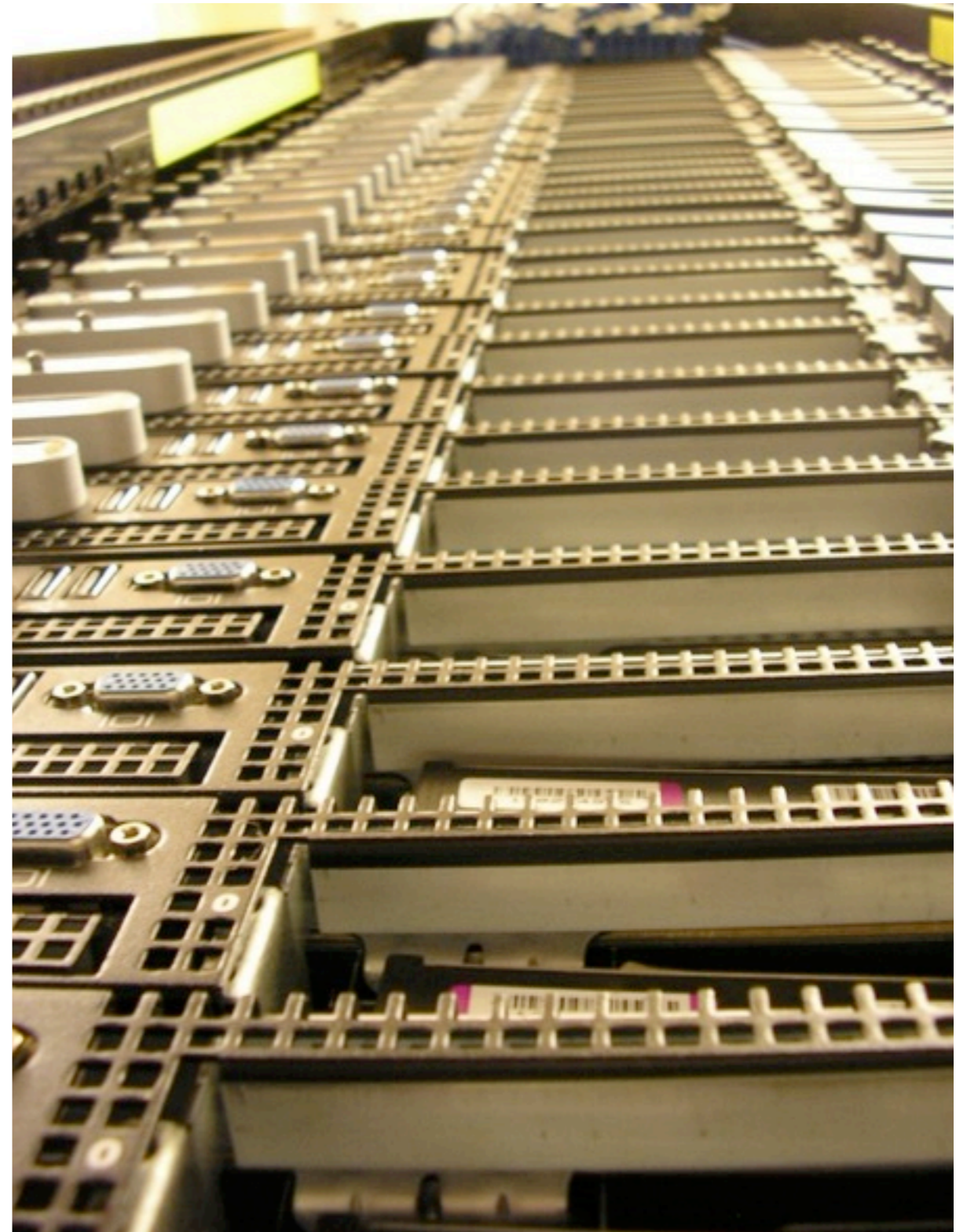


Why Parallel Computing?

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!

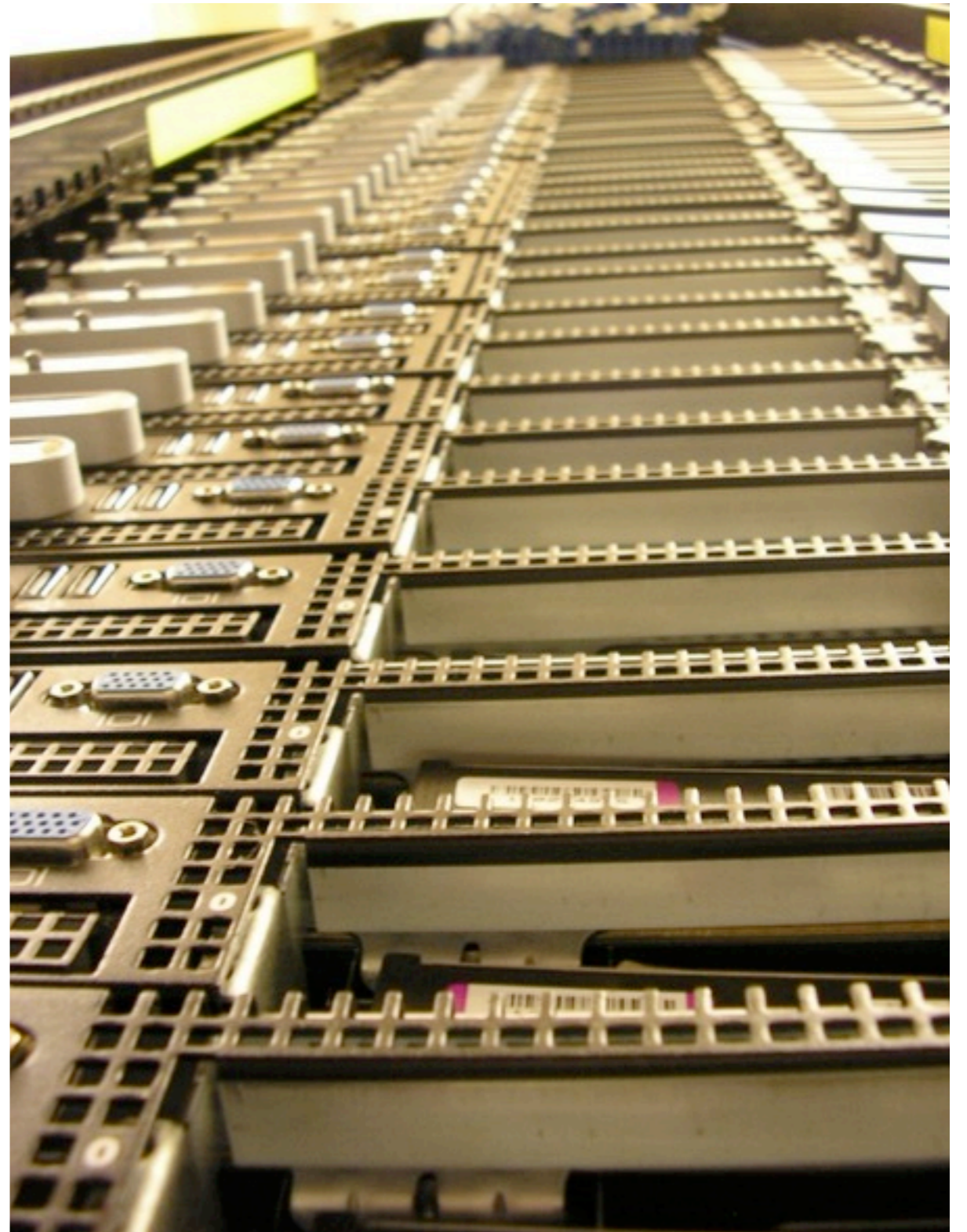


Why Parallel Computing?

More:

You have a program that runs in reasonable time on one processor but you want to run it **thousands of times**.

So use more computers!



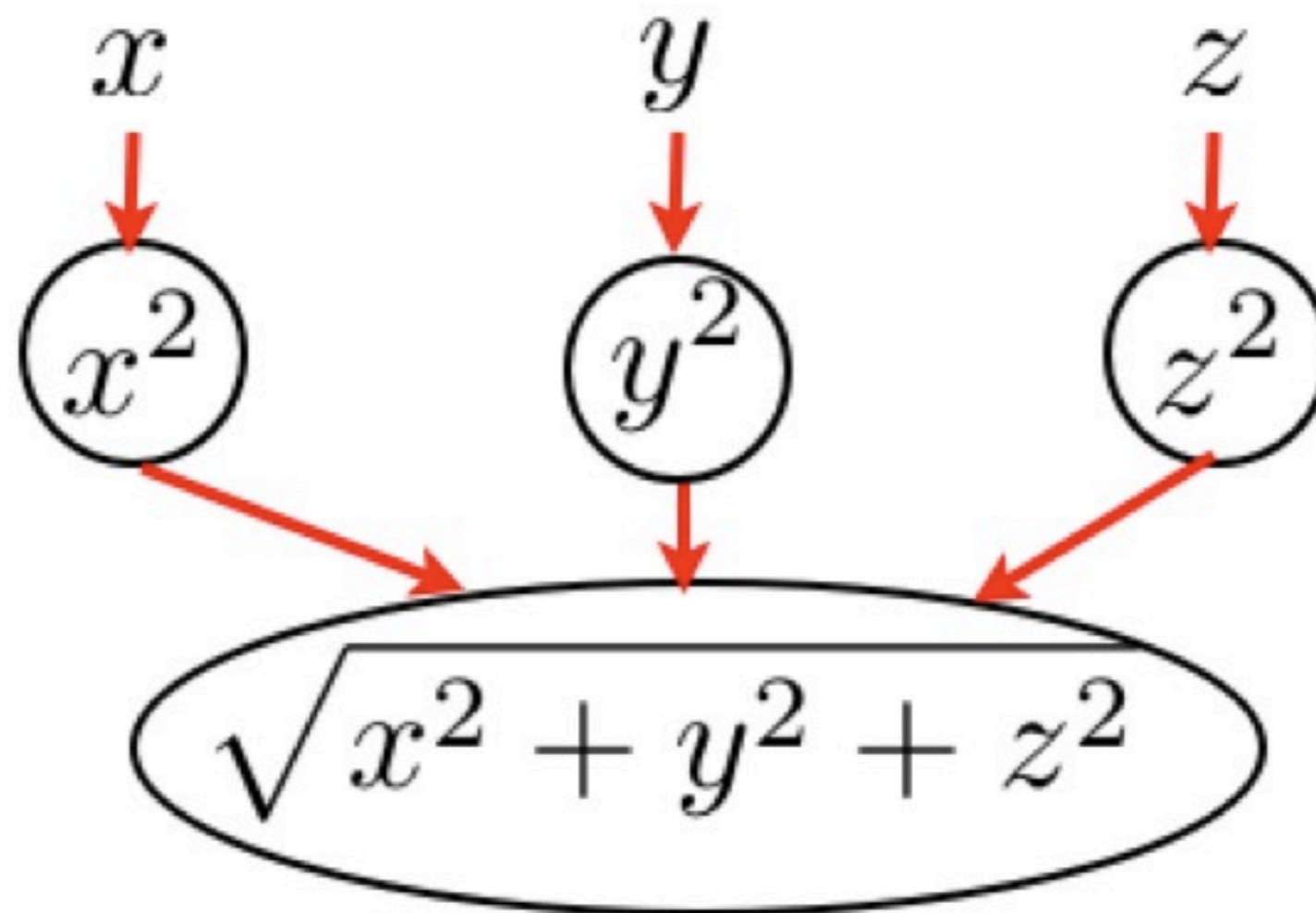
Concurrency

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



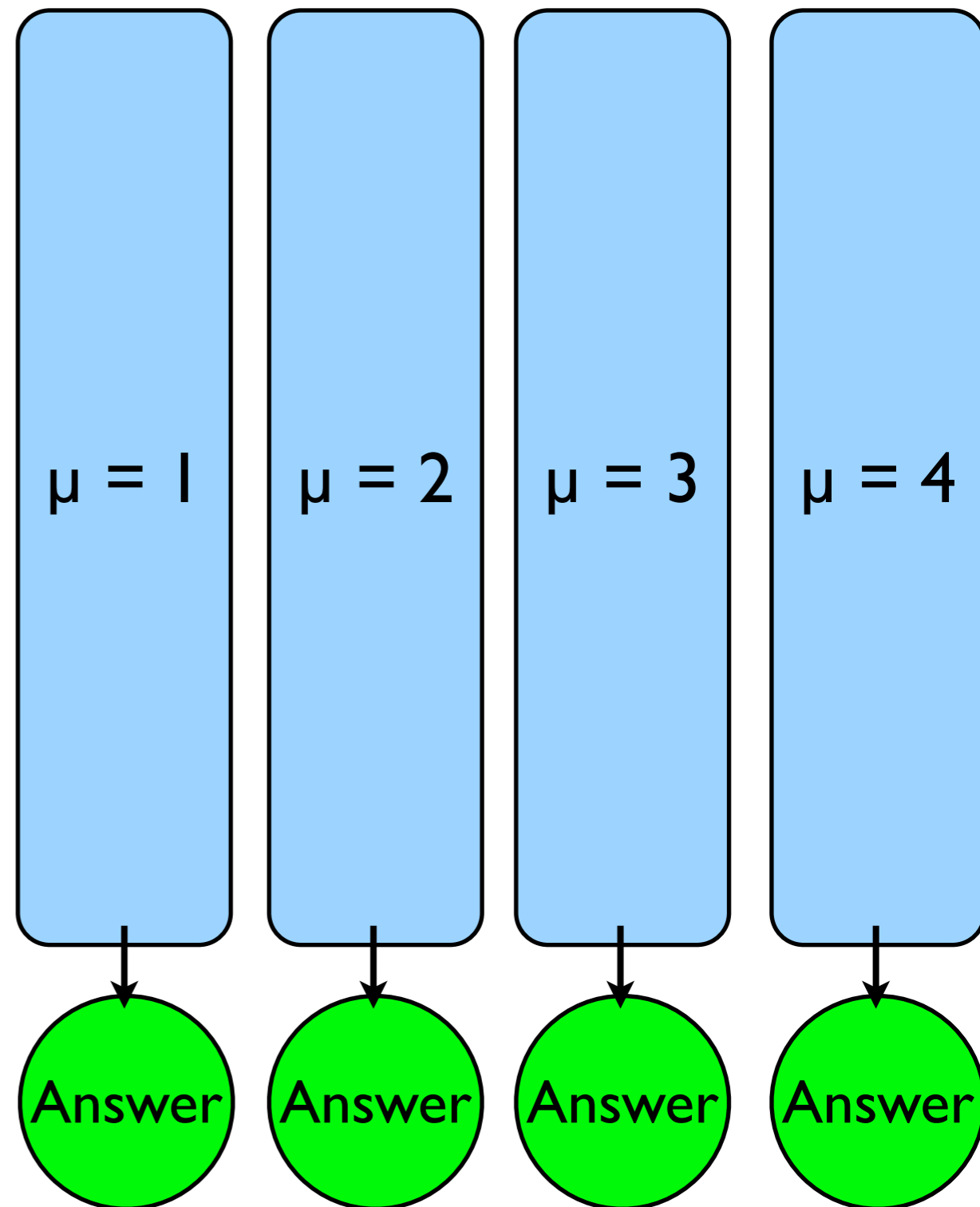
<http://flickr.com/photos/splorp/>

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’**

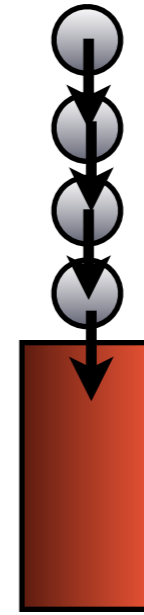


Throughput = Tasks/Time

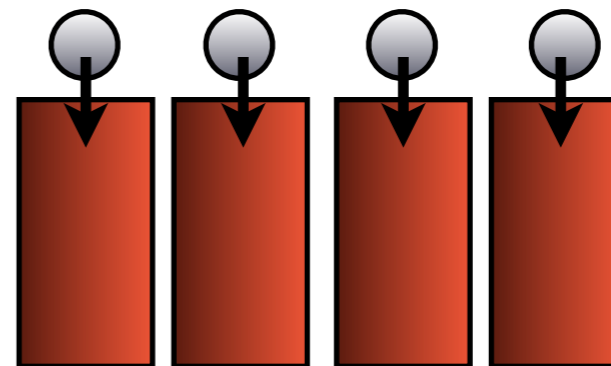
How long it takes to process the
N tasks you want done

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

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



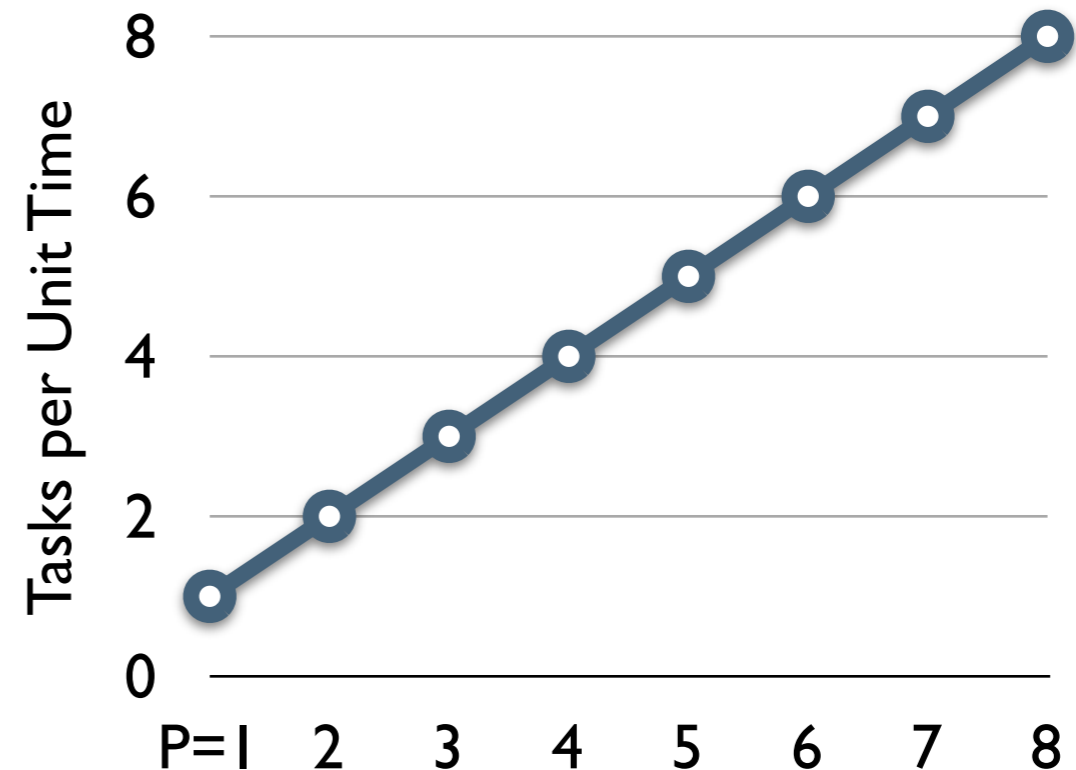
VS



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'



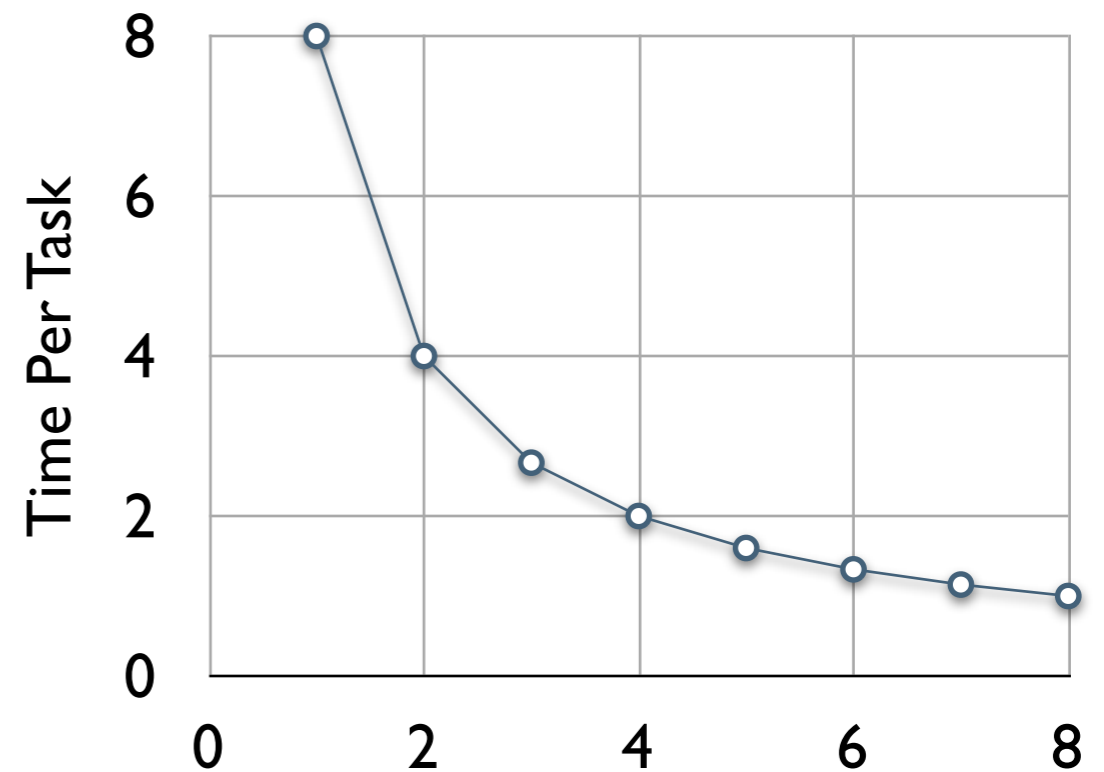
Scaling with P

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 $\sim 1/P$

P processors - P times faster



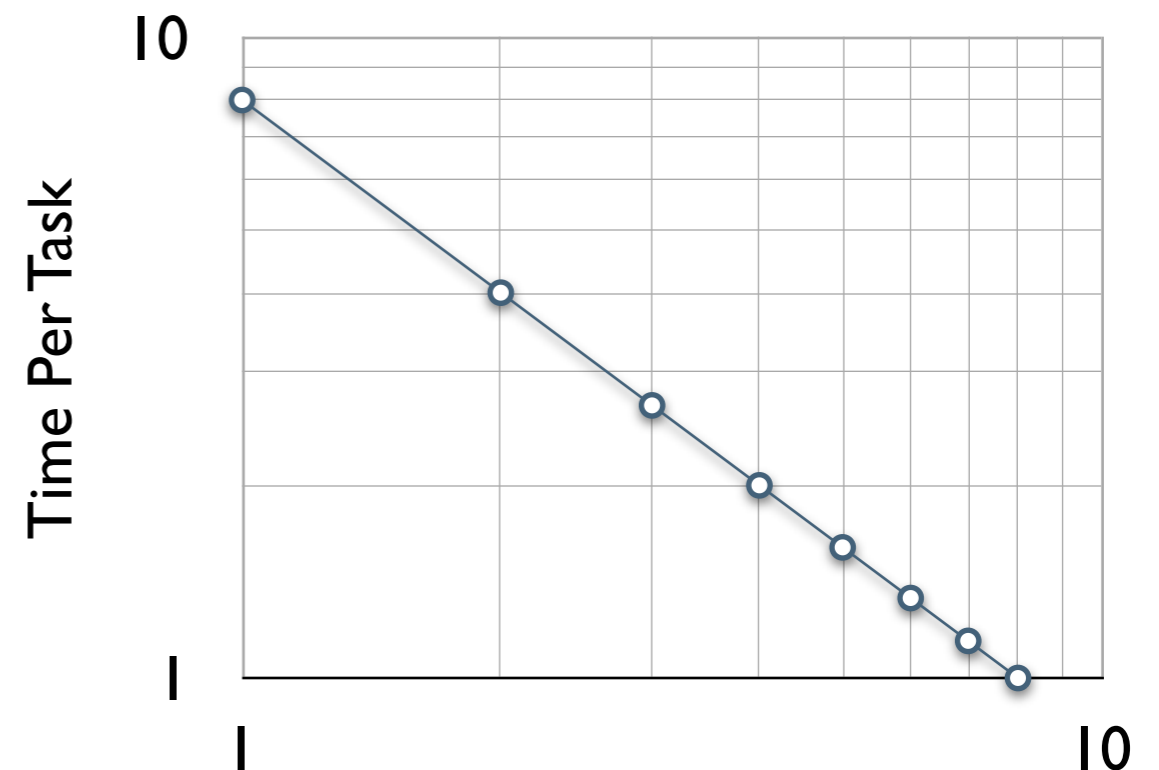
Scaling with P

Another way to look at it: time it takes to get some fixed amount of work done

More usual (and more important!)

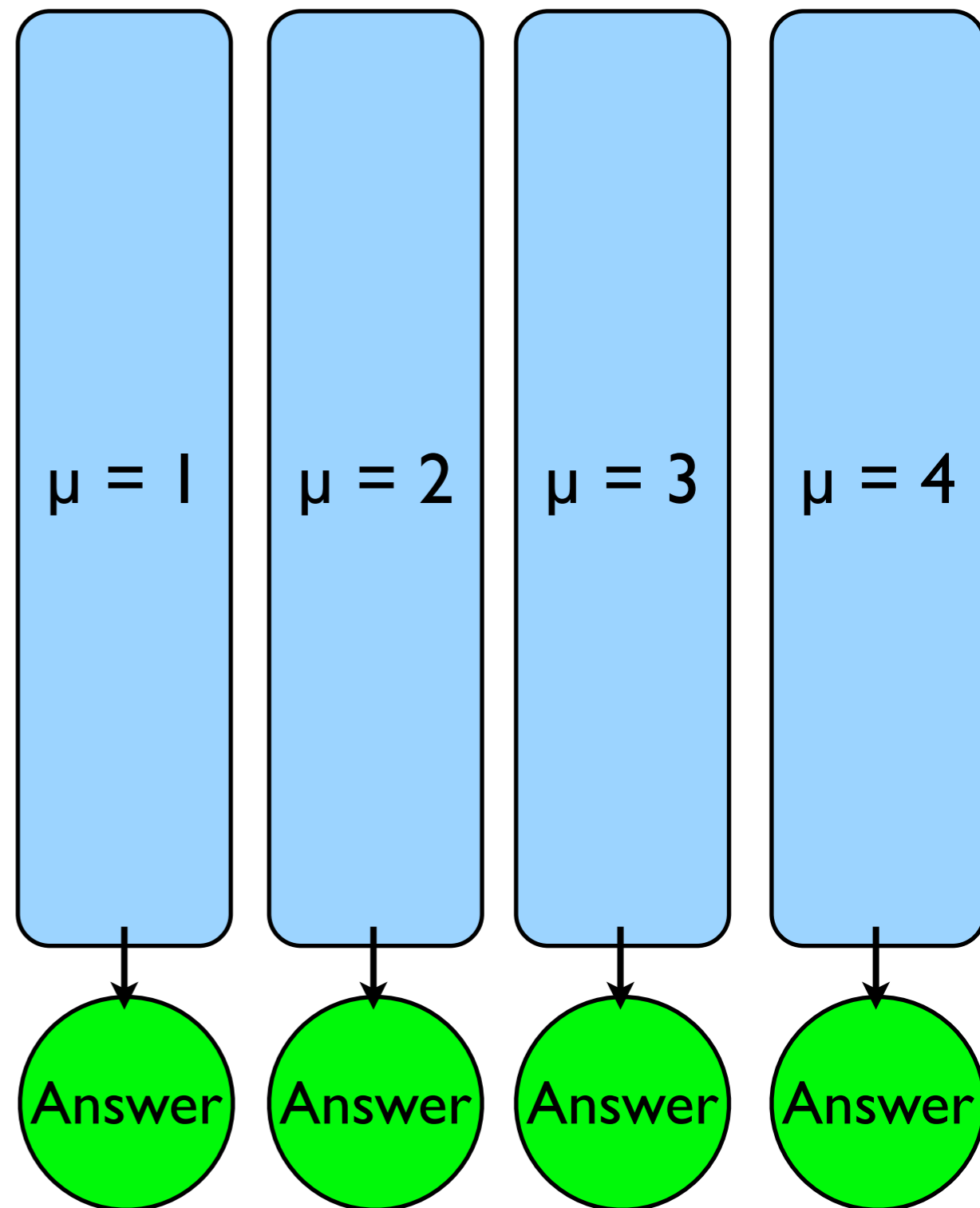
Perfect scaling: time to completion $\sim 1/P$

P processors - P times faster



Parameter Study: 'Embarrassingly Parallel'

