

Parallel Computing

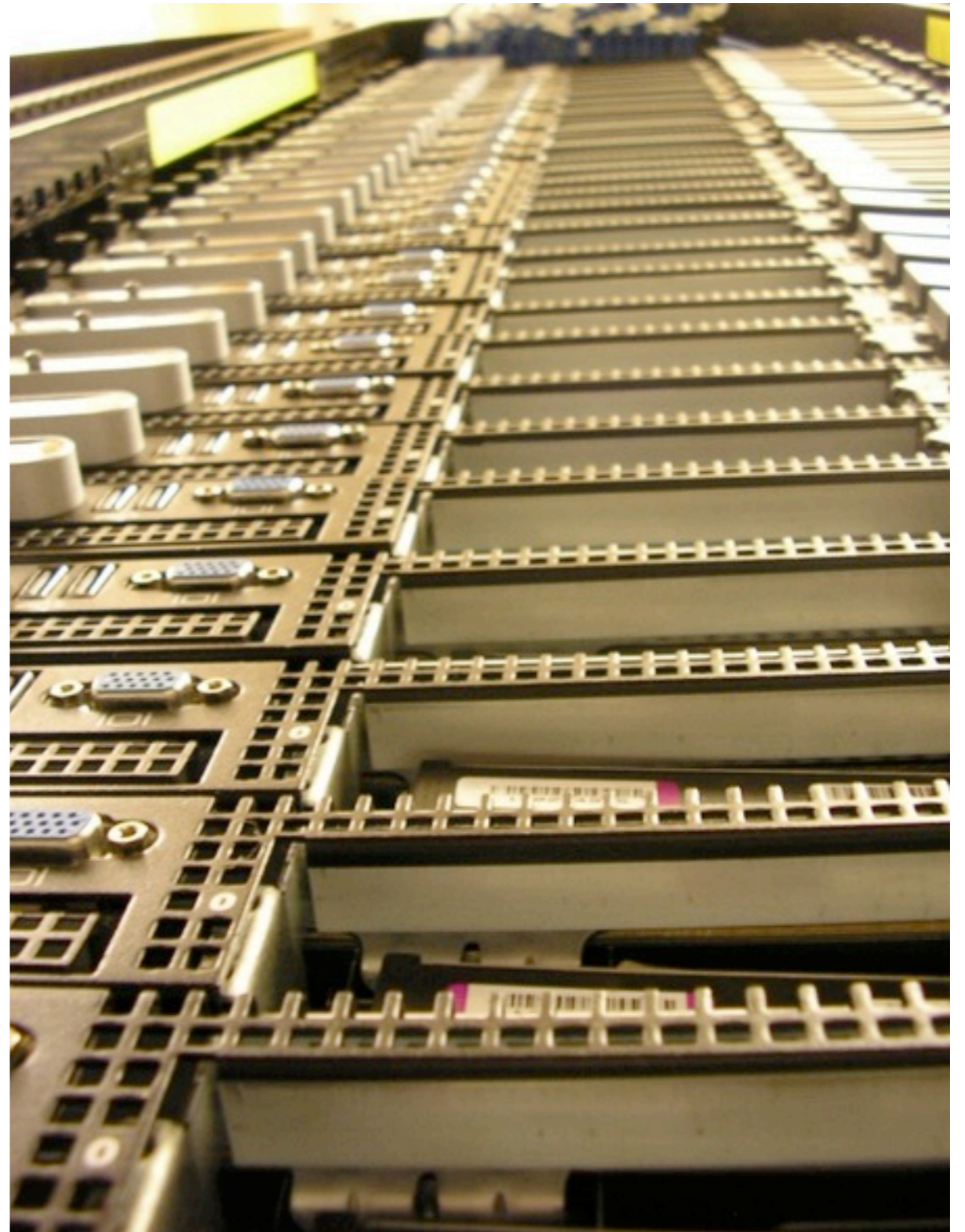
I: Concurrency, Amdahl's Law, and Locality

Why Parallel Computing?

Faster:

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

So use more computers!

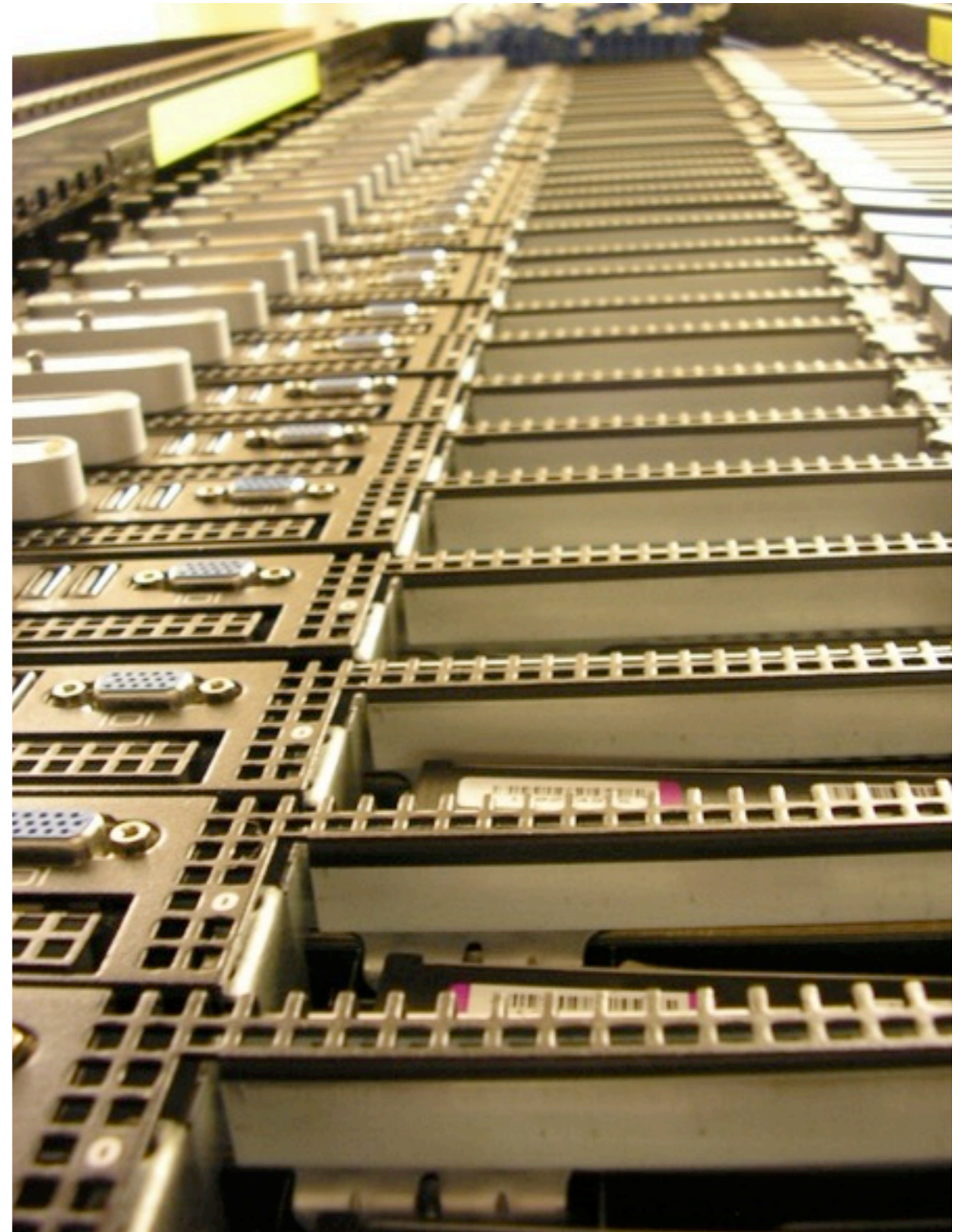


Why Parallel Computing?

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!

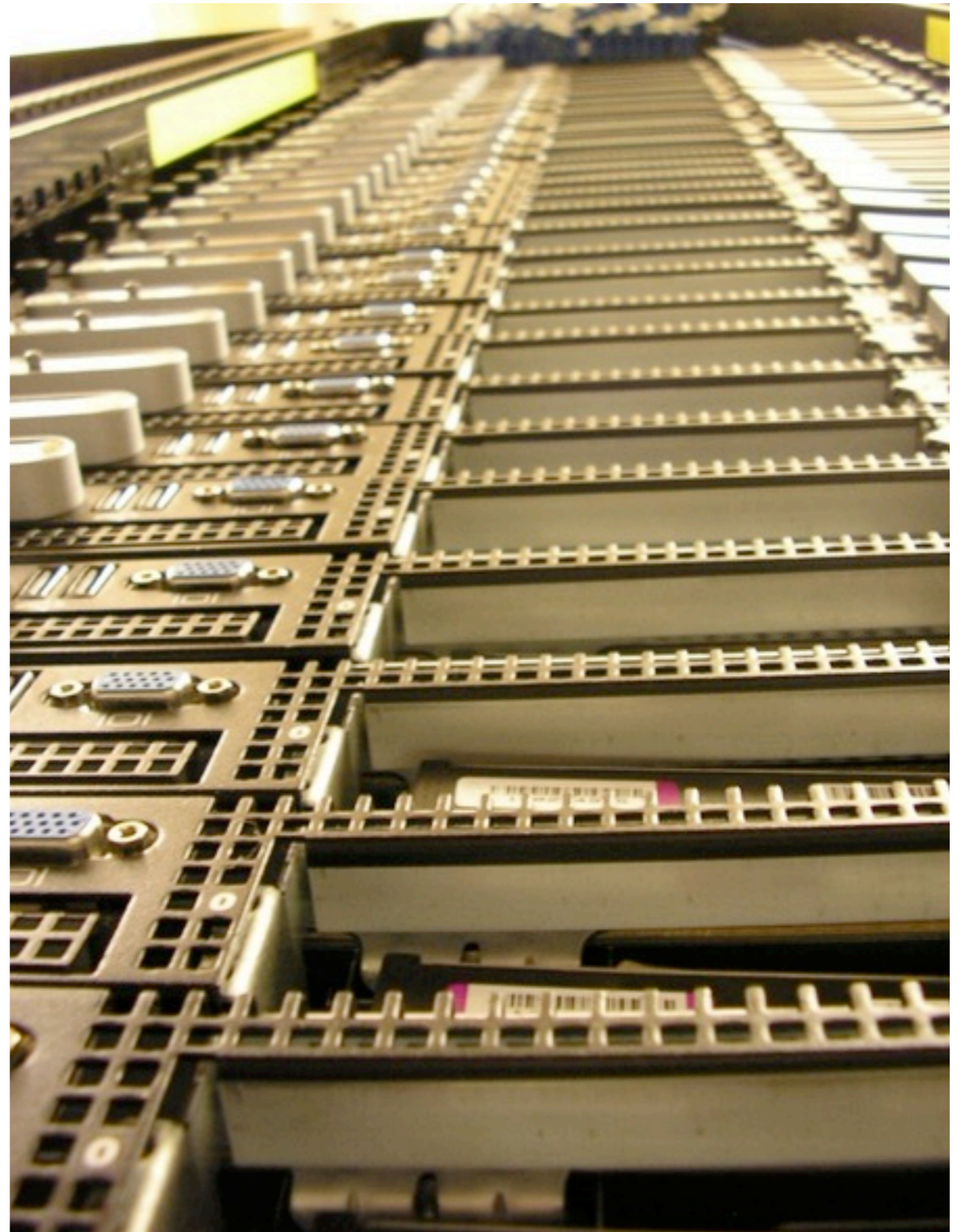


Why Parallel Computing?

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!



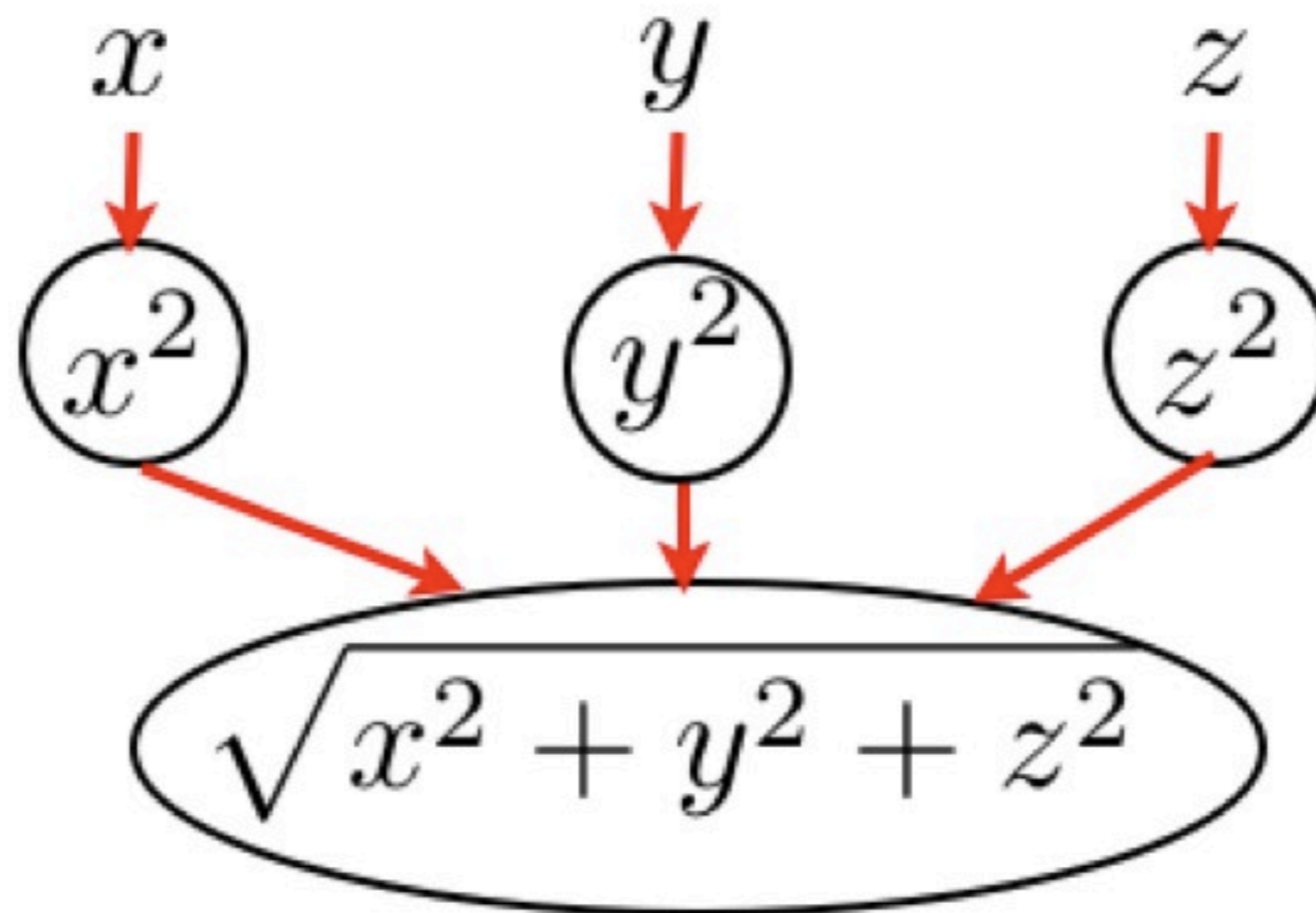
Concurrency

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



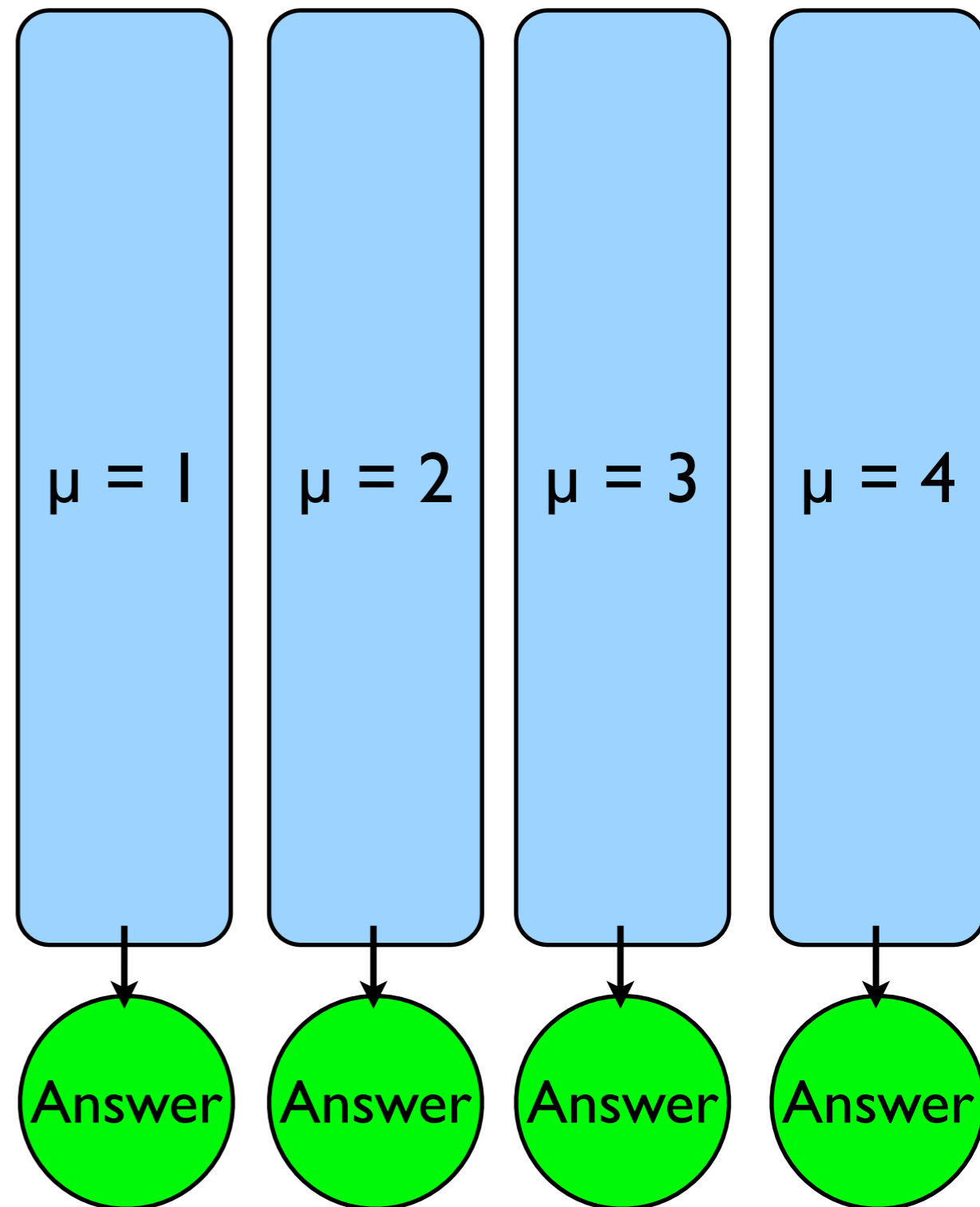
<http://flickr.com/photos/splorp/>

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’**

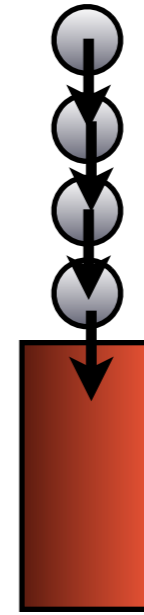


Throughput = Tasks/Time

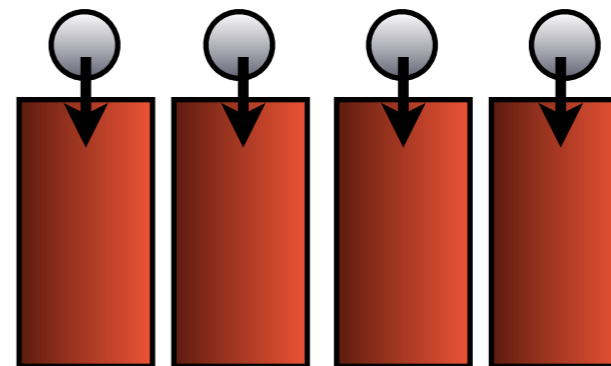
How long it takes to process the
N tasks you want done

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

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



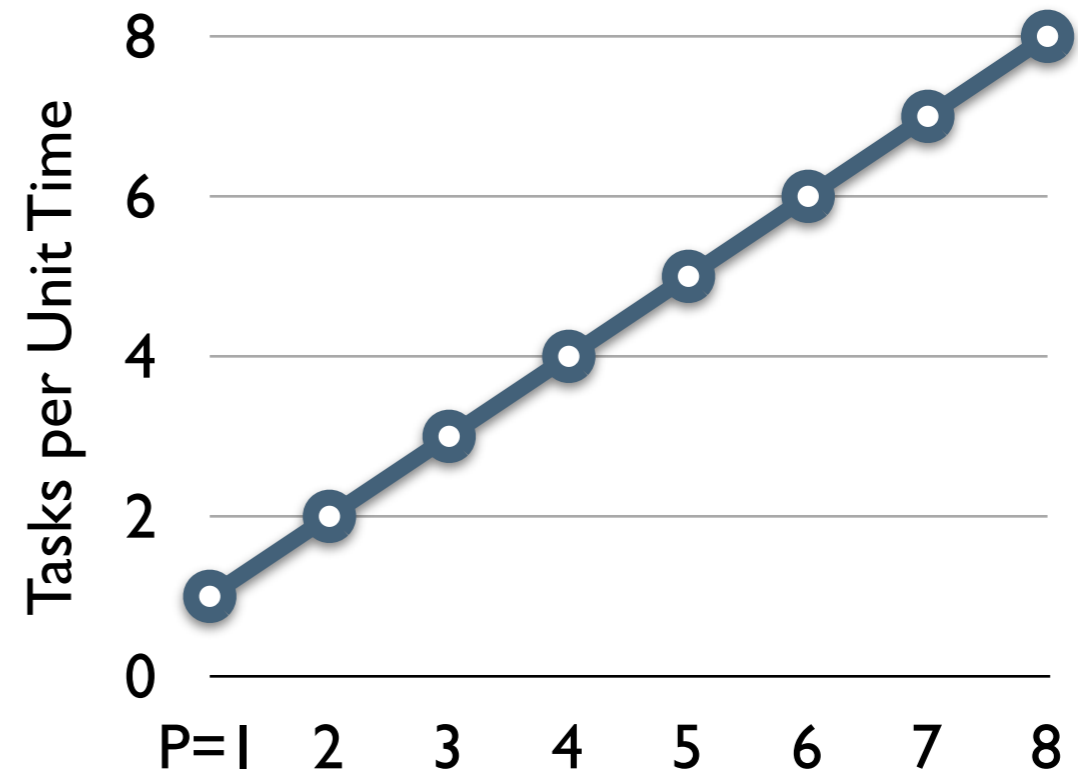
VS



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'



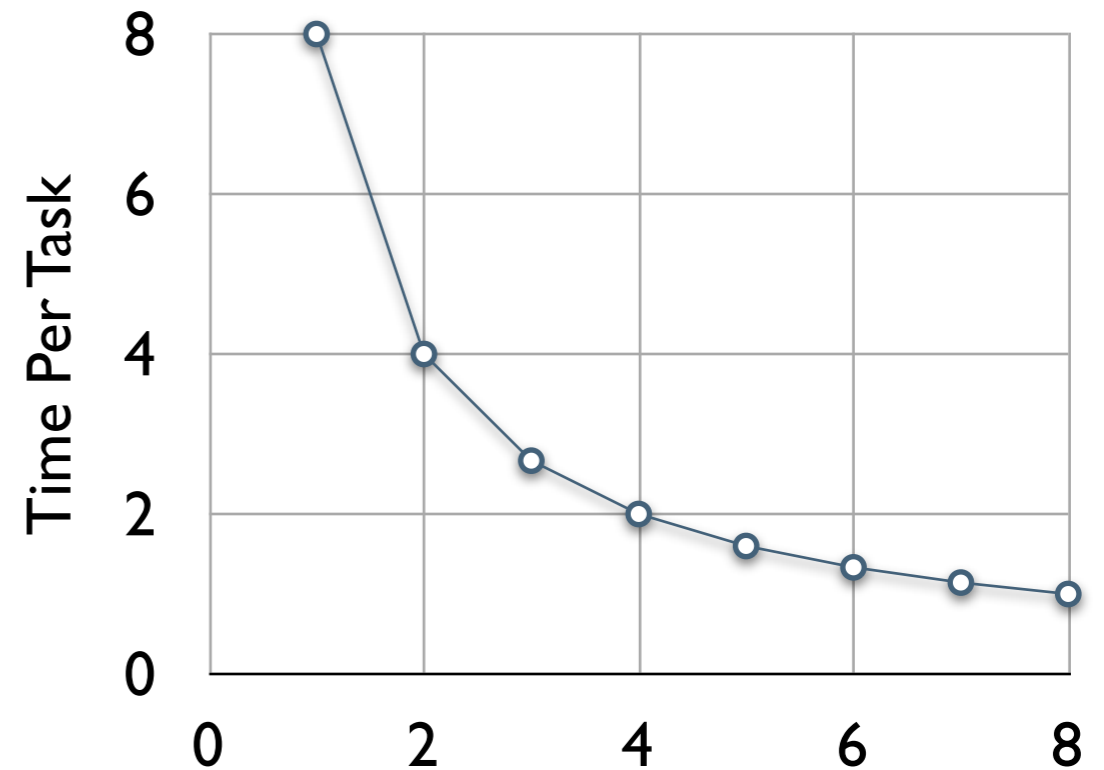
Scaling with P

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



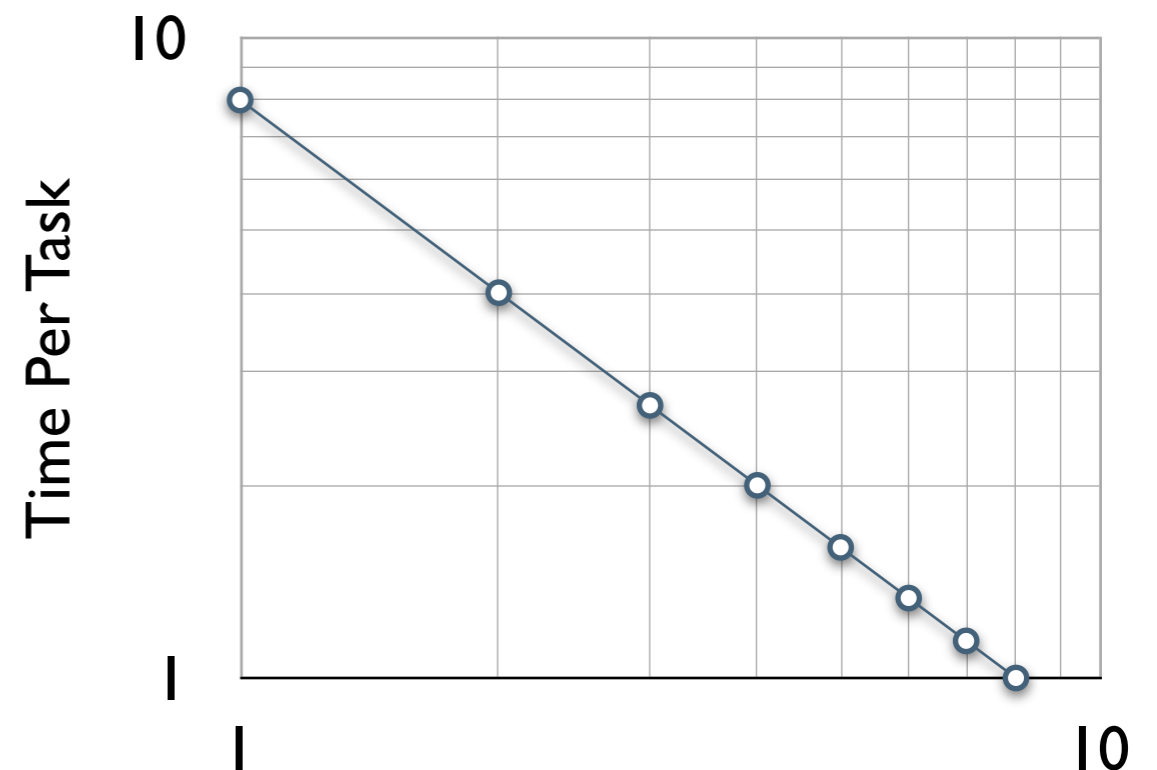
Scaling with P

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

More usual (and more important!)

