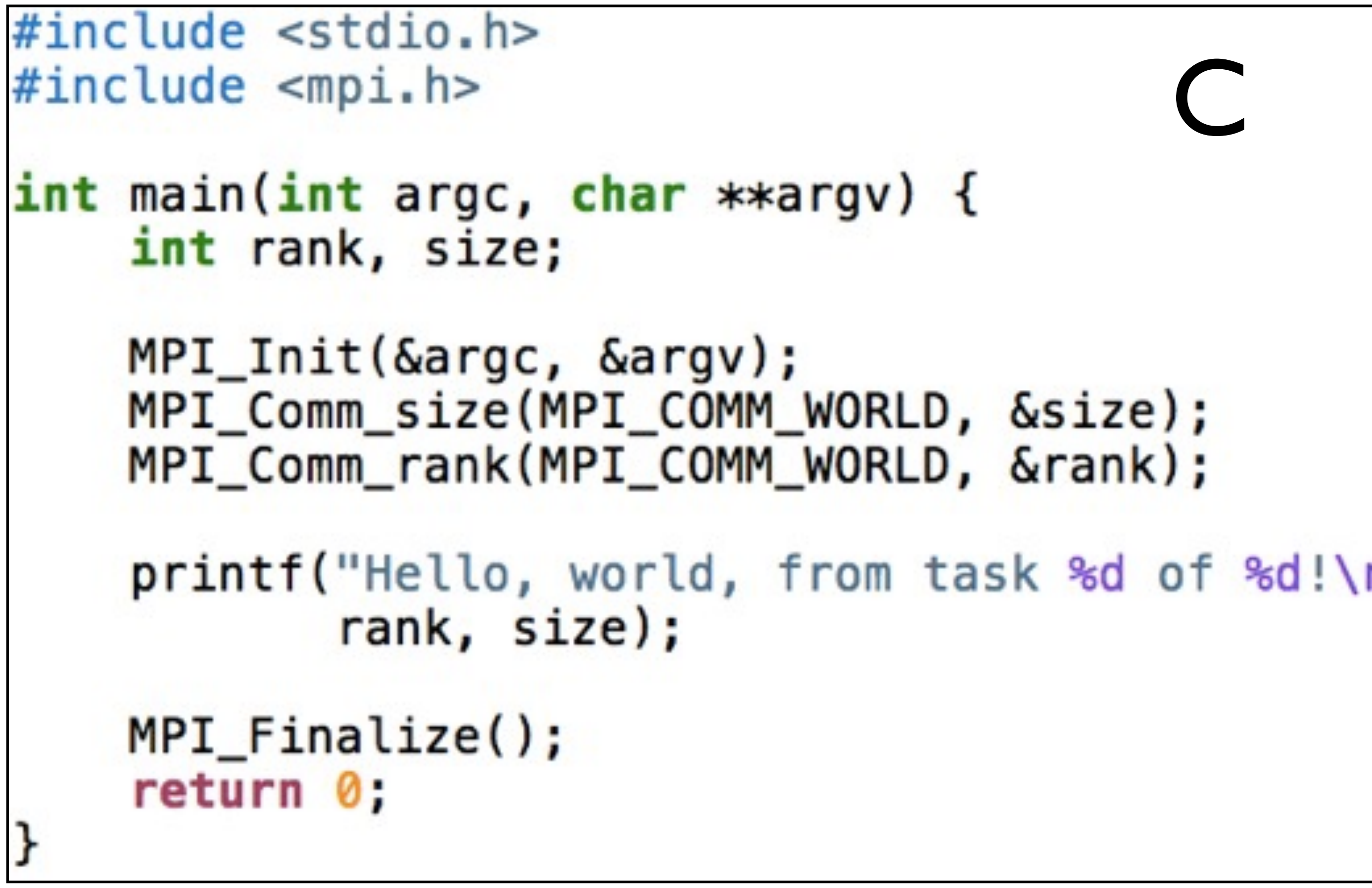
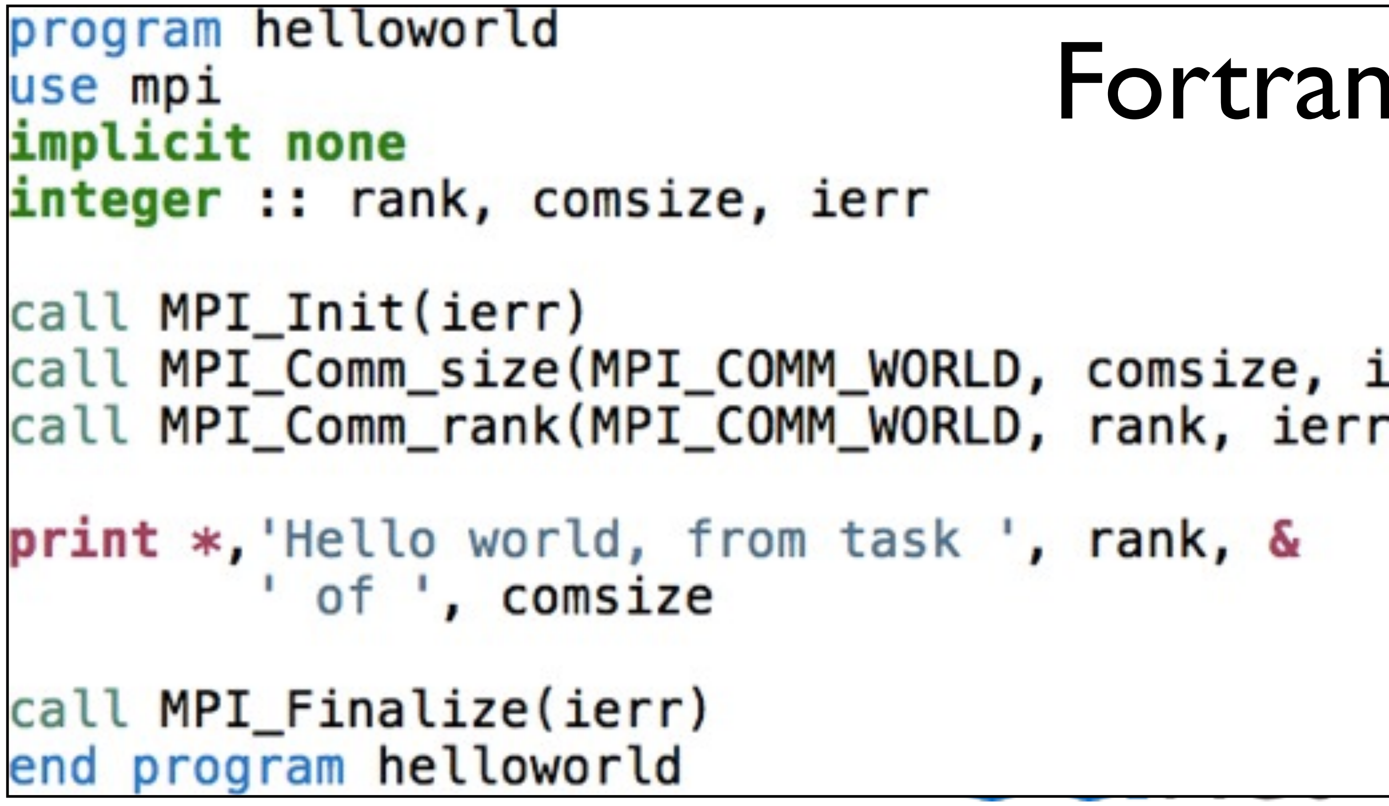
MPI is a **Library** for Message-Passing

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



C

```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!\
           rank, size);

    MPI_Finalize();
    return 0;
}
```

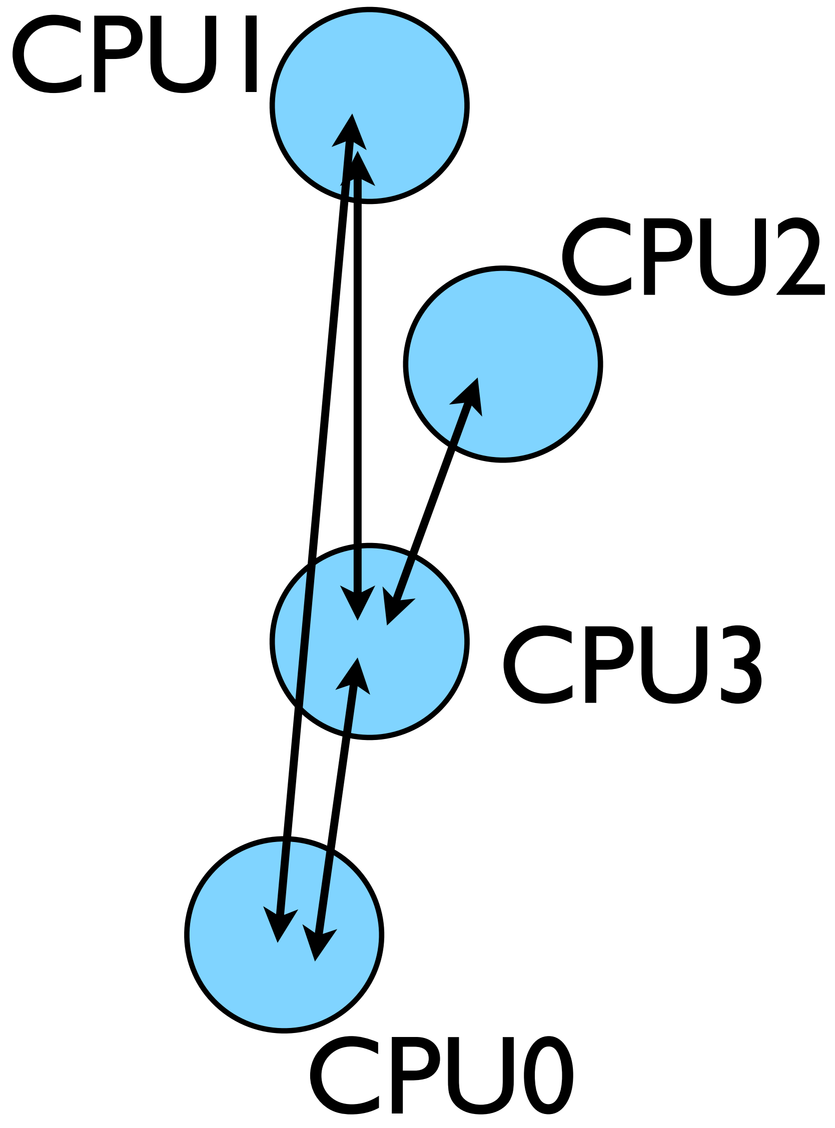


Fortran

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

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

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

call MPI_Finalize(ierr)
end program helloworld
```

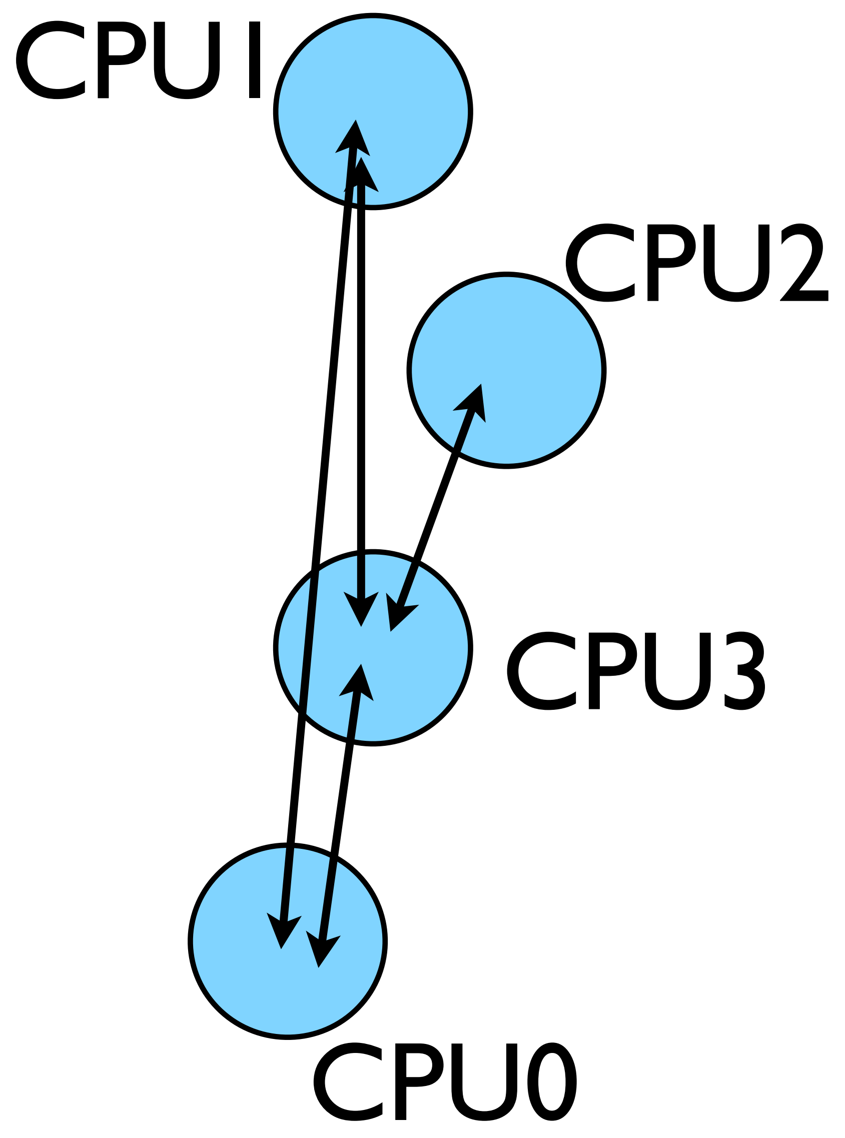
MPI is a Library for **Message-Passing**

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



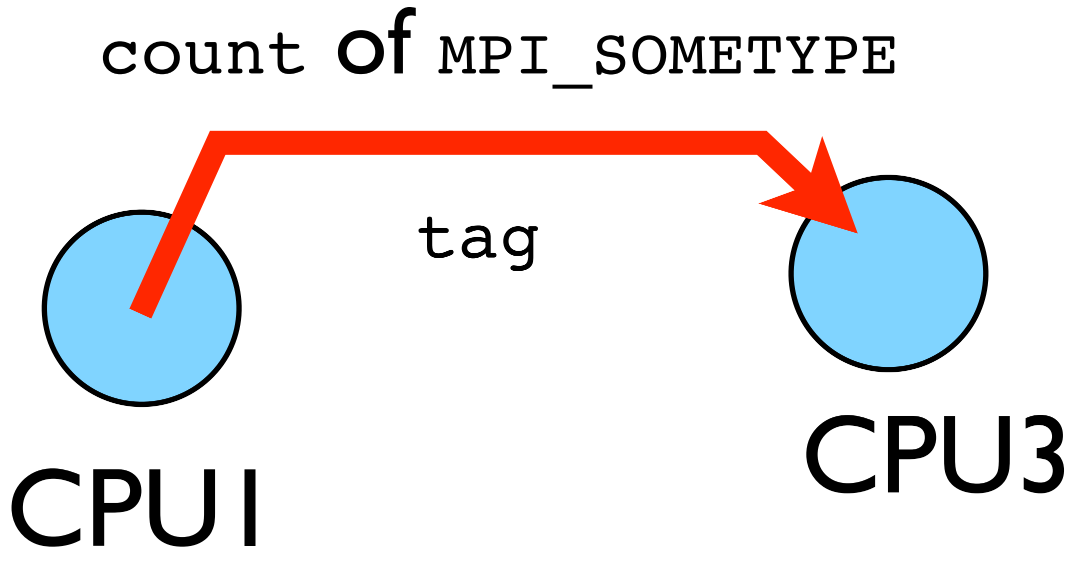
MPI is a Library for **Message-Passing**

- 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



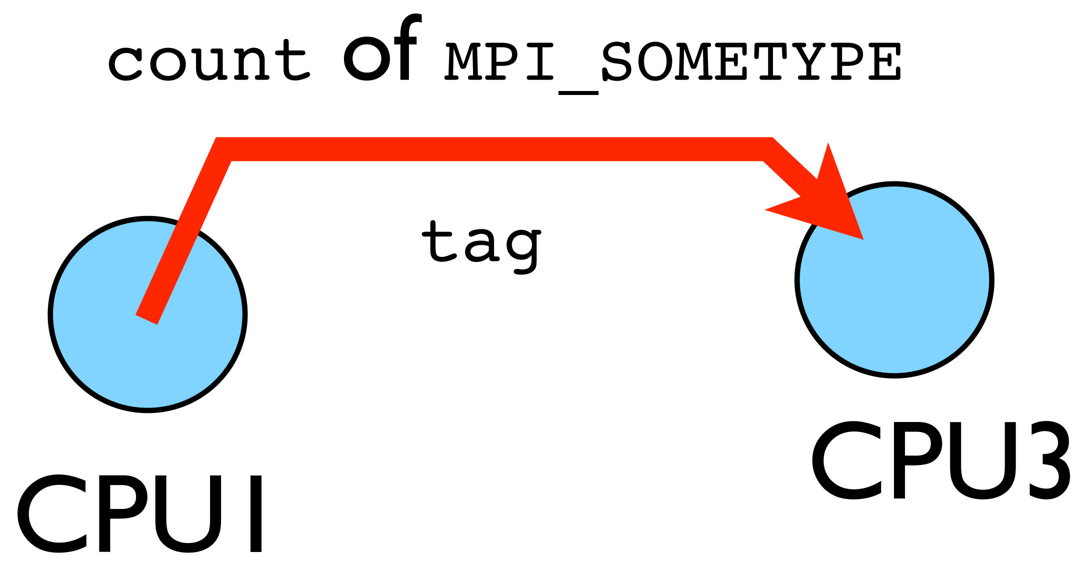
Messages

- 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 integer **`tag`** is also included - helps keep things straight if lots of messages are sent.



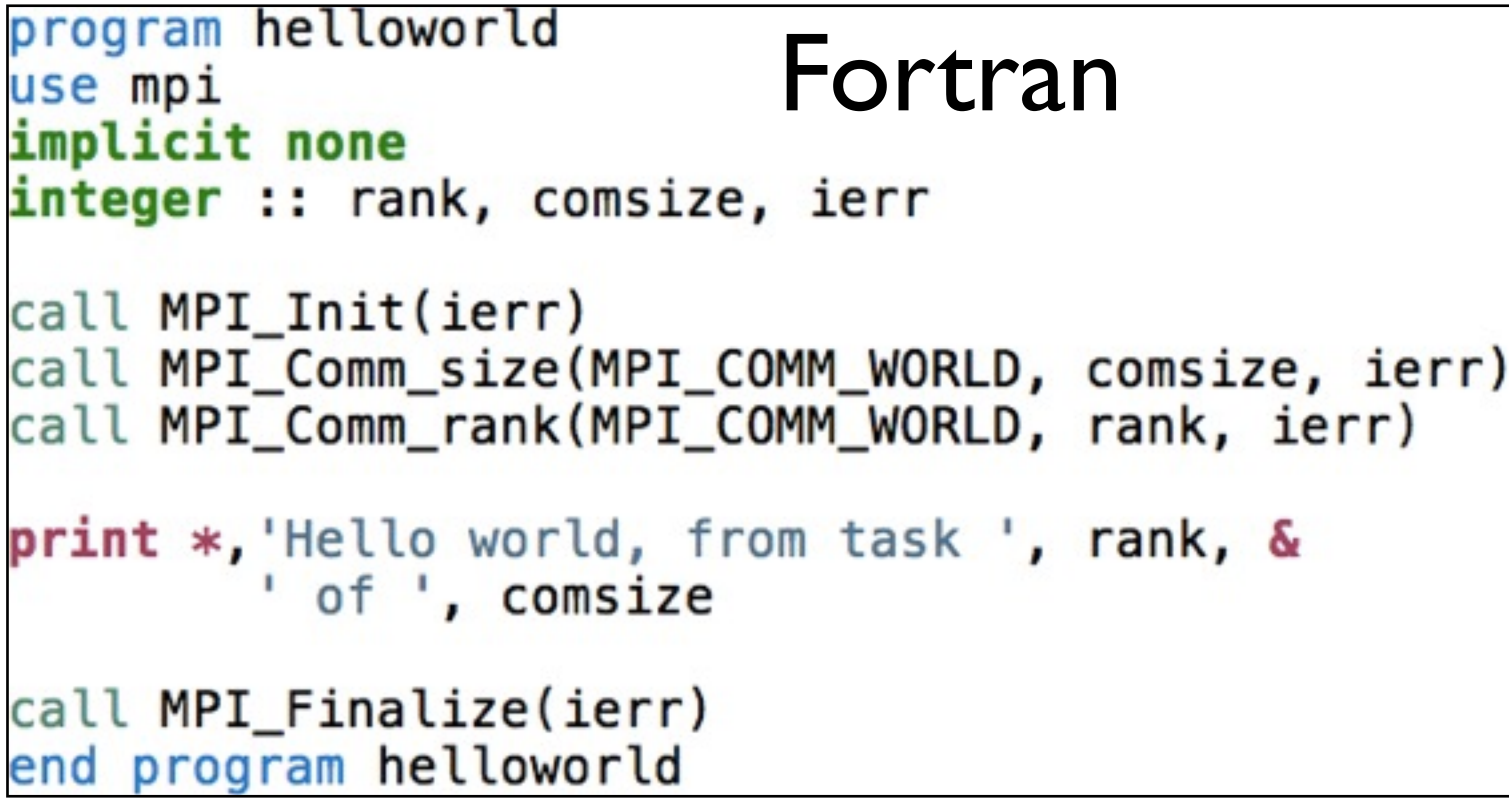
Size of MPI Library

- 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()
```

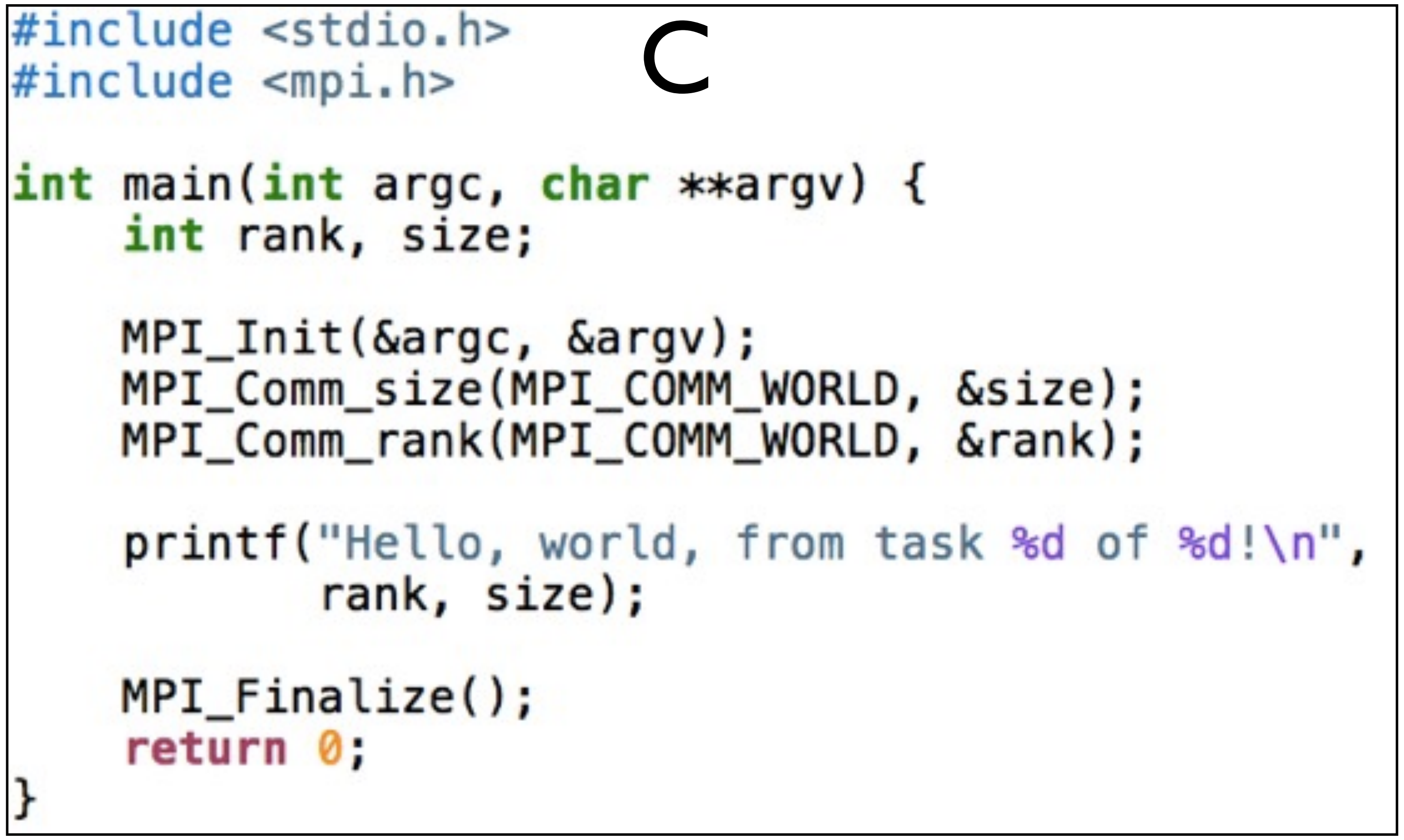
Hello World

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



Fortran

```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
```



C

```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;
}
```

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
```

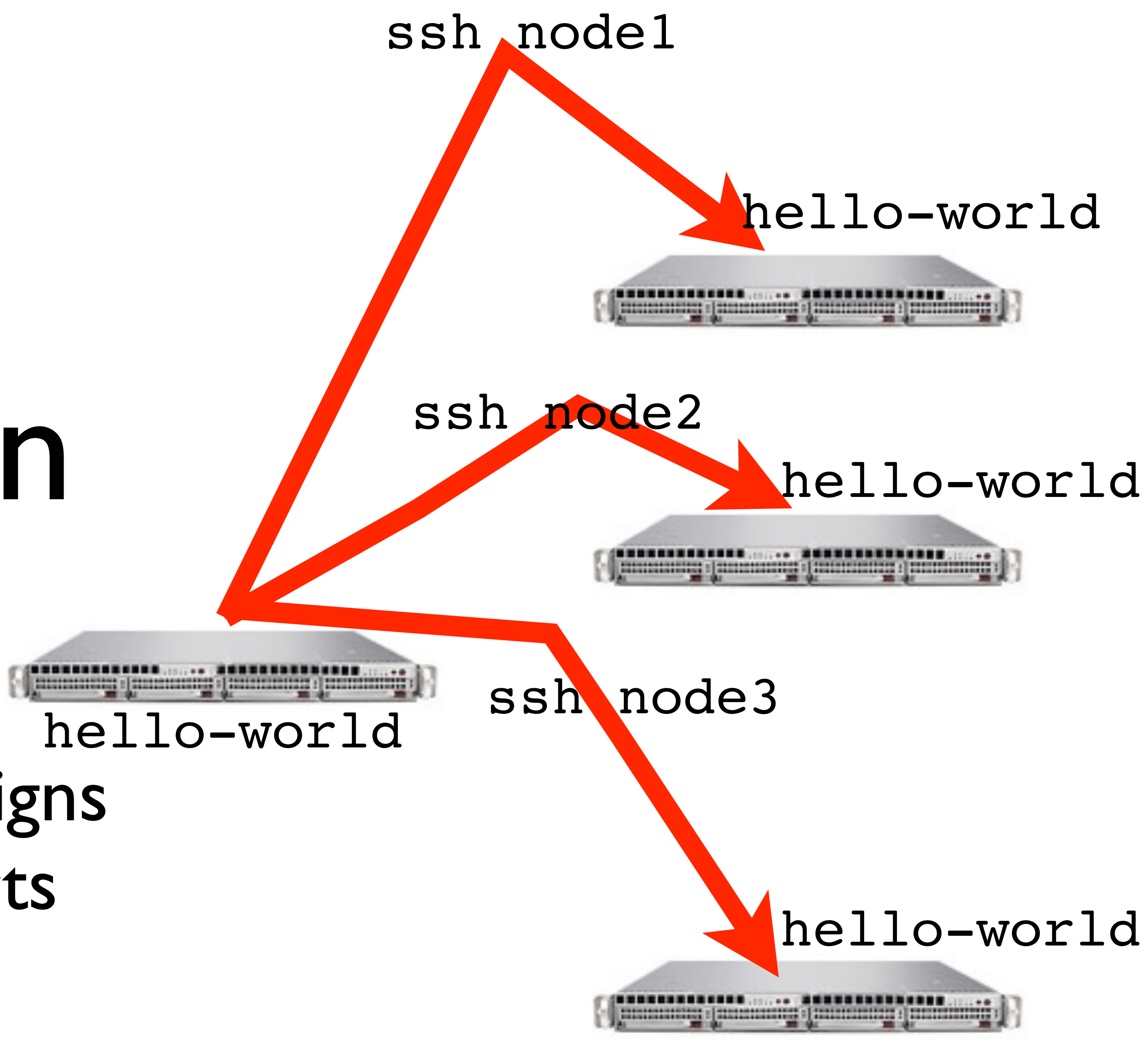
t

What mpicc/ mpif77 do

- Just wrappers for the system C, Fortran compilers that have the various -I, -L clauses in there automaticaly
- -v option (sharcnet) or --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 progam
- Sets variables somehow that mpi programs recognize so that they know which process they are

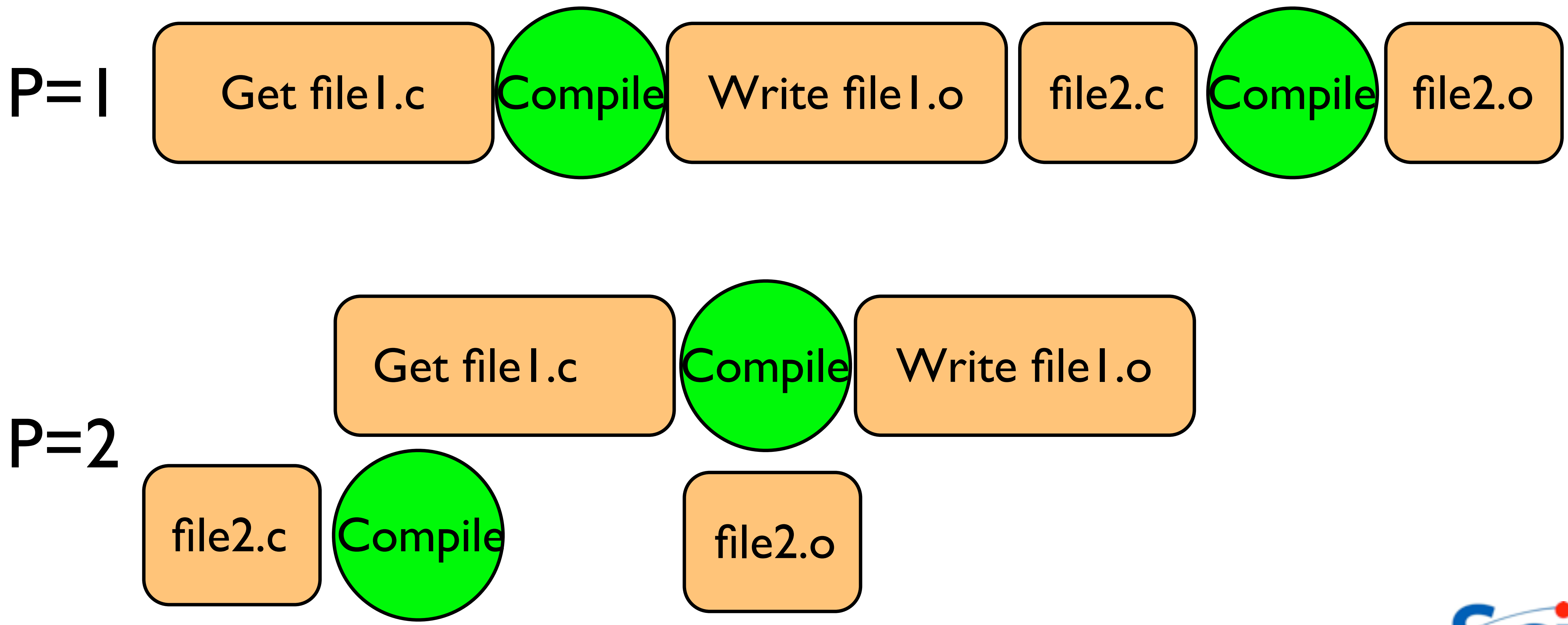
```
$ hostname
$ mpirun -np 4 hostname
$ ls
$ mpirun -np 4 ls
```

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

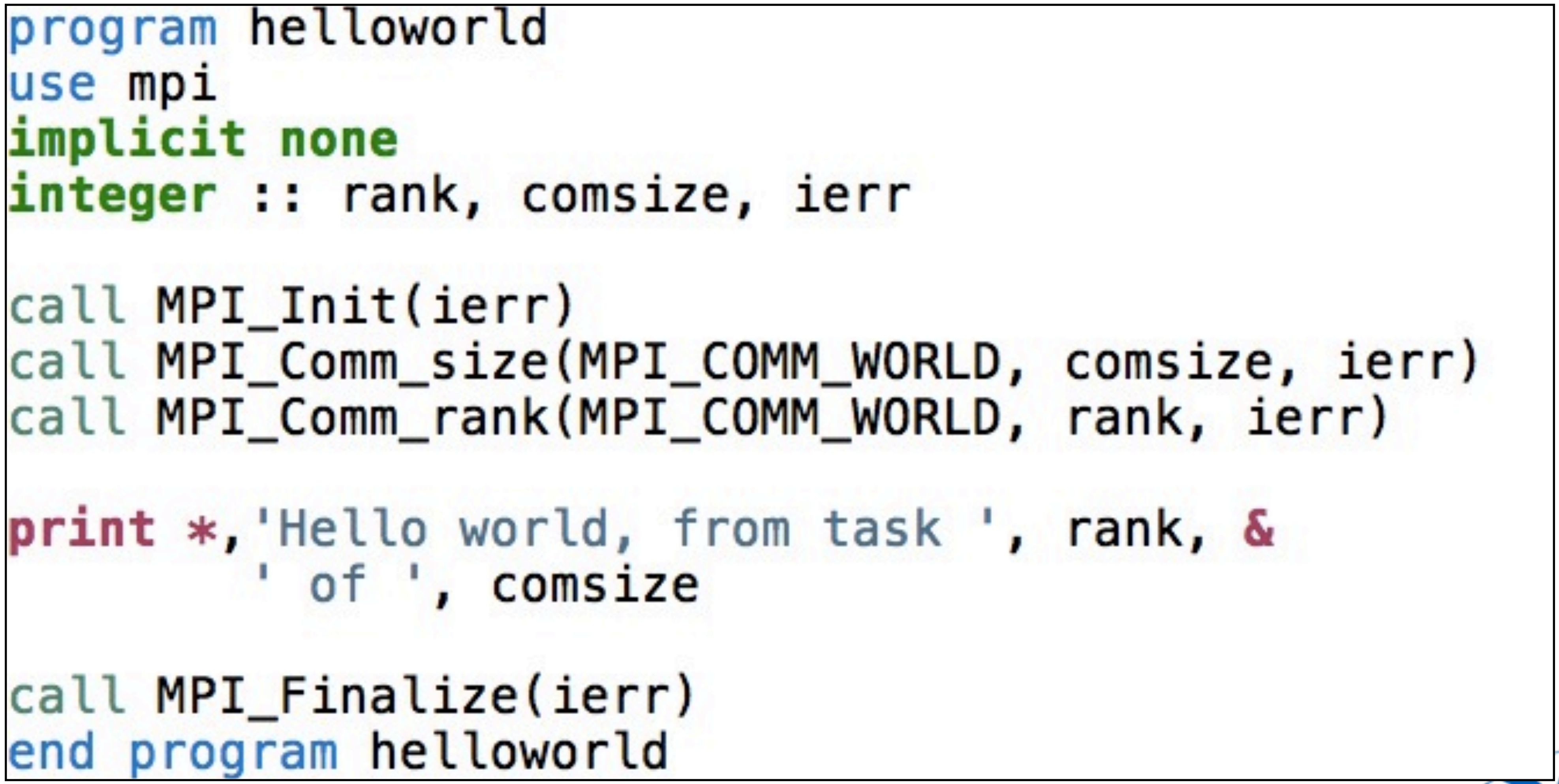
```
$ make
$ make -j 2
$ make -j
```

Overlapping Computation with I/O



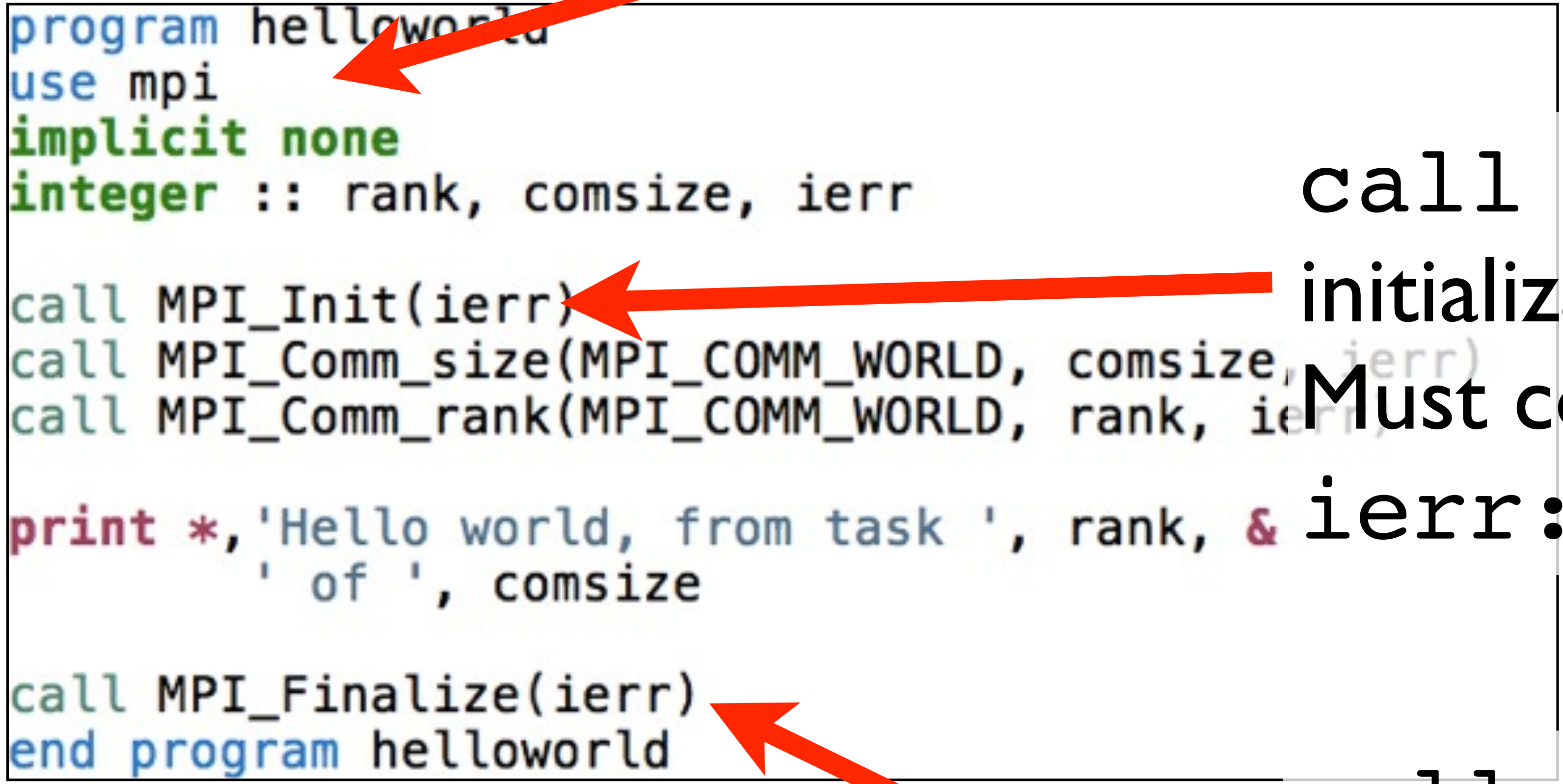
What the code does

- (FORTRAN version; C is similar)



```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
```

`use mpi` : imports declarations for MPI function calls

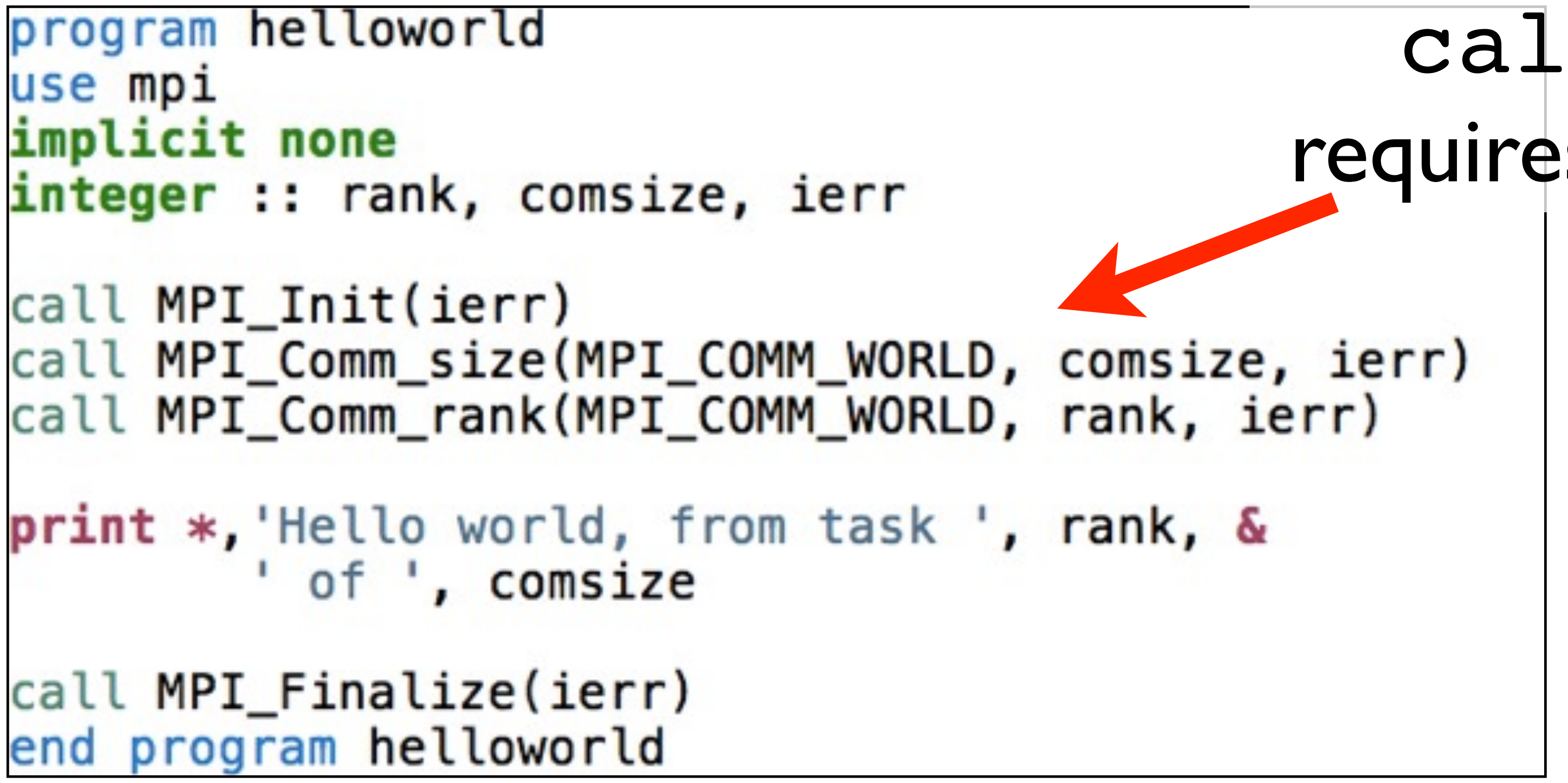


```
program helloworld
use mpi
implicit none
integer :: rank, comsize, ierr

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.