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

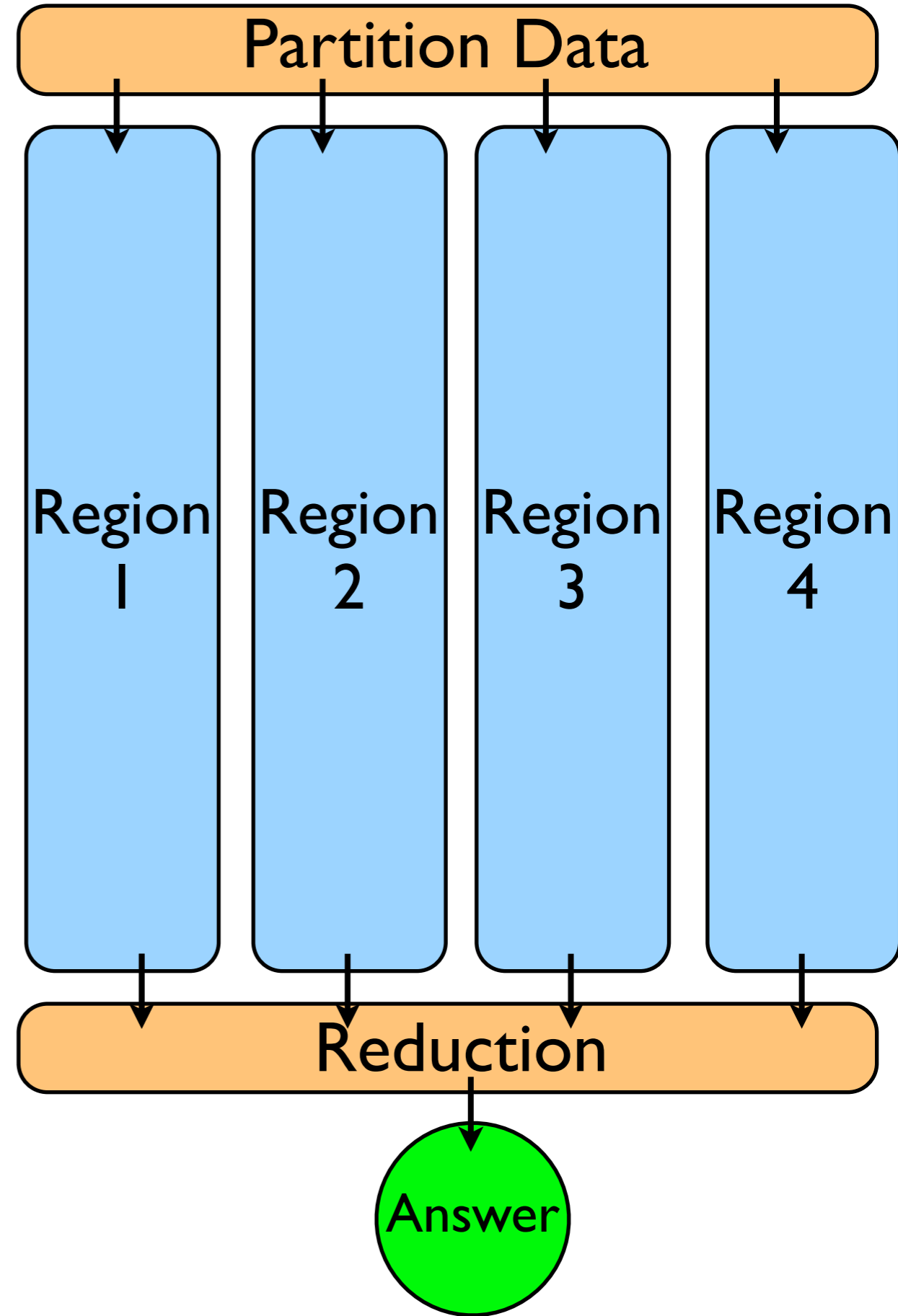


Problems Differ in amount of Concurrency

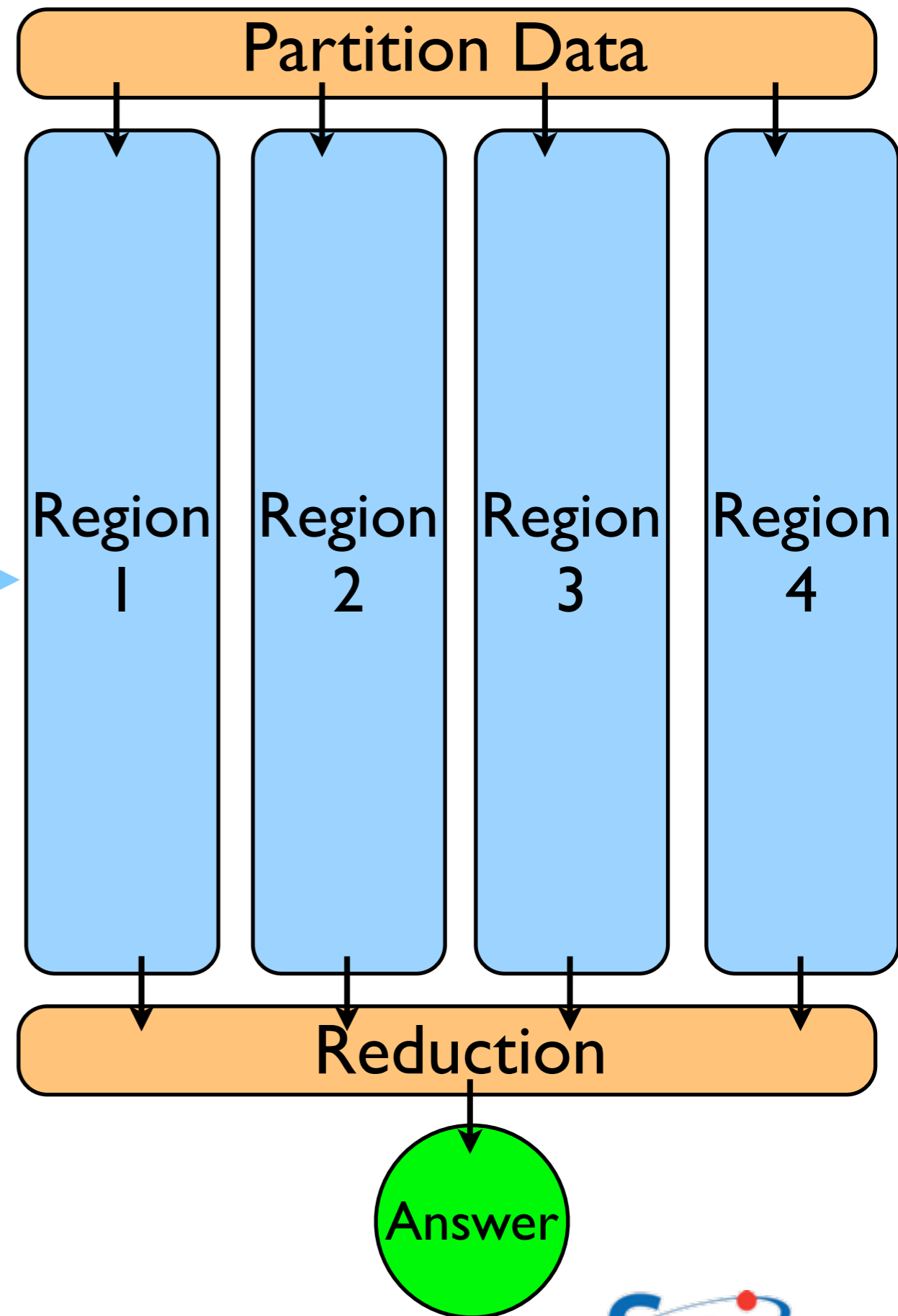
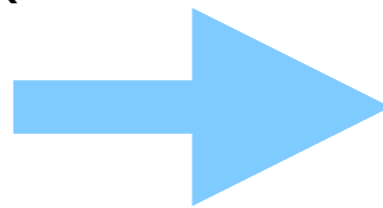
Integrate (or some other simple processing) tabulated experimental data

Integration of different regions can be summed by each processor

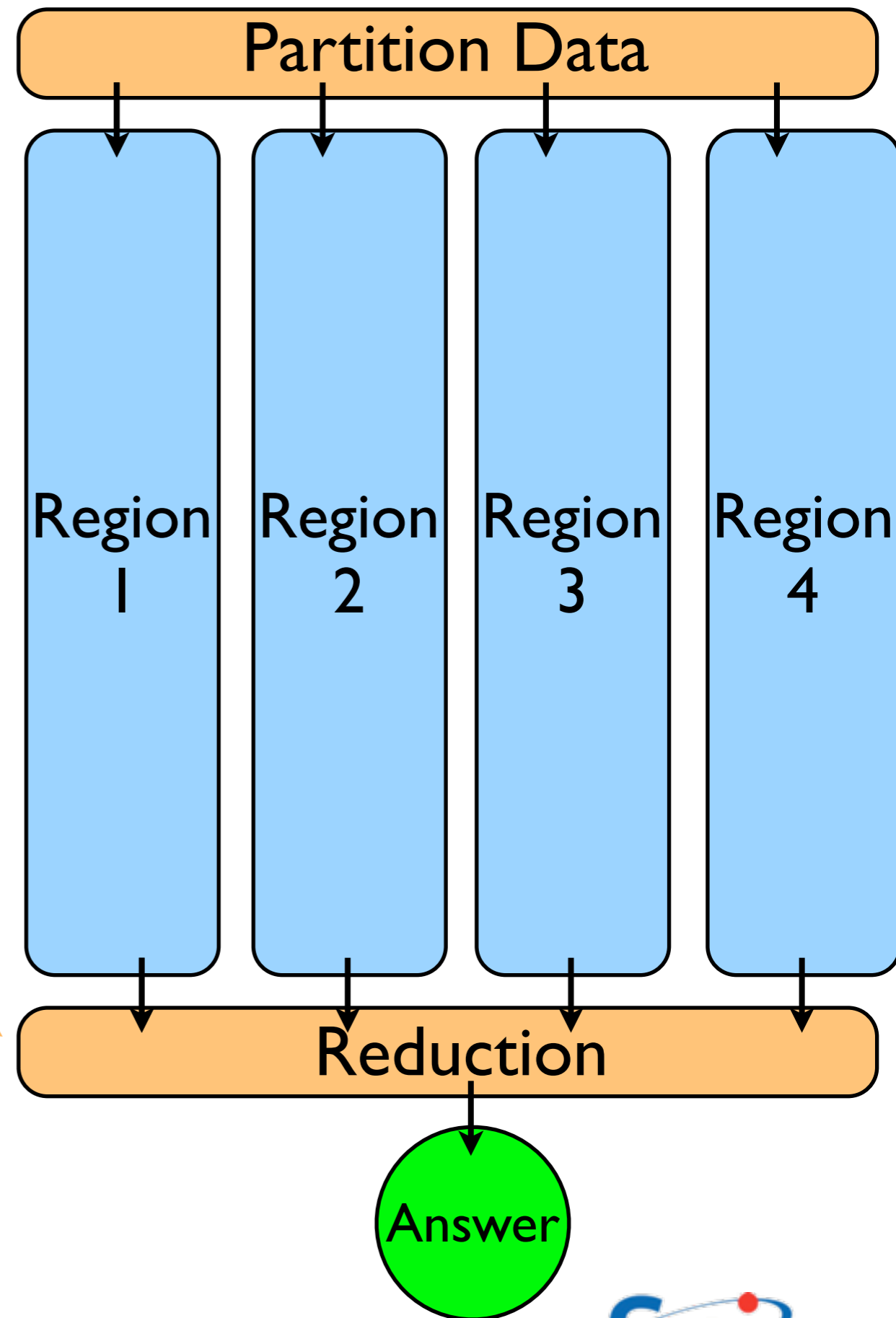
But first need to get data to processor, then bring together all the sums



Parallel Portion:
Perfectly Parallel (as
long as there is
enough work)
 $T \sim 1/P$



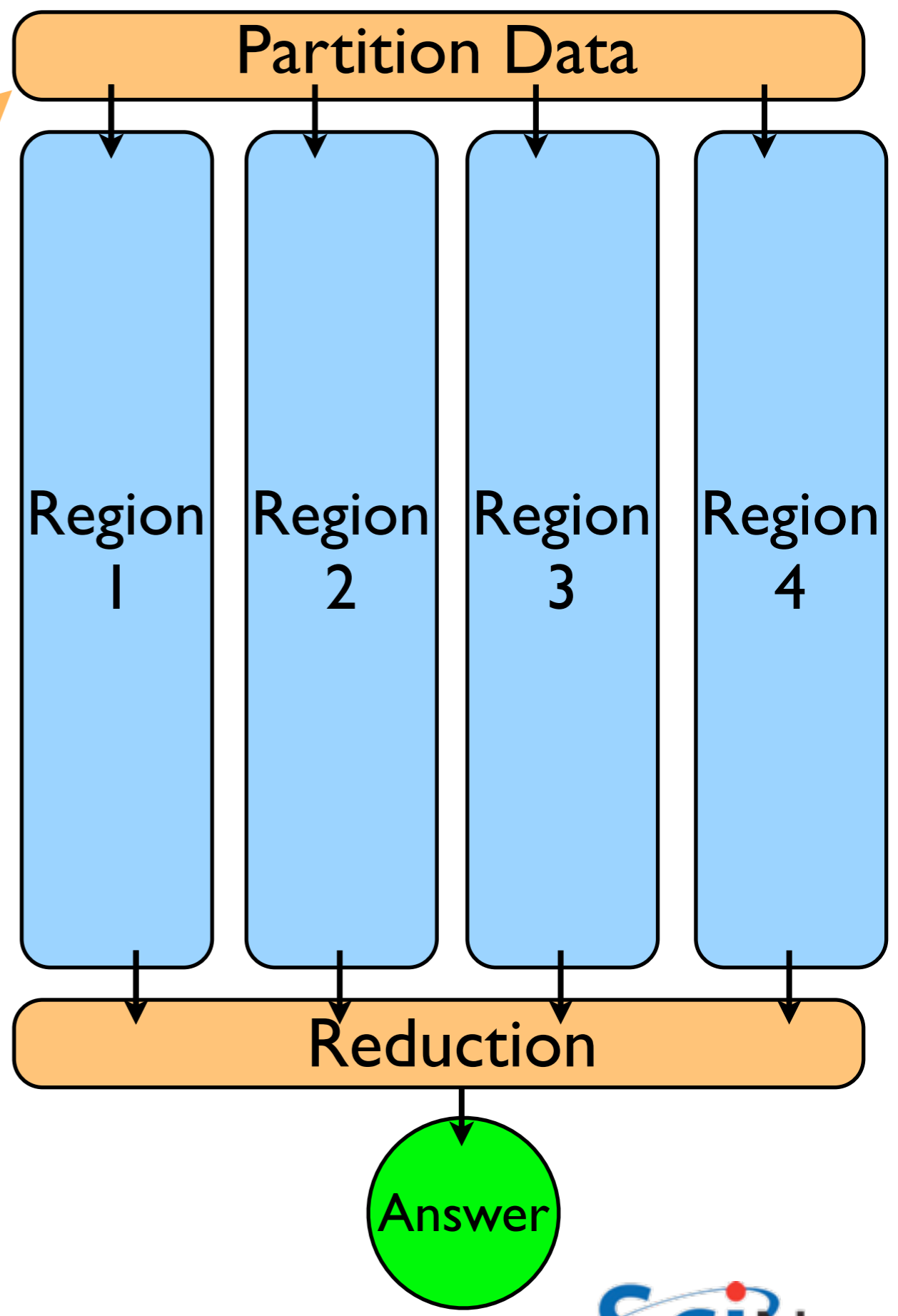
Serial Portion:
Sum has to be
done; if done on one
processor, just same
as serial:
 $T \sim \text{const}$



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*



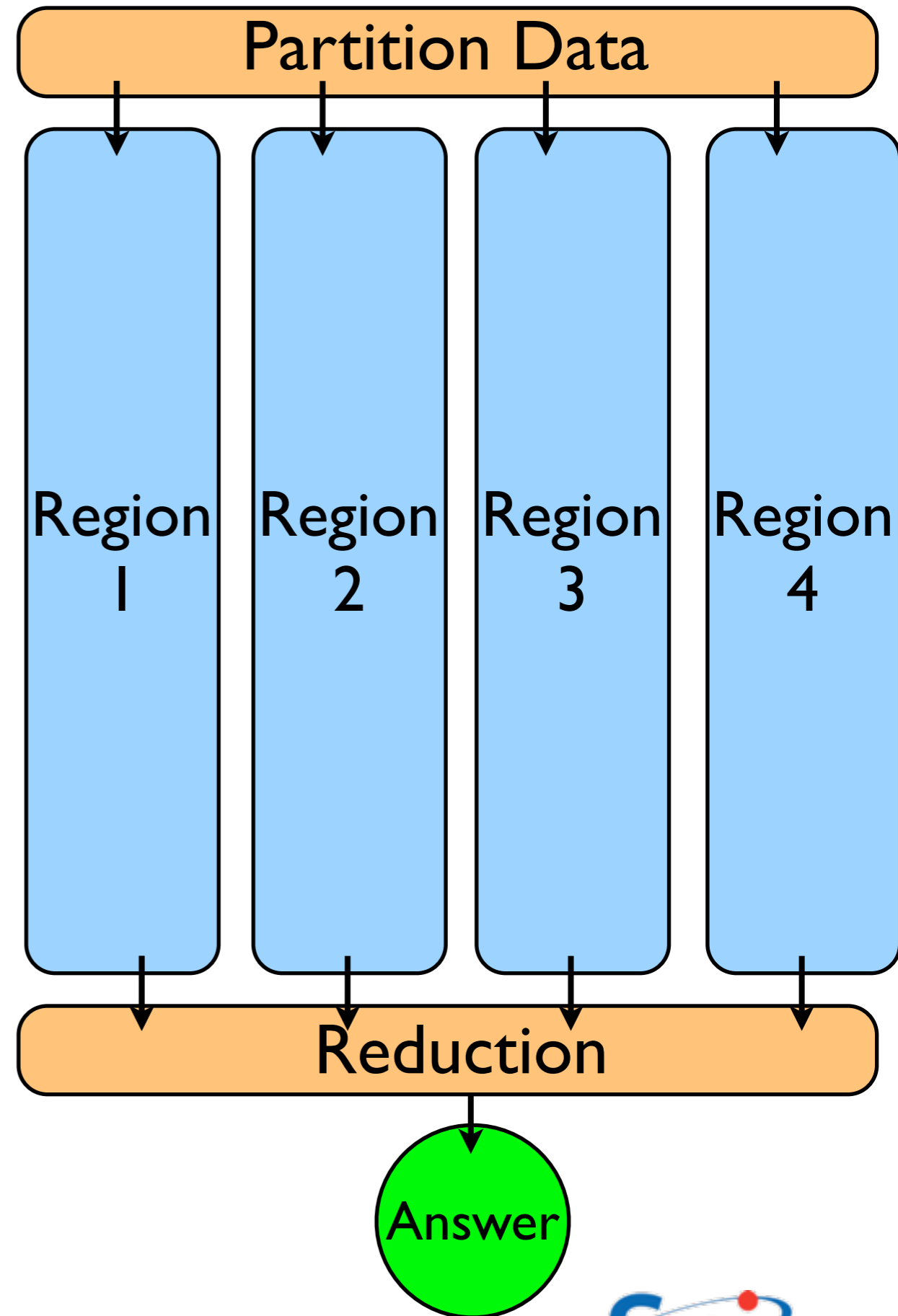
Total Time: Serial + Parallel

Ignoring data-moving costs (for now):

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

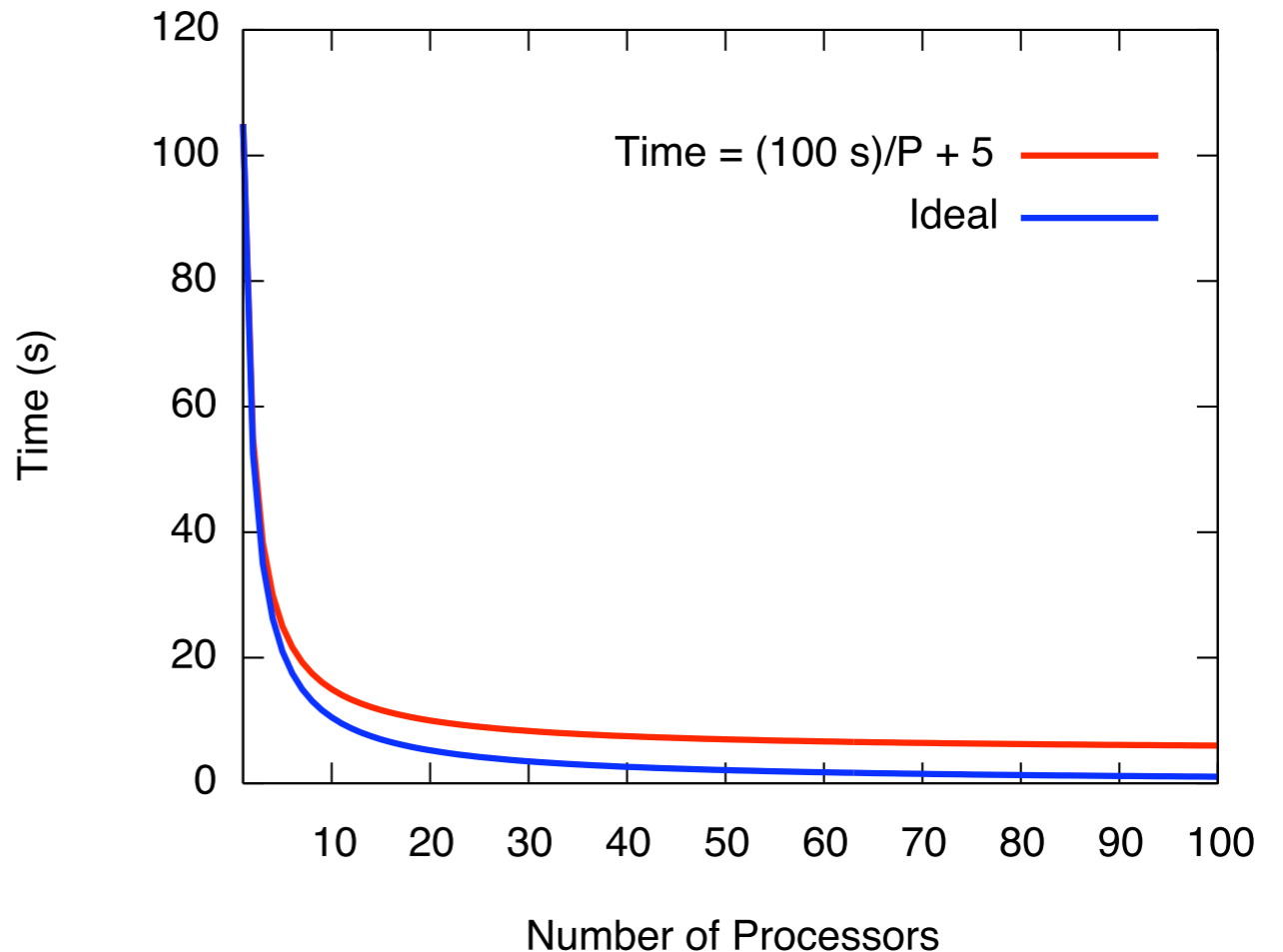
Typically linear in P (sum)

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?



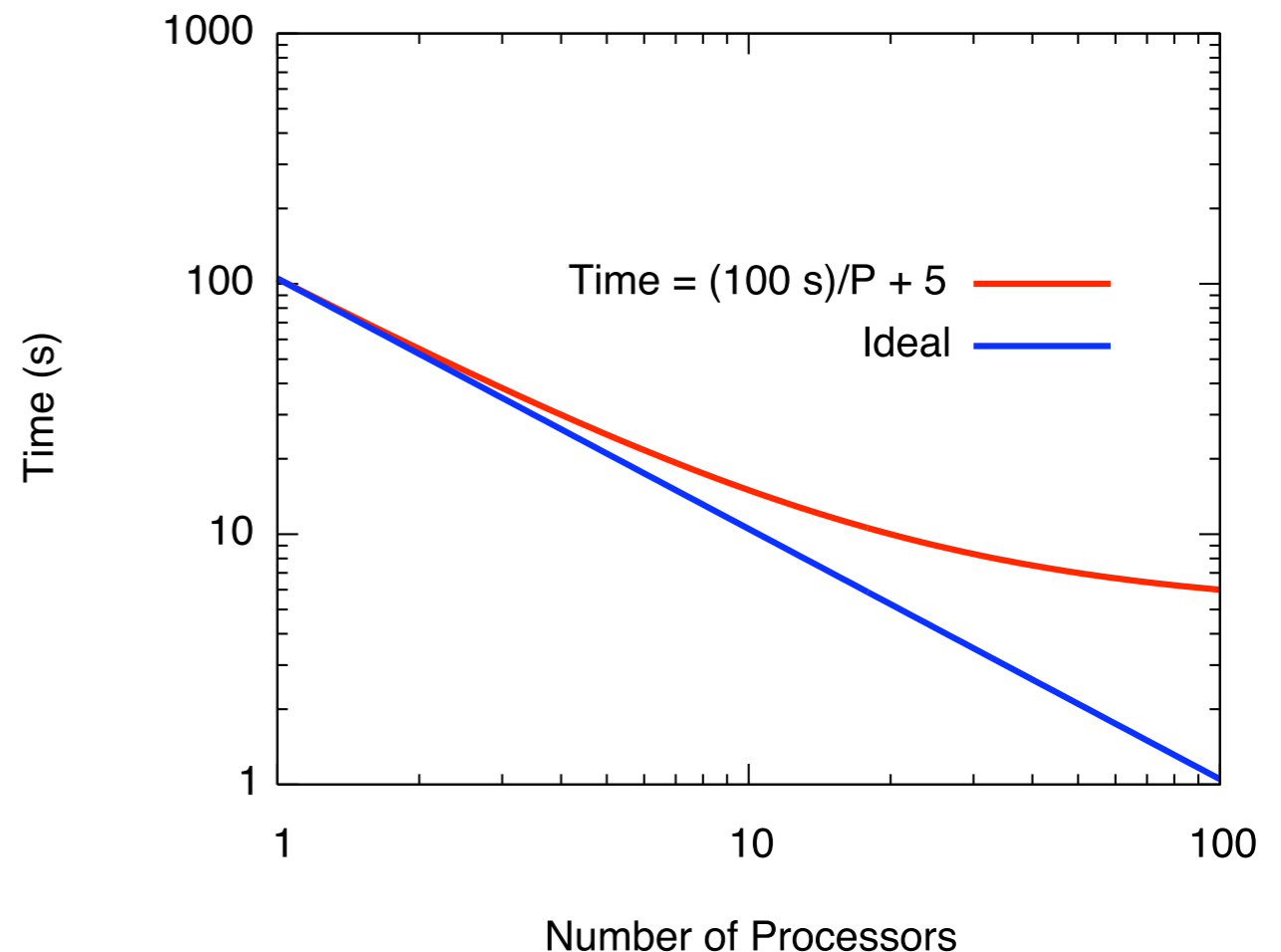
More processors per run don't always help

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: $t(50) = 7s$

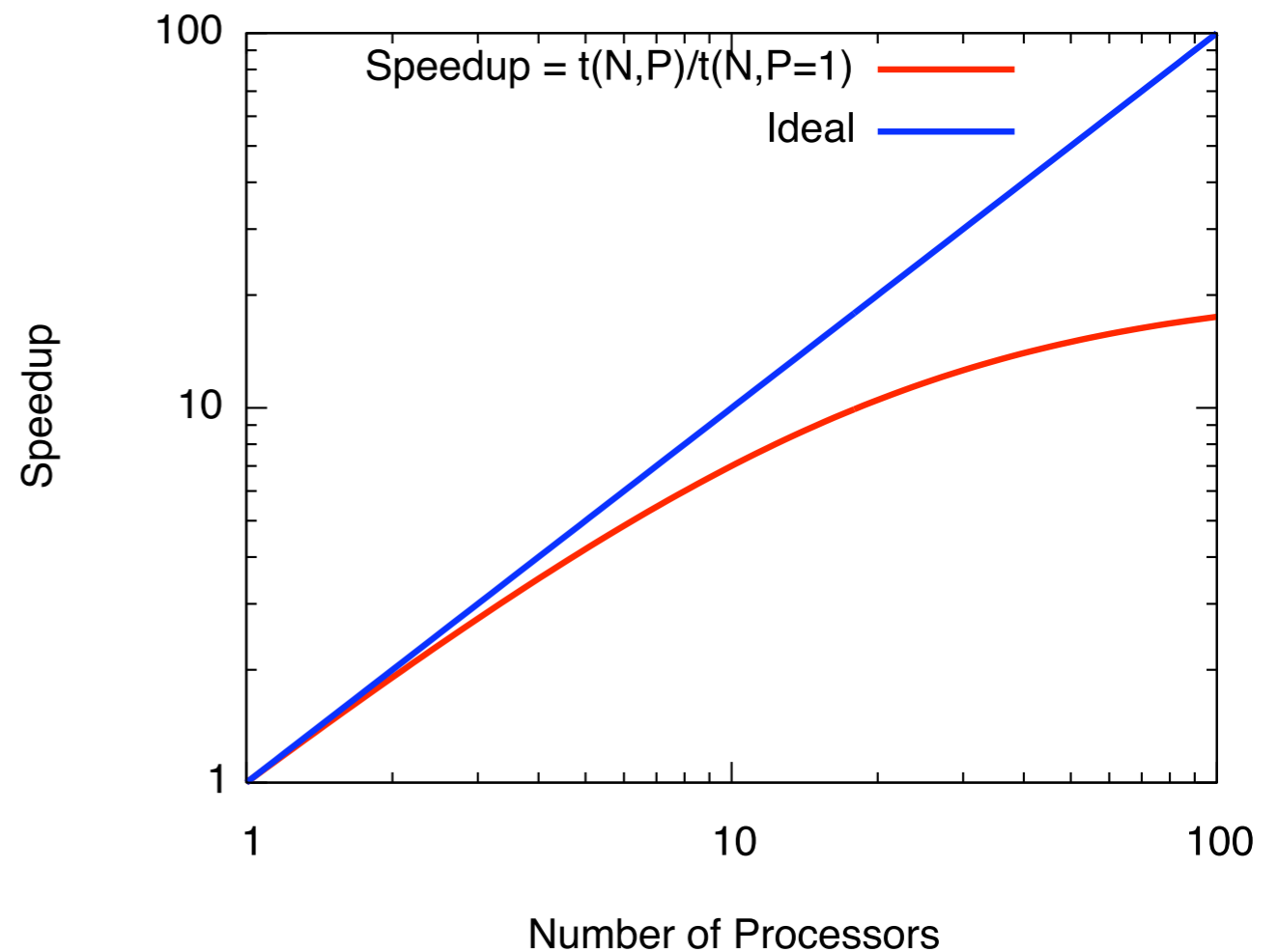
$t(25) = 9s$

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

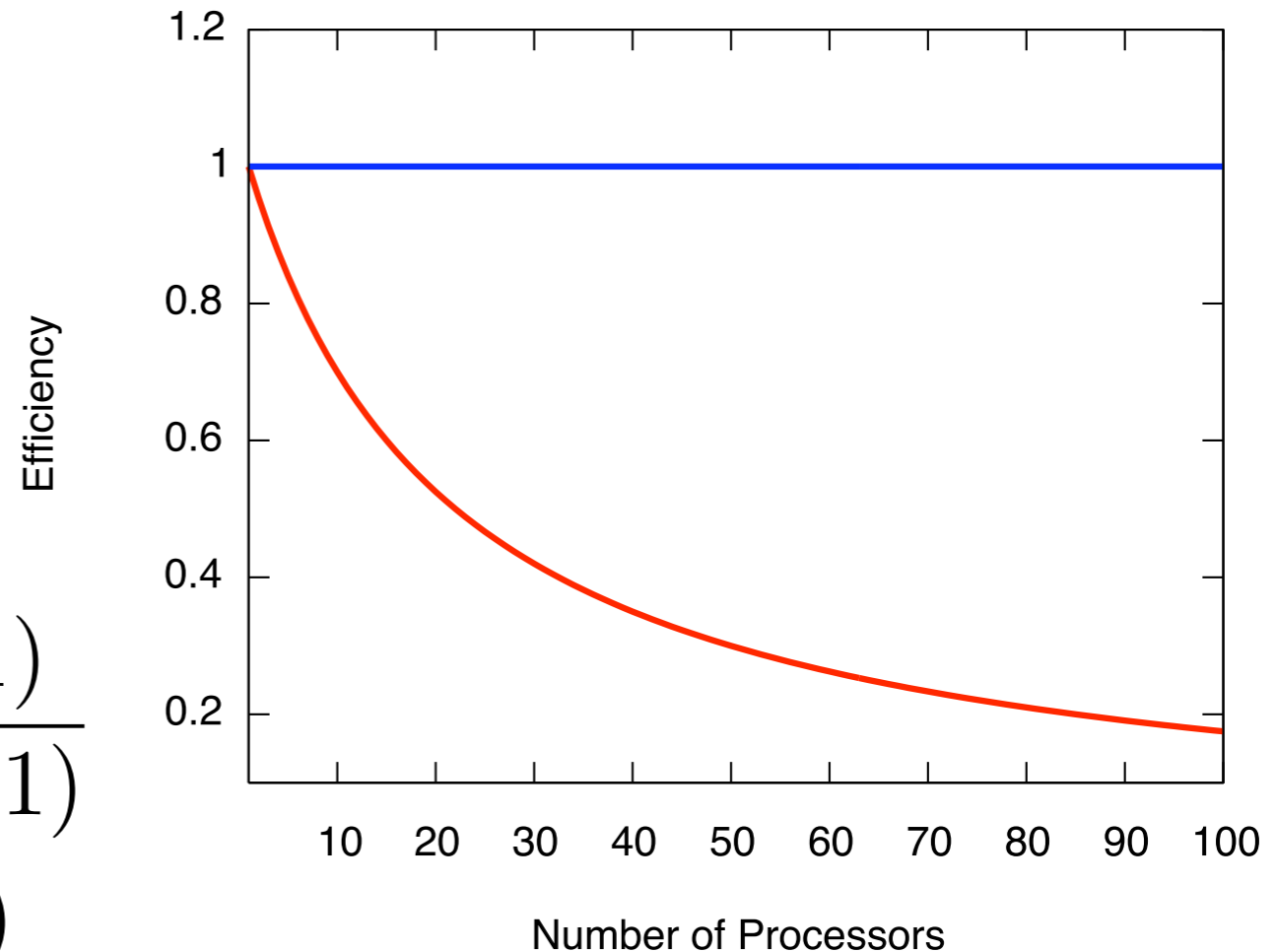
$$\text{speedup}(P) = \frac{t(N, P = 1)}{t(N, P)}$$