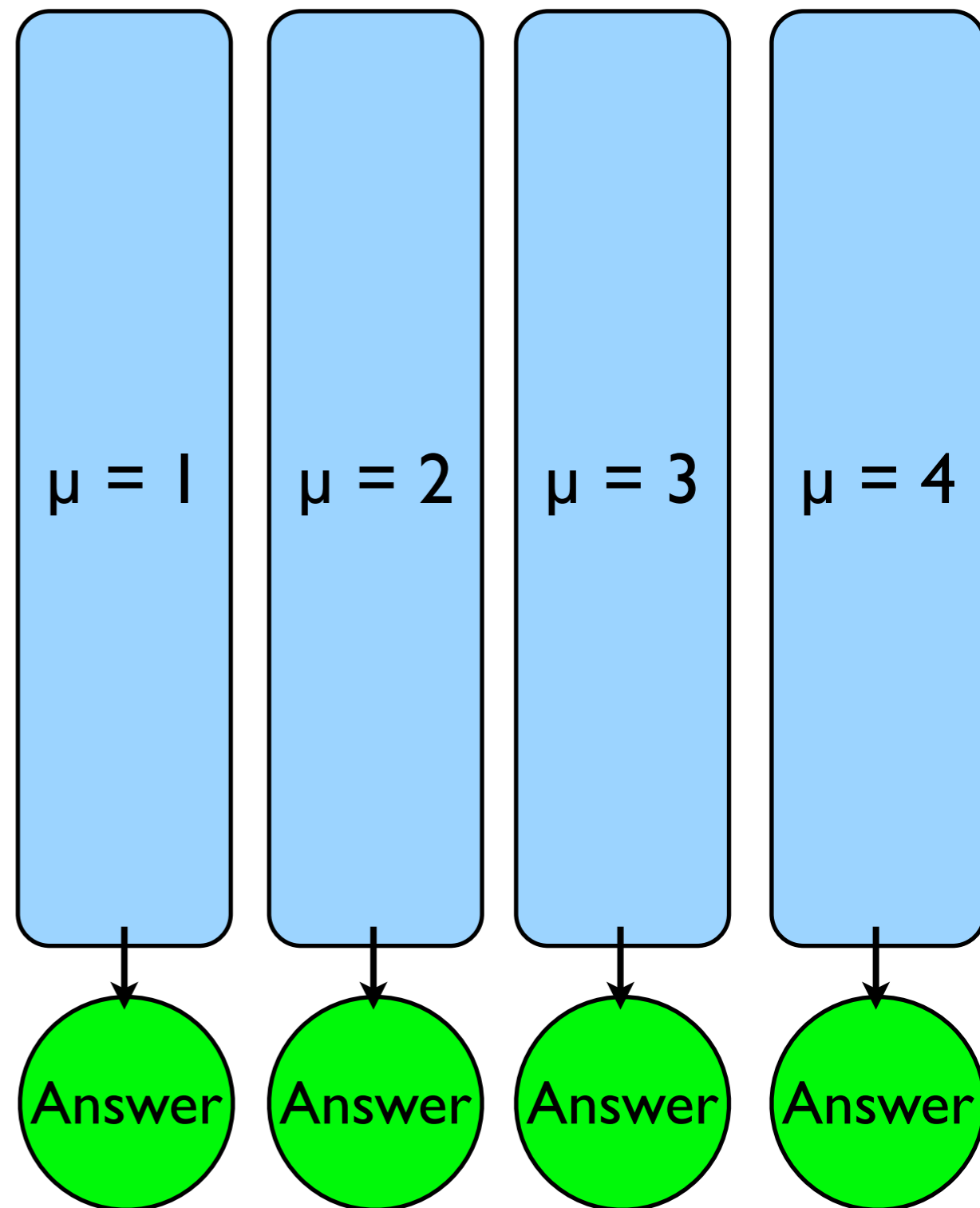
Perfect scaling: time to completion $\sim 1/P$

P processors - P times faster



Parameter Study: 'Embarrassingly Parallel'

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

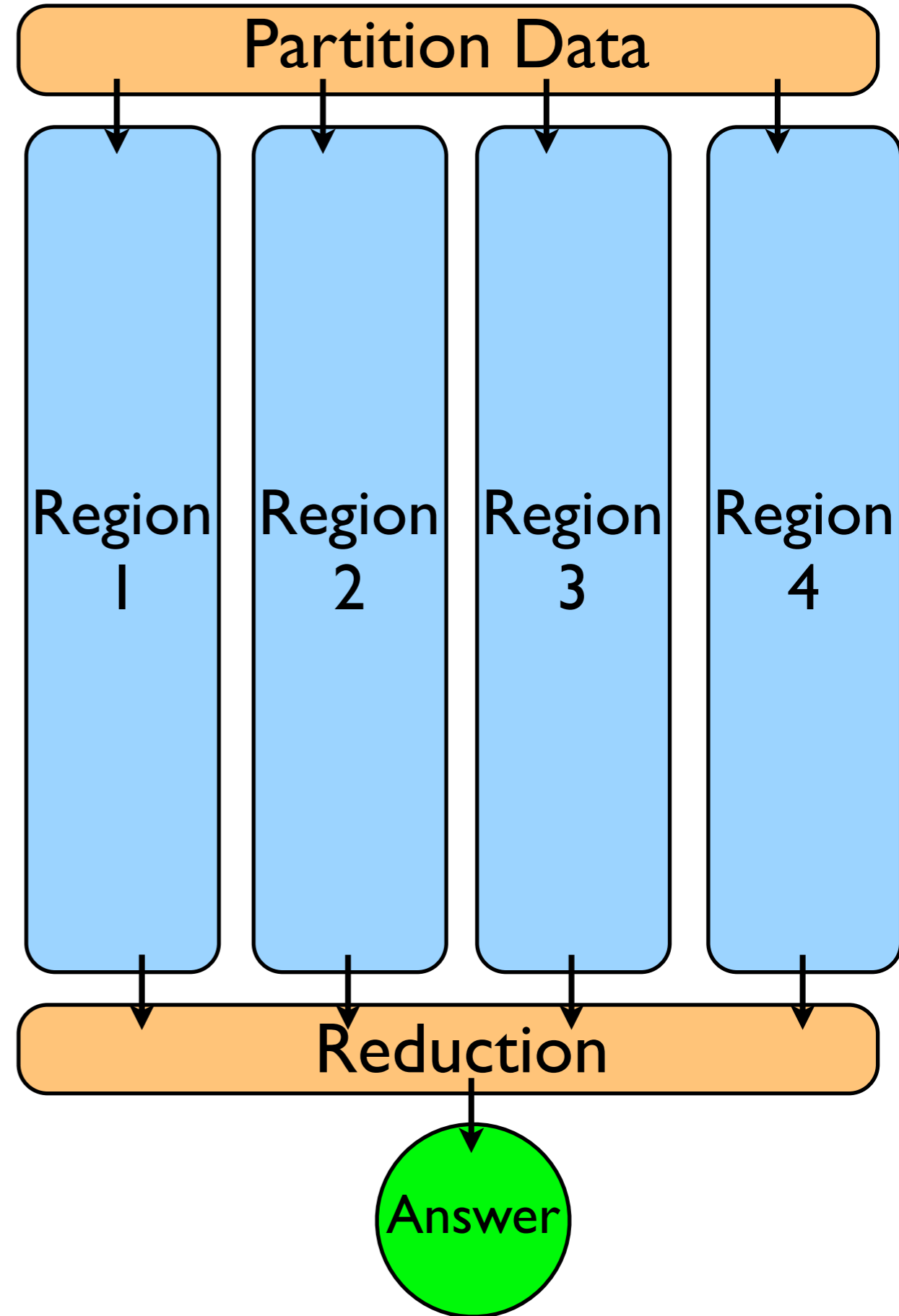


Problems Differ in amount of Concurrency

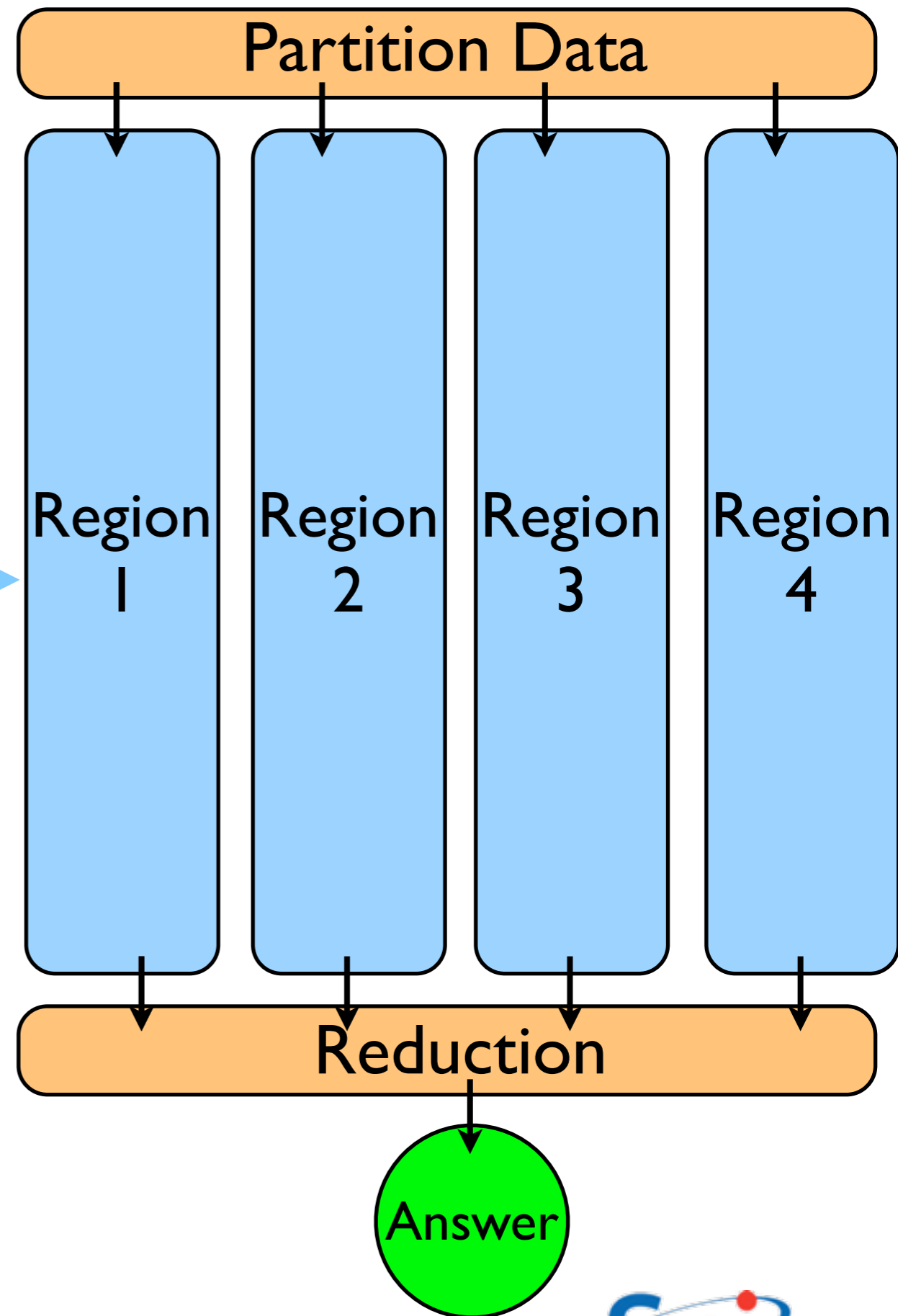
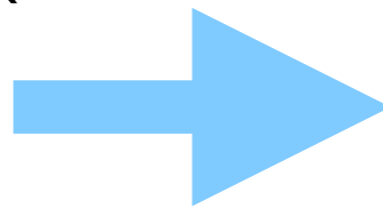
Integrate (or some other simple processing) tabulated experimental data

Integration of different regions can be summed by each processor

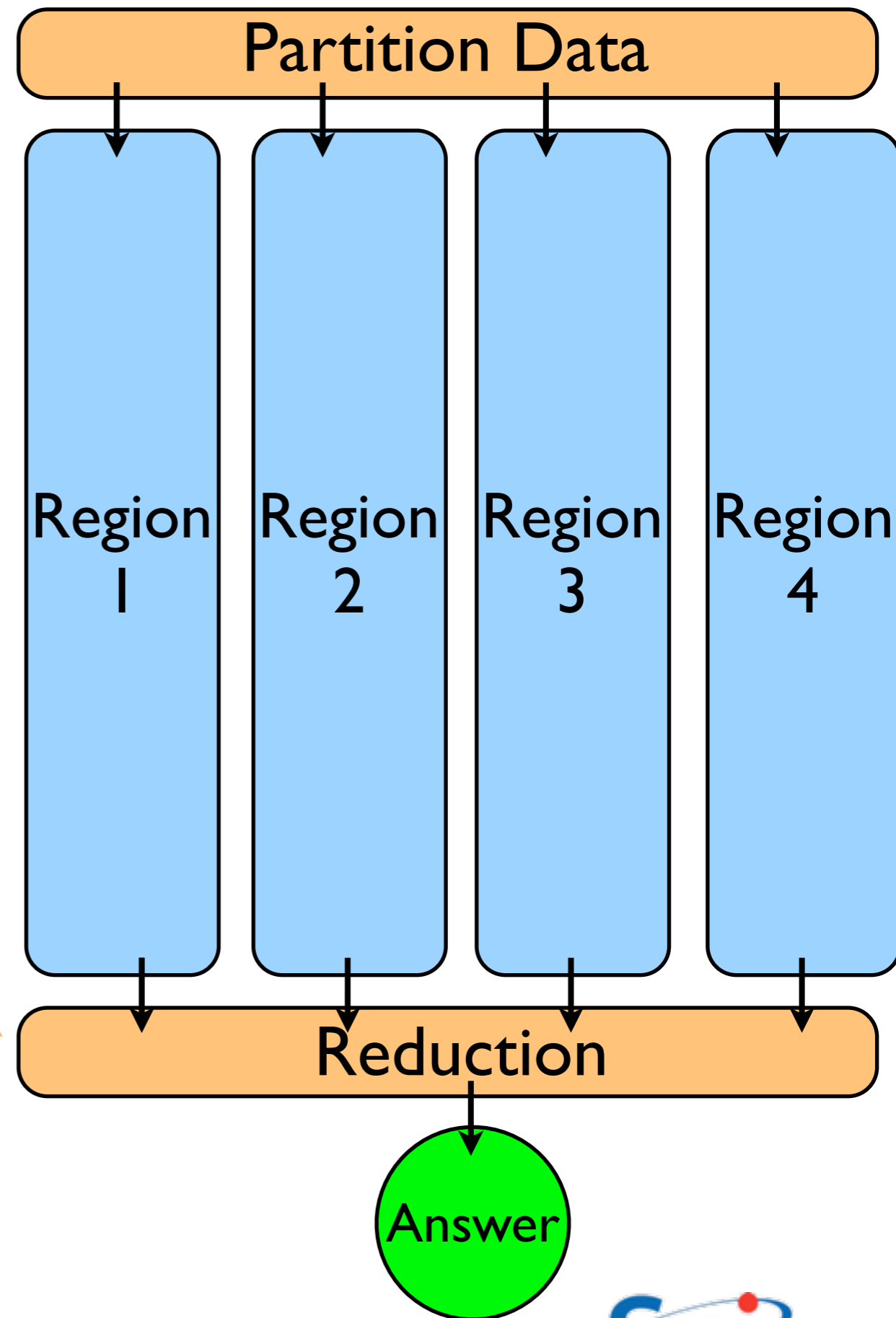
But first need to get data to processor, then bring together all the sums



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



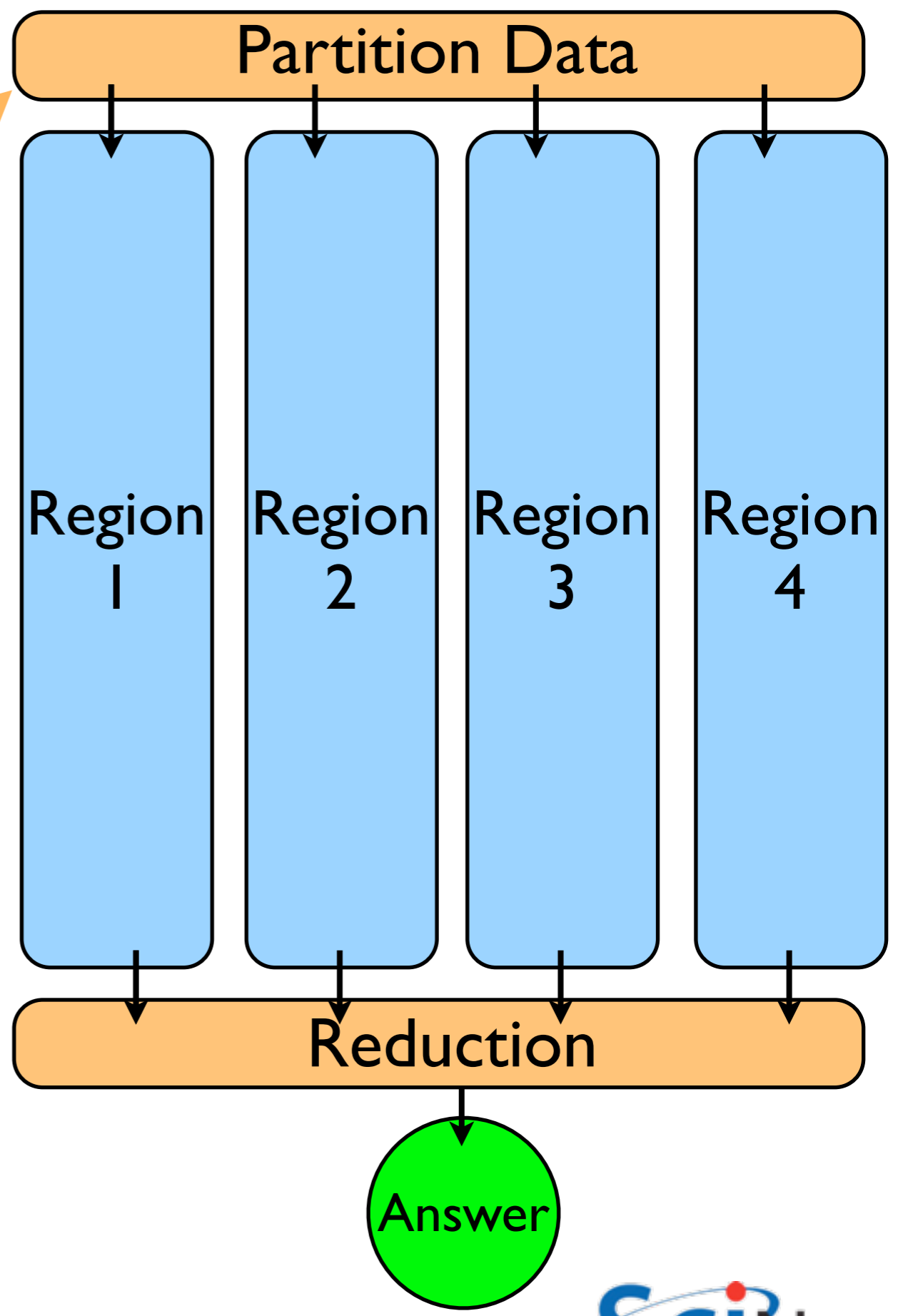
Serial Portion:
Sum has to be
done; if done on one
processor, just same
as serial:
 $T \sim \text{const}$



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*



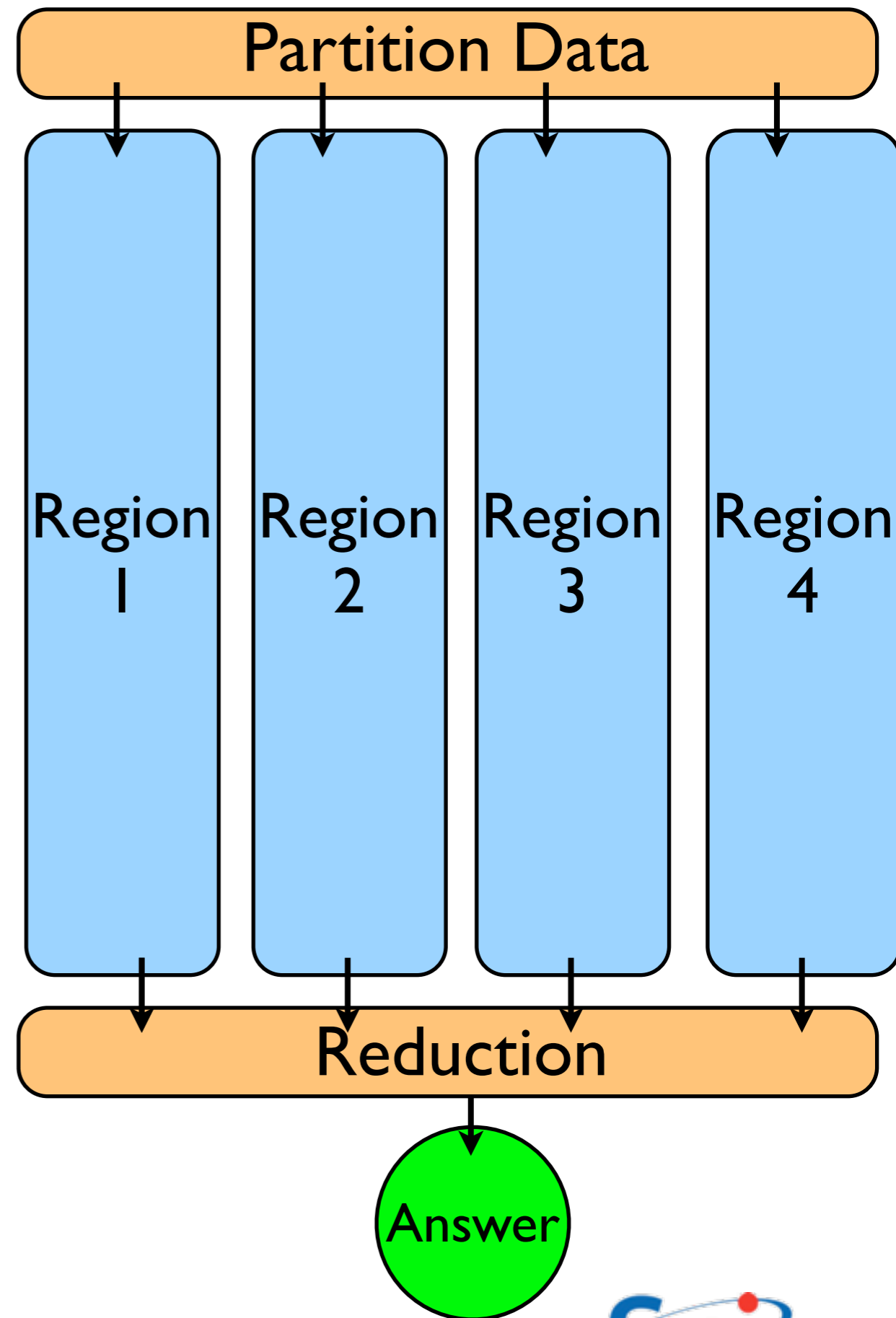
Total Time: Serial + Parallel

Ignoring data-moving costs (for now):

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

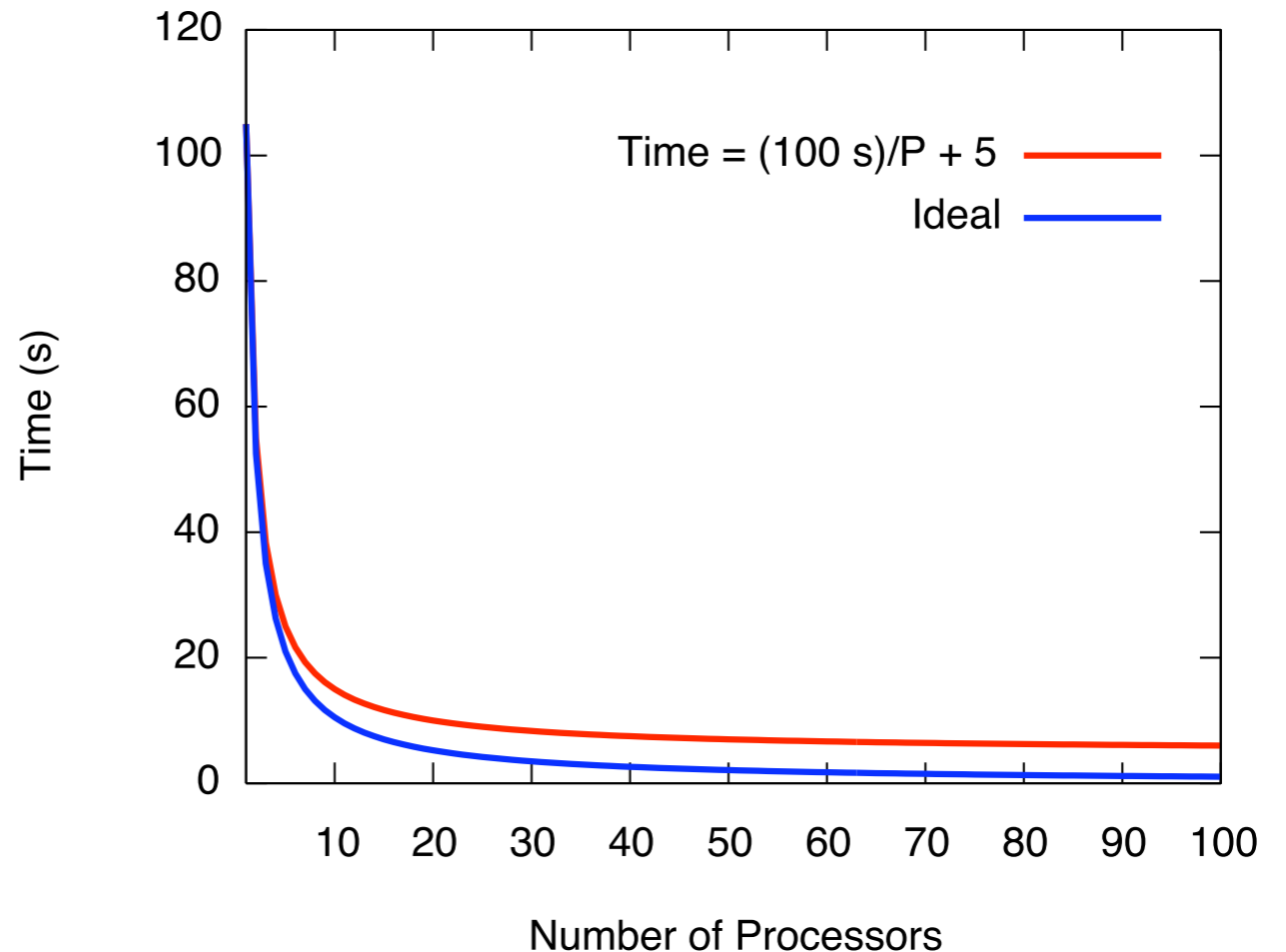
Typically linear in P (sum)

Eventually, as problem becomes increasingly scaled up, serial term dominates



Timing of simple case

Ignore data transfer costs; say:
100 s to sum up numbers
5 s in assembling the parts
How does this behave on many processors?



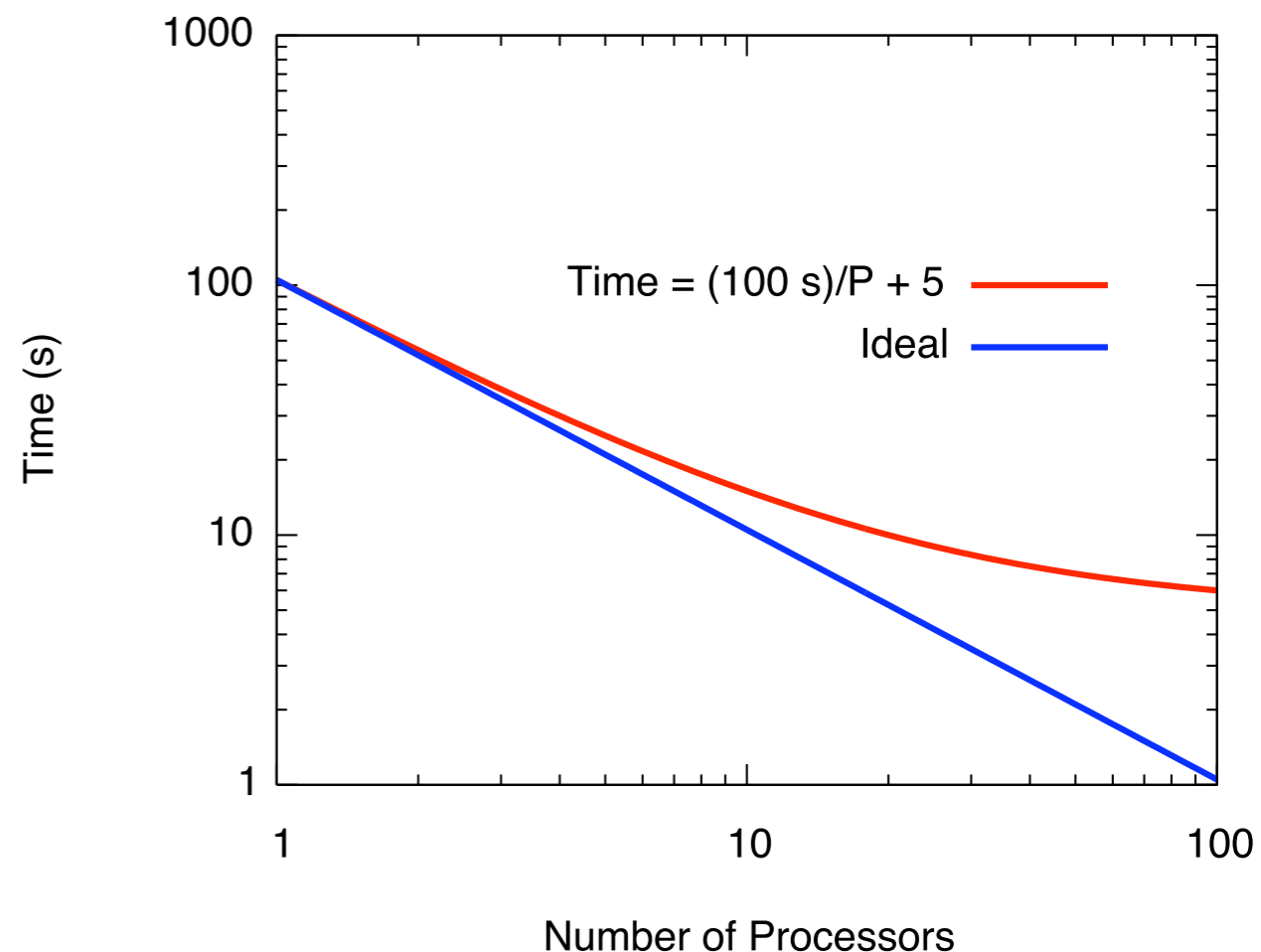
More processors per run don't always help

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: $t(50) = 7s$

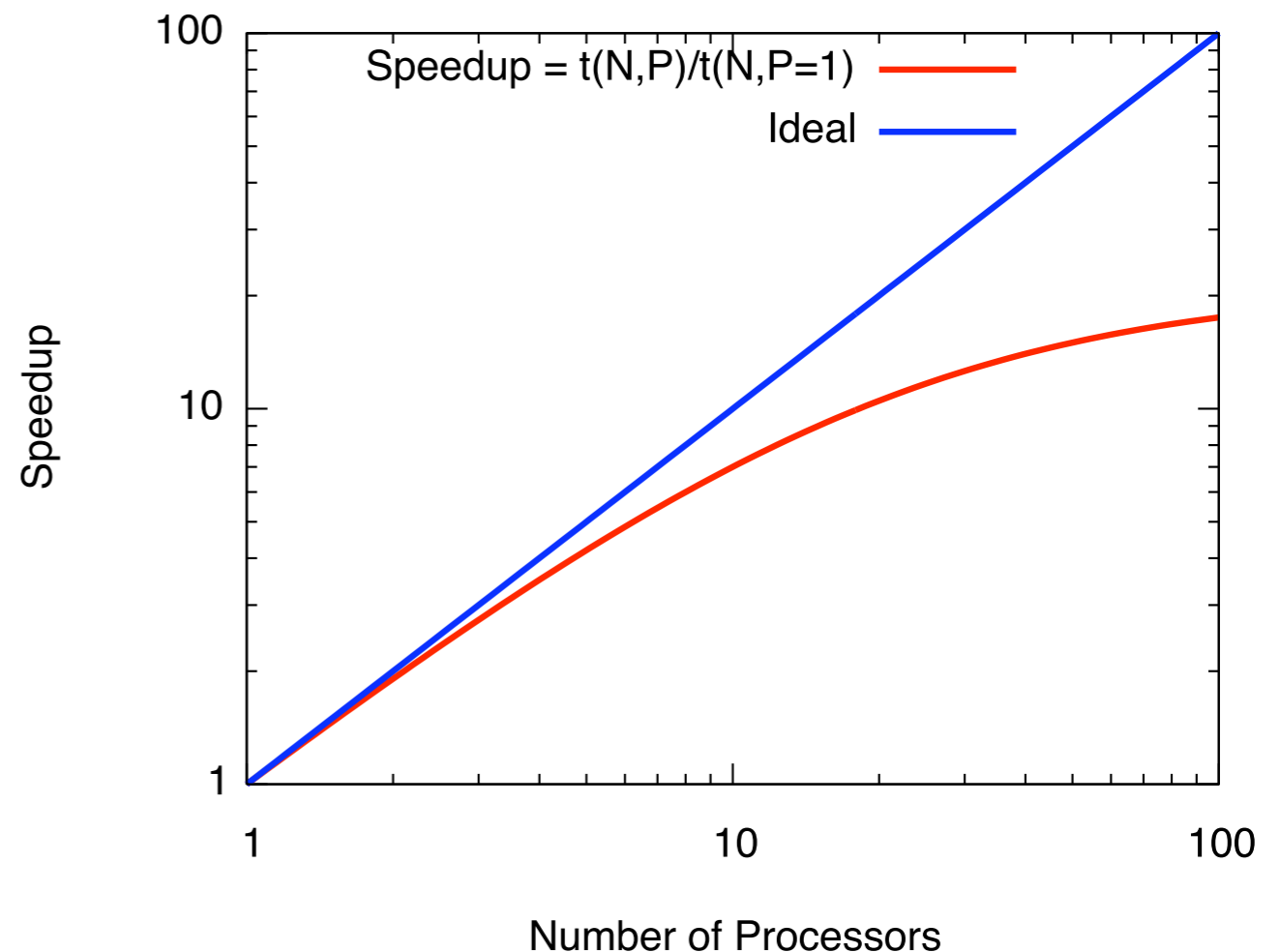
$t(25) = 9s$

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

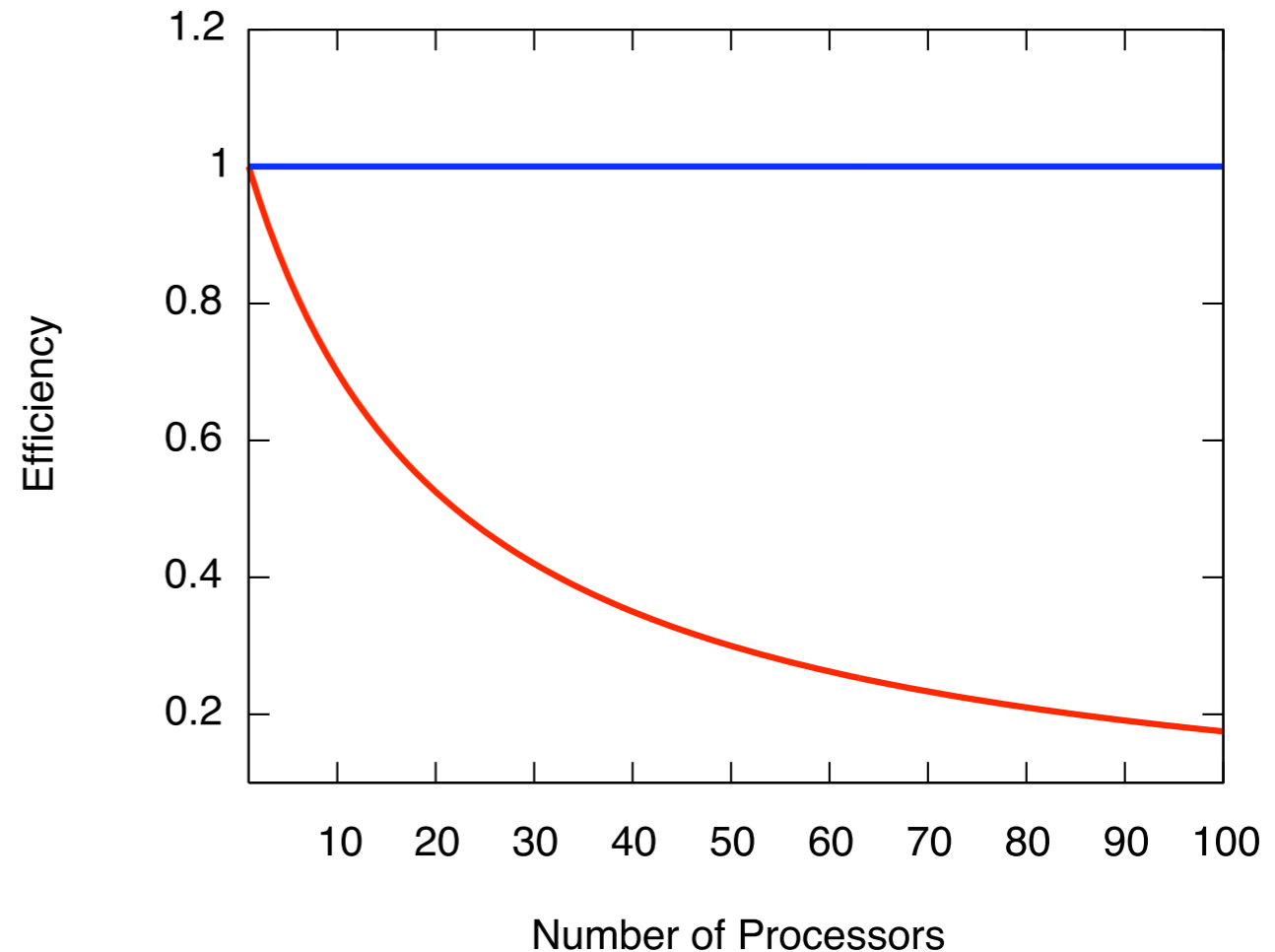
$$\text{speedup}(P) = \frac{t(N, P = 1)}{t(N, P)}$$