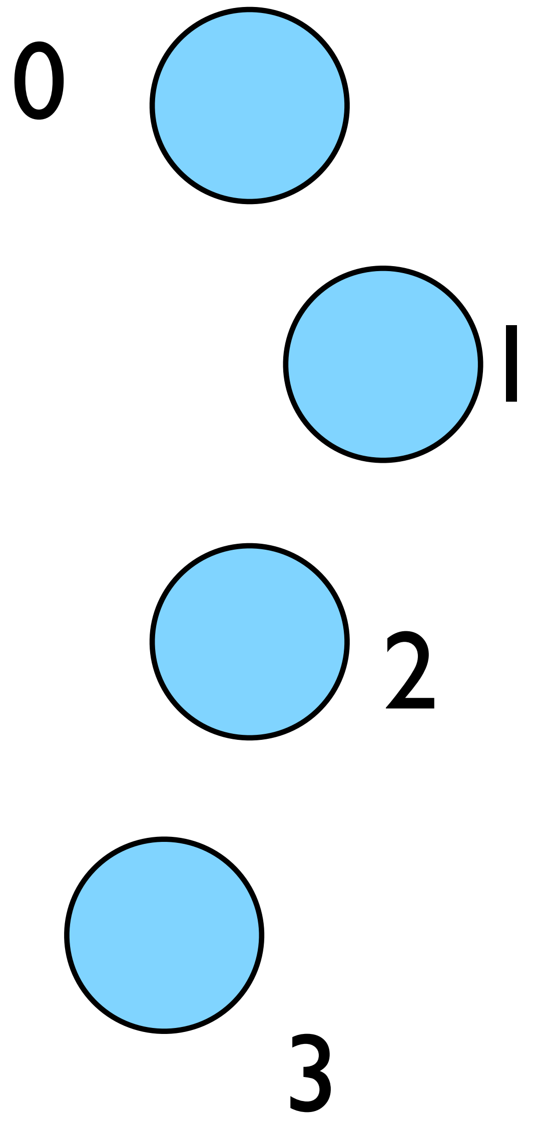
`call MPI_COMM_RANK`,
`call MPI_COMM_SIZE`:
requires a little more exposition.



```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
```

Communicators

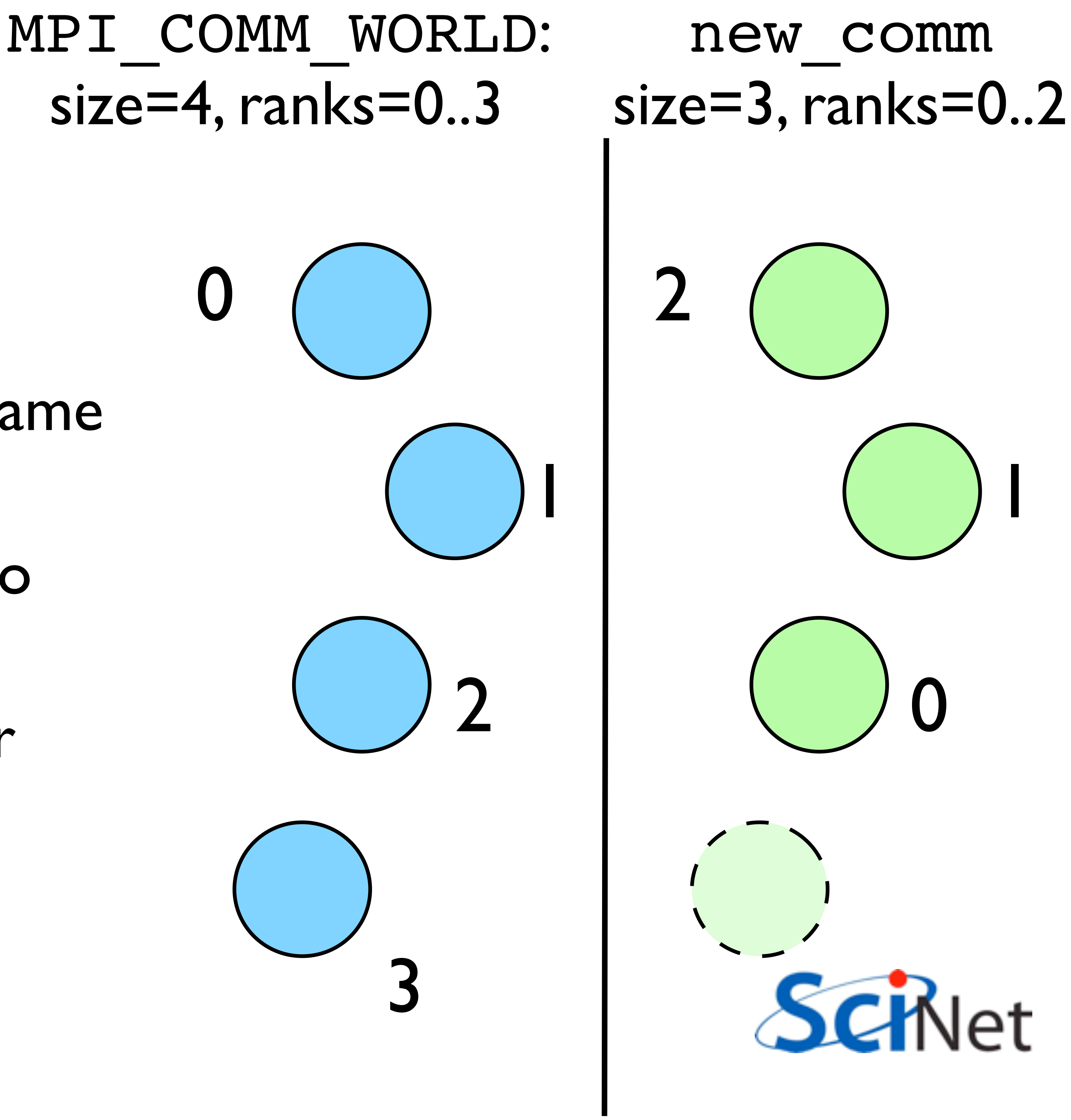
- MPI groups processes into communicators.
- Each communicator has some size -- number of tasks.
- Each task has a rank 0..size-1
- Every task in your program belongs to `MPI_COMM_WORLD`

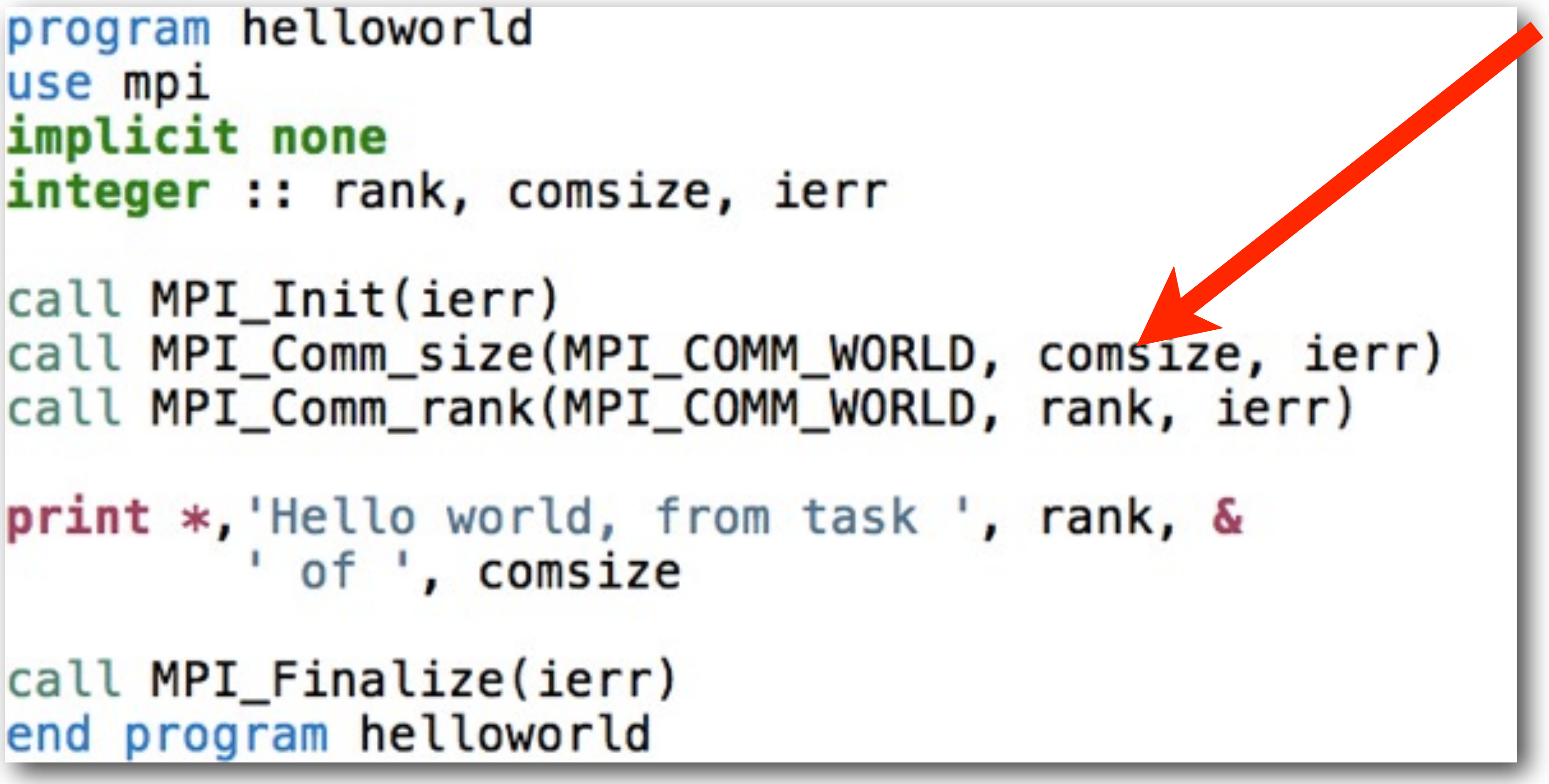


`MPI_COMM_WORLD:`
size=4, ranks=0..3

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





```
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
```

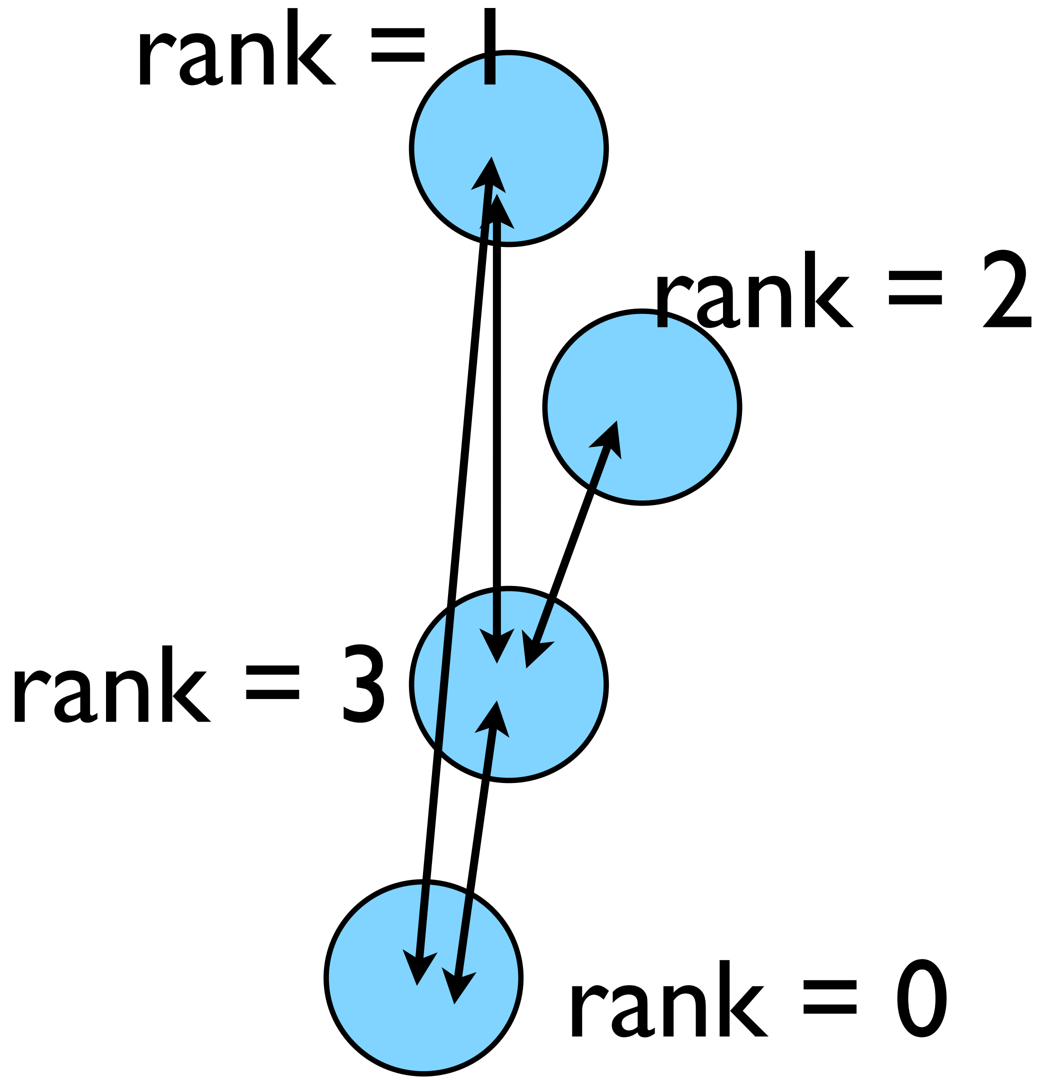
`call MPI_COMM_RANK,`
`call MPI_COMM_SIZE:`

get the size of communicato
the current tasks's rank with
communicator.

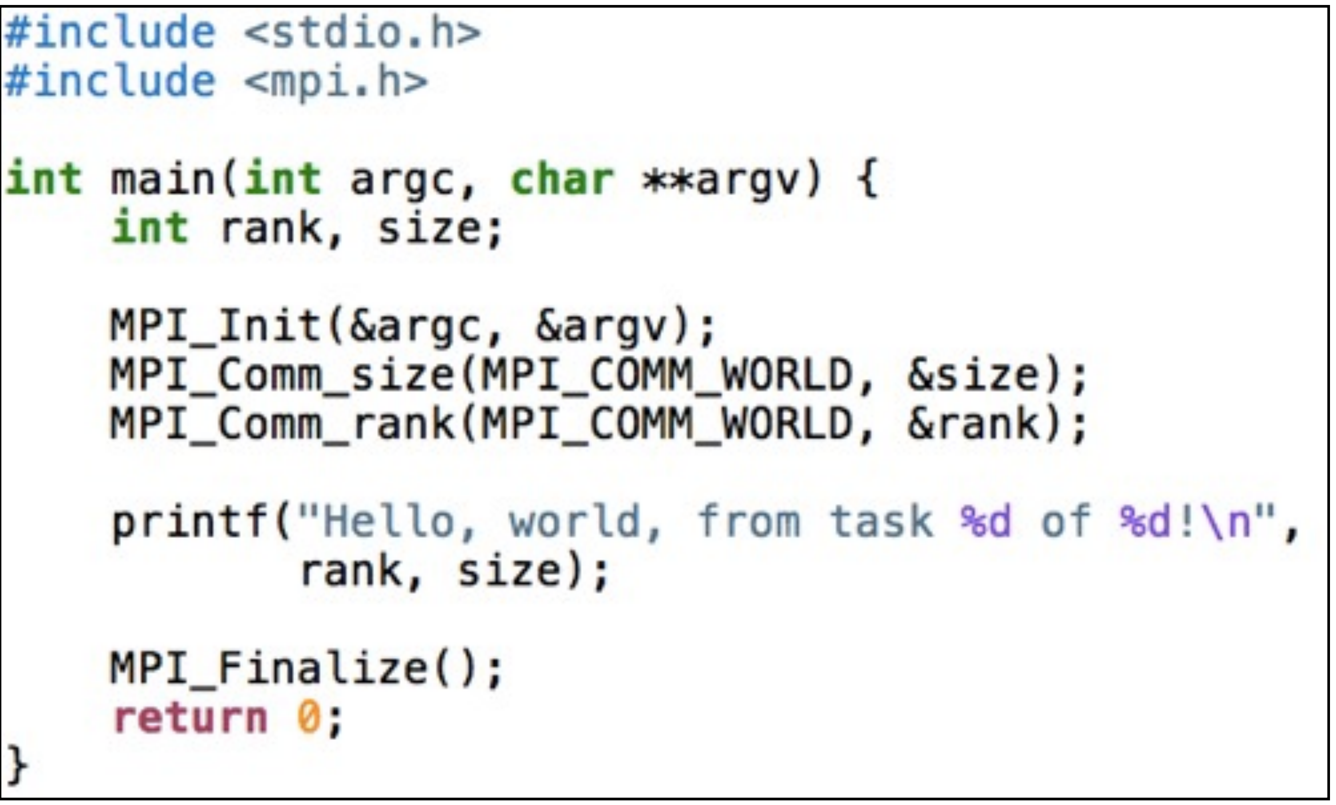
put answers in `rank` and
`size`

Rank and Size much more important in MPI than OpenMP

- In OpenMP, compiler assigns jobs to each thread; don't need to know which one you are.
- MPI: processes determine amongst themselves which piece of puzzle to work on, then communicate with appropriate others.

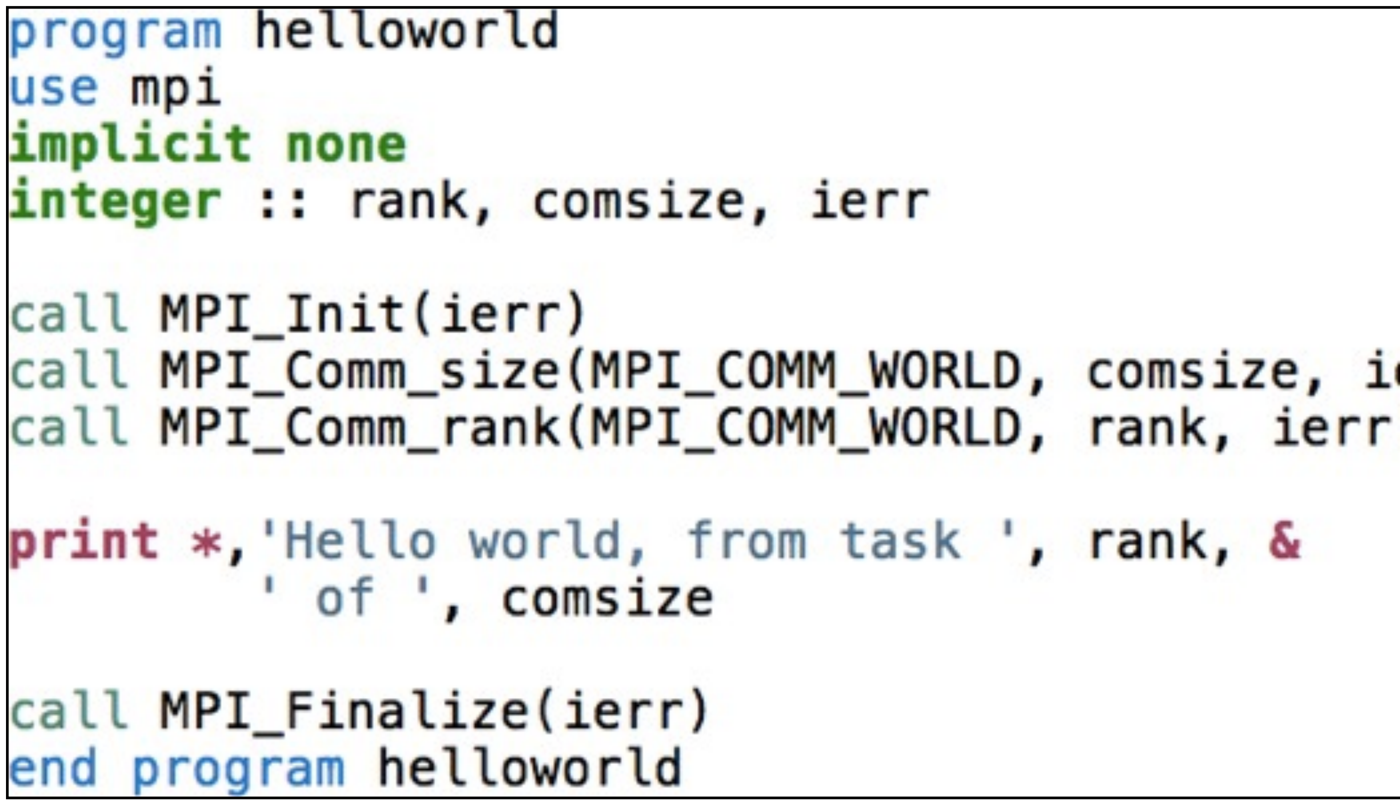


C



```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



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

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, i
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

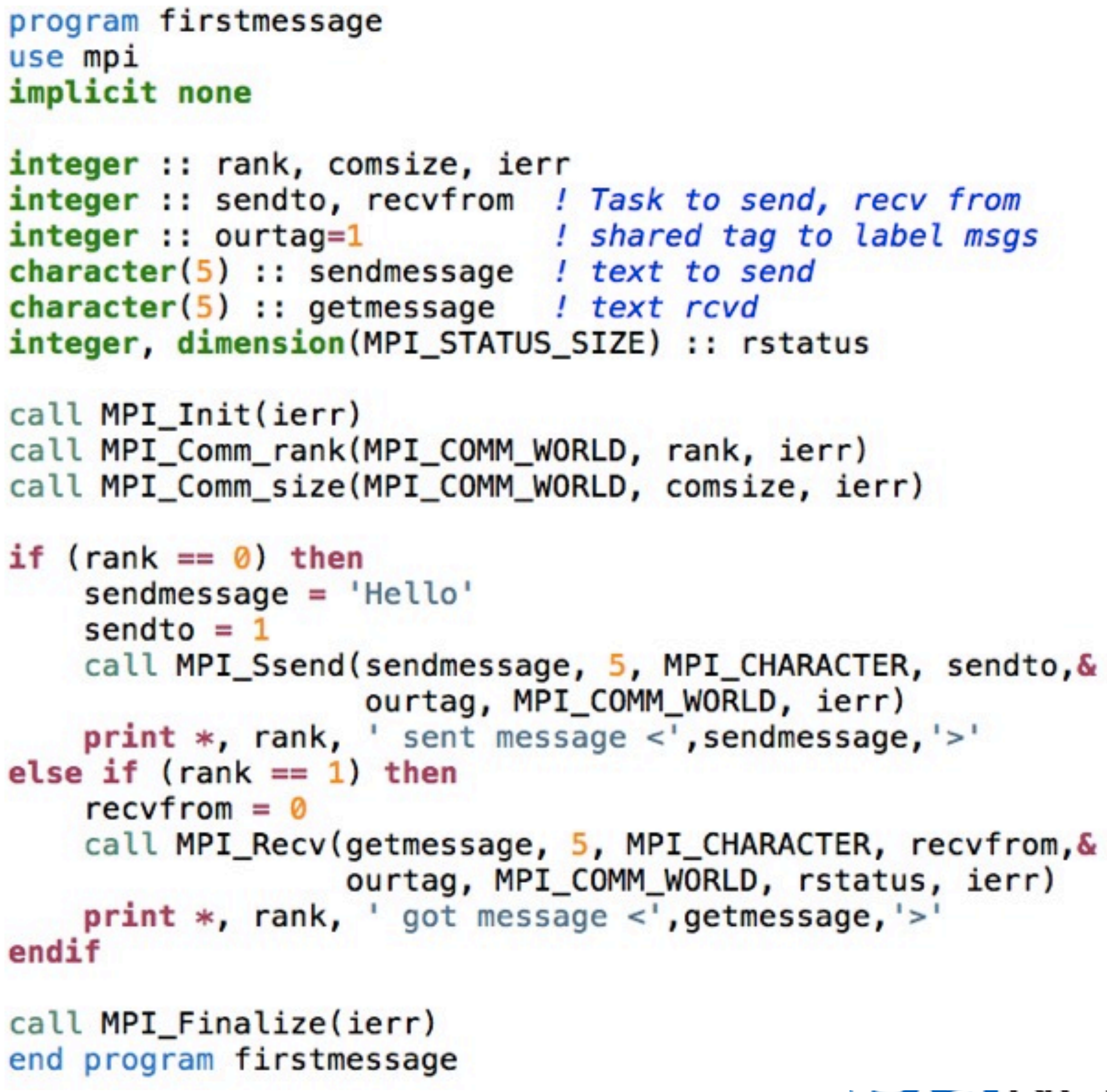
Our first real MPI program - but no Ms are P'ed!

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

```c
#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



```fortran
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(rcvarr, count, MPI_TYPE, source, tag,
                 Communicator, status, ierr)
```

Special Source/Dest: MPI_PROC_NULL

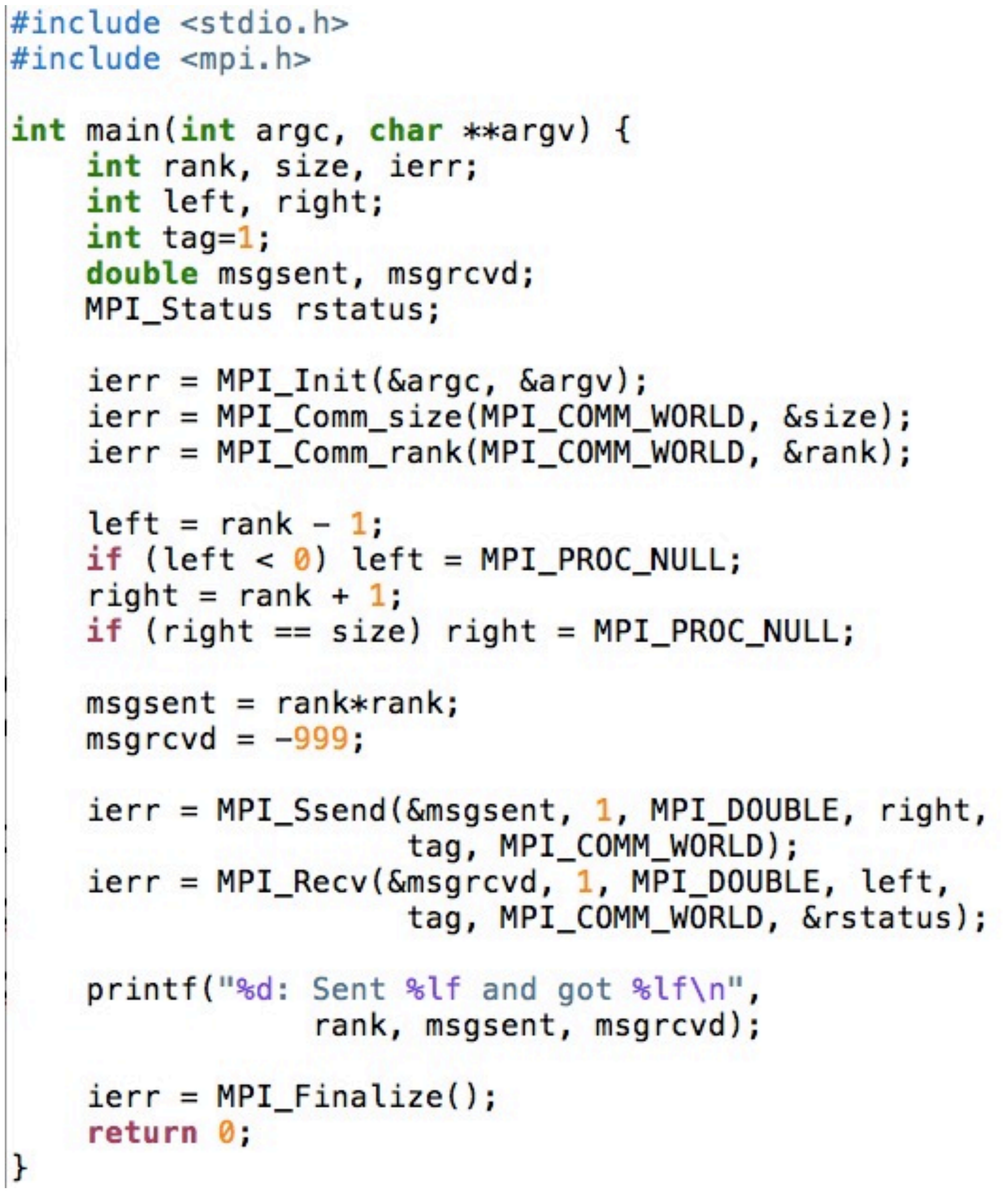
`MPI_PROC_NULL` basically ignores the relevant operation; can lead to cleaner code.

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;
}
```

t

More complicated example:

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

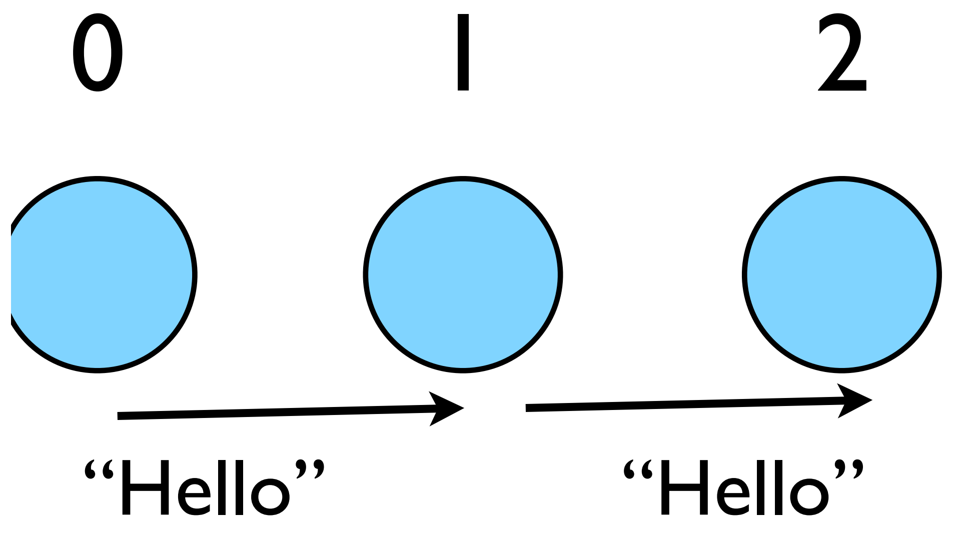
```
program secondmessage
use mpi
implicit none

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

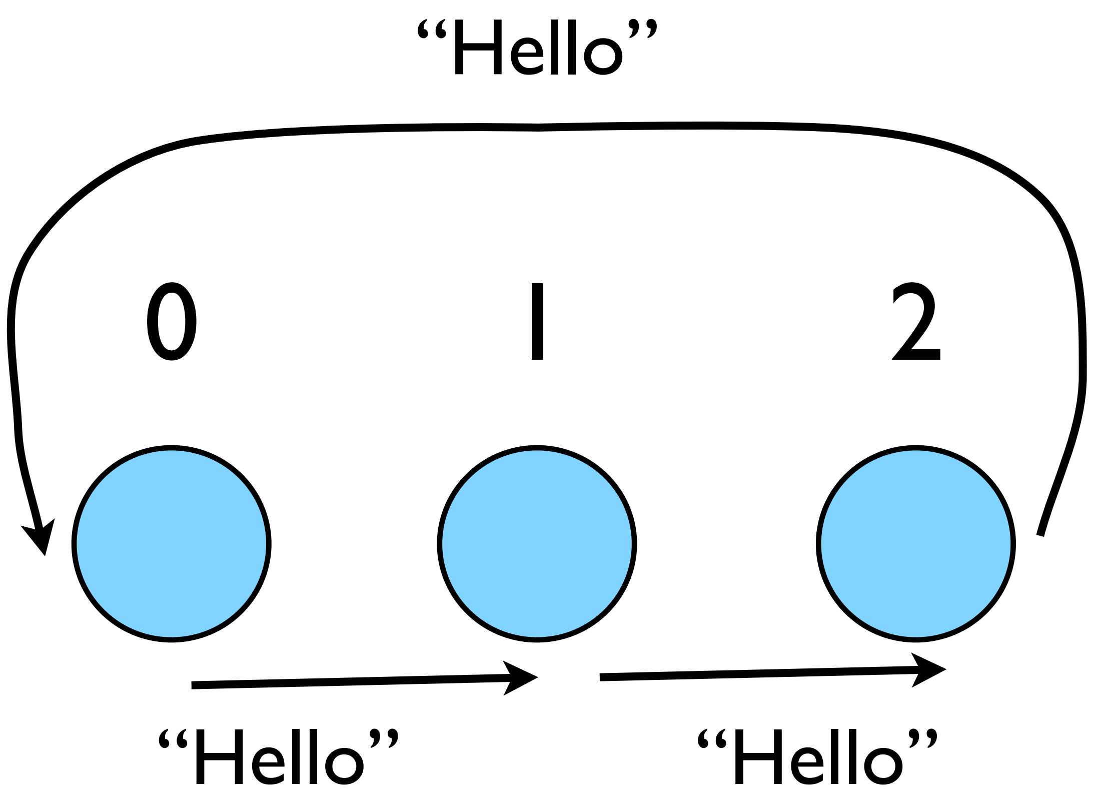
```
$ mpirun -np 4 ./secondmessage
3: Sent 9.000000 and got 4.000000
0: Sent 0.000000 and got -999.000
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
```

```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;
}
```

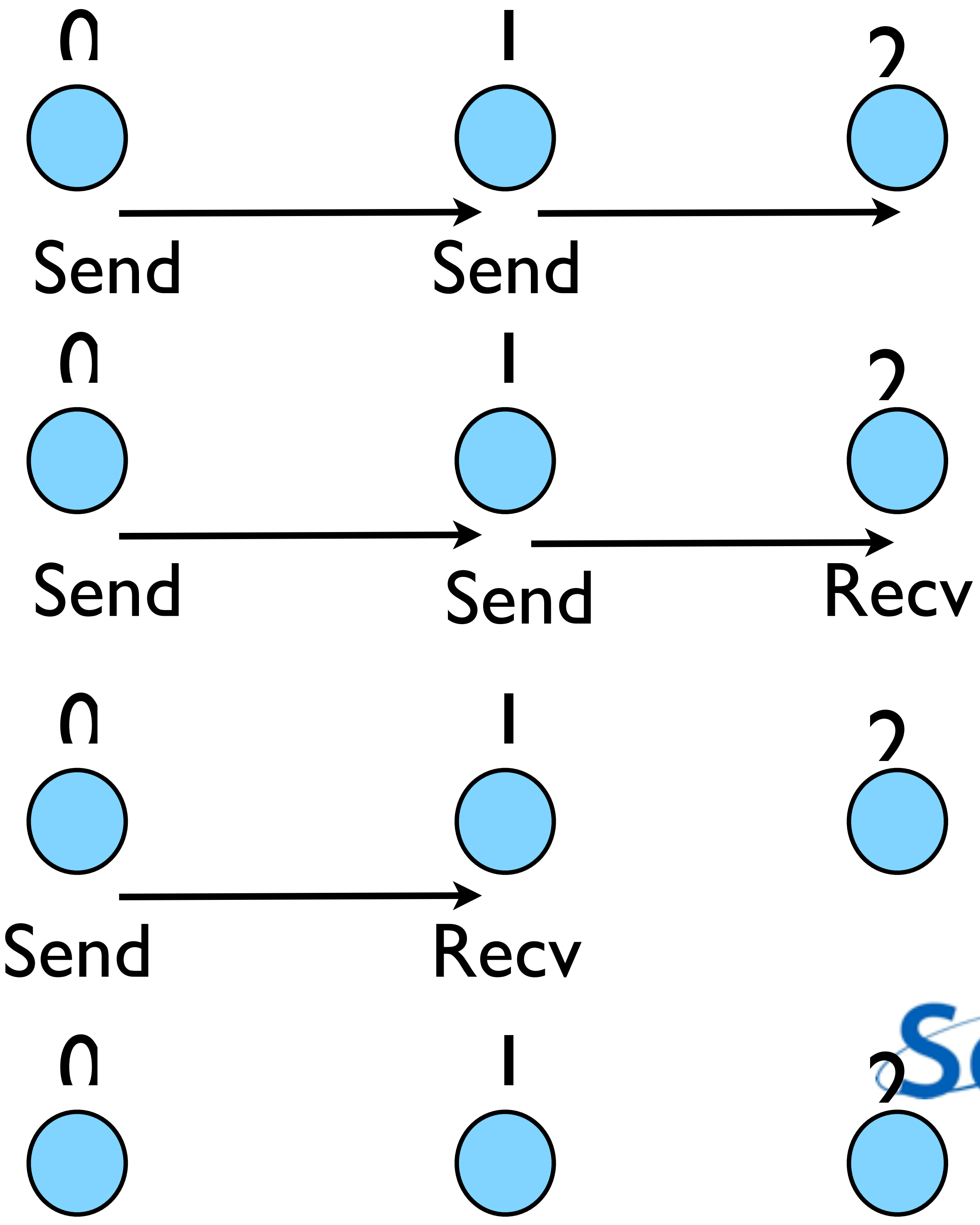


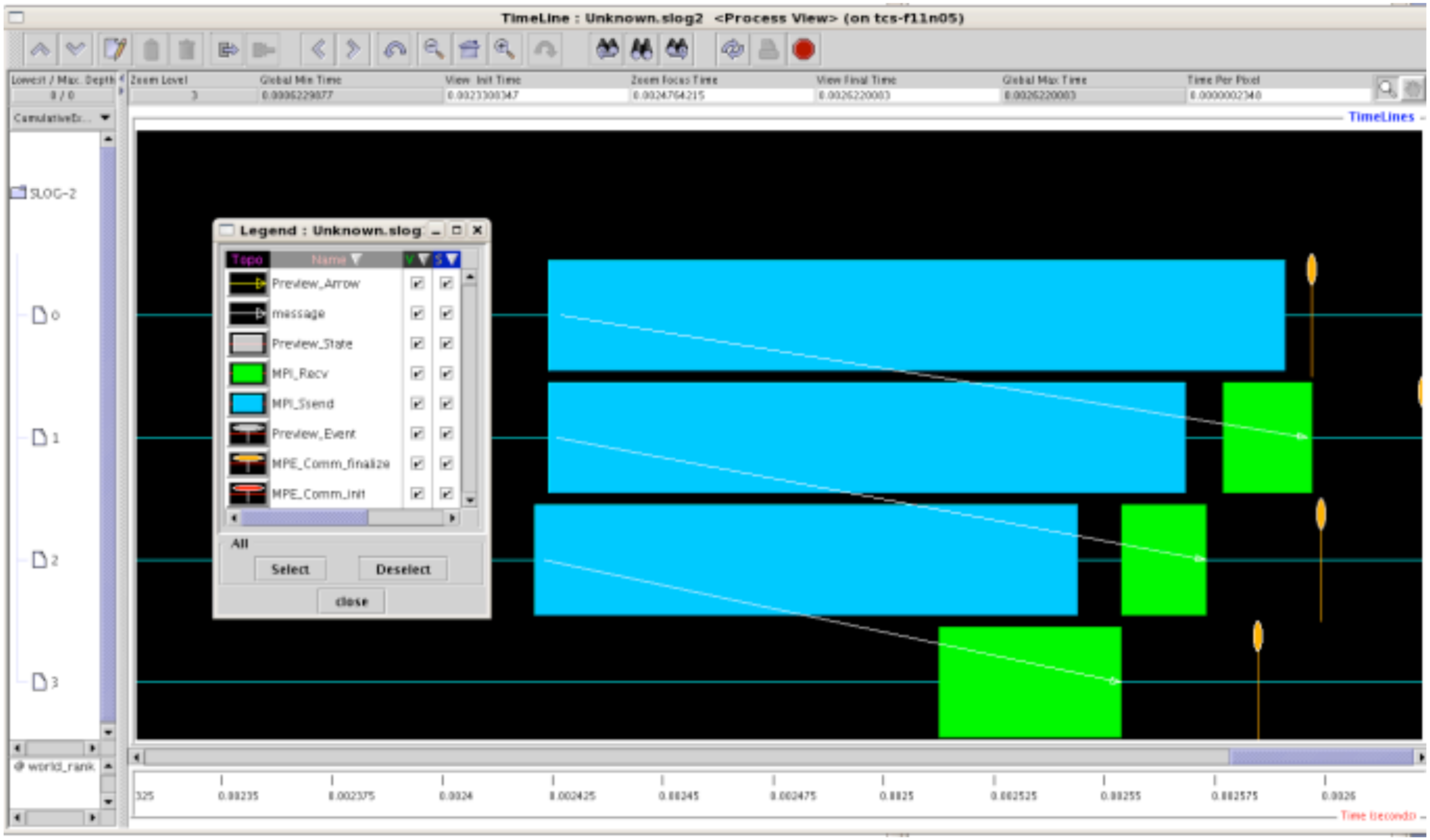
Implement periodic boundary conditions

- 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)
```





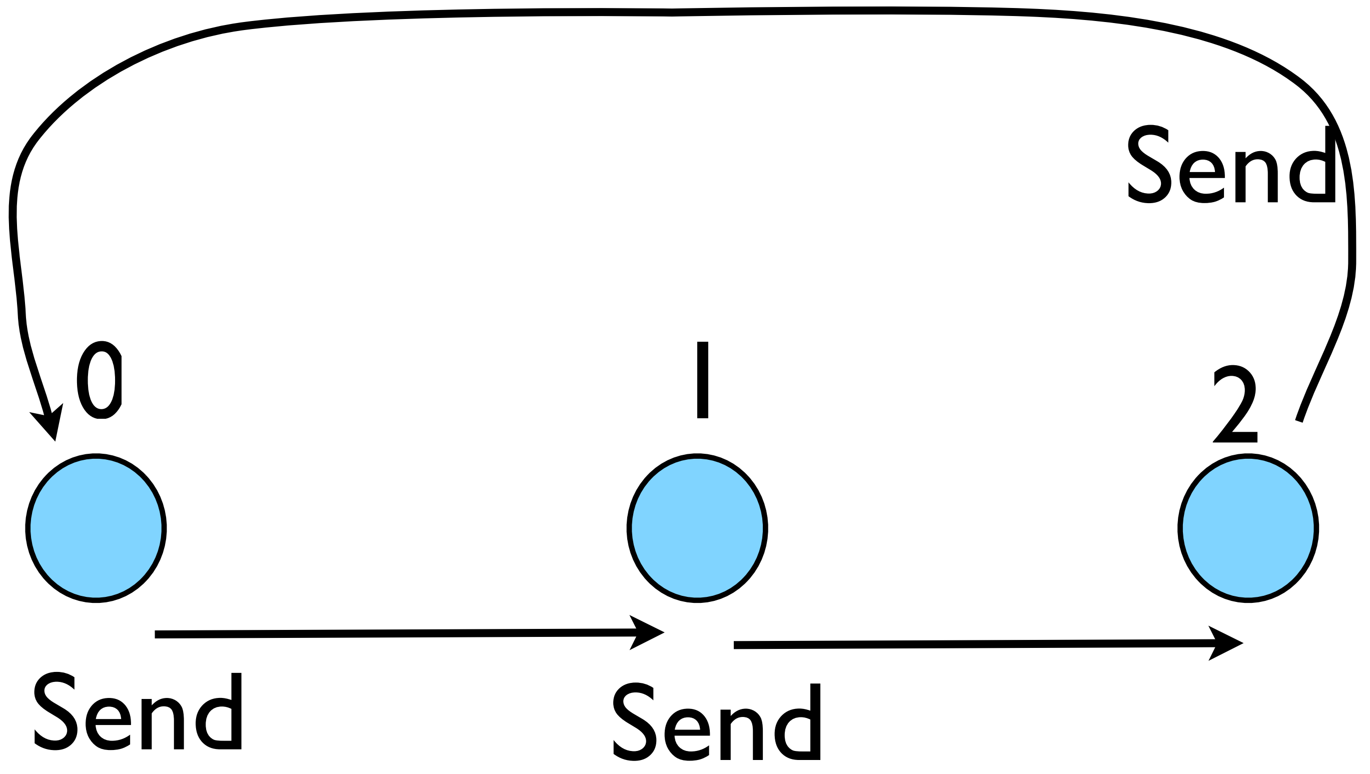
TimeLine : Unknown.slog2 <Process View> (on tcs-f11n05)

Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
0 / 0	3	0.0006229077	0.0023300347	0.0024764215	0.0026220003	0.0026220003	0.0000002340

Legend : Unknown.slog

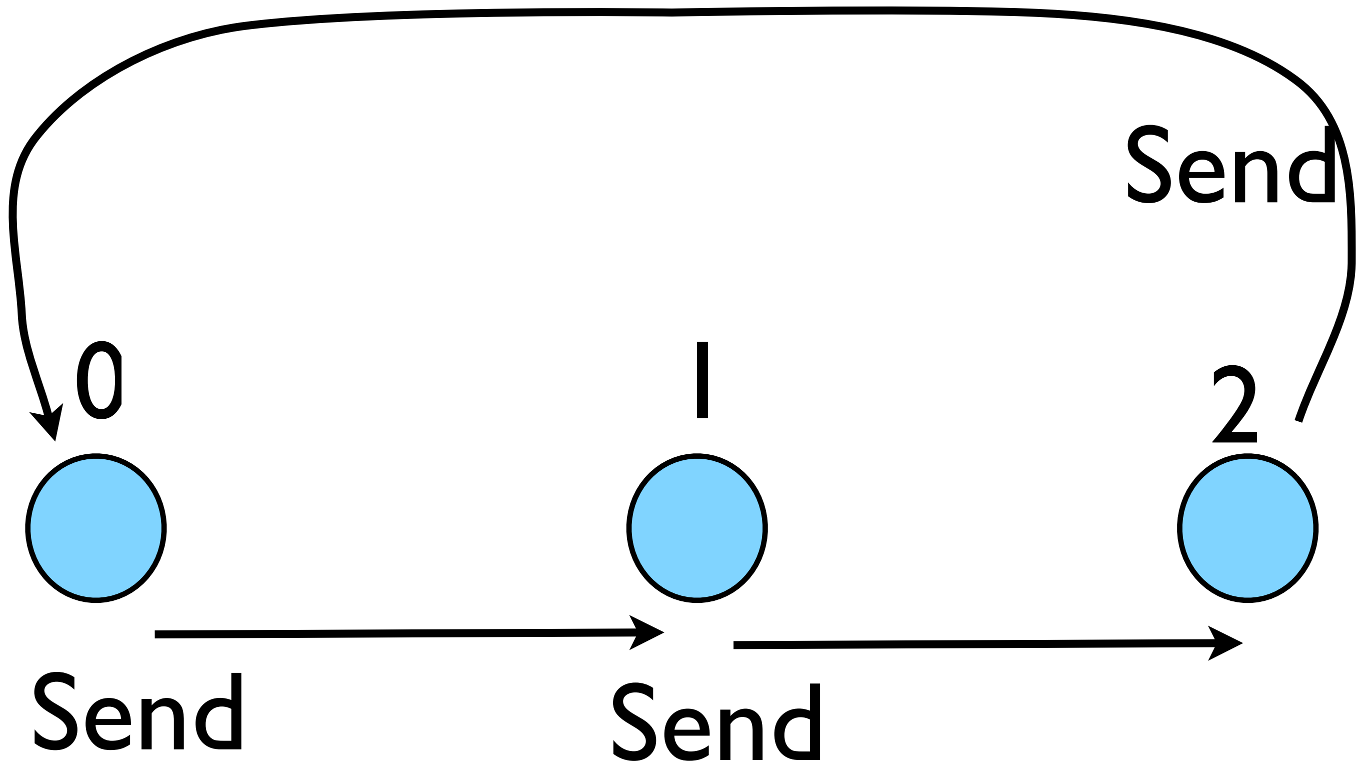
Topo	Name	V	S
→	Preview_Arrow	✓	✓
→	message	✓	✓
▭	Preview_State	✓	✓
🟩	MPI_Recv	✓	✓
🟦	MPI_Ssend	✓	✓
●	Preview_Event	✓	✓
●	MPE_Comm_finalize	✓	✓
●	MPE_Comm_init	✓	✓

SciNet



```
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.