Efficiency: Speedup should be $\sim P$

Related concept: Parallel
Efficiency (compared to serial
code)

$$\begin{aligned} \text{Efficiency}(P) &= \frac{t(N, P = 1)}{Pt(N, P)} \\ &= \frac{\text{speedup}(P)}{P} \end{aligned}$$



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

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

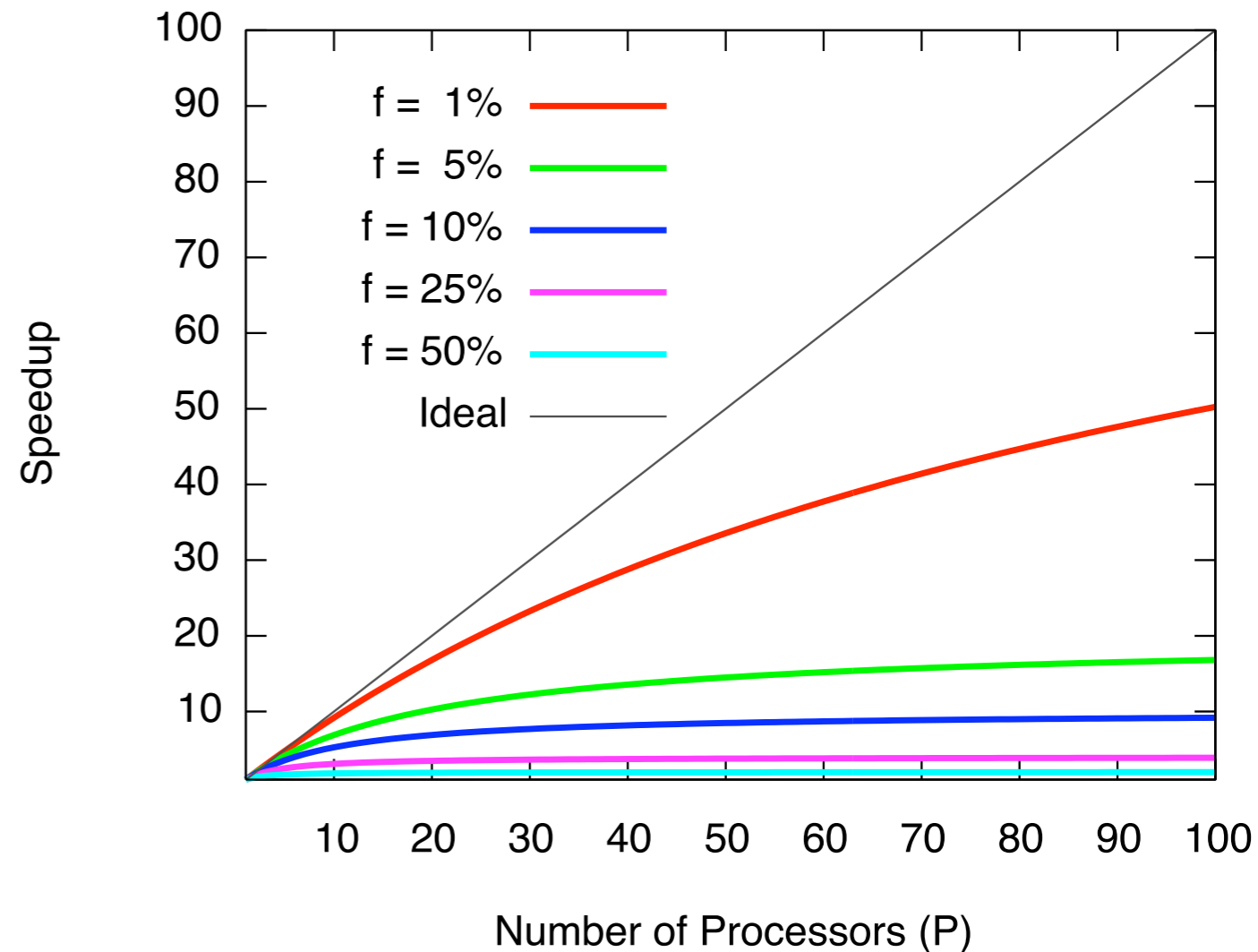
$$\text{Speedup} = \frac{1}{\left(f + \frac{1-f}{P} \right)}$$

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

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

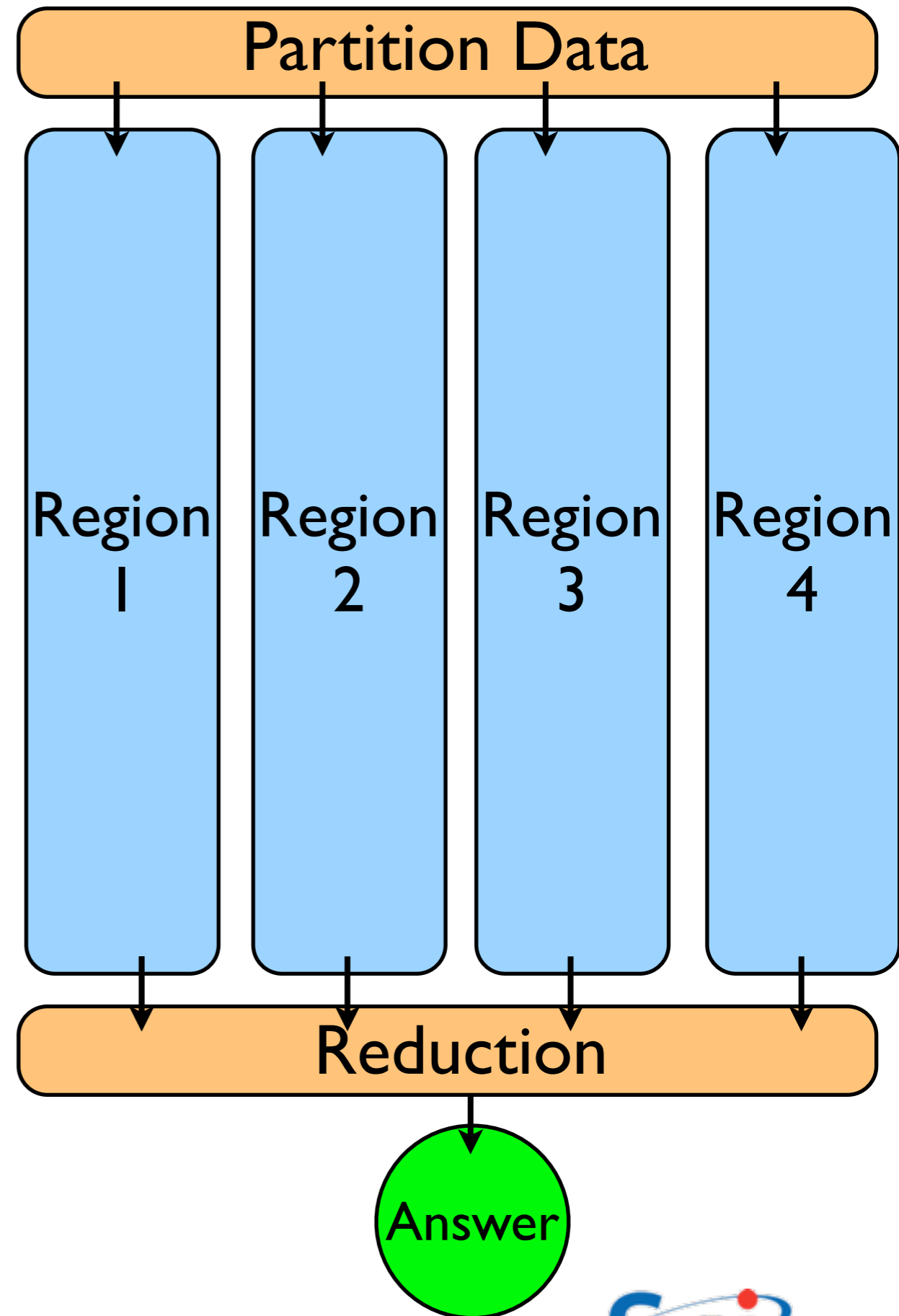
Amdahl's Law

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



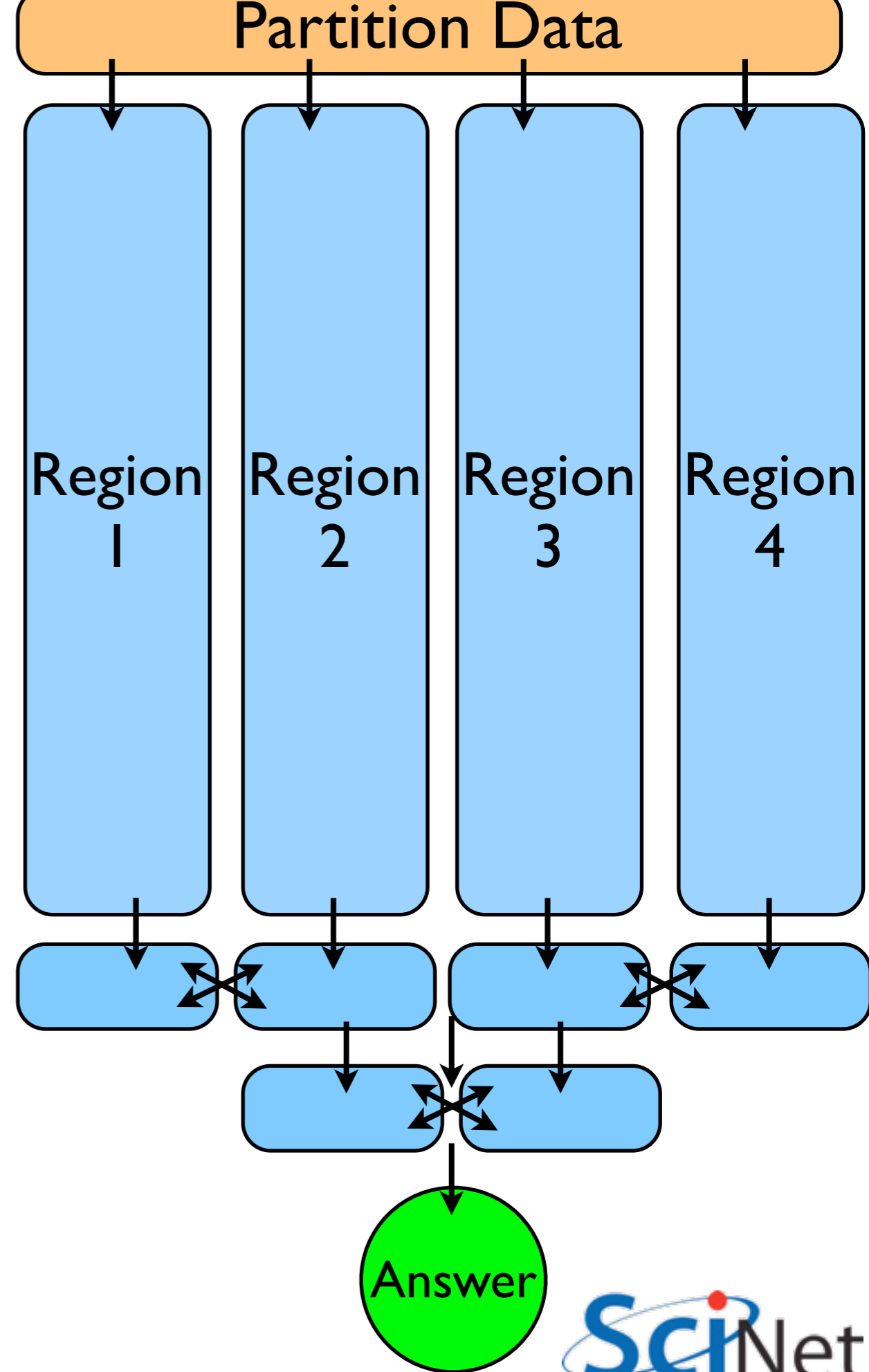
Avoiding Amdahl

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...)



Big Lesson #1

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

Trying to Beat Amdahl, #2 - Upsize

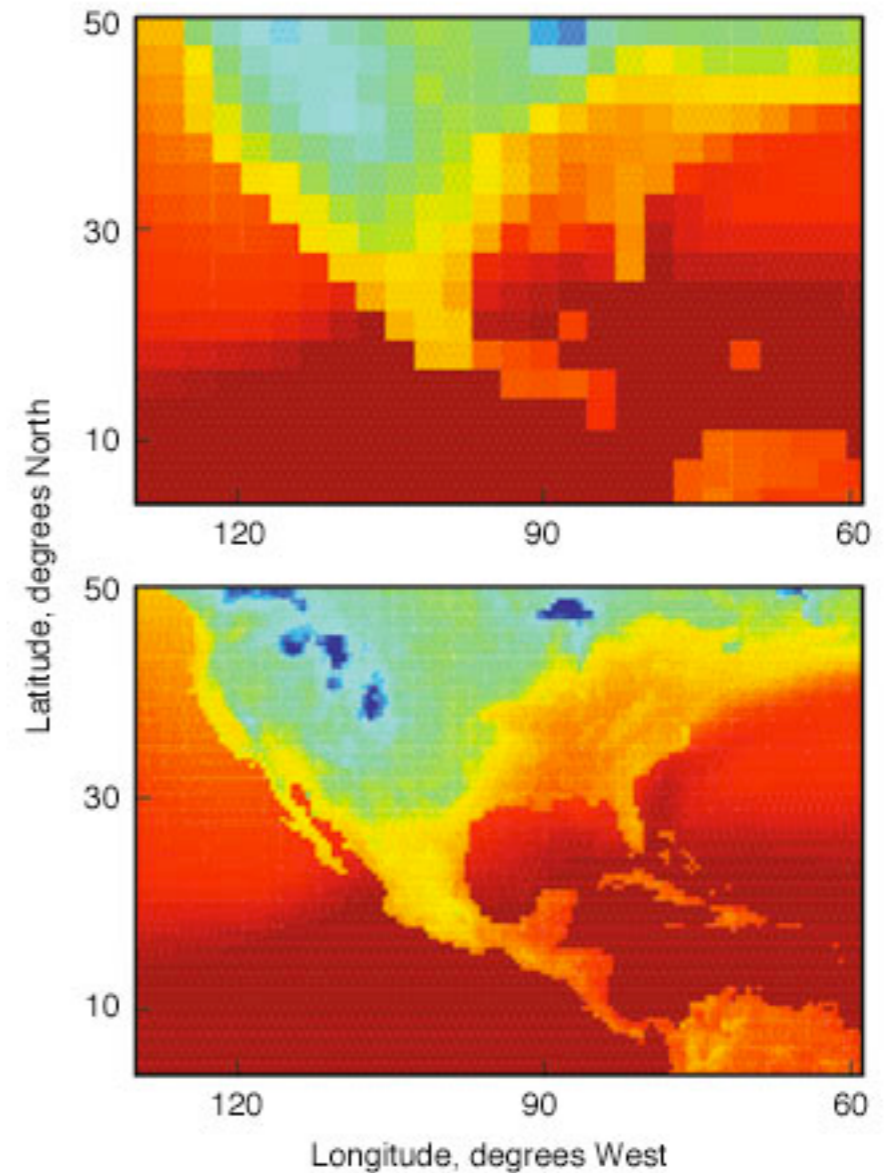
Desktop problem isn't a
supercomputer problem!

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.

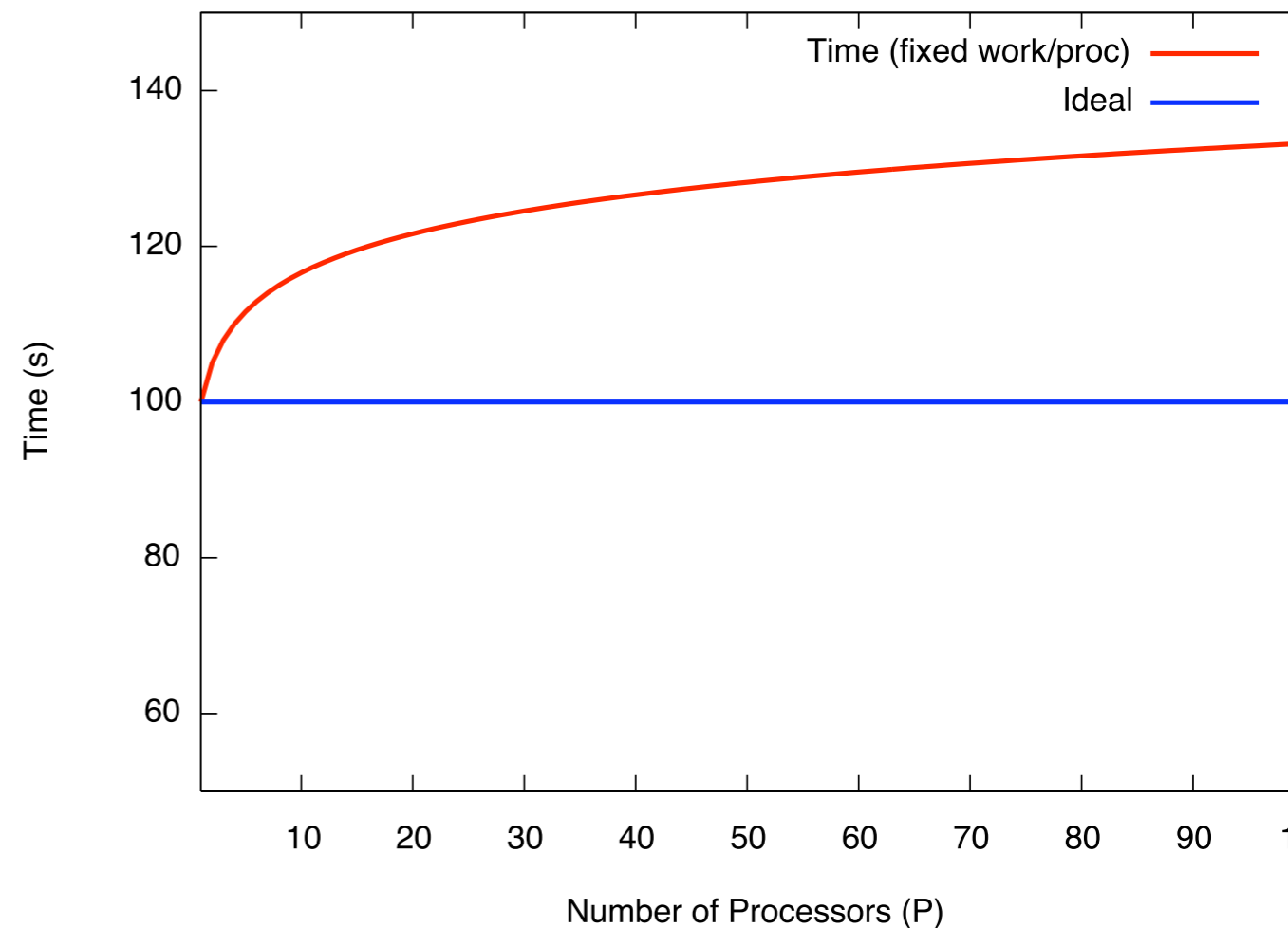


Weak Scaling

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

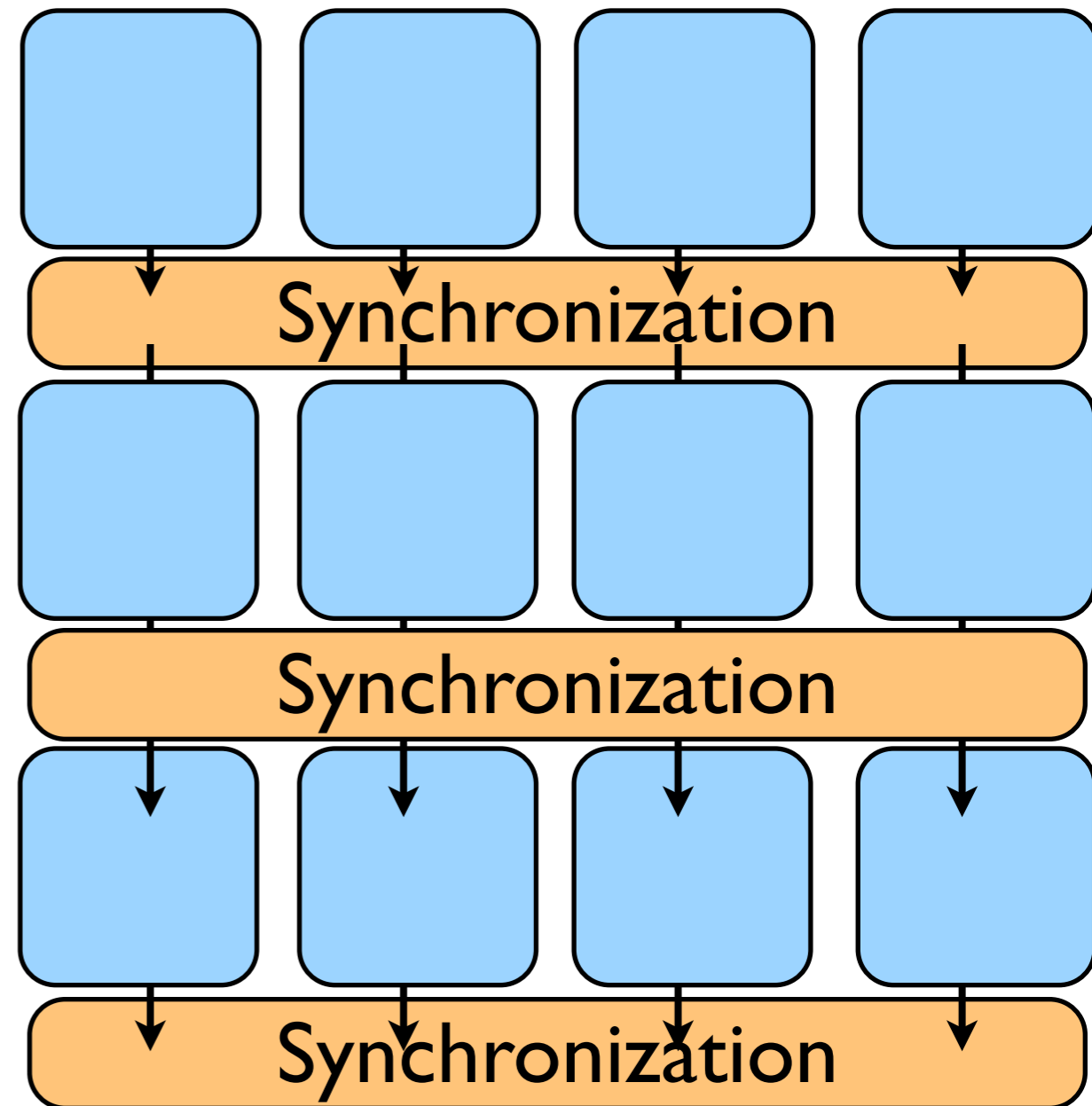
Most problems are not pure
concurrency

Some level of synchronization,
exchange of information needed
between tasks

This needs to be minimized

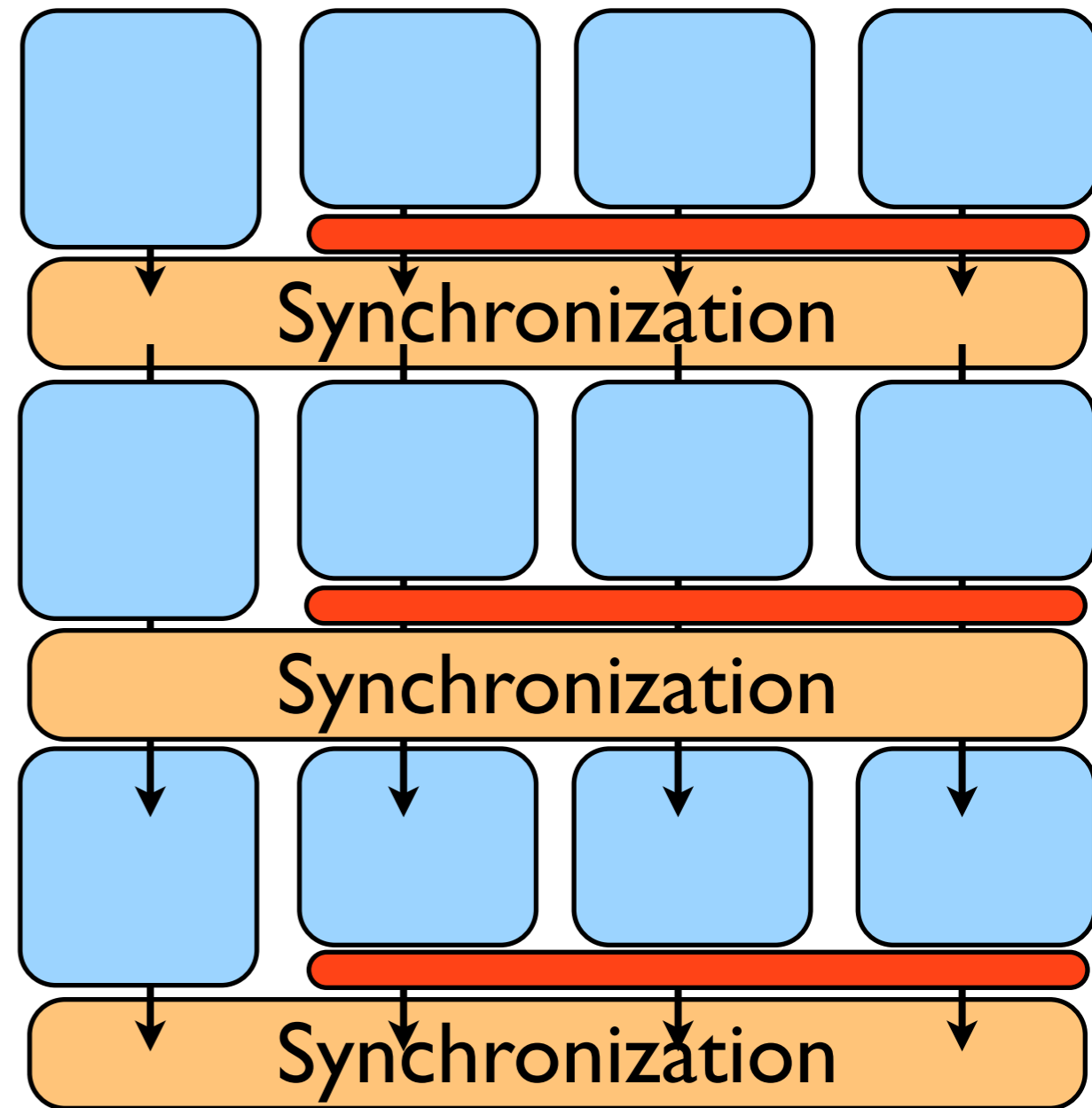
Increases Amdahl's 'f'