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

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

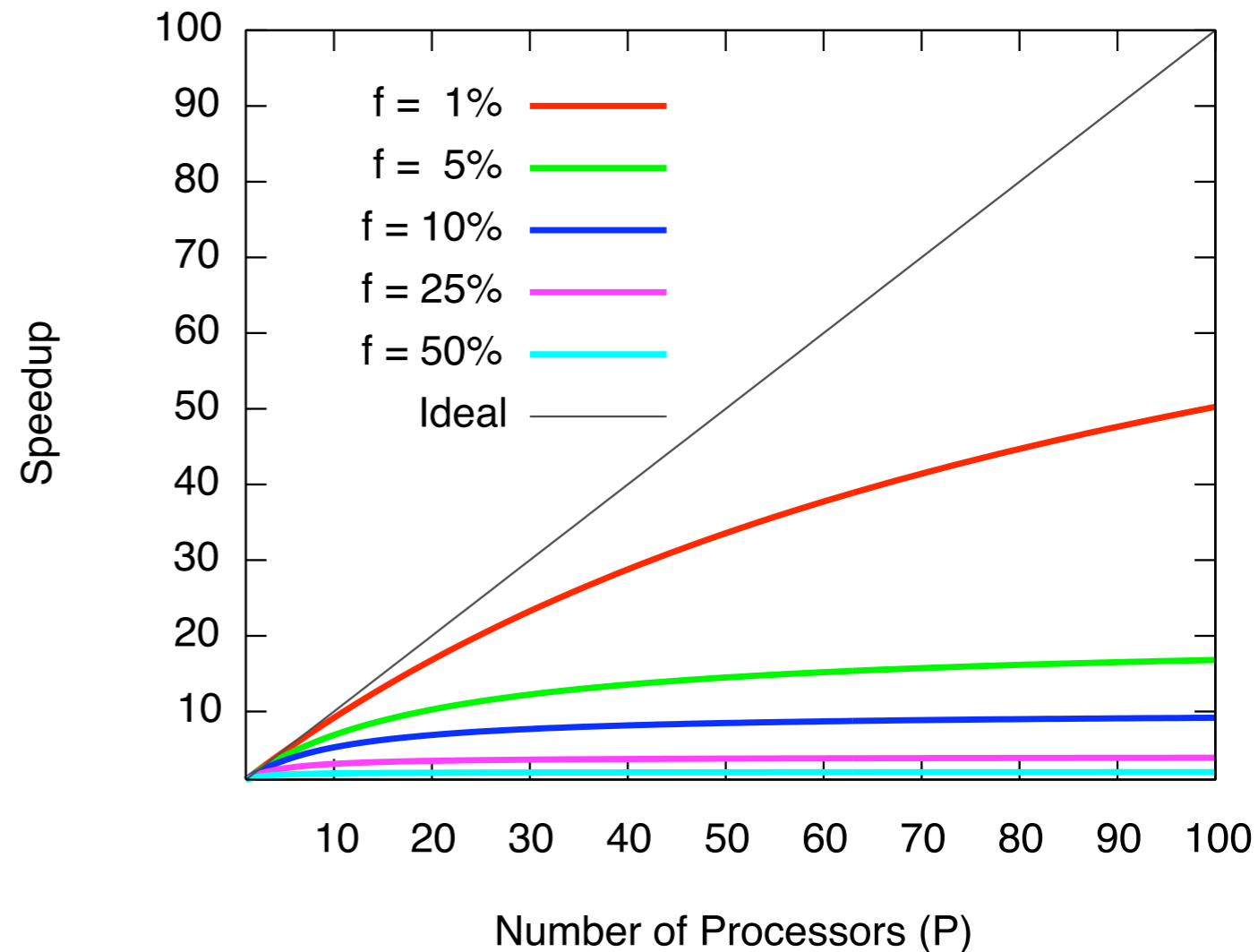
$$\text{Speedup} = \frac{1}{\left(f + \frac{1-f}{P} \right)}$$

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

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

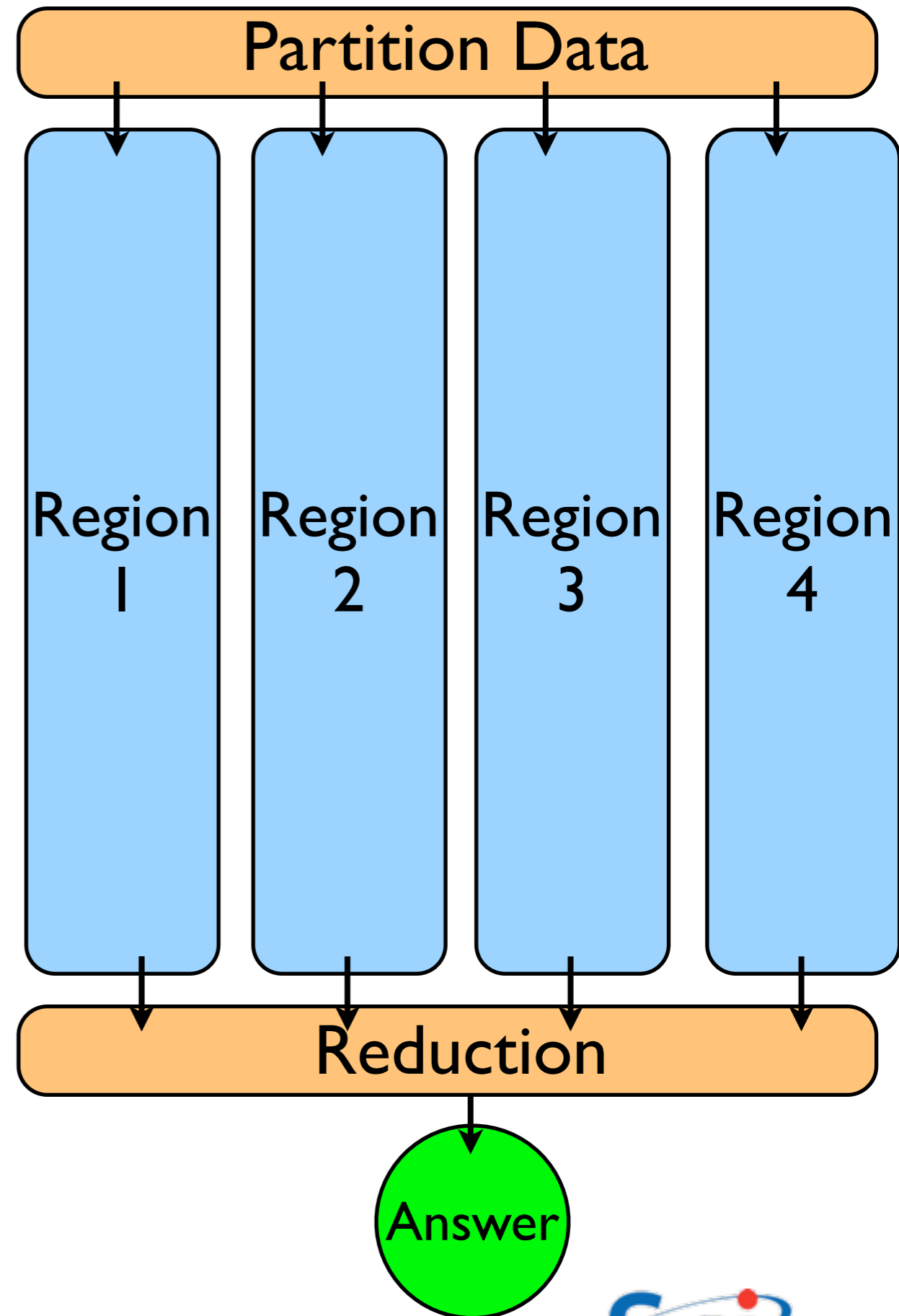
Amdahl's Law

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



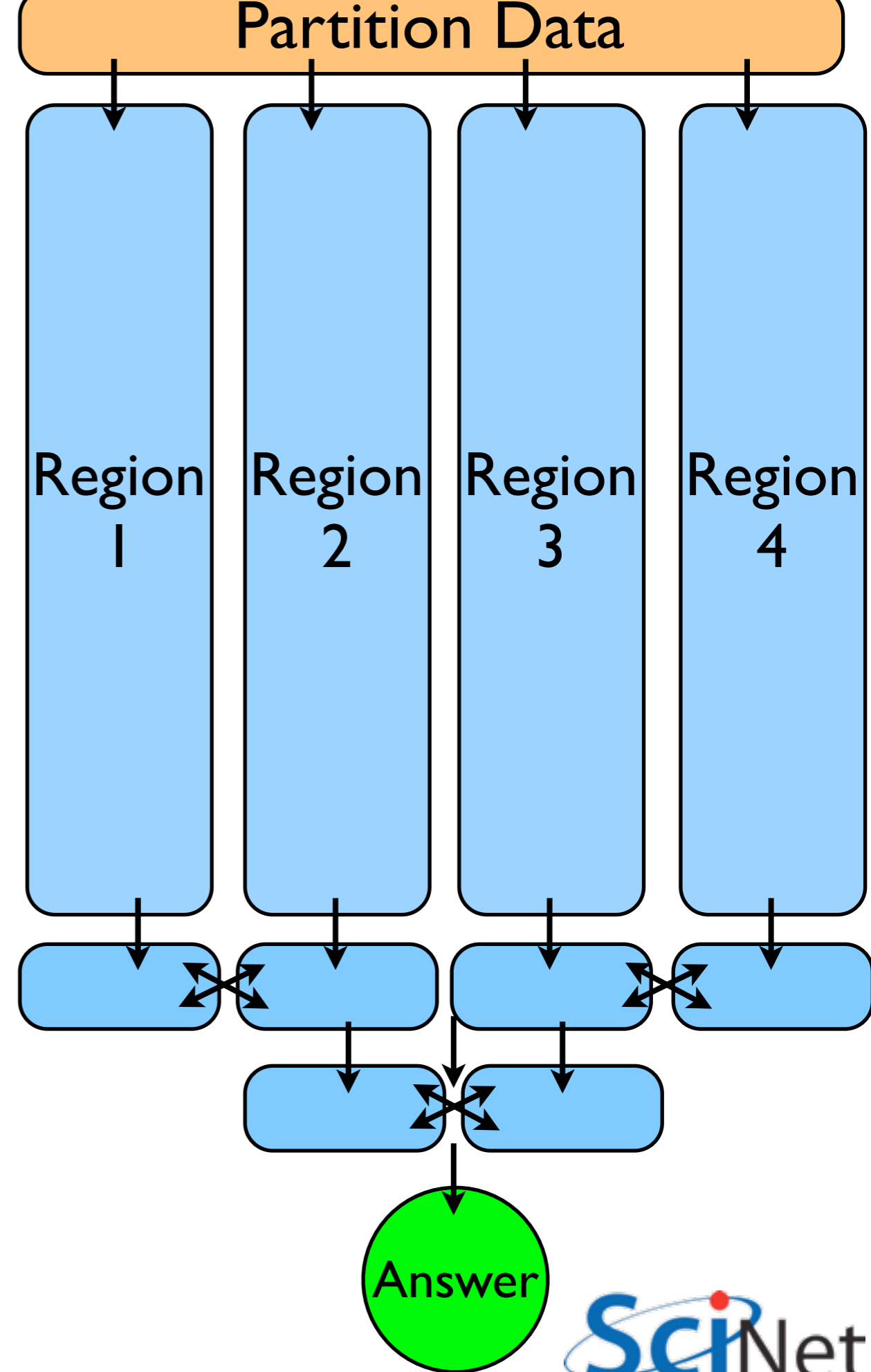
Avoiding Amdahl

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

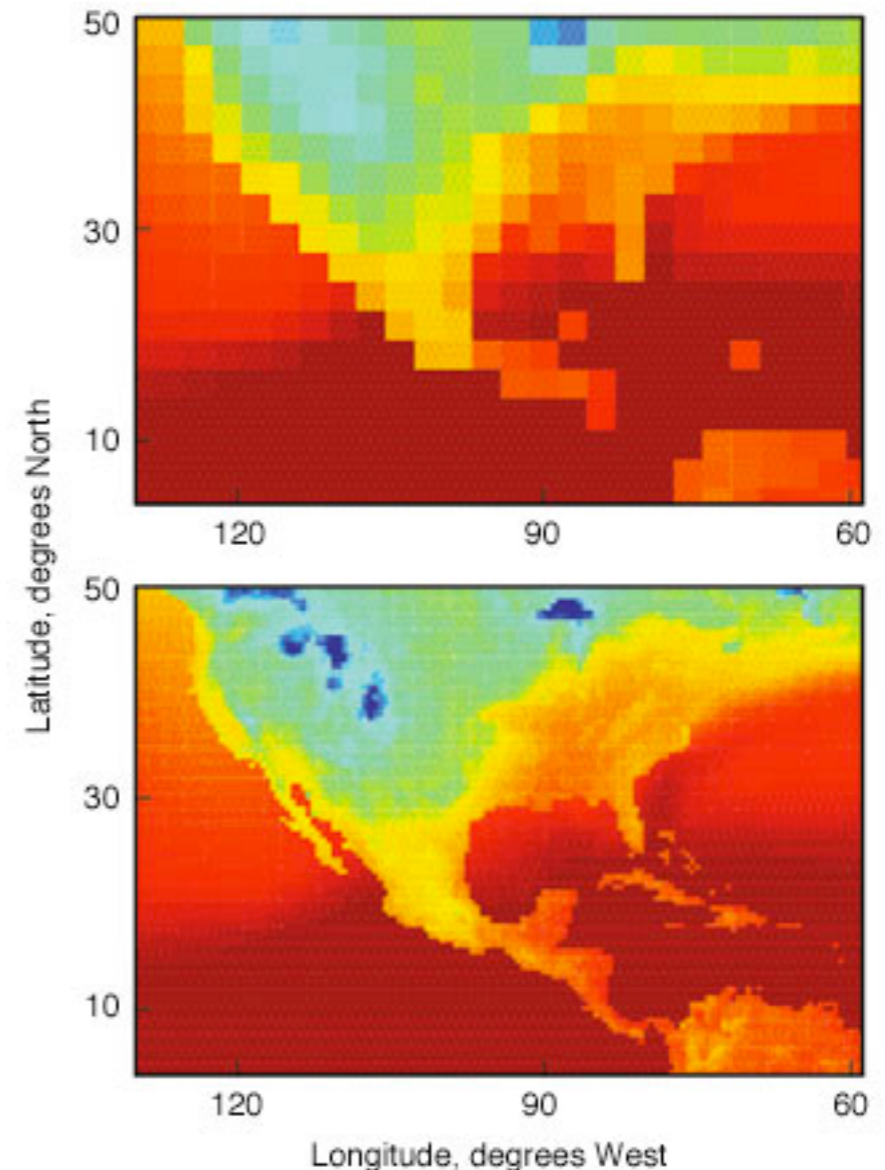
Desktop problem isn't a
supercomputer problem!

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.

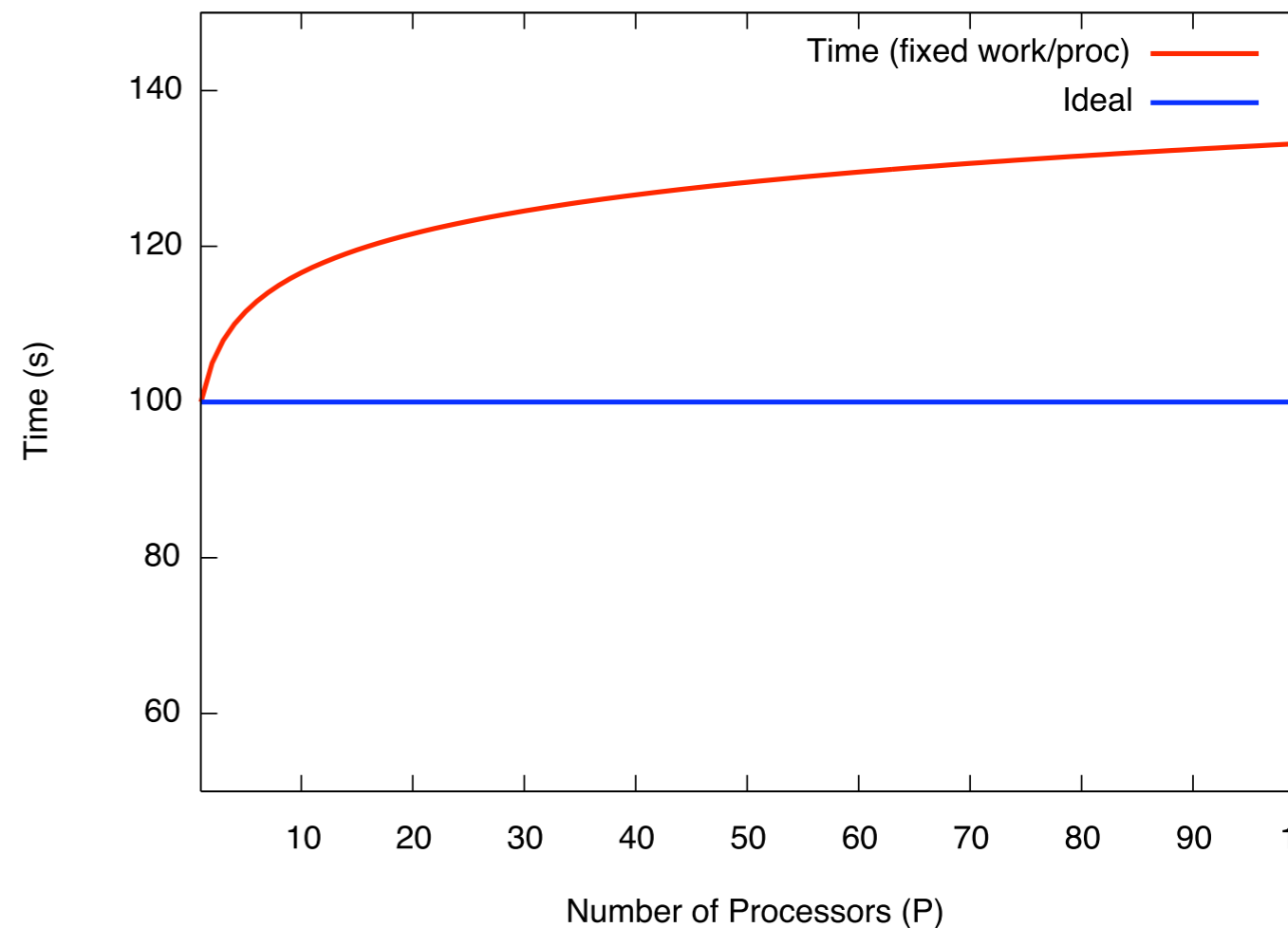


Weak Scaling

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

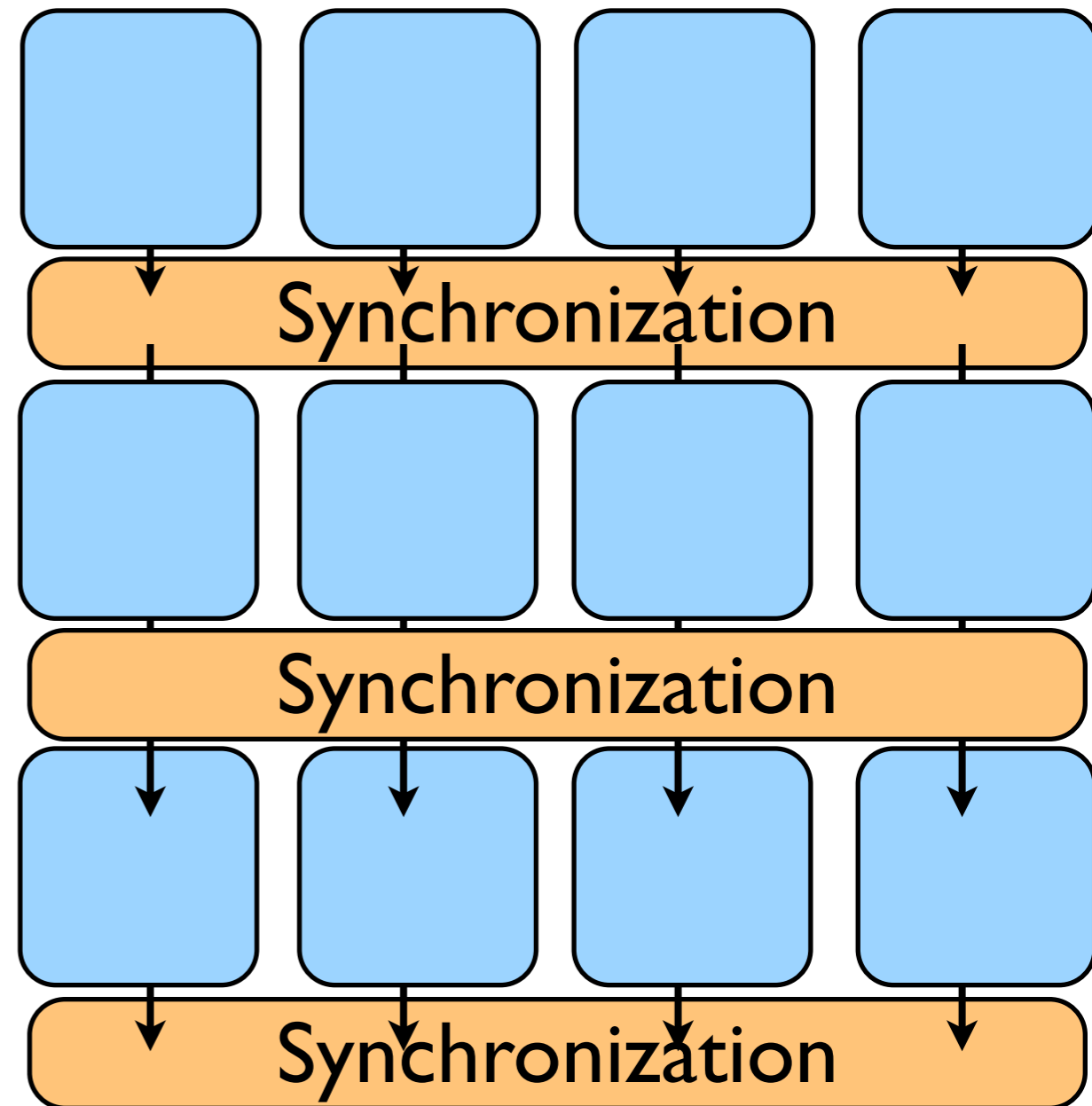
Most problems are not pure
concurrency

Some level of synchronization,
exchange of information needed
between tasks

This needs to be minimized

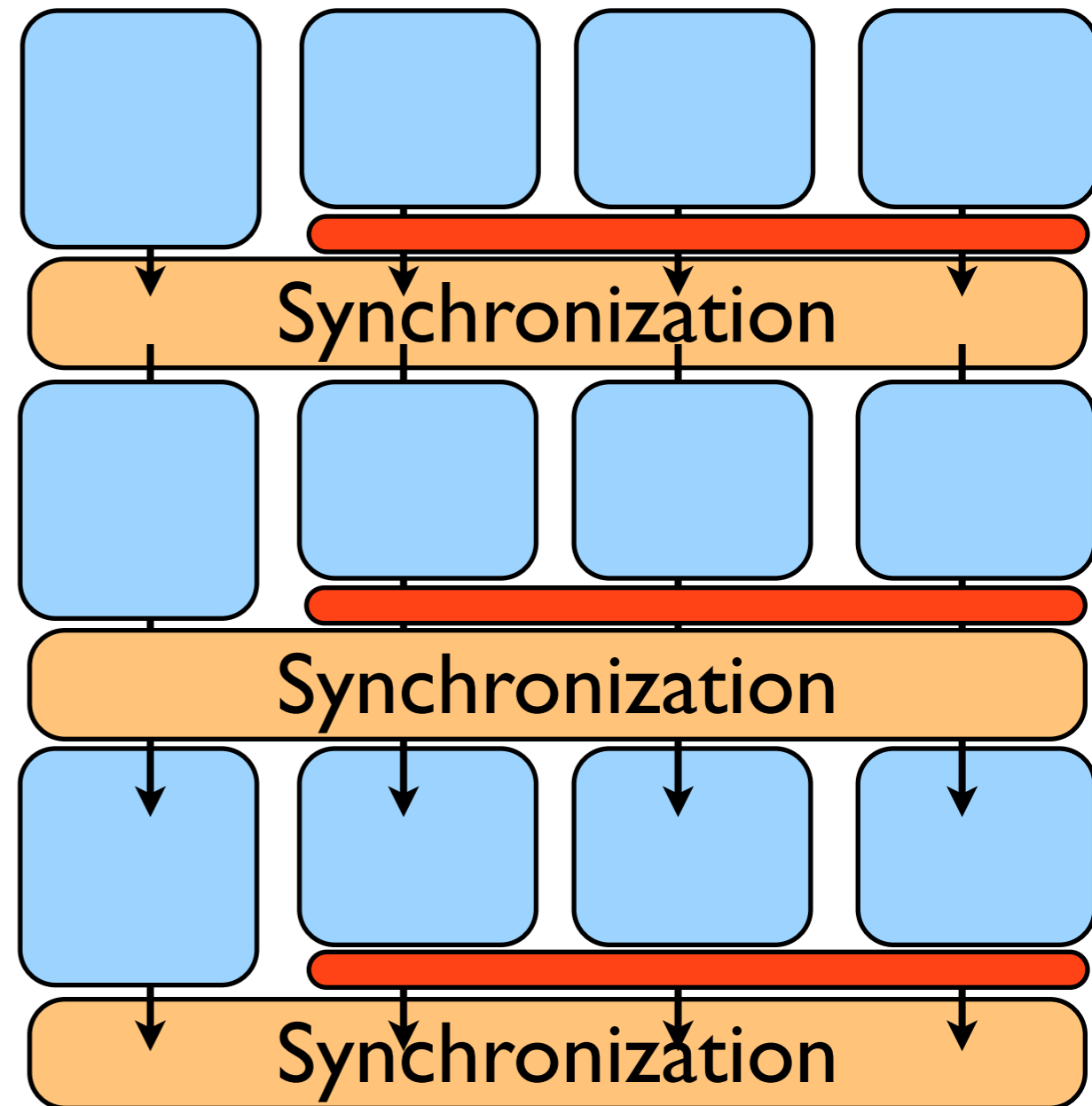
Increases Amdahl's 'f'

Are themselves costly



Concurrency

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



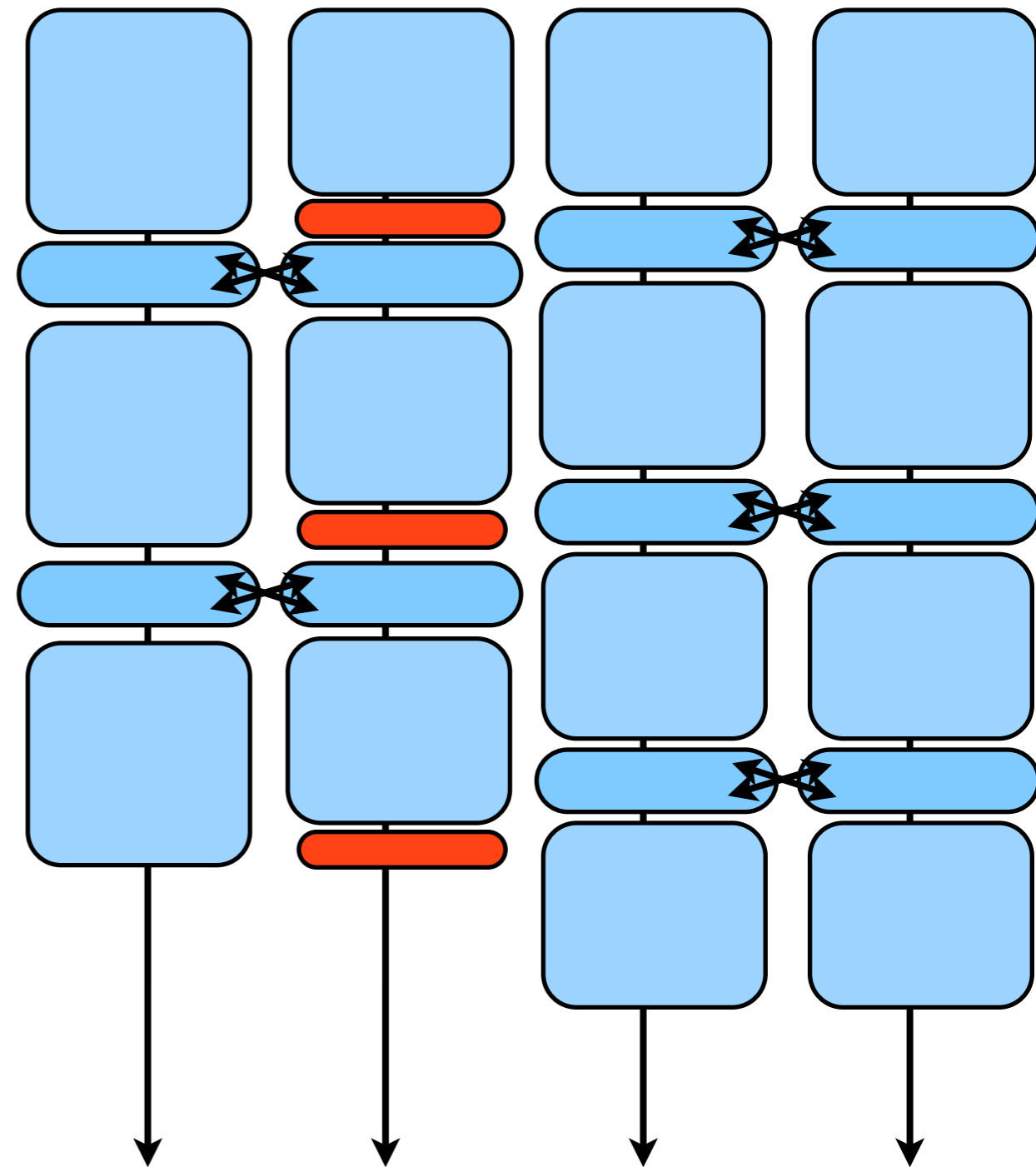
Locality

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

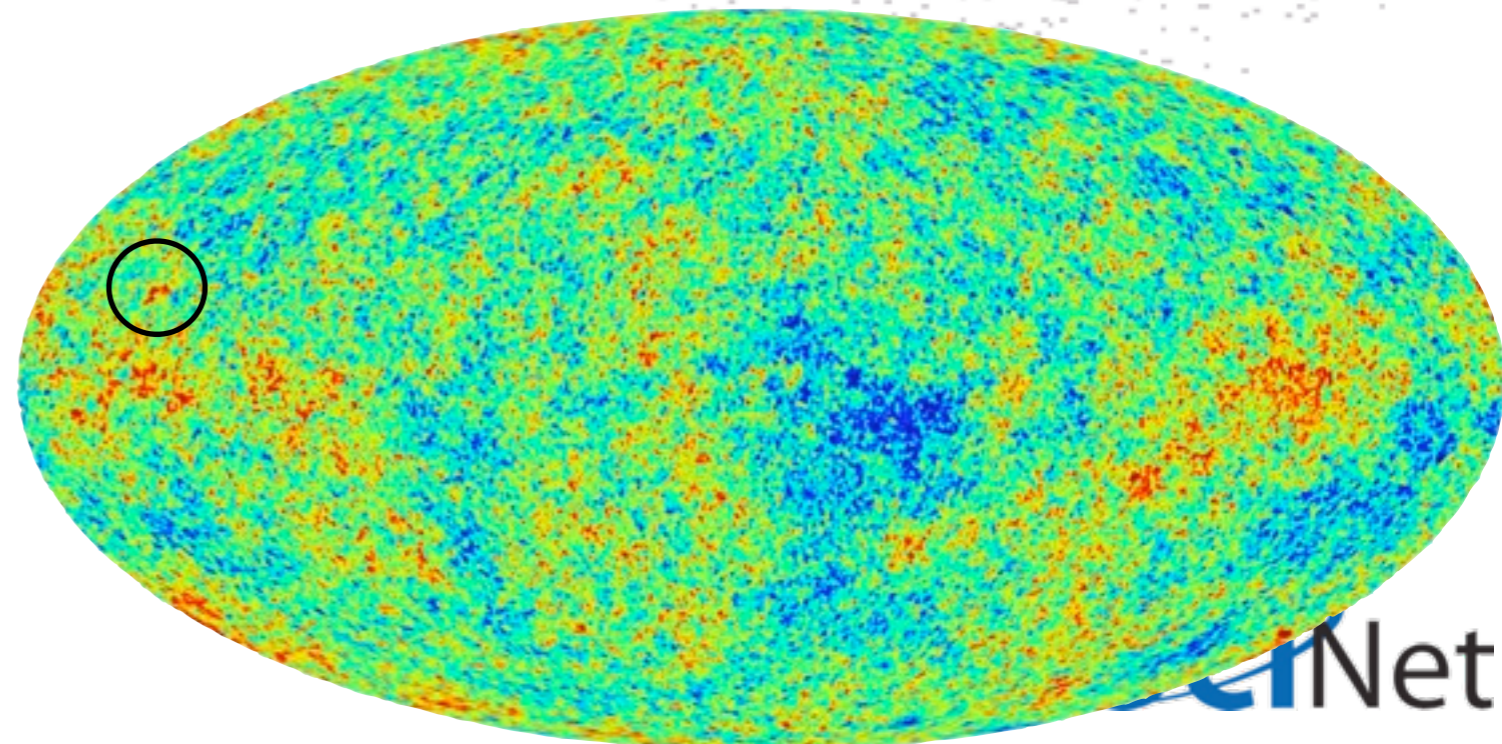
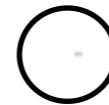
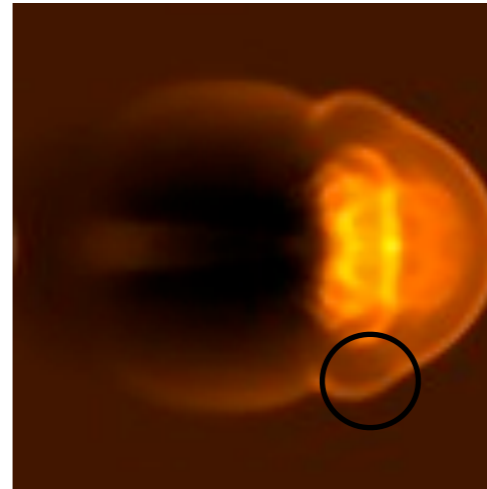
Parallel algorithm design is about finding as much concurrency as possible, and arranging it in a way that maximizes locality.