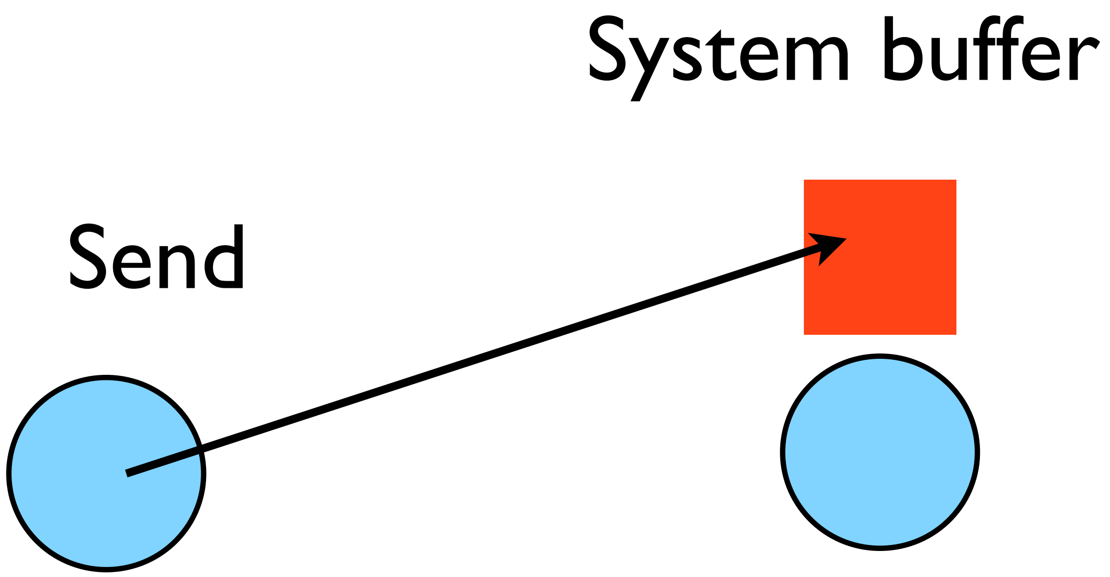
Big MPI Lesson #1

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

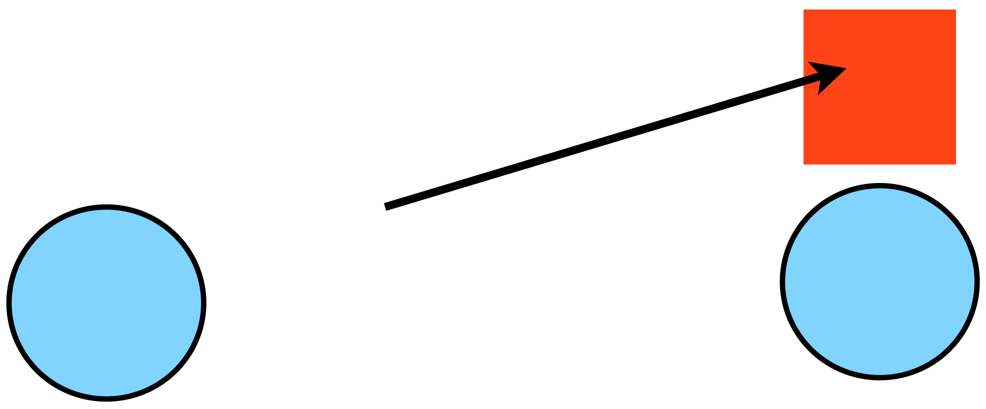
Different versions of SEND

- 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



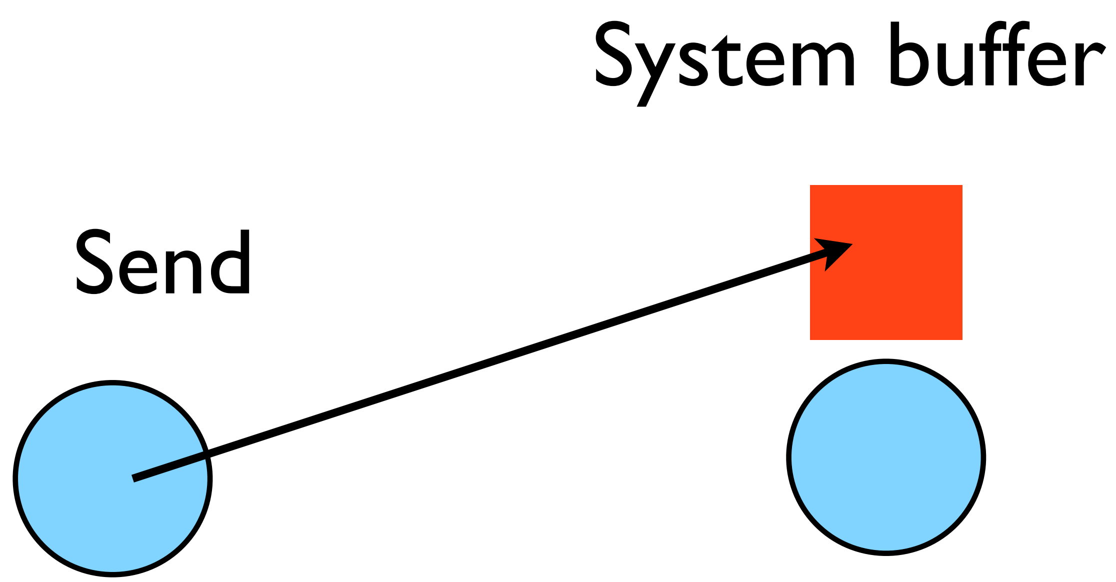
(Non) Blocking



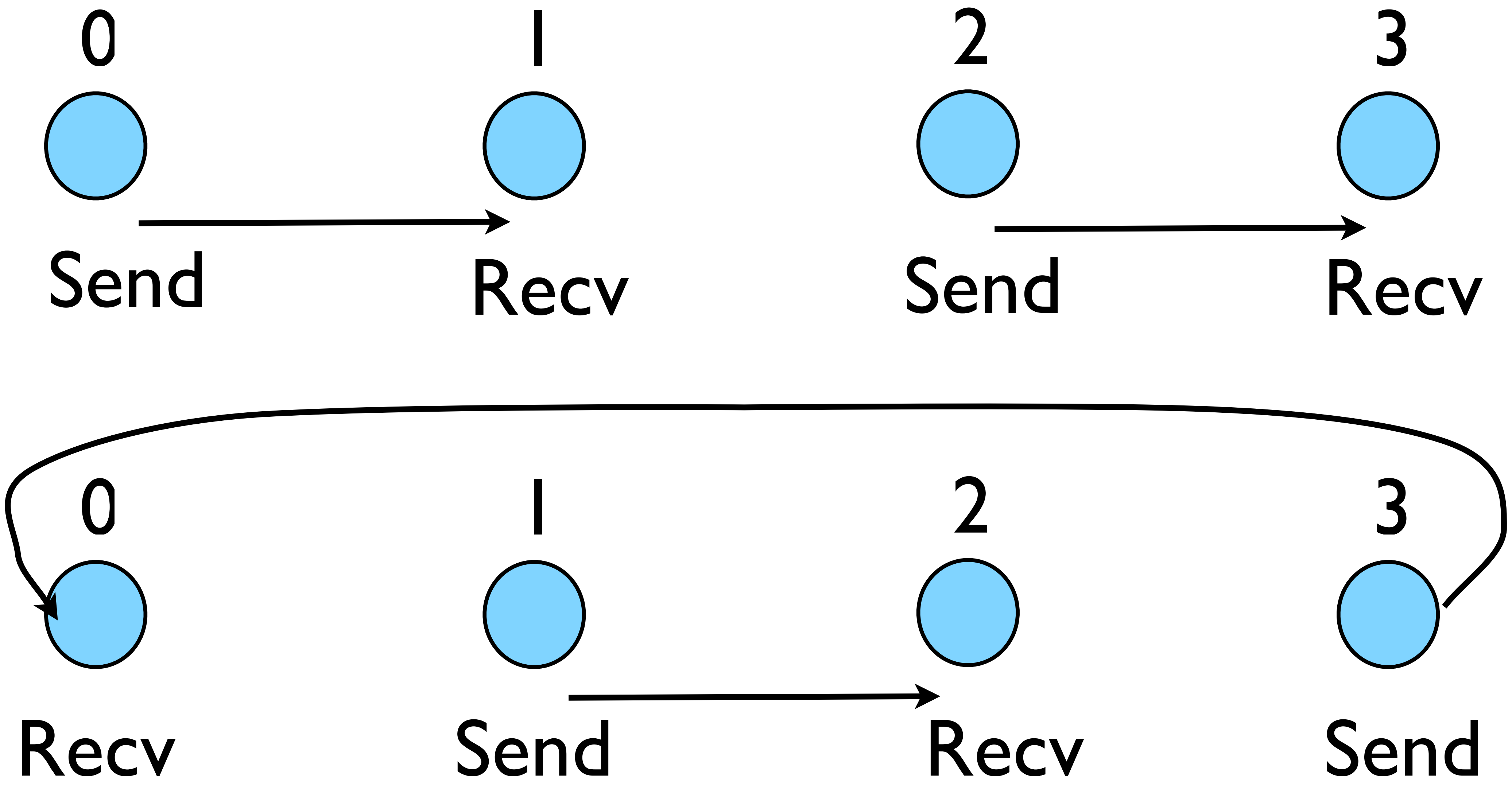
Buffering is dangerous!

- 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

```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 = 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;
}
```

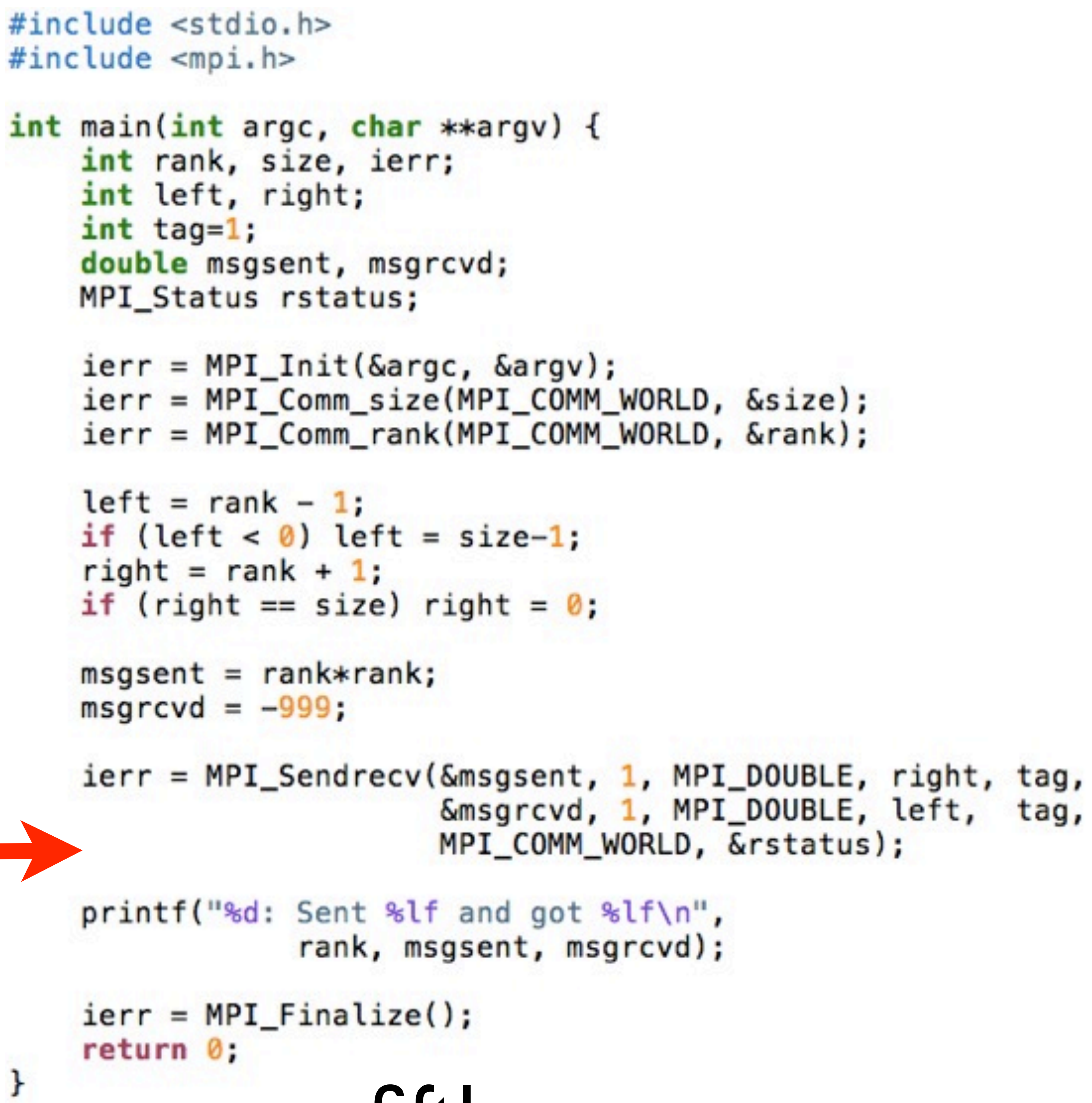
Evens send first

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.

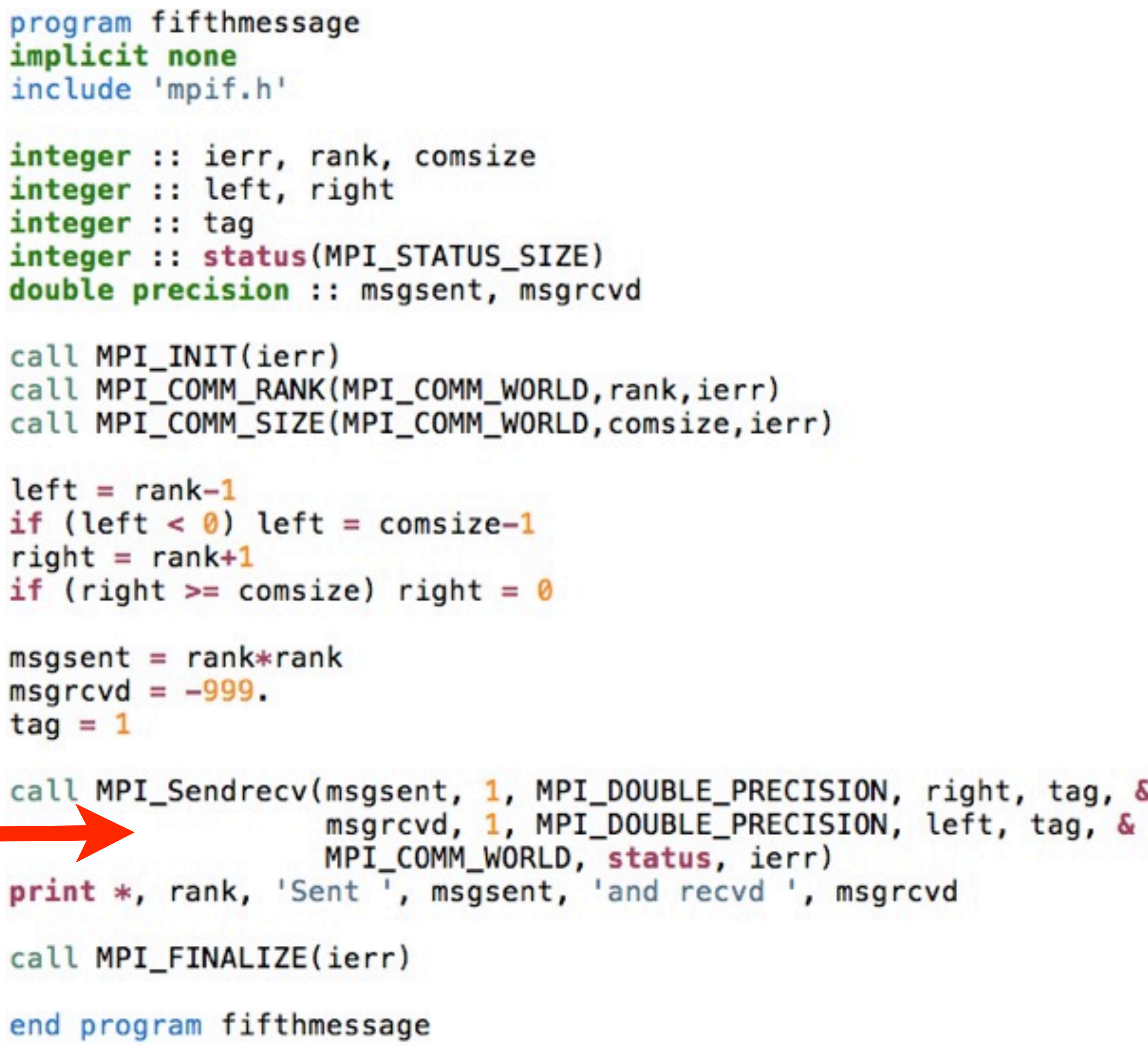


```
#include <stdio.h>
#include <mpi.h>

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

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

Sendrecv = Send + Recv

C syntax

Send Args

```
MPI_Status status;

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

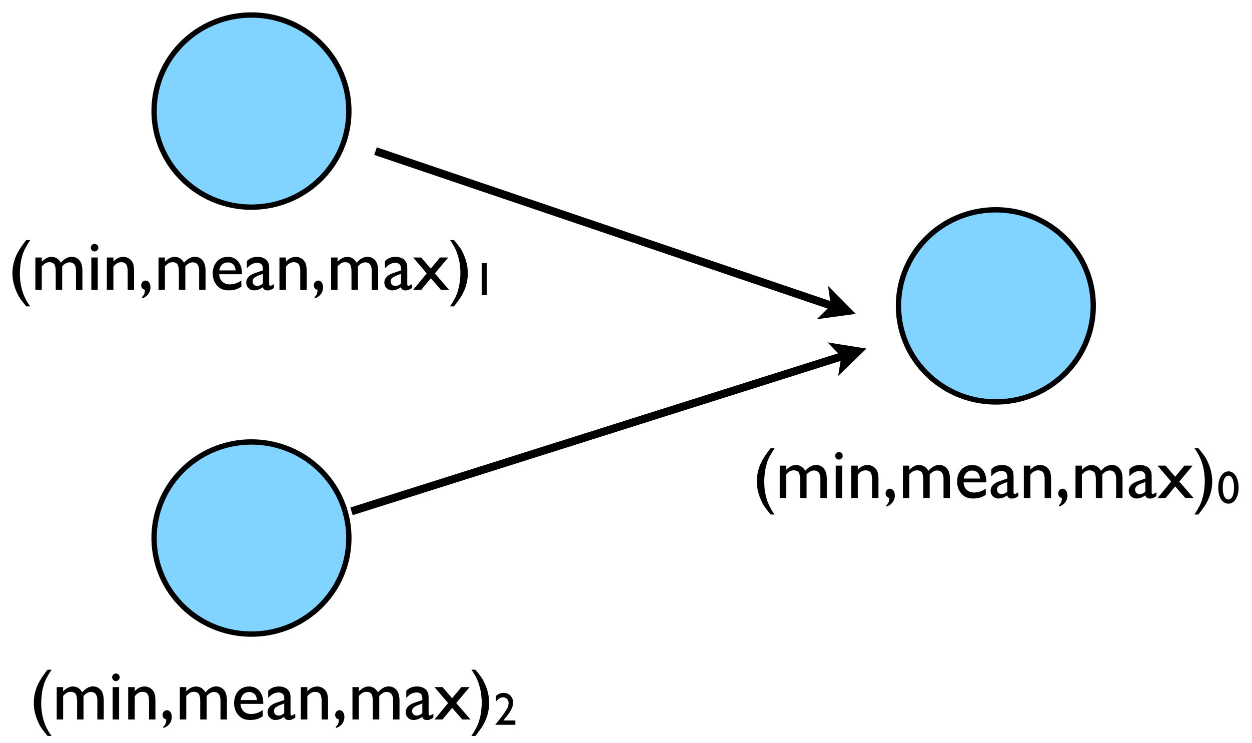
Recv Args

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?

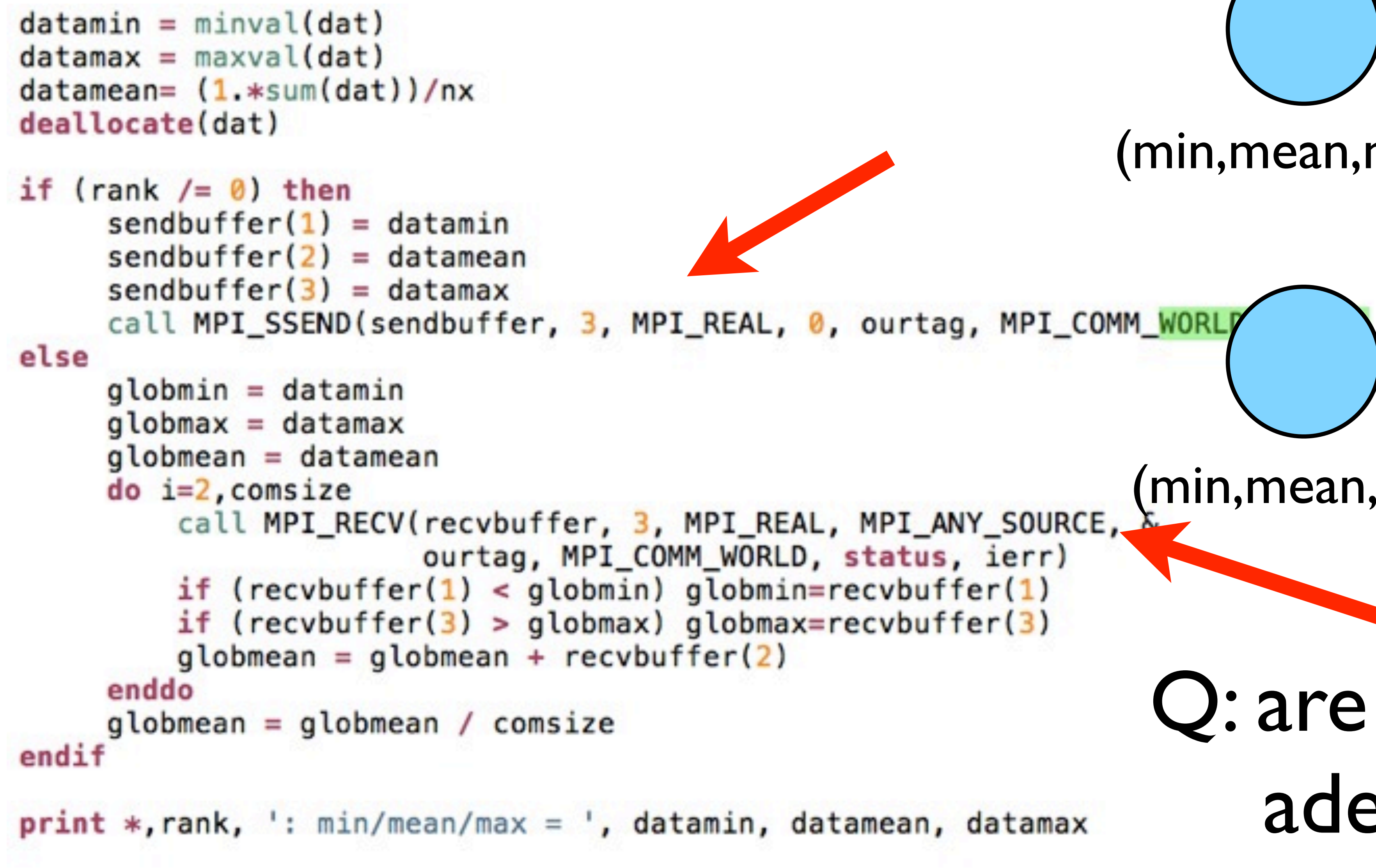
Min, Mean, Max of numbers


$(\text{min,mean,max})_1$
$(\text{min,mean,max})_0$
$(\text{min,mean,max})_2$

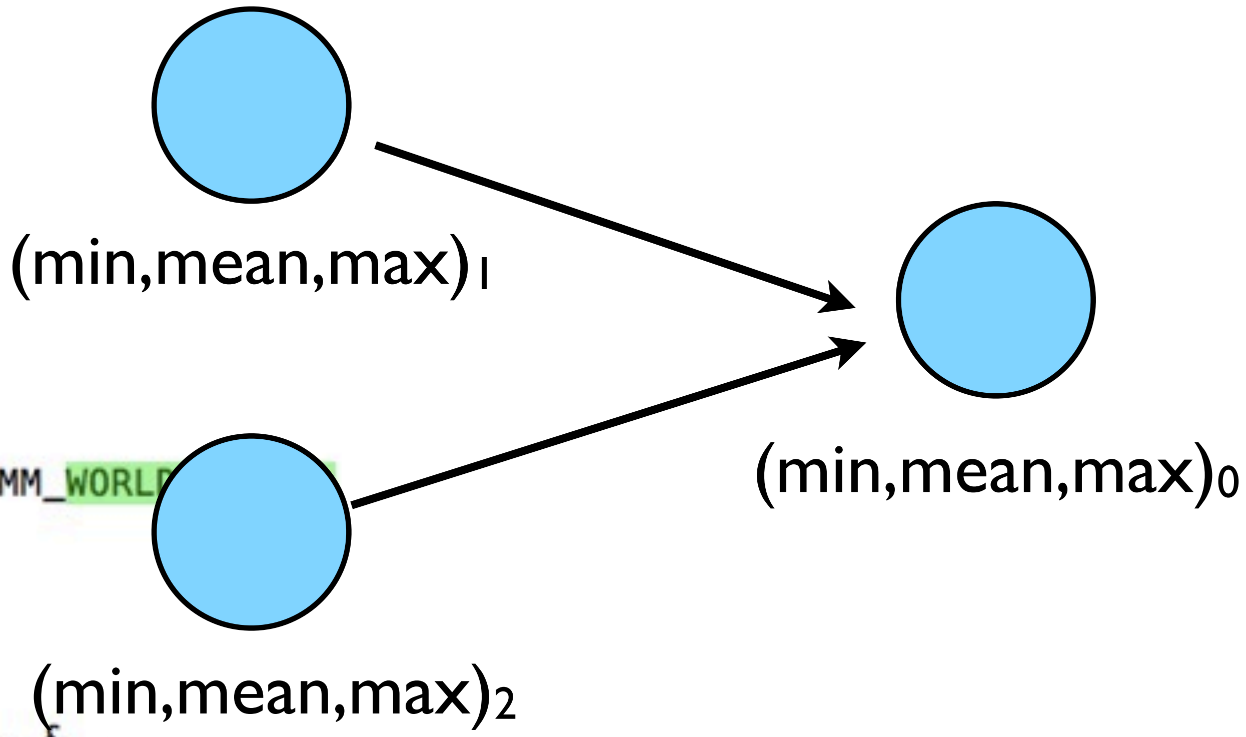
- Lets try some code that calculates the min/mean/max of a bunch of random numbers -1..1. Should go to -1,0,+1 for large N.
- Each gets their partial results and sends it to some node, say node 0 (why node 0?)
- ~/mpi/mpi-intro/minmeanmax.{c,f90}
- How to MPI it?

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

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

!
! random data
!
          allocate(dat(nx))
          call random_seed(put=[(i,i=1,8)])
          call random_number(dat)
          dat = 2*dat - 1.

!
! find min/mean/max
!
          datamin = minval(dat)
          datamax = maxval(dat)
          datamean= (1.*sum(dat))/nx

          deallocate(dat)

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

          return
          end
```

```c
/*
 * generate random data
 */

dat = (float *)malloc(nx * sizeof(float));
srand(0);
for (i=0;i<nx;i++) {
    dat[i] = 2*((float)rand()/RAND_MAX)-1.;
}

/*
 * find min/mean/max
 */

datamin = 1e+19;
datamax =-1e+19;
datamean = 0;


for (i=0;i<nx;i++) {
    if (dat[i] < datamin) datamin=dat[i];
    if (dat[i] > datamax) datamax=dat[i];
    datamean += dat[i];
}
datamean /= nx;
free(dat);

printf("Min/mean/max = %f,%f,%f\n", datamin,datamean,datamax);
```

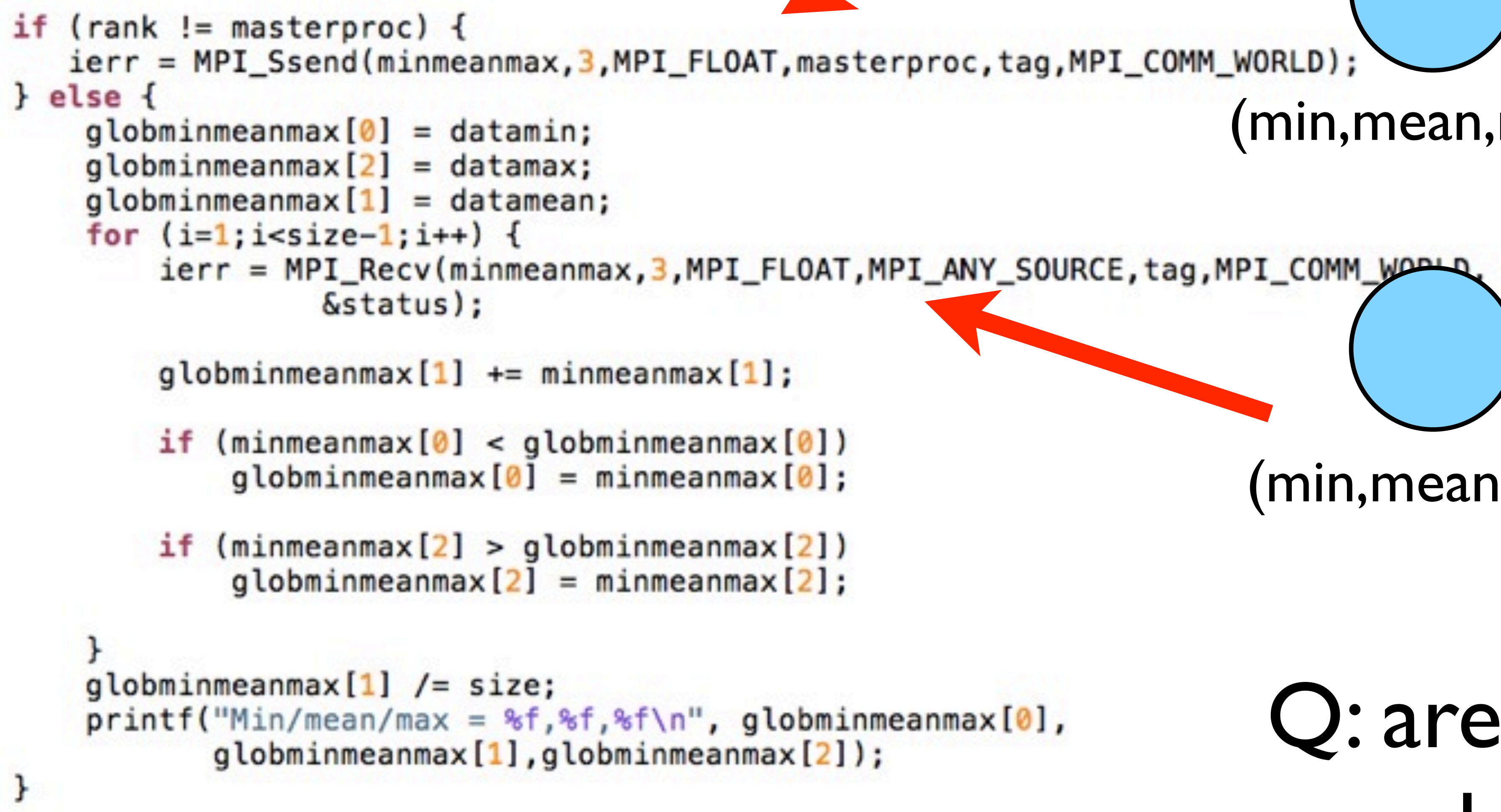


```
datamin = minval(dat)
datamax = maxval(dat)
datamean= (1.*sum(dat))/nx
deallocate(dat)

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

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

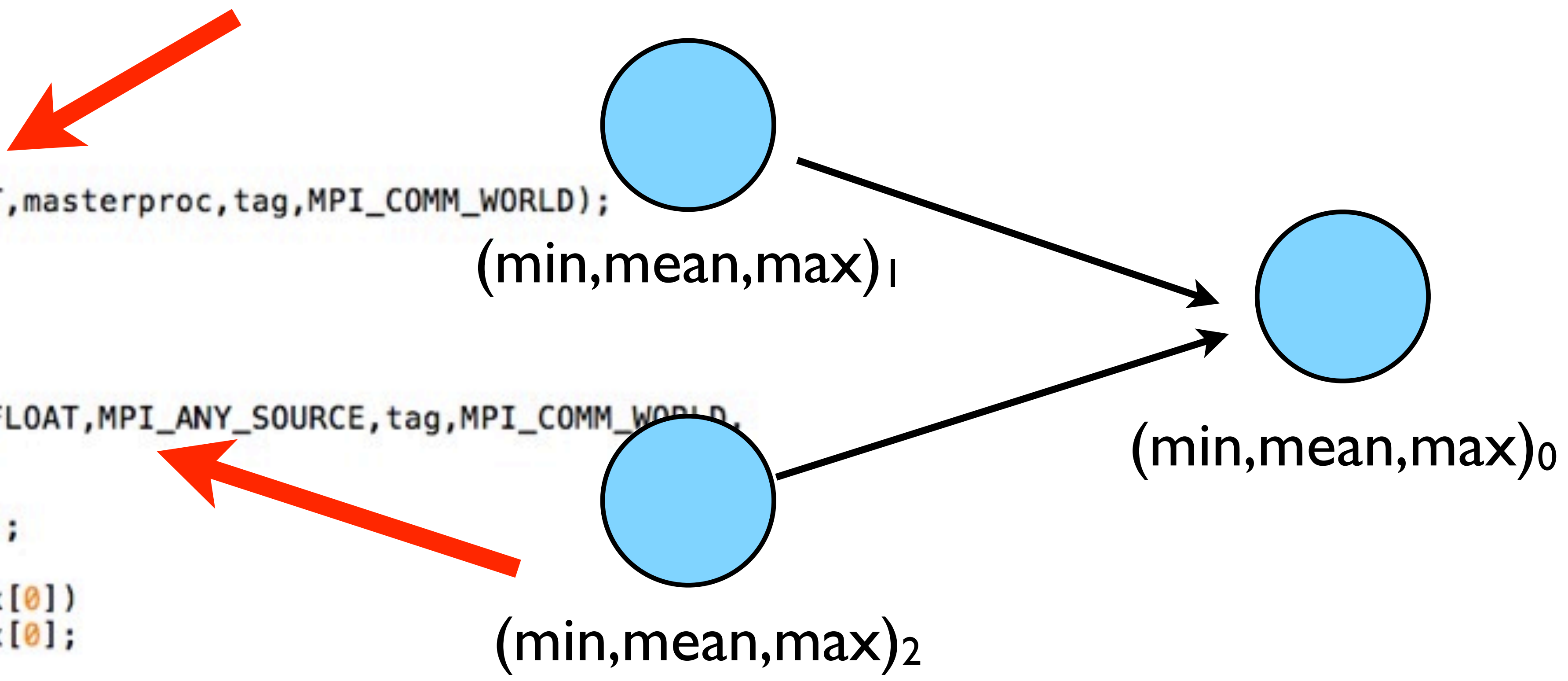


Q: are these sends/recvd adequately paired?

minmeanmax-mpi.f90



```c
if (rank != masterproc) {
   ierr = MPI_Ssend(minmeanmax,3,MPI_FLOAT,masterproc,tag,MPI_COMM_WORLD);
} else {
    globminmeanmax[0] = datamin;
    globminmeanmax[2] = datamax;
    globminmeanmax[1] = datamean;
    for (i=1;i<size-1;i++) {
        ierr = MPI_Recv(minmeanmax,3,MPI_FLOAT,MPI_ANY_SOURCE,tag,MPI_COMM_WORLD,
                 &status);

        globminmeanmax[1] += minmeanmax[1];

        if (minmeanmax[0] < globminmeanmax[0])
            globminmeanmax[0] = minmeanmax[0];

        if (minmeanmax[2] > globminmeanmax[2])
            globminmeanmax[2] = minmeanmax[2];

    }
    globminmeanmax[1] /= size;
    printf("Min/mean/max = %f,%f,%f\n", globminmeanmax[0],
           globminmeanmax[1],globminmeanmax[2]);
}
```

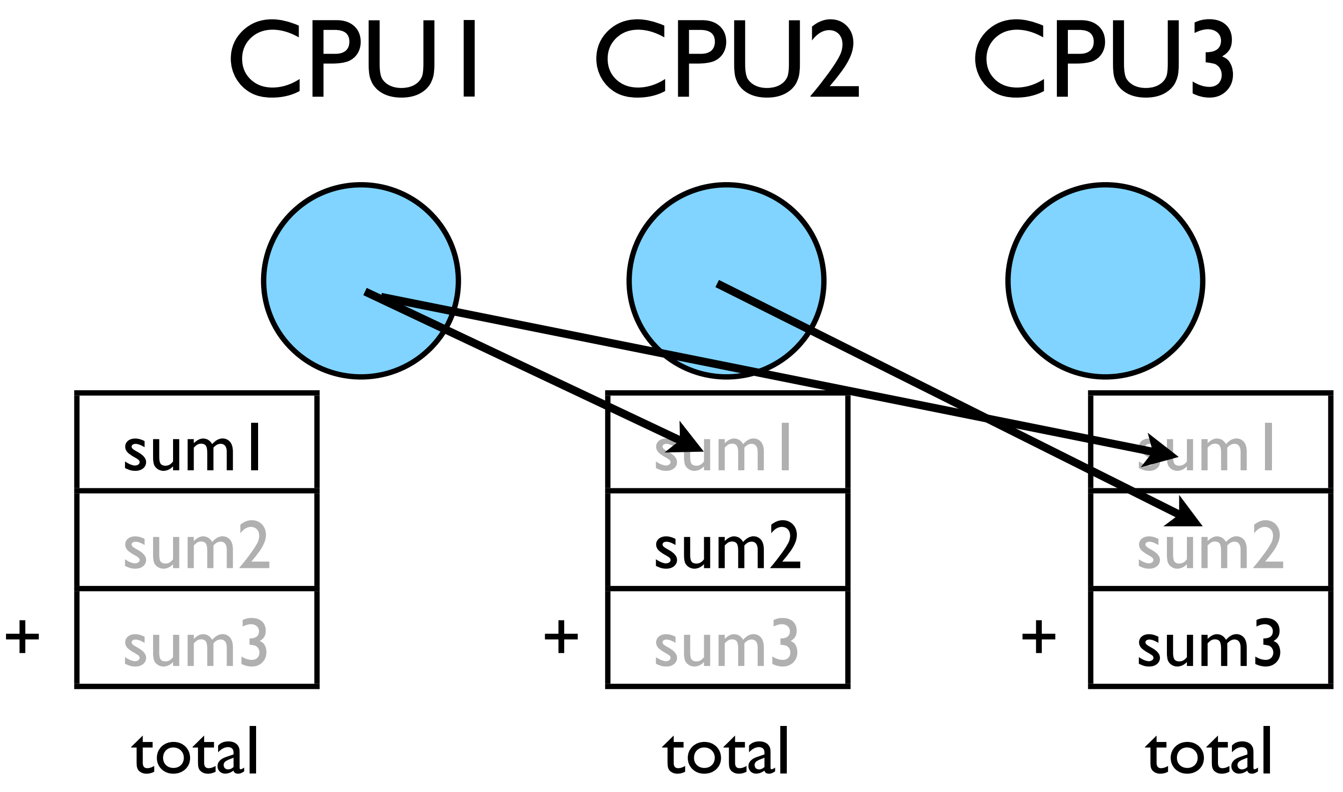


Q: are these sends/recvd adequately paired?

minmeanmax-mpi.c

Inefficient!