Are themselves costly



Concurrency

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



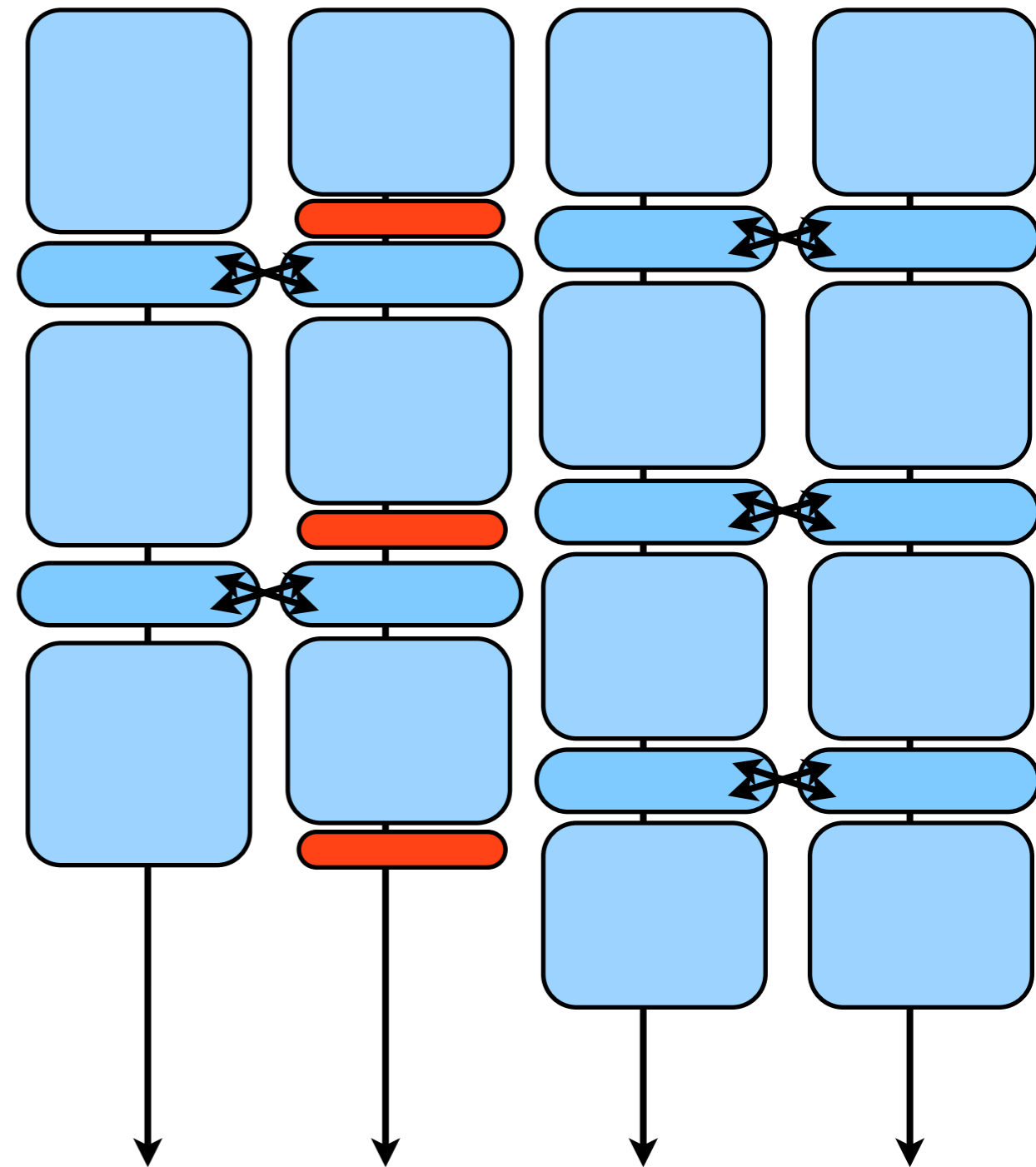
Locality

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 synchronization



Big Lesson #2

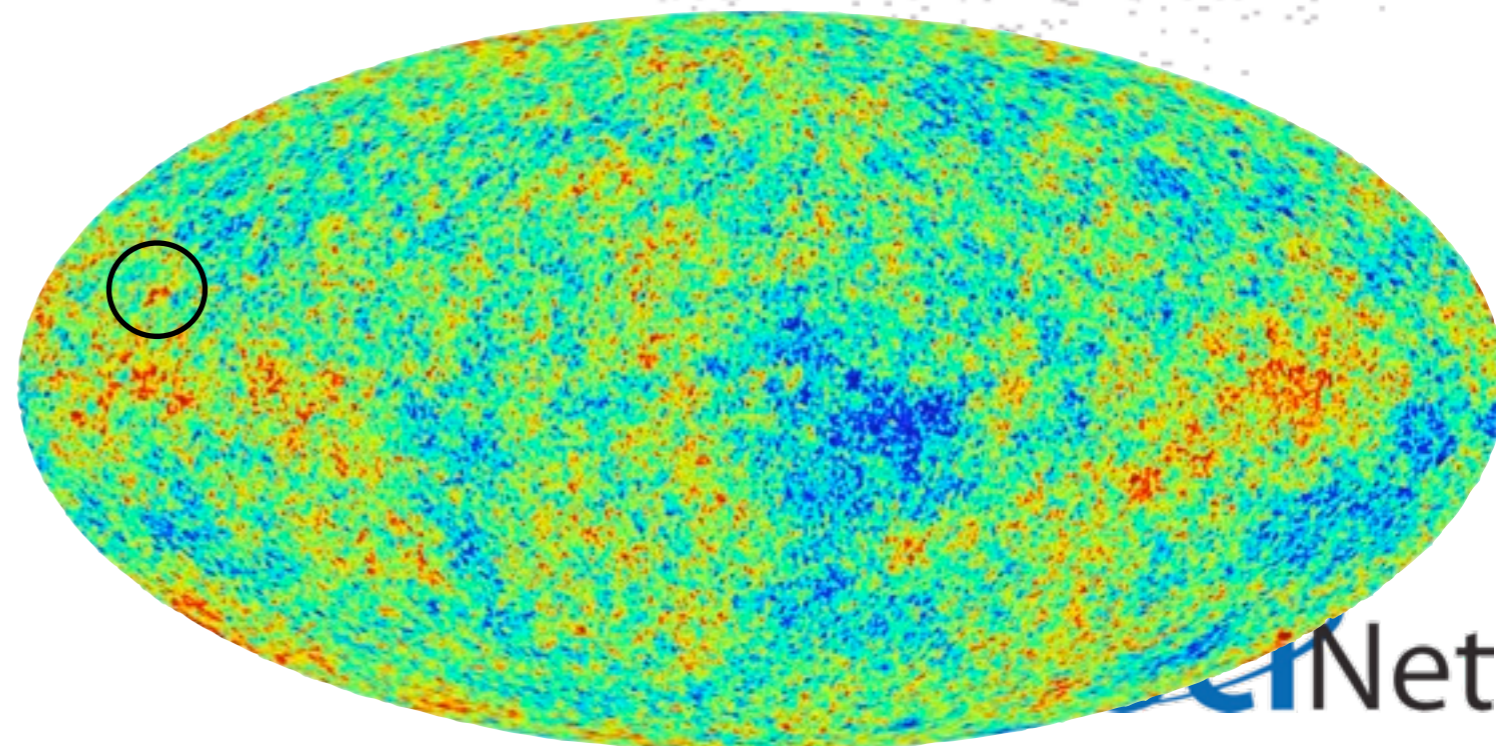
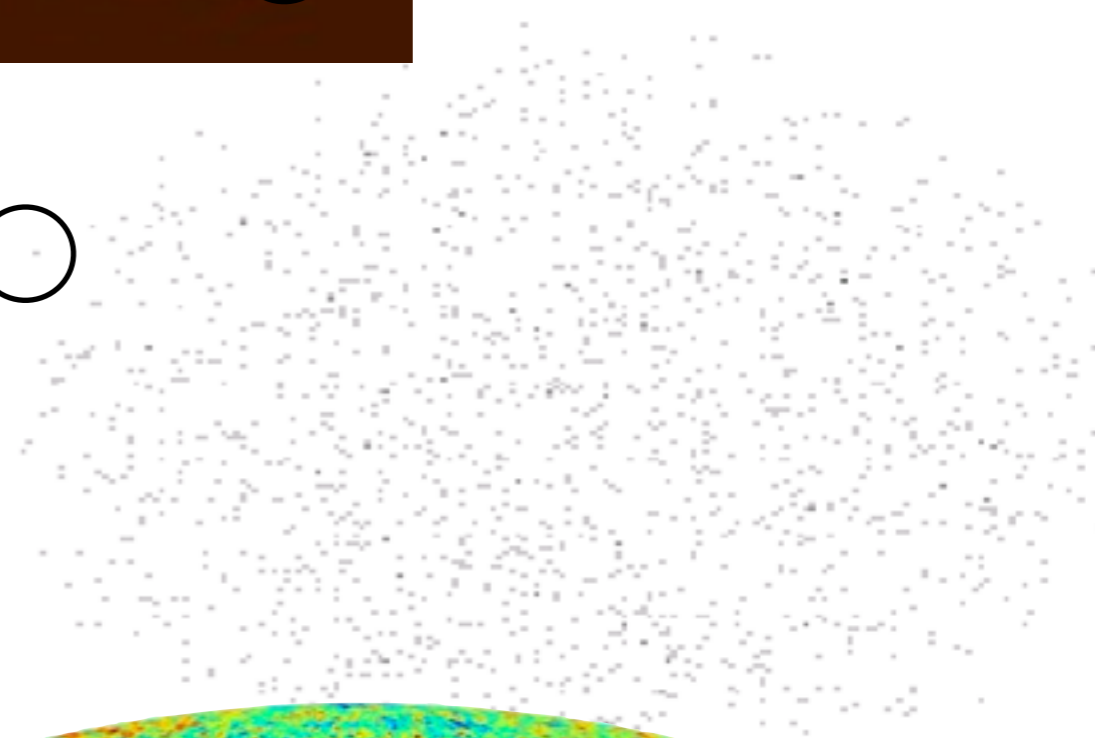
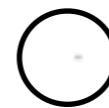
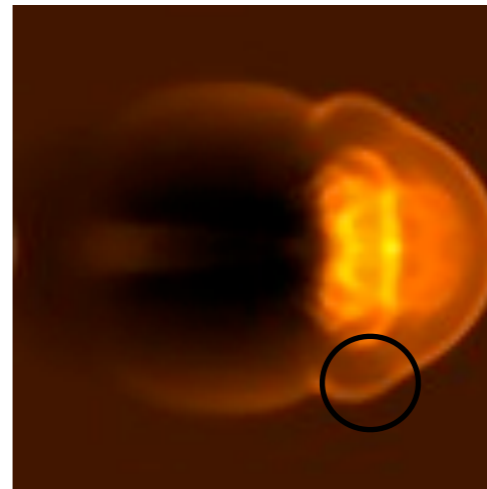
Parallel algorithm design is about finding as much concurrency as possible, and arranging it in a way that maximizes locality.

Finding Concurrency

Identify tasks that can be done
independently, order doesn't
matter

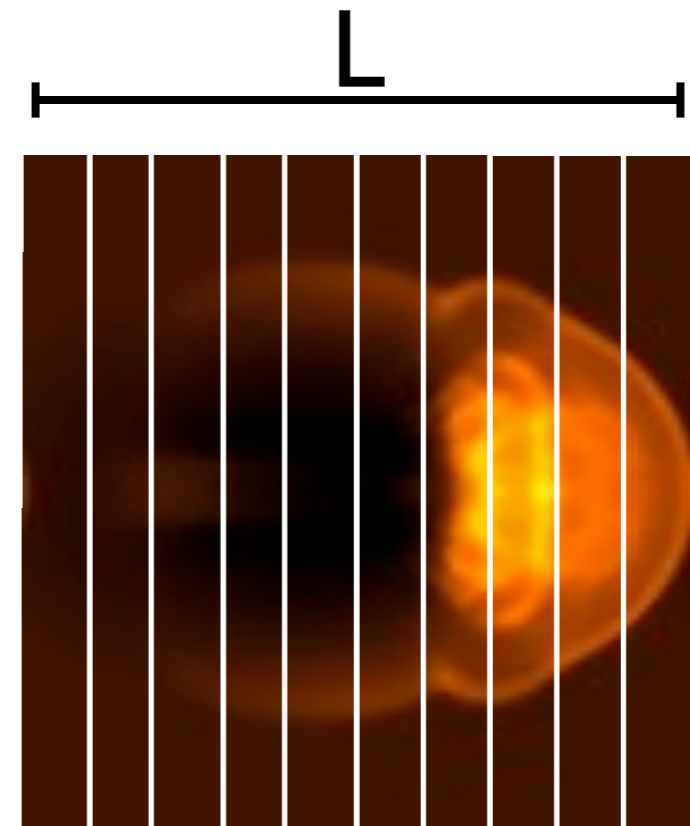
PDEs: parts of domain

N-body: particles (or
interactions)

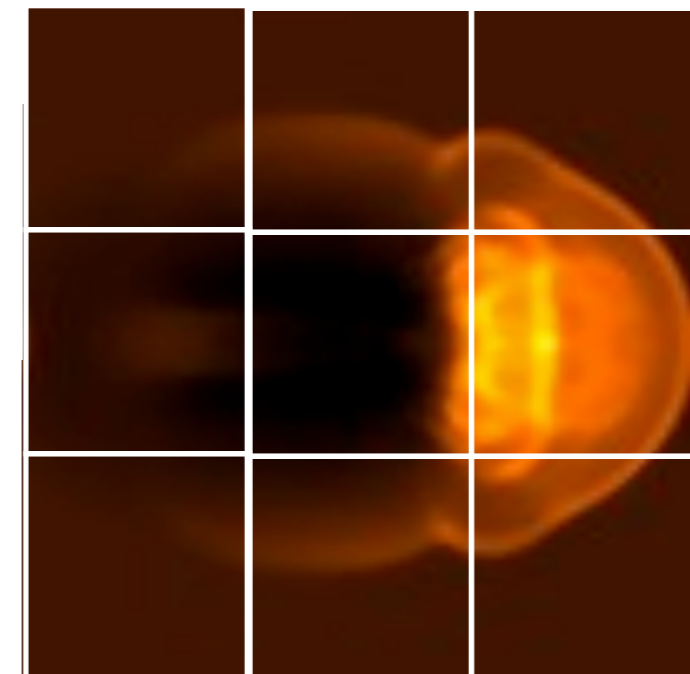


Maintaining Locality

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



perimeter
 $= 9L$



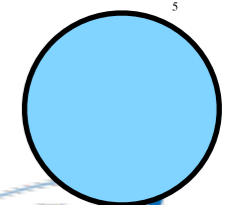
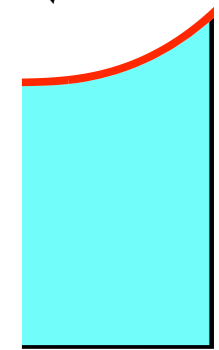
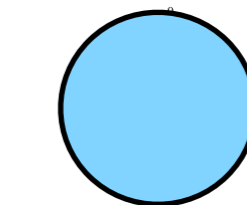
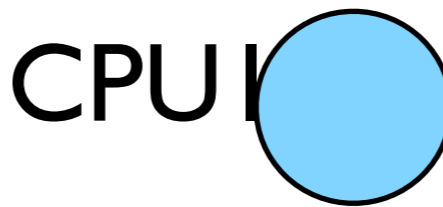
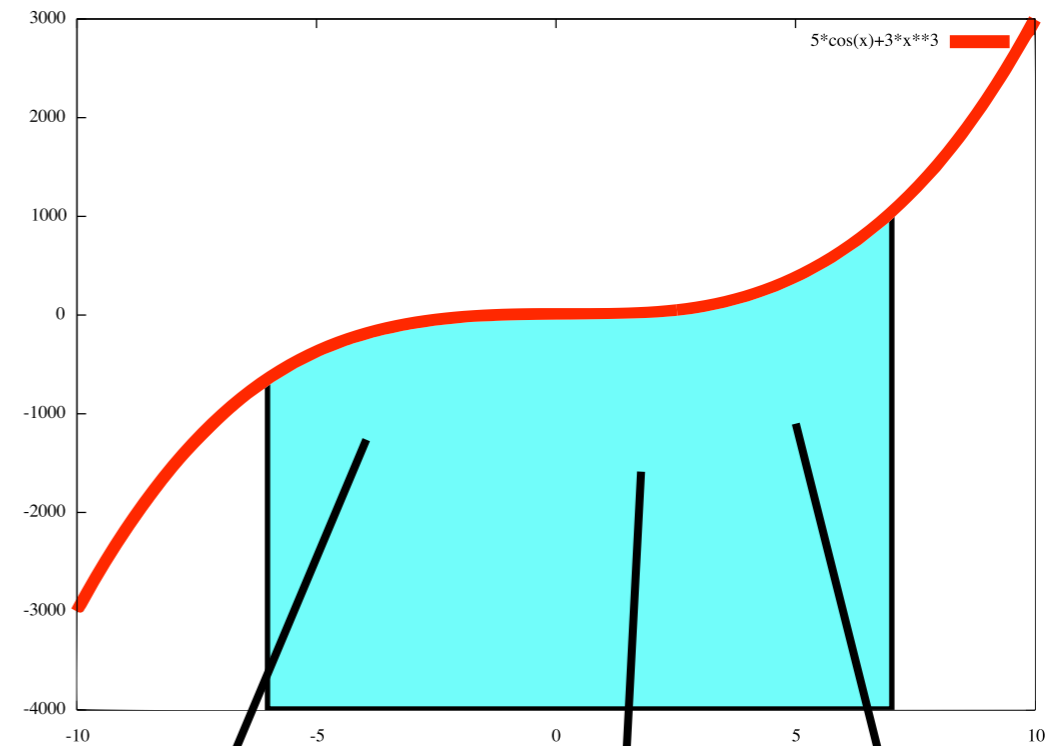
perimeter
 $= 4L$

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



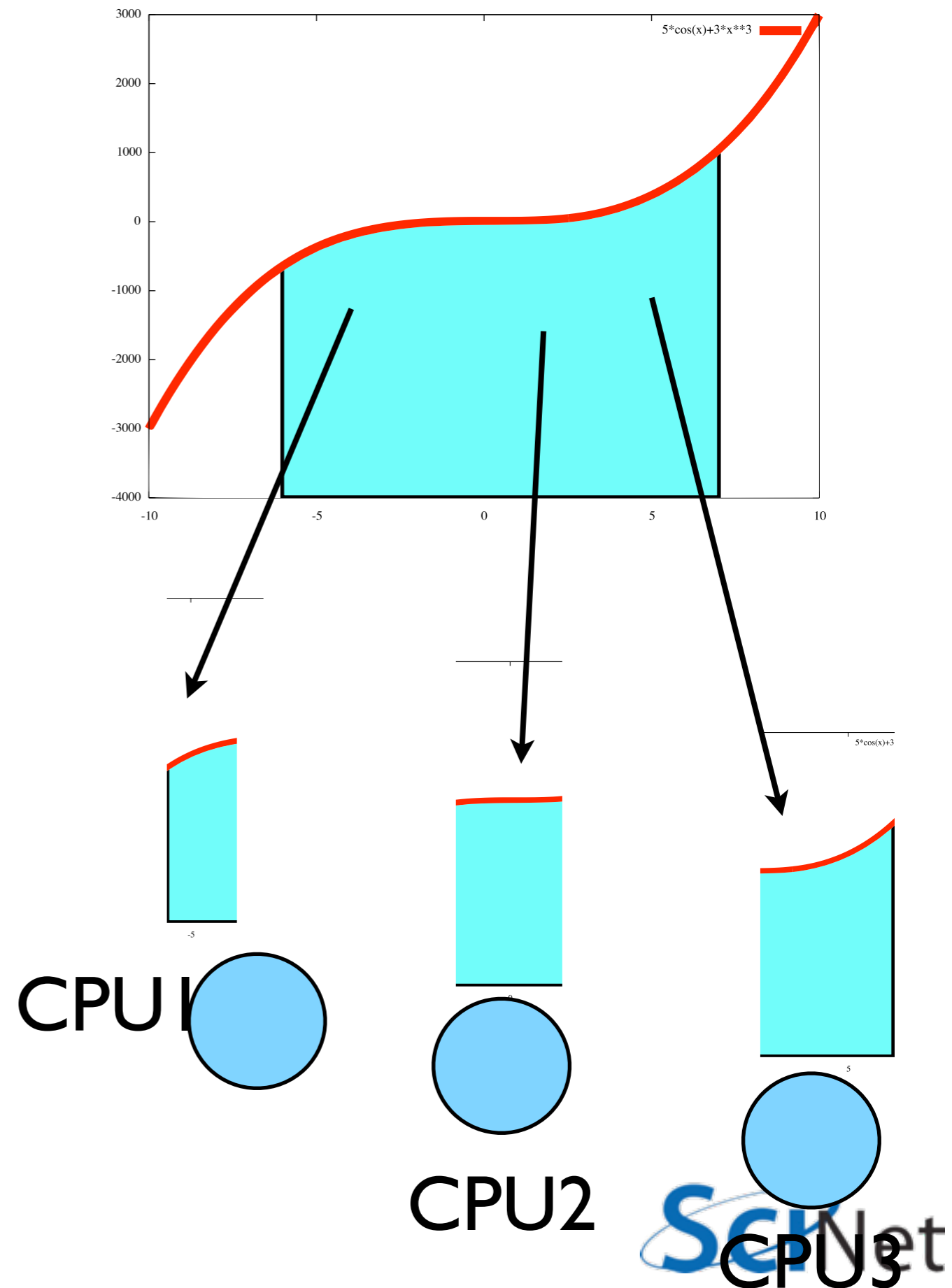
Example: 1d integration

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?

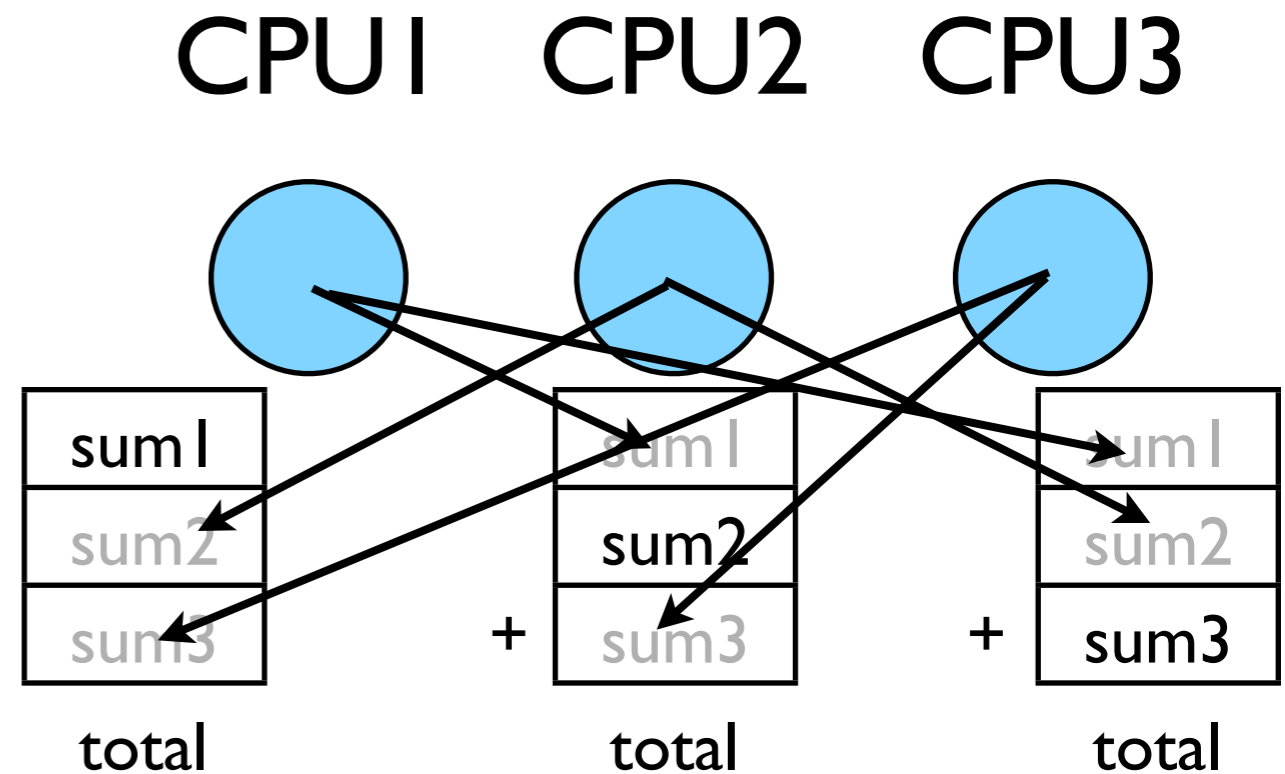


Example: 1d integration

Each processor sends partial sums to others, then all can do total

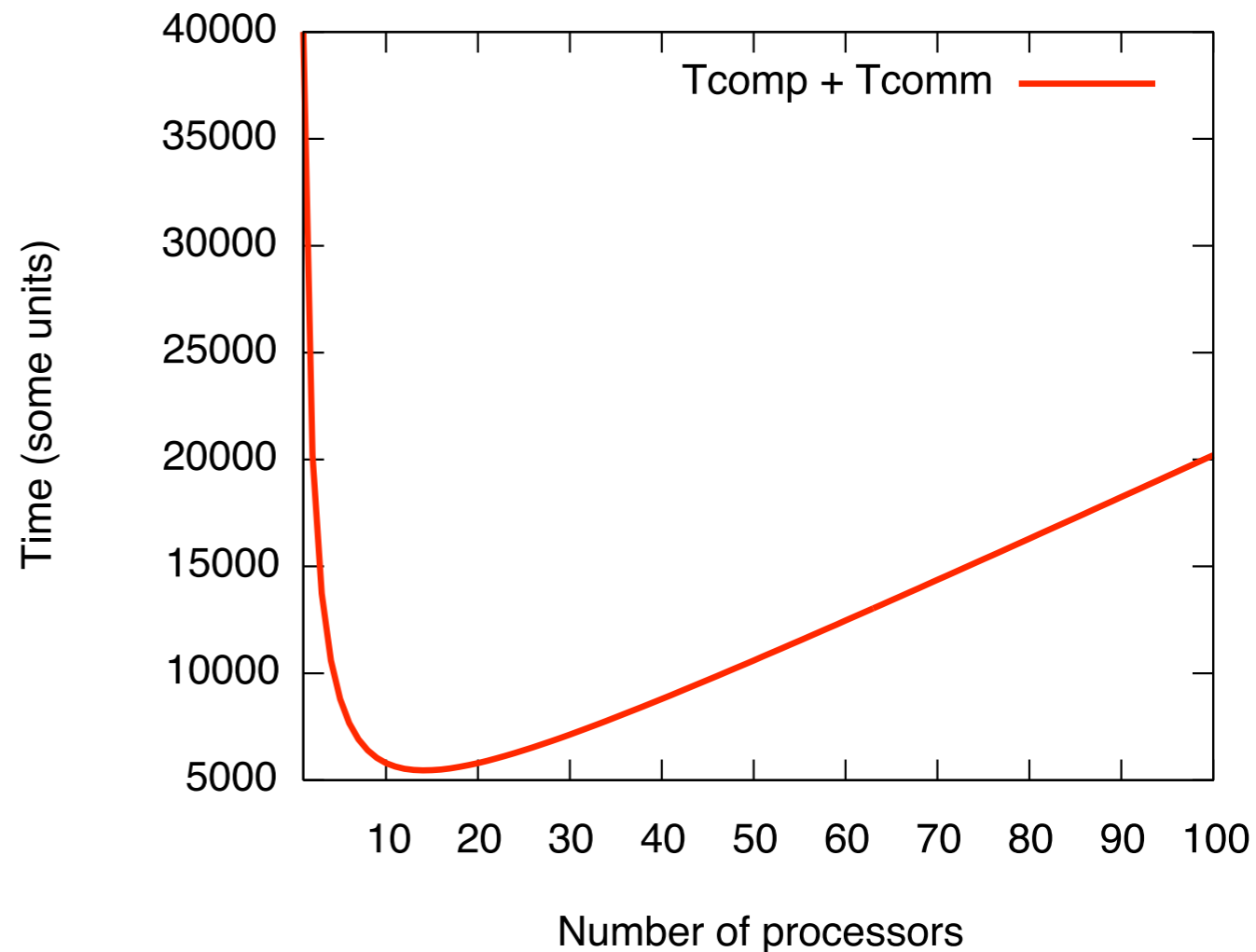
Each processor sends its result (P-1) times and receives (P-1) results

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



Integration with parallel costs:

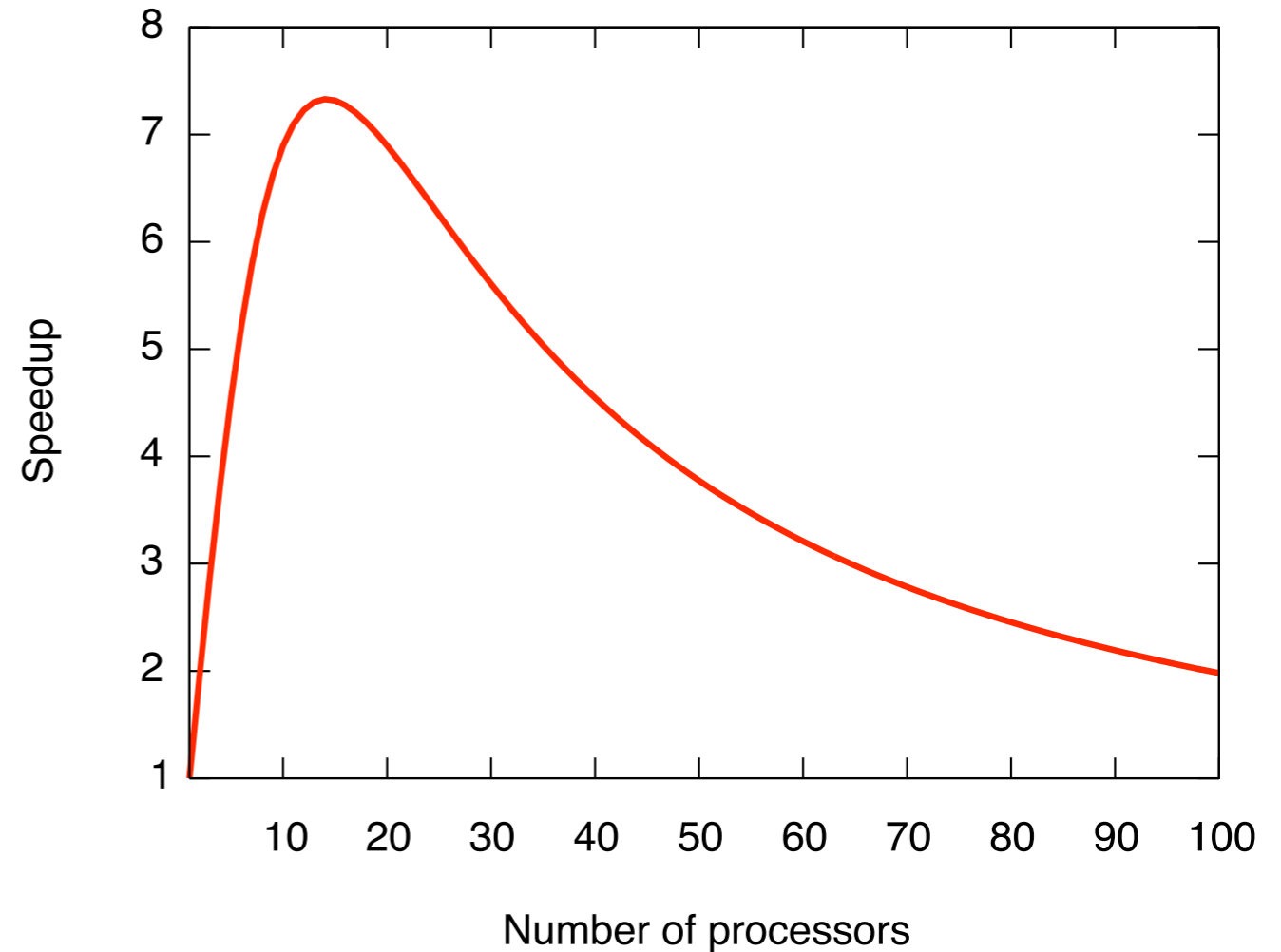
Can actually get worse with P!
Communication cost increases
with P



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

Integration with parallel costs:

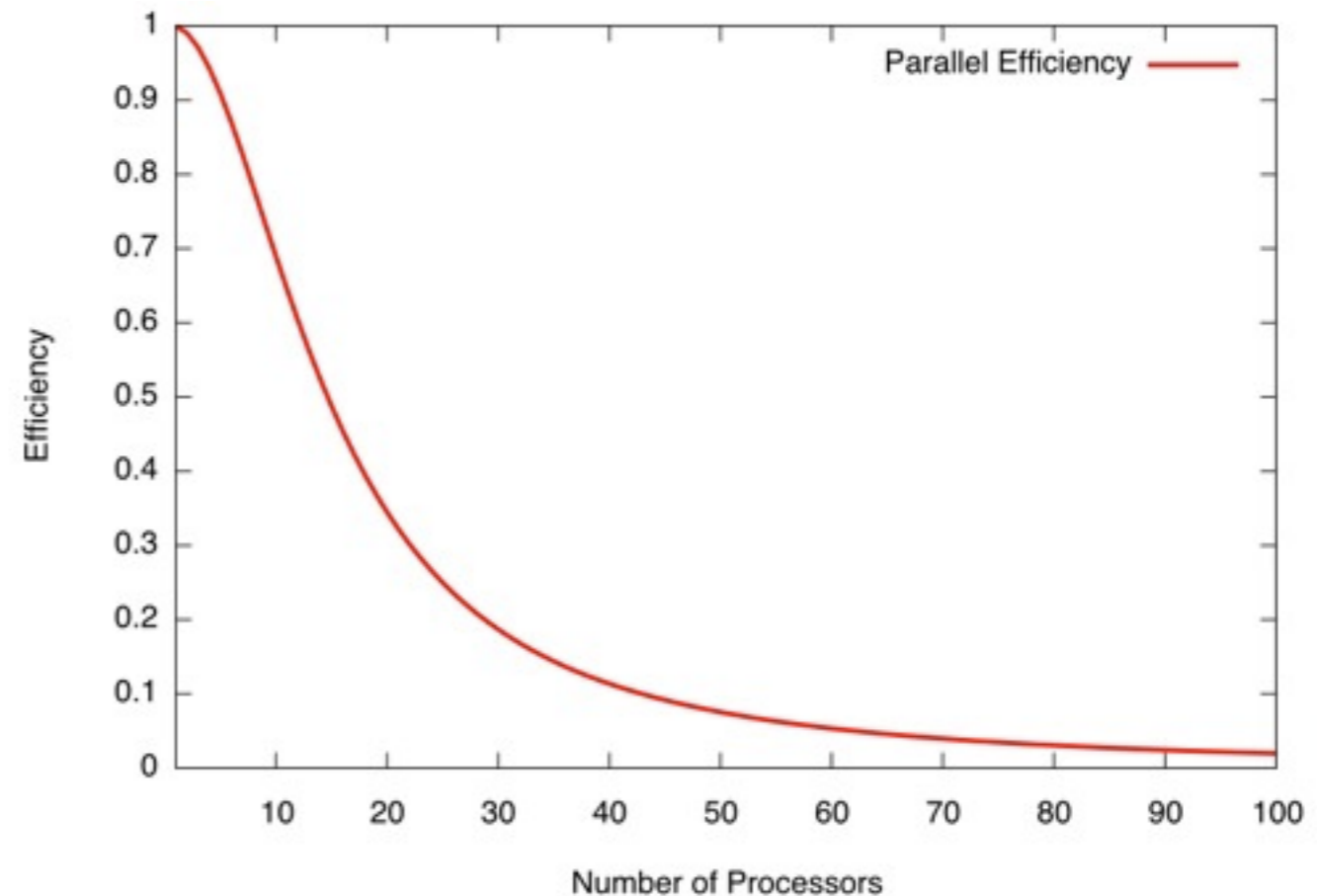
Can actually get worse with P!
Communication cost increases
with P



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

Integration with parallel costs:

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



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

$$\begin{aligned}
 \frac{T_{\text{comm}}}{T_{\text{comp}}} &= \frac{2(P-1)C_{\text{comm}}}{\frac{N}{P}N_{\text{SR}}C_{\text{comp}}} \\
 &= \frac{2P(P-1)}{N} \frac{1}{N_{\text{SR}}} \frac{C_{\text{comm}}}{C_{\text{comp}}} \\
 &\sim P^2
 \end{aligned}$$

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.