Finding Concurrency

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

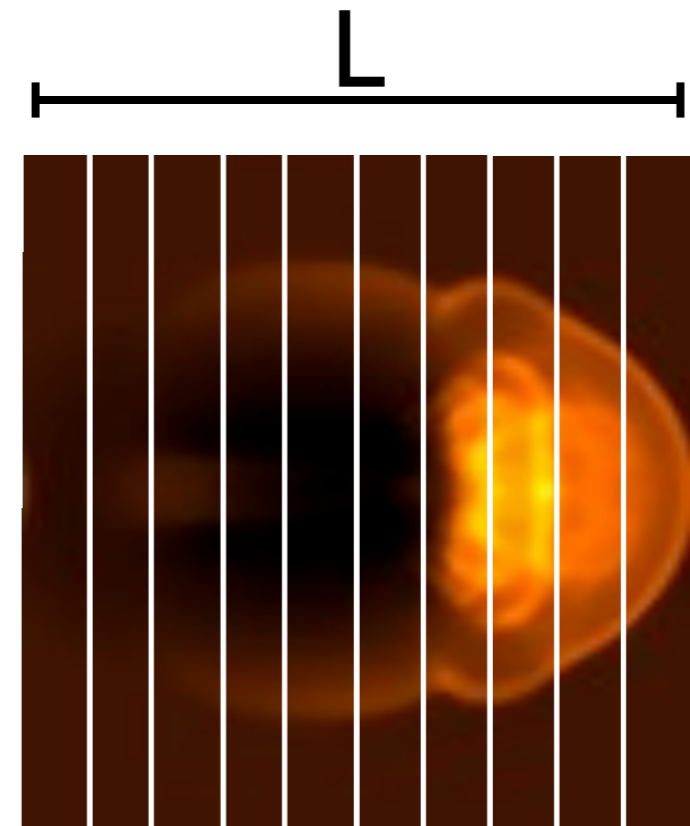
PDEs: parts of domain

N-body: particles (or
interactions)

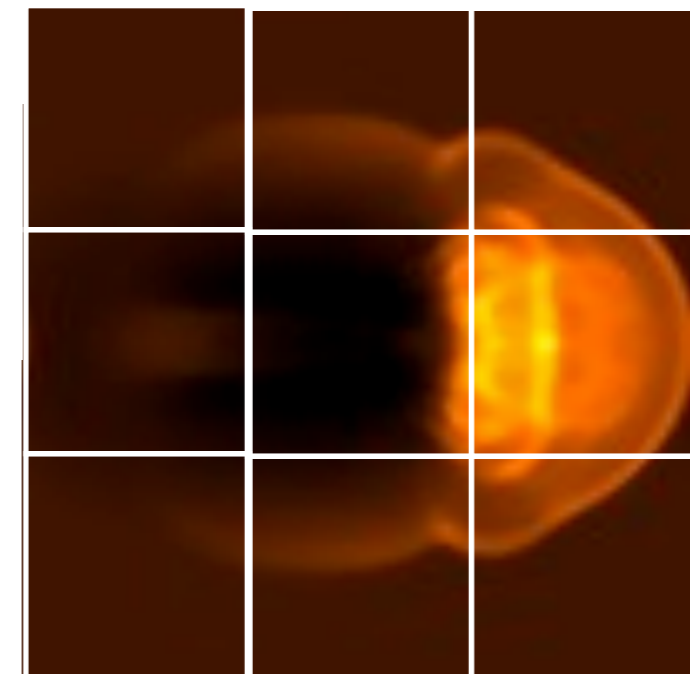


Maintaining Locality

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



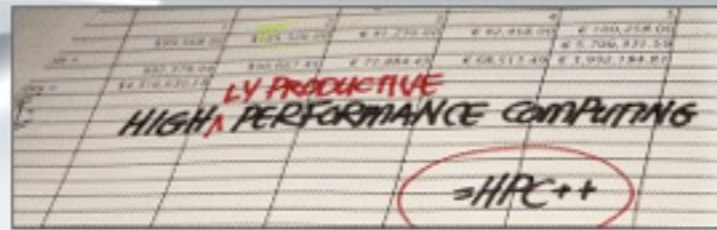
perimeter
 $= 9L$



perimeter
 $= 4L$

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.

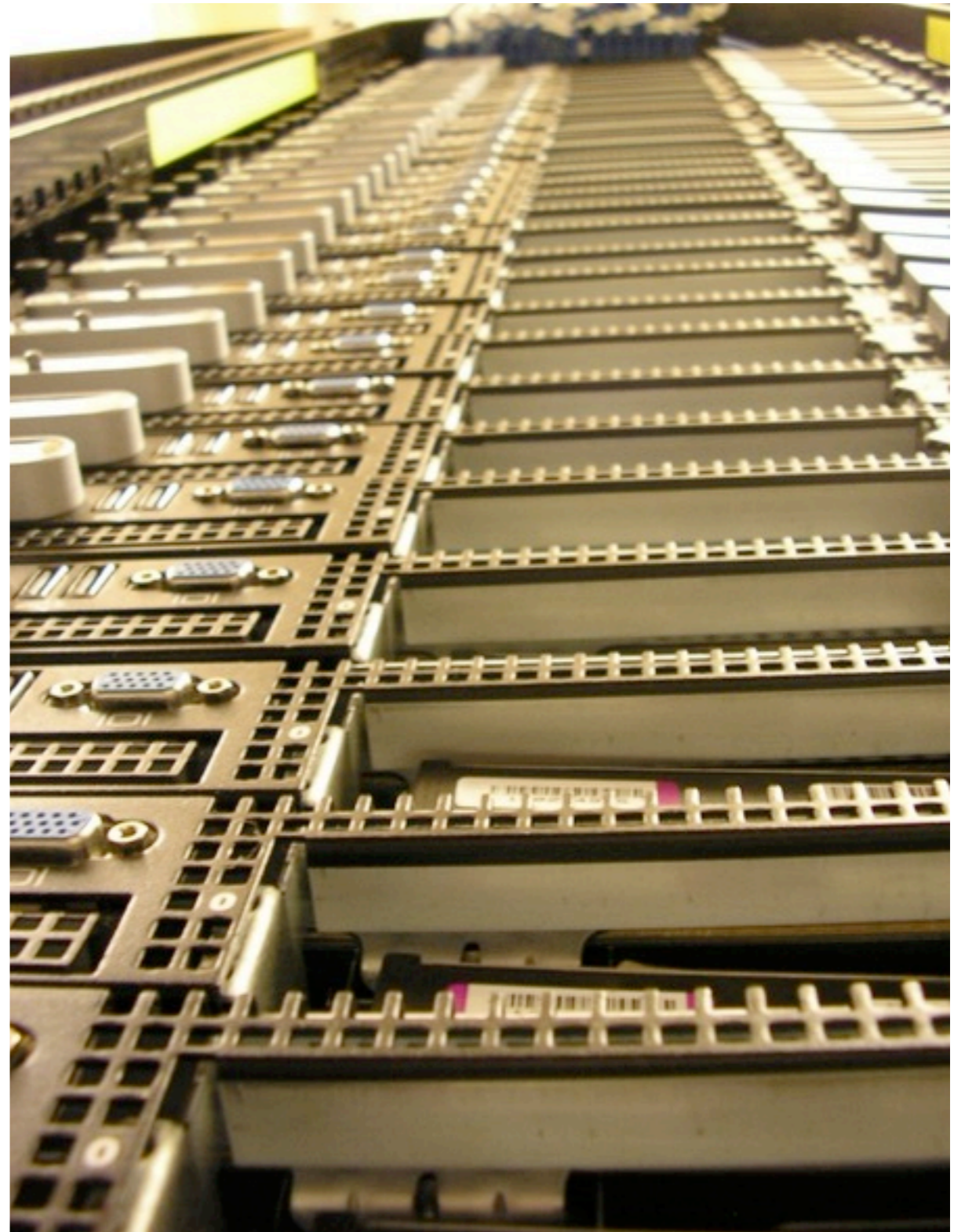
Info about architecture, ...

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



Computer Architectures

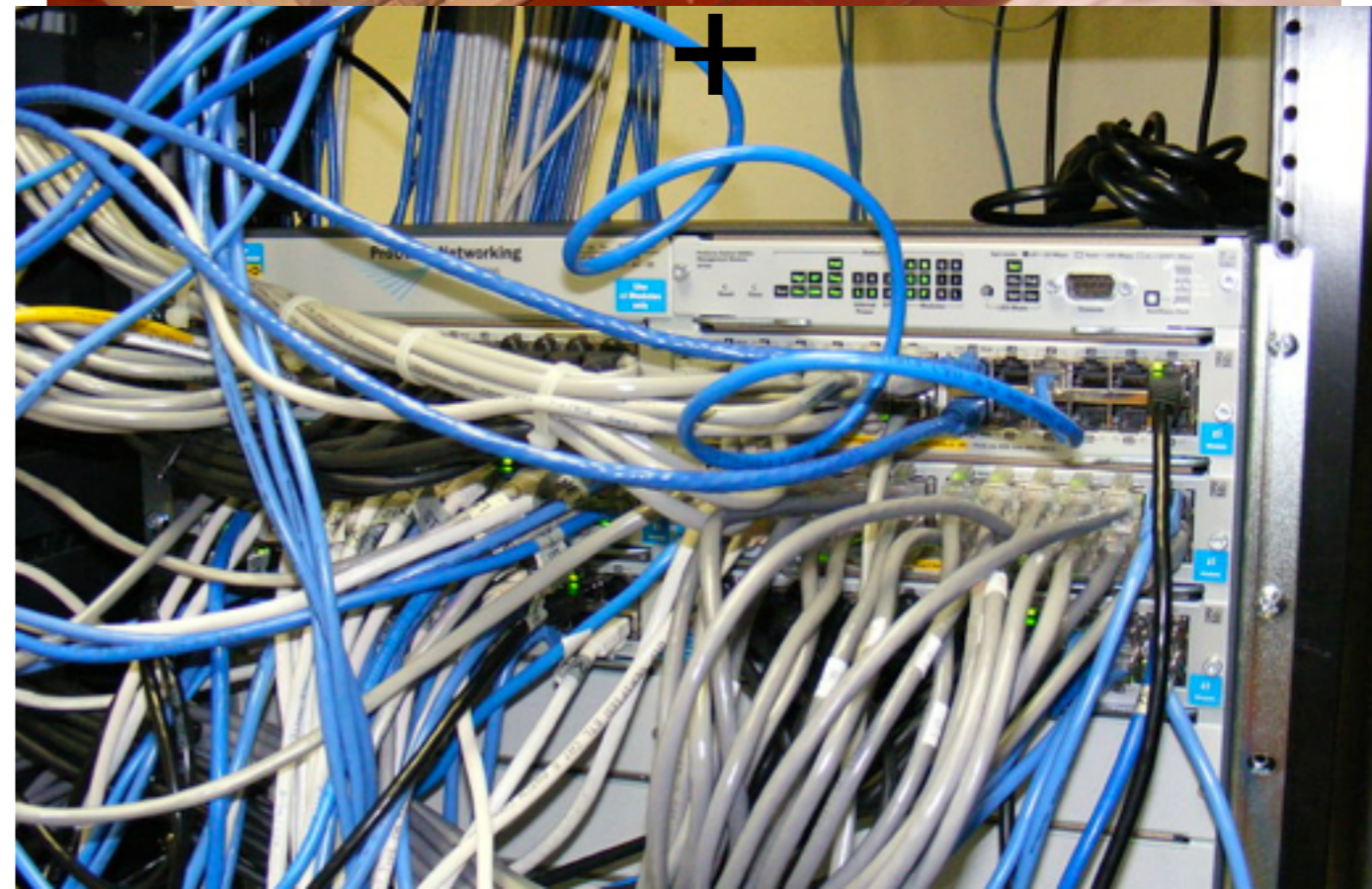
How the computers work shape
how best to program them
Shared Memory vs Distributed
Memory.
Vector computers...



Distributed Memory: Clusters

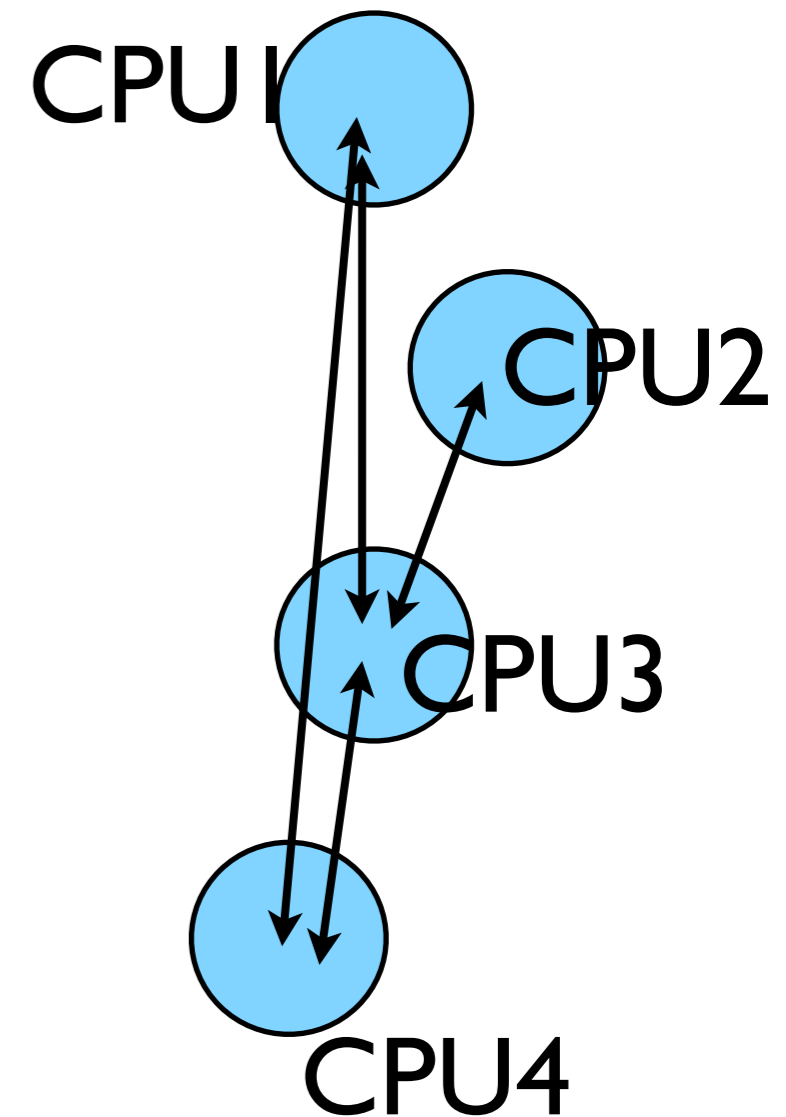
Simplest type of parallel
computer to build

- 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

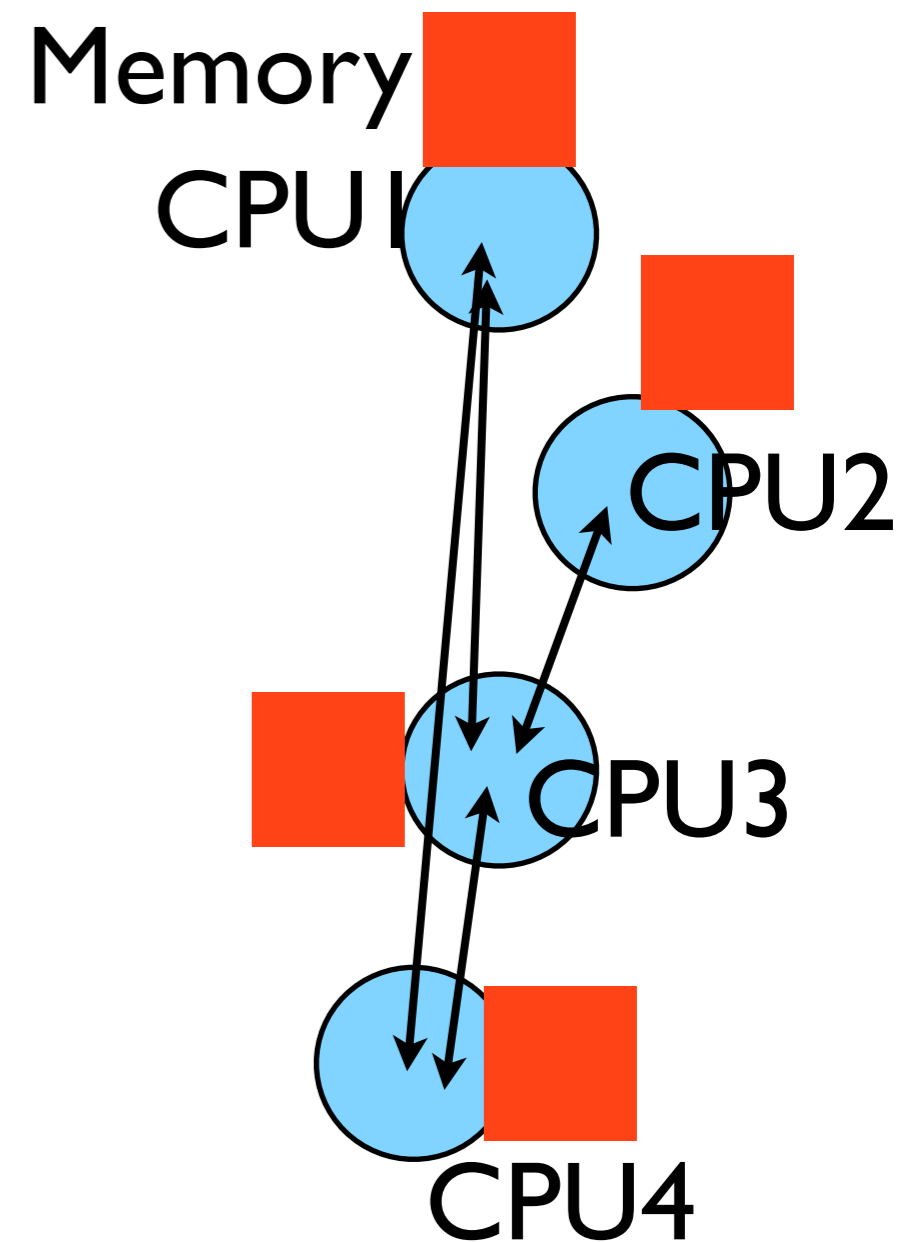


Each node has independent memory

Locally stores its own portion of problem

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

Usual model: 'message passing'

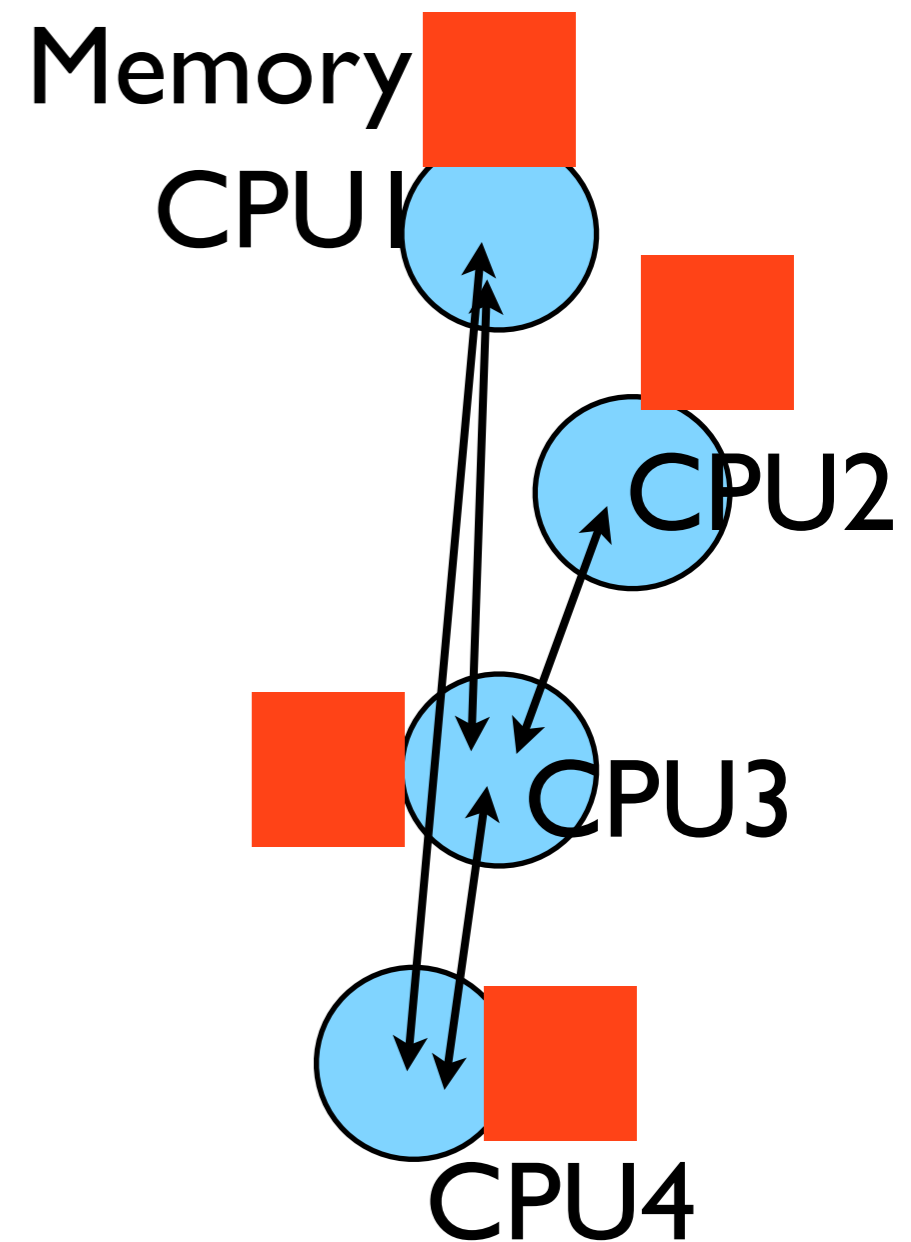


Clusters + Message Passing

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)

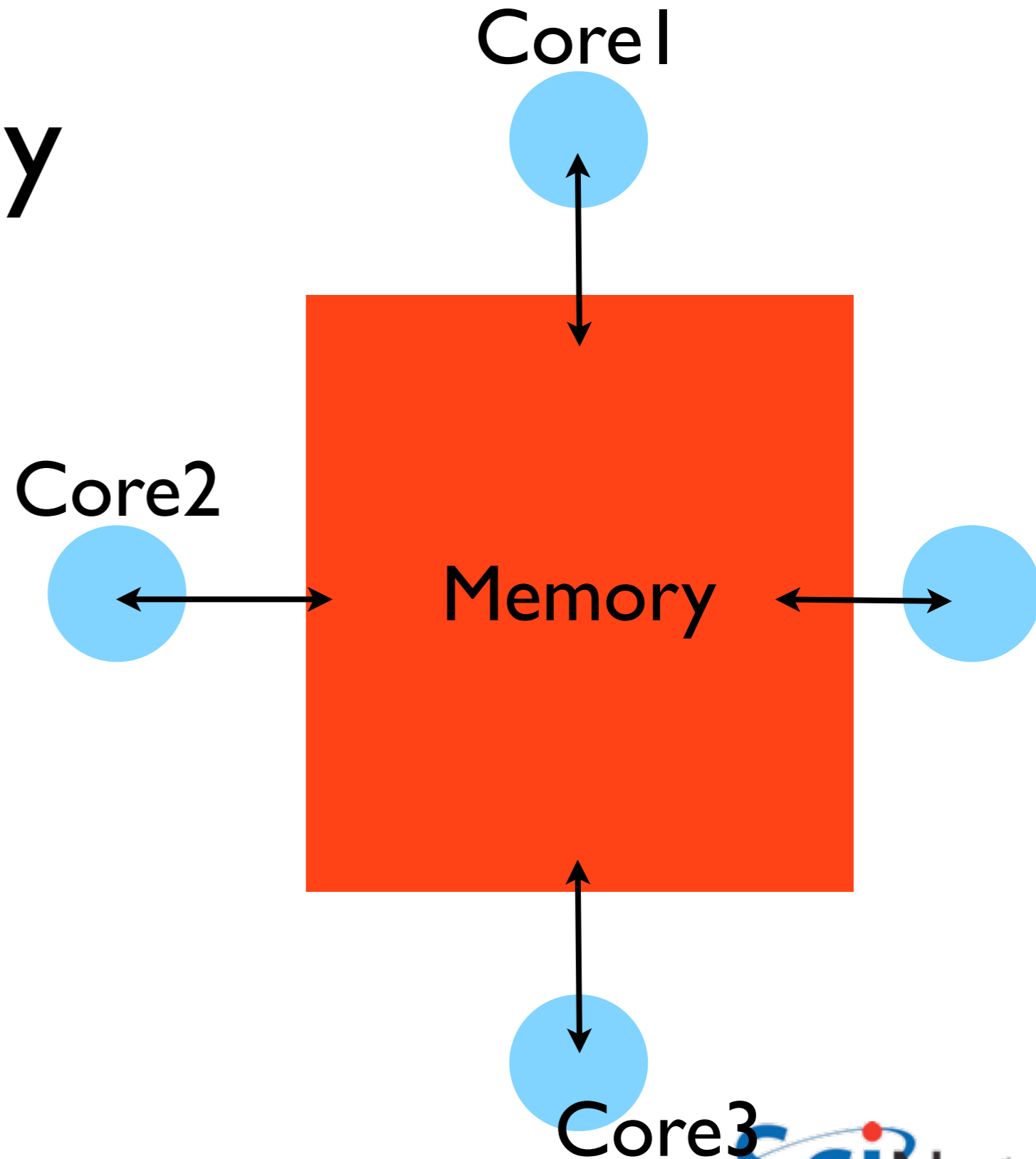
Processor speed: 1 FLOP ~ few ns or less

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

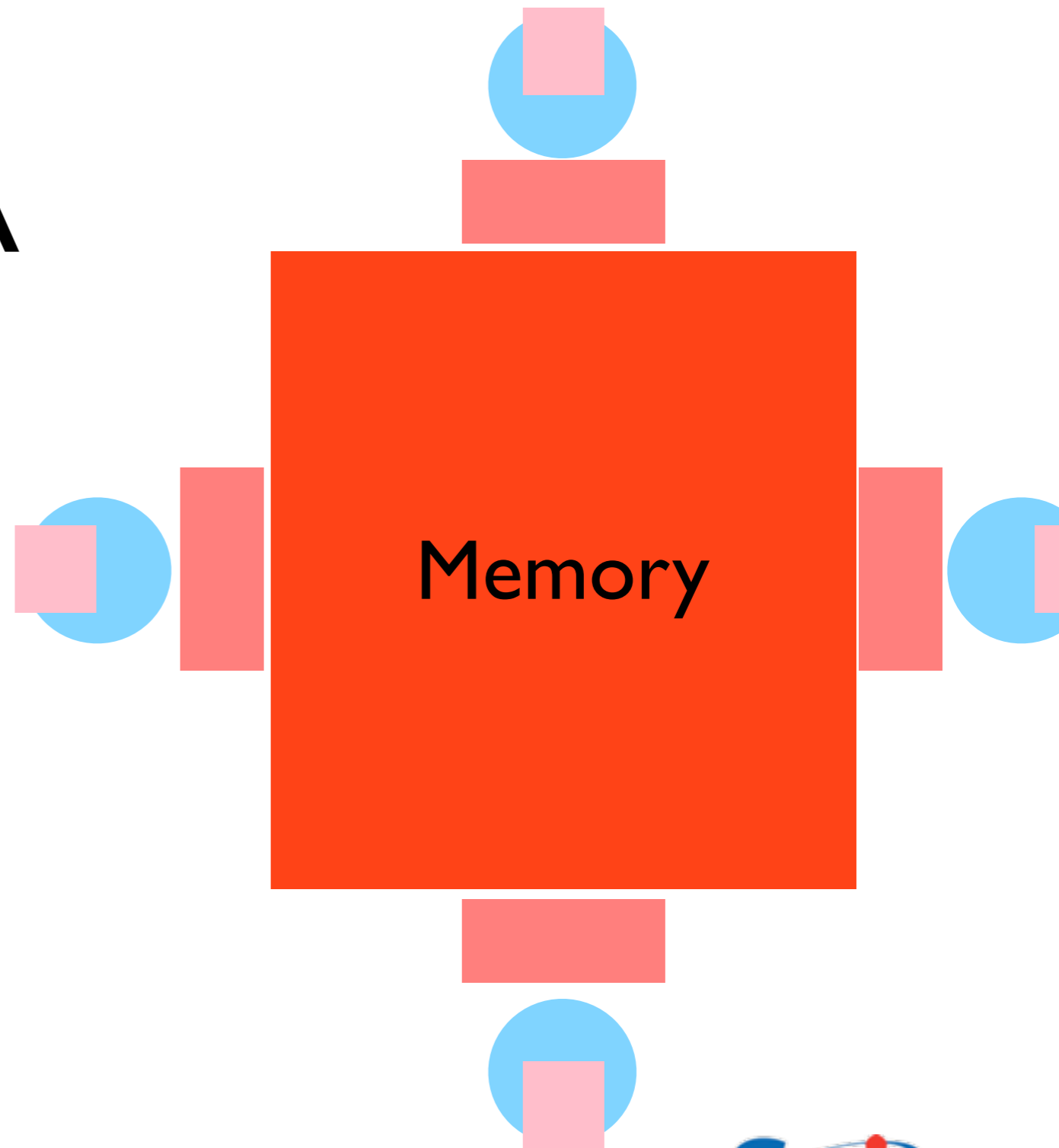
Complicating things: each core typically has some of its own memory

Non-Uniform Memory Access

Locality still matters

Cores have cache, too.