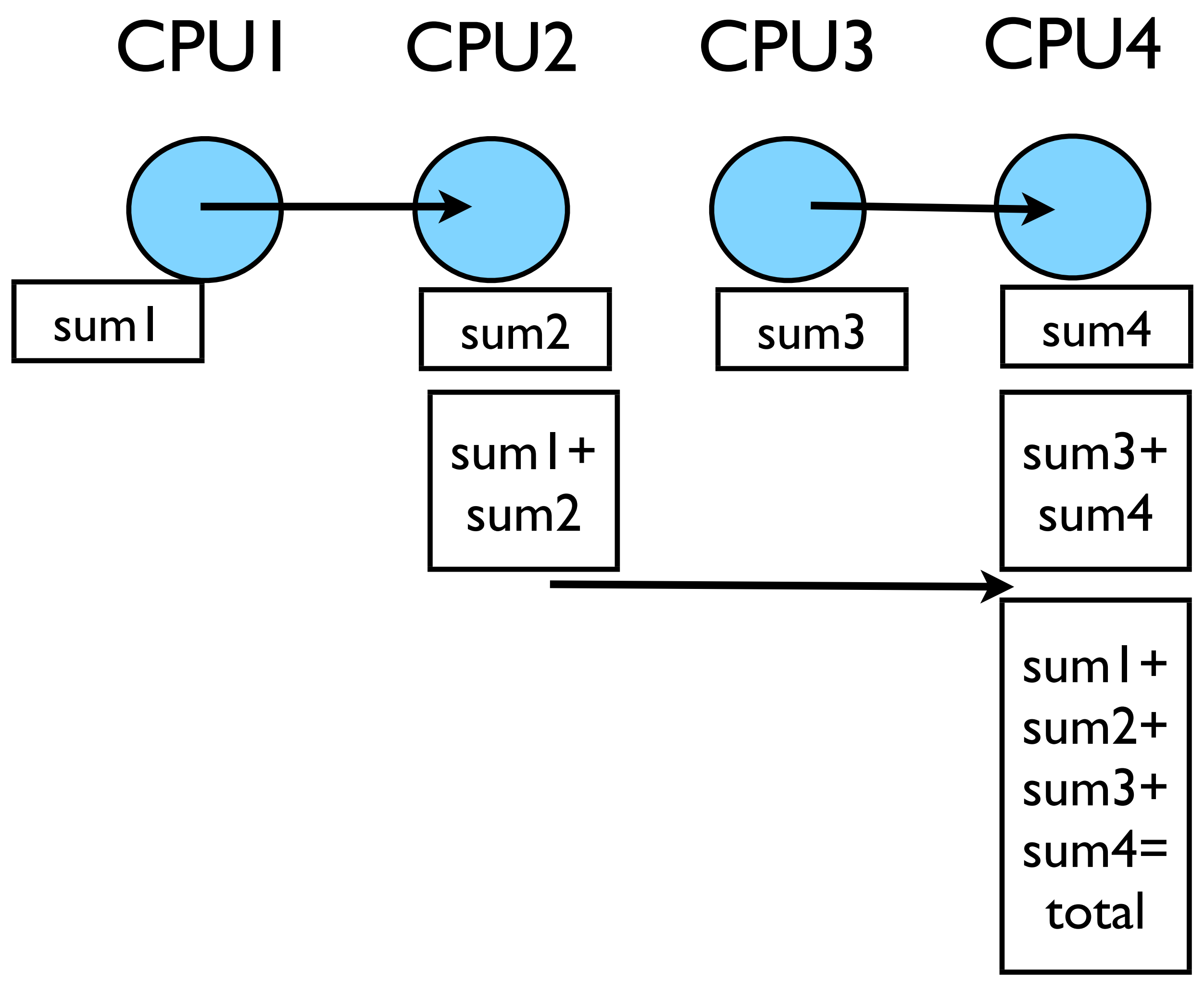
- Requires (P-1) messages, 2(P-1) if everyone then needs to get the answer.



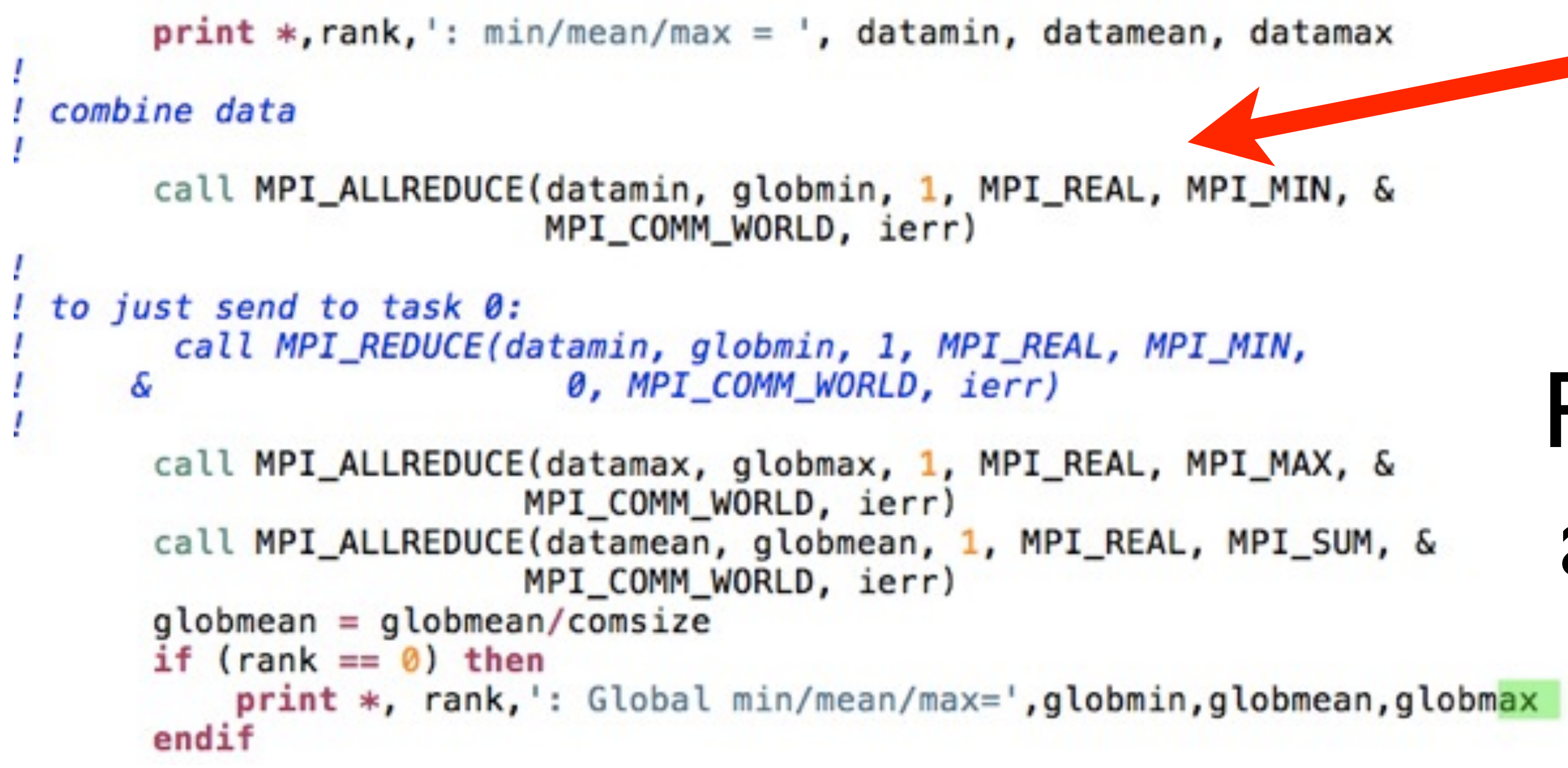
Better Summing

- Pairs of processors; send partial sums
- Max messages received $\log_2(P)$
- Can repeat to send total back

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



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



```
      print *,rank,': min/mean/max = ', datamin, datamean, datamax
!
! combine data
!
      call MPI_ALLREDUCE(datamin, globmin, 1, MPI_REAL, MPI_MIN, &
                         MPI_COMM_WORLD, ierr)
!
! to just send to task 0:
!       call MPI_REDUCE(datamin, globmin, 1, MPI_REAL, MPI_MIN,
!     &                       0, MPI_COMM_WORLD, ierr)
!
      call MPI_ALLREDUCE(datamax, globmax, 1, MPI_REAL, MPI_MAX, &
                        MPI_COMM_WORLD, ierr)
      call MPI_ALLREDUCE(datamean, globmean, 1, MPI_REAL, MPI_SUM, &
                        MPI_COMM_WORLD, ierr)
      globmean = globmean/comsize
      if (rank == 0) then
          print *, rank,': Global min/mean/max=',globmin,globmean,globmax
      endif
```

MPI_Reduce and MPI_Allreduce

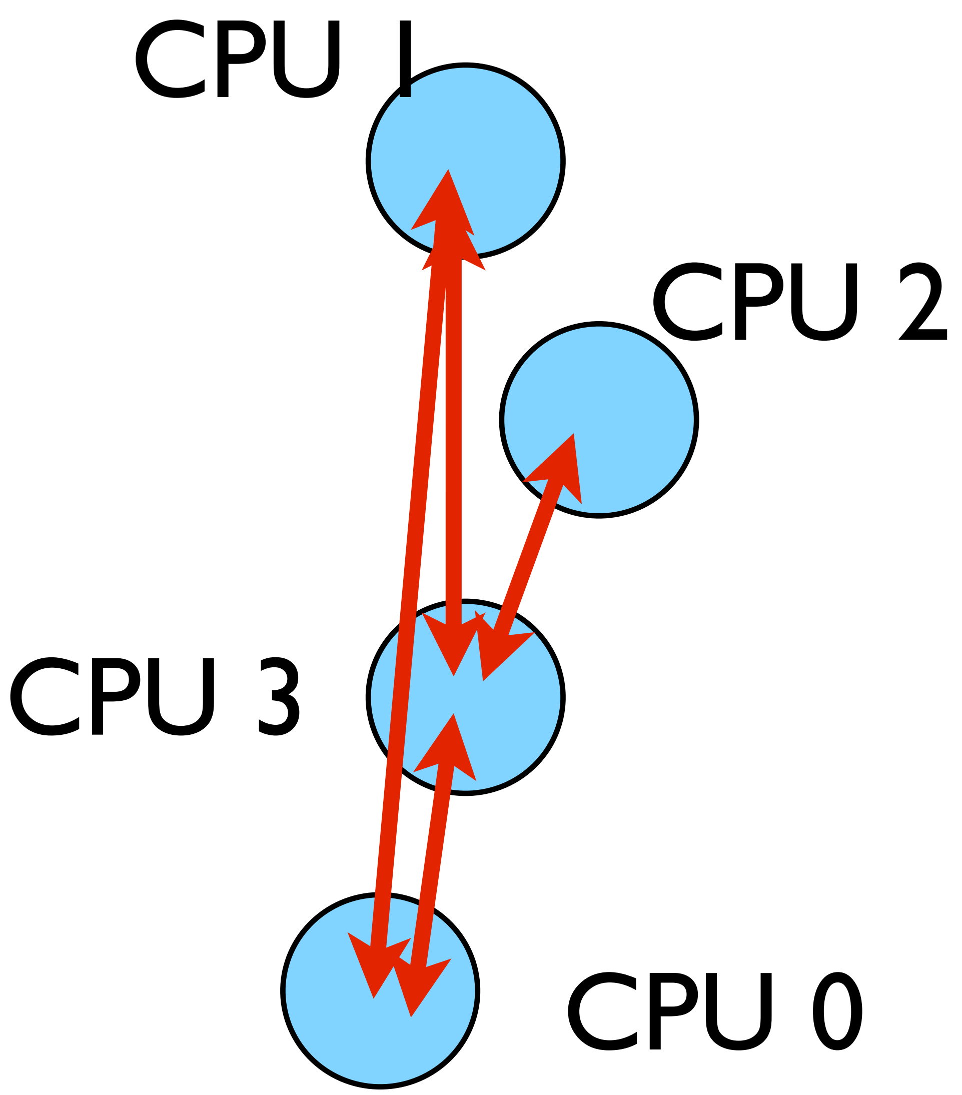
Performs a reduction and sends answer to one PE (Reduce) or all PEs (Allreduce)

minmeanmax-allreduce.f

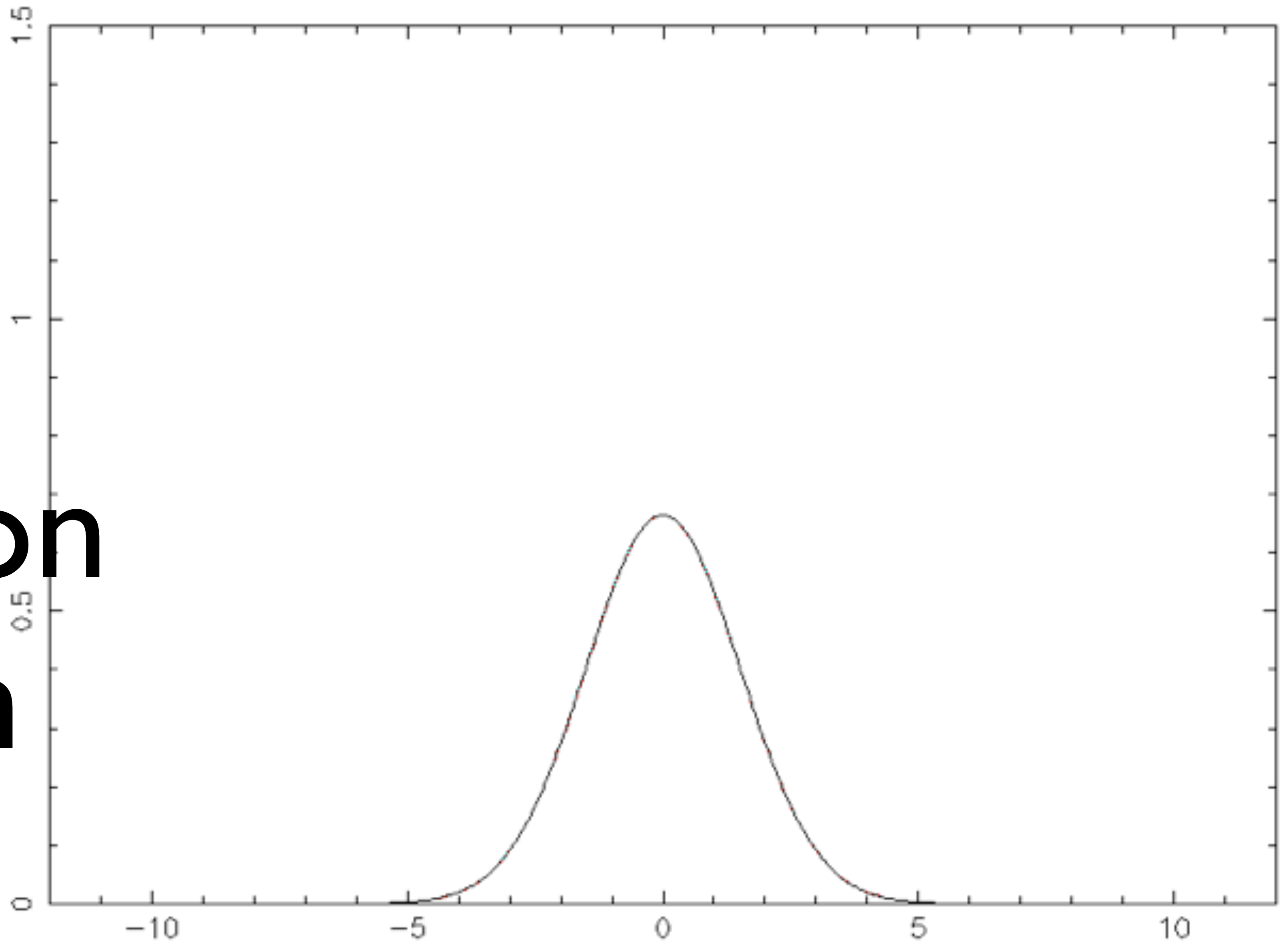
SciNet

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'



1d diffusion equation

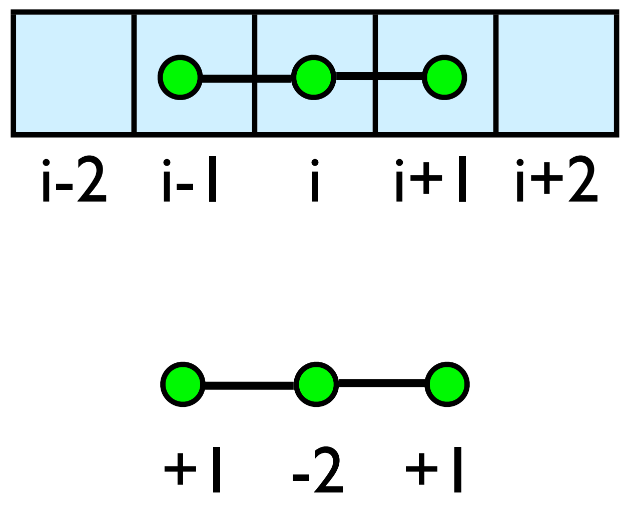


```
cd mpi/diffusion .
make diffusionf or make diffusionc
./diffusionf or ./diffusionc
```

Discretizing Derivatives

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

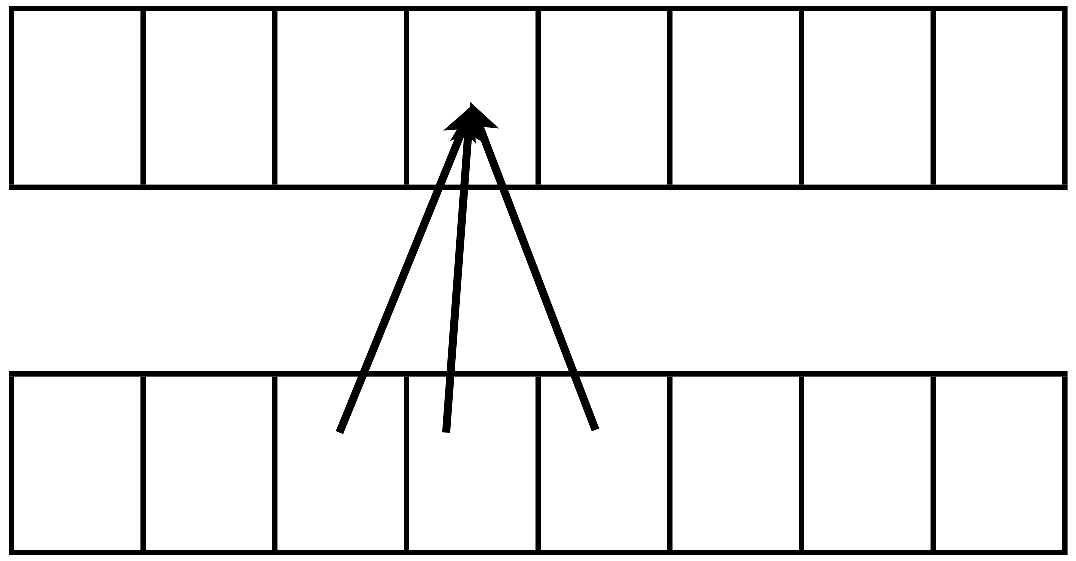
- Done by finite differencing the discretized values
- Implicitly or explicitly involves interpolating data and taking derivative of the interpolant
- More accuracy - larger ‘stencils’



Diffusion Equation

$$\begin{aligned}
\frac{\partial T}{\partial t} &= D\frac{\partial^2 T}{\partial x^2} \\
\frac{\partial T_i^{(n)}}{\partial t} &\approx \frac{T_i^{(n)} + T_i^{(n-1)}}{\Delta t} \\
\frac{\partial T_i^{(n)}}{\partial x} &\approx \frac{T_{i+1}^{(n)} - 2T_i^{(n)} + T_{i-1}^{(n)}}{\Delta x^2} \\
T_i^{(n+1)} &\approx T_i^{(n)} + \frac{D\Delta t}{\Delta x^2}\left(T_{i+1}^{(n)} - 2T_i^{(n)} + T_{i-1}^{(n)}\right)
\end{aligned}$$

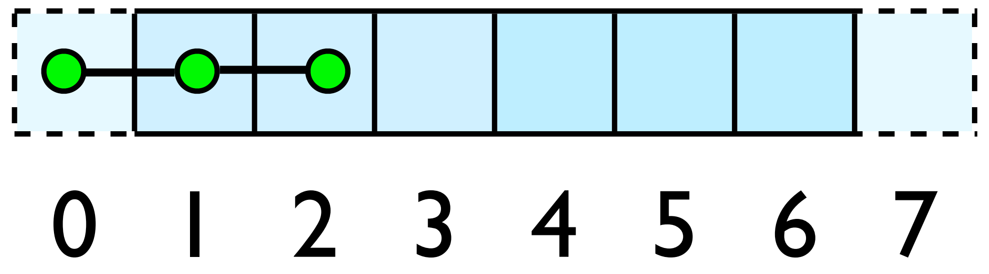
- Simple 1d PDE
- Each timestep, new data for T[i] requires old data for T[i+1],T[i],T[i-1]



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

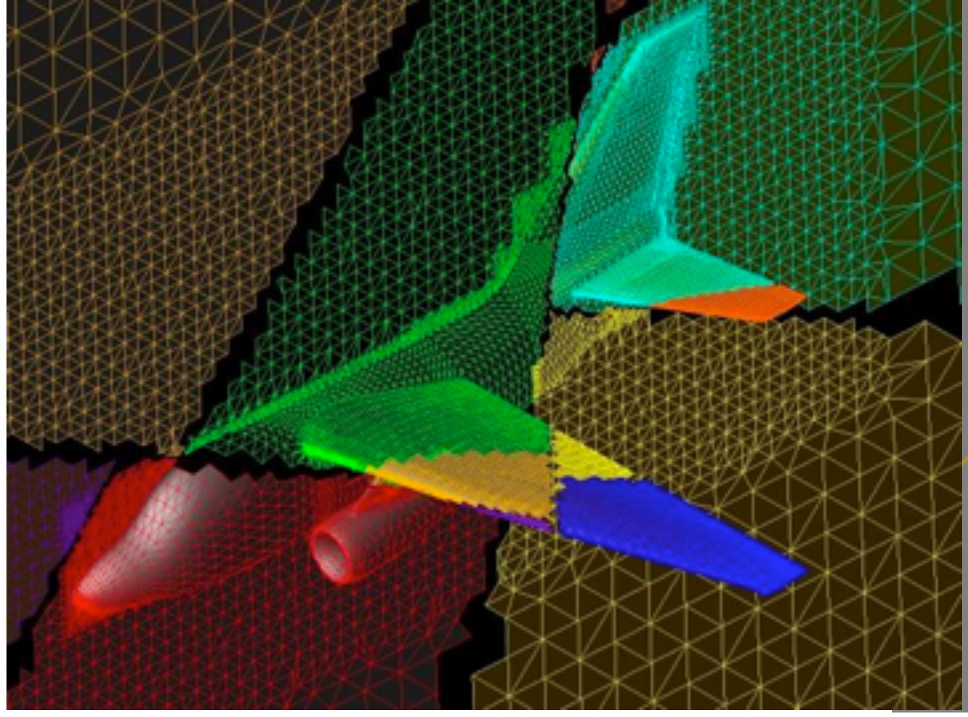
Global Domain

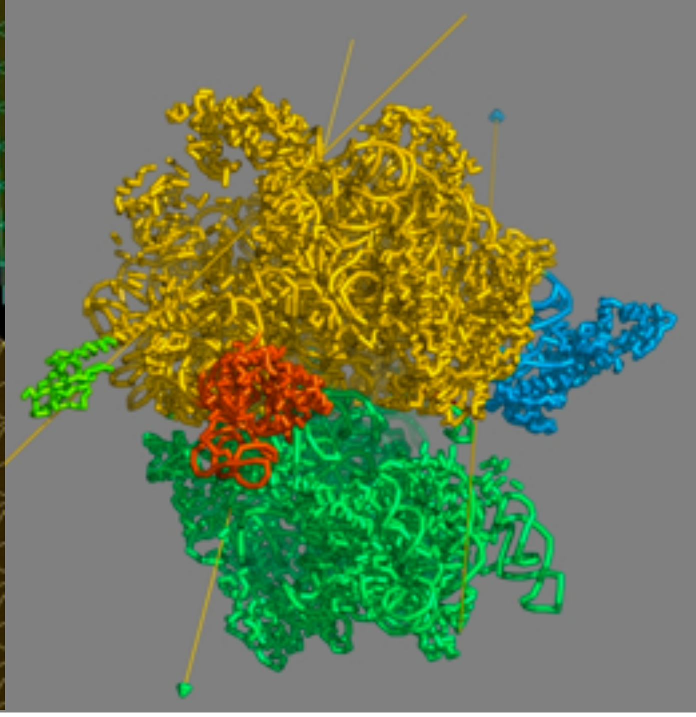


ng = 1

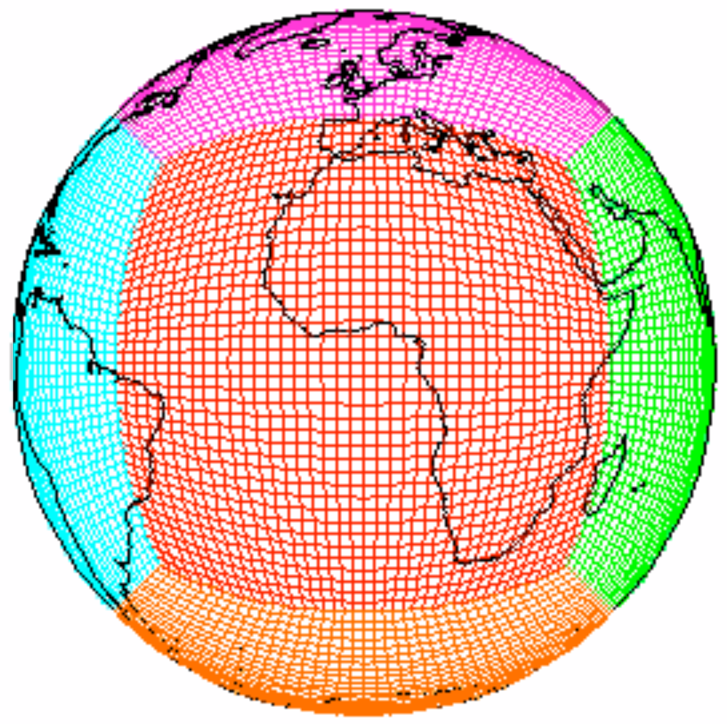
loop from ng, N - 2 ng

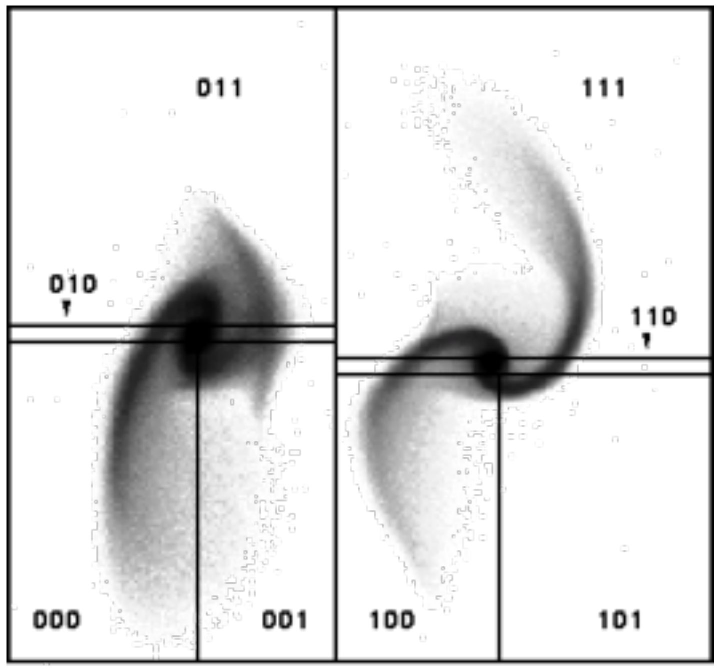
Domain Decomposition


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

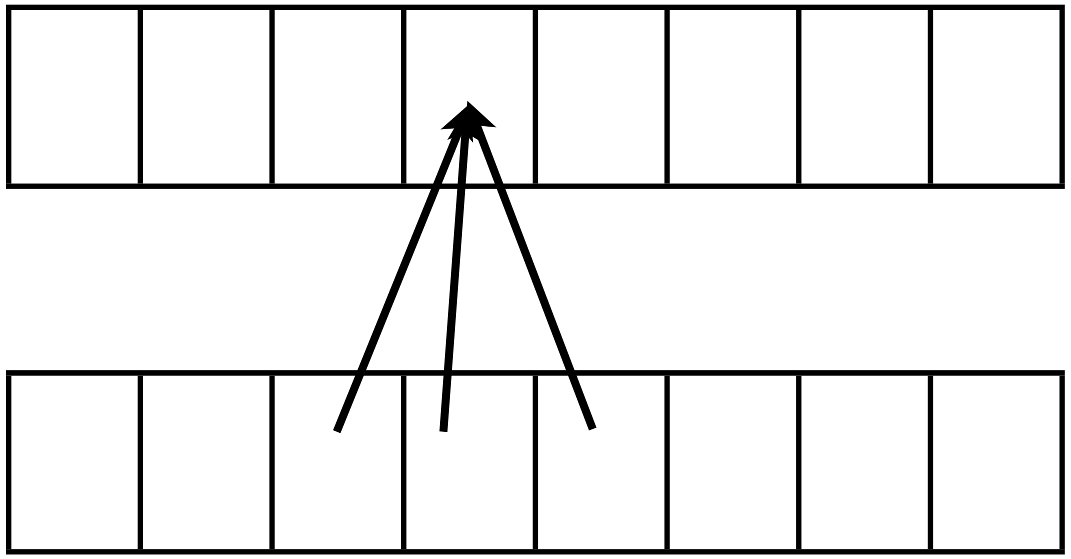


http://www.cita.utoronto.ca/~dubinski /treecode/node8.html

Implement a diffusion equation in MPI

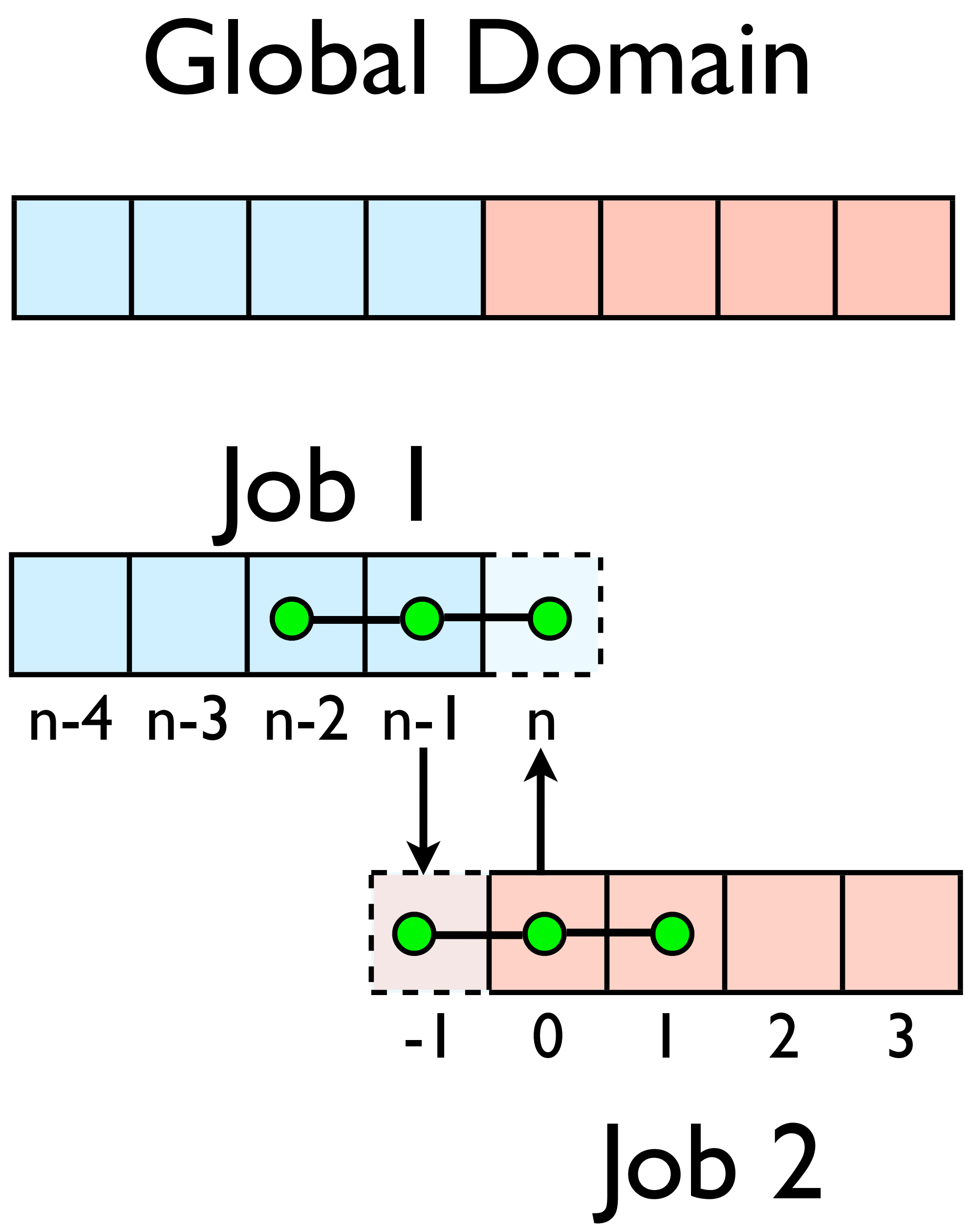
$$\begin{aligned}\frac{dT}{dt} &= D\frac{d^2T}{dx^2}\\ T_i^{n+1} &= T_i^n + \frac{D\Delta t}{\Delta x^2}\left(T_{i+1}^n - 2T_i^n + T_{i-1}^n\right)\end{aligned}$$

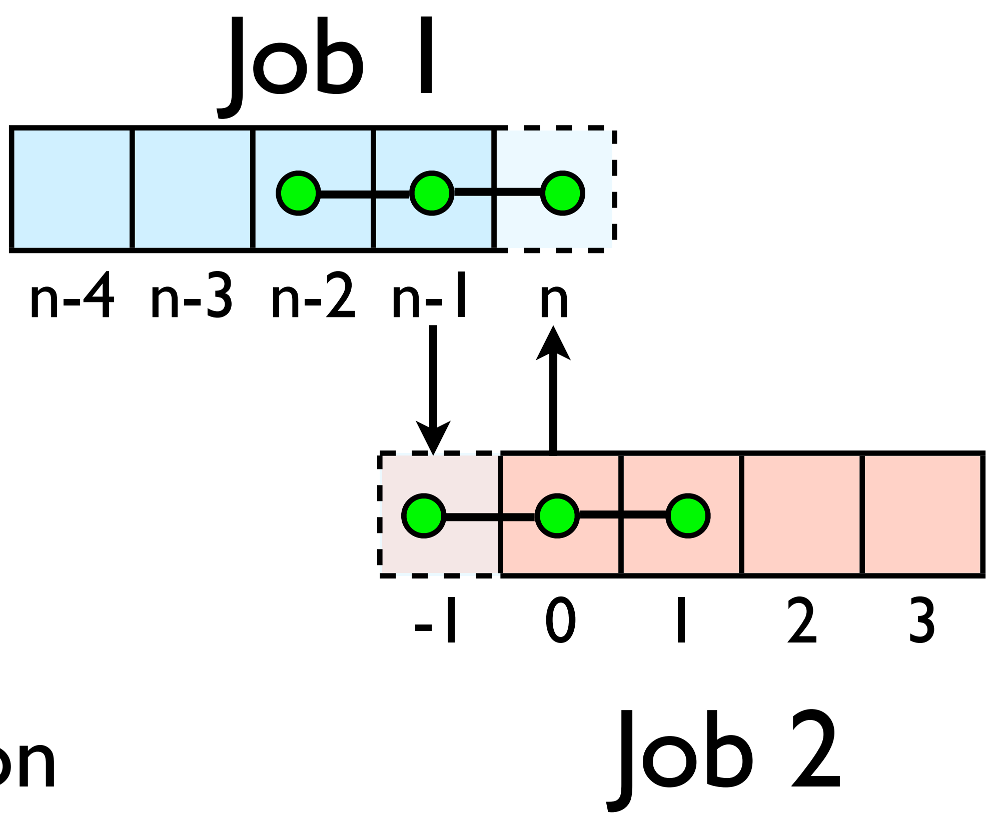
- Need one neighboring number per neighbor per timestep



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
- Hydro code: need guardcells 2 deep





- Do computation
- guardcell exchange: each cell has to do 2 sendrecvs
  - its rightmost cell with neighbors leftmost
  - its leftmost cell with neighbors rightmost
  - Everyone do right-filling first, then left-filling (say)
  - For simplicity, start with periodic BCs
  - then (re-)implement fixed-temperature BCs; temperature in first, last zones are fixed

Hands-on: MPI diffusion

- cp diffusionf.f90 diffusionf-mpi.f90 or
- cp diffusionc.c diffusionc-mpi.c or
- Make an MPI-ed version of diffusion equation
- (Build: `make diffusionf-mpi` or `make diffusionc-mpi`)
- Test on 1..8 procs

- add standard MPI calls: init, finalize, comm_size, comm_rank
- Figure out how many points PE is responsible for (~totpoints/size)
- Figure out neighbors
- Start at 1, but end at totpoints/size
- At end of step, exchange guardcells; use sendrecv
- Get total error

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,
                    recvptr, count, MPI_TYPE, source, tag,
                    Communicator, &status);
ierr = MPI_Allreduce(&mydata, &globaldata, count, MPI_TYPE,
                     MPI_OP, Communicator);

Communicator -> MPI_COMM_WORLD
MPI_Type -> MPI_FLOAT, MPI_DOUBLE, MPI_INT, MPI_CHAR...
MPI_OP -> MPI_SUM, MPI_MIN, MPI_MAX,...
```

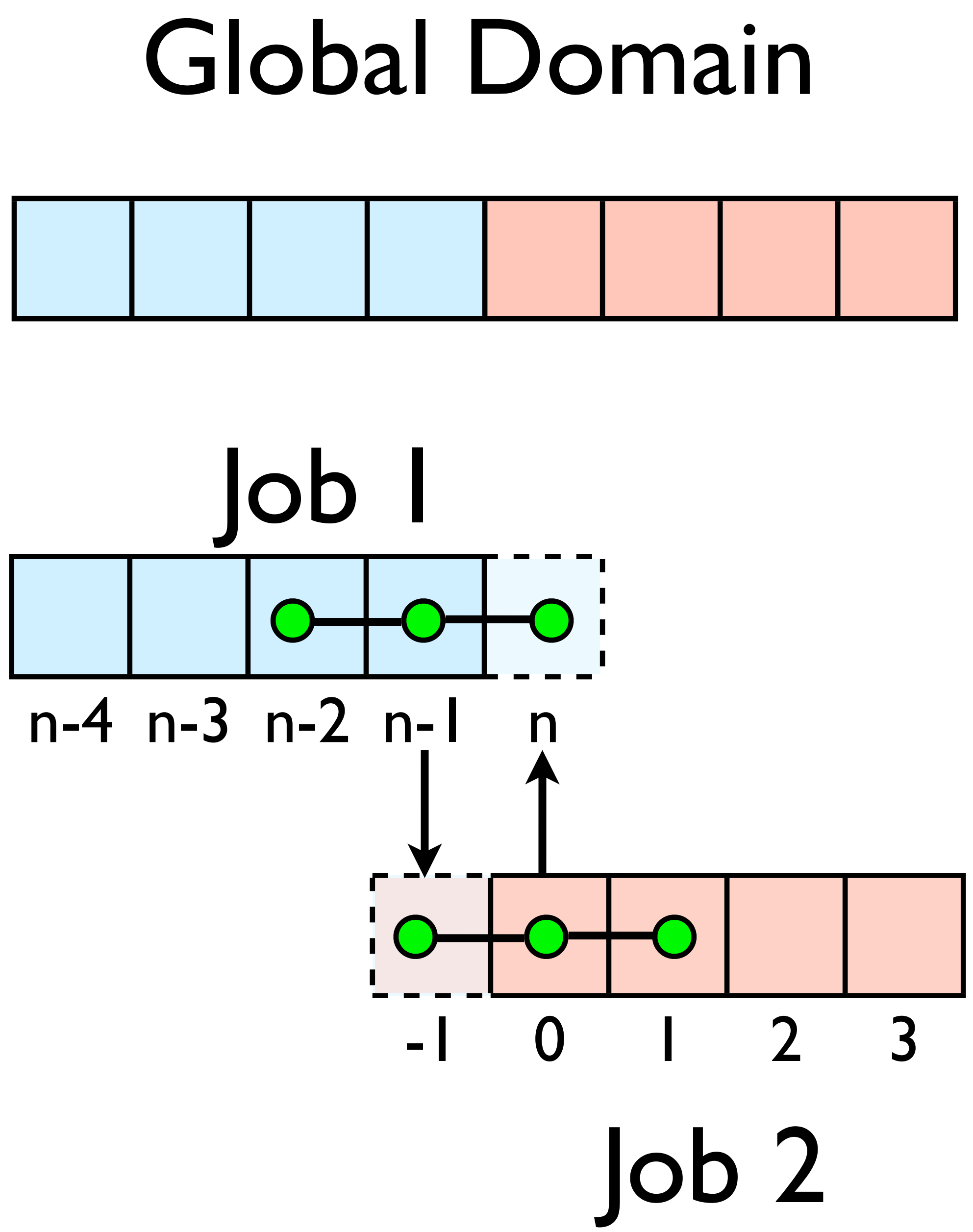
FORTRAN syntax