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

(Advanced: this even matters for serial computation, due to memory bandwidth limitations. “Arithmetic Intensity”)

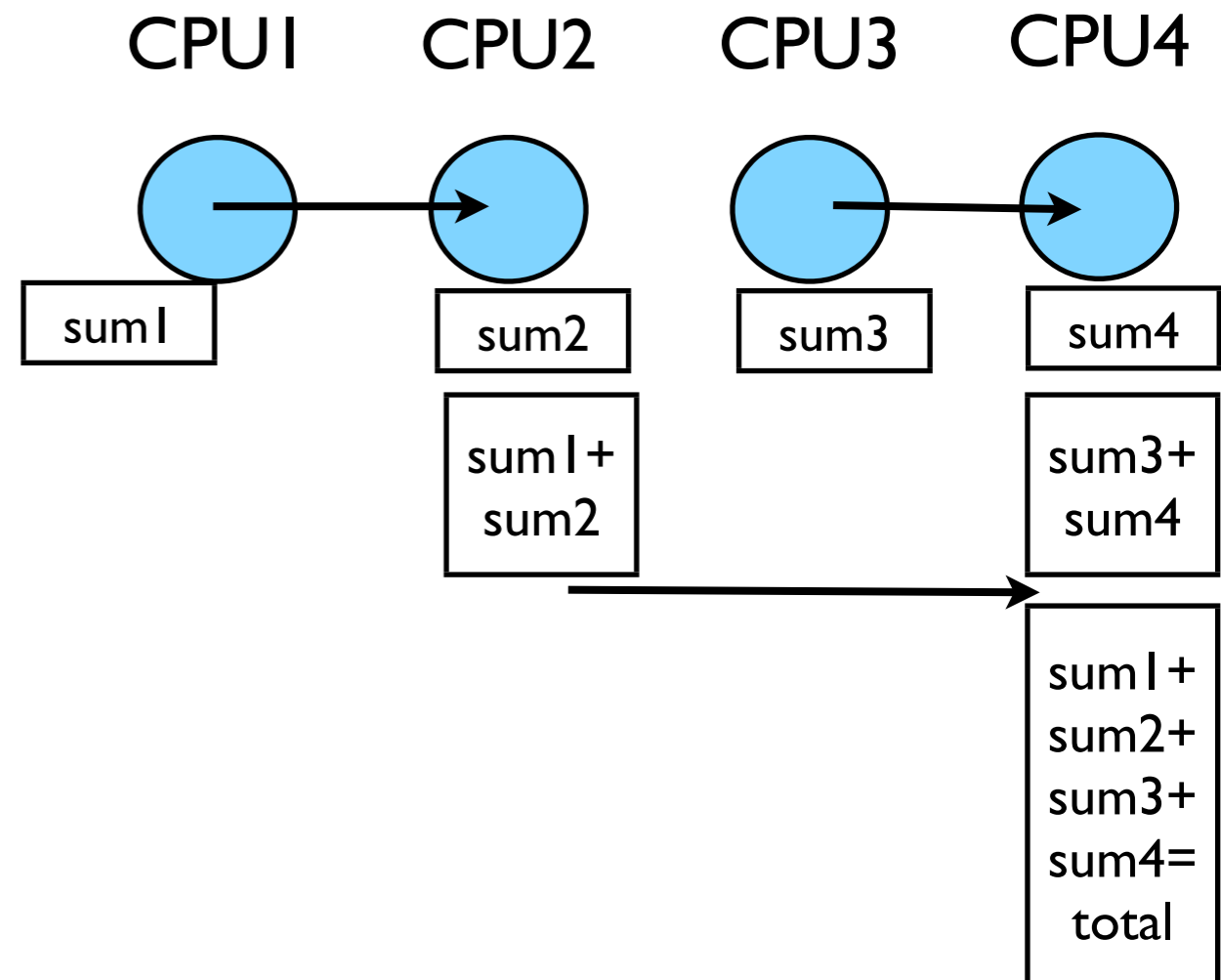
Better Summing

Pairs of processors; send partial
sums

Max messages received $\log_2(P)$

Can repeat to send total back

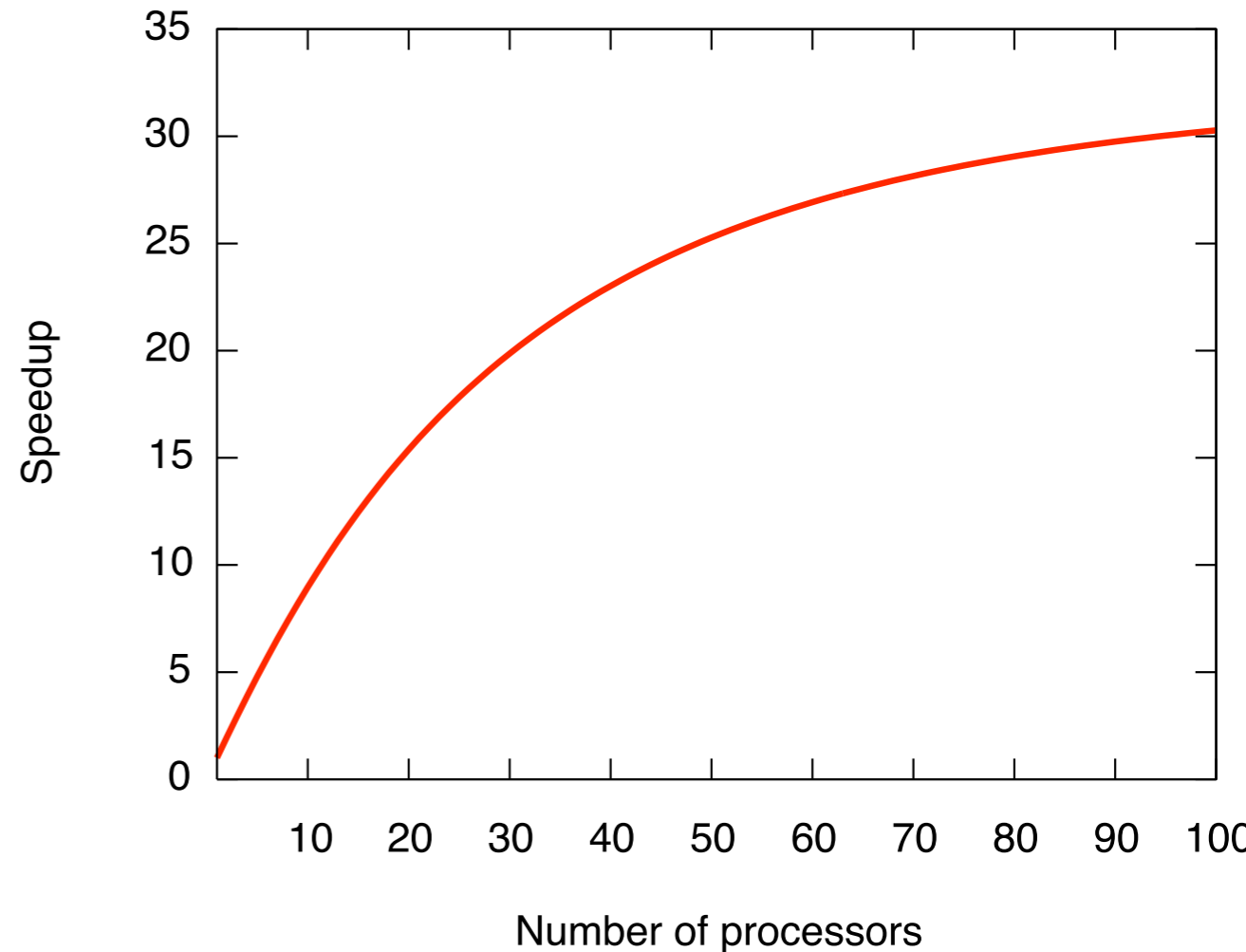
$$T_{\text{comm}} = 2 \log_2(P) C_{\text{comm}}$$



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

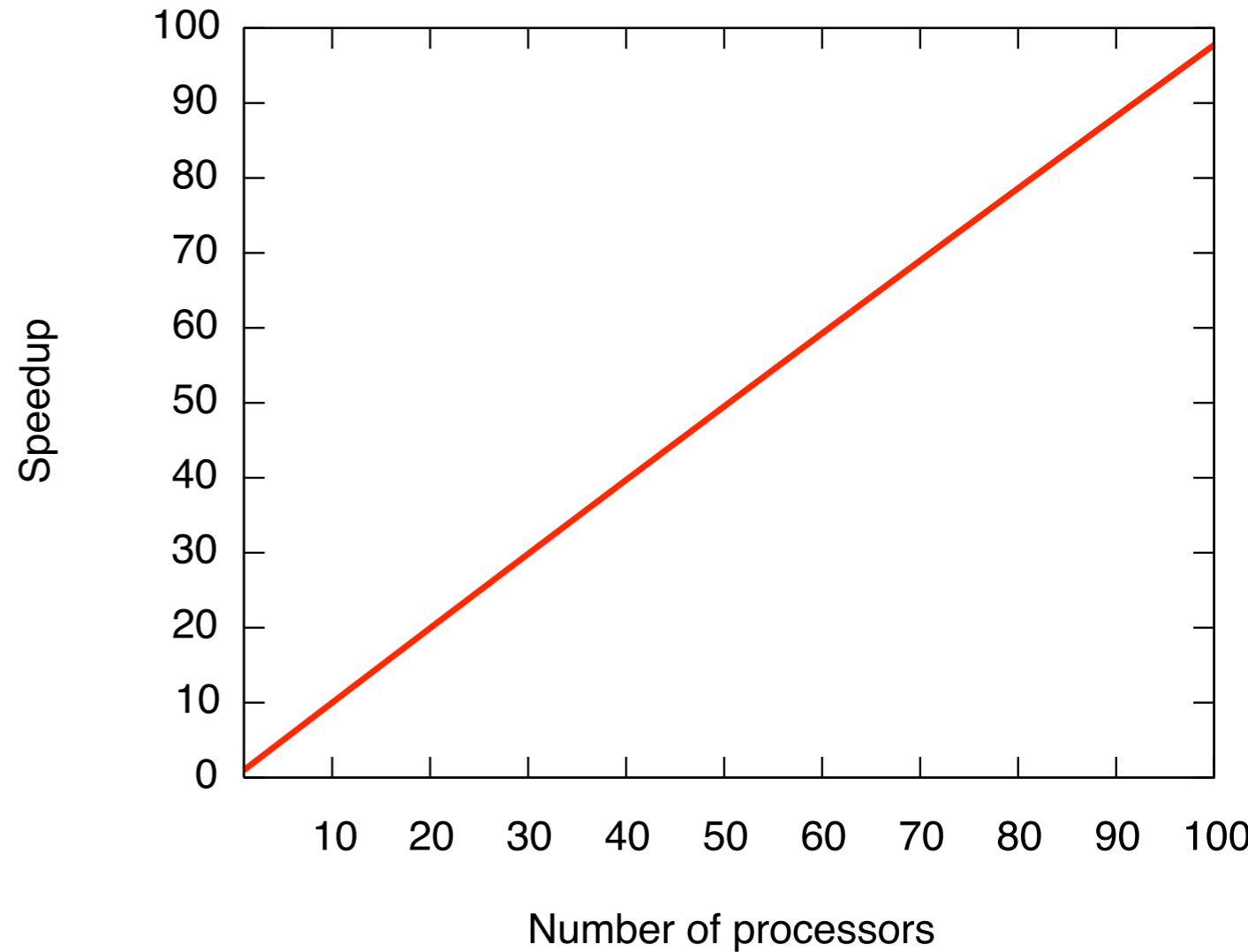
Speedup with reduction

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

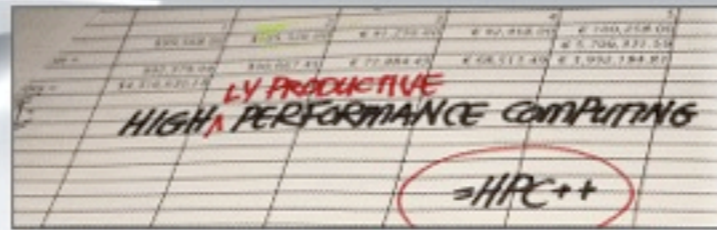
Much better!
As number of processors goes
up, relative cost of
communications goes up only
logarithmically.

$$\begin{aligned}\frac{T_{\text{comm}}}{T_{\text{comp}}} &= \frac{2 \log_2(P) C_{\text{comm}}}{\frac{N}{P} N_{\text{SR}} C_{\text{comp}}} \\ &= \frac{2P \log_2(P)}{N} \frac{1}{N_{\text{SR}}} \frac{C_{\text{comm}}}{C_{\text{comp}}} \\ &\sim P \log_2(P)\end{aligned}$$

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

Parallel Computing

II: Parallel Computers



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	2288.00

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

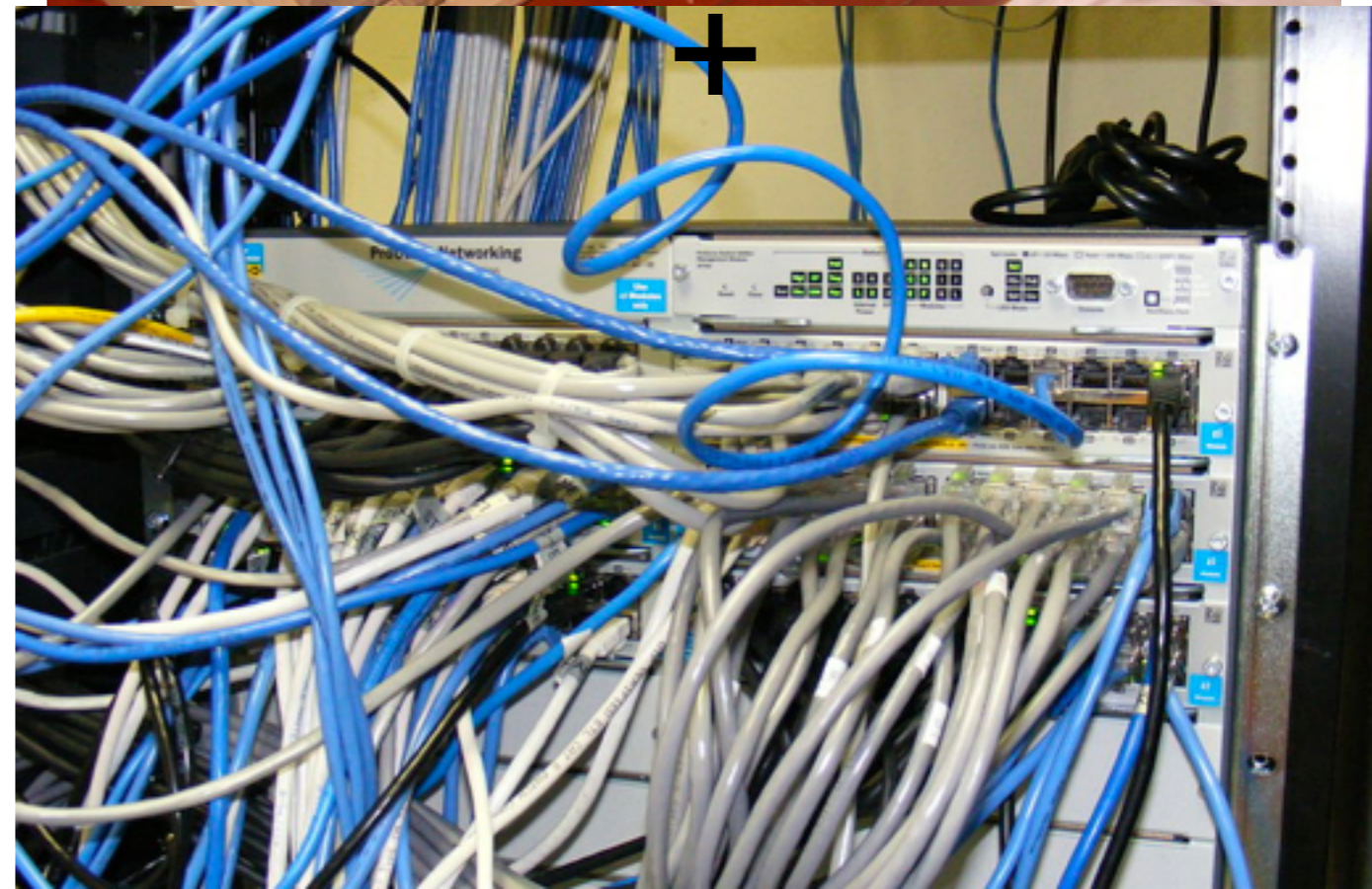
How the computers work shape
how best to program them
Shared Memory vs Distributed
Memory.
Vector computers...



Distributed Memory: Clusters

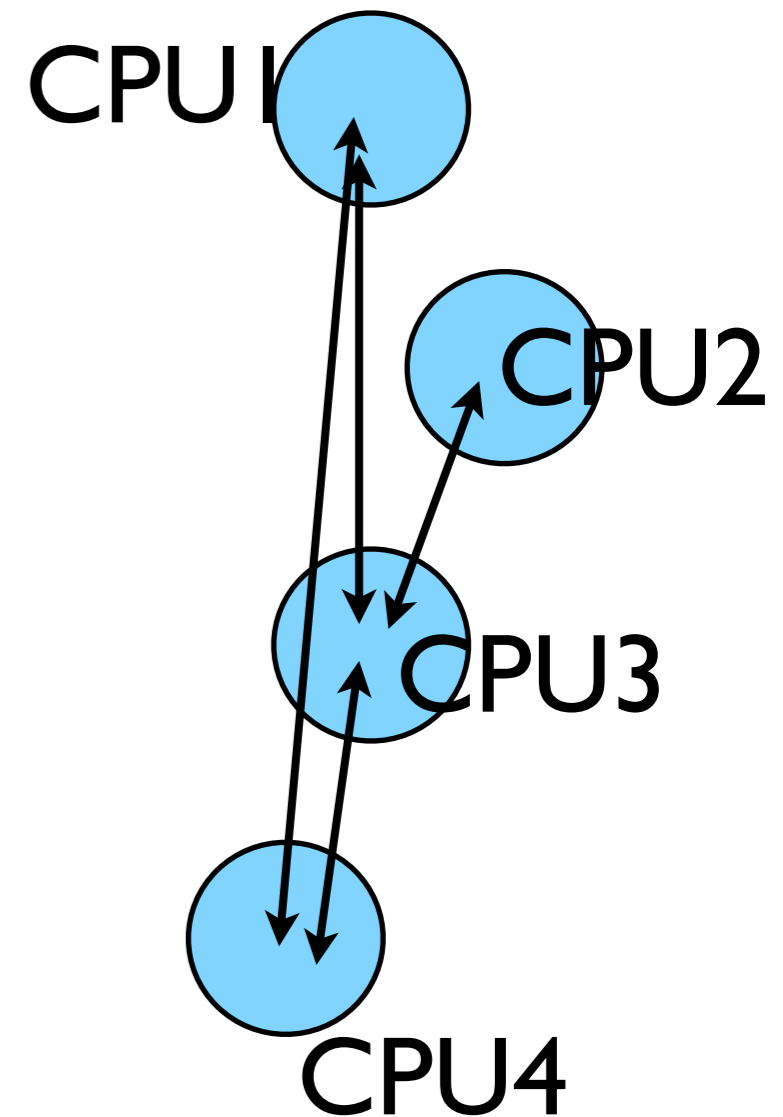
Simplest type of parallel
computer to build

- Take existing powerful
standalone computers
- And network them



Each Node is Independent

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

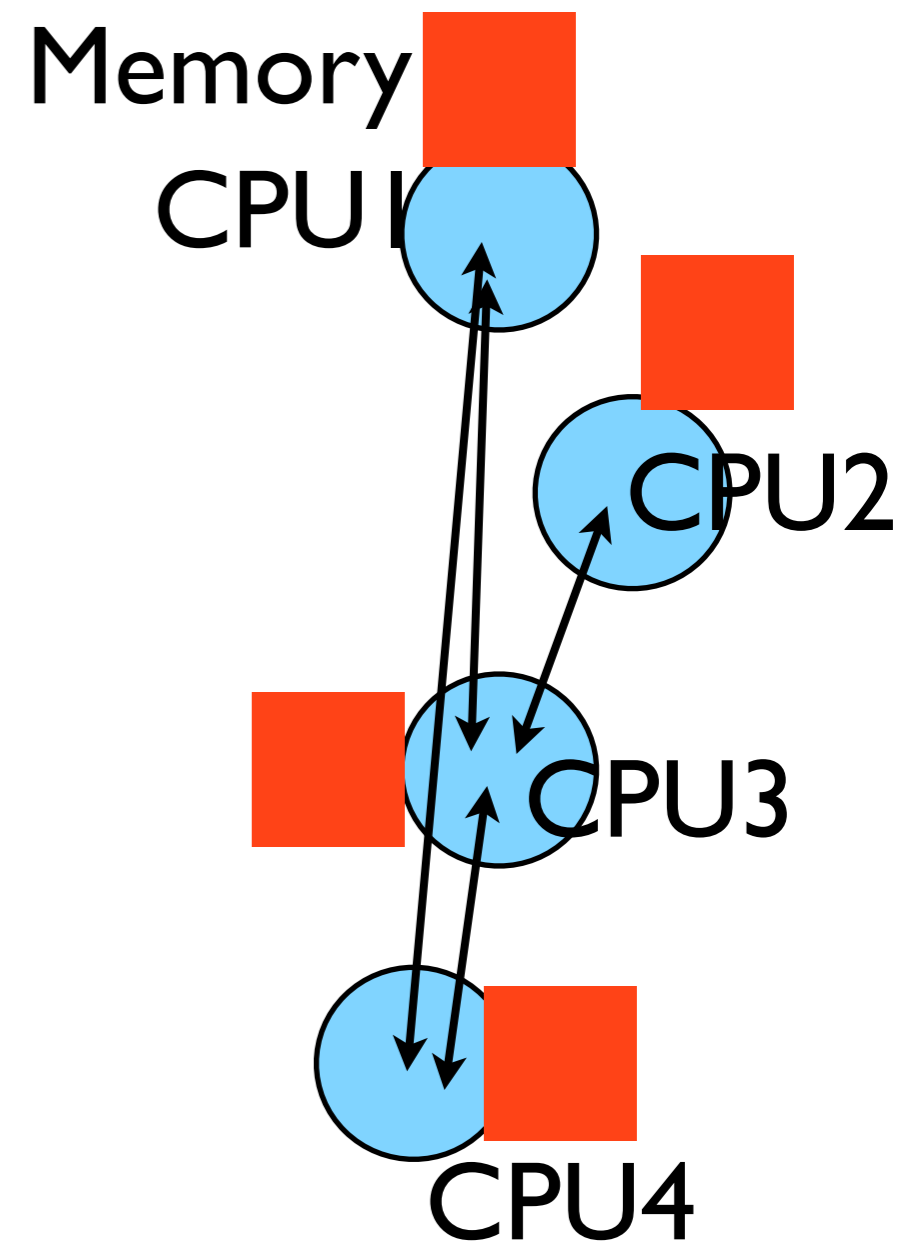


Each node has independent memory

Locally stores its own portion of problem

Whenever it needs information from another region, requests it from appropriate CPU

Usual model: 'message passing'

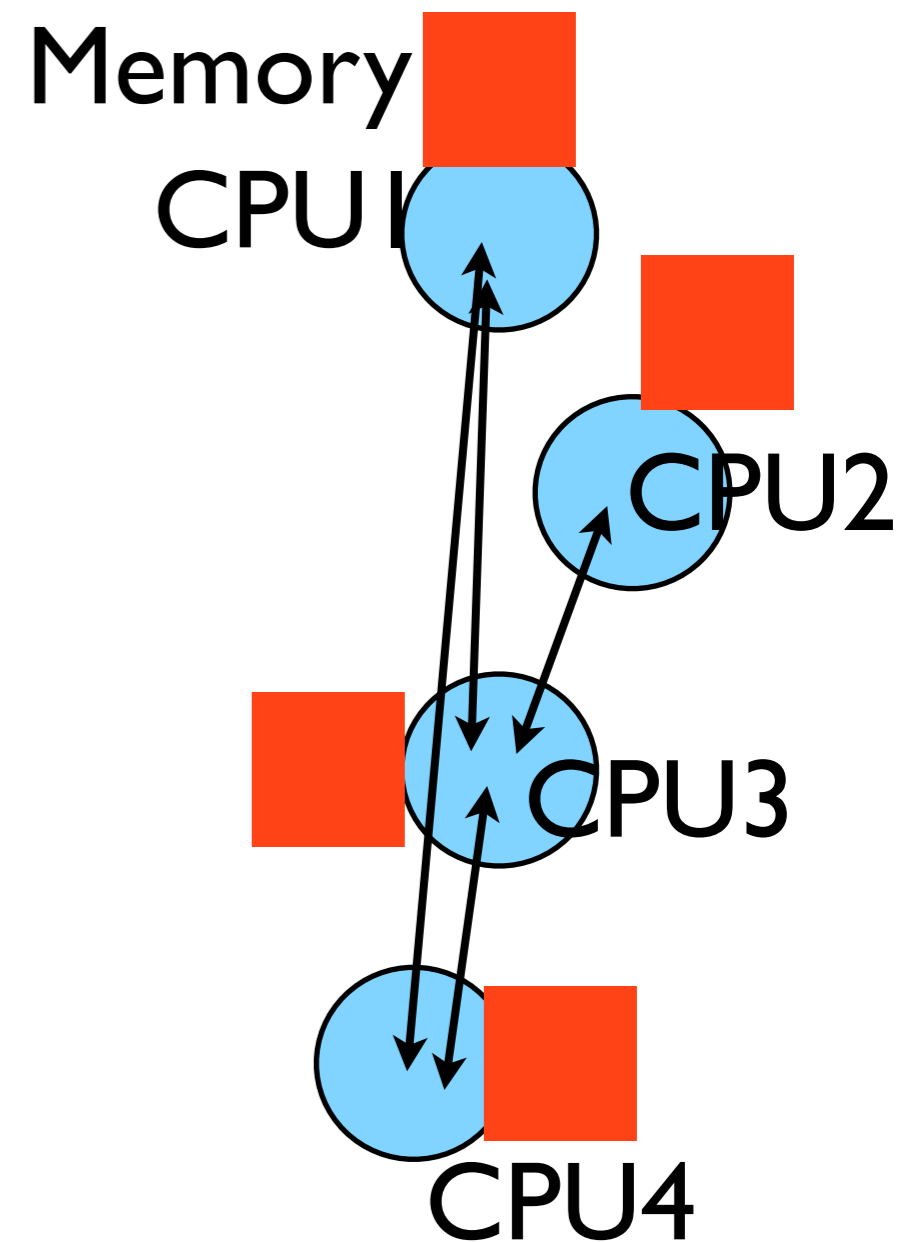


Clusters + Message Passing

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 μ s (10,000 ns)	1 Gb/s (~60 ns/double)
Infiniband	~2 μ s (2,000 ns)	2-10 Gb/s (~10 ns/double)

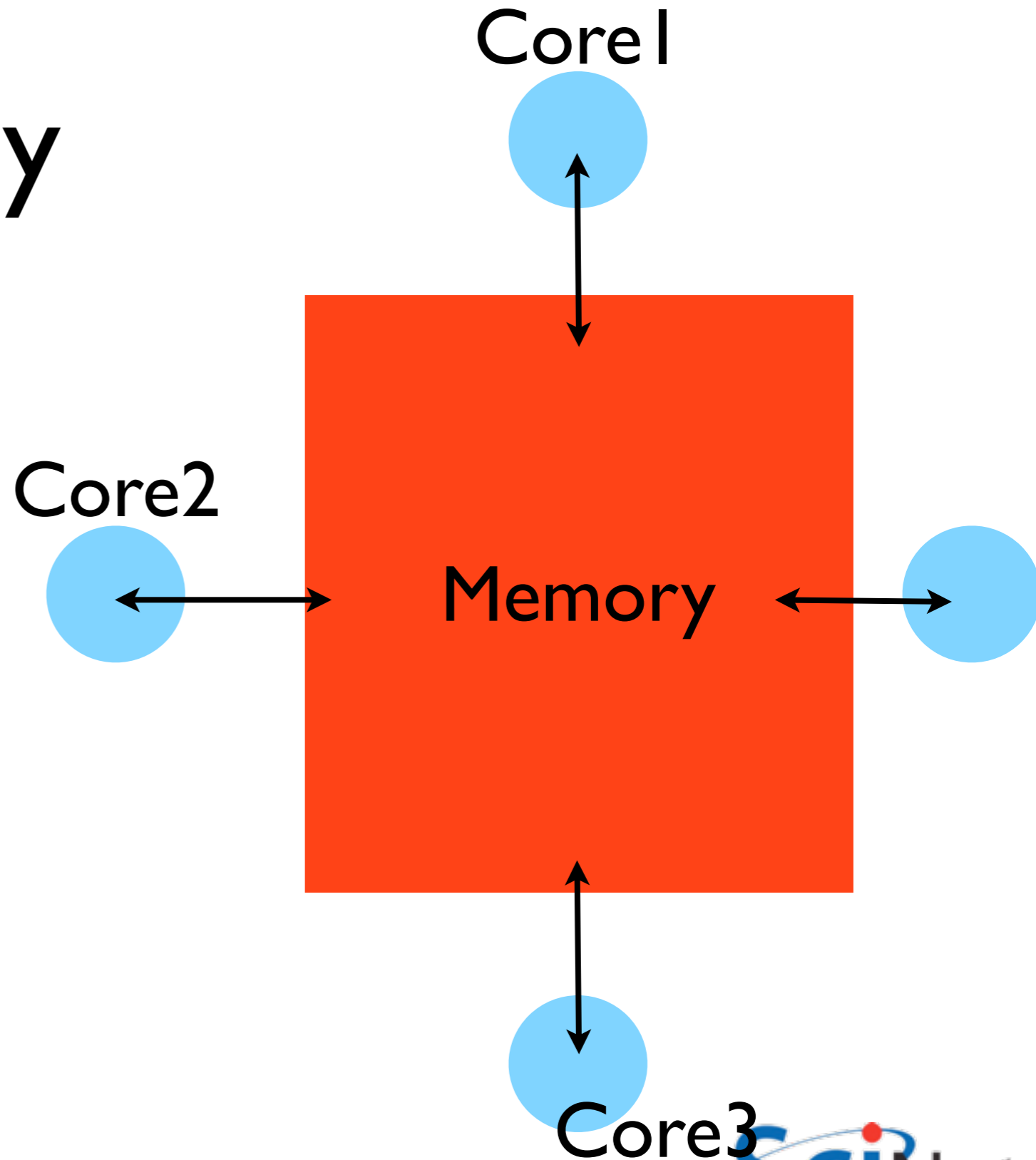
Processor speed: 1 FLOP ~ few ns or less

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: Independent tasks with their own memory, resources

Threads: Threads of execution within one process, 'seeing' the same memory, etc.