Keeping this memory *coherent* is extremely challenging



Coherency

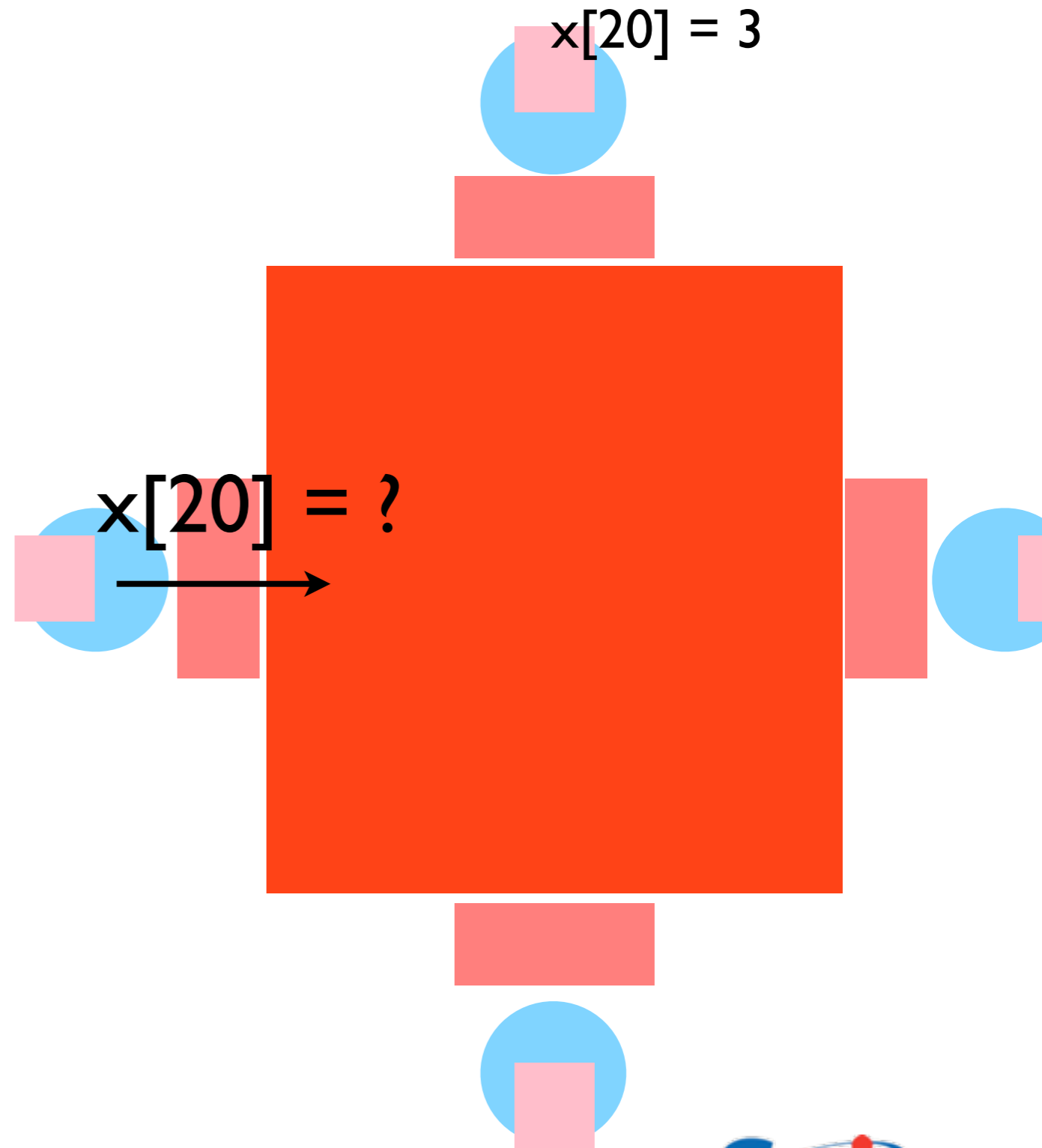
The different levels of memory
imply multiple copies of some
regions

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

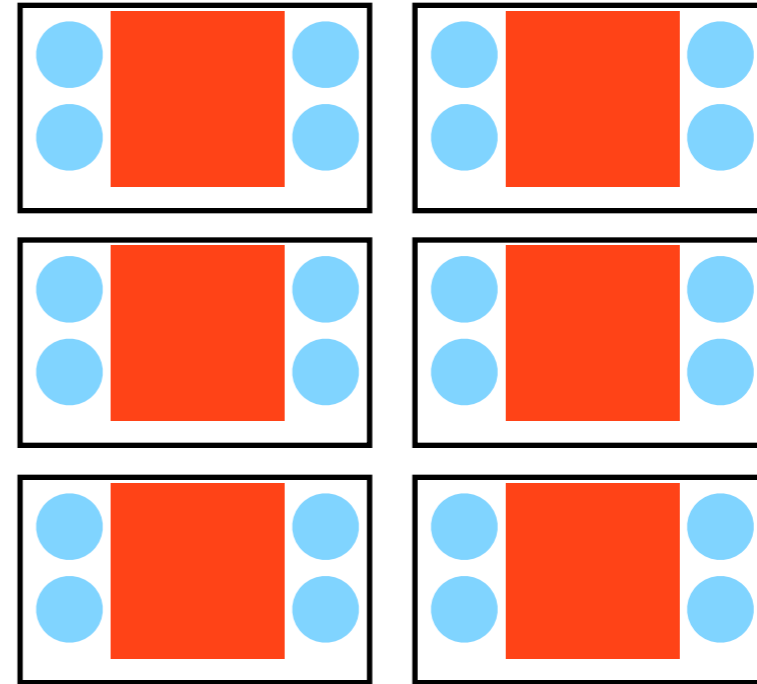
Big Lesson #3

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 MPI:

- login as instructed and ensure this works:
- `cp -R ~ljdursi/course/parCFD ~/`
- `source ~/parCFD/setup`
- `cd ~/parCFD/gettingstarted/`
- `make mpi_hello_world`
- `mpirun -np 8 ./mpi_hello_world`

An introduction to MPI

MPI is a **Library** for Message-Passing

- Not built in to compiler
- Function calls that can be made from any compiler, many languages
- Just link to it
- Wrappers: `mpicc`, `mpif77`

C

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

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

    int rank, size;
    int ierr;

    ierr = MPI_Init(&argc, &argv);

    ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("Hello from task %d of %d, world!\n", rank, size);

    MPI_Finalize();

    return 0;
}
```

Fortran

```
program hellompiworld
include "mpif.h"

integer rank, size
integer ierr

call MPI_INIT(ierr)

call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)

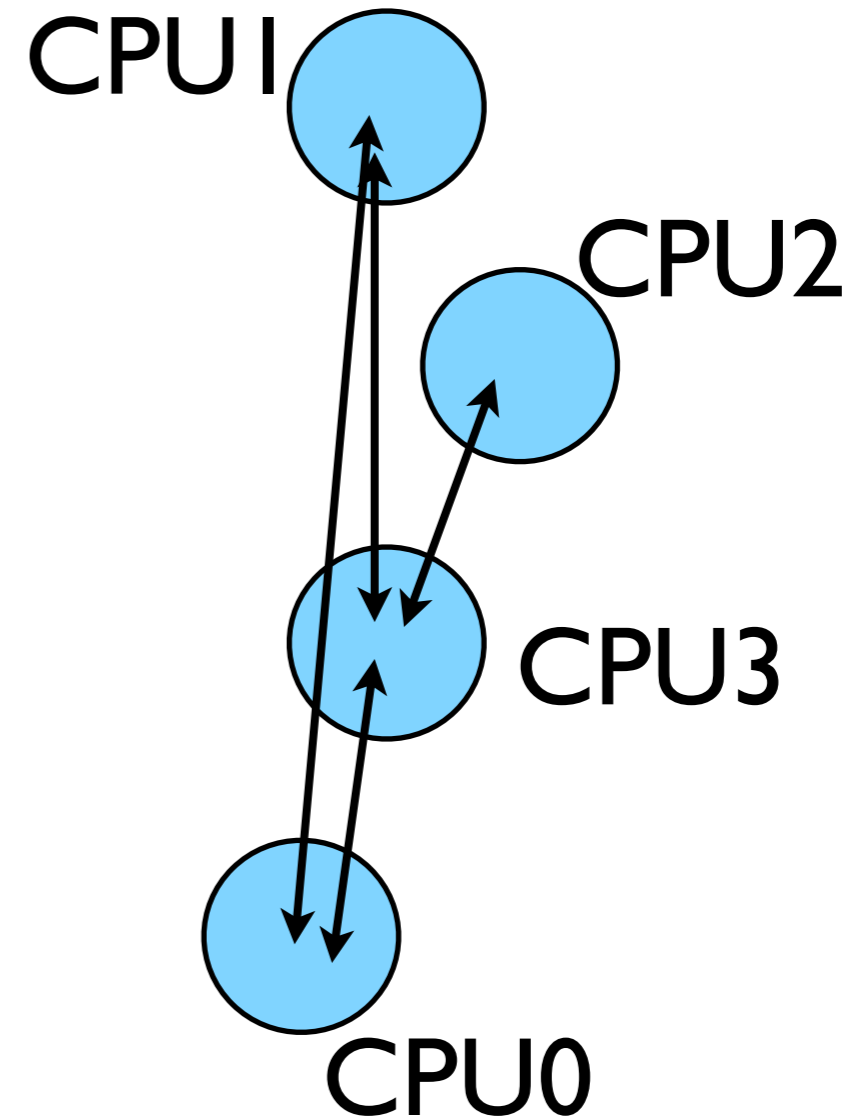
print *, "Hello from task ", rank, " of ", size, ", world!"

call MPI_FINALIZE(ierr)

return
end
```

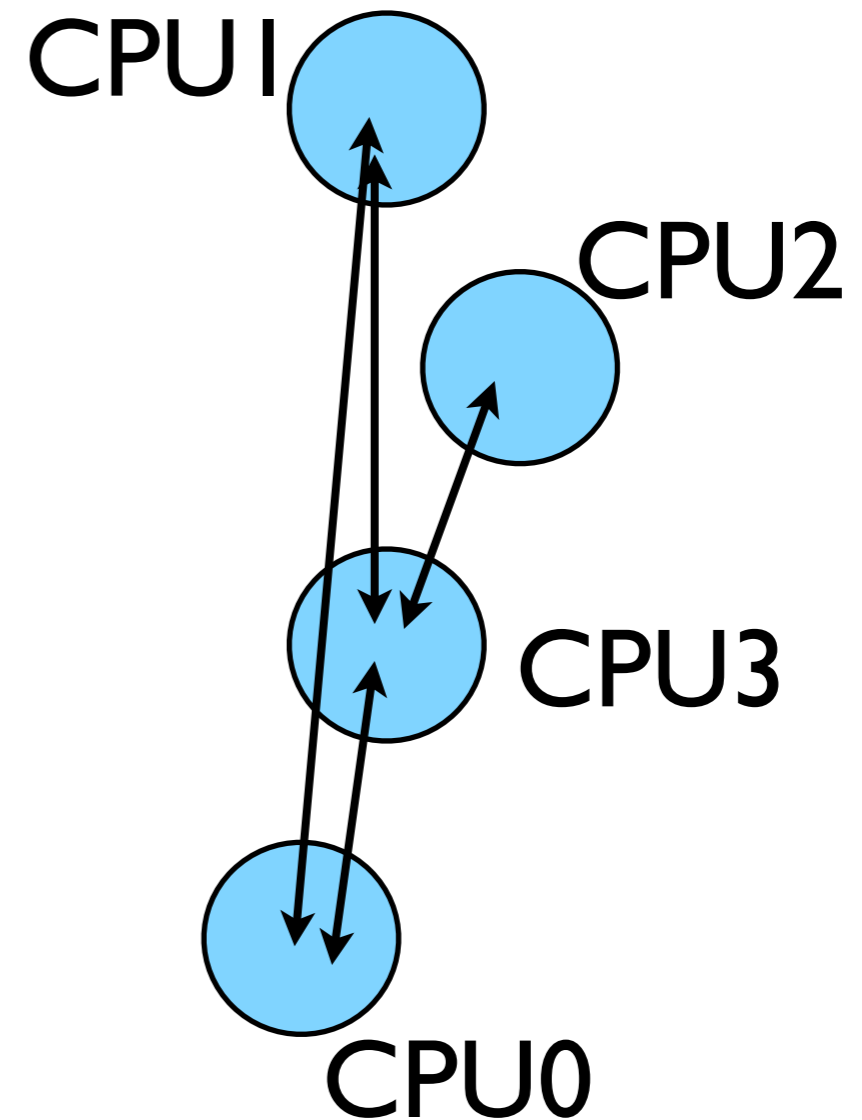
MPI is a Library for **Message-Passing**

- Communication/coordination between tasks done by sending and receiving messages.
- Each message involves a function call from each of the programs.



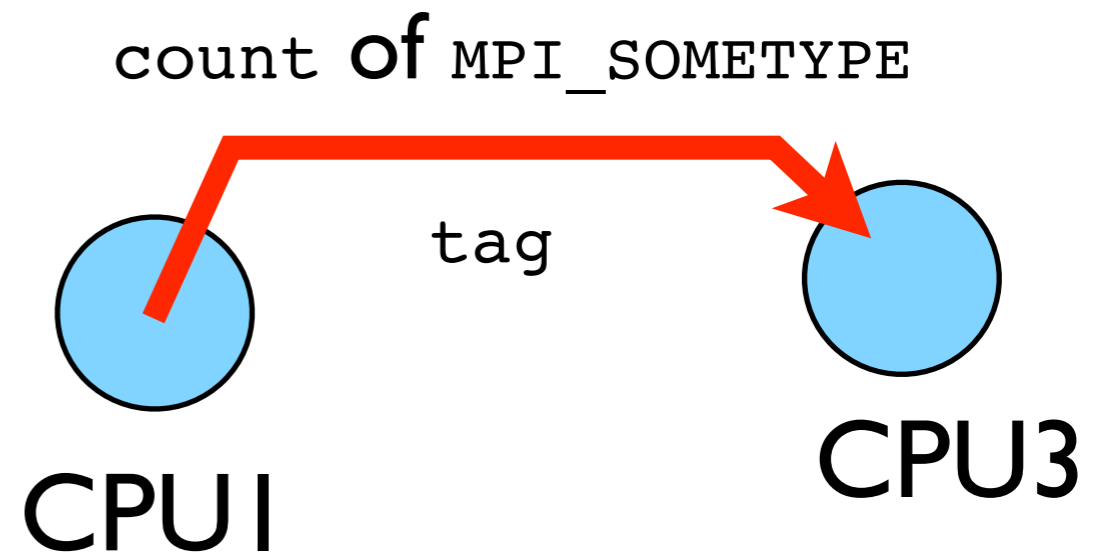
MPI is a Library for **Message-Passing**

- Three basic sets of functionality:
 - Pairwise communications via messages
 - Collective operations via messages
 - Efficient routines for getting data from memory into messages and vice versa



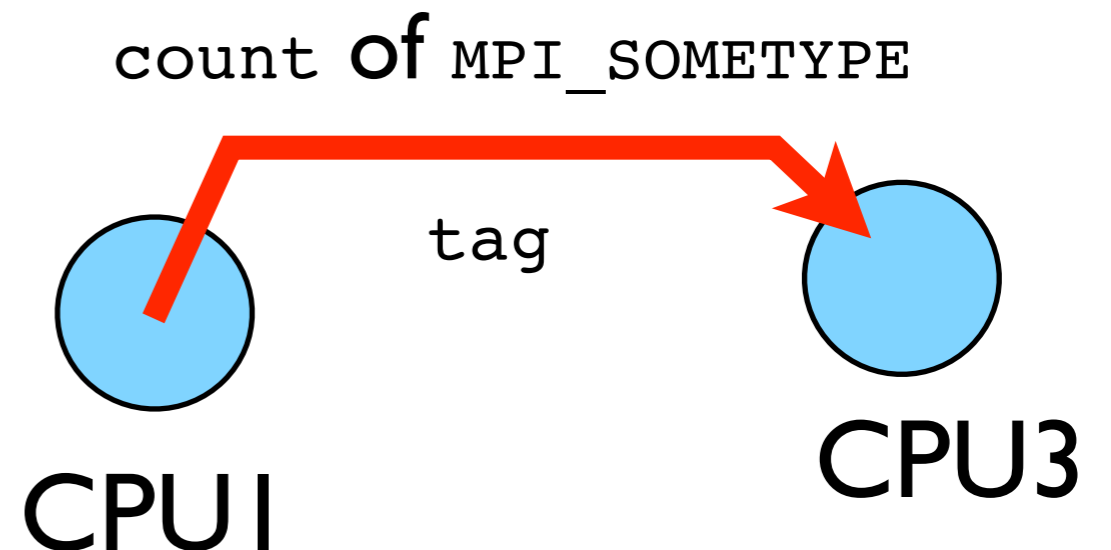
Messages

- Messages have a **sender** and a **receiver**
- When you are sending a message, don't need to specify sender (it's the current processor),
- A sent message has to be actively received by the receiving process



Messages

- MPI messages are a string of length **count** all of some fixed MPI **type**
- MPI types exist for characters, integers, floating point numbers, etc.
- An arbitrary non-negative integer **tag** is also included - helps keep things straight if lots of messages are sent.



Size of MPI Library

- Many, many functions (>200)
- Not nearly so many concepts
- We'll get started with just 10-12, use more as needed.

```
MPI_Init()  
MPI_Comm_size()  
MPI_Comm_rank()  
MPI_Ssend()  
MPI_Recv()  
MPI_Finalize()
```

Hello World

- The obligatory starting point
- `cd ~/parCFD/mpi-intro`
- Type it in, compile and run it

```
program helloworld
use mpi
implicit none
integer :: rank, comsize, ierr

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

print *, 'Hello world, from task ', rank, &
        ' of ', comsize

call MPI_Finalize(ierr)
end program helloworld
```

Fortran

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

int main(int argc, char **argv) {
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("Hello, world, from task %d of %d!\n",
           rank, size);

    MPI_Finalize();
    return 0;
}
```

C

edit hello-world.c or .f90

```
$ mpif90 hello-world.f90
-o hello-world
```

or

```
$ mpicc hello-world.c
-o hello-world
$ mpirun -np 1 hello-world
$ mpirun -np 2 hello-world
$ mpirun -np 8 hello-world
```

What mpicc/ mpif90 do

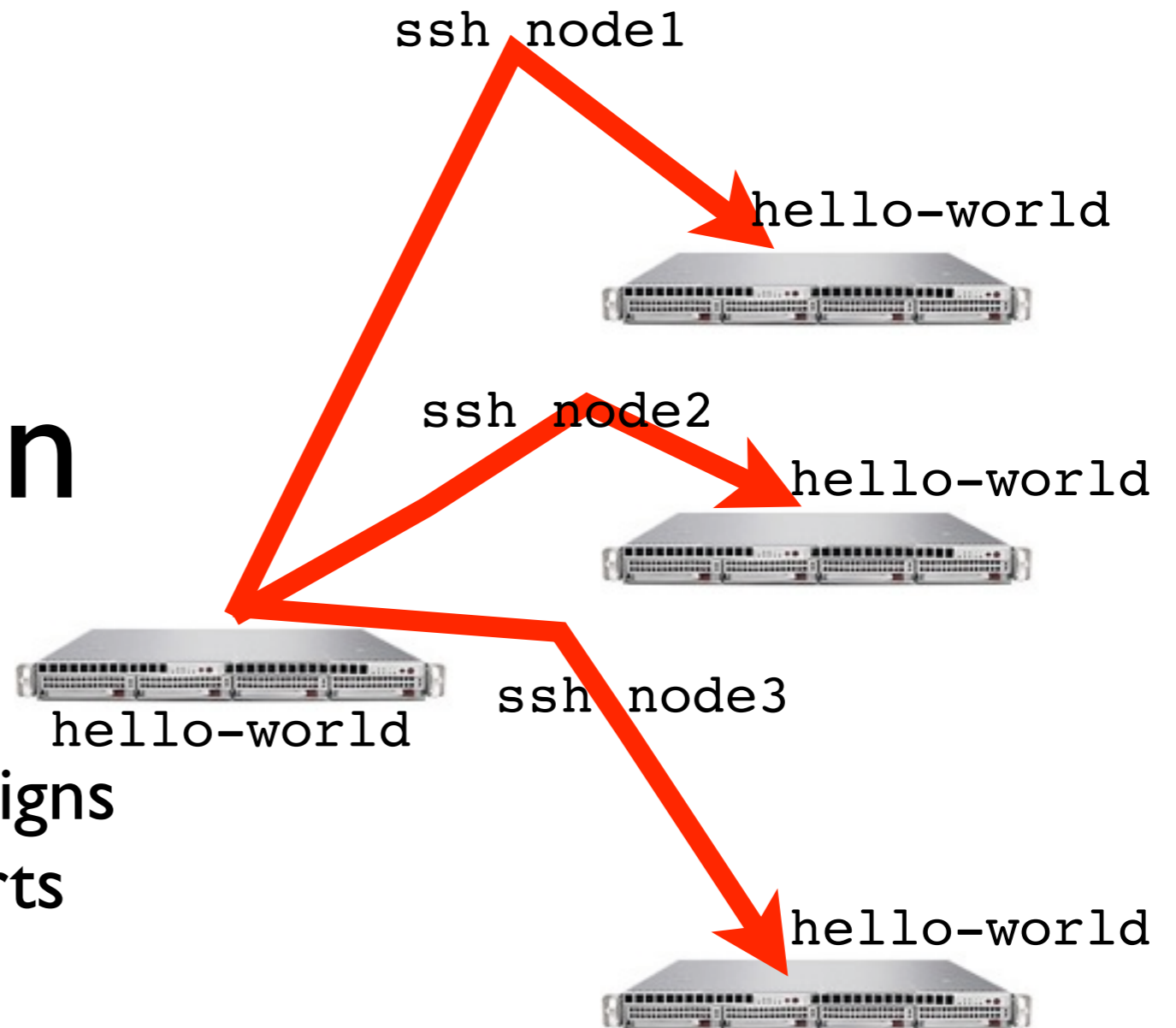
- Just wrappers for the system C, Fortran compilers that have the various -I, -L clauses in there automatically
- --showme (OpenMPI) shows which options are being used

```
$ mpicc --showme hello-world.c  
-o hello-world
```

```
gcc -I/usr/local/include  
-pthread hello-world.c -o  
hello-world -L/usr/local/lib  
-lmpi -lopen-rte -lopen-pal  
-ldl -Wl,--export-dynamic -lnsl  
-lutil -lm -ldl
```

What mpirun does

- Launches n processes, assigns each an MPI rank and starts the program
- For multinode run, has a list of nodes, ssh's to each node and launches the program



Number of Processes

- Number of processes to use is almost always equal to the number of processors
- But not necessarily.
- On your nodes, what happens when you run this?

```
$ mpirun -np 24 hello-world
```

mpirun runs *any* program

- mpirun will start that process-launching procedure for any program
- Sets variables somehow that mpi programs recognize so that they know which process they are

```
$ hostname  
$ mpirun -np 4 hostname  
$ ls  
$ mpirun -np 4 ls
```

What the code does

```
program helloworld
use mpi
implicit none
integer :: rank, comsize, ierr

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

print *, 'Hello world, from task ', rank, &
        ' of ', comsize

call MPI_Finalize(ierr)
end program helloworld
```

- (FORTRAN version; C is similar)

use mpi : imports declarations for MPI
function calls

```
program helloworld
use mpi
implicit none
integer :: rank, comsize, ierr

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

print *, 'Hello world, from task ', rank, &
        ' of ', comsize

call MPI_Finalize(ierr)
end program helloworld
```

call MPI_INIT(ierr):
initialization for MPI library.

Must come first.

ierr: Returns any error code.

call MPI_FINALIZE(ierr):
close up MPI stuff.

Must come last.

ierr: Returns any error code.



```
program helloworld
use mpi
implicit none
integer :: rank, comsize, ierr

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

print *, 'Hello world, from task ', rank, &
        ' of ', comsize

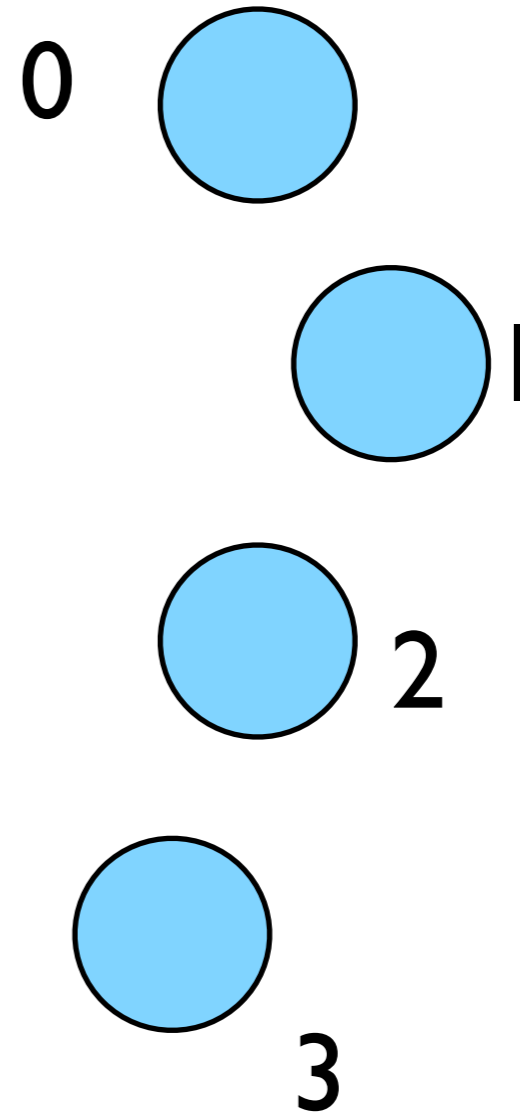
call MPI_Finalize(ierr)
end program helloworld
```

call MPI_COMM_RANK,
call MPI_COMM_SIZE:
requires a little more exposition.



Communicators

- MPI groups processes into communicators.
- Each communicator has some size -- number of tasks.
- Each task has a rank 0..size-1
- Every task in your program belongs to
`MPI_COMM_WORLD`

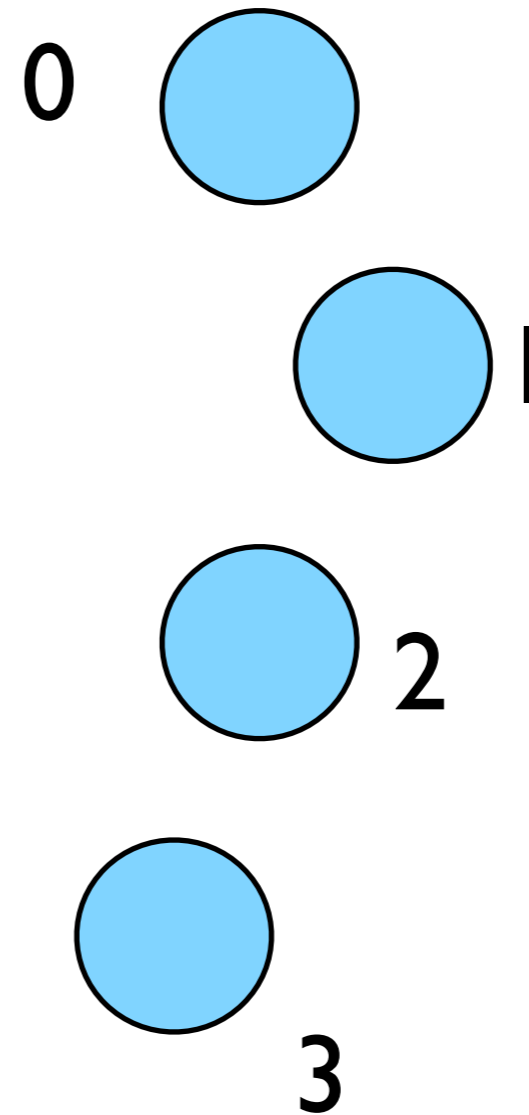


`MPI_COMM_WORLD:`
`size=4, ranks=0..3`

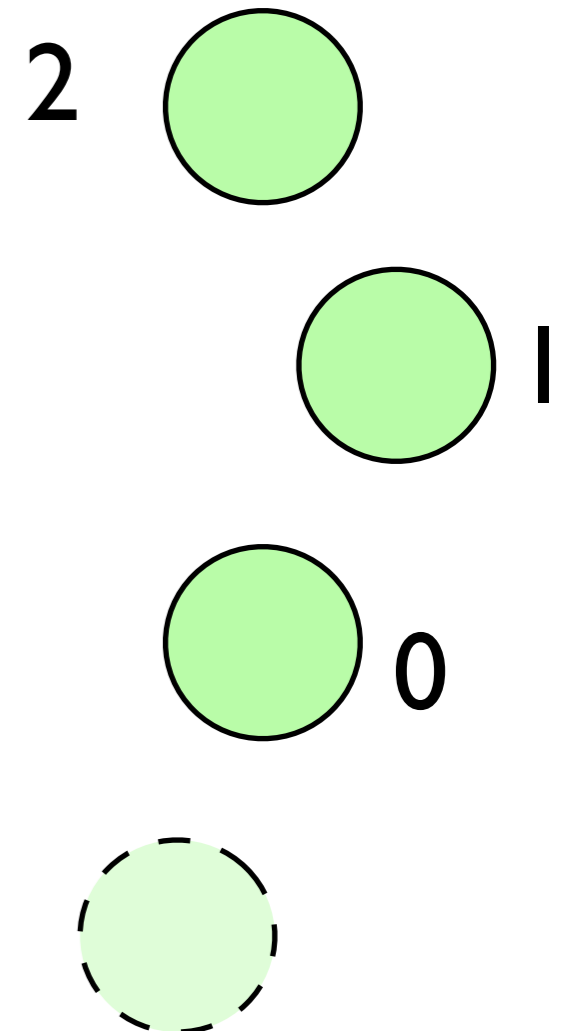
Communicators

- Can create our own communicators over the same tasks
- May break the tasks up into subgroups
- May just re-order them for some reason

MPI_COMM_WORLD:
size=4, ranks=0..3



new_comm
size=3, ranks=0..2



```
program helloworld
use mpi
implicit none
integer :: rank, comsize, ierr

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

print *, 'Hello world, from task ', rank, &
        ' of ', comsize

call MPI_Finalize(ierr)
end program helloworld
```

call MPI_COMM_RANK,
call MPI_COMM_SIZE:

get the size of communicator,
the current task's rank within
communicator.

put answers in rank and
size

C

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

int main(int argc, char **argv) {
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("Hello, world, from task %d of %d!\n",
           rank, size);

    MPI_Finalize();
    return 0;
}
```

Fortran

```
program helloworld
use mpi
implicit none
integer :: rank, comsize, ierr

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

print *, 'Hello world, from task ', rank, &
        ' of ', comsize

call MPI_Finalize(ierr)
end program helloworld
```

- #include <mpi.h> vs use mpi
- C - functions **return** ierr;
- Fortran - **pass** ierr
- MPI_Init

Our first real MPI program - but no Ms are P'ed!

- Let's fix this
- mpicc -o firstmessage firstmessage.c
- mpirun -np 2 ./firstmessage
- Note: C - MPI_CHAR

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

int main(int argc, char **argv) {
    int rank, size, ierr;
    int sendto, recvfrom; /* task to send, recv from */
    int ourtag=1; /* shared tag to label msgs*/
    char sendmessage[]="Hello"; /* text to send */
    char getmessage[6]; /* text to recieve */
    MPI_Status rstatus; /* MPI_Recv status info */

    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        sendto = 1;
        ierr = MPI_Ssend(sendmessage, 6, MPI_CHAR, sendto,
                        ourtag, MPI_COMM_WORLD);
        printf("%d: Sent message <%s>\n", rank, sendmessage);
    } else if (rank == 1) {
        recvfrom = 0;
        ierr = MPI_Recv(getmessage, 6, MPI_CHAR, recvfrom,
                        ourtag, MPI_COMM_WORLD, &rstatus);
        printf("%d: Got message <%s>\n", rank, getmessage);
    }
    ierr = MPI_Finalize();
    return 0;
}
```

Fortran version

- Let's fix this
- mpif90 -o
firstmessage
firstmessage.f90
- mpirun -np 2 ./
firstmessage
- FORTRAN -
MPI_CHARACTER

```
program firstmessage
use mpi
implicit none

integer :: rank, comsize, ierr
integer :: sendto, recvfrom ! Task to send, recv from
integer :: ourtag=1 ! shared tag to label msgs
character(5) :: sendmessage ! text to send
character(5) :: getmessage ! text rcvd
integer, dimension(MPI_STATUS_SIZE) :: rstatus

call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, ierr)

if (rank == 0) then
  sendmessage = 'Hello'
  sendto = 1
  call MPI_Ssend(sendmessage, 5, MPI_CHARACTER, sendto, &
                ourtag, MPI_COMM_WORLD, ierr)
  print *, rank, ' sent message <', sendmessage, '>'
else if (rank == 1) then
  recvfrom = 0
  call MPI_Recv(getmessage, 5, MPI_CHARACTER, recvfrom, &
               ourtag, MPI_COMM_WORLD, rstatus, ierr)
  print *, rank, ' got message <', getmessage, '>'
endif

call MPI_Finalize(ierr)
end program firstmessage
```

C - Send and Receive

```
MPI_Status status;
```

```
ierr = MPI_Ssend(sendptr, count, MPI_TYPE, destination,  
                tag, Communicator);
```

```
ierr = MPI_Recv(rcvptr, count, MPI_TYPE, source, tag,  
               Communicator, status);
```


Fortran - Send and Receive

```
integer status(MPI_STATUS_SIZE)

call MPI_SSEND(sendarr, count, MPI_TYPE, destination,
              tag, Communicator, ierr)

call MPI_RECV(rcvvarr, count, MPI_TYPE, source, tag,
             Communicator, status, ierr)
```

Special Source/Dest: MPI_PROC_NULL

`MPI_PROC_NULL` basically ignores the relevant operation; can lead to cleaner code.