```
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)
call MPI_RECV(rcvarr, 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
MPI_Type -> MPI_REAL, MPI_DOUBLE_PRECISION,
           MPI_INTEGER, MPI_CHARACTER
MPI_OP -> MPI_SUM, MPI_MIN, MPI_MAX,...
```

Non-blocking communications

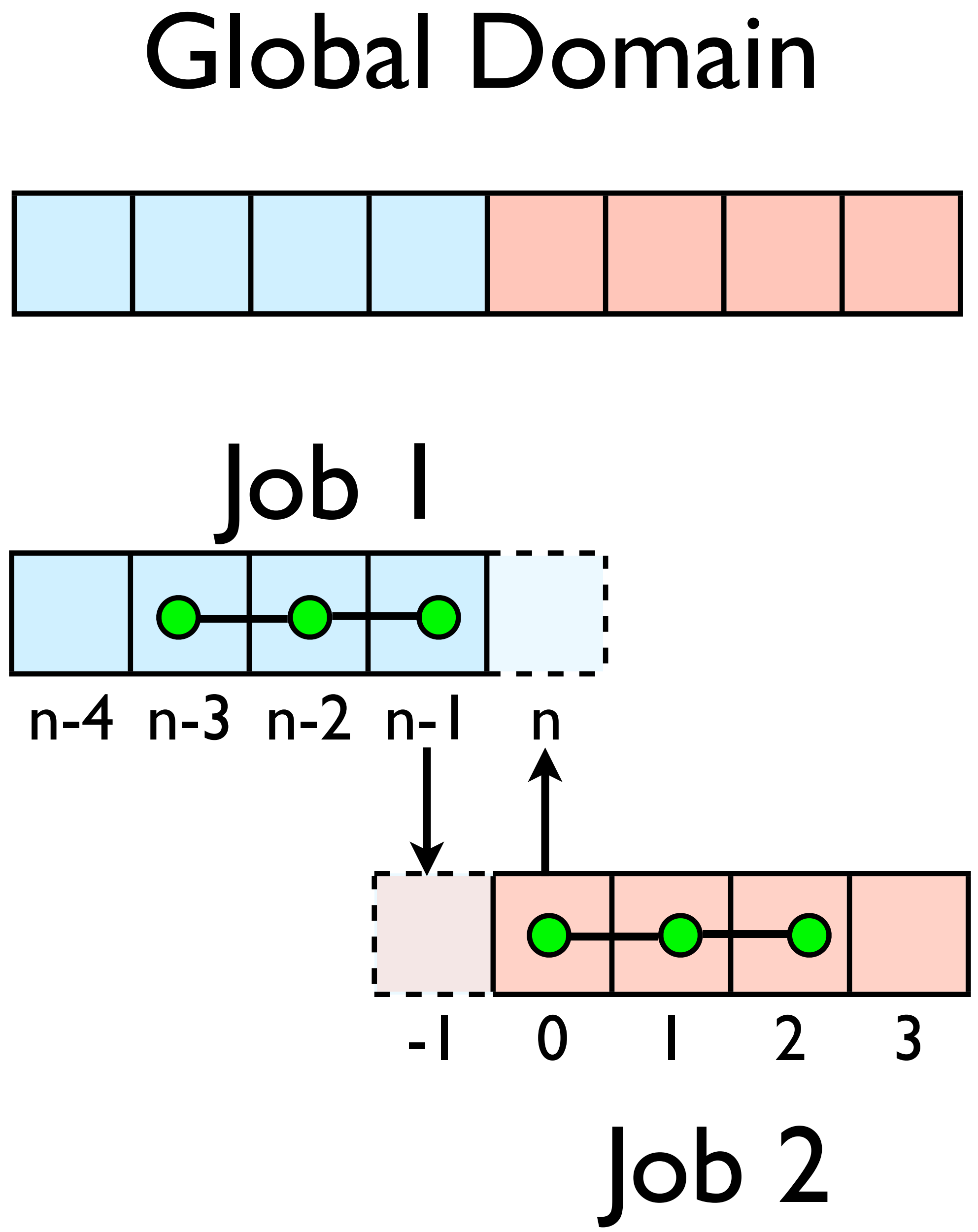
Diffusion: Had to wait for communications to compute

- Could not compute end points without guardcell data
- All work halted while all communications occurred
- Significant parallel overhead



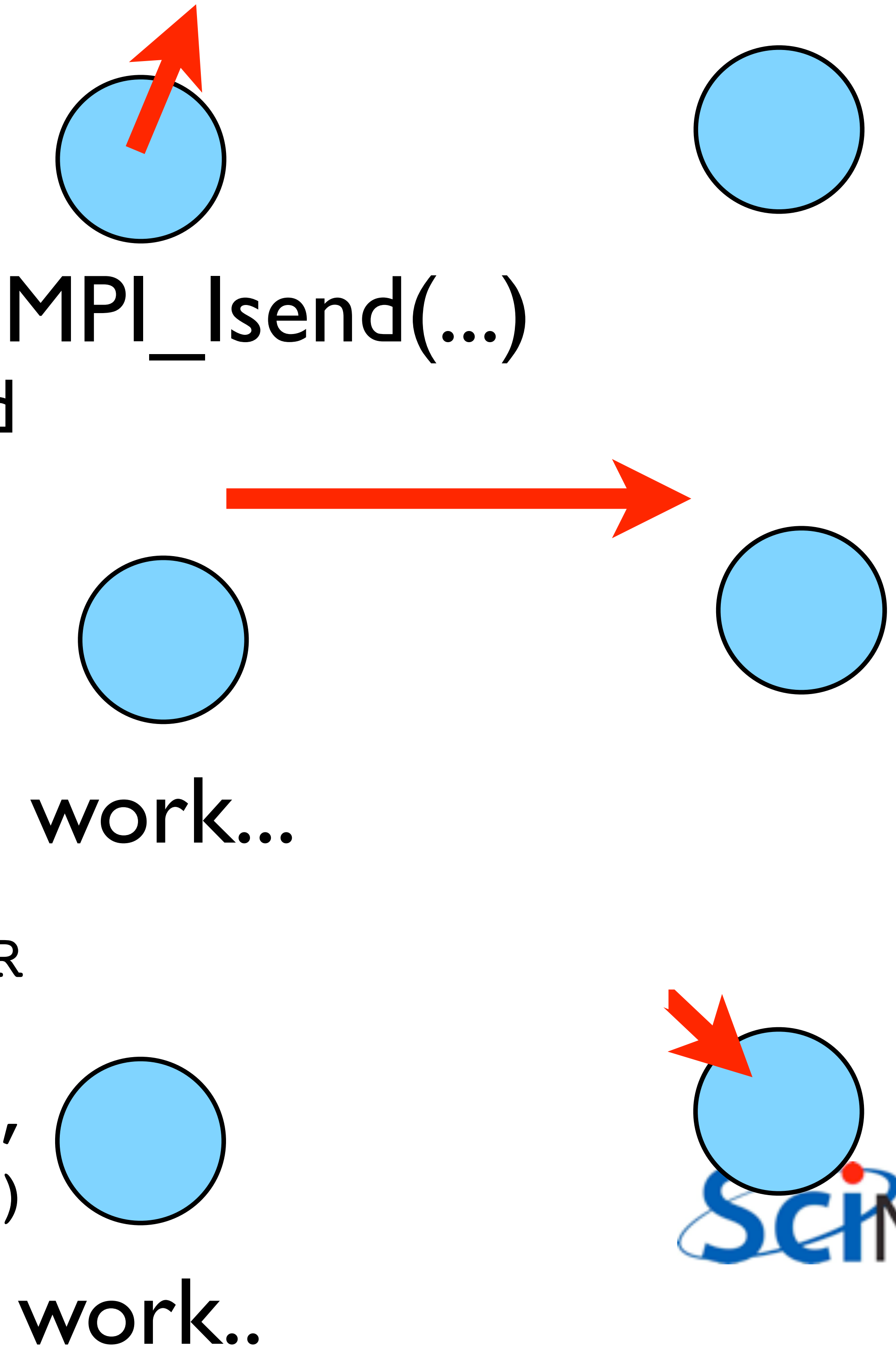
Diffusion: *Had* to wait?

- But inner zones could have been computed just fine
- Ideally, would do inner zones work while communications is being done; then go back and do end points.



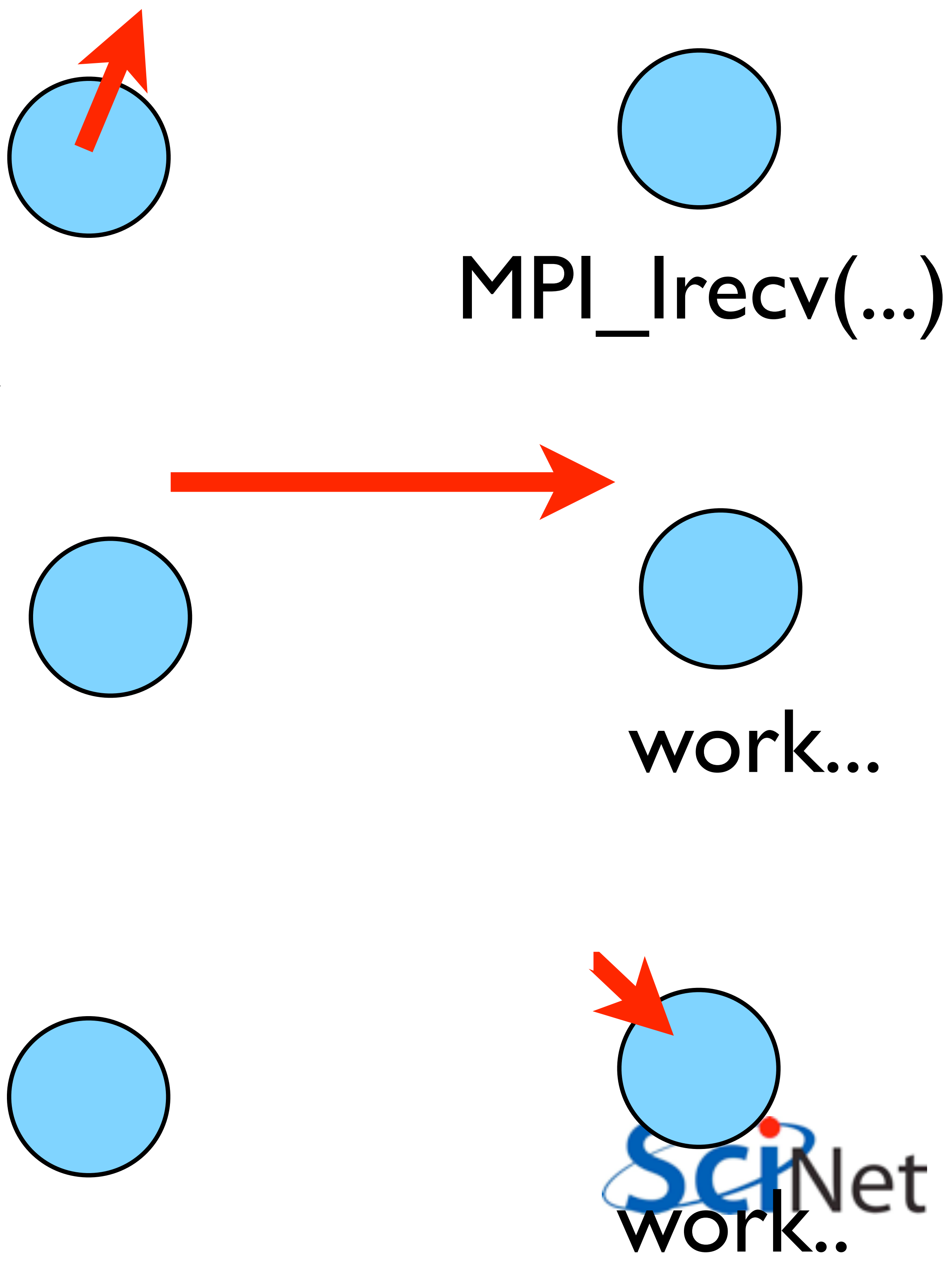
Nonblocking Sends

- Allows you to get work done while message is ‘in flight’
- Must **not** alter send buffer until send has completed.
- C: `MPI_Isend( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, `**`MPI_Request *request`**` )`
- FORTRAN: `MPI_ISEND(BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG, INTEGER COMM, `**`INTEGER REQUEST`**`,INTEGER IERROR)`



Nonblocking Recv

- Allows you to get work done while message is 'in flight'
- Must **not** access recv buffer until recv has completed.
- C: `MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, `**`MPI_Request *request`**` )`
- FORTRAN: `MPI_IREV(BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER SOURCE,INTEGER TAG, INTEGER COMM, `**`INTEGER REQUEST`**`,INTEGER IERROR)`



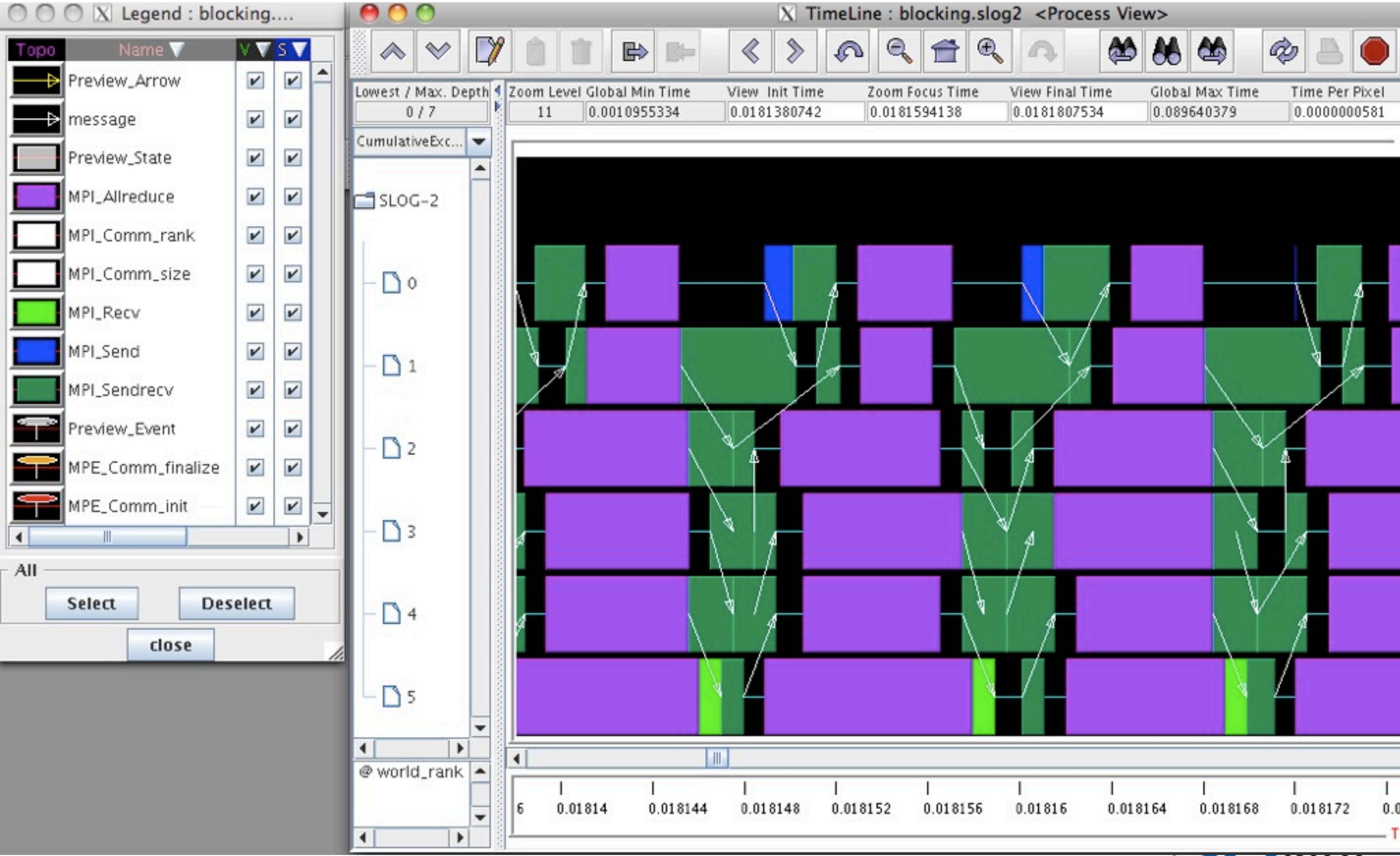
SciNet

How to tell if message is completed?

- `int MPI_Wait(MPI_Request *request,MPI_Status *status);`
- `MPI_WAIT(INTEGER REQUEST,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)`

- `int MPI_Waitall(int count,MPI_Request *array_of_requests, MPI_Status *array_of_statuses);`
- `MPI_WAITALL(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),INTEGER`

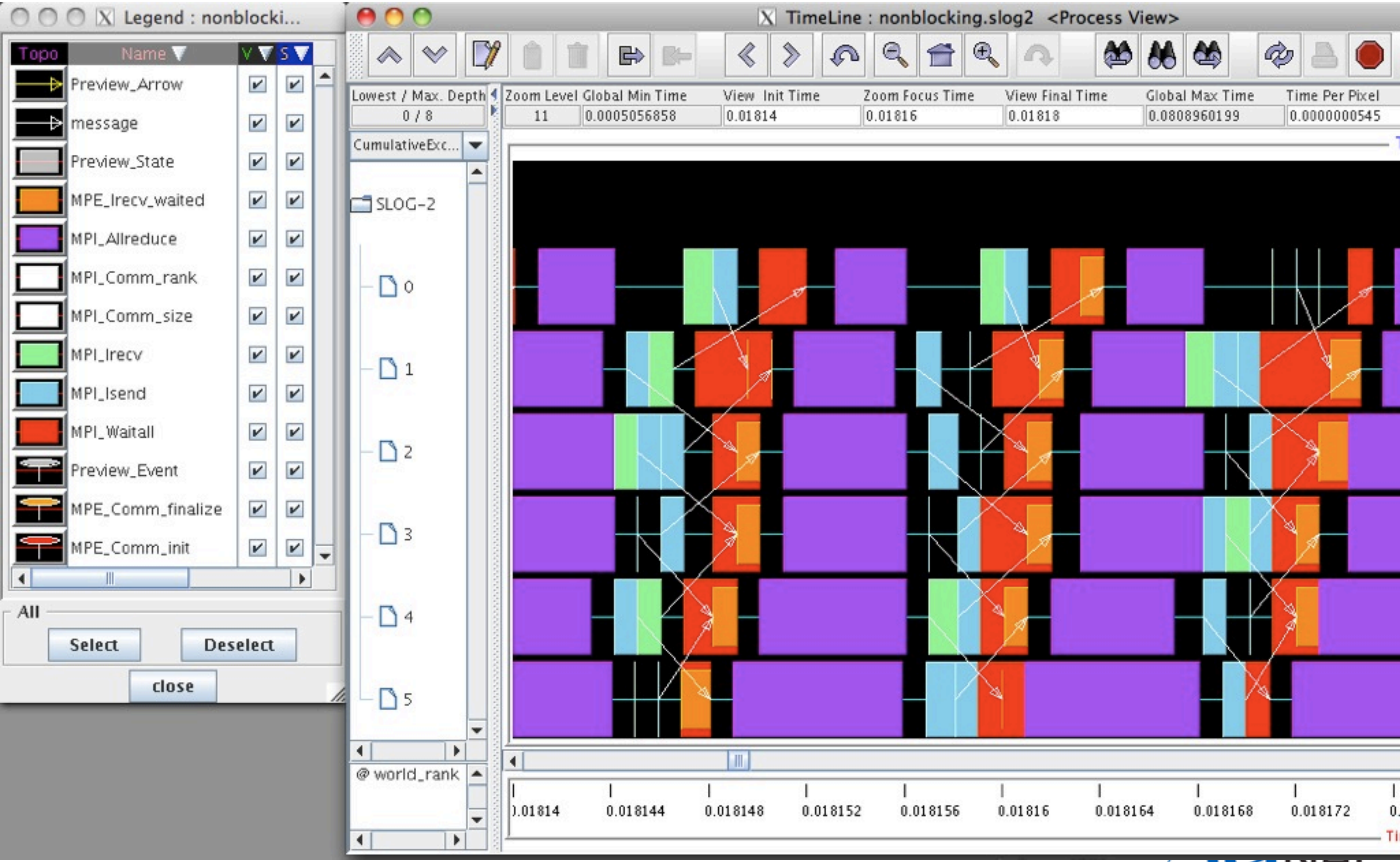
Also: MPI_Waitany, MPI_Test...


Legend : blocking....

Topo	Name	V	S
	Preview_Arrow	✓	✓
	message	✓	✓
▨	Preview_State	✓	✓
🟪	MPI_Allreduce	✓	✓
⬜	MPI_Comm_rank	✓	✓
⬜	MPI_Comm_size	✓	✓
🟩	MPI_Recv	✓	✓
🟦	MPI_Send	✓	✓
🟩	MPI_Sendrecv	✓	✓
	Preview_Event	✓	✓
	MPE_Comm_finalize	✓	✓
	MPE_Comm_init	✓	✓

TimeLine : blocking.slog2 <Process View>

Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
0 / 7	11	0.0010955334	0.0181380742	0.0181594138	0.0181807534	0.089640379	0.0000000581



Legend : nonblocki...

Topo	Name	V	S
	Preview_Arrow	✓	✓
	message	✓	✓
	Preview_State	✓	✓
🟧	MPE_Irecv_waited	✓	✓
🟪	MPI_Allreduce	✓	✓
⬜	MPI_Comm_rank	✓	✓
⬜	MPI_Comm_size	✓	✓
🟩	MPI_Irecv	✓	✓
🟦	MPI_Isend	✓	✓
🟥	MPI_Waitall	✓	✓
	Preview_Event	✓	✓
	MPE_Comm_finalize	✓	✓
	MPE_Comm_init	✓	✓

All

Select Deselect

close

TimeLine : nonblocking.slog2 <Process View>

Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
0 / 8	11	0.0005056858	0.01814	0.01816	0.01818	0.0808960199	0.0000000545

CumulativeExc...

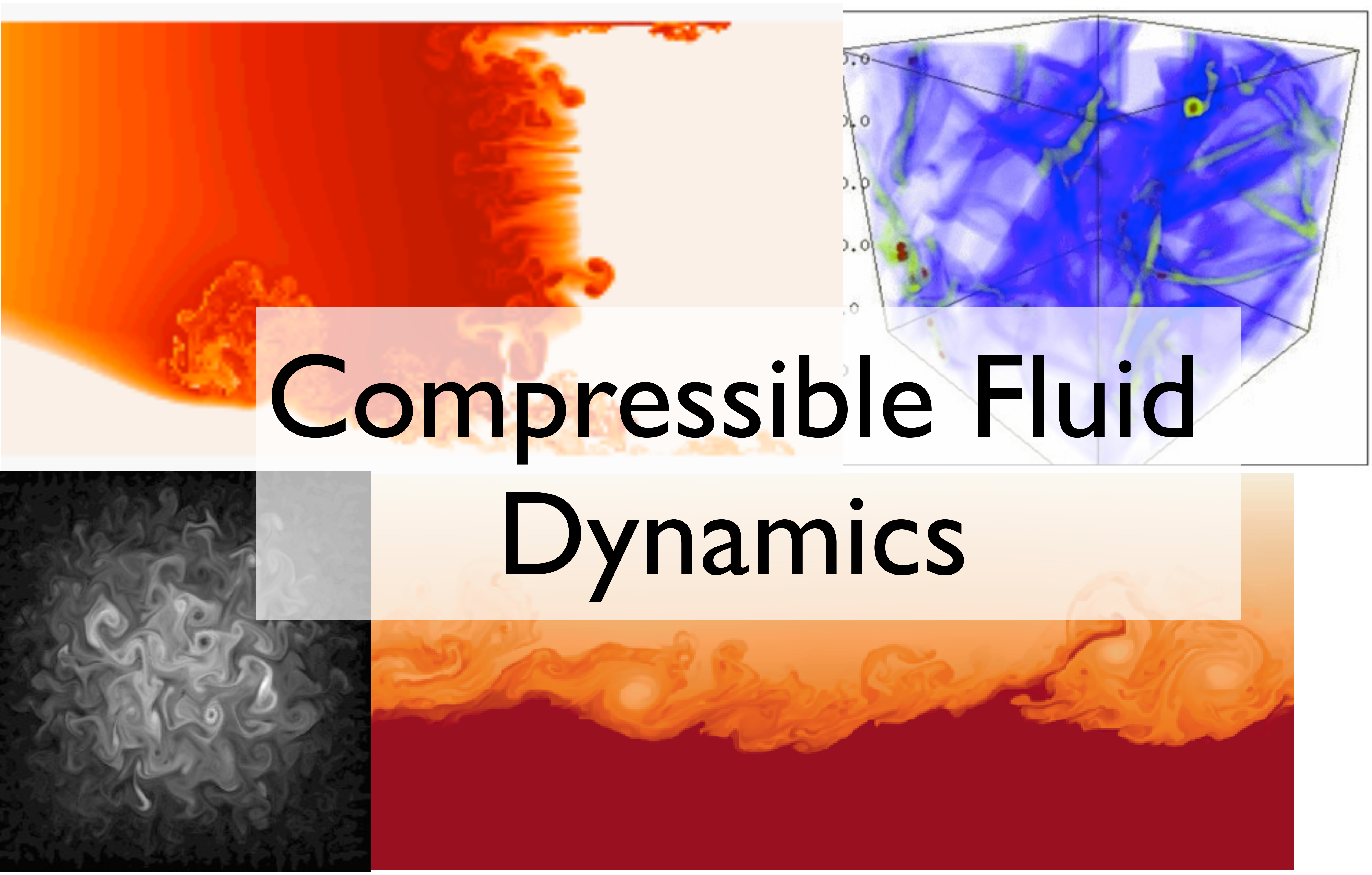
SLOG-2

0
1
2
3
4
5

@ world_rank

Hands On

- In diffusion directory, cp diffusion{c,f}-mpi.{c,f90} to diffusion{c,f}-mpi-nonblocking.{c,f90}
- Change to do non-blocking IO; post sends/recvs, do inner work, wait for messages to clear, do end points



Compressible Fluid Dynamics

Equations of Hydrodynamics

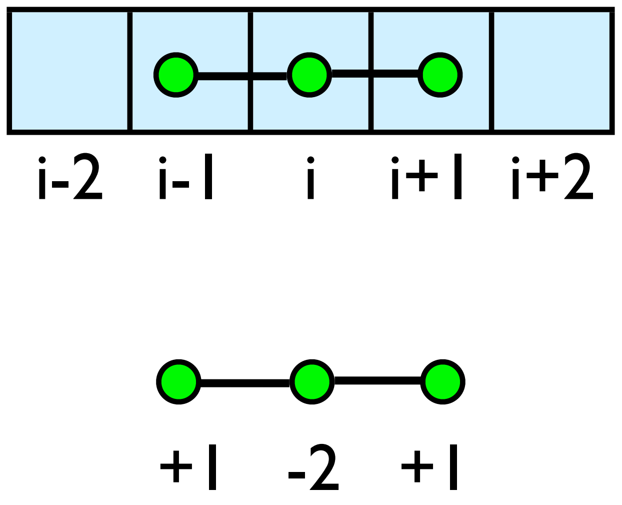
- Density, momentum, and energy equations
- Supplemented by an equation of state - pressure as a function of dens, energy

$$\begin{aligned}
\frac{\partial}{\partial t}\rho + \nabla \cdot (\rho \mathbf{v}) &= 0 \\
\frac{\partial}{\partial t}(\rho \mathbf{v}) + \nabla \cdot (\rho \mathbf{v}\mathbf{v}) &= -\nabla p \\
\frac{\partial}{\partial t}(\rho E) + \nabla \cdot ((\rho E + p)\, \mathbf{v}) &= 0
\end{aligned}$$

Discretizing Derivatives

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

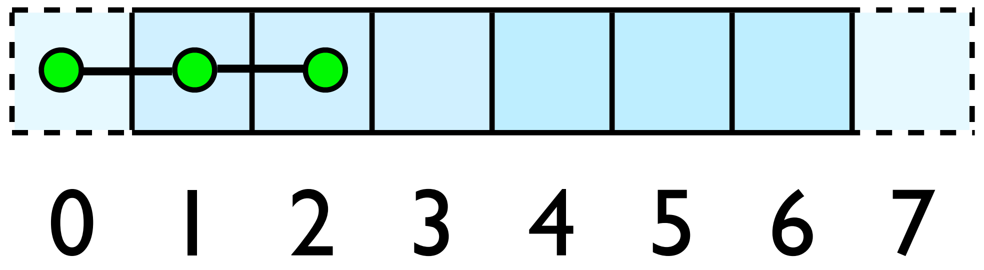
- Done by finite differencing the discretized values
- Implicitly or explicitly involves interpolating data and taking derivative of the interpolant
- More accuracy - larger 'stencils'



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 0th point in domain
- Fill guard cells with values such that the required boundary conditions are met

Global Domain

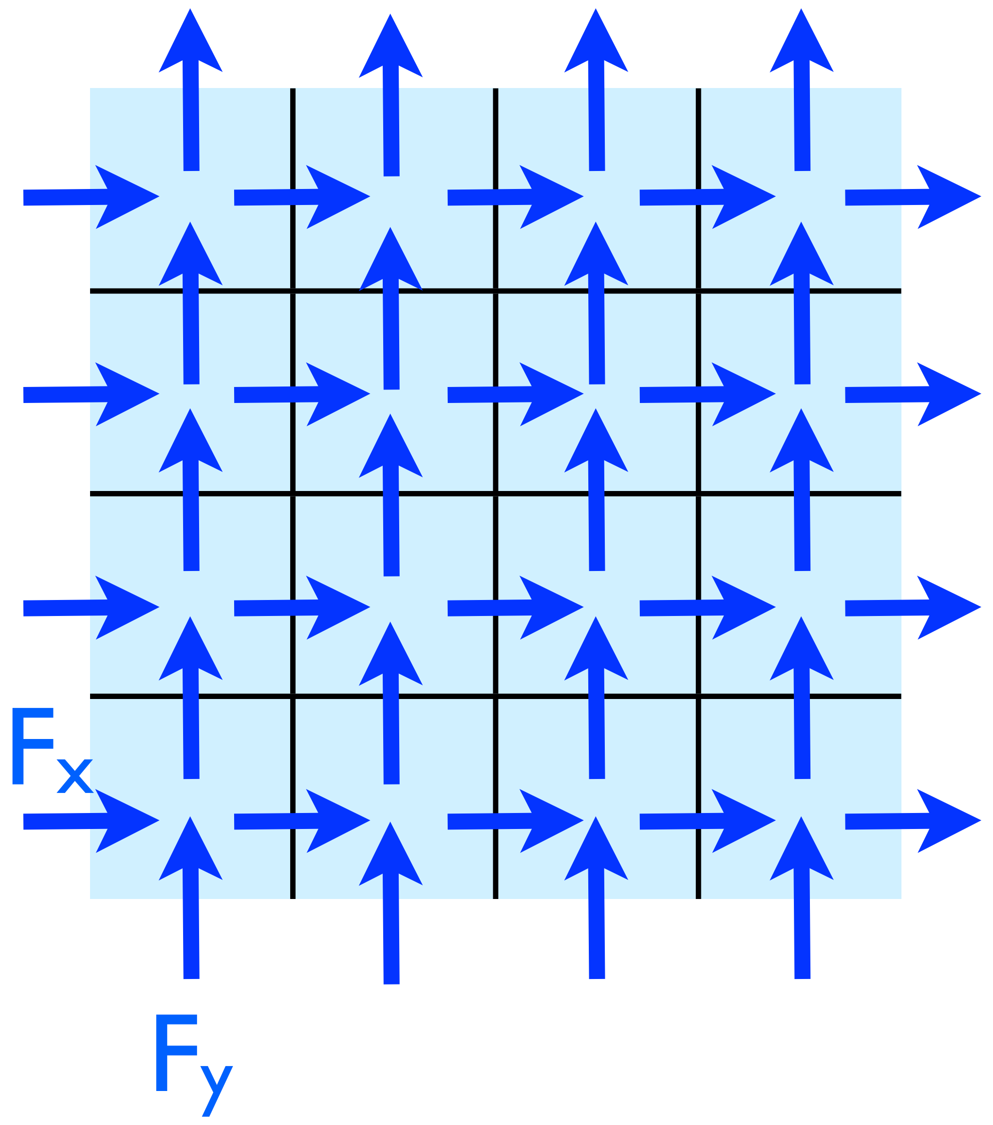


ng = 1

loop from ng, N - 2 ng

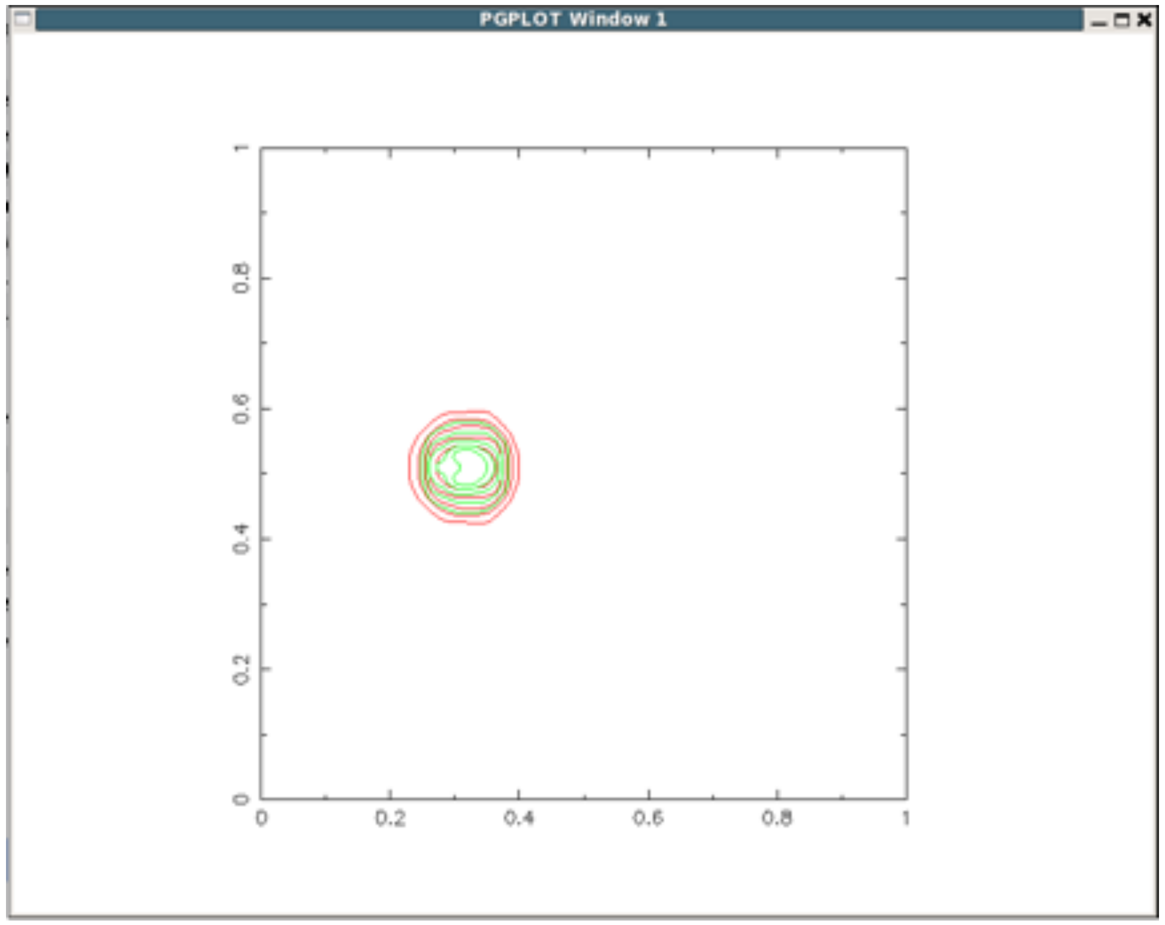
Finite Volume Method

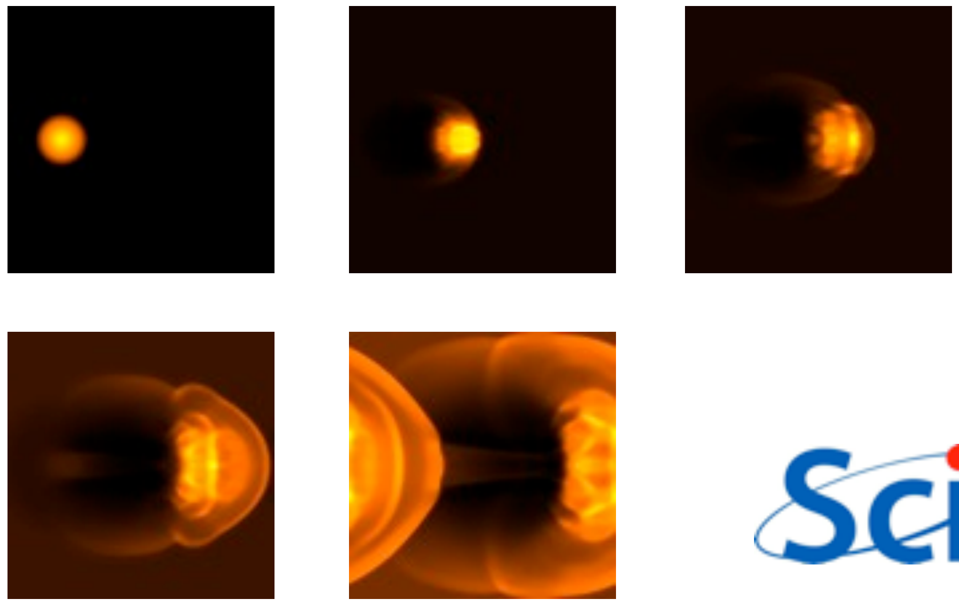
- Conservative; very well suited to high-speed flows with shocks
- At each timestep, calculate fluxes using interpolation/finite differences, and update cell quantities.
- Use conserved variables -- eg, momentum, not velocity.



Single-Processor hydro code

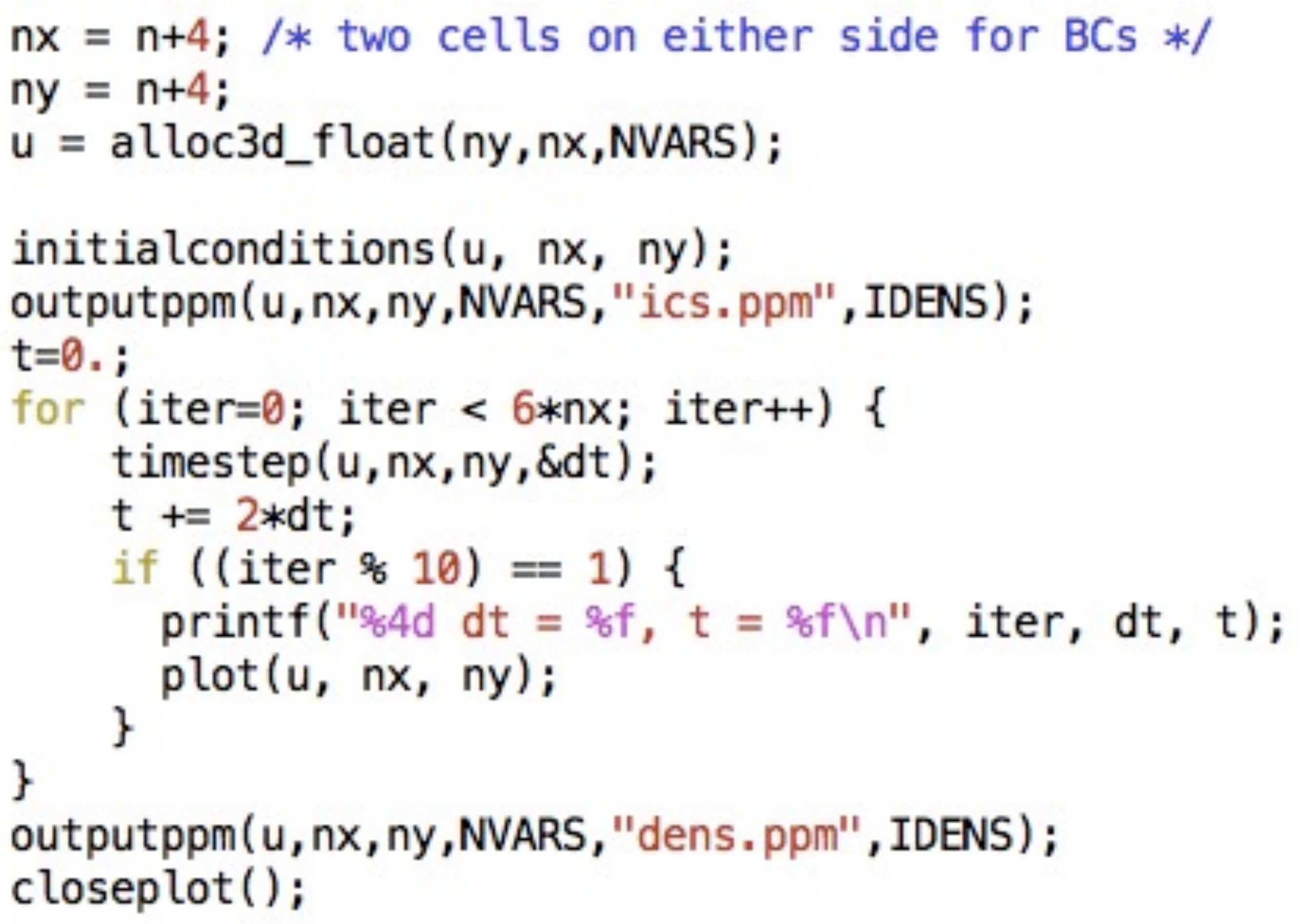
- `cd hydro{c,f};  make`
- `./hydro 100`
- Takes options:
  - number of points to write
- Outputs image (ppm) of initial conditions, final state (plots density)
- display ics.ppm
- display dens.ppm





Single-Processor hydro code

- Set initial conditions
- Loop, calling *timestep()* and maybe some output routines (*plot()* - contours)
- At beginning and end, save an image file with *outputppm()*
- All data stored in array *u*.

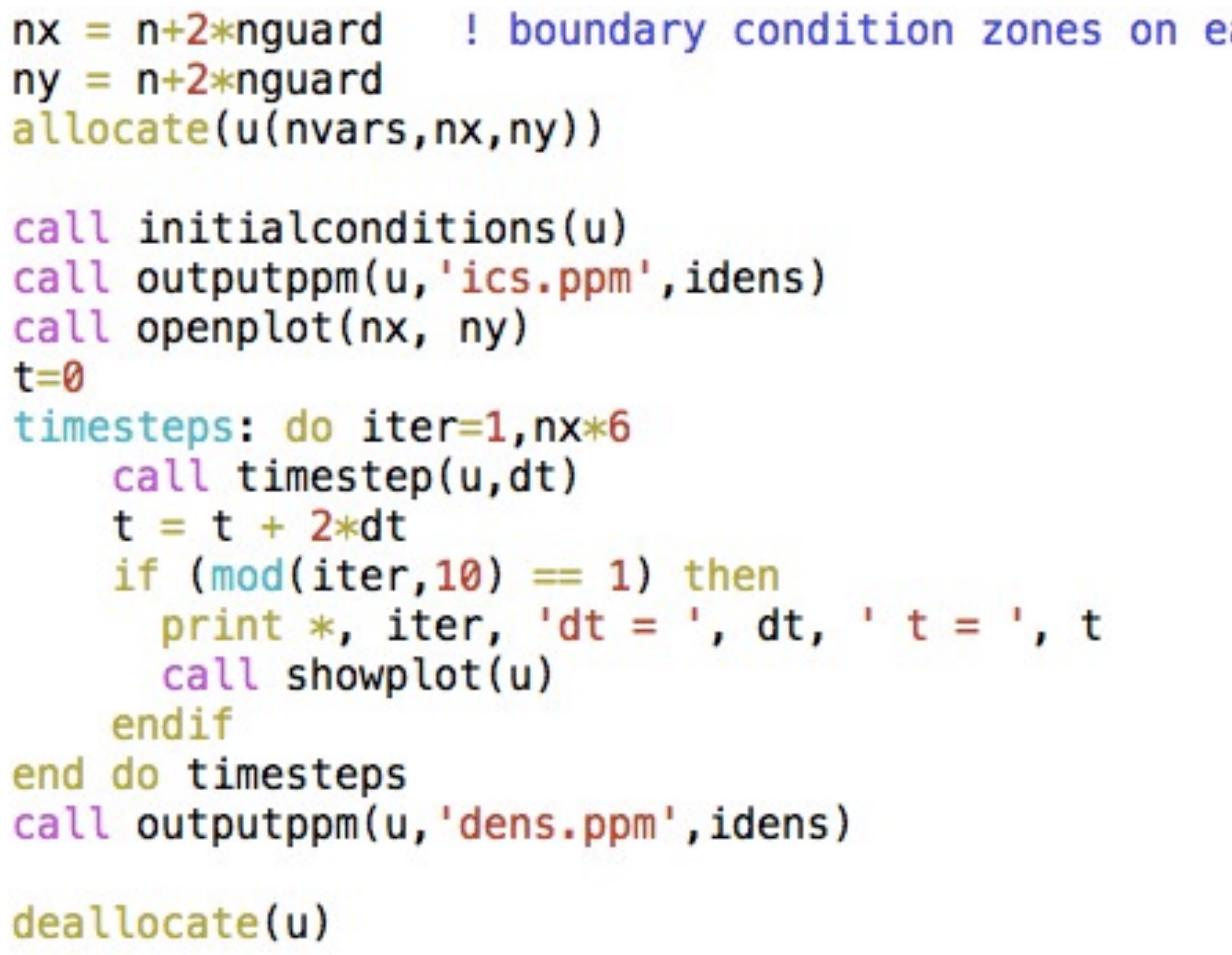