OMP
Threads

```
ljdursi@gp
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%st
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%st
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
 18397 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

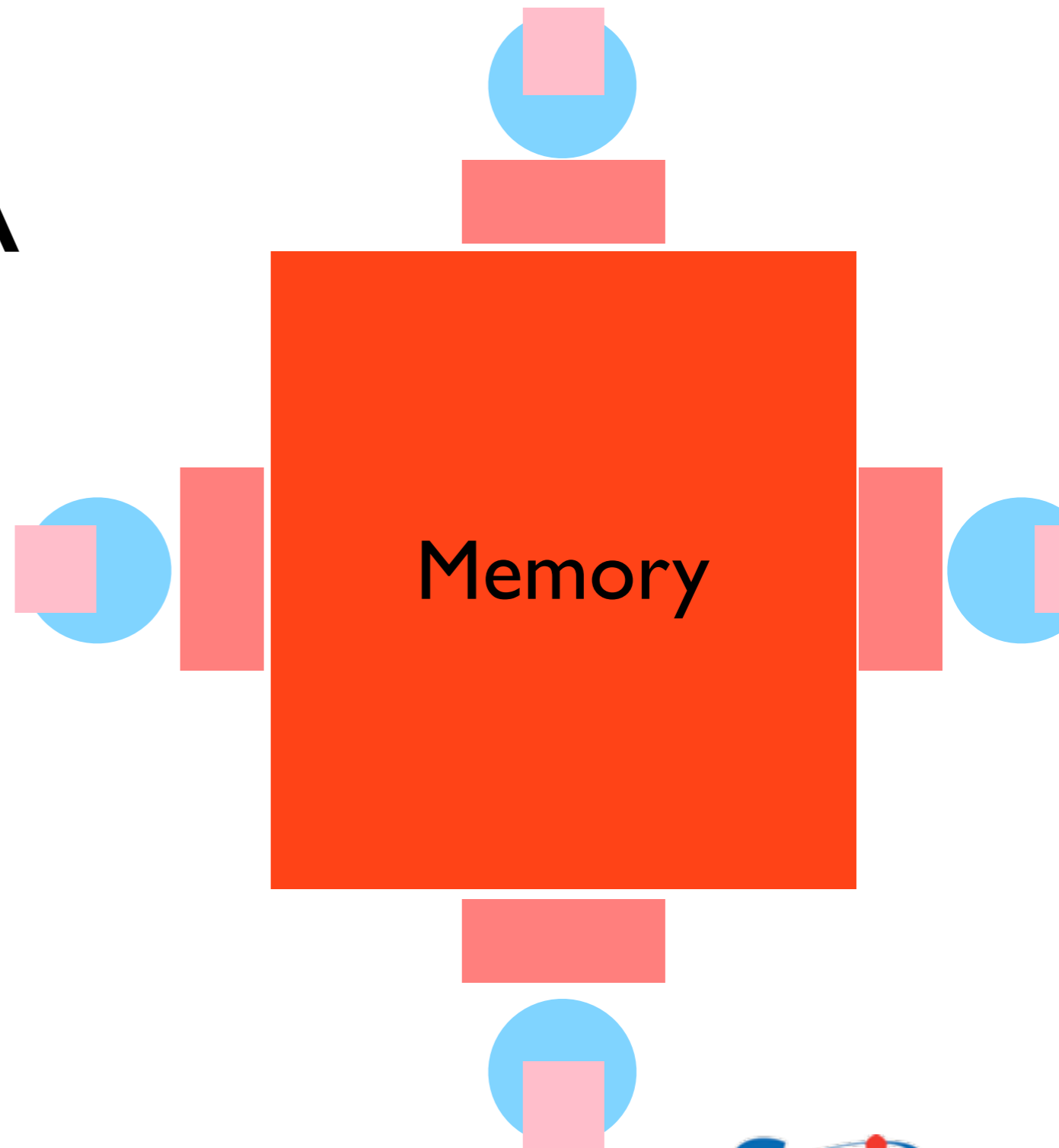
Complicating things: each core typically has some of its own memory

Non-Uniform Memory Access

Locality still matters

Cores have cache, too.

Keeping this memory *coherent* is extremely challenging



Coherency

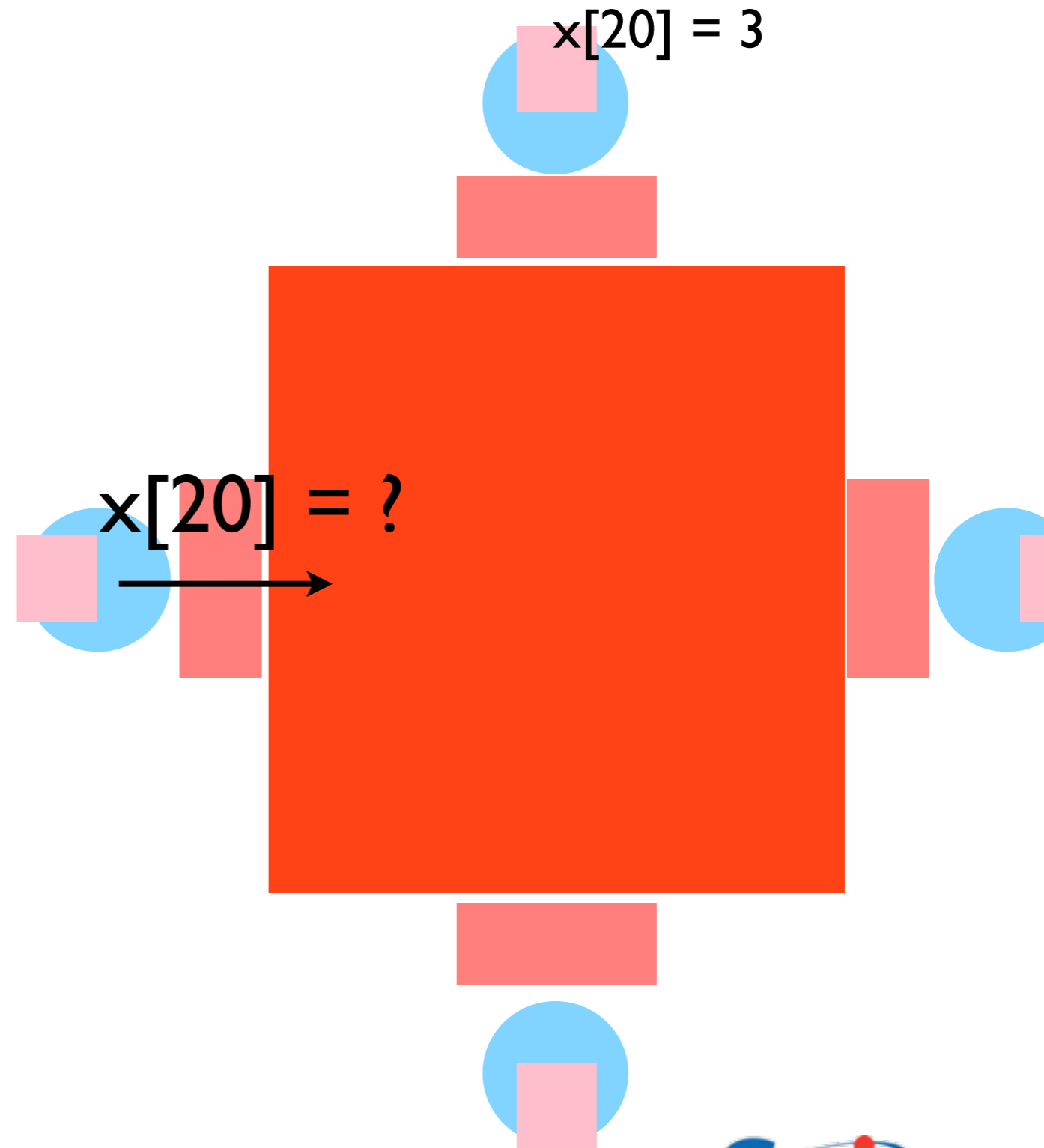
The different levels of memory
imply multiple copies of some
regions

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 μ s (10,000 ns)	1 Gb/s (~60 ns/double)
Infiniband	~2 μ s (2,000 ns)	2-10 Gb/s (~10 ns/double)
NUMA Shared Mem	~0.1 μ s (100 ns)	10-20 Gb/s (~4 ns/double)

Processor speed: 1 FLOP ~ ns or less

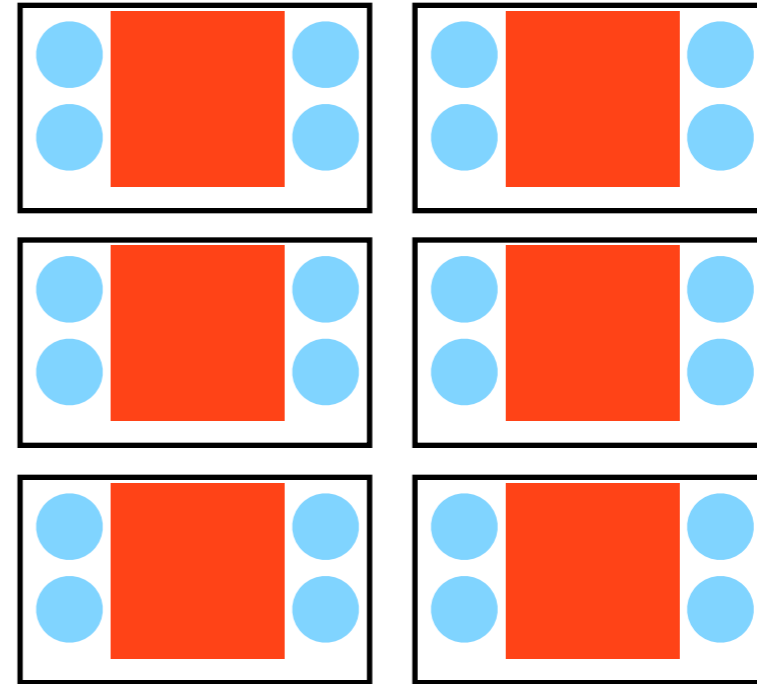
Big Lesson #3

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

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(I node) goes up, hybrid approaches start to make sense.



Before we start with OpenMP:

- `cp -R ~ljdursi/intro-ppp ~/`
- `source ~/intro-ppp/setup`
- `cd ~/intro-ppp/
gettingstarted/`
- `make omp_hello_world`
- `./omp_hello_world`
- `make mpi_hello_world`
- `mpirun -np 8
./mpi_hello_world`
- `qsub -l -X` into your reserved node as per instruction sheet and ensure this works

An introduction to OpenMP

OpenMP

- For Shared Memory systems
- Add Parallelism to functioning serial code
- Add compiler directives to code
- <http://openmp.org> - tonnes of useful info



The screenshot shows the OpenMP.org website. The main header features the "OpenMP" logo and the tagline "THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING". The page is organized into several sections:

- Navigation:** A sidebar on the left contains links for "OpenMP Specifications", "About OpenMP", "Compilers", "Resources", and "Discussion Forum".
- OpenMP News:** The central content area features a "Call for Papers" for the 7th International Workshop on OpenMP (IWOMP 2011) in Chicago, USA, and a section for "IWOMP 2010 Material Available" which includes a list of PDF documents like "Welcome", "Basic Concepts in Parallelization", and "Getting OpenMP Up To Speed".
- Events:** A section titled "Events" lists the "IWOMP 2011 Call for Papers" with dates and location.
- Input Register:** A section for users to alert the webmaster about new products or updates.
- Search and Archives:** A search bar and a list of archives for September 2010, July 2010, and May 2010.

OpenMP

- Compiler, run-time environment does a lot of work for us
- Divides up work
- But we have to tell it how to use variables, where to run in parallel



The screenshot shows the OpenMP.org website. The main header features the OpenMP logo and the text "THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING". Below the header, there is a navigation menu with links for "OpenMP Specifications", "About OpenMP", "Compilers", "Resources", and "Discussion Forum". The "OpenMP News" section is prominent, featuring a "Call for Papers" for the 7th International Workshop on OpenMP (IWOMP 2011) in Chicago, USA, and a notice about the availability of materials from the 2010 workshop. The website also includes a search bar, an input register, and an archives section.

OpenMP

- Mark off parallel regions
- in those regions, all available threads do same work
- Markup designed to be invisible to non-OpenMP compilers; should result in working serial code



The screenshot shows the OpenMP.org website. At the top, the OpenMP logo is displayed with the tagline "THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING". Below the logo is a navigation menu with links for "OpenMP Specifications", "About OpenMP", "Compilers", "Resources", and "Discussion Forum". The main content area features "OpenMP News" with two articles: "IWOMP 2011 - Call For Papers" and "IWOMP 2010 Material Available". The "IWOMP 2011" article includes a call for papers for the 7th International Workshop on OpenMP (IWOMP 2011) held in Chicago, USA, from June 13-15, 2011. The "IWOMP 2010" article mentions that the workshop was held in Tsukuba, Japan, and that the papers presented are available as a book published by Springer Verlag, titled "Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More". A sidebar on the left contains a "Subscribe to the News Feed" link, an "Input Register" form, a "Search OpenMP.org" box, and an "Archives" section with links for September 2010, July 2010, and May 2010. A sidebar on the right contains sections for "The OpenMP" (supporting multi-platform memory parallel programming in C/C++ and Fortran), "Get" (OpenMP specs), "Use" (OpenMP Compilers), and "Learn" (Using OpenMP).

C: omp-hello-world.c

gcc -fopenmp -o omp-hello-world omp-hello-world.c -lgomp

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    printf("At start of program\n");
#pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
              omp_get_thread_num());
    }
    return 0;
}
```

F90: omp-hello-world.f90

gfortran -fopenmp -o omp-hello-world omp-hello-world.f90 -lgomp

```
program omp_hello_world
use omp_lib
implicit none

print *, 'At start of program'
!$omp parallel
    print *, 'Hello world from thread ', &
            omp_get_thread_num(), '!'
!$omp end parallel
end program omp_hello_world
```

```
$ gcc -o omp-hello-world omp-hello-world.c -fopenmp -lgomp
or
$ gfortran -o omp-hello-world omp-hello-world.f90 -fopenmp -lgomp

$ export OMP_NUM_THREADS=8
$ ./omp-hello-world
...
$ export OMP_NUM_THREADS=1
$ ./omp-hello-world
...
$ export OMP_NUM_THREADS=32
$ ./omp-hello-world
...
```

```
gpc-f102n084-$ gcc -o omp-hello-world omp-hello-world.c -fopenmp -lgomp
gpc-f102n084-$ export OMP_NUM_THREADS=8
gpc-f102n084-$ ./omp-hello-world
At start of program
Hello, world, from thread 0!
Hello, world, from thread 6!
Hello, world, from thread 5!
Hello, world, from thread 4!
Hello, world, from thread 2!
Hello, world, from thread 1!
Hello, world, from thread 7!
Hello, world, from thread 3!
gpc-f102n084-$ export OMP_NUM_THREADS=1
gpc-f102n084-$ ./omp-hello-world
At start of program
Hello, world, from thread 0!
gpc-f102n084-$ export OMP_NUM_THREADS=32
gpc-f102n084-$ ./omp-hello-world
At start of program
Hello, world, from thread 11!
Hello, world, from thread 1!
Hello, world, from thread 16!
...
```


What did happen?

- OMP_NUM_THREADS threads launched
- Each print “Hello world...”
- In seemingly random order
- Only one ‘At start of program’

```
gpc-f102n084-$ gcc -o omp-hello-world omp-hello-world.c
-fopenmp -lgomp
gpc-f102n084-$ export OMP_NUM_THREADS=8
gpc-f102n084-$ ./omp-hello-world
At start of program
Hello, world, from thread 0!
Hello, world, from thread 6!
Hello, world, from thread 5!
Hello, world, from thread 4!
Hello, world, from thread 2!
Hello, world, from thread 1!
Hello, world, from thread 7!
Hello, world, from thread 3!
gpc-f102n084-$ export OMP_NUM_THREADS=1
gpc-f102n084-$ ./omp-hello-world
At start of program
Hello, world, from thread 0!
gpc-f102n084-$ export OMP_NUM_THREADS=32
gpc-f102n084-$ ./omp-hello-world
At start of program
Hello, world, from thread 11!
Hello, world, from thread 1!
Hello, world, from thread 16!
...
```

```

#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    printf("At start of program\n");
#pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
              omp_get_thread_num());
    }
    return 0;
}

```

Include definitions
for OpenMP
supporting library
(omp_get_thread_num())

```

program omp_hello_world
use omp_lib
implicit none

print *, 'At start of program'
!$omp parallel
    print *, 'Hello world from thread ', &
            omp_get_thread_num(), '!'
!$omp end parallel

end program omp_hello_world

```

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    printf("At start of program\n");
    #pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
            omp_get_thread_num());
    }
    return 0;
}
```

Program starts normally
(Single thread of
execution)

```
program omp_hello_world
use omp_lib
implicit none

print *, 'At start of program'
!$omp parallel
    print *, 'Hello world from thread ', &
        omp_get_thread_num(), '!'
!$omp end parallel

end program omp_hello_world
```

```

#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    printf("At start of program\n");
#pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
            omp_get_thread_num());
    }
    return 0;
}

```

At start of parallel section, **OMP_NUM_THREADS** threads are launched, each execute same code.

```

program omp_hello_world
use omp_lib
implicit none

print *, 'At start of program'
!$omp parallel
    print *, 'Hello world from thread ', &
        omp_get_thread_num(), '!'
!$omp end parallel

end program omp_hello_world

```

```

#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    printf("At start of program\n");
#pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
              omp_get_thread_num());
    }
    return 0;
}

```

```

program omp_hello_world
use omp_lib
implicit none

print *, 'At start of program'
!$omp parallel
    print *, 'Hello world from thread ', &
           omp_get_thread_num(), '!'
!$omp end parallel

end program omp_hello_world

```

At end of parallel section, the threads join back up and back to serial execution

```

#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    printf("At start of program\n");
#pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
               omp_get_thread_num());
    }
    return 0;
}

```

Special OMP function
called to find the thread
number of current thread
(first = 0)

```

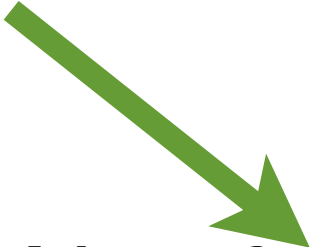
program omp_hello_world
use omp_lib
implicit none

print *, 'At start of program'
!$omp parallel
    print *, 'Hello world from thread ', &
            omp_get_thread_num(), '!'
!$omp end parallel


end program omp_hello_world

```

Turn OpenMP on in compiler (default off; incantation varies from compiler to compiler. Intel: `-openmp`).
Always needed for OpenMP code.



```
$ gcc -o omp-hello-world omp-hello-world.c -fopenmp -lgomp  
or  
$ gfortran -o omp-hello-world omp-hello-world.f90 -fopenmp -lgomp
```



Link in OpenMP libraries;
normally only needed if
you use functions like
`omp_get_num_threads()`.
Only at link time.

\$ gcc -o omp-hello-world omp-hello-world.c -fopenmp **-lgomp**
or

\$ gfortran -o omp-hello-world omp-hello-world.f90 -fopenmp **-lgomp**


```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    printf("At start of program\n");
#pragma omp parallel
    {
        printf("Hello world from thread %d of %d!\n",
               omp_get_thread_num(),
               omp_get_num_threads());
    }
    return 0;
}
```

(Advanced: can set num_threads (but not thread_num), too.)



```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    printf("At start of program\n");
#pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
            omp_get_thread_num());
    }
    printf("There were %d threads.\n",
        omp_get_num_threads());
    return 0;
}
```

Variables in OpenMP

- Need to put a variable in the parallel section to store the value
- But variables in parallel sections are a little tricky.

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {

    printf("At start of program\n");
#pragma omp parallel
    {
        printf("Hello world from thread %d!\n",
              omp_get_thread_num());
    }
    printf("There were %d threads.\n",
          omp_get_num_threads() );
    return 0;
}
```

C: omp-vars.c

gcc -fopenmp -o omp-vars omp-vars.c -lgomp

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
    int mythread, nthreads;
    #pragma omp parallel default(none), shared(nthreads), private(mythread)
    {
        mythread = omp_get_thread_num();
        if (mythread == 0)
            nthreads = omp_get_num_threads();
    }
    printf("Number of threads was %d.\n", nthreads);
    return 0;
}
```

FORTRAN: omp-vars.f90

`gfortran -fopenmp -o omp-vars omp-vars.f90 -lgomp`

```
program omp_vars
use omp_lib
implicit none

integer :: mythread, nthreads

!$omp parallel default(none), private(mythread), shared(nthreads)
  mythread = omp_get_thread_num()
  if (mythread == 0) then
    nthreads = omp_get_num_threads()
  endif
!$omp end parallel

print *, 'Number of threads was ', nthreads, '.'

end program omp_vars
```

Variable definitions, and how they are used in the parallel block.

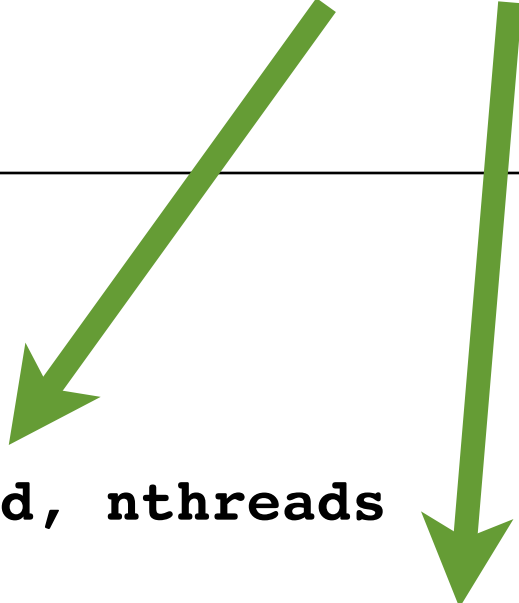
```
program omp_vars
use omp_lib
implicit none

integer :: mythread, nthreads

!$omp parallel default(none), private(mythread), shared(nthreads)
  mythread = omp_get_thread_num()
  if (mythread == 0) then
    nthreads = omp_get_num_threads()
  endif
!$omp end parallel

print *, 'Number of threads was ', nthreads, '.'

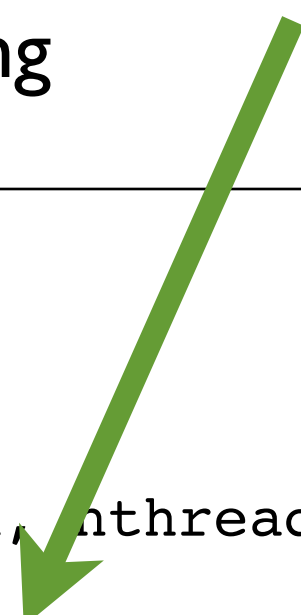
end program omp_vars
```



Strongly, strongly, strongly recommended.

Inconvenient?

30 seconds of extra typing can save you *hours* of debugging



```
program omp_vars
use omp_lib
implicit none

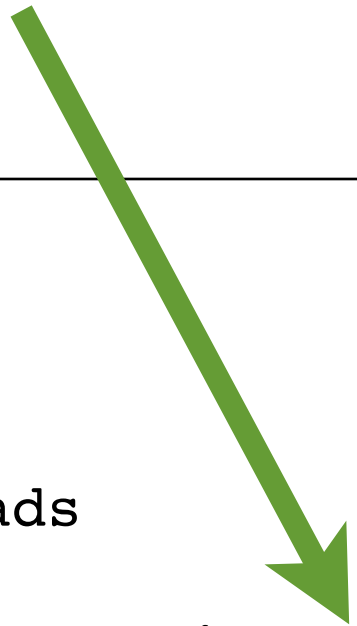
integer :: mythread, nthreads

!$omp parallel default(none), private(mythread), shared(nthreads)
  mythread = omp_get_thread_num()
  if (mythread == 0) then
    nthreads = omp_get_num_threads()
  endif
!$omp end parallel

print *, 'Number of threads was ', nthreads, '.'

end program omp_vars
```

Each thread gets its own private copy of mythread to do with as it pleases. No other thread can see, modify.



```
program omp_vars
use omp_lib
implicit none

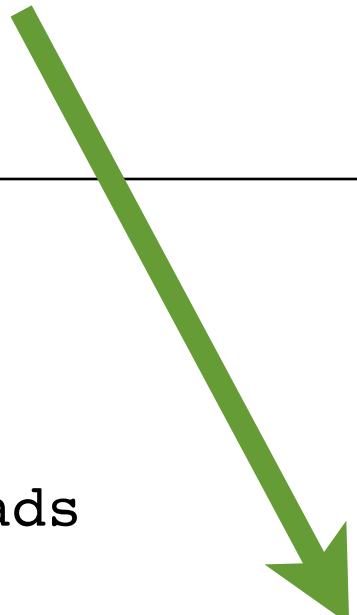
integer :: mythread, nthreads

!$omp parallel default(none), private(mythread), shared(nthreads)
  mythread = omp_get_thread_num()
  if (mythread == 0) then
    nthreads = omp_get_num_threads()
  endif
!$omp end parallel

print *, 'Number of threads was ', nthreads, '.'

end program omp_vars
```


A thread-private variable has *undefined value* inside a parallel block.



```
program omp_vars
use omp_lib
implicit none

integer :: mythread, nthreads

!$omp parallel default(none), private(mythread), shared(nthreads)
  mythread = omp_get_thread_num()
  if (mythread == 0) then
    nthreads = omp_get_num_threads()
  endif
!$omp end parallel

print *, 'Number of threads was ', nthreads, '.'

end program omp_vars
```

(Advanced: firstprivate, lastprivate - copy in/out.)

Everyone can see (ok), modify (danger! danger!) a shared variable. Keeps its value between serial/parallel sections

```
program omp_vars
use omp_lib
implicit none

integer :: mythread, nthreads

!$omp parallel default(none), private(mythread), shared(nthreads)
  mythread = omp_get_thread_num()
  if (mythread == 0) then
    nthreads = omp_get_num_threads()
  endif
!$omp end parallel

print *, 'Number of threads was ', nthreads, '.'

end program omp_vars
```

Variables in OpenMP

- Program runs, launches threads.
- Each thread gets its own copy of mythread
- **Only** thread 0 writes to nthreads
- Outputs number of threads
- What would mythread be if we printed it?

```
program omp_vars
use omp_lib
implicit none

integer :: mythread, nthreads

!$omp parallel default(none), private(mythread), shared
(nthreads)
    mythread = omp_get_thread_num()
    if (mythread == 0) then
        nthreads = omp_get_num_threads()
    endif
!$omp end parallel

print *, 'Number of threads was ', nthreads, '.'

end program omp_vars
```

For C folks:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
    int nthreads;
#pragma omp parallel default(none), shared(nthreads)
    {
        int mythread;
        mythread = omp_get_thread_num();
        if (mythread == 0)
            nthreads = omp_get_num_threads();
    }
    printf("Number of threads was %d.\n",nthreads);
    return 0;
}
```

Local definitions are powerful, and avoid lots of bugs!
Variables defined in a parallel block are automatically
thread private.

Single Execution in OpenMP

- Do we care that it's thread 0 in particular that updates nthreads?
- Why did we pick 0?
- Often we just want the first thread through to do something, don't care who.

```
program omp_vars
use omp_lib
implicit none

integer :: mythread, nthreads

!$omp parallel default(none), private(mythread), shared
(nthreads)
    mythread = omp_get_thread_num()
    if (mythread == 0) then
        nthreads = omp_get_num_threads()
    endif
!$omp end parallel

print *, 'Number of threads was ', nthreads, '.'

end program omp_vars
```

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
    int nthreads;
#pragma omp parallel default(none), shared(nthreads)
#pragma omp single
        nthreads = omp_get_num_threads();
    printf("Number of threads was %d.\n",nthreads);
    return 0;
}
```

```
program omp_vars
use omp_lib
implicit none

integer :: nthreads

!$omp parallel default(none), shared(nthreads)
!$omp single
    nthreads = omp_get_num_threads()
!$omp end single
!$omp end parallel

print *, 'Number of threads was ', nthreads, '.'

end program omp_vars
```

Loops in OpenMP

- Now let's try something a little more interesting
- copy one of your omp programs to `omp_loop.c` (or `omp_loop.f90`) and let's put a loop in the parallel section

```

#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
    int i, mythread;
#pragma omp parallel default(none) XXXX(i) XXXX(mythread)
    {
        mythread = omp_get_thread_num();
        for (i=0; i<16;i++) {
            printf("Thread %d gets i=%d\n",mythread,i);
        }
    }
    return 0;
}

```

```

program omp_loop
use omp_lib
implicit none

integer :: i, mythread

!$omp parallel default(none) XXXX(i) XXXX(mythread)
    mythread = omp_get_thread_num()
    do i=1,16
        print *, 'thread ', mythread, ' gets i=', i
    enddo
!$omp end parallel

end program omp_loop

```


Worksharing constructs in OpenMP

- We don't generally want tasks to do exactly the same thing
- Want to partition a problem into pieces, each thread works on a piece
- Most scientific programming full of work-heavy loops
- OpenMP has a worksharing construct: `omp for` (or `omp do`)

```
program omp_loop
use omp_lib
implicit none

integer :: i, mythread

!$omp parallel default(none) XXXX(i) XXXX(mythread)
  mythread = omp_get_thread_num()
  do i=1,16
    print *, 'thread ', mythread, ' gets i=', i
  enddo
!$omp end parallel

end program omp_loop
```

(Advanced: Can combine `parallel` and `for` into one `omp` line.)

```

#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv) {
    int i, mythread;
#pragma omp parallel default(none) XXXX(i) XXXX(mythread)
    {
        mythread = omp_get_thread_num();
#pragma omp for
        for (i=0; i<16;i++) {
            printf("Thread %d gets i=%d\n",mythread,i);
        }
    }
    return 0;
}

```

```

program omp_loop
use omp_lib
implicit none

integer :: i, mythread
!$omp parallel default(none) XXXX(i) XXXX(mythread)
    mythread = omp_get_thread_num()
!$omp do
    do i=1,16
        print *, 'thread ', mythread, ' gets i=', i
    enddo
!$omp end parallel

end program omp_loop

```

Worksharing constructs in OpenMP

- `omp for / omp do` construct breaks up the iterations by thread.
- If doesn't divide evenly, does the best it can.
- Allows easy breaking up of work!