Special Source: MPI_ANY_SOURCE

`MPI_ANY_SOURCE` is a wildcard; matches any source when receiving.

More complicated example:

- Let's look at secondmessage.f90, secondmessage.c

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

int main(int argc, char **argv) {
    int rank, size, ierr;
    int left, right;
    int tag=1;
    double msgsent, msgrcvd;
    MPI_Status rstatus;

    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    left = rank - 1;
    if (left < 0) left = MPI_PROC_NULL;
    right = rank + 1;
    if (right == size) right = MPI_PROC_NULL;

    msgsent = rank*rank;
    msgrcvd = -999;

    ierr = MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right,
                    tag, MPI_COMM_WORLD);
    ierr = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left,
                    tag, MPI_COMM_WORLD, &rstatus);

    printf("%d: Sent %lf and got %lf\n",
           rank, msgsent, msgrcvd);

    ierr = MPI_Finalize();
    return 0;
}
```

More complicated example:

- Let's look at secondmessage.f90, secondmessage.c

```
program secondmessage
use mpi
implicit none

integer :: ierr, rank, comsize
integer :: left, right
integer :: tag
integer :: status(MPI_STATUS_SIZE)
double precision :: msgsent, msgrcvd

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,comsize,ierr)

left = rank-1
if (left < 0) left = MPI_PROC_NULL
right = rank+1
if (right >= comsize) right = MPI_PROC_NULL

msgsent = rank*rank
msgrcvd = -999.
tag = 1

call MPI_Ssend(msgsent, 1, MPI_DOUBLE_PRECISION, right, &
               tag, MPI_COMM_WORLD, ierr)
call MPI_Recv(msgrcvd, 1, MPI_DOUBLE_PRECISION, left, &
              tag, MPI_COMM_WORLD, status, ierr)

print *, rank, 'Sent ', msgsent, 'and recvd ', msgrcvd

call MPI_FINALIZE(ierr)

end program secondmessage
```

Compile and run

- `mpi{cc,f90} -o secondmessage secondmessage.{c,f90}`
- `mpirun -np 4 ./secondmessage`

```
$ mpirun -np 4 ./secondmessage
3: Sent 9.000000 and got 4.000000
0: Sent 0.000000 and got -999.000000
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
```

```
$ mpirun -np 10 ./secondmessage
8: Sent 64.000000 and got 49.000000
9: Sent 81.000000 and got 64.000000
7: Sent 49.000000 and got 36.000000
6: Sent 36.000000 and got 25.000000
5: Sent 25.000000 and got 16.000000
3: Sent 9.000000 and got 4.000000
4: Sent 16.000000 and got 9.000000
0: Sent 0.000000 and got -999.000000
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
```

```

#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv) {
    int rank, size, ierr;
    int left, right;
    int tag=1;
    double msgsent, msgrcvd;
    MPI_Status rstatus;

    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    left = rank - 1;
    if (left < 0) left = MPI_PROC_NULL;
    right = rank + 1;
    if (right == size) right = MPI_PROC_NULL;

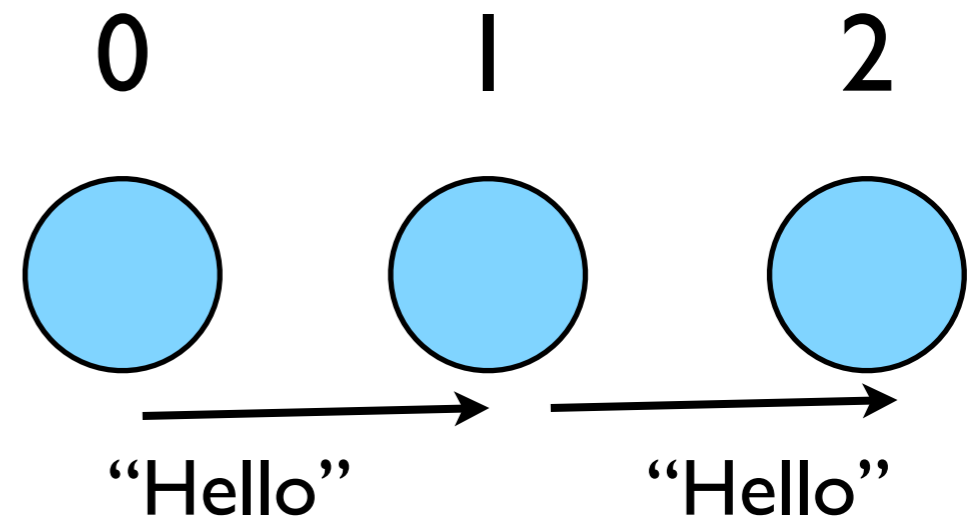
    msgsent = rank*rank;
    msgrcvd = -999;

    ierr = MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right,
                    tag, MPI_COMM_WORLD);
    ierr = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left,
                    tag, MPI_COMM_WORLD, &rstatus);

    printf("%d: Sent %lf and got %lf\n",
           rank, msgsent, msgrcvd);

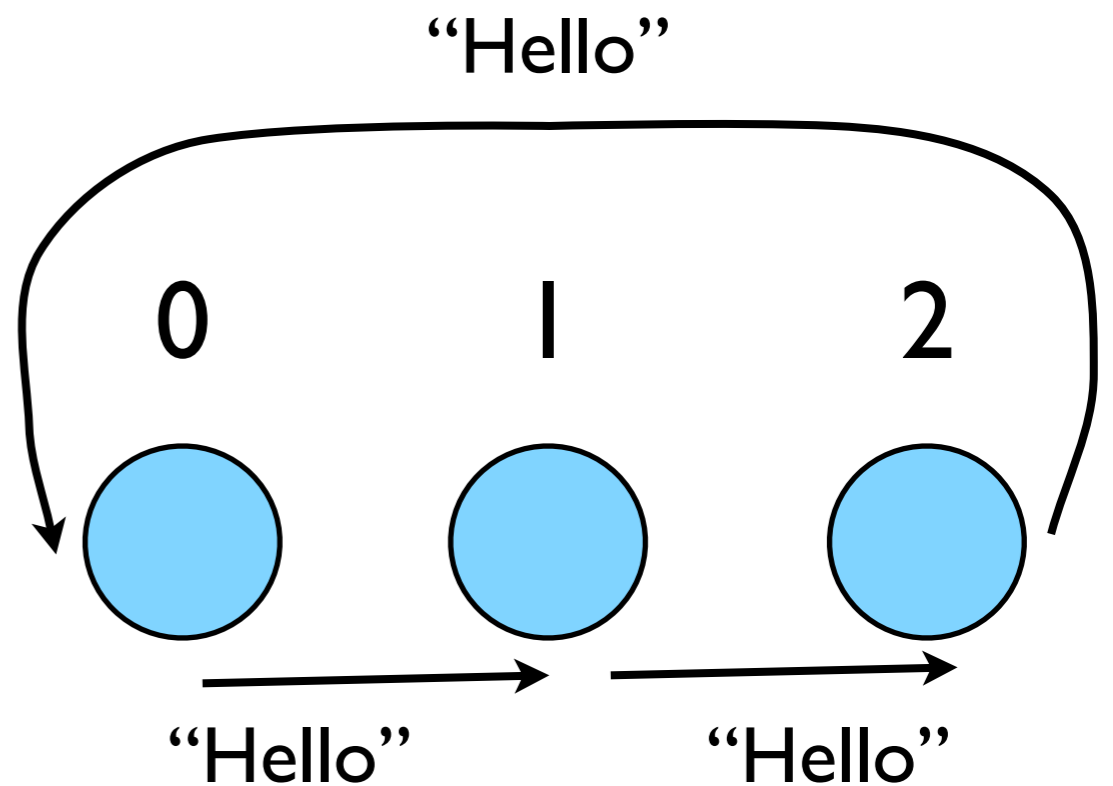
    ierr = MPI_Finalize();
    return 0;
}

```



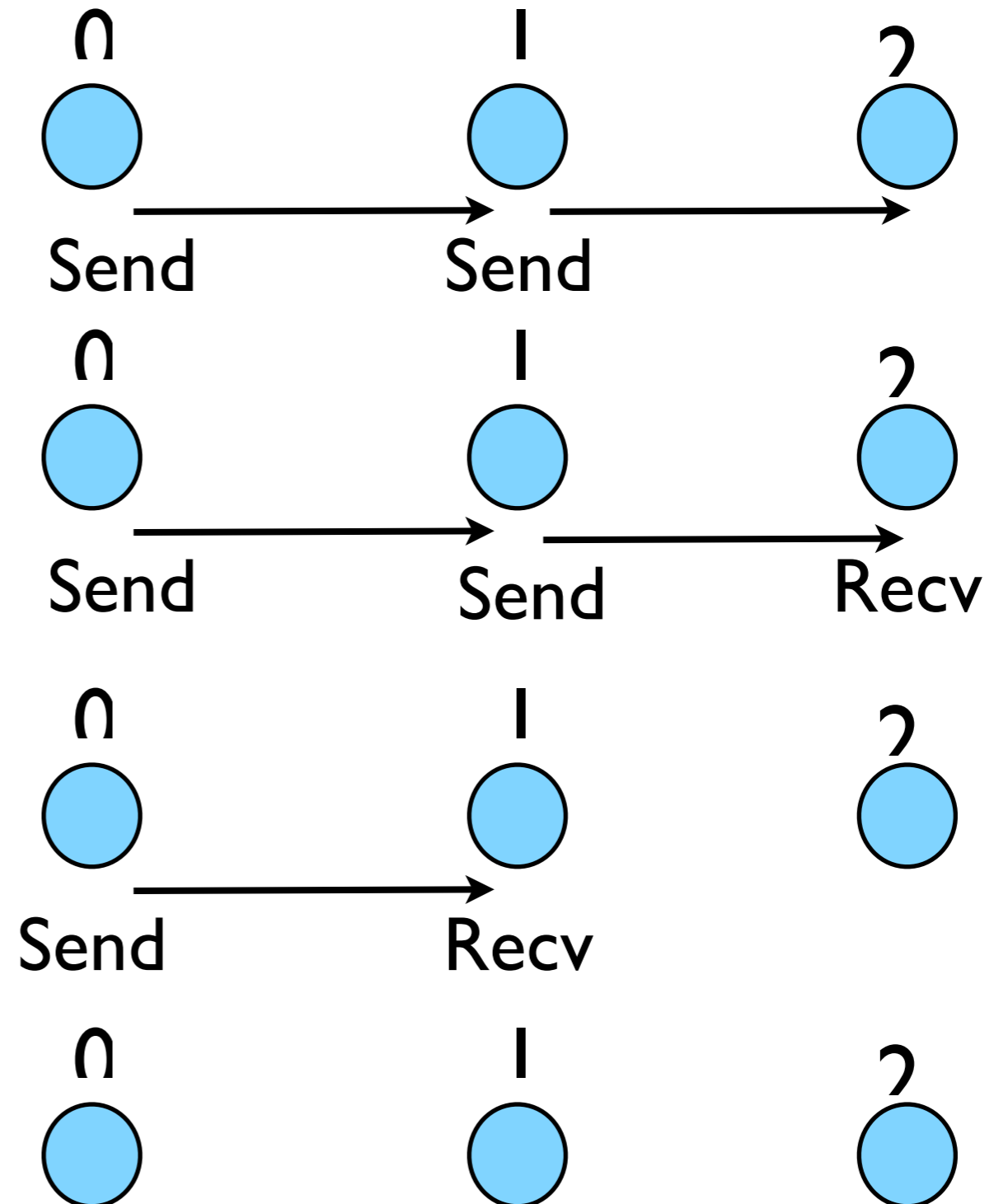
Implement periodic boundary conditions

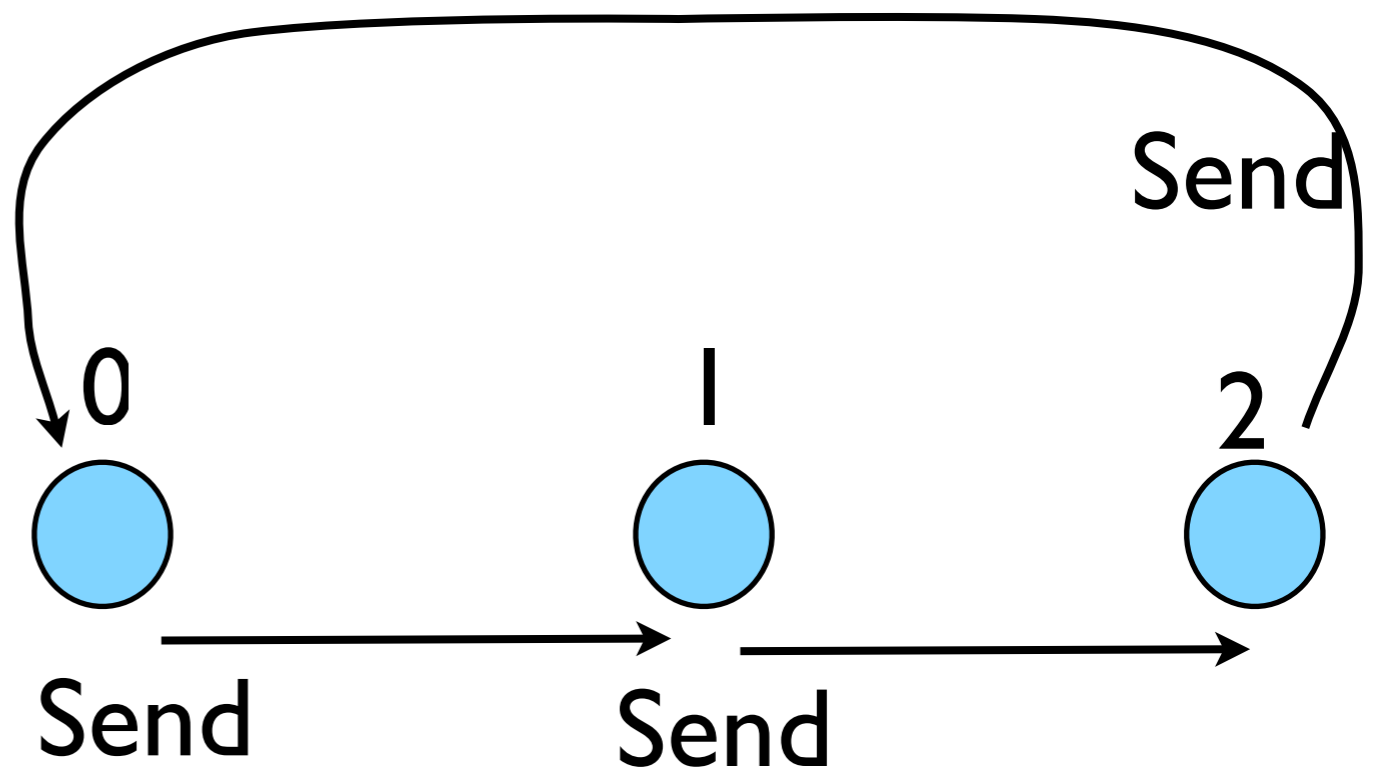
- `cp secondmessage.{c,f90}`
`thirdmessage.{c,f90}`
- edit so it `wraps around`
- `mpi{cc,f90} thirdmessage.`
`{c,f90} -o thirdmessage`
- `mpirun -np 3 thirdmessage`



```
left = rank-1
if (left < 0) left = comsize-1
right = rank+1
if (right >= comsize) right = 0
```

```
call MPI_Ssend(msgsent, 1, MPI_DOUBLE_PRECISION, right, &
              tag, MPI_COMM_WORLD, ierr)
call MPI_Recv(msgrcvd, 1, MPI_DOUBLE_PRECISION, left, &
             tag, MPI_COMM_WORLD, status, ierr)
```





```

left = rank-1
if (left < 0) left = comsize-1
right = rank+1
if (right >= comsize) right = 0

```

```

call MPI_Ssend(msgsent, 1, MPI_DOUBLE_PRECISION, right, &
tag, MPI_COMM_WORLD, ierr)
call MPI_Recv(msgrcvd, 1, MPI_DOUBLE_PRECISION, left, &
tag, MPI_COMM_WORLD, status, ierr)

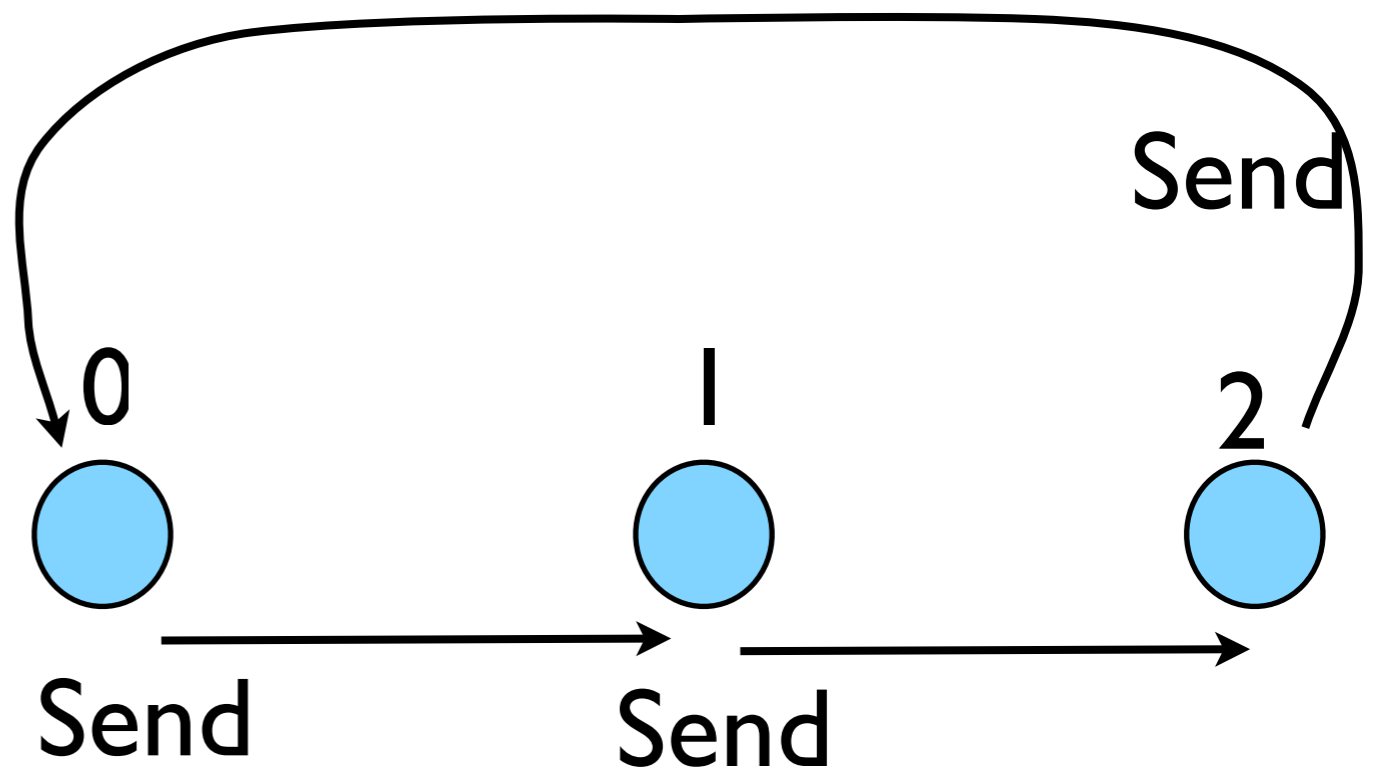
```



0,1,2

Deadlock

- A classic parallel bug
- Occurs when a cycle of tasks are for the others to finish.
- Whenever you see a closed cycle, you likely have (or risk) deadlock.



Big MPI

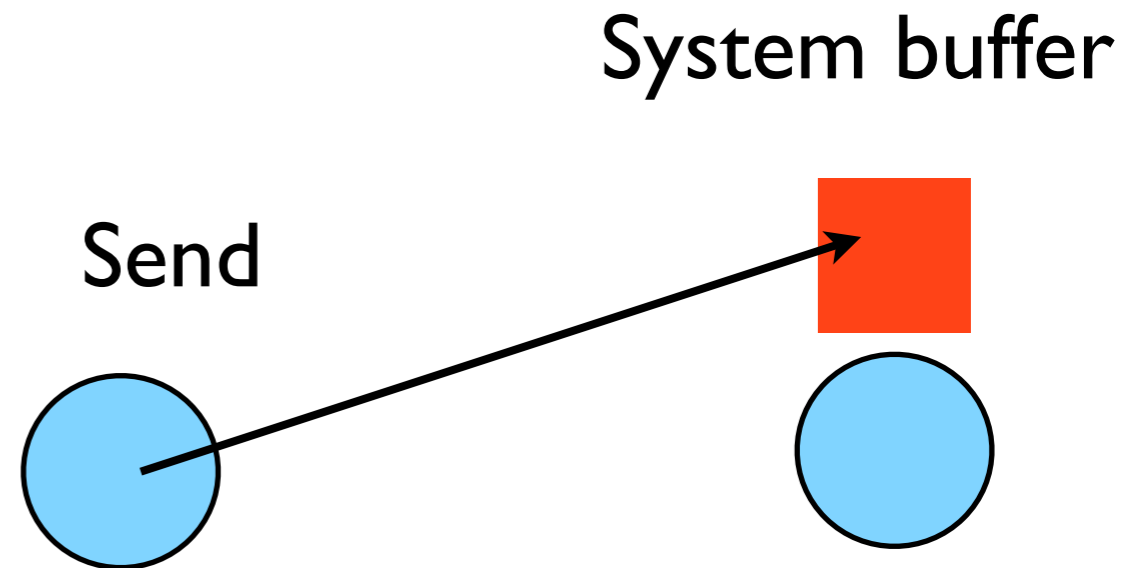
Lesson #1

All sends and receives must be paired, **at time of sending**

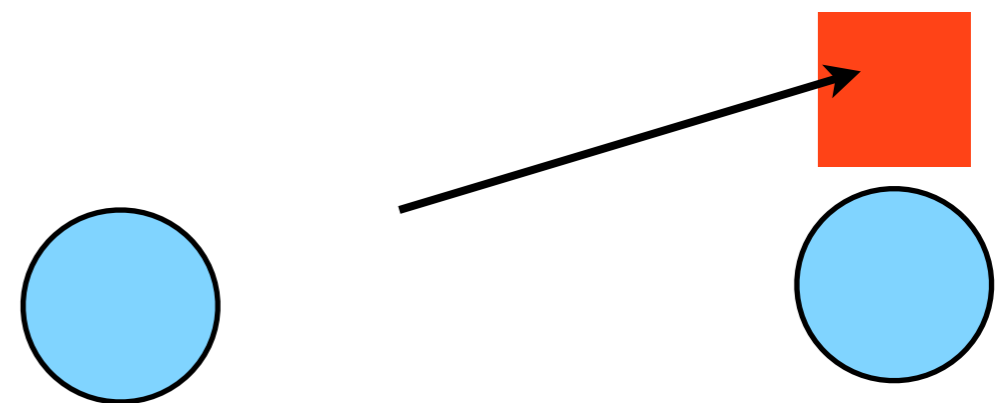
Different versions of SEND

- SSEND: safe send; doesn't return until receive has started. Blocking, no buffering.
- SEND: Undefined. Blocking, probably buffering
- ISEND : Unblocking, no buffering
- IBSEND: Unblocking, buffering

Buffering



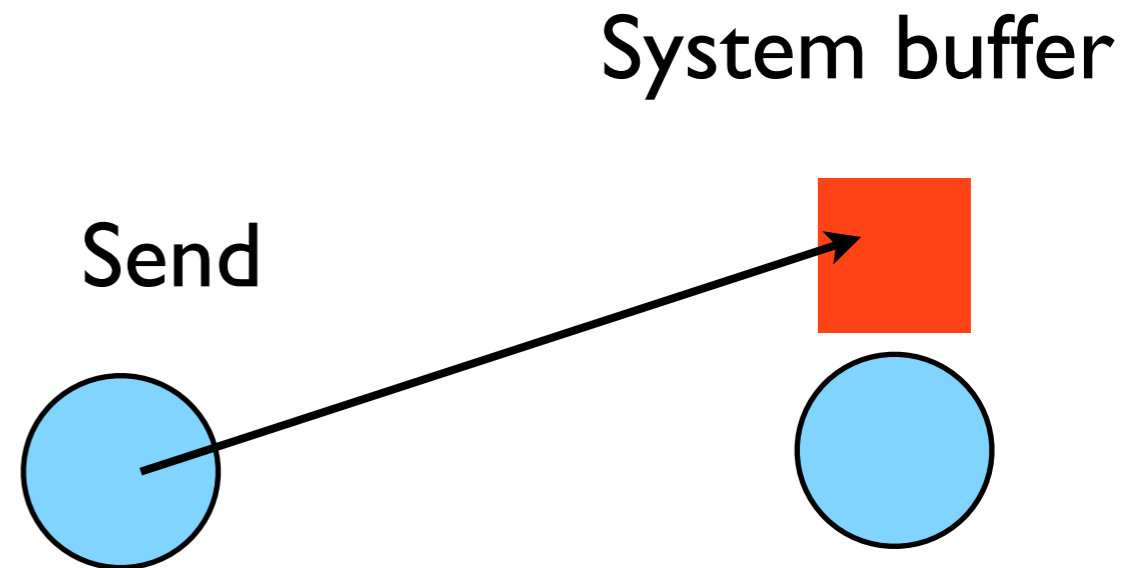
(Non) Blocking



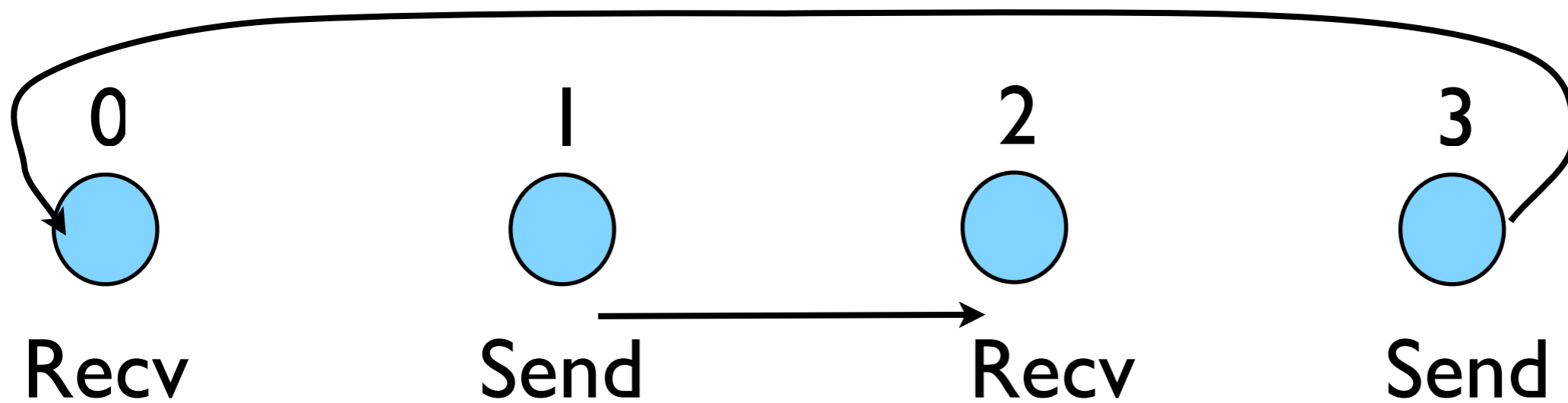
Buffering is dangerous!

- Worst kind of danger: will usually work.
- Think voice mail; message sent, reader reads when ready
- But voice mail boxes do fill
- Message fails.
- Program fails/hangs mysteriously.
- (Can allocate your own buffers)

Buffering



**Without using new MPI
routines, how can we fix
this?**



- First: evens send, odds receive
- Then: odds send, evens receive
- Will this work with an odd # of processes?
- How about 2? 1?

```

program fourthmessage
implicit none
include 'mpif.h'

integer :: ierr, rank, comsize
integer :: left, right
integer :: tag
integer :: status(MPI_STATUS_SIZE)
double precision :: msgsent, msgrcvd

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,comsize,ierr)

left = rank-1
if (left < 0) left = comsize-1
right = rank+1
if (right >= comsize) right = 0

msgsent = rank*rank
msgrcvd = -999.
tag = 1

if (mod(rank,2) == 0) then
  call MPI_Ssend(msgsent, 1, MPI_DOUBLE_PRECISION, right, &
               tag, MPI_COMM_WORLD, ierr)
  call MPI_Recv(msgrcvd, 1, MPI_DOUBLE_PRECISION, left, &
               tag, MPI_COMM_WORLD, status, ierr)
else
  call MPI_Recv(msgrcvd, 1, MPI_DOUBLE_PRECISION, left, &
               tag, MPI_COMM_WORLD, status, ierr)
  call MPI_Ssend(msgsent, 1, MPI_DOUBLE_PRECISION, right, &
               tag, MPI_COMM_WORLD, ierr)
endif
print *, rank, 'Sent ', msgsent, 'and recvd ', msgrcvd

call MPI_FINALIZE(ierr)

end program fourthmessage

```

Evens send first



Then odds



fourthmessage.f90


```

#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv) {
    int rank, size, ierr;
    int left, right;
    int tag=1;
    double msgsent, msgrcvd;
    MPI_Status rstatus;

    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    left = rank - 1;
    if (left < 0) left = size-1;
    right = rank + 1;
    if (right == size) right = 0;

    msgsent = rank*rank;
    msgrcvd = -999;

    if (rank % 2 == 0) {
        ierr = MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right,
                        tag, MPI_COMM_WORLD);
        ierr = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left,
                       tag, MPI_COMM_WORLD, &rstatus);
    } else {
        ierr = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left,
                       tag, MPI_COMM_WORLD, &rstatus);
        ierr = MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right,
                        tag, MPI_COMM_WORLD);
    }

    printf("%d: Sent %lf and got %lf\n",
           rank, msgsent, msgrcvd);

    ierr = MPI_Finalize();
    return 0;
}

```

Evens send first



Then odds



fourthmessage.c

Something new: Sendrecv

- A blocking send and receive built in together
- Lets them happen simultaneously
- Can automatically pair the sends/recvs!
- dest, source does not have to be same; nor do types or size.