```c
nx = n+4; /* two cells on either side for BCs */
ny = n+4;
u = alloc3d_float(ny,nx,NVARS);

initialconditions(u, nx, ny);
outputppm(u,nx,ny,NVARS,"ics.ppm",IDENS);
t=0.;
for (iter=0; iter < 6*nx; iter++) {
    timestep(u,nx,ny,&dt);
    t += 2*dt;
    if ((iter % 10) == 1) {
      printf("%4d dt = %f, t = %f\n", iter, dt, t);
      plot(u, nx, ny);
    }
}
outputppm(u,nx,ny,NVARS,"dens.ppm",IDENS);
closeplot();
```

hydro.c SciNet

Single-Processor hydro code

- Set initial conditions
- Loop, calling *timestep()* and maybe some output routines (*plot()* - contours)
- At beginning and end, save an image file with *outputppm()*
- All data stored in array *u*.

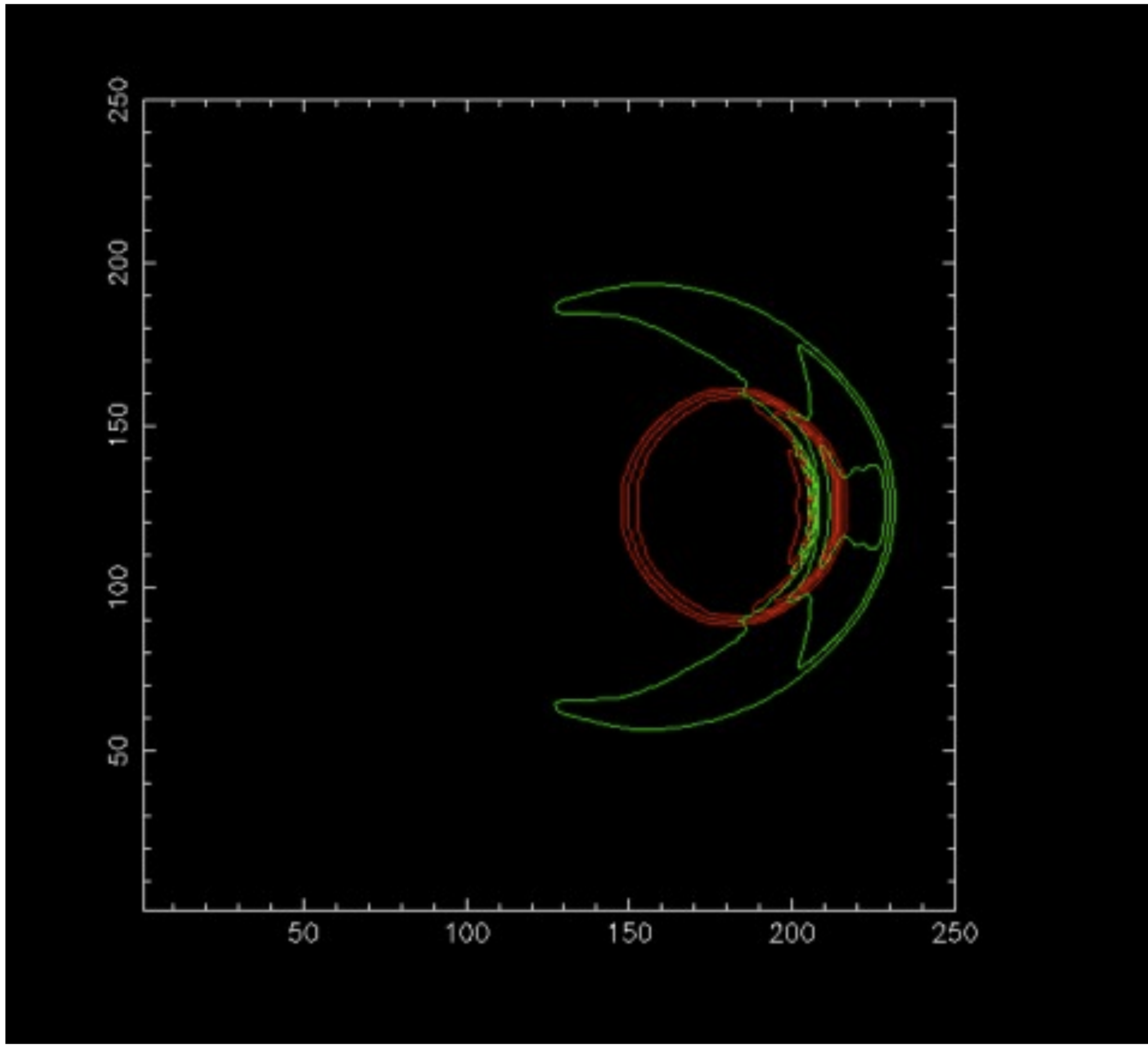


```
nx = n+2*nguard   ! boundary condition zones on e
ny = n+2*nguard
allocate(u(nvars,nx,ny))

call initialconditions(u)
call outputppm(u,'ics.ppm',idens)
call openplot(nx, ny)
t=0
timesteps: do iter=1,nx*6
    call timestep(u,dt)
    t = t + 2*dt
    if (mod(iter,10) == 1) then
      print *, iter, 'dt = ', dt, ' t = ', t
      call showplot(u)
    endif
end do timesteps
call outputppm(u,'dens.ppm',idens)

deallocate(u)
```

hydro.f90

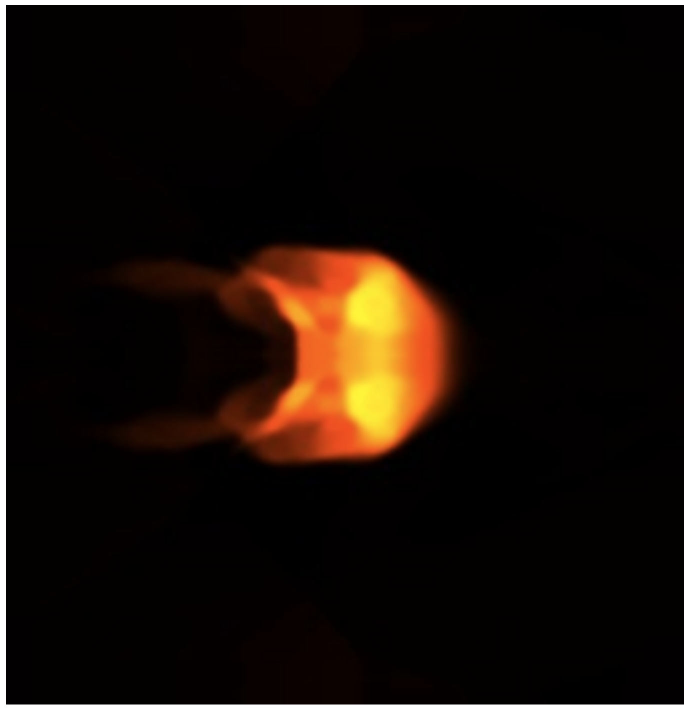
Plotting to screen

- plot.c, plot.f90
- Every 10 timesteps
- Find min, max of pressure, density
- Plot 5 contours of density (red) and pressure (green)
- pgplot library (old, but works).



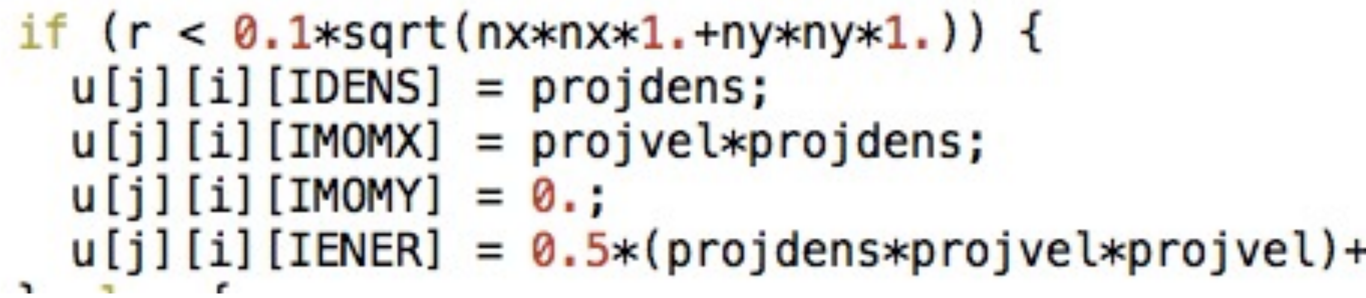
Plotting to file

- ppm.c, ppm.f90
- PPM format -- binary (w/ ascii header)
- Find min, max of density
- Calculate r,g,b values for scaled density (black = min, yellow = max)
- Write header, then data.



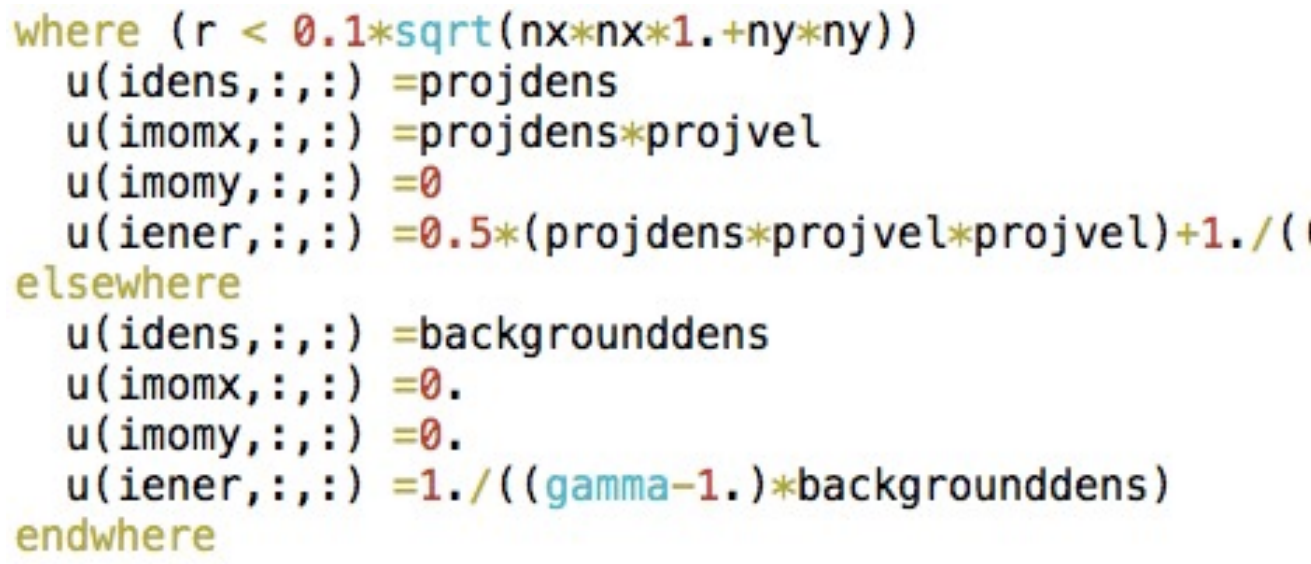
Data structure

- *u* : 3 dimensional array containing each variable in 2d space
- eg, u[j][i][IDENS]
- or u(idens, i, j)



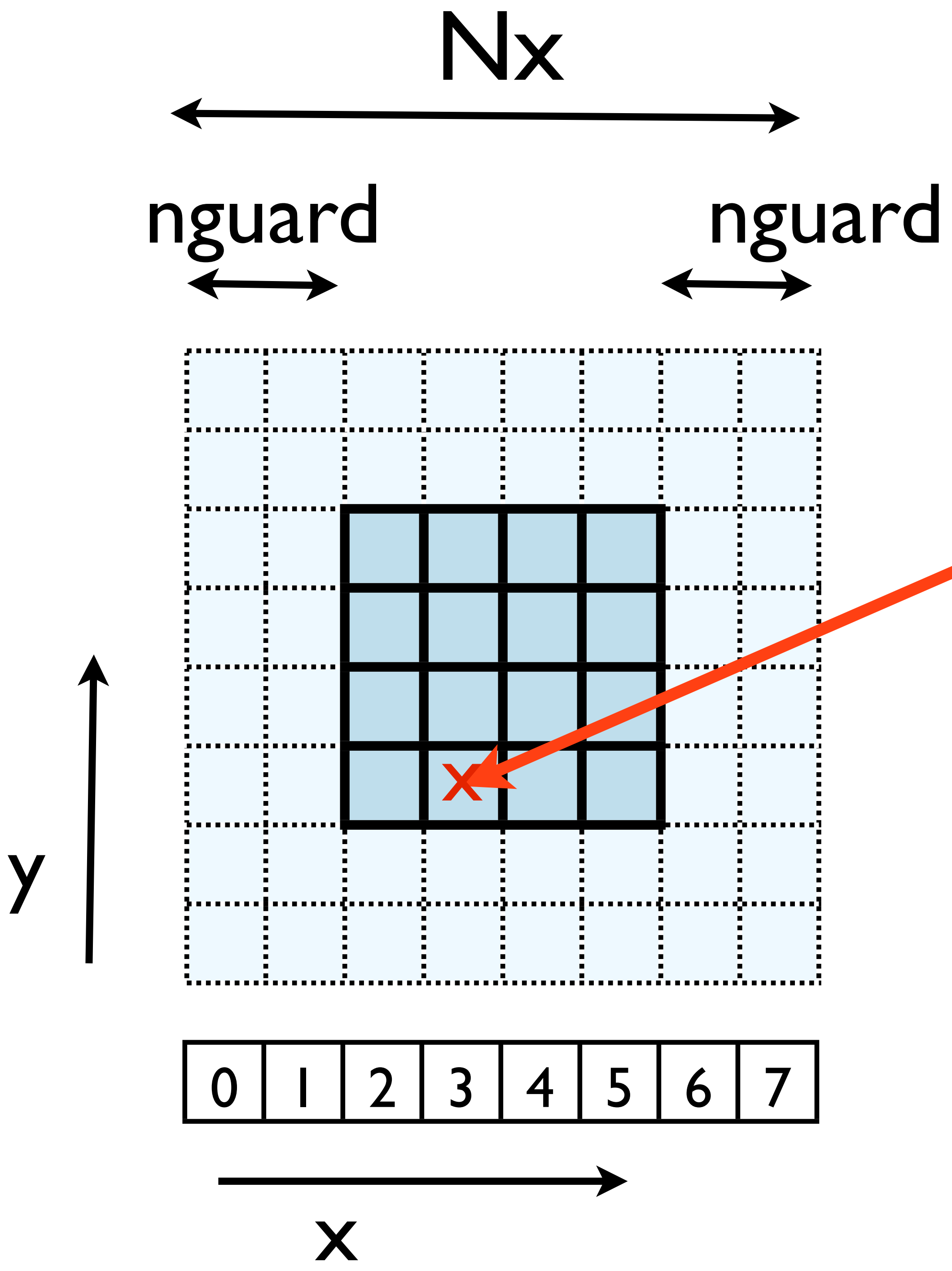
```
if (r < 0.1*sqrt(nx*nx*1.+ny*ny*1.)) {
  u[j][i][IDENS] = projdens;
  u[j][i][IMOMX] = projvel*projdens;
  u[j][i][IMOMY] = 0.;
  u[j][i][IENER] = 0.5*(projdens*projvel*projvel)+
```

solver.c (initialconditions)



```
where (r < 0.1*sqrt(nx*nx*1.+ny*ny))
  u(idens,:,:) =projdens
  u(imomx,:,:) =projdens*projvel
  u(imomy,:,:) =0
  u(iener,:,:) =0.5*(projdens*projvel*projvel)+1./(
elsewhere
  u(idens,:,:) =backgrounddens
  u(imomx,:,:) =0.
  u(imomy,:,:) =0.
  u(iener,:,:) =1./((gamma-1.)*backgrounddens)
endwhere
```

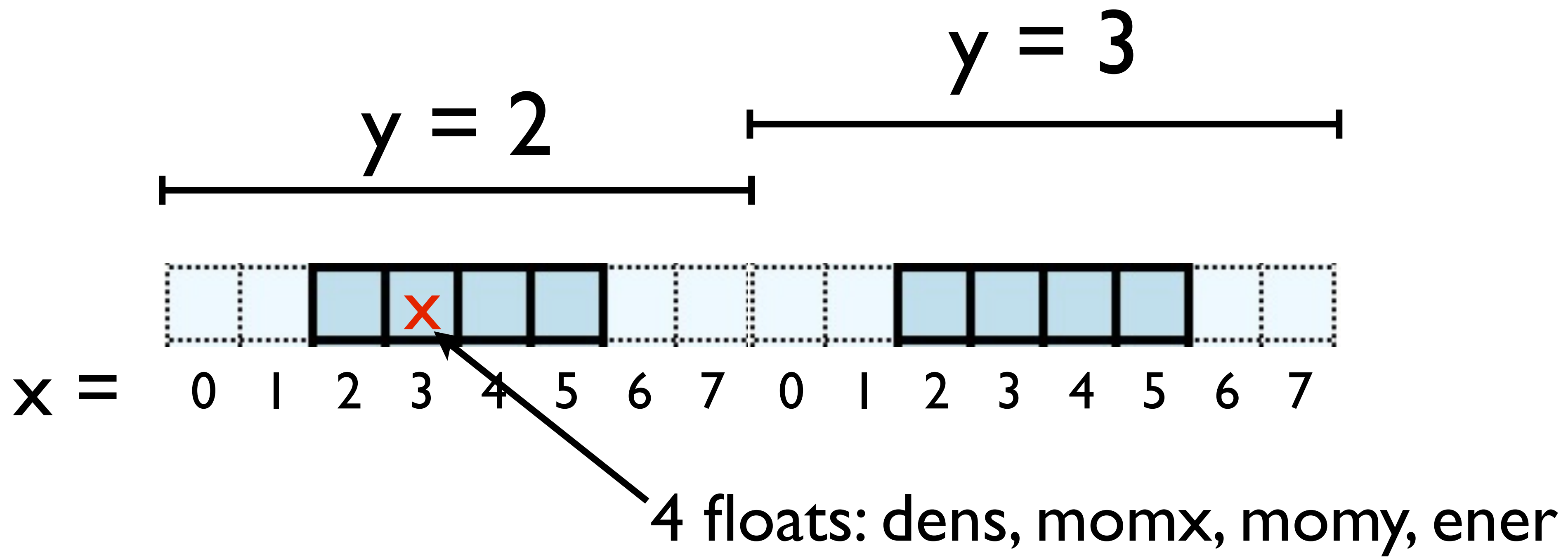
solver.f90 (initialconditions)



```
u[2][3][DENSVAR];
u[2][3][MOMXVAR];
u[2][3][MOMYVAR];
u[2][3][ENERVAR];
```

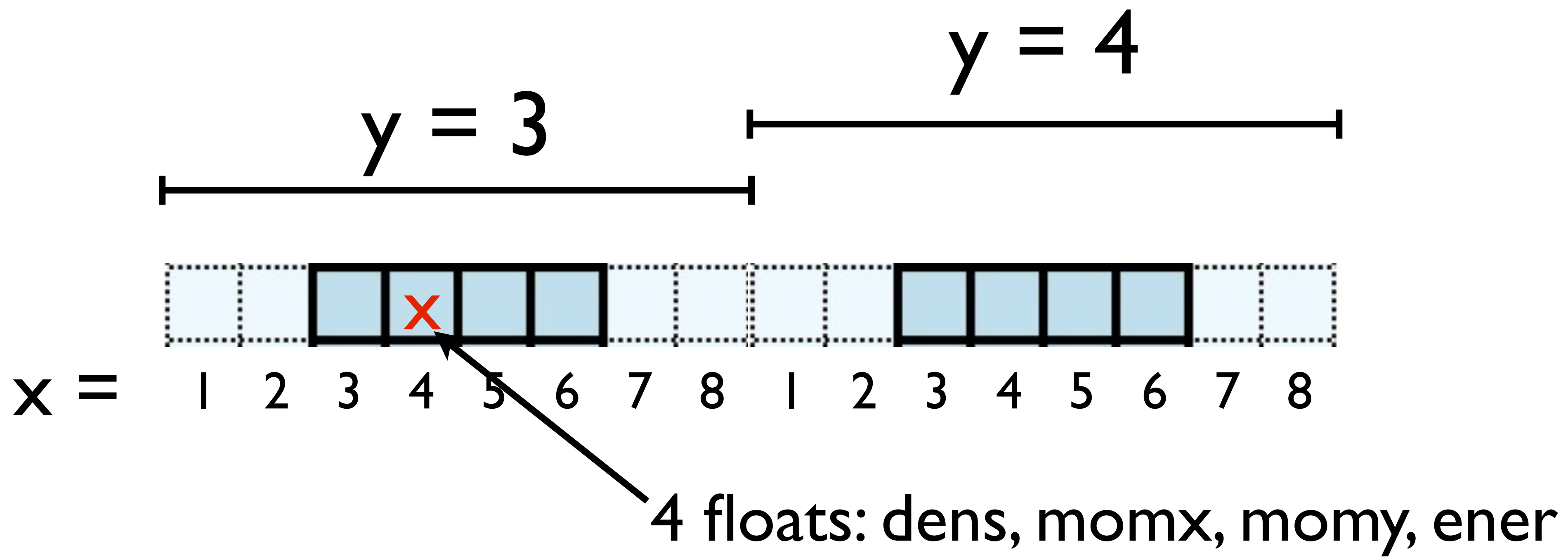
```
u(idens,4,3)
u(imomx,4,3)
u(imomy,4,3)
u(iener,4,3)
```

Laid out in memory (C)


y = 2
y = 3
x
x = 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
4 floats: dens, momx, momy, ener

Same way as in an image file
(one horizontal row at a time)

Laid out in memory (FORTRAN)

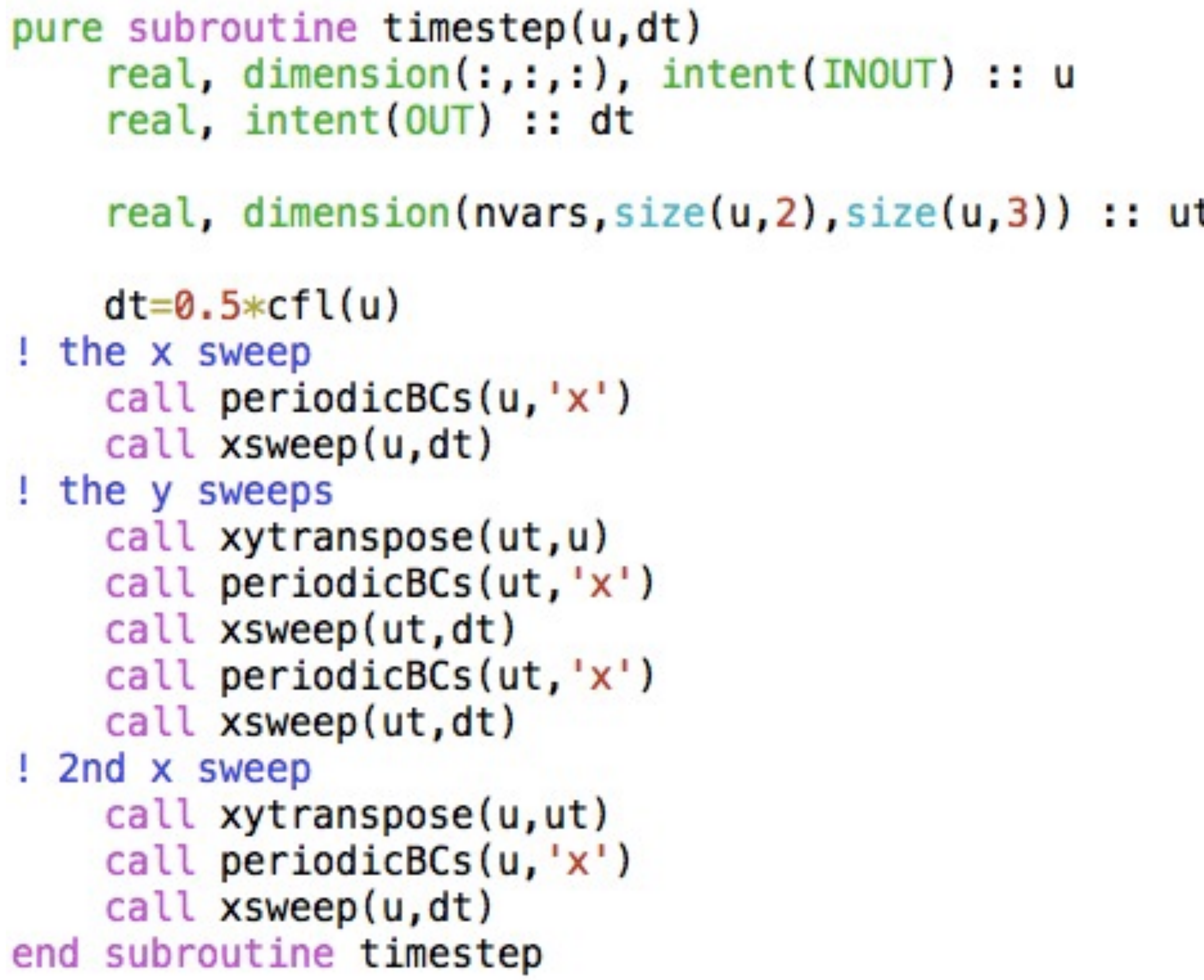


y = 4
y = 3
x
x = 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8
4 floats: dens, momx, momy, ener

Same way as in an image file
(one horizontal row at a time)

Timestep routine

- Apply boundary conditions
- X sweep, Y sweep
- Transpose entire domain , so Y sweep is just an X sweep
- (unusual approach! But has advantages. Like matrix multiply.)
- Note - dt calculated each step (minimum across domain.)

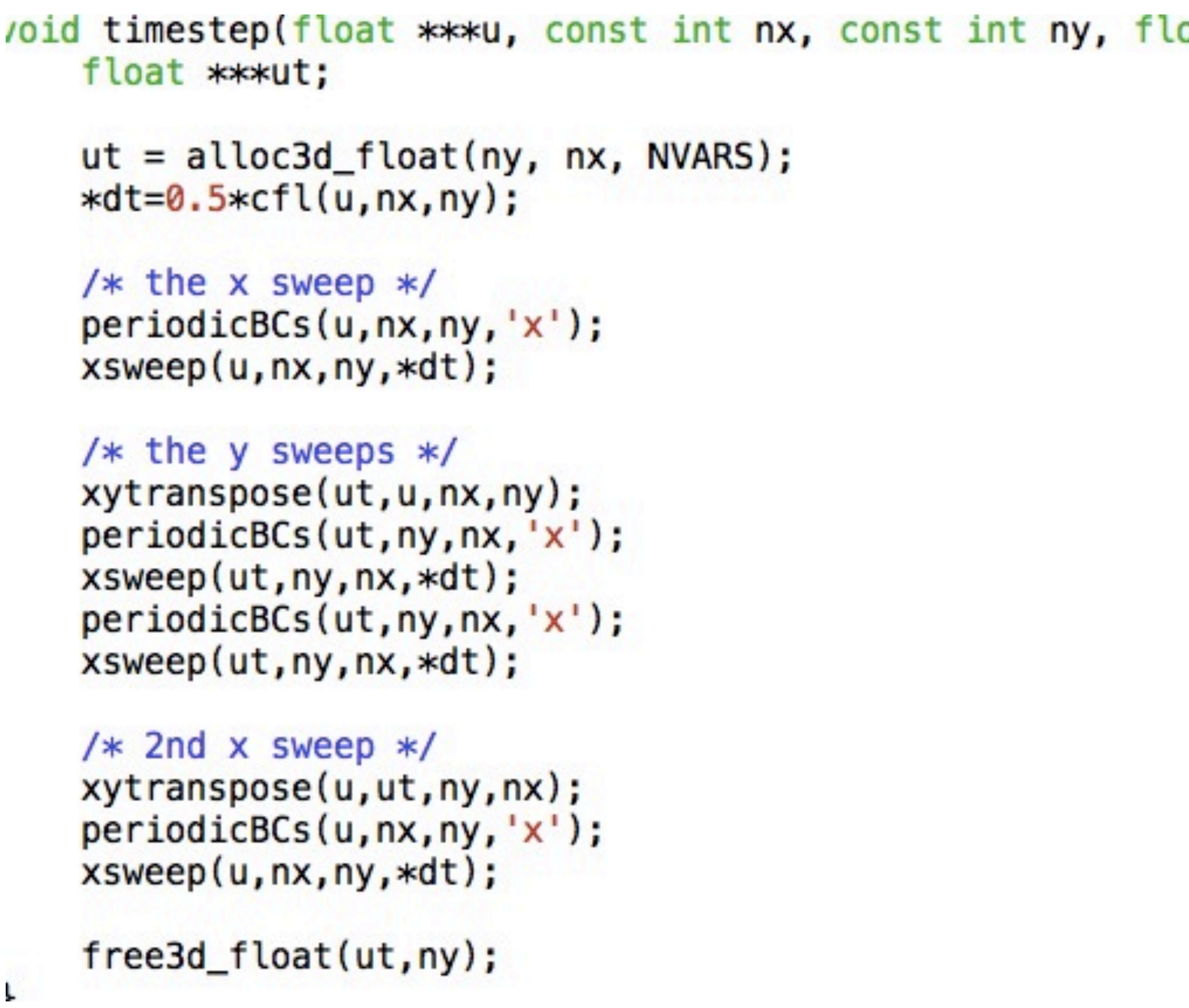


```
pure subroutine timestep(u,dt)
    real, dimension(:,:,:), intent(INOUT) :: u
    real, intent(OUT) :: dt

    real, dimension(nvars,size(u,2),size(u,3)) :: ut

    dt=0.5*cfl(u)
! the x sweep
    call periodicBCs(u,'x')
    call xsweep(u,dt)
! the y sweeps
    call xytranspose(ut,u)
    call periodicBCs(ut,'x')
    call xsweep(ut,dt)
    call periodicBCs(ut,'x')
    call xsweep(ut,dt)
! 2nd x sweep
    call xytranspose(u,ut)
    call periodicBCs(u,'x')
    call xsweep(u,dt)
end subroutine timestep
```

timestep
solver.f90

Timestep routine

- Apply boundary conditions
- X sweep, Y sweep
- Transpose entire domain , so Y sweep is just an X sweep
- (unusual approach! But has advantages. Like matrix multiply.)
- Note - dt calculated each step (minimum across domain.)

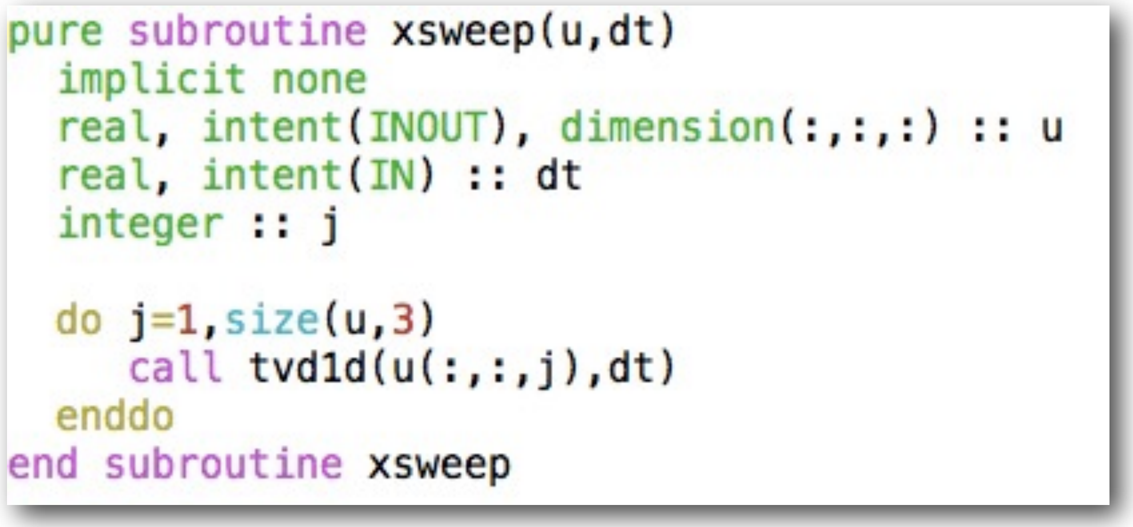


```
/oid timestep(float ***u, const int nx, const int ny, flo
    float ***ut;

    ut = alloc3d_float(ny, nx, NVARS);
    *dt=0.5*cfl(u,nx,ny);

    /* the x sweep */
    periodicBCs(u,nx,ny,'x');
    xsweep(u,nx,ny,*dt);

    /* the y sweeps */
    xytranspose(ut,u,nx,ny);
    periodicBCs(ut,ny,nx,'x');
    xsweep(ut,ny,nx,*dt);
    periodicBCs(ut,ny,nx,'x');
    xsweep(ut,ny,nx,*dt);

    /* 2nd x sweep */
    xytranspose(u,ut,ny,nx);
    periodicBCs(u,nx,ny,'x');
    xsweep(u,nx,ny,*dt);

    free3d_float(ut,ny);
}
```

timestep
solver.c

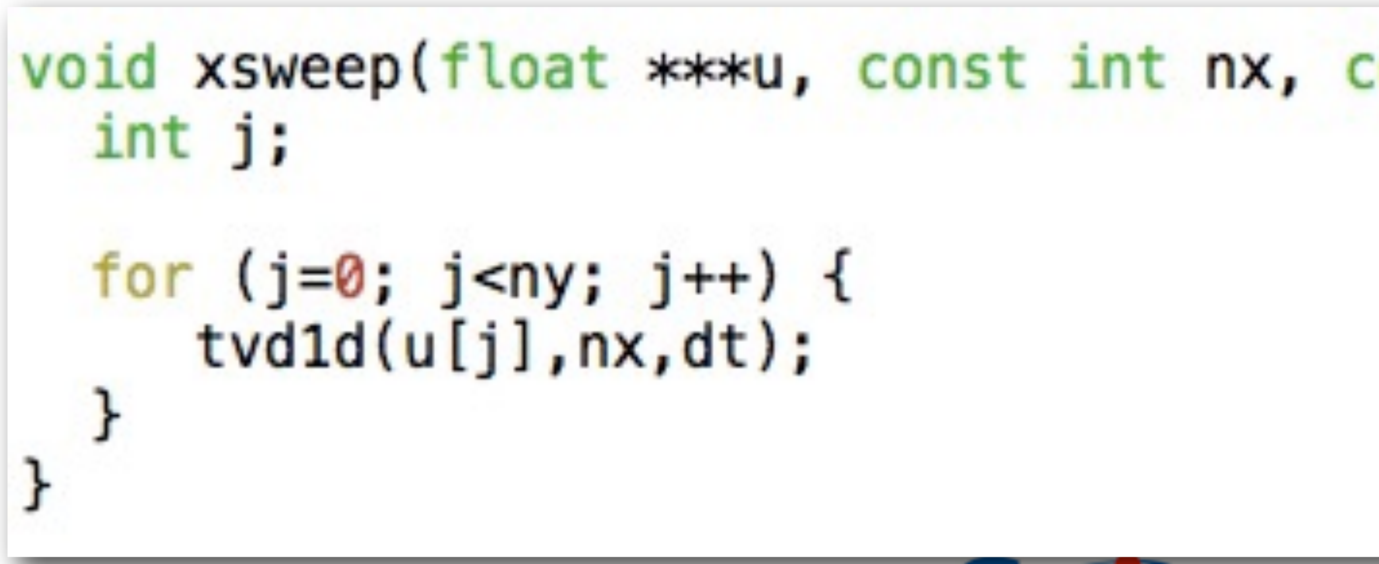
Xsweep routine

- Go through each x "pencil" of cells
- Do 1d hydrodynamics routine on that pencil.



```
pure subroutine xsweep(u,dt)
  implicit none
  real, intent(INOUT), dimension(:,:,:) :: u
  real, intent(IN) :: dt
  integer :: j

  do j=1,size(u,3)
     call tvd1d(u(:,:,j),dt)
  enddo
end subroutine xsweep
```

xsweep
solver.f90



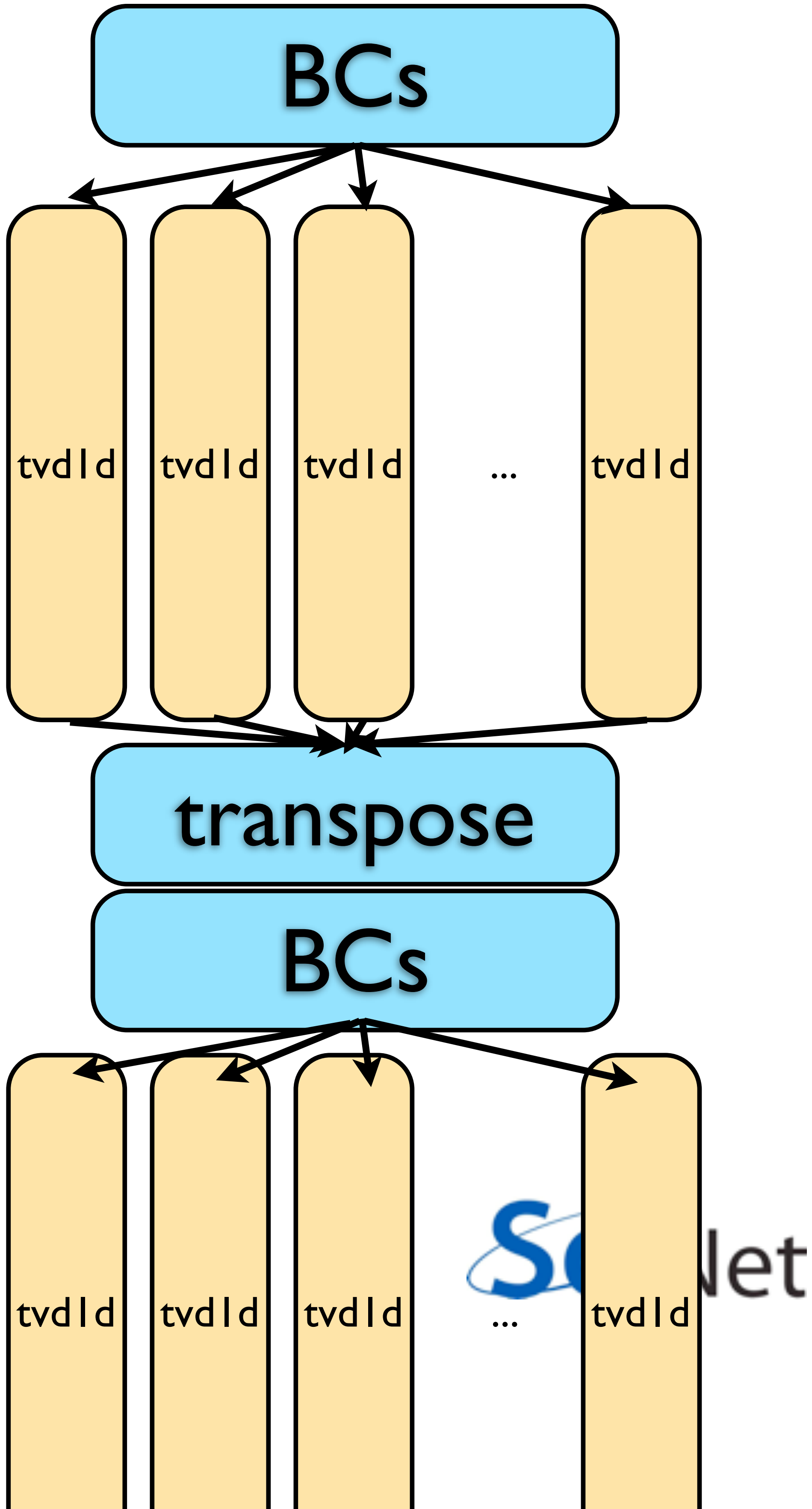
```
void xsweep(float ***u, const int nx, c
  int j;