```
$ ./omp_loop
thread      3  gets i=      7
thread      3  gets i=      8
thread      4  gets i=      9
thread      4  gets i=     10
thread      5  gets i=     11
thread      5  gets i=     12
thread      6  gets i=     13
thread      6  gets i=     14
thread      1  gets i=      3
thread      1  gets i=      4
thread      0  gets i=      1
thread      0  gets i=      2
thread      2  gets i=      5
thread      2  gets i=      6
thread      7  gets i=     15
thread      7  gets i=     16
$
```

(Advanced: can break up work of arbitrary blocks of code with “`omp task`” construct.)

DAXPY

- multiply a vector by a scalar, add a vector.
- (a X plus Y, in double precision)
- Implement this, first serially, then with OpenMP
- `daxpy.c` or `daxpy.f90`
- `make daxpy` or `make fdaxpy`

$$\hat{z} = a\hat{x} + \hat{y}$$

make

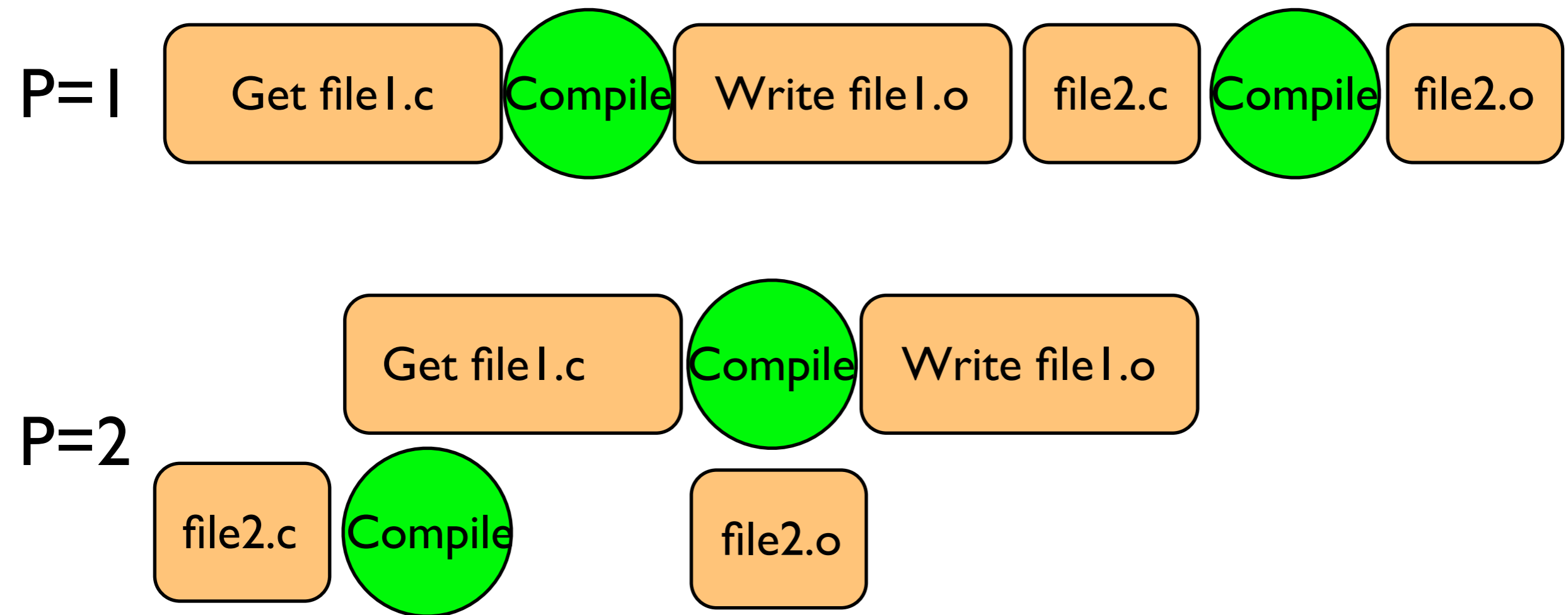
- Make builds an executable from a list of source code files and rules
- Many files to do, of which order doesn't matter for most
- Parallelism!
- `make -j N` - launches N processes to do it
- `make -j 2` often shows speed increase even on single processor systems

```
$ make
```

```
$ make -j 2
```

```
$ make -j
```

Overlapping Computation with I/O



```

#include <stdio.h>
#include "pca_utils.h"

void daxpy(int n, NType a, NType *x, NType *y, NType *z)
{
    for (int i=0; i<n; i++) {
        x[i] = (NType)i*(NType)i;
        y[i] = ((NType)i+1.)*((NType)i-1.);
    }

    for (int i=0; i<n; i++)
        z[i] += a * x[i] + y[i];
}

int main(int argc, char *argv[]) {
    int n=1e7;
    NType *x = vector(n);
    NType *y = vector(n);
    NType *z = vector(n);
    NType a = 5./3.;

    pca_time tt;
    tick(&tt);
    daxpy(n,a,x,y,z);
    tock(&tt);

    free(z);
    free(y);
    free(x);
}

```

Utilities for this course; NType is a numerical type which can be set to single or double precision

```
#include <stdio.h>
#include "pca_utils.h"

void daxpy(int n, NType a, NType *x, NType *y, NType *z)
{
    for (int i=0; i<n; i++) {
        x[i] = (NType)i*(NType)i;
        y[i] = ((NType)i+1.)*((NType)i-1.);
    }

    for (int i=0; i<n; i++)
        z[i] += a * x[i] + y[i];
}

int main(int argc, char *argv[]) {
    int n=1e7;
    NType *x = vector(n);
    NType *y = vector(n);
    NType *z = vector(n);
    NType a = 5./3.;

    pca_time tt;
    tick(&tt);
    daxpy(n, a, x, y, z);
    tock(&tt);

    free(z);
    free(y);
    free(x);
}
```

Fill arrays with
calculated values




```
#include <stdio.h>
#include "pca_utils.h"

void daxpy(int n, NType a, NType *x, NType *y, NType *z)
{
    for (int i=0; i<n; i++) {
        x[i] = (NType)i*(NType)i;
        y[i] = ((NType)i+1.)*((NType)i-1.);
    }

    for (int i=0; i<n; i++)
        z[i] += a * x[i] + y[i];
}

int main(int argc, char *argv[]) {
    int n=1e7;
    NType *x = vector(n);
    NType *y = vector(n);
    NType *z = vector(n);
    NType a = 5./3.;

    pca_time tt;
    tick(&tt);
    daxpy(n, a, x, y, z);
    tock(&tt);

    free(z);
    free(y);
    free(x);
}
```



Do calculation

```

#include <stdio.h>
#include "pca_utils.h"

void daxpy(int n, NType a, NType *x, NType *y, NType *z)
{
    for (int i=0; i<n; i++) {
        x[i] = (NType)i*(NType)i;
        y[i] = ((NType)i+1.)*((NType)i-1.);
    }

    for (int i=0; i<n; i++)
        z[i] += a * x[i] + y[i];
}

int main(int argc, char *argv[]) {
    int n=1e7;
    NType *x = vector(n);
    NType *y = vector(n);
    NType *z = vector(n);
    NType a = 5./3.;

    pca_time tt;
    tick(&tt);
    daxpy(n, a, x, y, z);
    tock(&tt);

    free(z);
    free(y);
    free(x);
}

```

Driver - do timings,
etc. (nothing needs
to be changed in
here).



OpenMPing DAXPY

- How do we OpenMP this?
- Try it (~5-10 min)

```
#include <stdio.h>
#include "pca_utils.h"

void daxpy(int n, NType a, NType *x, NType *y, NType *z)
{
    for (int i=0; i<n; i++) {
        x[i] = (NType)i*(NType)i;
        y[i] = ((NType)i+1.)*((NType)i-1.);
    }

    for (int i=0; i<n; i++)
        z[i] += a * x[i] + y[i];
}

int main(int argc, char *argv[]) {
    int n=1e7;
    NType *x = vector(n);
    NType *y = vector(n);
    NType *z = vector(n);
    NType a = 5./3.;

    pca_time tt;
    tick(&tt);
    daxpy(n,a,x,y,z);
    tock(&tt);

    free(z);
    free(y);
    free(x);
    return 0;
}
```

```

void daxpy(int n, NType a, NType *x, NType *y, NType *z)
{
#pragma omp parallel default(none) shared(n,x,y,a,z) private(i)
{
#pragma omp for
    for (int i=0; i<n; i++) {
        x[i] = (NType)i*(NType)i;
        y[i] = ((NType)i+1.)*((NType)i-1.);
    }

#pragma omp for
    for (int i=0; i<n; i++)
        z[i] += a * x[i] + y[i];
}
}

```

```

!$omp parallel default(none) private(i) shared(a,x,b,y,z)
!$omp do
    do i=1,n
        x(i) = (i)*(i)
        y(i) = (i+1.)*(i-1.)
    enddo
!$omp do
    do i=1,n
        z(i) = a*x(i) + y(i)
    enddo
!$omp end parallel

```

```
$ ./daxpy
Tock registers      2.5538e-01 seconds.

[..add OpenMP..]

$ make daxpy
gcc -std=c99 -g -DPGPLOT -I/home/ljdursi/intro-ppp//util/ -I/scinet/gpc/
Libraries/pgplot/5.2.2-gcc -fopenmp -c daxpy.c -o daxpy.o
gcc -std=c99 -g -DPGPLOT -I/home/ljdursi/intro-ppp//util/ -I/scinet/gpc/
Libraries/pgplot/5.2.2-gcc -fopenmp daxpy.o -o daxpy /home/ljdursi/intro-
ppp//util//pca_utils.o -lm

$ export OMP_NUM_THREADS=8
$ ./daxpy
Tock registers      6.9107e-02 seconds.

$ export OMP_NUM_THREADS=4
$ ./daxpy
Tock registers      1.0347e-01 seconds.

$ export OMP_NUM_THREADS=2
$ ./daxpy
Tock registers      1.8619e-01 seconds.
```

```

$ ./daxpy
Tock registers      2.5538e-01 seconds.

[..add OpenMP..]

$ make daxpy
gcc -std=c99 -g -DPGPLOT -I/home/ljdursi/intro-ppp//util/ -I/scinet/gpc/
Libraries/pgplot/5.2.2-gcc -fopenmp -c daxpy.c -o daxpy.o
gcc -std=c99 -g -DPGPLOT -I/home/ljdursi/intro-ppp//util/ -I/scinet/gpc/
Libraries/pgplot/5.2.2-gcc -fopenmp daxpy.o -o daxpy /home/ljdursi/intro-
ppp//util//pca_utils.o -lm

$ export OMP_NUM_THREADS=8
$ ./daxpy
Tock registers      6.9107e-02 seconds.      3.69x speedup, 46% efficiency

$ export OMP_NUM_THREADS=4
$ ./daxpy
Tock registers      1.0347e-01 seconds.      2.44x speedup, 61% efficiency

$ export OMP_NUM_THREADS=2
$ ./daxpy
Tock registers      1.8619e-01 seconds.      1.86x speedup, 93% efficiency

```

```

void daxpy(int n, NType a, NType *x, NType *y, NType *z)
{
#pragma omp parallel default(none) shared(n,x,y,a,z) private(i)
{
#pragma omp for
    for (int i=0; i<n; i++) {
        x[i] = (NType)i*(NType)i;
        y[i] = ((NType)i+1.)*((NType)i-1.);
    }

#pragma omp for
    for (int i=0; i<n; i++)
        z[i] += a * x[i] + y[i];
}
}

```

Why is this safe?
Everyone's modifying x,y,z

```

!$omp parallel default(none) private(i) shared(a,x,b,y,z)
!$omp do
    do i=1,n
        x(i) = (i)*(i)
        y(i) = (i+1.)*(i-1.)
    enddo
!$omp do
    do i=1,n
        z(i) = a*x(i) + y(i)
    enddo
!$omp end parallel

```

Dot Product

- Dot product of two vectors
- Implement this, first serially, then with OpenMP
- `ndot.c` or `ndot.f90`
- `make ndot` or `make ndotf`
- Tells time, answer, correct answer.

$$\begin{aligned}n &= \hat{x} \cdot \hat{y} \\ &= \sum_i x_i y_i\end{aligned}$$

```
$ ./ndot
Dot product is      3.3333e+20
(vs  3.3333e+20) for n=10000000.
Took    5.3578e-02 seconds.
```



```

...main program...
print *, 'Dot product is ', res, '(vs ', ans, ') for n = ', n, '.
Took ', time, 'sec.'

deallocate(x,y)

contains

double precision function calc_ndot(n, x, y)
  implicit none
  integer, intent(in) :: n
  double precision, dimension(n) :: x
  double precision, dimension(n) :: y
  double precision :: ndot
  integer :: i

  ndot = 0.
  do i=1,n
    ndot = ndot + x(i)*y(i)
  enddo
  calc_ndot = ndot
end function calc_ndot

```

How to OpenMP this?

```
double precision function calc_ndot(n, x, y)
  implicit none
  integer, intent(in) :: n
  double precision, dimension(n) :: x
  double precision, dimension(n) :: y
  double precision :: ndot
  integer :: i
!$omp parallel default(none) shared(ndot,x,y,n) private(i)
  ndot = 0.
  do i=1,n
    ndot = ndot + x(i)*y(i)
  enddo
!$omp end parallel
  calc_ndot = ndot
end function calc_ndot
```

fomp_ndot_race.f90
omp_ndot_race.c

fomp_ndot_race.f90
omp_ndot_race.c

```
double precision function calc_ndot(n, x, y)
  implicit none
  integer, intent(in) :: n
  double precision, dimension(n) :: x
  double precision, dimension(n) :: y
  double precision :: ndot
  integer :: i
!$omp parallel default(none) shared(ndot,x,y,n) private(i)
  ndot = 0.
  do i=1,n
    ndot = ndot + x(i)*y(i)
  enddo
!$omp end parallel
  calc_ndot = ndot
end function calc_ndot
```

```
$ ./ndotf
Dot product is 3.333332833333717098E+020 (vs 3.33333363469873840E+020 )
for n = 10000000 . Took 5.00000007E-02 sec.
$ export OMP_NUM_THREADS=8
$ ./fomp_ndot_race
Dot product is 6.06898061003712922E+019 (vs 3.33333363469873840E+020 )
for n = 10000000 . Took 0.16300000 sec.
```

Wrong answer - and slower!

Race Condition - why it's wrong

$ndot = 0.$

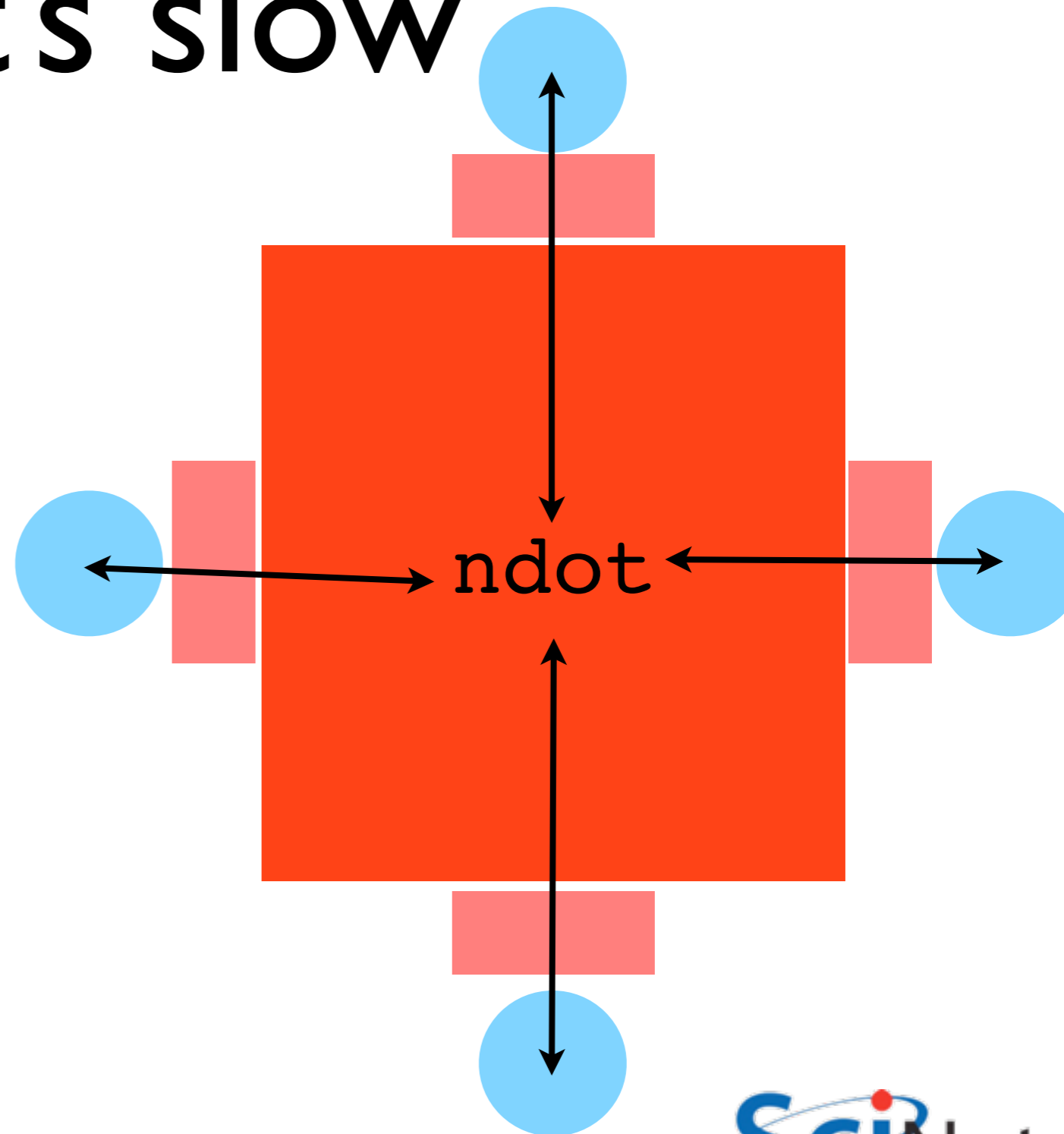
- Classic parallel bug
- Multiple writers to some shared resource
- Can be very subtle, and only appear intermittently
- Your program can have a bug but not display any symptoms for small runs!
- Primarily a problem with shared memory

Thread 0: add 1	Thread 1: add 2
read $ndot (=0)$ into register	
$reg = reg + 1$	read $ndot (=0)$ into register
store $reg (=1)$ into $ndot$	$reg = reg + 2$
	store $reg (=2)$ into $ndot$

$ndot = 2$

Memory contention - why it's slow

- Multiple cores repeatedly trying to read, access, store same variable in memory
- Not (such) a problem for constants (read only); but a big problem for writing.
- Sections of arrays -- better.



OpenMP critical construct

- Defines a “critical region”
- Only one thread can be operating within this region at a time
- Keeps modifications to shared resources safe
- `#pragma omp critical` or `!$omp critical / !$omp end critical`

```
NType ndot_critical(int n, NType *x, NT
{
    NType tot=0;
#pragma omp parallel for shared(x,y,n,t
    for (int i=0; i<n; i++)
#pragma omp critical
        tot += x[i] * y[i];
    return tot;
}
```

```
        ndot = 0.
!$omp parallel default(none) shared(ndo
!$omp do
    do i=1,n
!$omp critical
        ndot = ndot + x(i)*y(i)
!$omp end critical
    enddo
!$omp end parallel
    calc_ndot = ndot
end function calc_ndot
```

OpenMP atomic construct

- Most hardware has support for atomic (indivisible - eg, can't get interrupted) instructions
- Small subset, but load/add/store usually one
- Not as general as critical
- Much lower overhead
- Better -- 'only' 18x slower than serial! Still some overhead, still memory contention.

```
$ ./ndot
Dot product is      3.3333e+20
(vs      3.3333e+20) for n=10000000.
Took    5.3570e-02 seconds.

$ ./omp_ndot_atomic
Dot product is      3.3333e+20
(vs      3.3333e+20) for n=10000000.
Took    9.7981e-01 seconds.
```

How should we fix this?

$$\begin{aligned}n &= \hat{x} \cdot \hat{y} \\ &= \sum_i x_i y_i\end{aligned}$$

How should we fix this?

- Local sums
- Each processor sums its local value ($10^7/P$ additions)
- And *then* sums to *ntot* (only P additions) with critical, or atomic..
- Try this (5-10 min)
- cp one of the `omp_ndot.c`'s or `fomp_ndot.c`'s to `omp_ndot_local.c` (or `fomp_ndot_local.f90`)

$$\begin{aligned}n &= \hat{x} \cdot \hat{y} \\ &= \sum_i x_i y_i \\ &= \sum_p \left(\sum_i x_i y_i \right)\end{aligned}$$

Local variables:

```
#pragma omp parallel shared(x,y,n,tot)
private(mytot)
{
    mytot = 0;
    #pragma omp for
    for (int i=0; i<n; i++)
        mytot += x[i] * y[i];

    #pragma omp atomic
    tot += mytot;
}
```

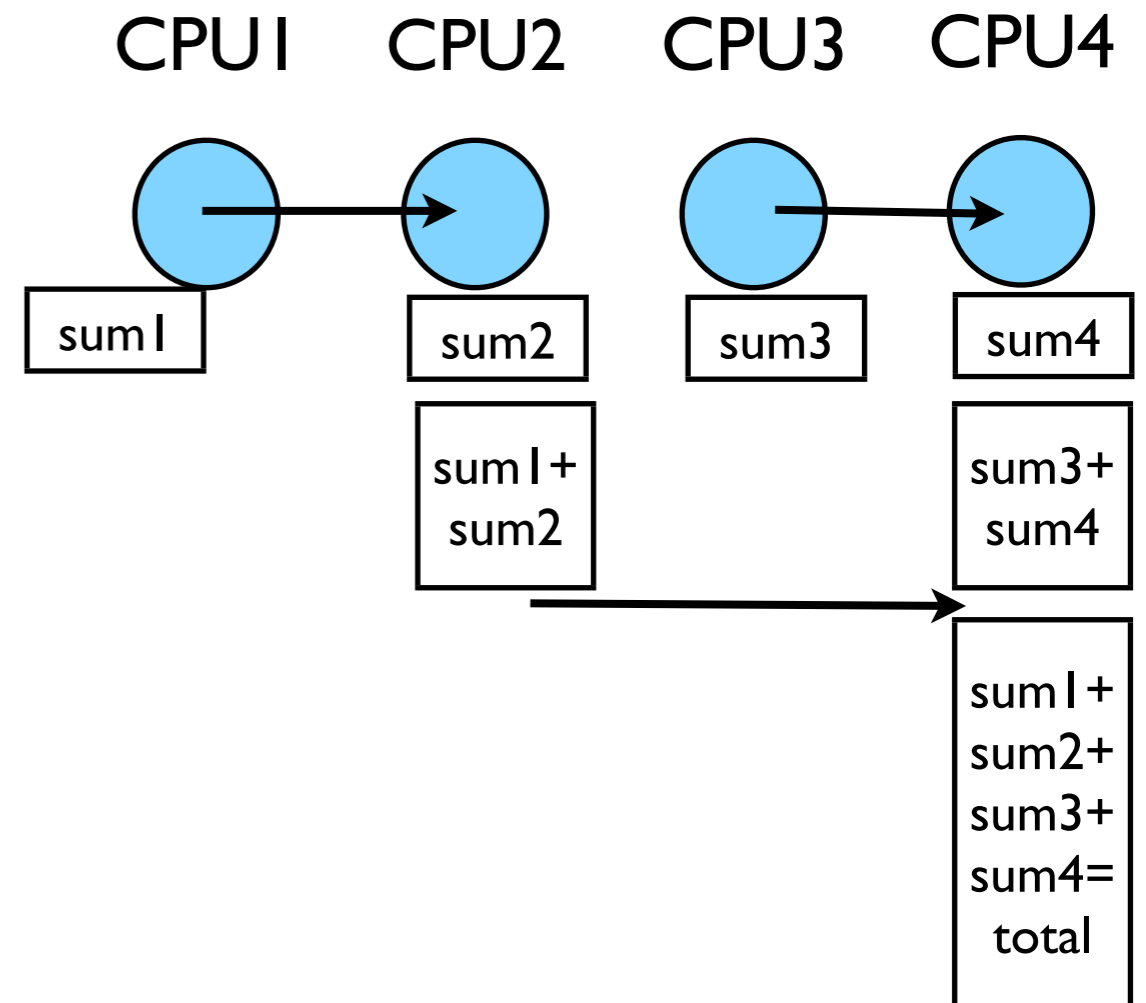
```
ndot = 0.
!$omp parallel default(none)
    shared(ndot,n,x,y) private(i,mytot)
    mytot = 0.
!$omp do
    do i=1,n
        mytot = mytot + x(i)*y(i)
    enddo
!$omp atomic
    ndot = ndot + mytot
!$omp end parallel
calc_ndot = ndot
```

```
$ ./ndot
Dot product is      3.3333e+20
(vs      3.3333e+20) for n=10000000.
Took    5.3570e-02 seconds.
```

```
$ export OMP_NUM_THREADS=8
$ ./omp_ndot_local
Dot product is      3.3333e+20
(vs      3.3333e+20) for n=10000000.
Took    1.8334e-02 seconds.
```

OpenMP Reduction Operations

- This is such a common operation, there is something built into OpenMP to handle it
- “reduction” variables - like shared or private
- Can support several types of operations - +, *...
- `omp_ndot_reduction.c`,
`fomp_ndot_reduction.f90`



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

OpenMP Reduction Operations

```
NType ndot_atomic(int n, NType *x, NType *y)
{
    NType tot=0;
    #pragma omp parallel shared(x,y,n), reduction(+:tot)
    {
        #pragma omp for
        for (int i=0; i<n; i++)
            tot += x[i] * y[i];
    }
    return tot;
}
```

OpenMP Reduction Operations

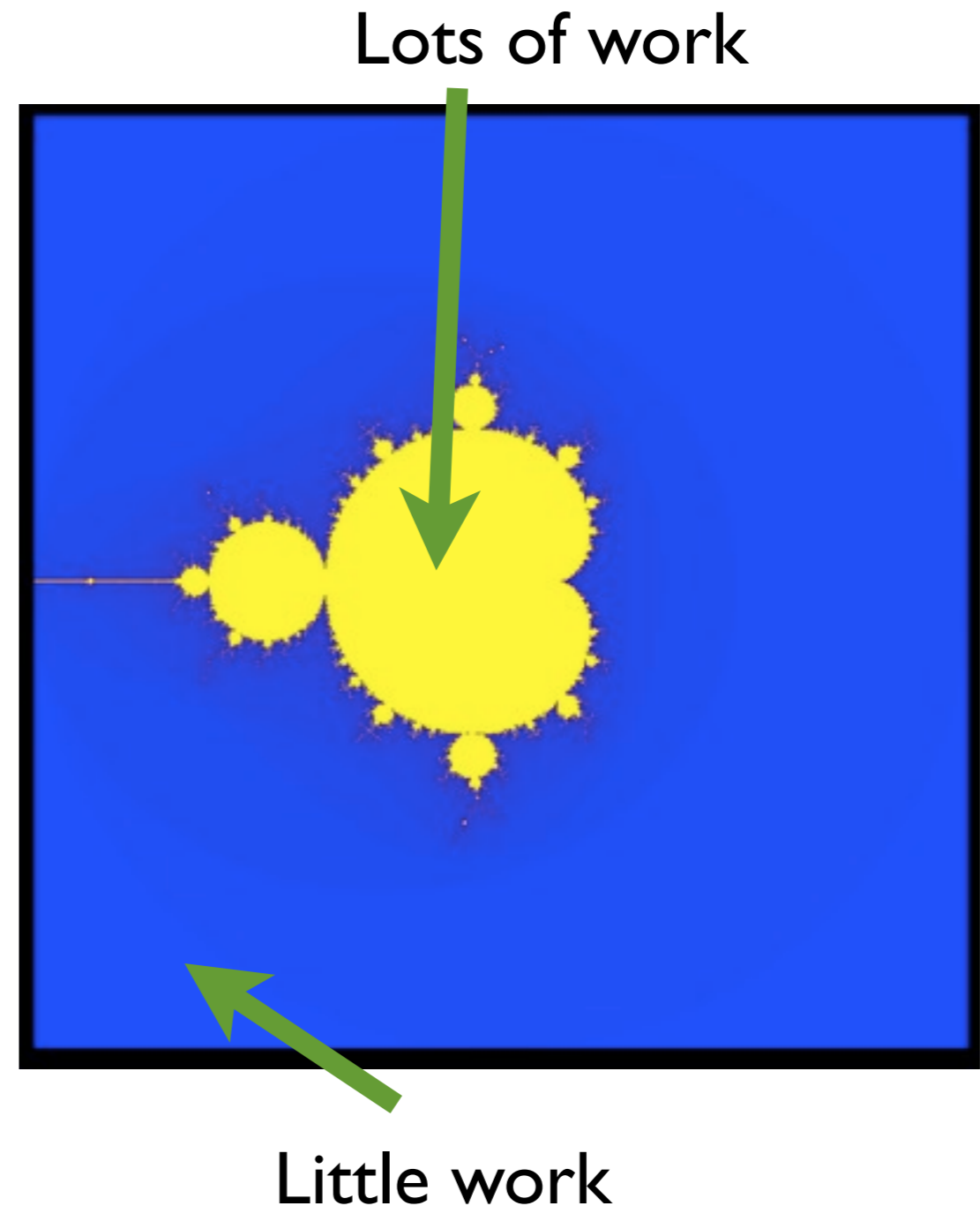
```
double precision function calc_ndot(n, x, y)
implicit none
integer, intent(in) :: n
double precision, dimension(n) :: x
double precision, dimension(n) :: y
double precision :: ndot
integer :: i

ndot = 0.
!$omp parallel default(none) shared(n,x,y) reduction(+:ndot) private(i)
!$omp do
    do i=1,n
        ndot = ndot + x(i)*y(i)
    enddo
!$omp end parallel
calc_ndot = ndot