```
program fifthmessage
implicit none
include 'mpif.h'

integer :: ierr, rank, comsize
integer :: left, right
integer :: tag
integer :: status(MPI_STATUS_SIZE)
double precision :: msgsent, msgrcvd

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,comsize,ierr)

left = rank-1
if (left < 0) left = comsize-1
right = rank+1
if (right >= comsize) right = 0

msgsent = rank*rank
msgrcvd = -999.
tag = 1

call MPI_Sendrecv(msgsent, 1, MPI_DOUBLE_PRECISION, right, tag, &
                  msgrcvd, 1, MPI_DOUBLE_PRECISION, left, tag, &
                  MPI_COMM_WORLD, status, ierr)
print *, rank, 'Sent ', msgsent, 'and recvd ', msgrcvd

call MPI_FINALIZE(ierr)

end program fifthmessage
```

fifthmessage.f90

Something new: Sendrecv

- A blocking send and receive built in together
- Lets them happen simultaneously
- Can automatically pair the sends/recvs!
- dest, source does not have to be same; nor do types or size.

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

int main(int argc, char **argv) {
    int rank, size, ierr;
    int left, right;
    int tag=1;
    double msgsent, msgrcvd;
    MPI_Status rstatus;

    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    left = rank - 1;
    if (left < 0) left = size-1;
    right = rank + 1;
    if (right == size) right = 0;

    msgsent = rank*rank;
    msgrcvd = -999;

    ierr = MPI_Sendrecv(&msgsent, 1, MPI_DOUBLE, right, tag,
                        &msgrcvd, 1, MPI_DOUBLE, left, tag,
                        MPI_COMM_WORLD, &rstatus);

    printf("%d: Sent %lf and got %lf\n",
           rank, msgsent, msgrcvd);

    ierr = MPI_Finalize();
    return 0;
}
```

fifthmessage.c

Sendrecv = Send + Recv

C syntax

```
MPI_Status status;
```

Send Args

```
ierr = MPI_Sendrecv(sendptr, count, MPI_TYPE, destination, tag,  
recvptr, count, MPI_TYPE, source, tag,  
Communicator, &status);
```

Recv Args

FORTRAN syntax

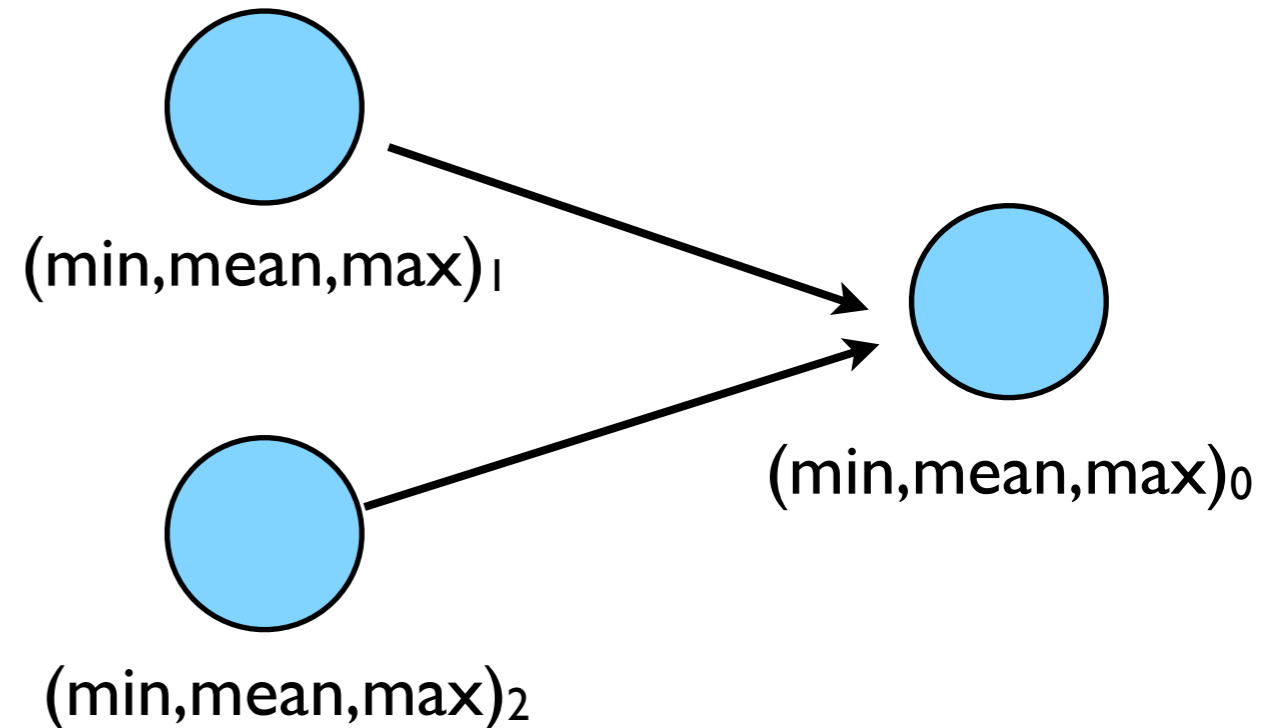
```
integer status(MPI_STATUS_SIZE)
```

```
call MPI_SENDRECV(sendptr, count, MPI_TYPE, destination, tag,  
recvptr, count, MPI_TYPE, source, tag,  
Communicator, status, ierr)
```

Why are there two different tags/types/counts?

Min, Mean, Max of numbers

- Lets try some code that calculates the min/mean/max of a bunch of random numbers $-1..1$. Should go to $-1, 0, +1$ for large N .
- Each gets their partial results and sends it to some node, say node 0 (why node 0?)
- `minmeanmax.{c,f90}`
- How to MPI it?



```

program randomdata
implicit none
integer,parameter :: nx=1500
real, allocatable :: dat(:)

integer :: i
real    :: datamin, datamax, datamean

!
! random data
!
allocate(dat(nx))
call srand(0)
do i=1,nx
  dat(i) = 2*rand(0)-1.
enddo

!
! find min/mean/max
!
datamin = 1e+19
datamax = -1e+19
datamean = 0.

do i=1,nx
  if (dat(i) .lt. datamin) datamin = dat(i)
  if (dat(i) .ge. datamax) datamax = dat(i)
  datamean = datamean + dat(i)
enddo
datamean = datamean/(1.*nx)
deallocate(dat)

print *, 'min/mean/max = ', datamin, datamean, datamax

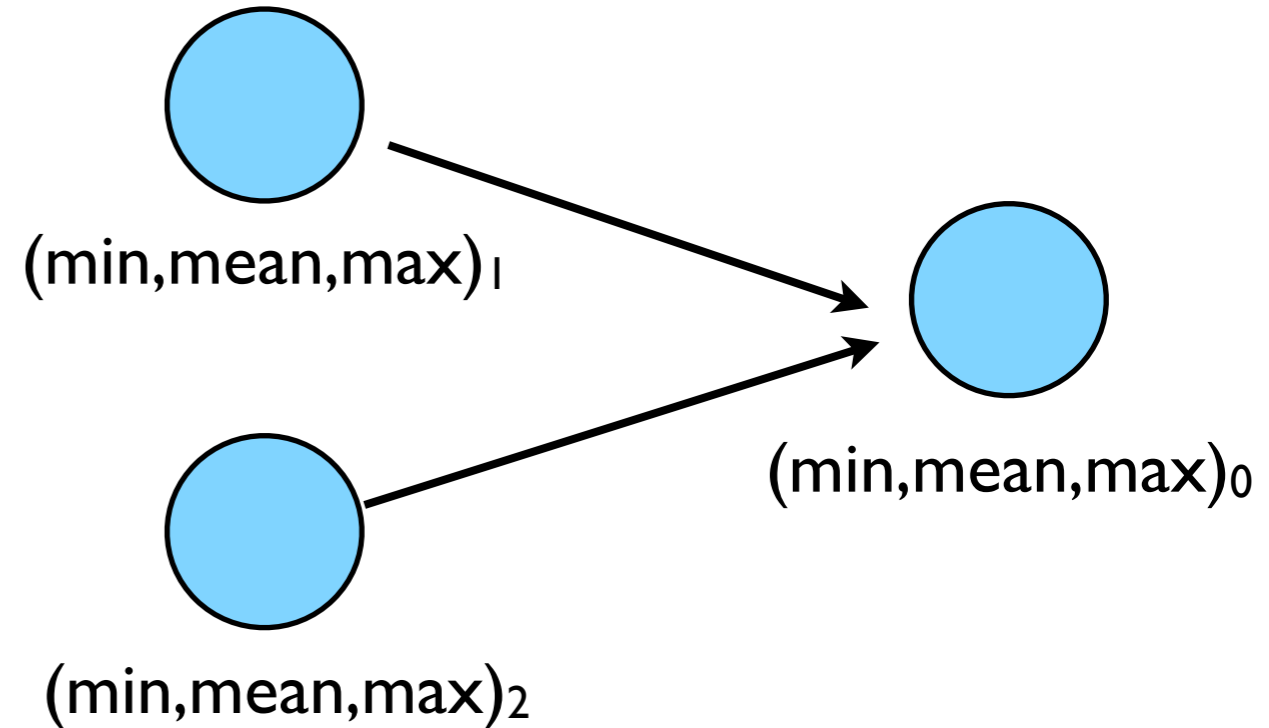
return
end

```

```

if (rank /= 0) then
  sendbuffer(1) = datamin
  sendbuffer(2) = datamean
  sendbuffer(3) = datamax
  call MPI_SSEND(sendbuffer, 3, MPI_REAL, 0, &
    ourtag, MPI_COMM_WORLD, ierr)
else
  do i=1,comsize-1
    call MPI_RECV(recvbuffer, 3, MPI_REAL, MPI_ANY_SOURCE, &
      ourtag, MPI_COMM_WORLD, status, ierr)
    if (recvbuffer(1) < datamin) datamin=recvbuffer(1)
    if (recvbuffer(3) > datamax) datamax=recvbuffer(3)
    datamean = datamean + recvbuffer(2)
  enddo
  datamean = datamean / comsize
endif
deallocate(dat)

```

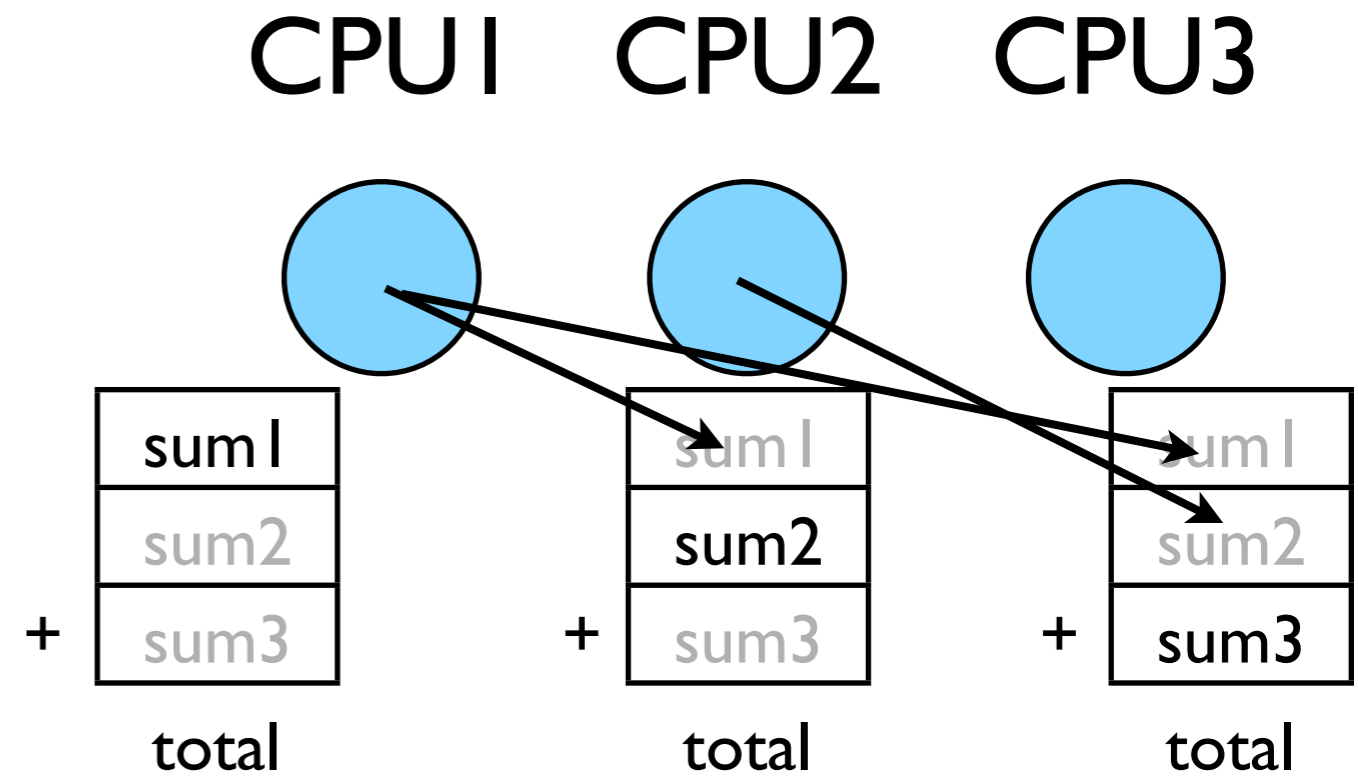


Q: are these sends/recvd adequately paired?

minmeanmax-mpi.f90

Inefficient!

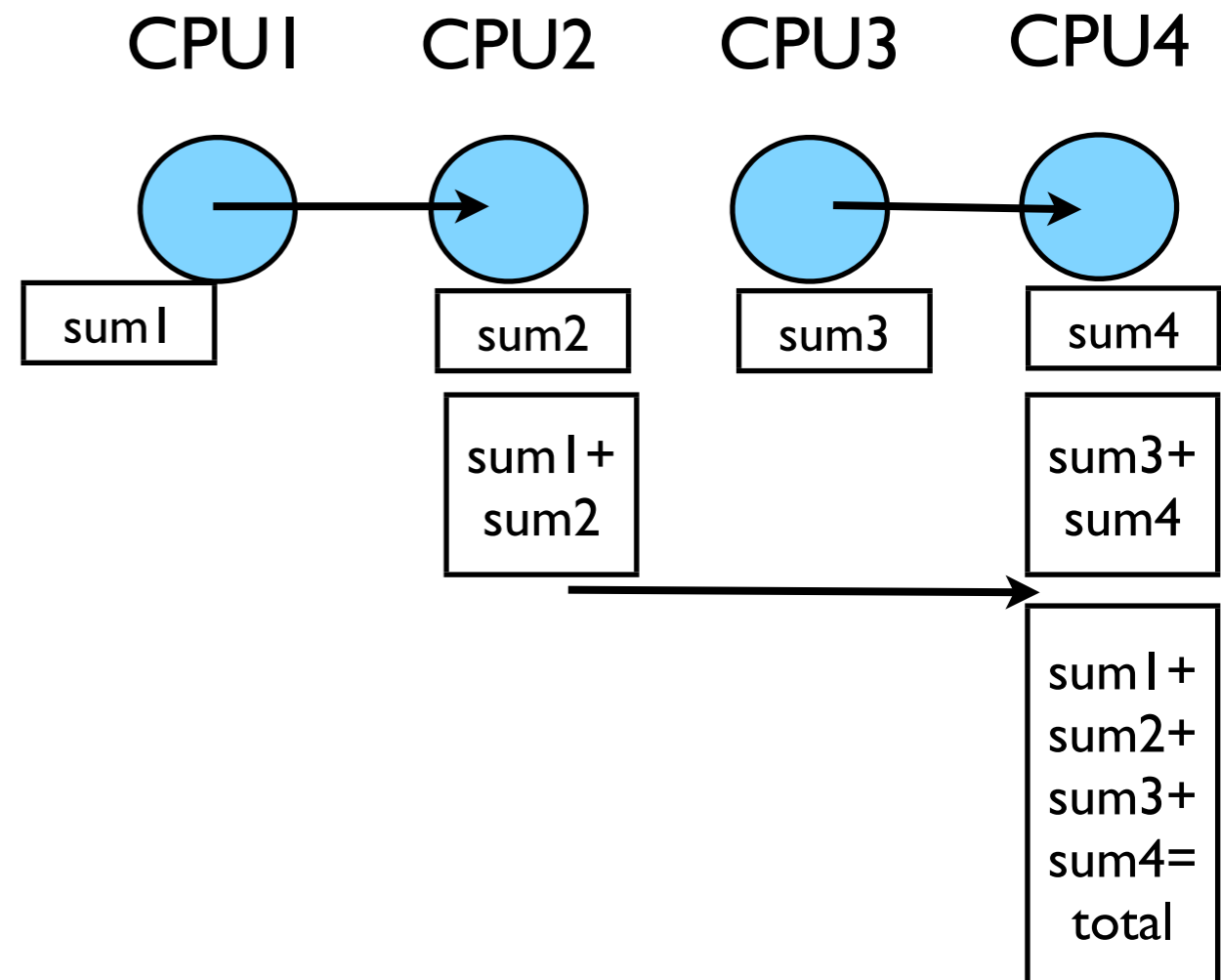
- Requires $(P-1)$ messages, $2(P-1)$ if everyone then needs to get the answer.



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...)

```

do i=1,nx
  if (dat(i) .lt. datamin) datamin = dat(i)
  if (dat(i) .ge. datamax) datamax = dat(i)
  datamean = datamean + dat(i)
enddo
datamean = datamean/(1.*nx)

call MPI_Allreduce(datamin, globalmin, 1, MPI_REAL, &
                  MPI_MIN, MPI_COMM_WORLD, ierr)
!
! to just sent to rank 0:
! call MPI_Reduce(datamin, globalmin, 1, MPI_REAL, &
!               MPI_MIN, 0, MPI_COMM_WORLD, ierr)
!
call MPI_Allreduce(datamax, globalmax, 1, MPI_REAL, &
                  MPI_MAX, MPI_COMM_WORLD, ierr)
call MPI_Allreduce(datamean, globalmean, 1, MPI_REAL, &
                  MPI_SUM, MPI_COMM_WORLD, ierr)

globalmean = globalmean/comsize

print *, 'min/mean/max = ', datamin, datamean, datamax

if (rank==0) then
  print *, 'global min/mean/max = ', globalmin, globalmean, globalmax
endif

```

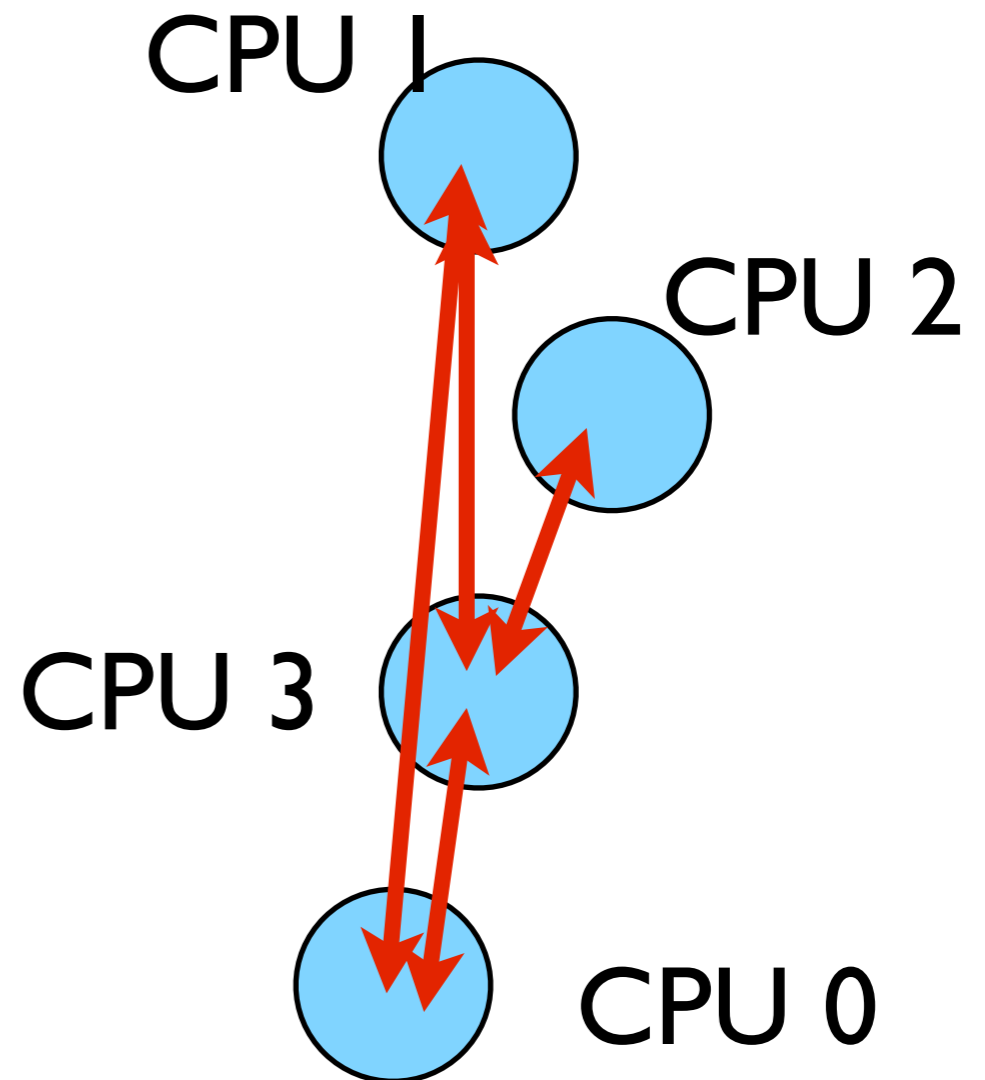
MPI_Reduce and MPI_Allreduce

Performs a reduction
and sends answer to
one PE (Reduce)
or all PEs (Allreduce)

minmeanmax-allreduce.f

Collective Operations

- As opposed to the pairwise messages we've seen
- **All** processes in the communicator must participate
- Cannot proceed until all have participated
- Don't necessarily know what goes on 'under the hood'



C syntax

```
MPI_Status status;
```

```
ierr = MPI_Init(&argc, &argv);
```

```
ierr = MPI_Comm_{size,rank}(Communicator, &{size,rank});
```

```
ierr = MPI_Send(sendptr, count, MPI_TYPE, destination,  
                tag, Communicator);
```

```
ierr = MPI_Recv(rcvptr, count, MPI_TYPE, source, tag,  
                Communicator, &status);
```

```
ierr = MPI_Sendrecv(sendptr, count, MPI_TYPE, destination, tag,  
                    rcvptr, count, MPI_TYPE, source, tag,  
                    Communicator, &status);
```

```
ierr = MPI_Allreduce(&mydata, &globaldata, count, MPI_TYPE,  
                    MPI_OP, Communicator);
```

Communicator -> MPI_COMM_WORLD

status -> MPI_Status

MPI_Type -> MPI_FLOAT, MPI_DOUBLE, MPI_INT, MPI_CHAR...

MPI_OP -> MPI_SUM, MPI_MIN, MPI_MAX, ...

FORTRAN syntax

```
integer status(MPI_STATUS_SIZE)
```

```
call MPI_INIT(ierr)
```

```
call MPI_COMM_{SIZE,RANK}(Communicator, {size,rank},ierr)
```

```
call MPI_SSEND(sendarr, count, MPI_TYPE, destination,  
              tag, Communicator, ierr)
```

```
call MPI_RECV(rcvvarr, count, MPI_TYPE, destination,tag,  
             Communicator, status, ierr)
```

```
call MPI_SENDRECV(sendptr, count, MPI_TYPE, destination,tag,  
                 recvptr, count, MPI_TYPE, source, tag,  
                 Communicator, status, ierr)
```

```
call MPI_ALLREDUCE(&mydata, &globaldata, count, MPI_TYPE,  
                 MPI_OP, Communicator, ierr)
```

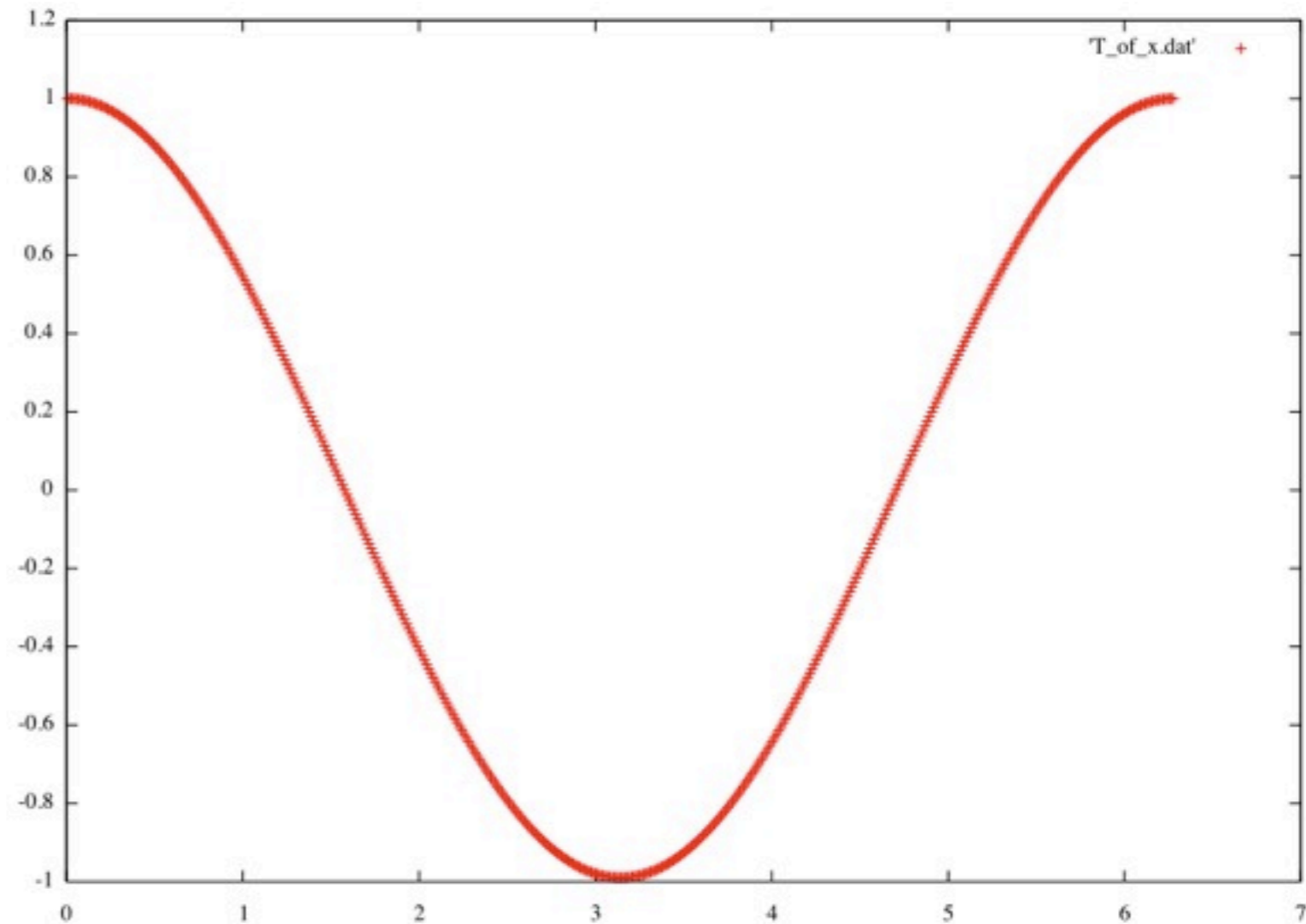
Communicator -> MPI_COMM_WORLD

status -> integer(MPI_STATUS_SIZE)

MPI_Type -> MPI_REAL, MPI_DOUBLE_PRECISION,
 MPI_INTEGER, MPI_CHARACTER

MPI_OP -> MPI_SUM, MPI_MIN, MPI_MAX,...

1d diffusion equation

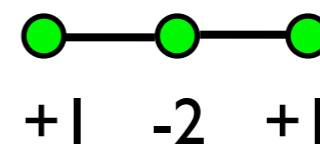
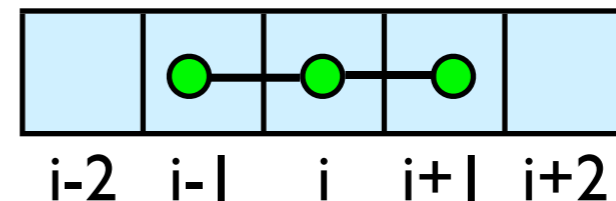


```
$ cd ~/parCFD/heateqn  
$ make  
$ ./heateqn --method=1 --out="out-serial-GS.txt"  
--log="log-serial-GS.txt"  
$ gnuplot  
gnuplot> plot 'out-serial-GS.txt'
```

Discretizing Derivatives

- Done by finite differencing the discretized values
- Implicitly or explicitly involves interpolating data and taking derivative of the interpolant
- More accuracy - larger 'stencils'

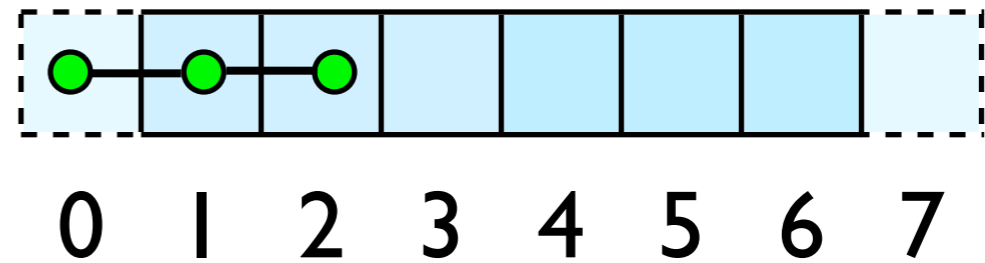
$$\left. \frac{d^2 Q}{dx^2} \right|_i \approx \frac{Q_{i+1} - 2Q_i + Q_{i-1}}{\Delta x^2}$$



Guardcells

- How to deal with boundaries?
- Because stencil juts out, need information on cells beyond those you are updating
- Pad domain with 'guard cells' so that stencil works even for the first point in domain
- Fill guard cells with values such that the required boundary conditions are met

Global Domain



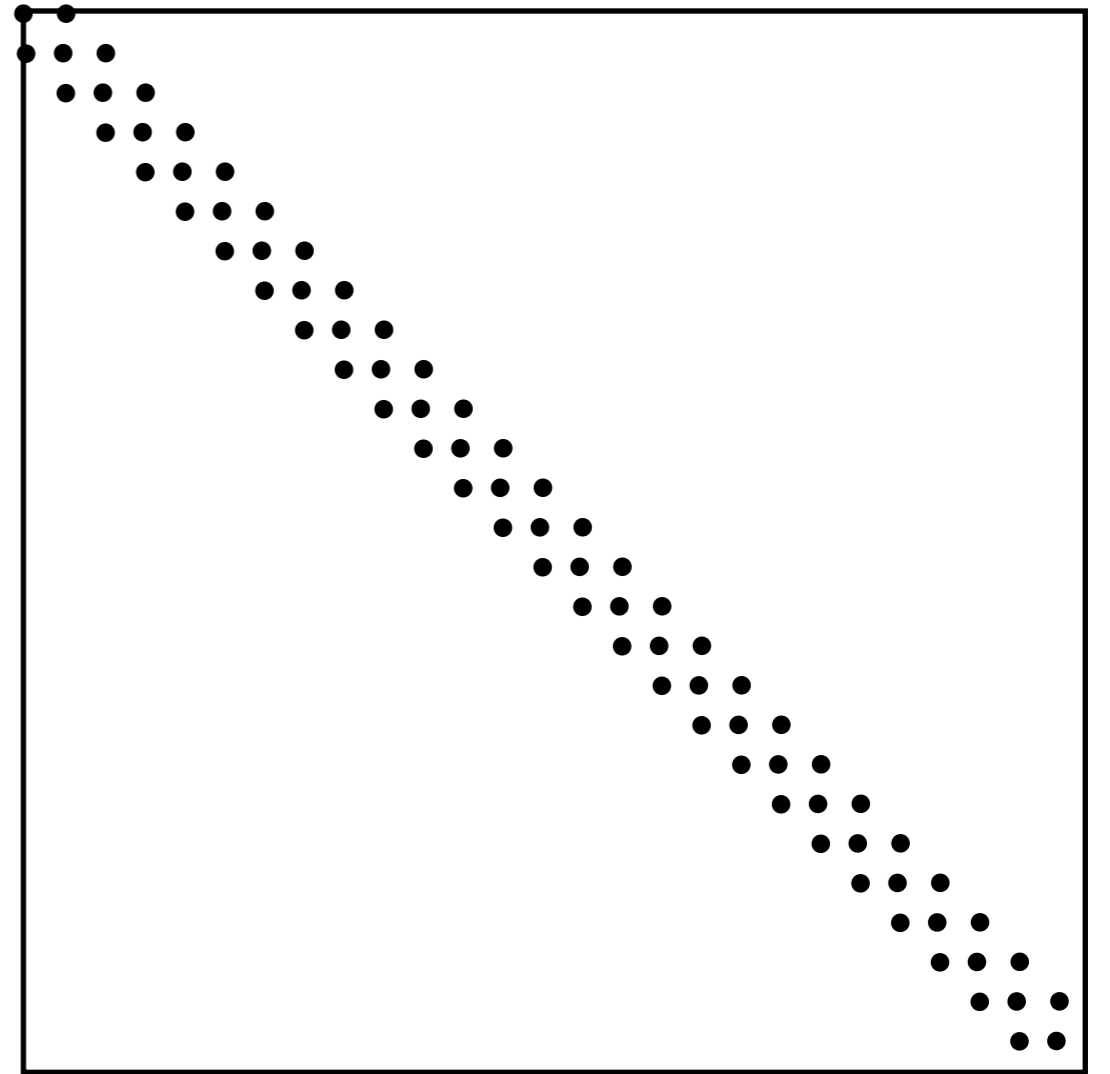
$$ng = 1$$

loop from ng , $N - 2 \cdot ng$

Solution of Linear System

- Gauss Seidel (method=1)

$$x_i = \frac{1}{d_i} (RHS_i - l_i x_{i-1} - r_i x_{i+1})$$



```

void gaussSeidel(int n, double tol, double *l, double *d, double *r,
                double *rhs, double *x, const int maxiters, int *iters,
                double *errors, double *times, double Tleft, double Tright) {

```

```

    double err;
    struct timeval start;

    tick(&start);
    /* initial guess: */
    for (int i=1; i<=n; i++)
        x[i] = rhs[i]/d[i];

    *iters = 0;
    err = 1.e+9;
    do {
        /* update x */
        for (int i=1; i<=n; i++) {
            x[i] = (rhs[i] - l[i]*x[i-1] - r[i]*x[i+1])/d[i];
        }

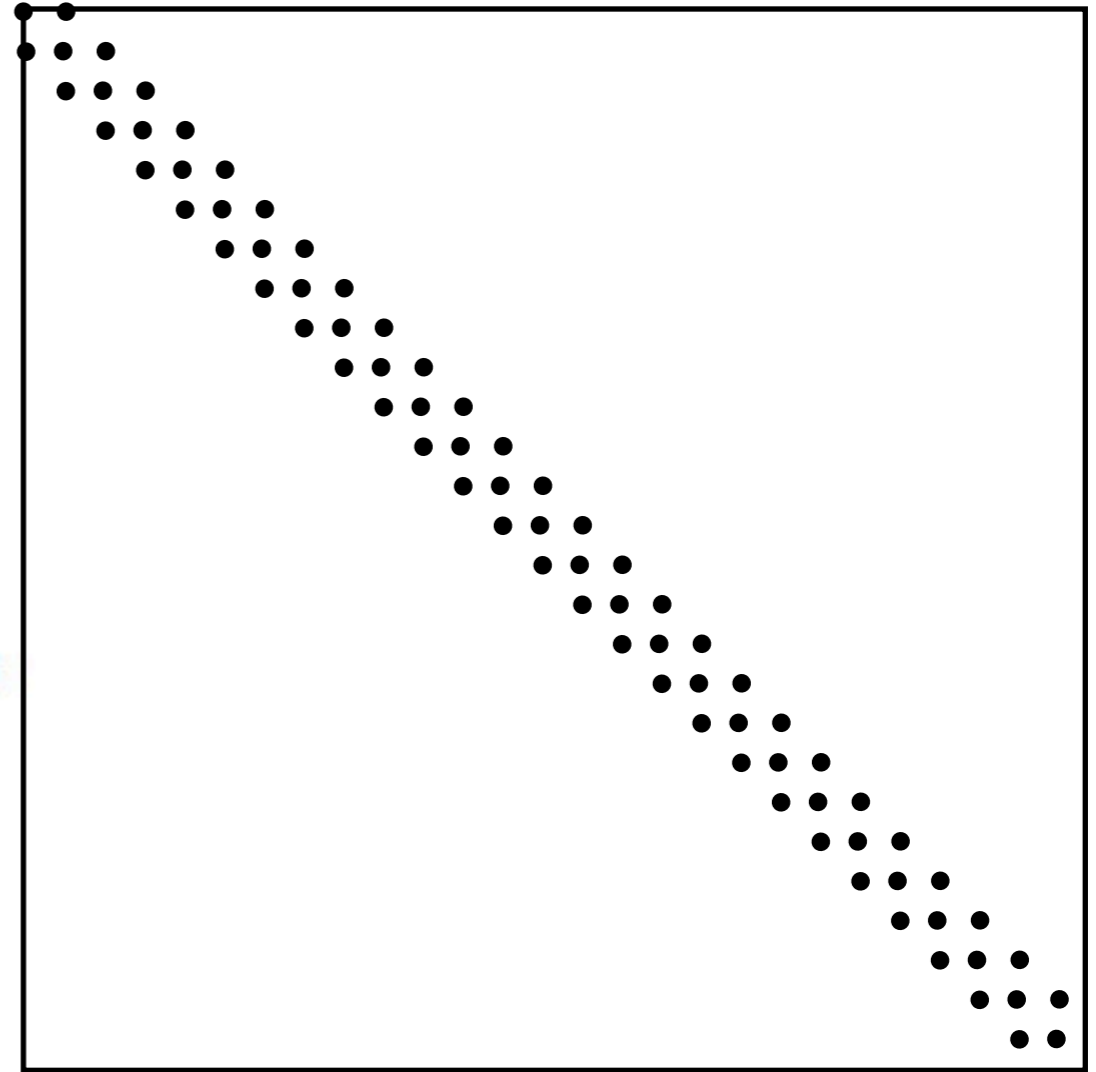
        boundaryConditions(n,x,Tleft,Tright);

        /* update RMS error */
        err = rmsResidual(n,x,l,d,r,rhs);

        errors[*iters] = err;
        times[*iters] = tock(&start);
        (*iters)++;
    } while (err > tol && *iters < maxiters);

    return;
}

```

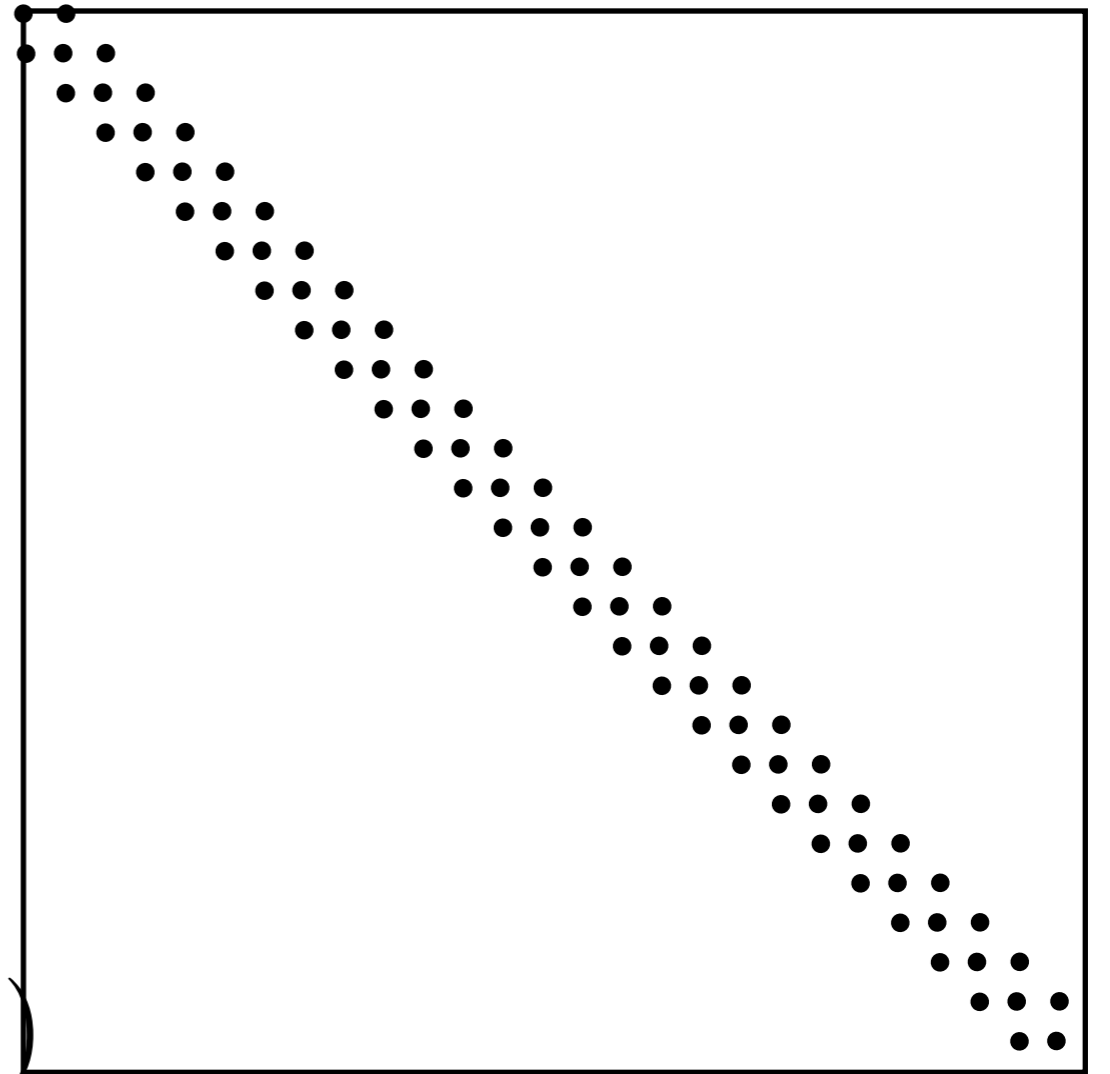


Solution of Linear System

- Jacobi (method=3)

$$x_i^{\text{new}} = \frac{1}{d_i} \left(RHS_i - l_i x_{i-1}^{\text{old}} - r_i x_{i+1}^{\text{old}} \right)$$

- doesn't use partial results
- slower (2x) convergence.



```

void jacobi(int n, double tol, double *l, double *d, double *r,
           double *rhs, double *x, const int maxiters, int *iters,
           double *errors, double *times, double Tleft, double Tright) {

    double err;
    double *work, *xold, *xnew;
    struct timeval start;

    work = (double *)malloc((n+2)*sizeof(double));
    xnew = work;
    xold = x;

    /* initial guess: */
    for (int i=1; i<=n; i++)
        xold[i] = rhs[i]/d[i];

    *iters = 0;
    err = 1.e+9;
    tick(&start);
    do {
        /* update x */
        for (int i=1; i<=n; i++) {
            xnew[i] = (rhs[i] - l[i]*xold[i-1] - r[i]*xold[i+1])/d[i];
        }

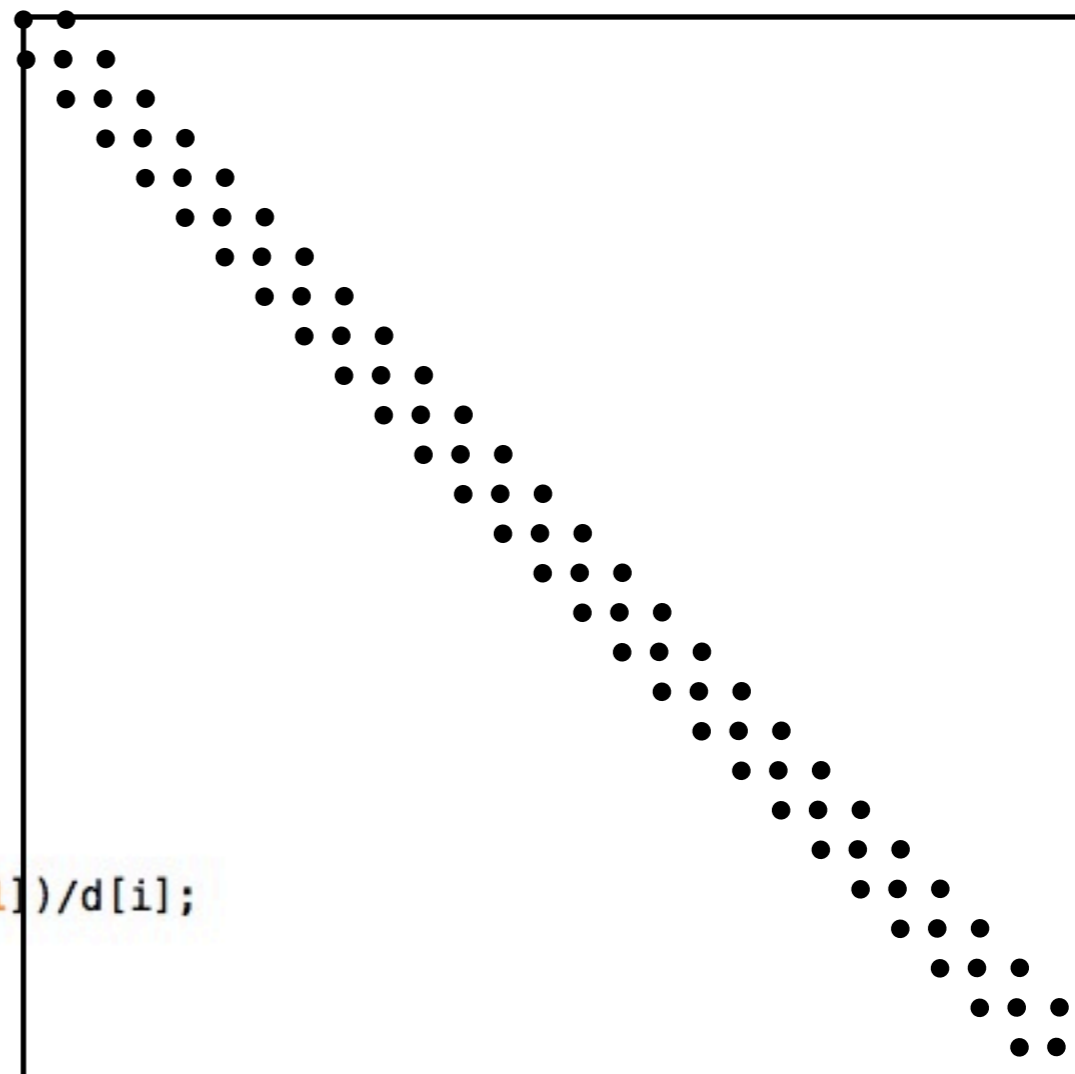
        boundaryConditions(n,xnew,Tleft,Tright);

        /* update RMS error */
        err = rmsResidual(n,xnew,l,d,r,rhs);

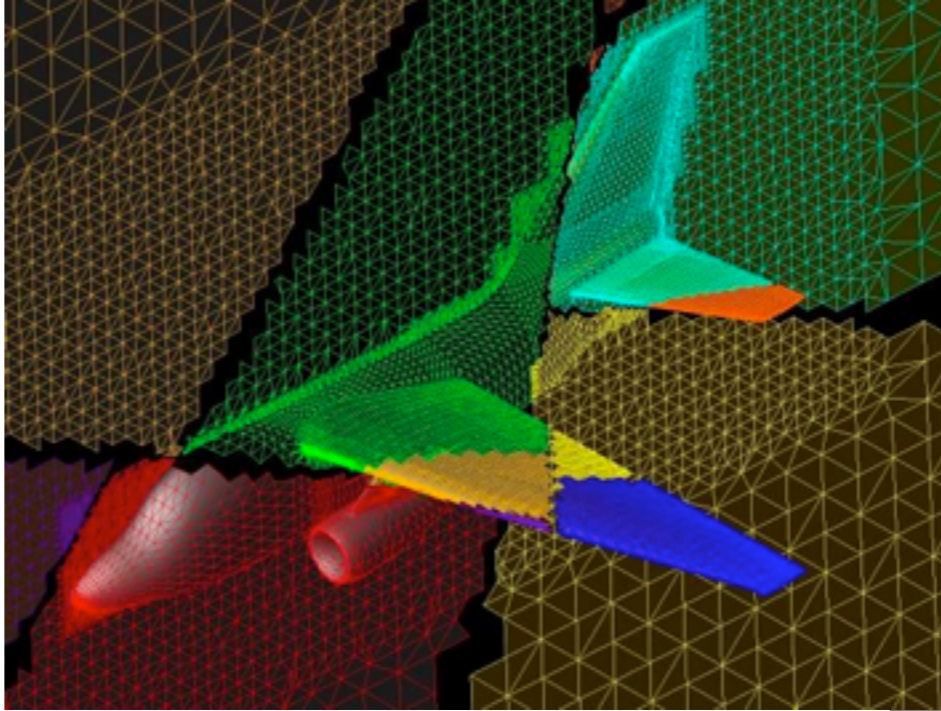
        errors[*iters] = err;
        times[*iters] = tock(&start);
        (*iters)++;

        double *p = xold;
        xold = xnew;
        xnew = p;
    } while (err > tol && *iters < maxiters);
}

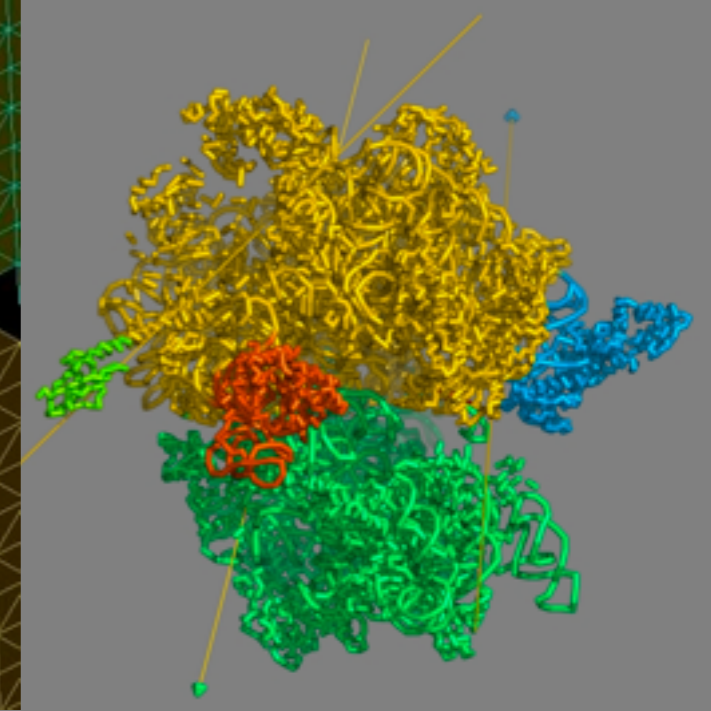
```



Domain Decomposition

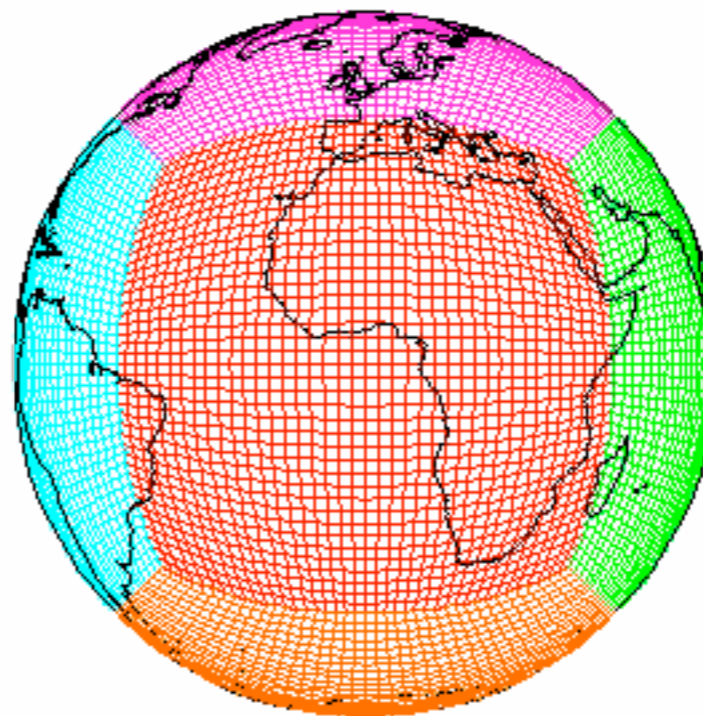


<http://adg.stanford.edu/aa241/design/compaero.html>

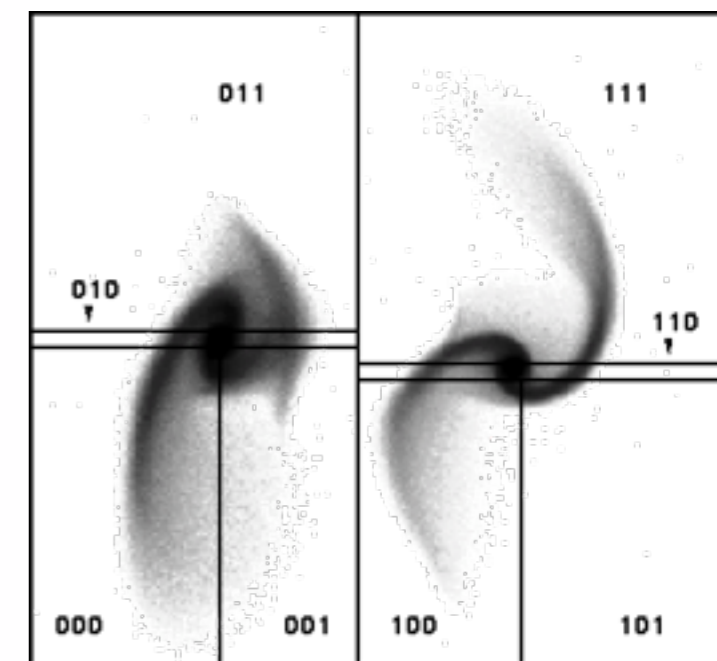


<http://www.uea.ac.uk/cmp/research/cmpbio/Protein+Dynamics,+Structure+and+Function>

- A very common approach to parallelizing on distributed memory computers
- Maintain Locality; need local data mostly, this means only surface data needs to be sent between processes.



http://sivo.gsfc.nasa.gov/cubedsphere_comp.html

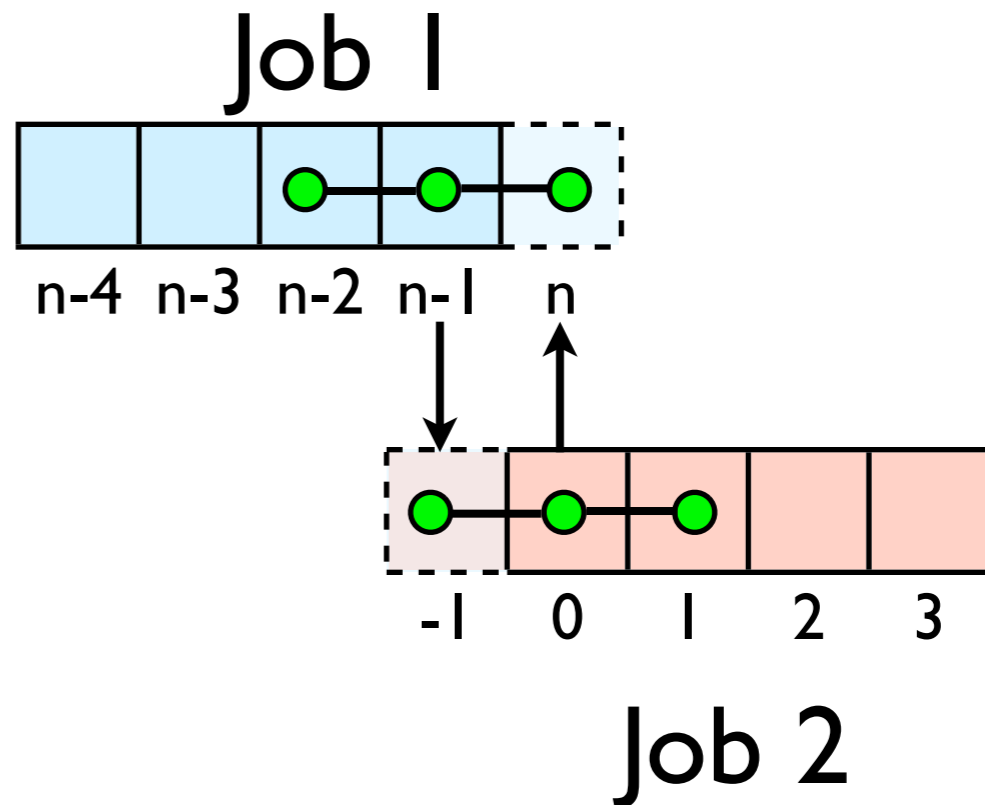
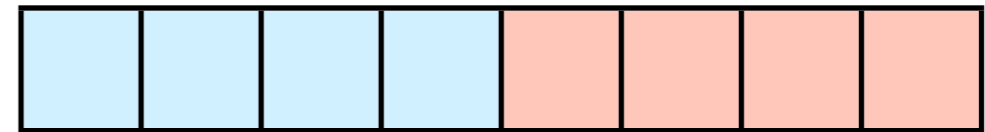


<http://www.cita.utoronto.ca/~dubinski/treecode/node8.html>

Guardcells

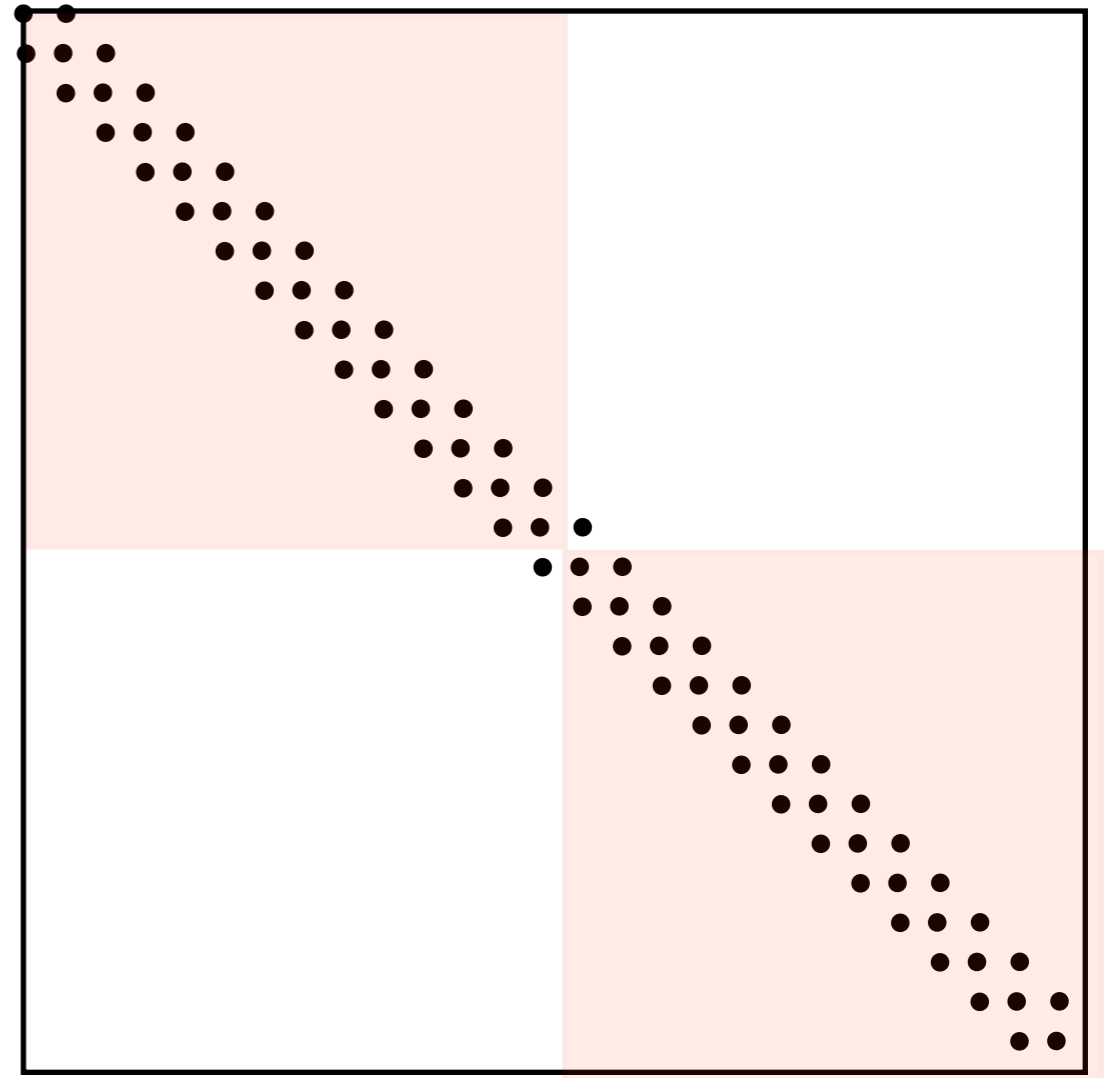
- Works for parallel decomposition!
- Job 1 needs info on Job 2s 0th zone, Job 2 needs info on Job 1s last zone
- Pad array with 'guardcells' and fill them with the info from the appropriate node by message passing or shared memory

Global Domain



Decomposition:

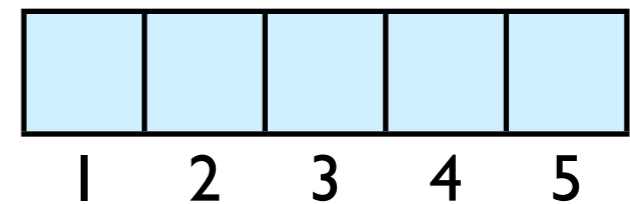
- Gauss Seidel requires x_{i-1}^{new}
- 2nd processor can't start until 1st processor is done?
- One approximation: ignore corners; use old data at start of 2nd processor's work.
- At end of each iteration, exchange neighbour cells.



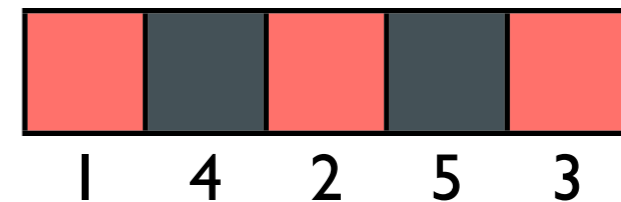
Red-Black Reordering

- No law that says we have to number cells 1,2,3..
- Even/odd
- Now, all red cells can be done at once independently generating new values
- Black cells now use all-new red data
- Exchange neighbour information after each “colour”.

Gauss-Seidel



Red-Black



Decomposition:

- Jacobi only requires x^{old}_{i-1}
- Easily parallelizable!
- At end of each iteration, exchange x^{new} neighbour cells.