end function calc_ndot
```

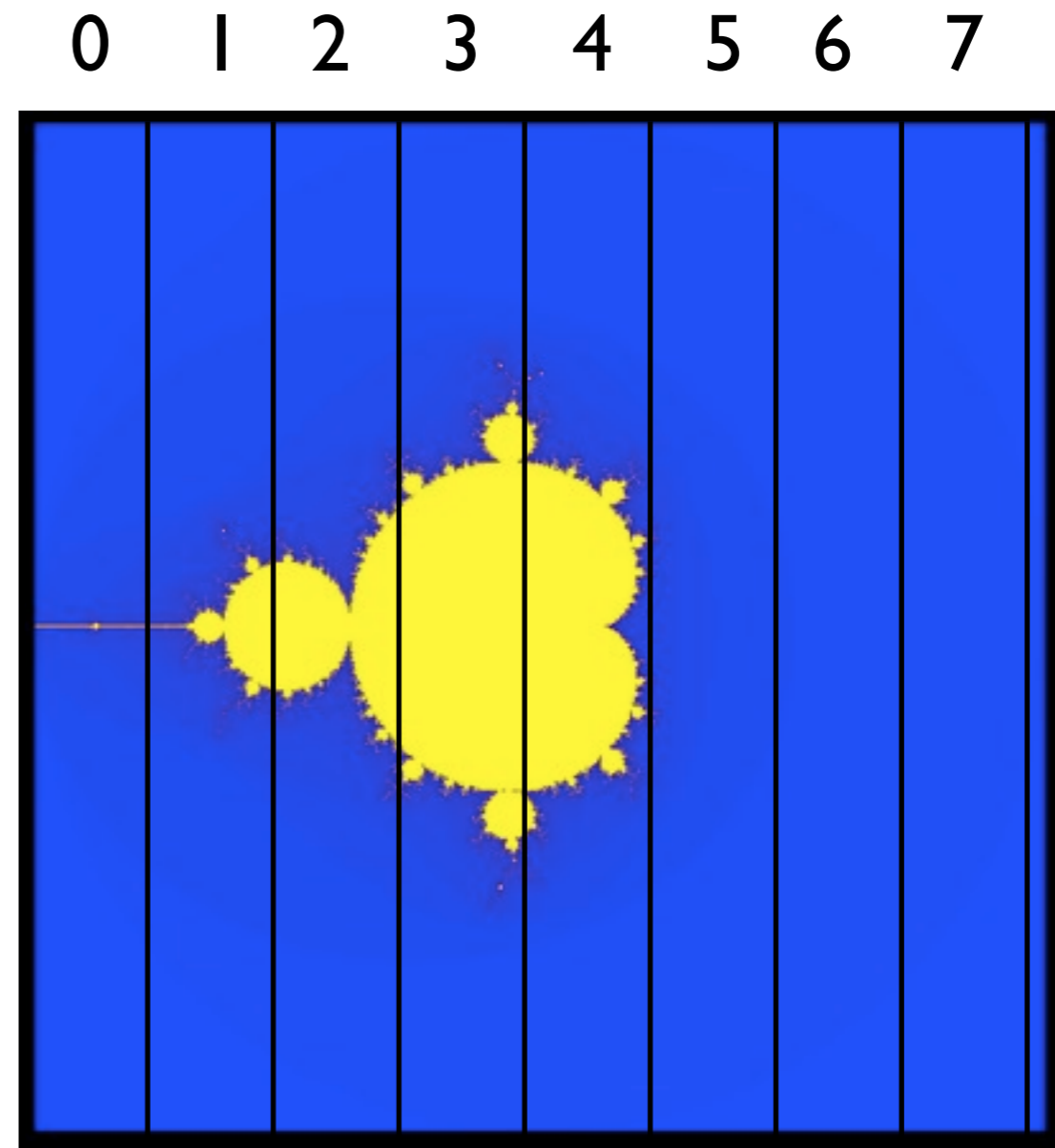
Load-Balancing

- So far, every iteration of the loop has had the same amount of work:
- Not always the case
- `make mandel; ./mandel`
- Plots a function at every pixel with different amount of work - in fact, amount of work is basically the plotted color.



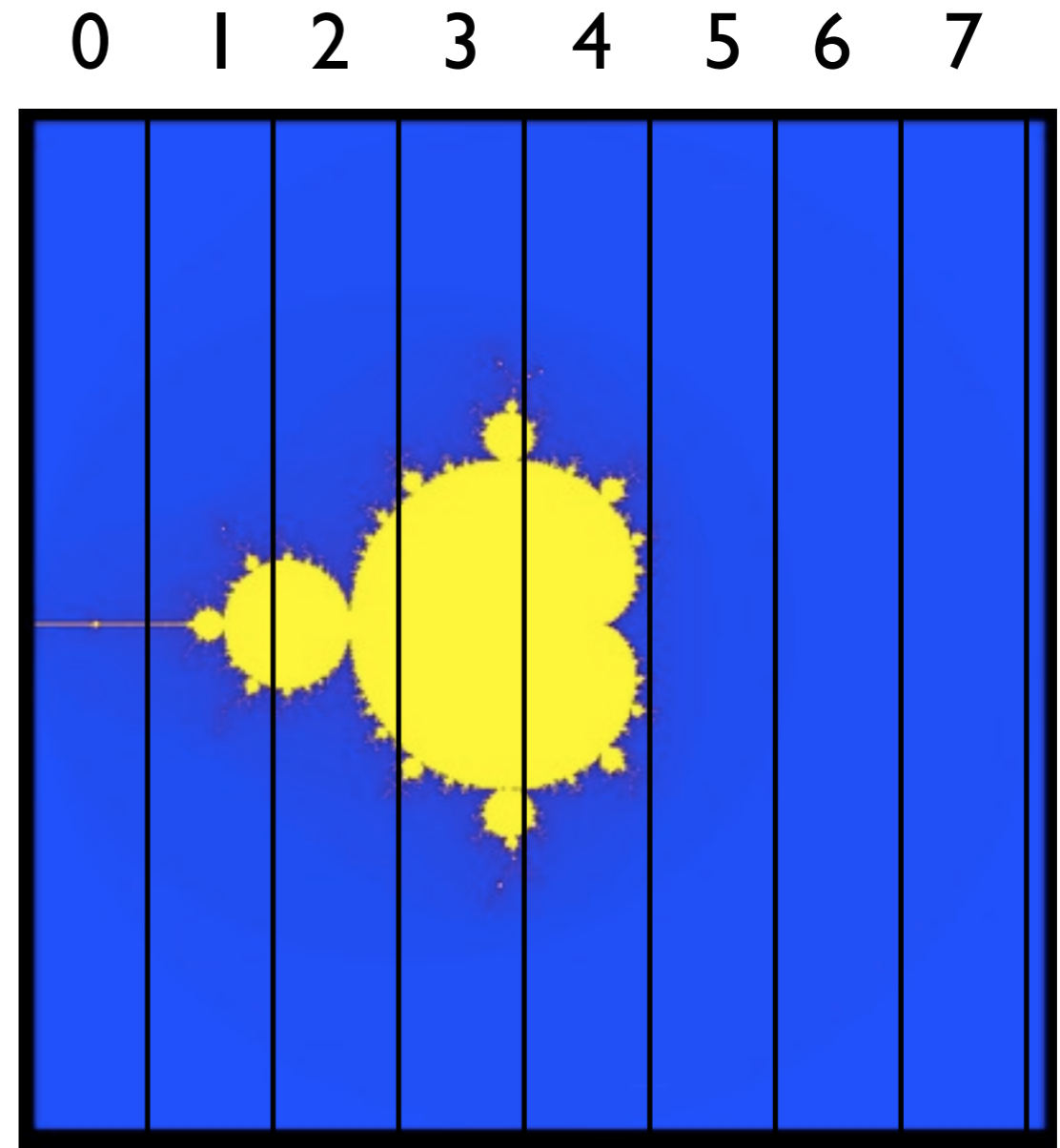
Load-Balancing

- Default work sharing breaks N iterations into $\sim N/n$ threads contiguous chunks and assigns them to threads
- But now threads 7, 6, 5 will be done and sitting idle while threads 3,4 work alone...
- Inefficient use of resources



Load-Balancing

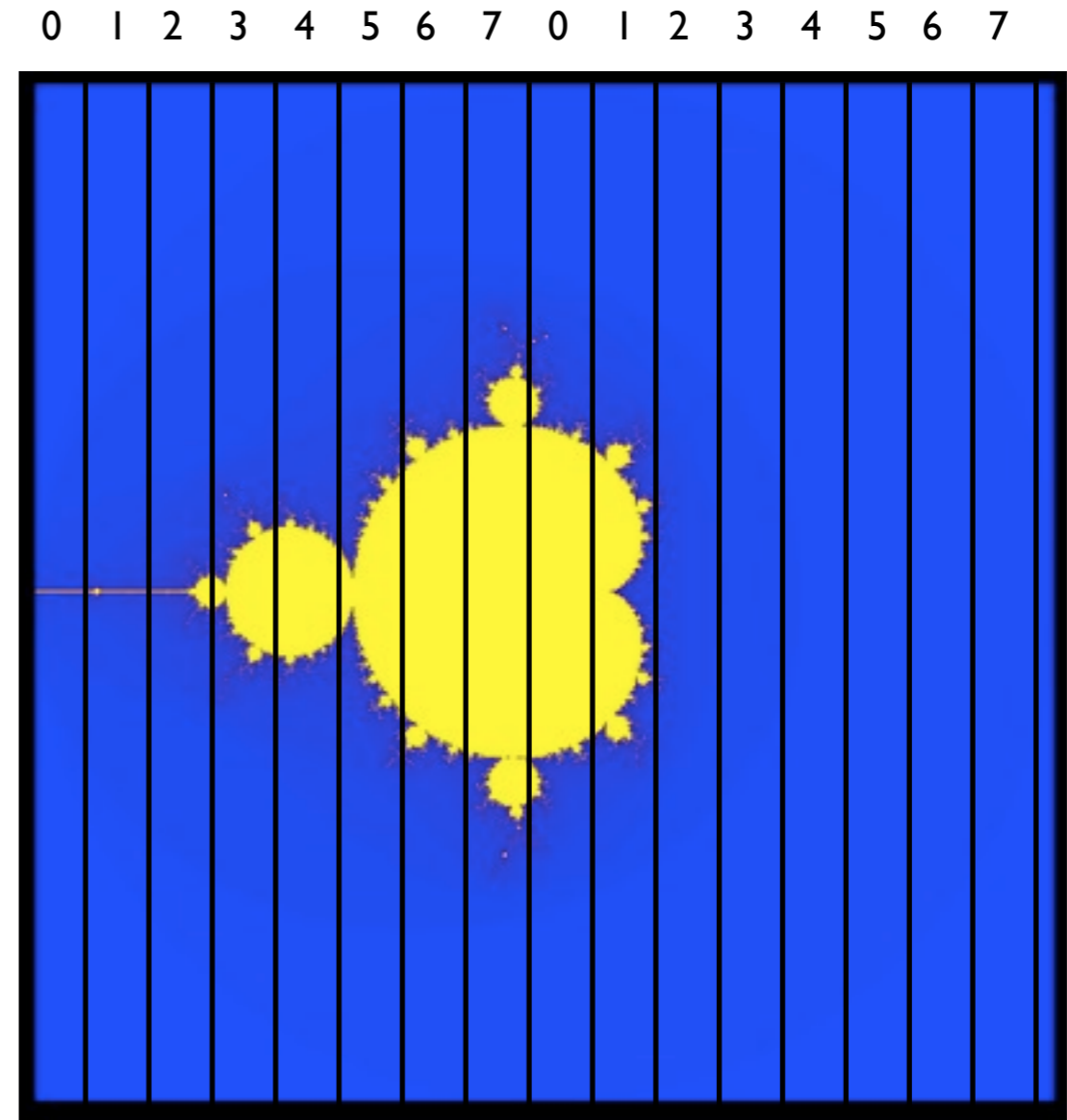
Serial	0.63s
Nthreads=8	0.29s
Speedup	2.2x
Efficiency	27%



800x800 pix; N/nthreads ~ 100x800

Load-Balancing

- Can change the 'chunk size' from $\sim N/n$ threads to arbitrary number
- In this case, more columns - work distributed a bit better
- Now, for instance, chunk size ~ 50 , and thread 7 gets both a big work chunk and a little work chunk.



Load-Balancing

0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7

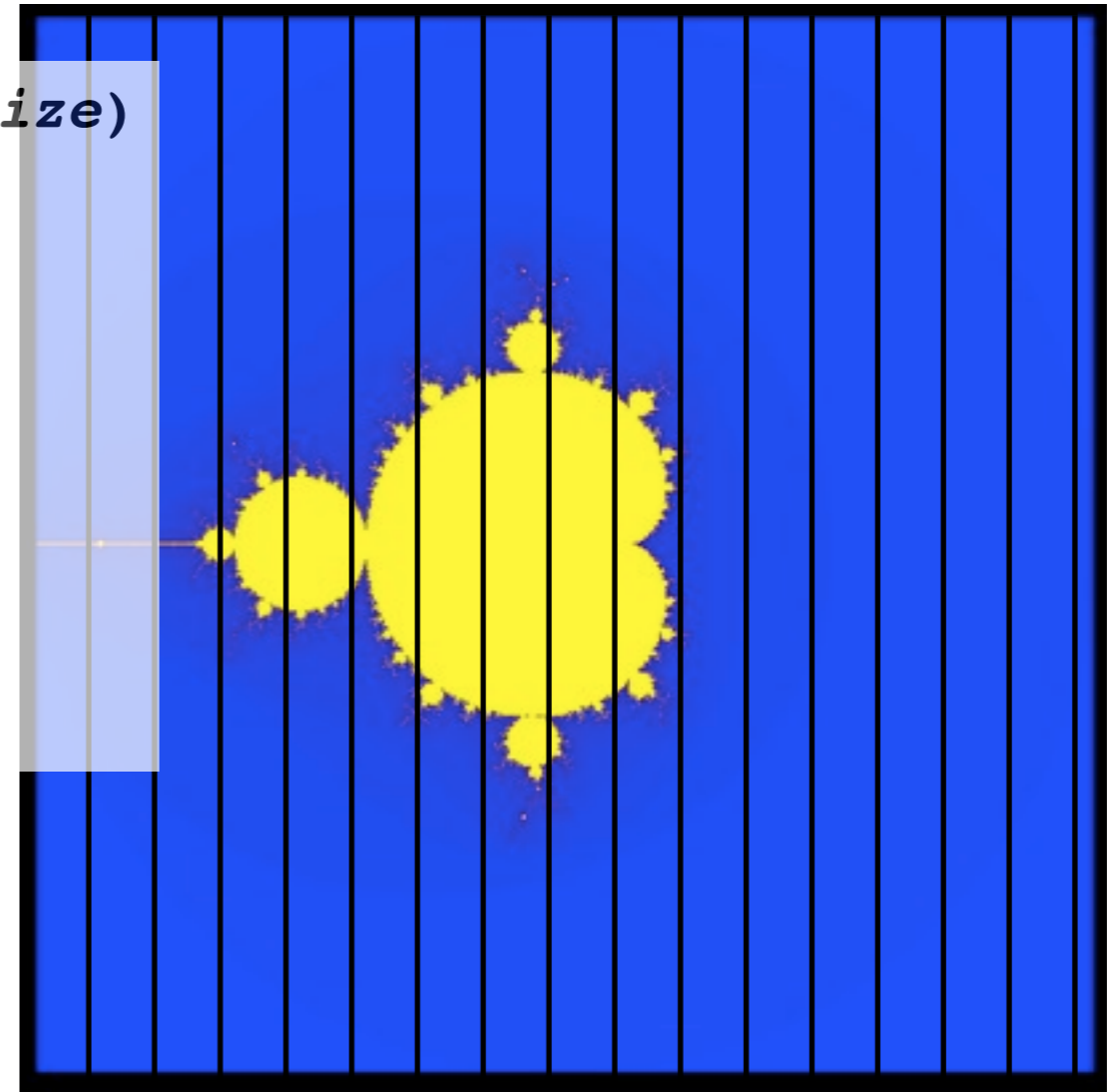
```
#pragma omp for schedule(static, chunksize)
```

or

```
!$omp do schedule(static, chunksize)
```

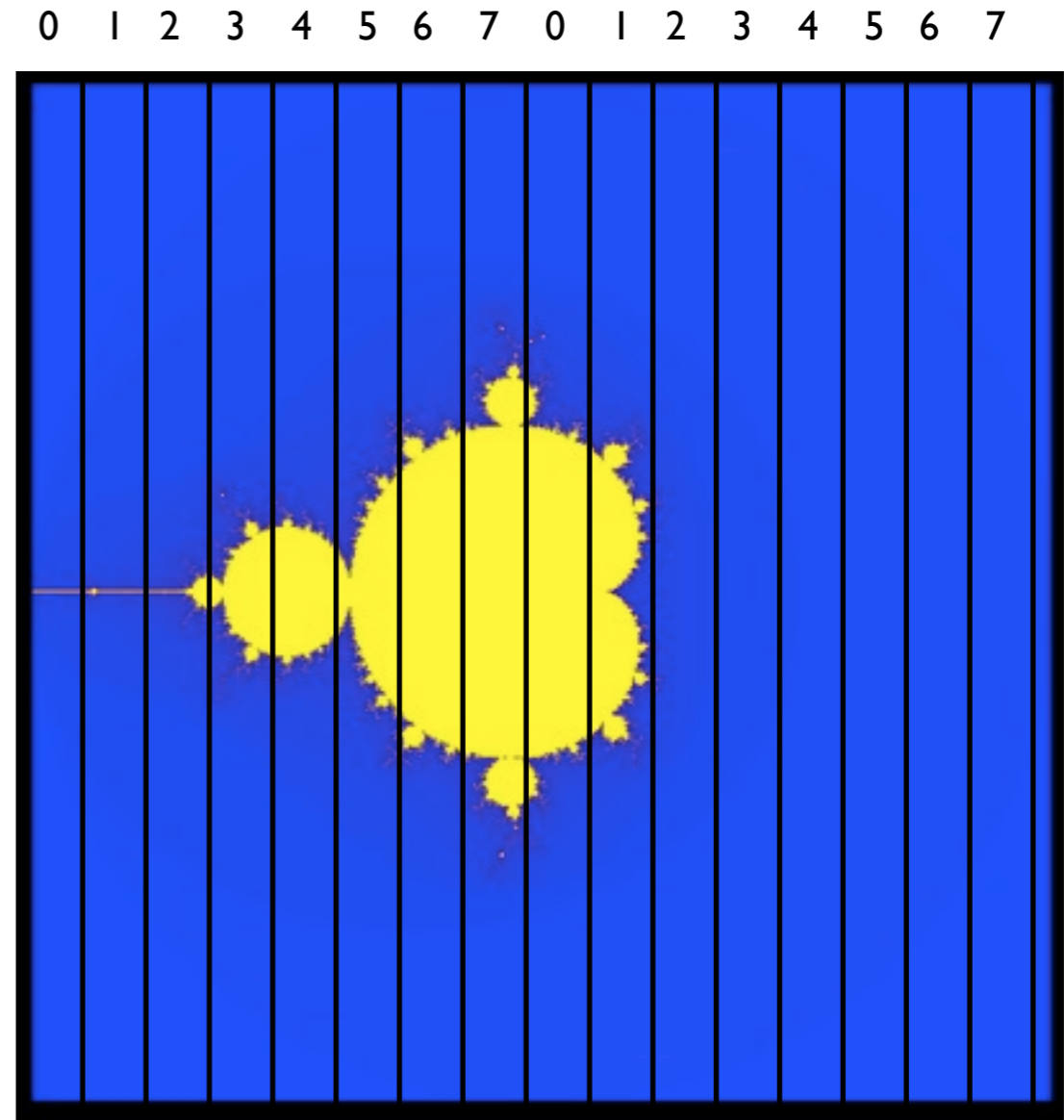
Here, *chunksize* = 50.

Static scheduling



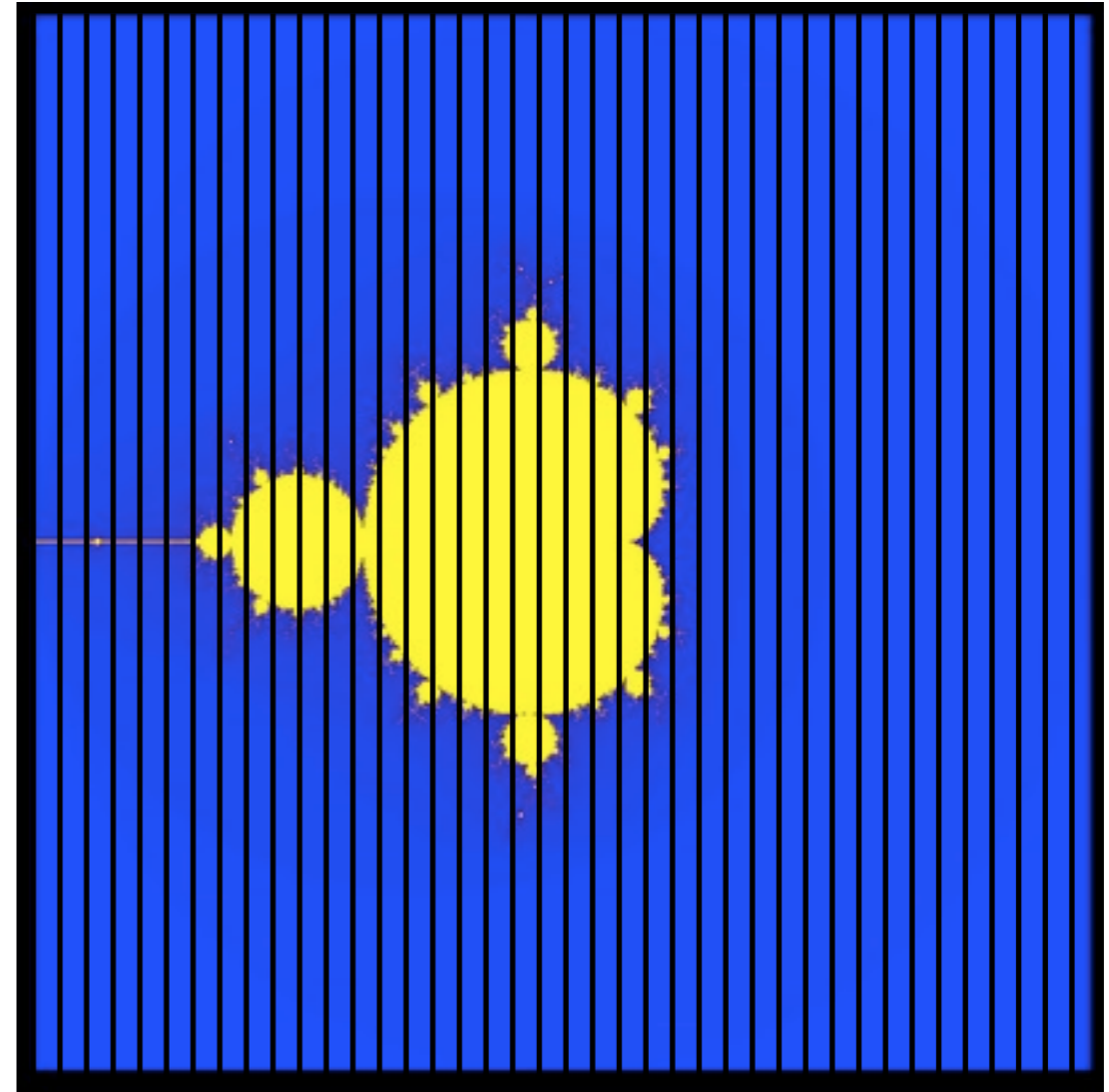
schedule(static,50)

Serial	0.63s
Nthreads=8	0.15s
Speedup	4.2x
Efficiency	52%



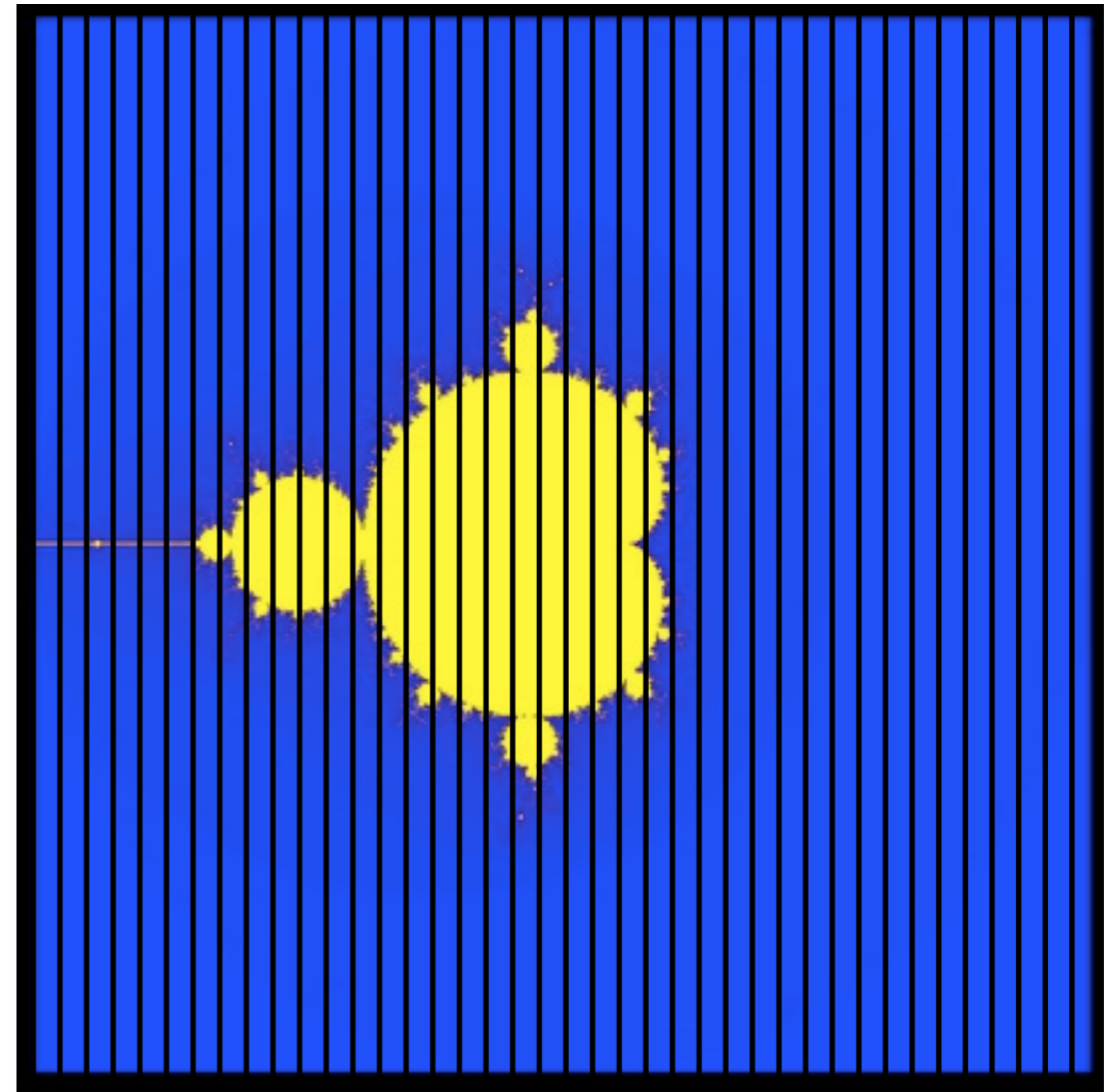
schedule(dynamic)

- Still another choice is to break it up into many pieces and hand them to threads when they are ready
- dynamic scheduling
- Has increased overhead, but can do a very good job
- can also choose chunksize for dynamic



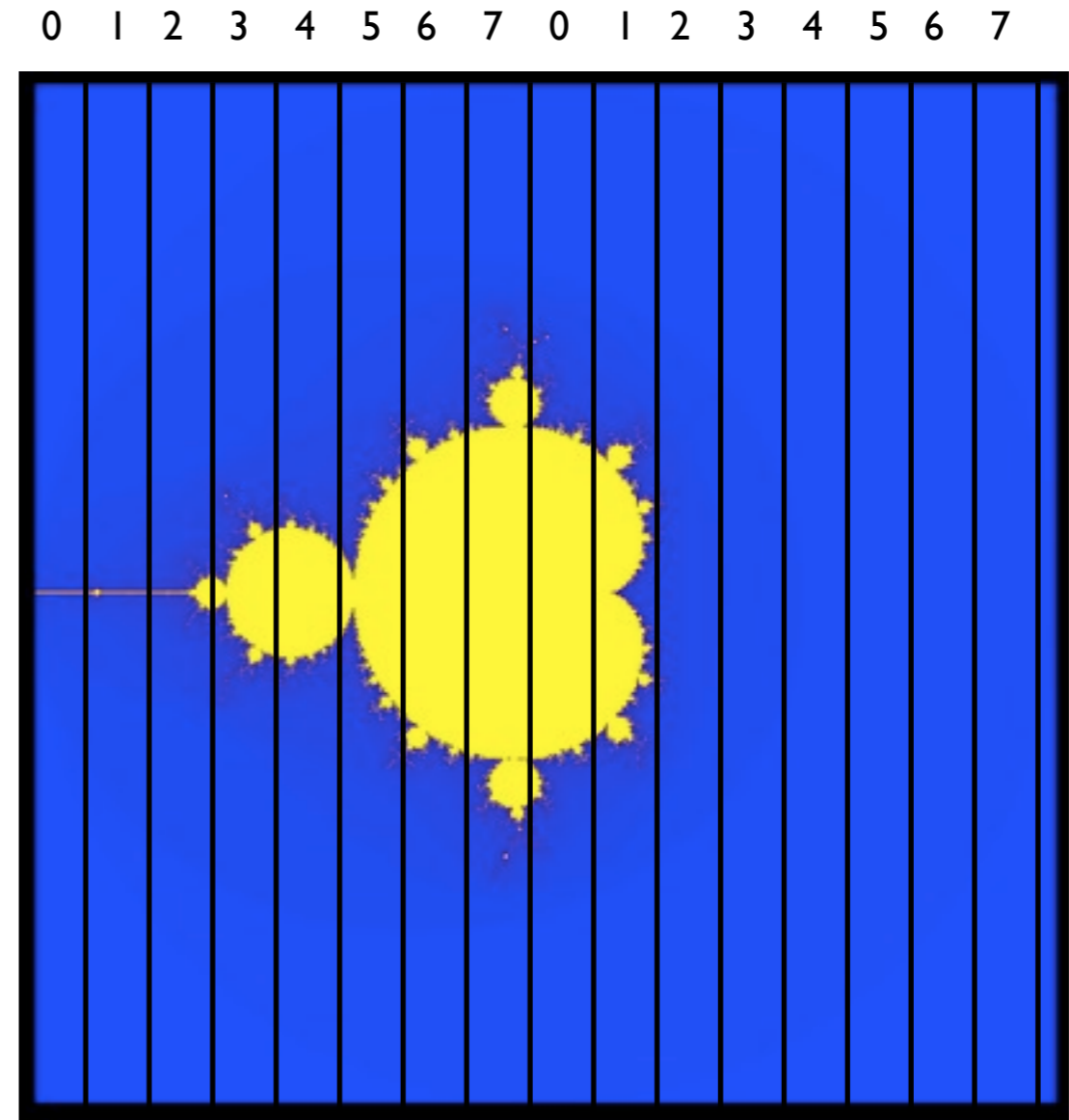
schedule(dynamic)

Serial	0.63s
Nthreads=8	0.10
Speedup	6.3x
Efficiency	79%



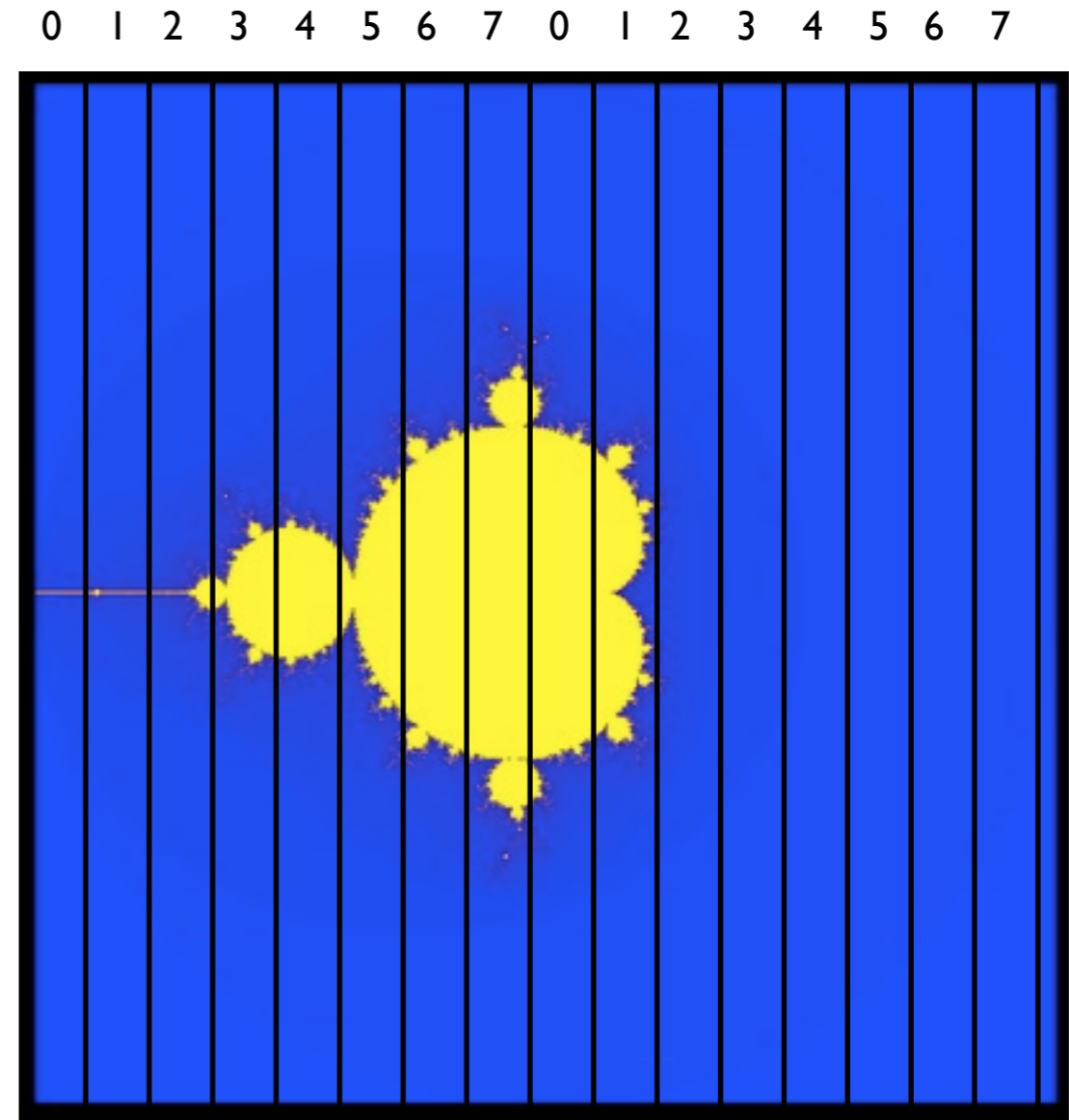
Tuning

- `schedule(static)` (default) or `schedule(dynamic)` are good starting places
- To get best performance in badly imbalanced problems, may have to play with chunk sizes - will depend on your problem, and hardware.



Tuning

(static,4)	(dynamic,16)
0.084s	0.099s
7.6x	6.4x
95%	80%



Two-level loops

- In scientific code, we usually have nested loops where all the work is.
- Almost without exception, want the loop on the *outside-most* loop. Why?

```
#pragma omp for schedule(static,4)
for (int i=0;i<npix;i++)
  for (int j=0;j<npix;j++) {
    double x=((double)i)/((double)npix);
    double y=((double)j)/((double)npix);
    double complex a=x+I*y;
    mymap[i][j]=how_many_iter_real(a);
  }
```

mandel.c

Summary

- omp parallel
- omp single
- shared/private/reduction variables
- omp atomic, omp critical
- omp for