  for (j=0; j<ny; j++) {
     tvd1d(u[j],nx,dt);
  }
}
```

xsweep
solver.c

What do data dependancies look like for this?

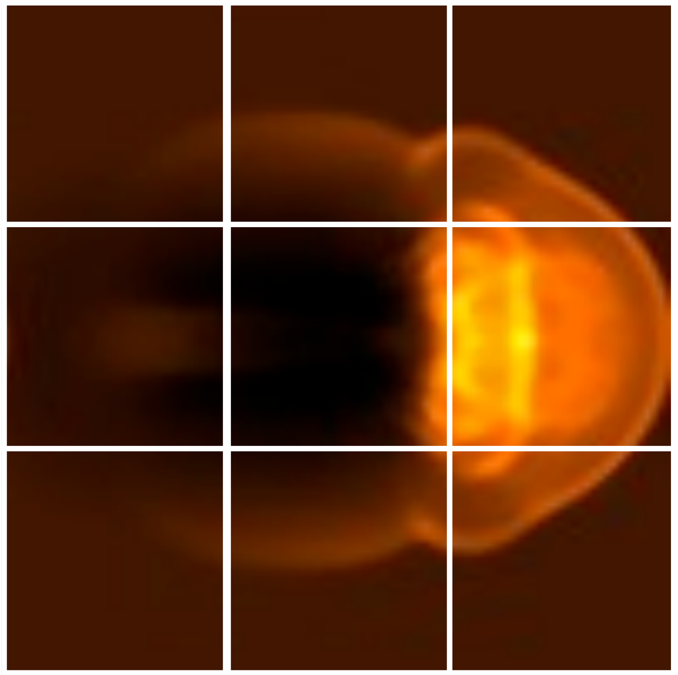
Data dependencies

- Previous timestep must be completed before next one started.
- Within each timestep,
- Each tvd1d "pencil" can be done independently
- All must be done before transpose, BCs



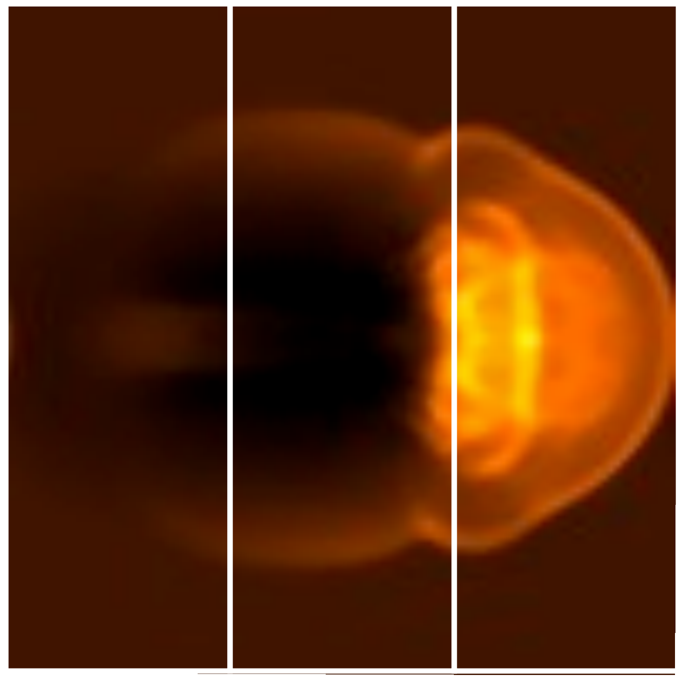
MPIing the code

- Domain decomposition



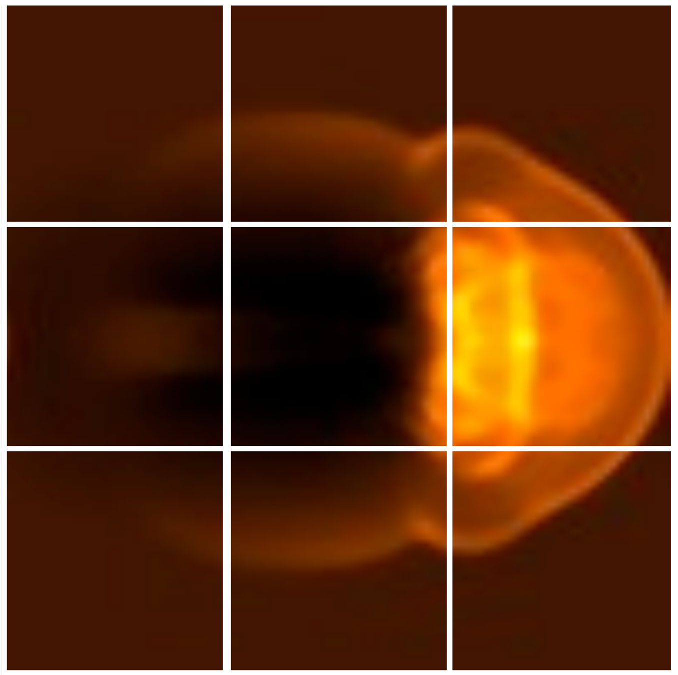
MPIing the code

- Domain decomposition
- For simplicity, for now we'll just implement decomposition in one direction, but we will design for full 2d decomposition



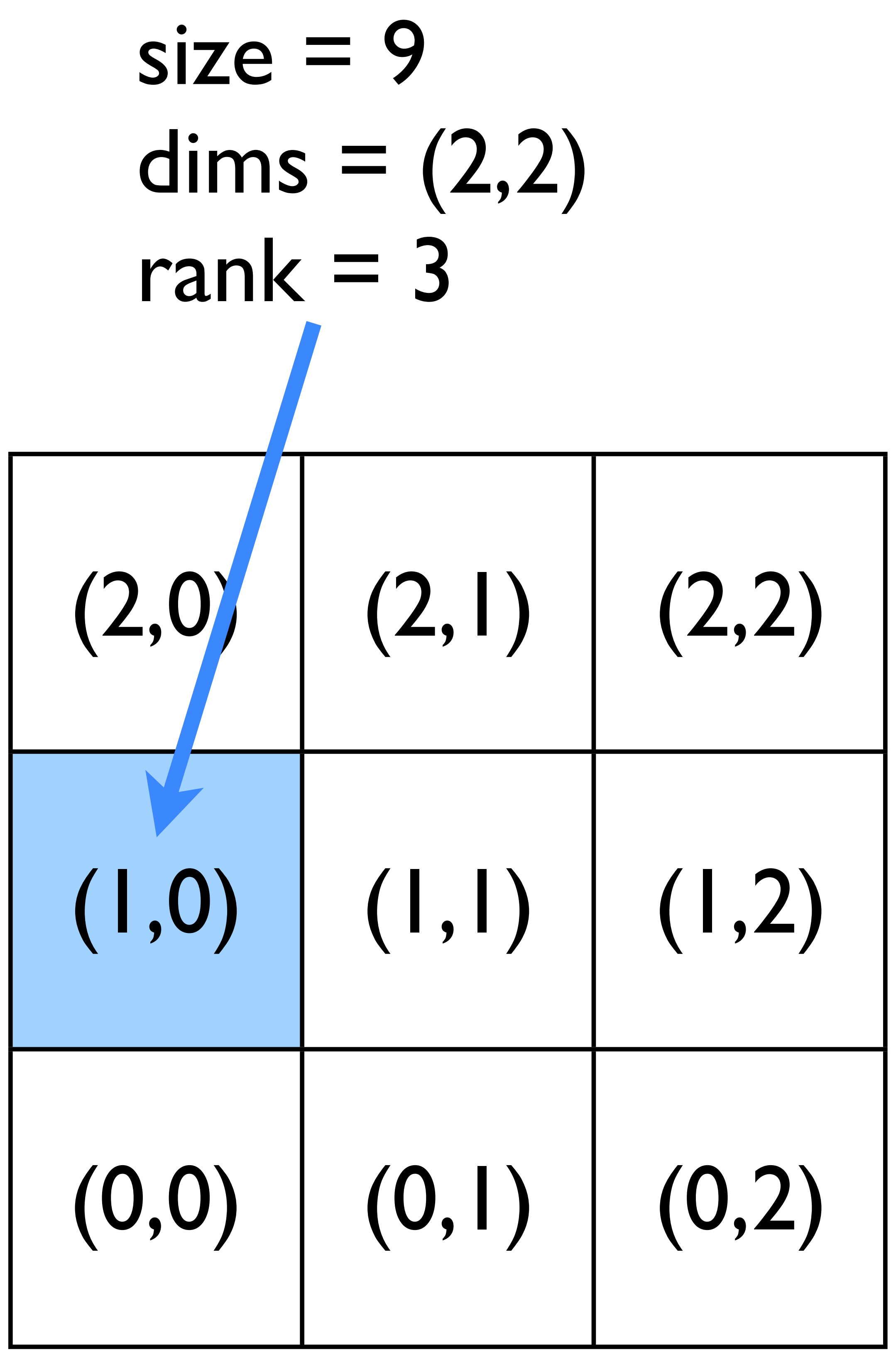
MPIing the code

- Domain decomposition
- We can do as with diffusion and figure out out neighbours by hand, but MPI has a better way...



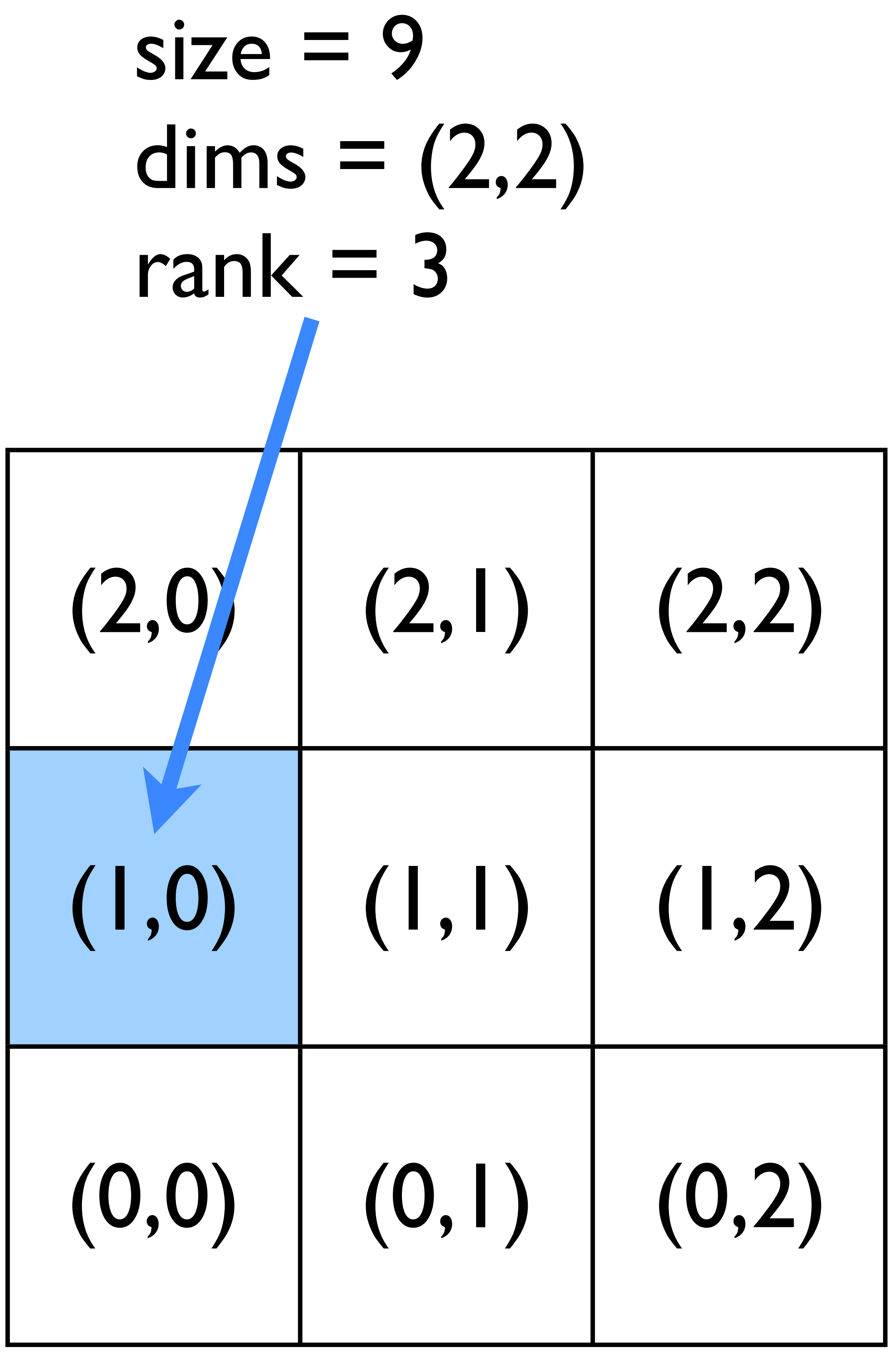
Create new communicator with new topology

- MPI_Cart_create (MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)

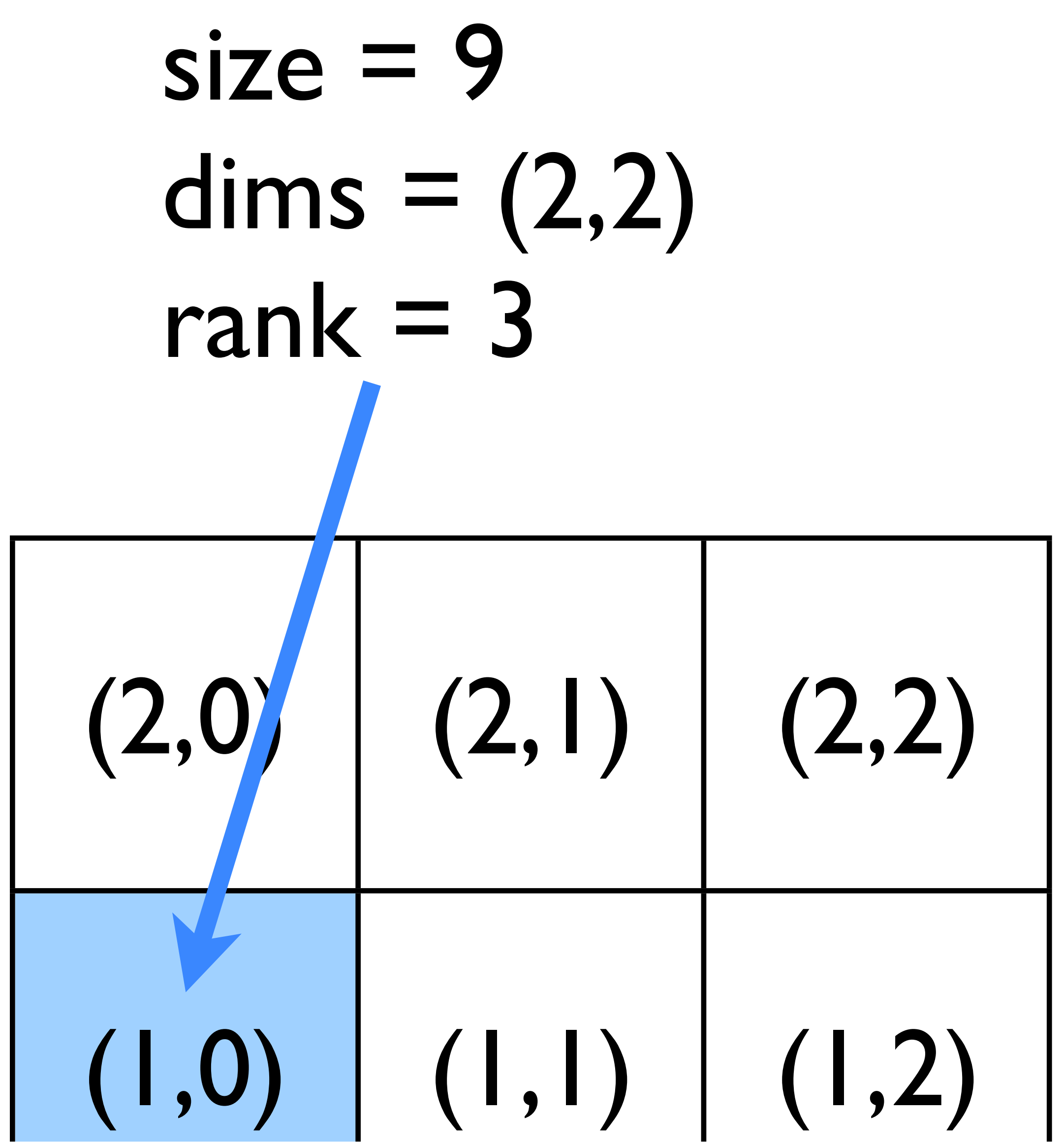


Create new communicator with new topology

- MPI_Cart_create (integer comm_old, integer ndims, integer [dims], logical [periods], integer reorder, integer comm_cart, integer ierr)



Create new communicator with new topology

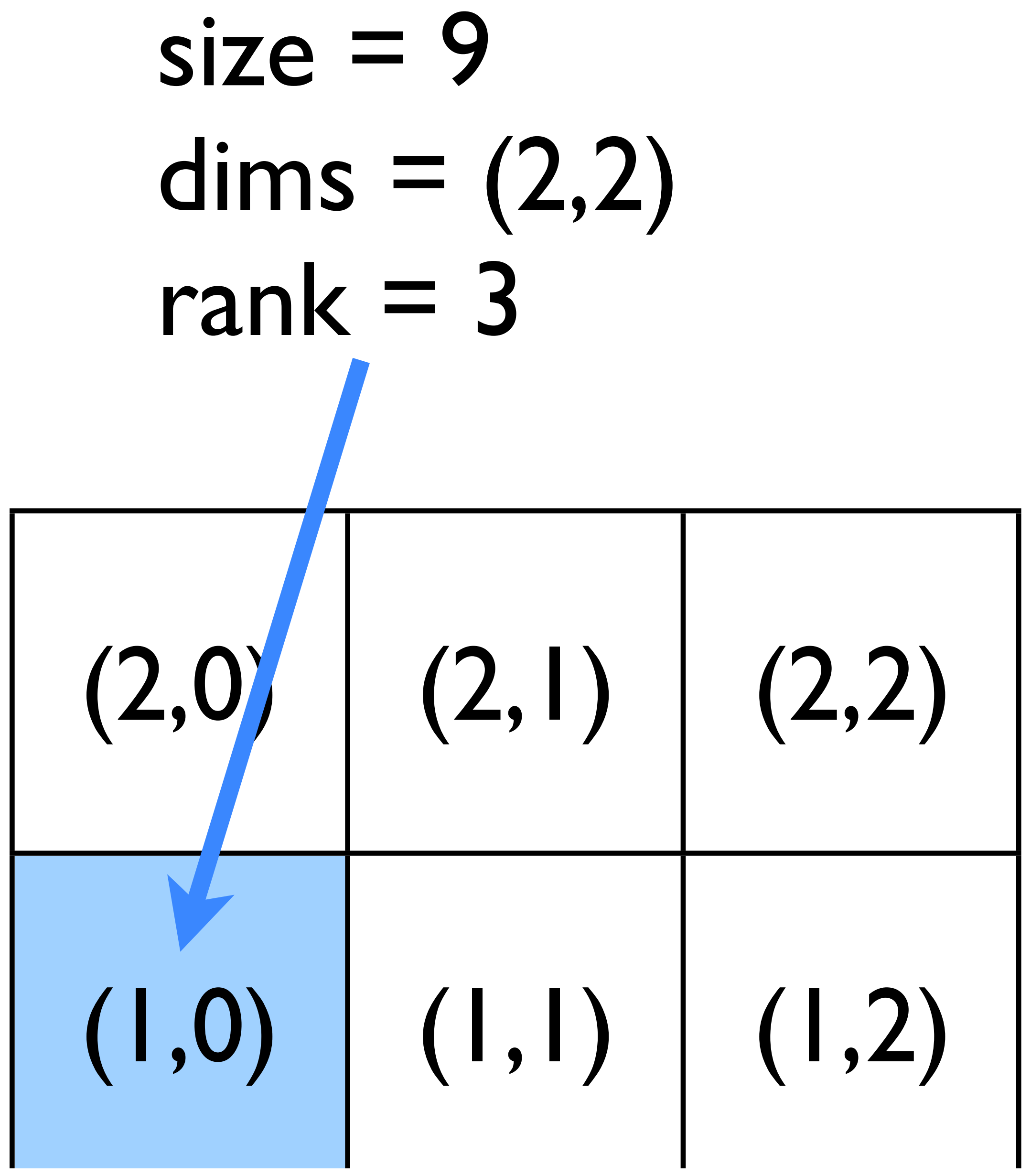


C

```
ierr = MPI_Cart_shift(MPI_COMM new_comm, int dim,
       int shift, int *left, int *right)
ierr = MPI_Cart_coords(MPI_COMM new_comm, int rank,
       int ndims, int *gridcoords)
```

SciNet

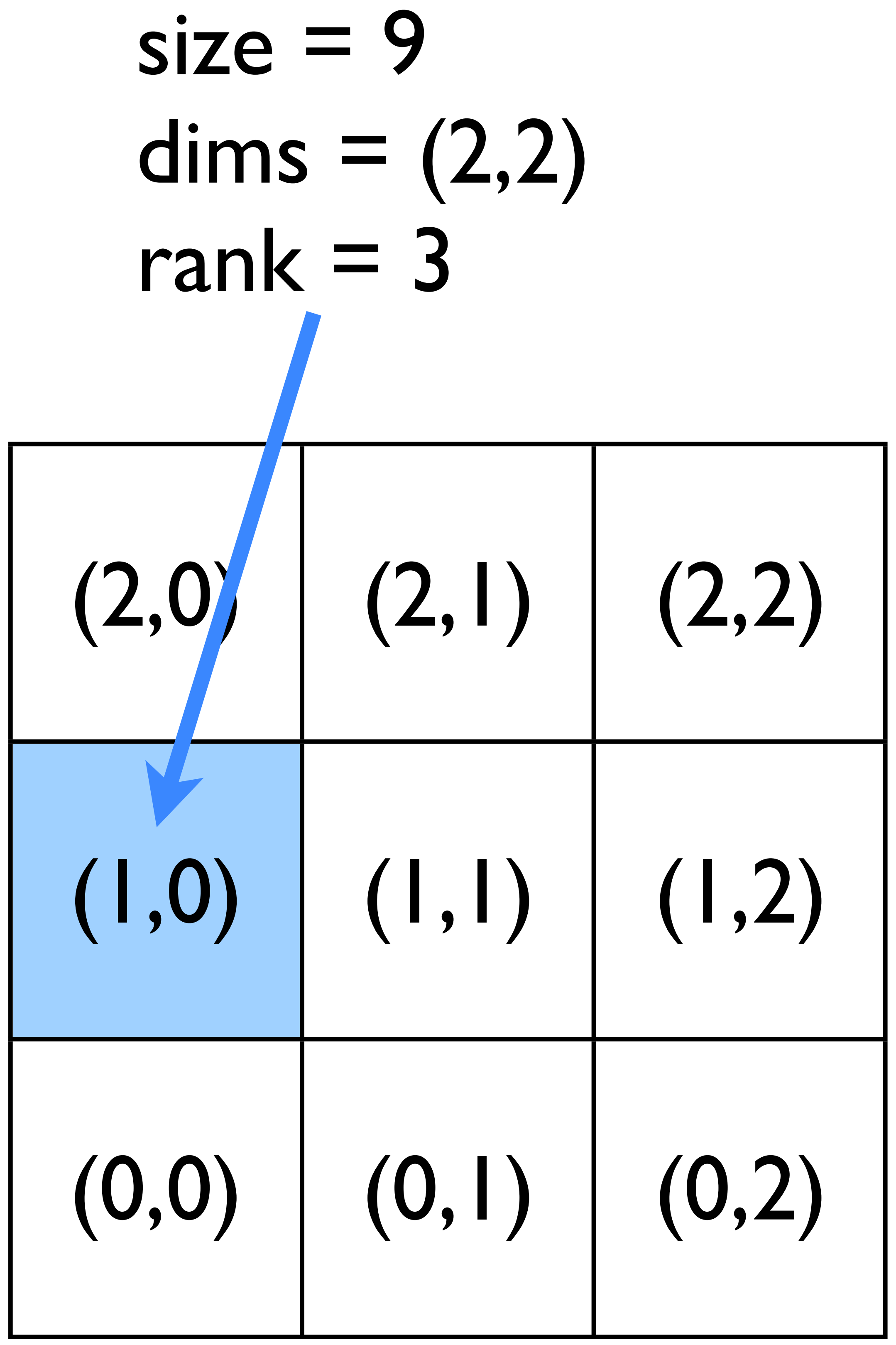
Create new communicator with new topology



```
FORTRAN
call MPI_Cart_shift(integer new_comm, dim, shift,
       left, right, ierr)
call MPI_Cart_coords(integer new_comm, rank,
       ndims, [gridcoords], ierr)
```

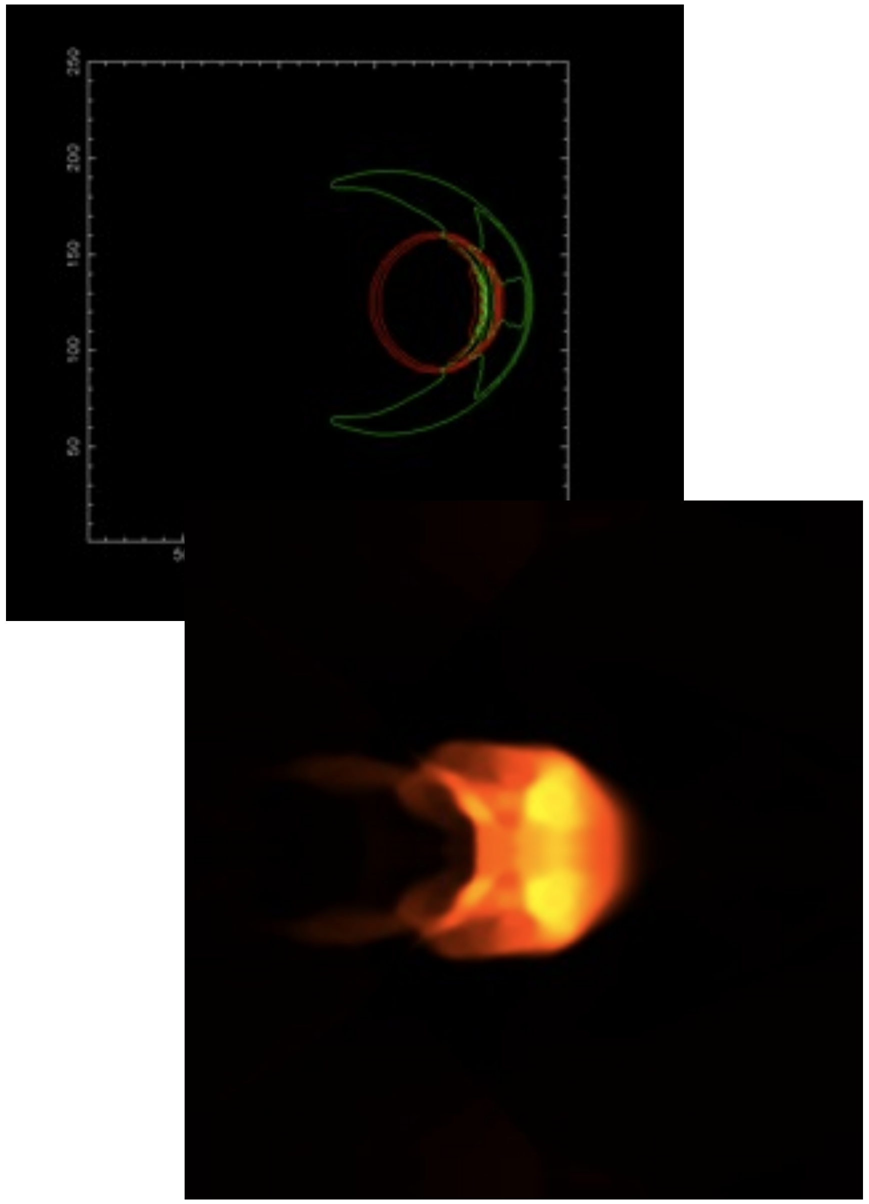
Let's try starting to do this together

- In a new directory:
- add mpi_init, _finalize, comm_size.
- mpi_cart_create
- rank on *new* communicator.
- neighbours
- Only do part of domain



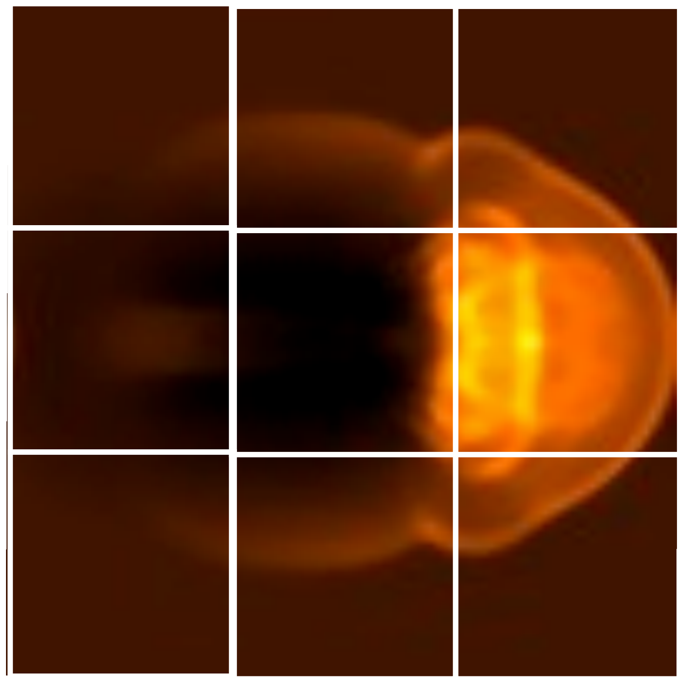
Next

- File IO - have each process write its own file so don't overwrite
- Coordinate min, max across processes for contours, images.
- Coordinate min in cfl routine.



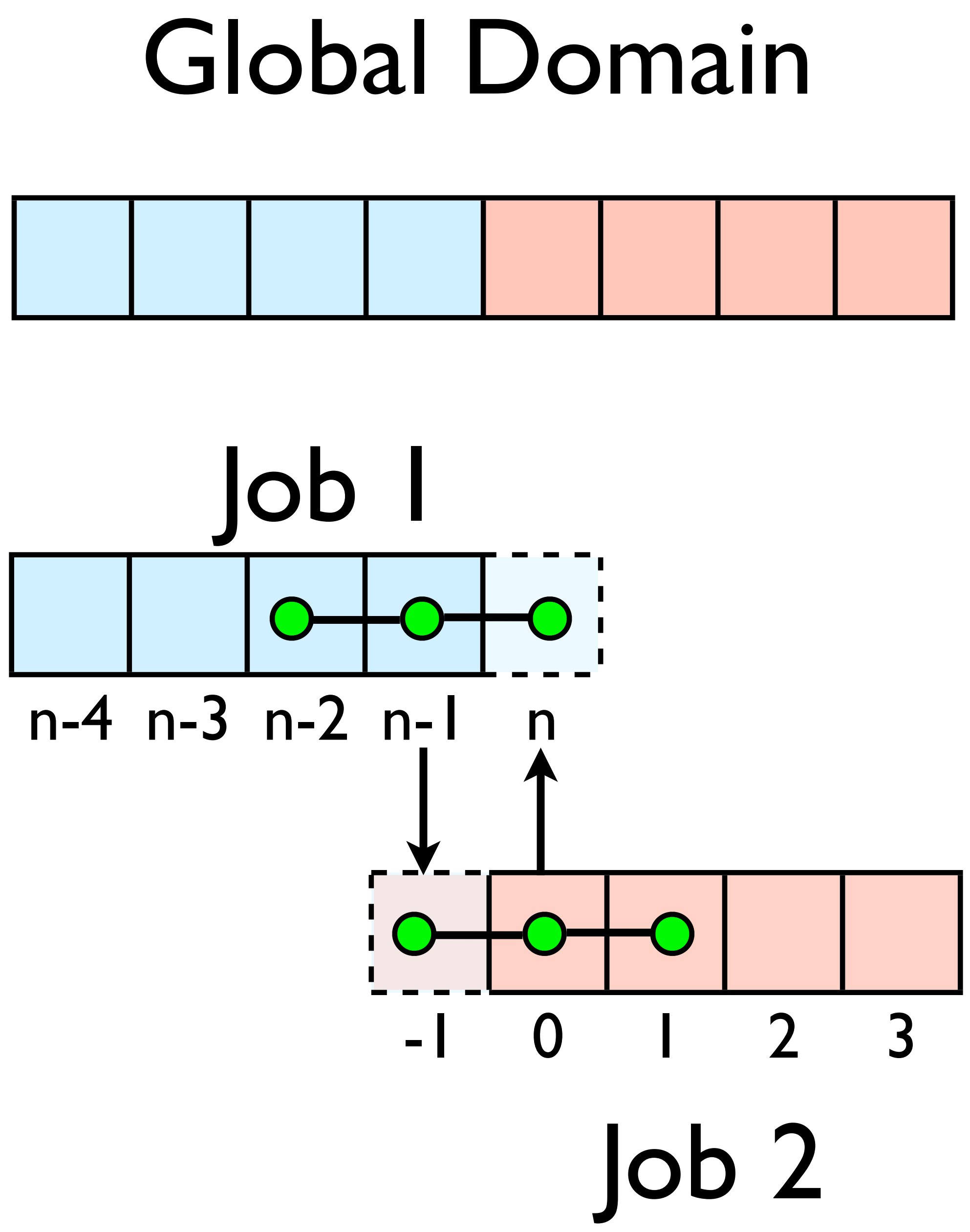
MPIing the code

- Domain decomposition
- Lots of data - ensures locality
- How are we going to handle getting non-local information across processors?



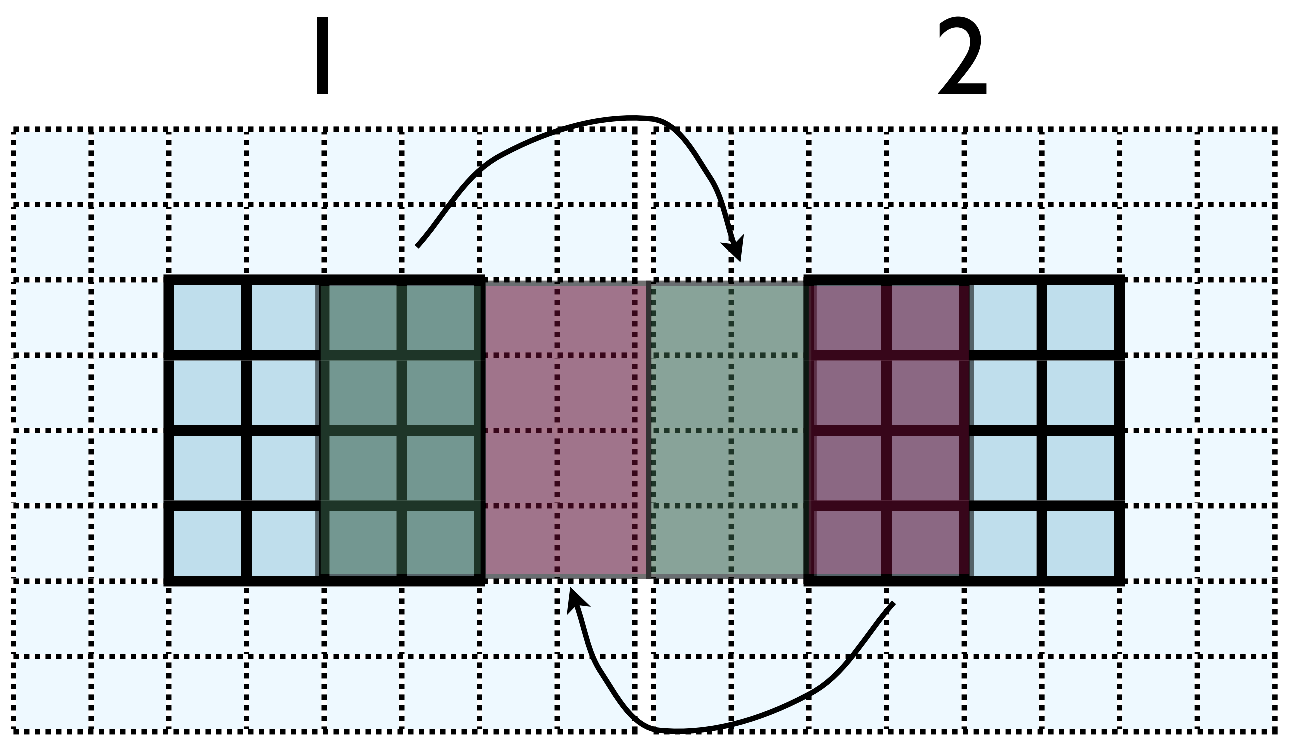
Guardcells

- 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
- Hydro code: need guardcells 2 deep



Guard cell fill

- When we're doing boundary conditions.
- Swap guardcells with neighbour.



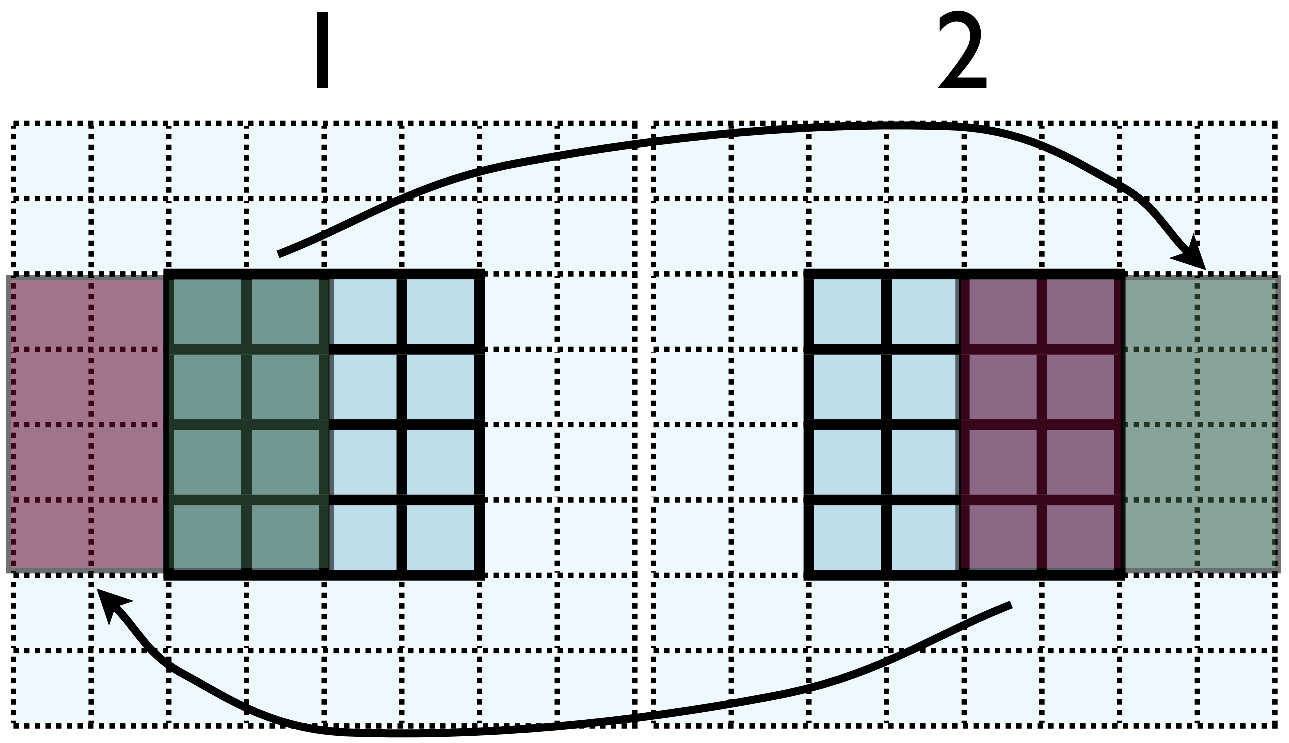
1: u(:, nx:nx+ng, ng:ny-ng)
→ 2: u(:,1:ng, ng:ny-ng)

2: u(:, ng+1:2*ng, ng:ny-ng)
→ 1: u(:, nx+ng+1:nx+2*ng, ng:ny-ng)

(ny-2*ng)*ng values to swap

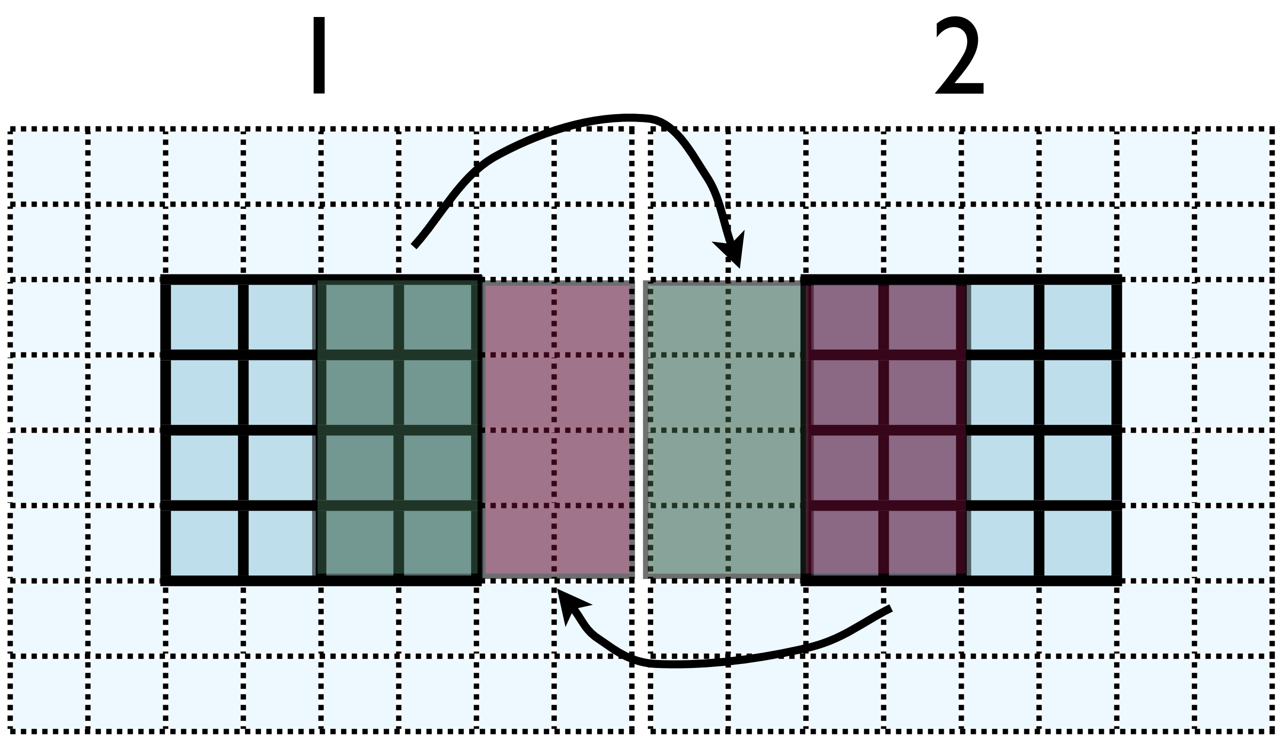
Cute way for Periodic BCs

- Actually make the decomposed mesh periodic;
- Make the far ends of the mesh neighbors
- Don't know the difference between that and any other neighboring grid
- Cart_create sets this up for us automatically upon request.



Implementing in MPI

- No different in principle than diffusion
- Just more values
- And more variables: dens, ener, imomx....
- Simplest way: copy all the variables into an NVARS*(ny-2*ng)*ng sized



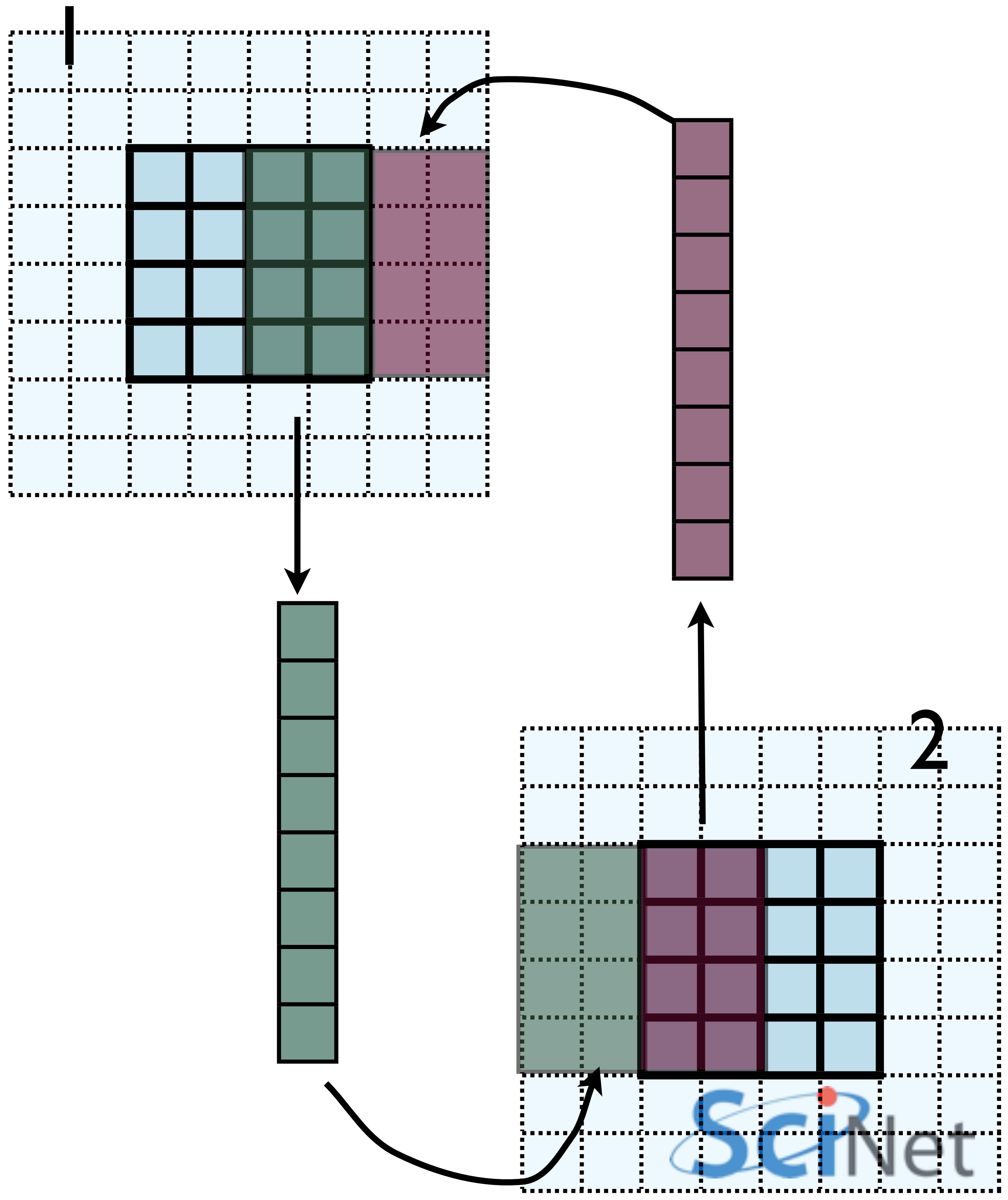
1: u(:, nx:nx+ng, ng:ny-ng)
→ 2: u(:,1:ng, ng:ny-ng)

2: u(:, ng+1:2*ng, ng:ny-ng)
→ 1: u(:, nx+ng+1:nx+2*ng, ng:ny-ng)

nvars*(ny-2*ng)*ng values to swap

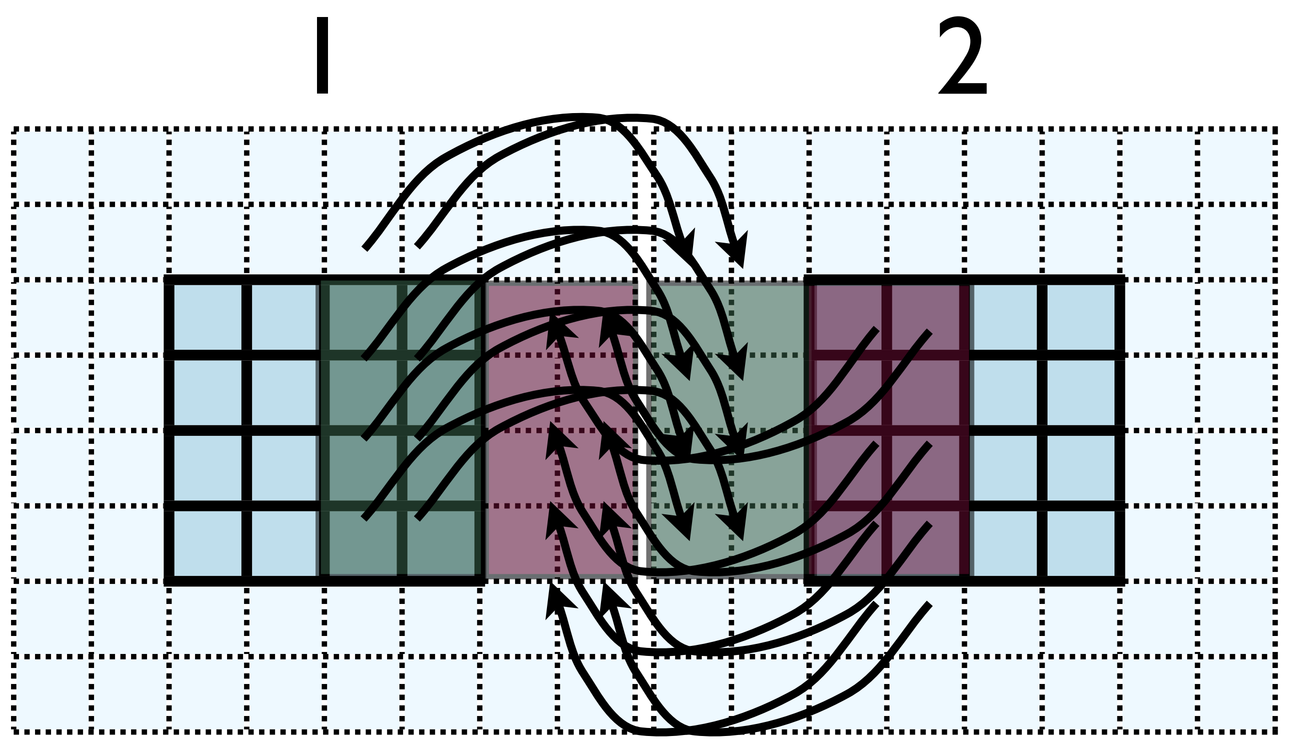
Implementing in MPI

- No different in principle than diffusion
- Just more values
- And more variables: dens, ener, temp....
- Simplest way: copy all the variables into an NVARS*(ny-2*ng)*ng sized



Implementing in MPI

- Even simpler way:
- Loop over values, sending each one, rather than copying into buffer.
- NVARS*nguard*(ny-2*nguard) latency hit.
- Would completely dominate communications cost.

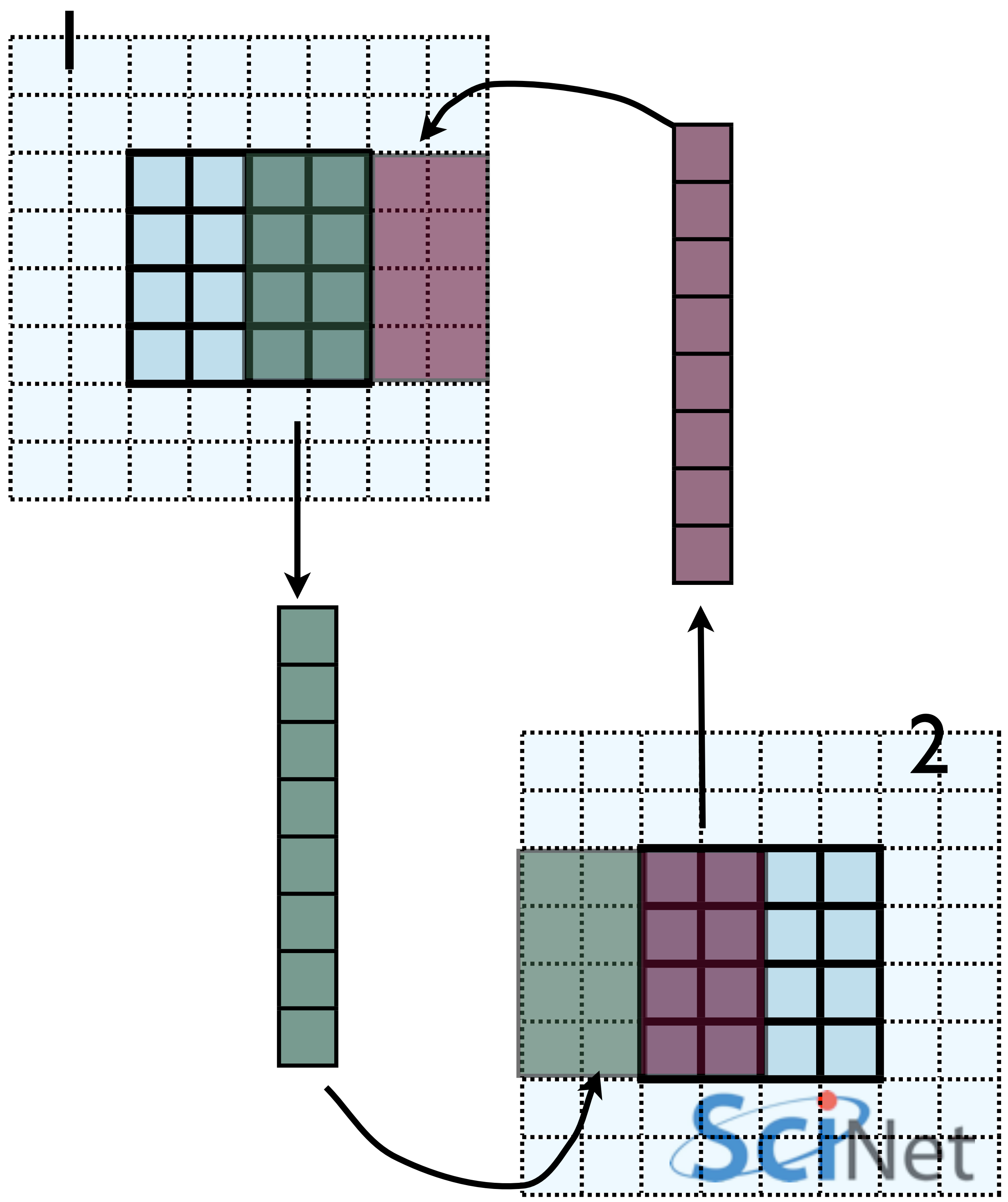


Implementing in MPI

- Let's do this together
- solver.f90/solver.c; implement to bufferGuardcells
- When do we call this in timestep?

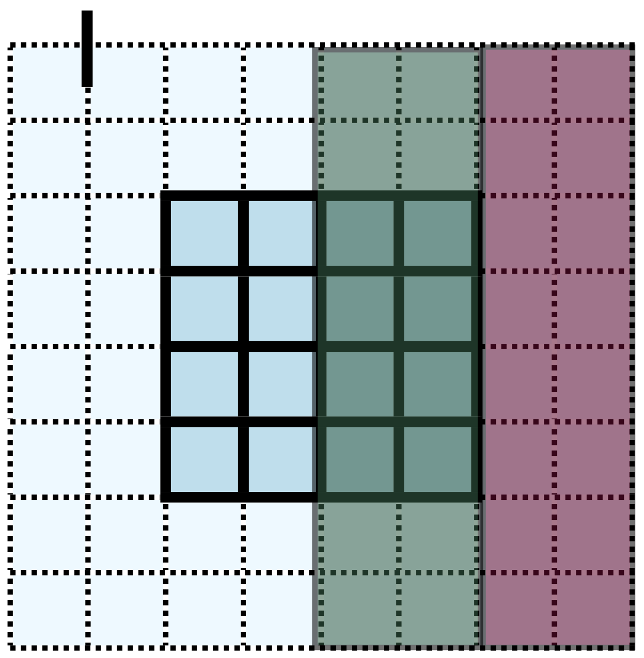
Implementing in MPI

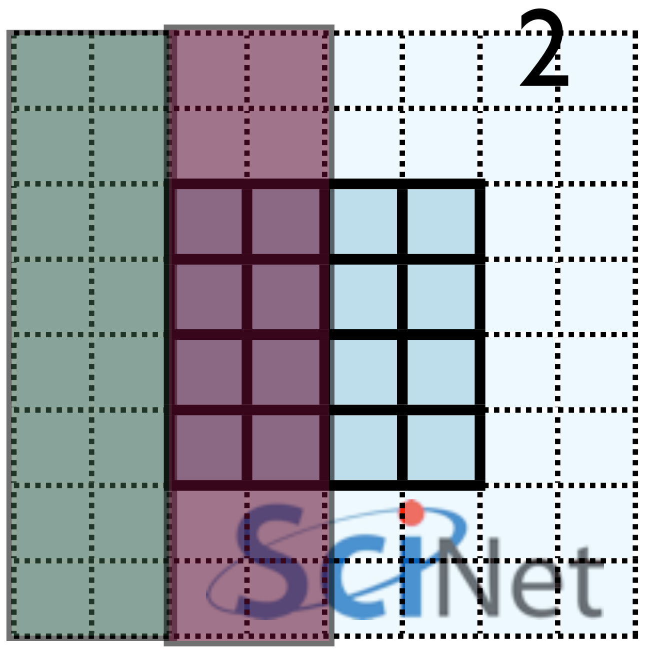
- This approach is simple, but introduces extraneous copies
- Memory bandwidth is already a bottleneck for these codes
- It would be nice to just point at the start of the guardcell data and have MPI read it from there.



Implementing in MPI

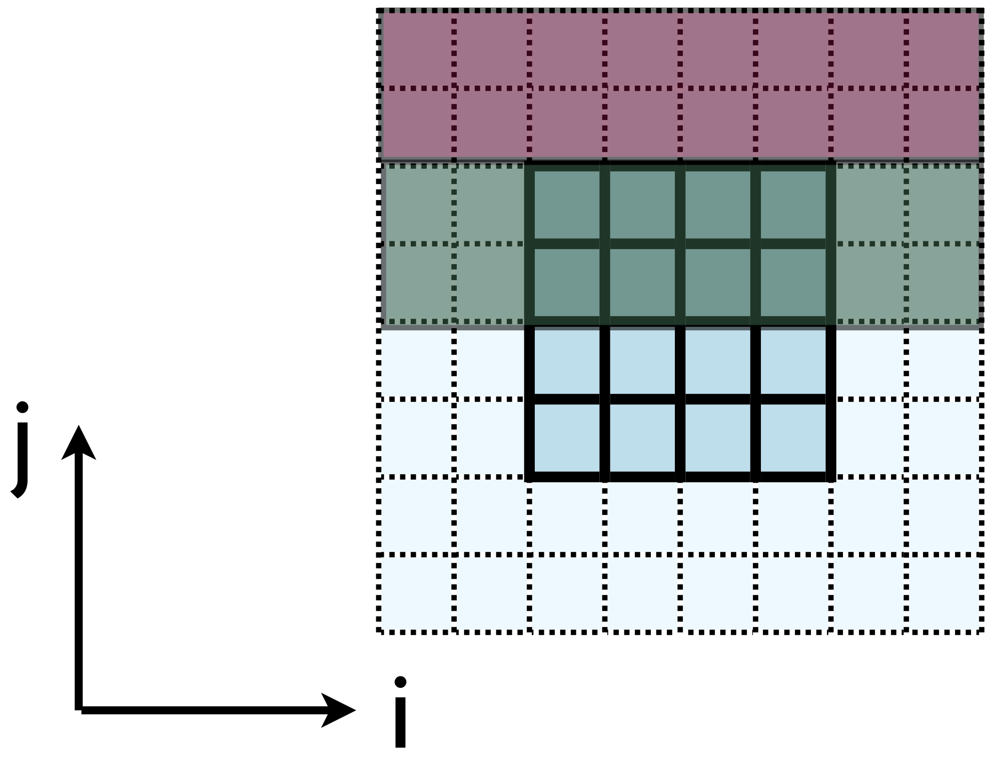
- Let me make one simplification for now; copy whole stripes
- This isn't necessary, but will make stuff simpler at first
- Only a cost of $2xNg^2 = 8$ extra cells (small fraction of ~200-2000 that would normally be copied)

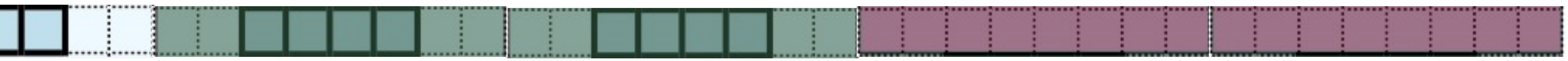




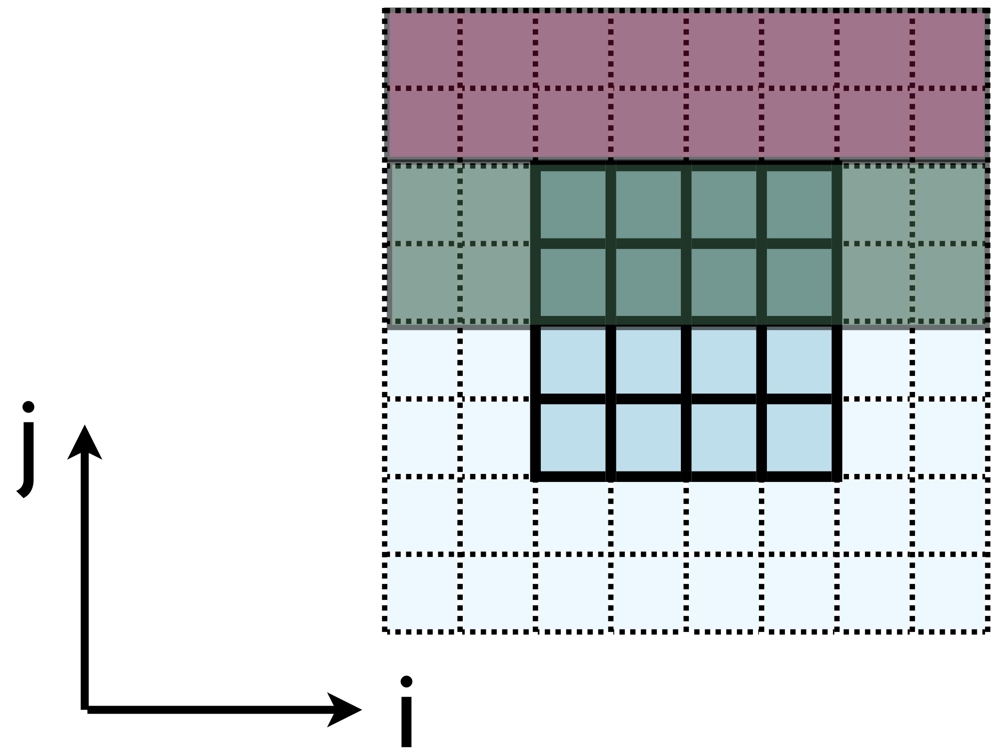
Implementing in MPI

- Recall how 2d memory is laid out
- y-direction guardcells contiguous



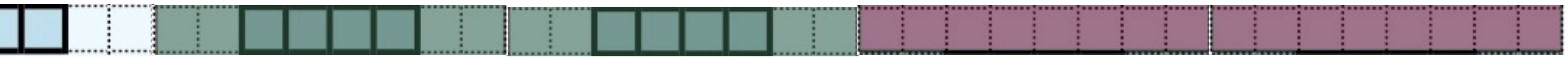


Implementing in MPI



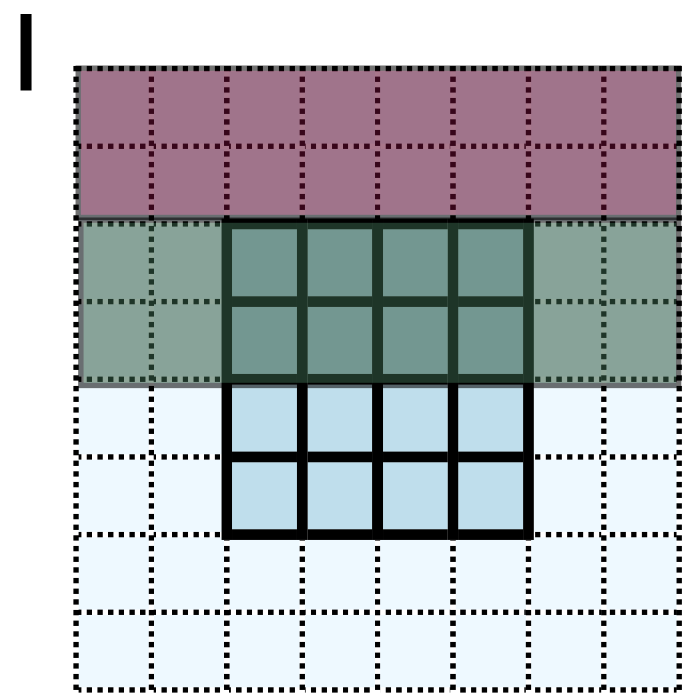
- Can send in one go:

```
call MPI_Send(u(1,1,ny), nvars*nguard*ny, MPI_REAL, ....)
ierr = MPI_Send(&(u[ny][0][0]), nvars*nguard*ny, MPI_FLOAT, ....)
```



Implementing in MPI

- Creating MPI Data types.
- MPI_Type_contiguous: simplest case. Lets you build a string of some other type.

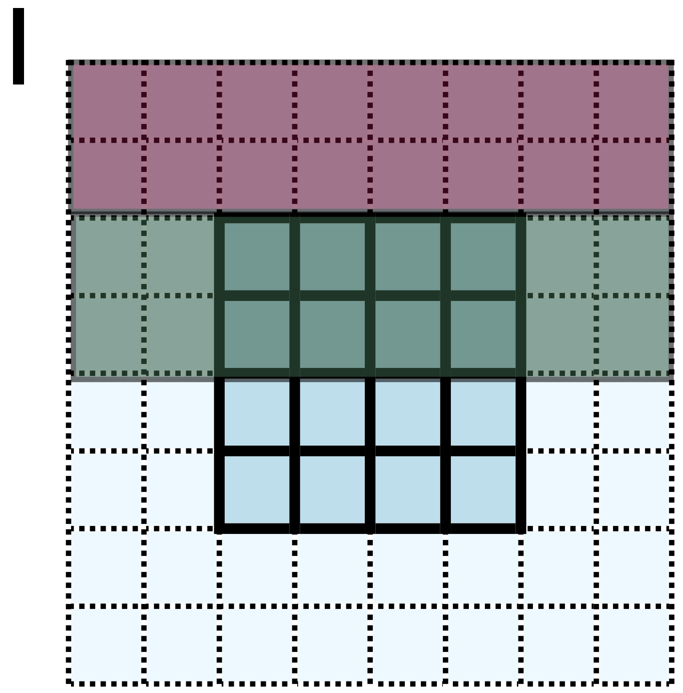


Count OldType &NewType

```
MPI_Datatype ybctype;

ierr = MPI_Type_contiguous(nvals*nguard*(ny), MPI_REAL, &ybctype);
ierr = MPI_Type_commit(&ybctype);

MPI_Send(&(u[ny][0][0]), 1, ybctype, ....)

ierr = MPI_Type_free(&ybctype);
```

Implementing in MPI

- Creating MPI Data types.
- MPI_Type_contiguous: simplest case. Lets you build a string of some other type.

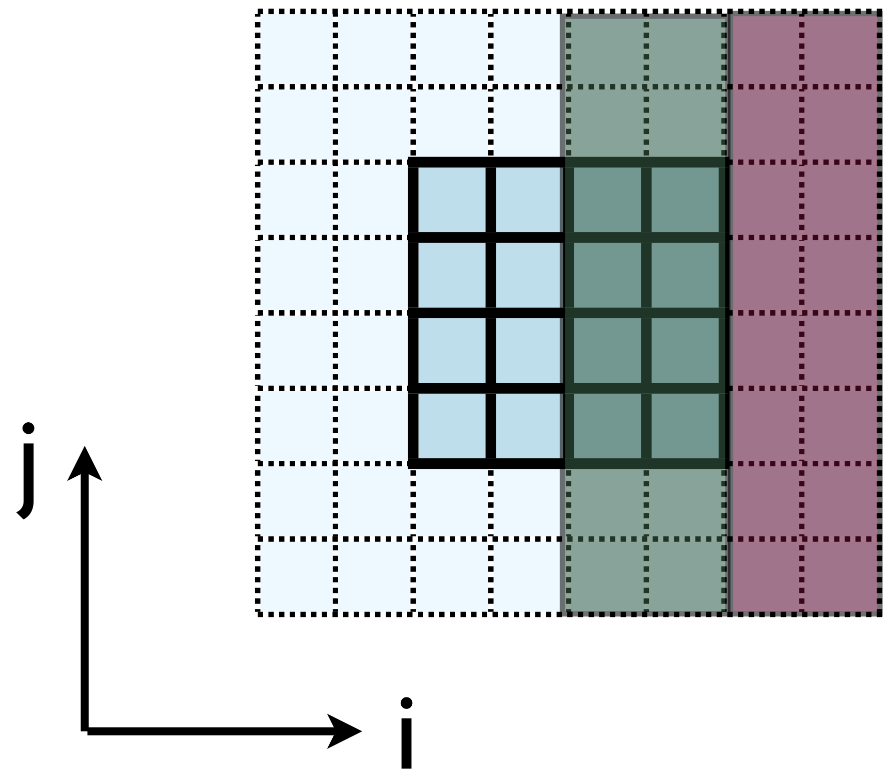


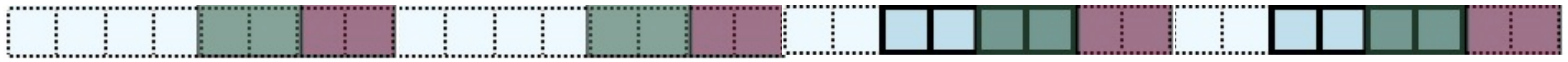
Count OldType NewType

```
integer :: ybctype

call MPI_Type_contiguous(nvals*nguard*(ny), MPI_REAL, ybctype, ierr)
call MPI_Type_commit(ybctype, ierr)

MPI_Send(u(1,1,ny), 1, ybctype, ....)

call MPI_Type_free(ybctype, ierr)
```

Implementing in MPI

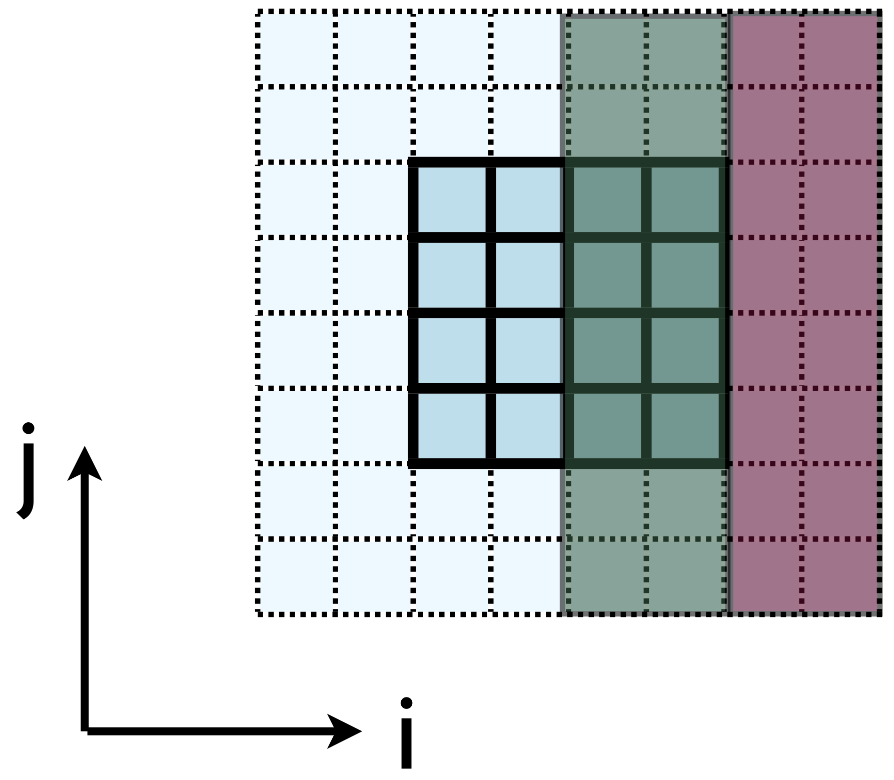
- Recall how 2d memory is laid out
- x gcs or boundary values *not* contiguous
- How do we do something like this for the x-direction?

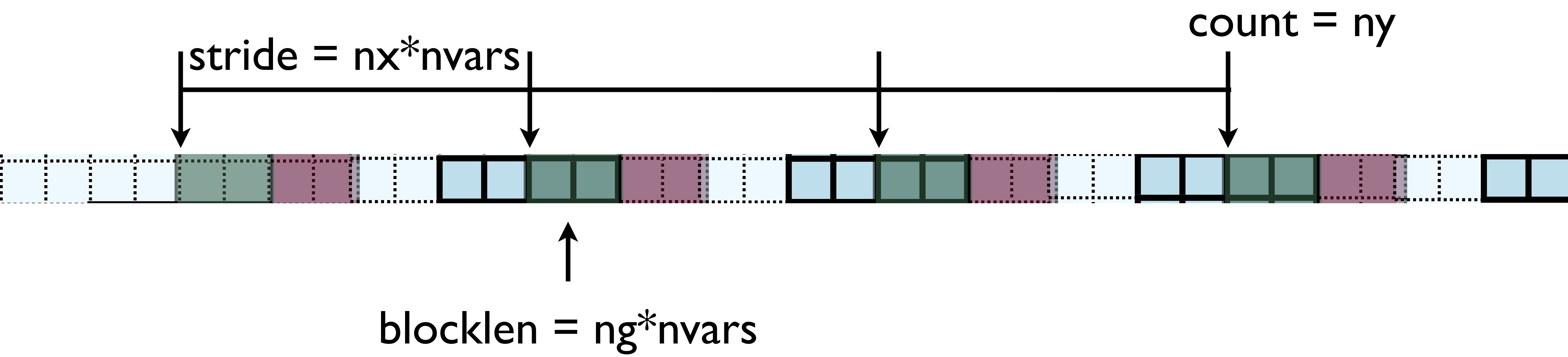




Implementing in MPI

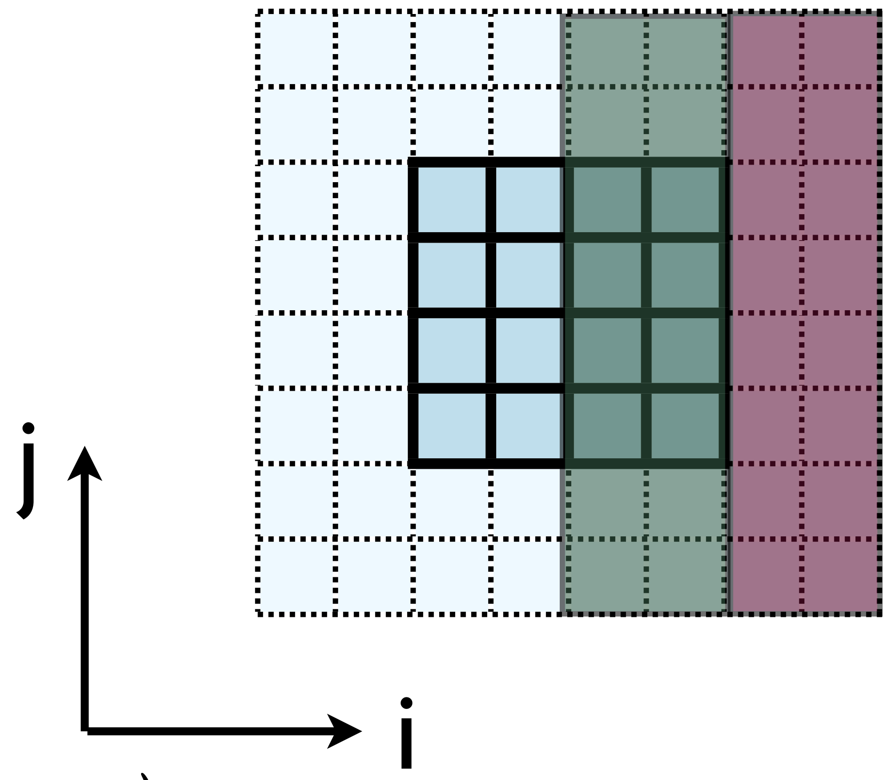
```
int MPI_Type_vector(
        int count,
        int blocklen,
        int stride,
        MPI_Datatype old_type,
        MPI_Datatype *newtype );
```

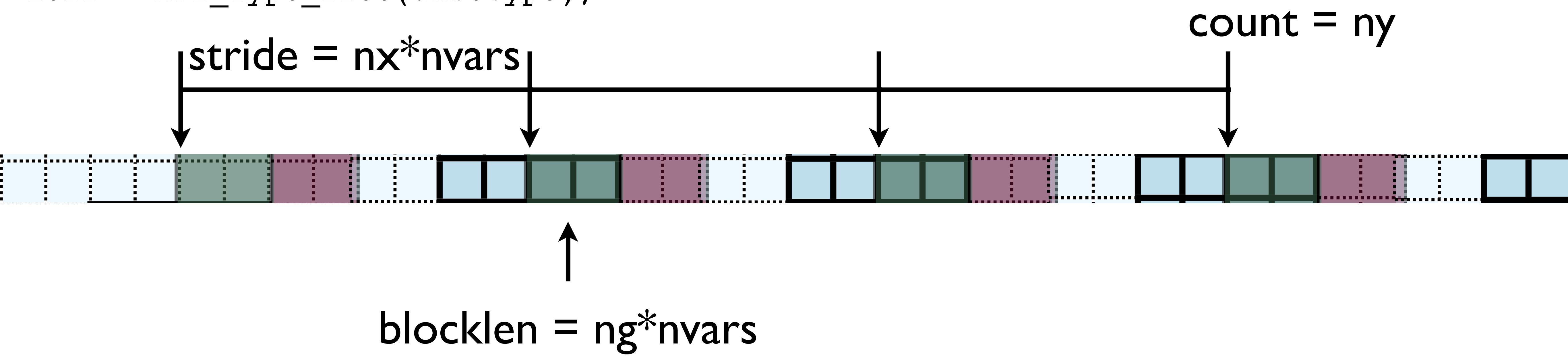




Implementing in MPI

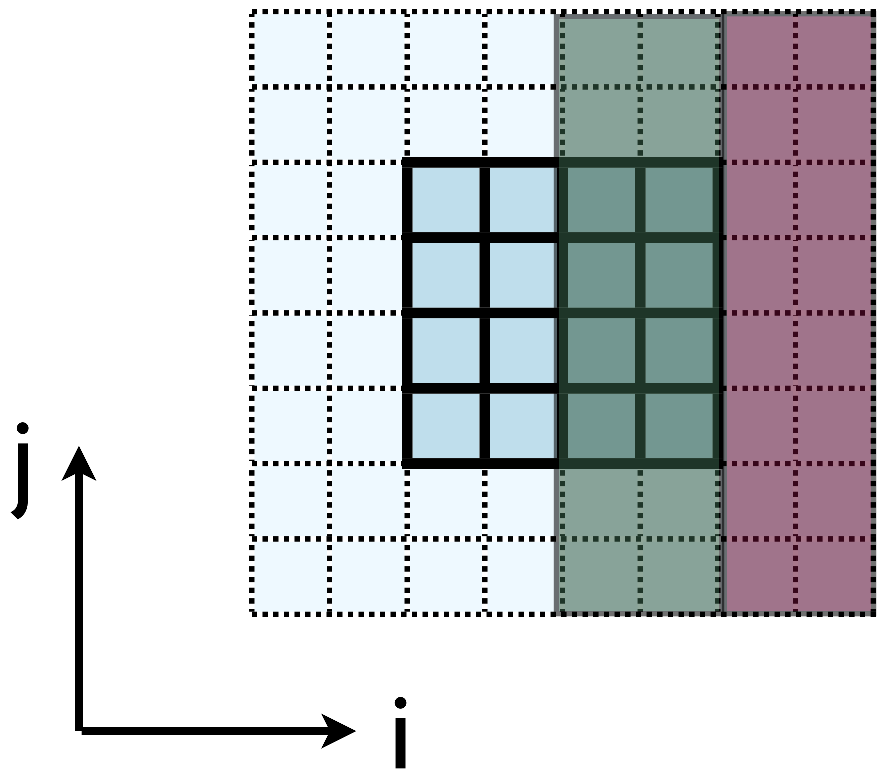
```
ierr = MPI_Type_vector(ny, nguard*nvars,
          nx*nvars, MPI_FLOAT, &xbctype);

ierr = MPI_Type_commit(&xbctype);

ierr = MPI_Send(&(u[0][nx][0]), 1, xbctype, ....)

ierr = MPI_Type_free(&xbctype);
```

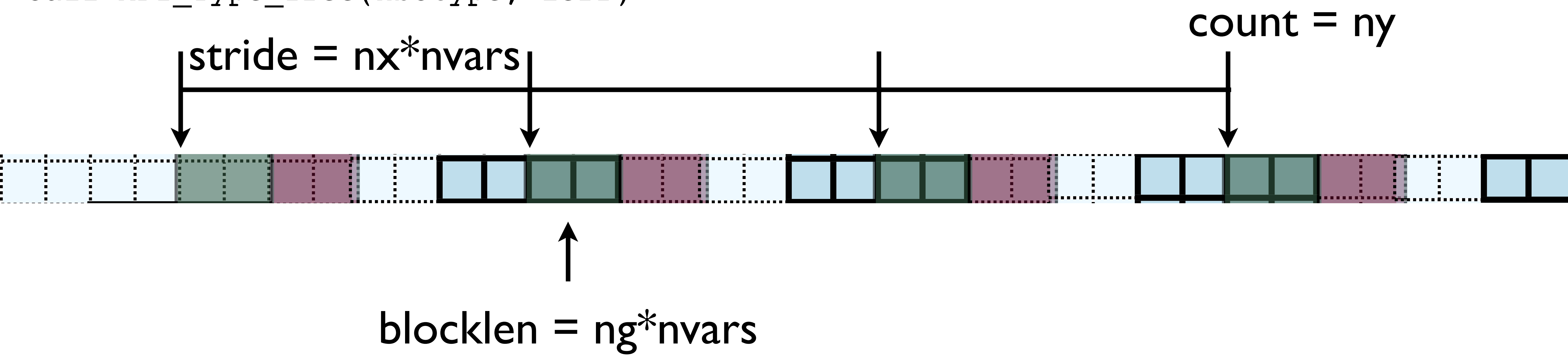




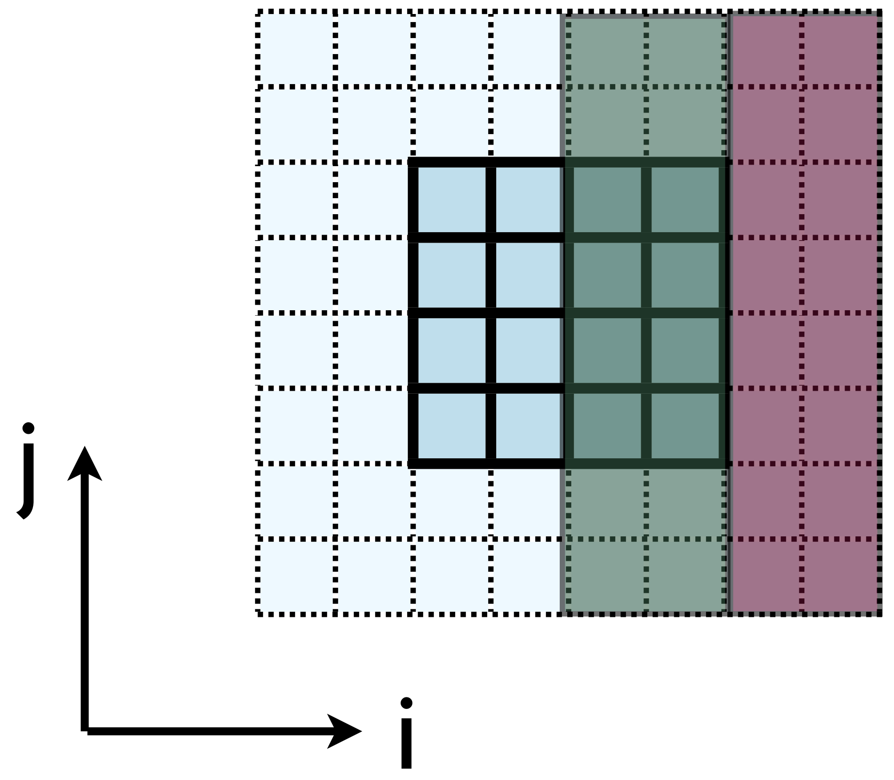
Implementing in MPI

```
call MPI_Type_vector(ny, nguard*nvars,
        nx*nvars, MPI_REAL, xbctype, ierr)

call MPI_Type_commit(xbctype, ierr)

call MPI_Send(u(1,nx,1), 1, ybctype, ....)

call MPI_Type_free(xbctype, ierr)
```

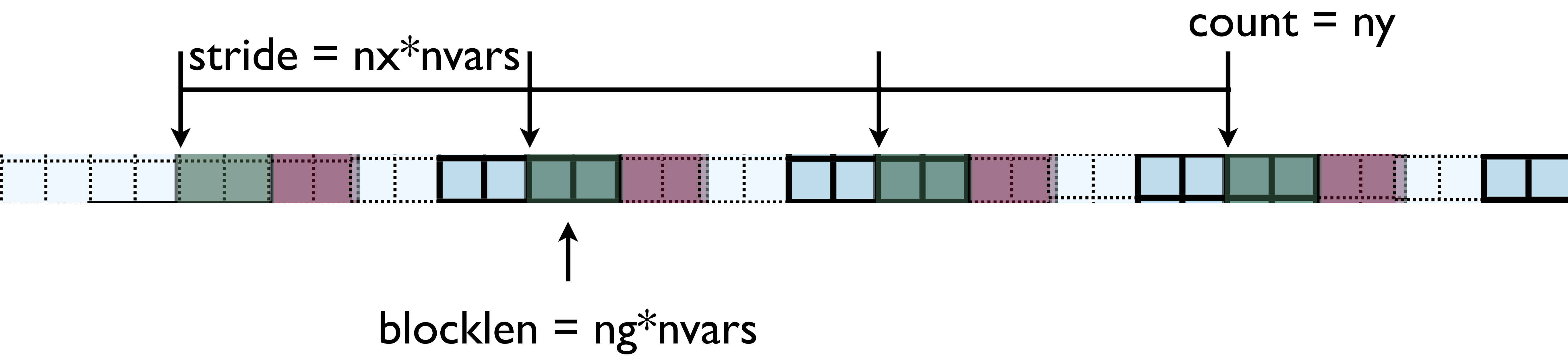




Implementing in MPI

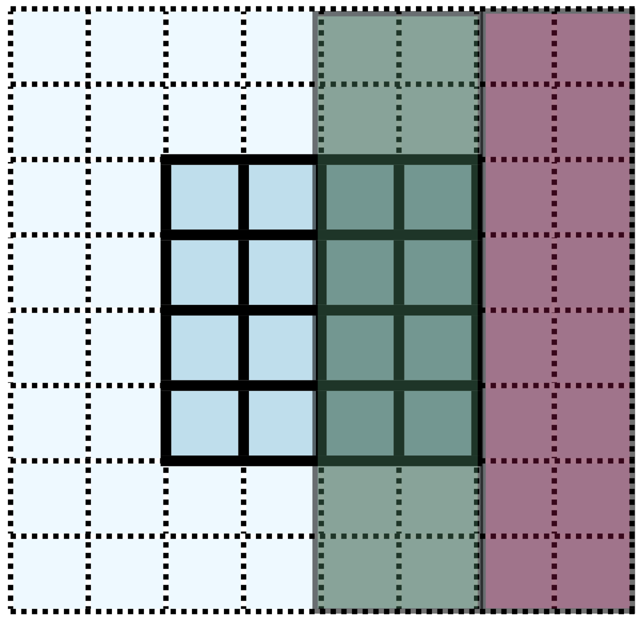


- Check: total amount of data = blocklen*count = ny*ng*nvars
- Skipped over stride*count = nx*ny*nvars



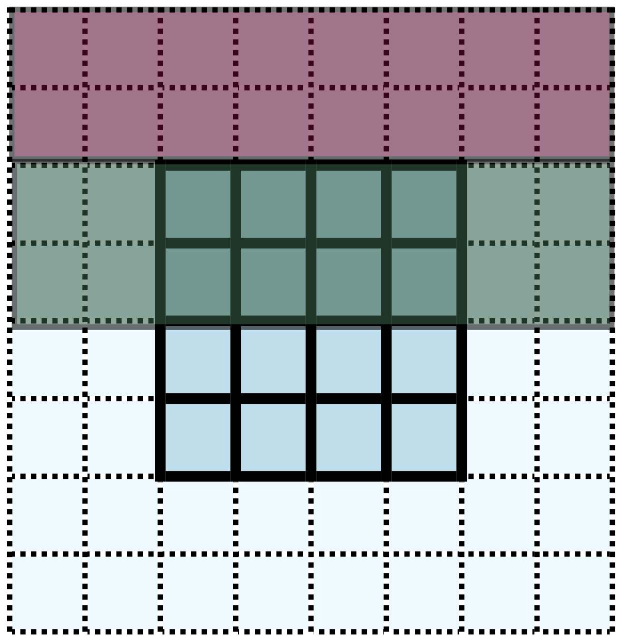
Implementing in MPI

- Hands-On: Implement X guardcell filling with types.
- Implement vectorGuardCells
- For now, create/free type each cycle through; ideally, we'd create/free these once.



In MPI, there's always more than one way..

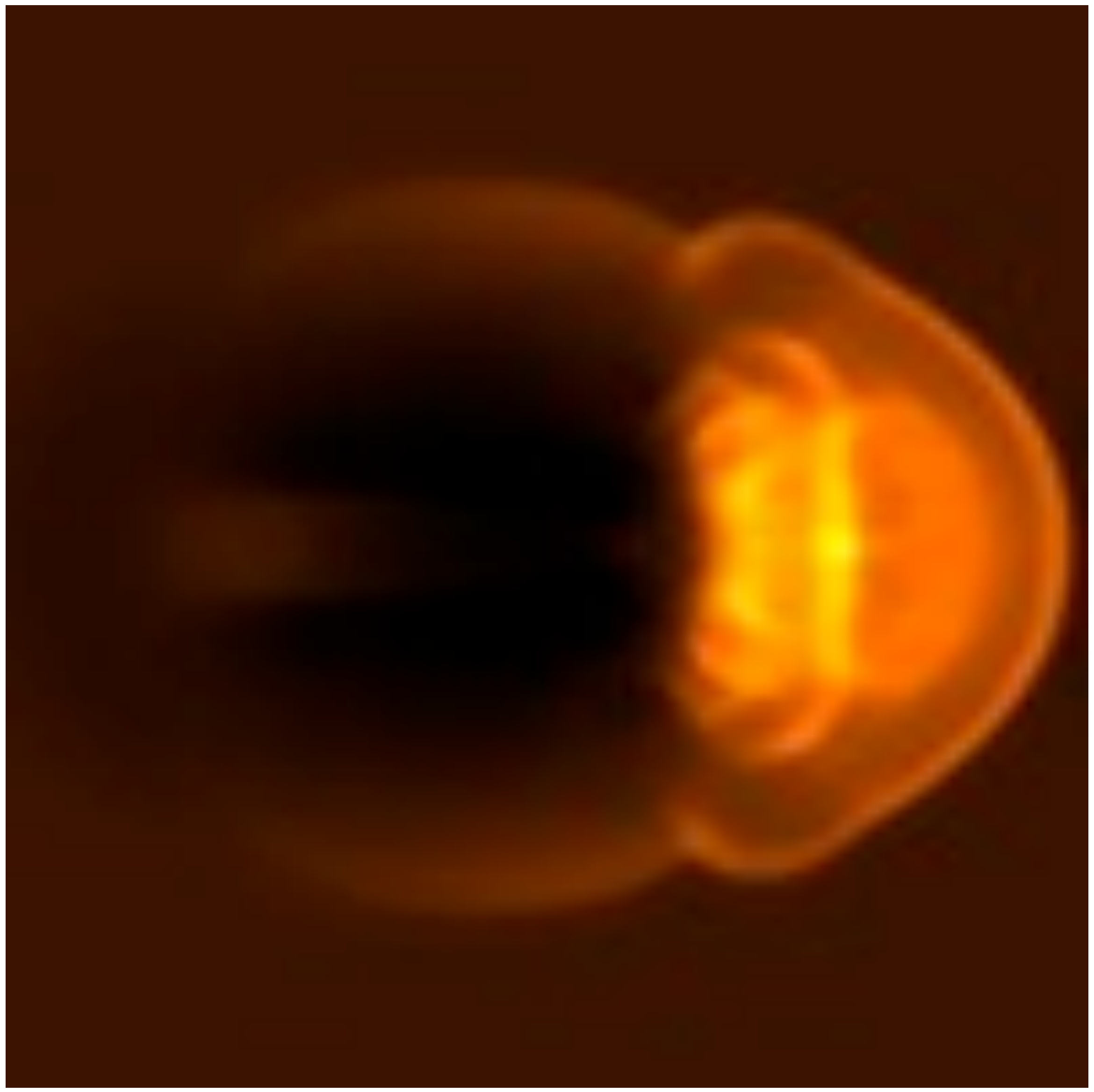
- MPI_Type_create_subarray ; piece of a multi-dimensional array.
- *Much* more convenient for higher-dimensional arrays
- (Otherwise, need vectors of vectors of vectors...)



```
int MPI_Type_create_subarray(
        int ndims, int *array_of_sizes,
        int *array_of_subsizes,
        int *array_of_starts,
        int order,
        MPI_Datatype oldtype,
        MPI_Datatype &newtype);

call MPI_Type_create_subarray(
        integer ndims, [array_of_sizes],
        [array_of_subsizes],
        [array_of_starts],
        order, oldtype,
        newtype, ierr)
```

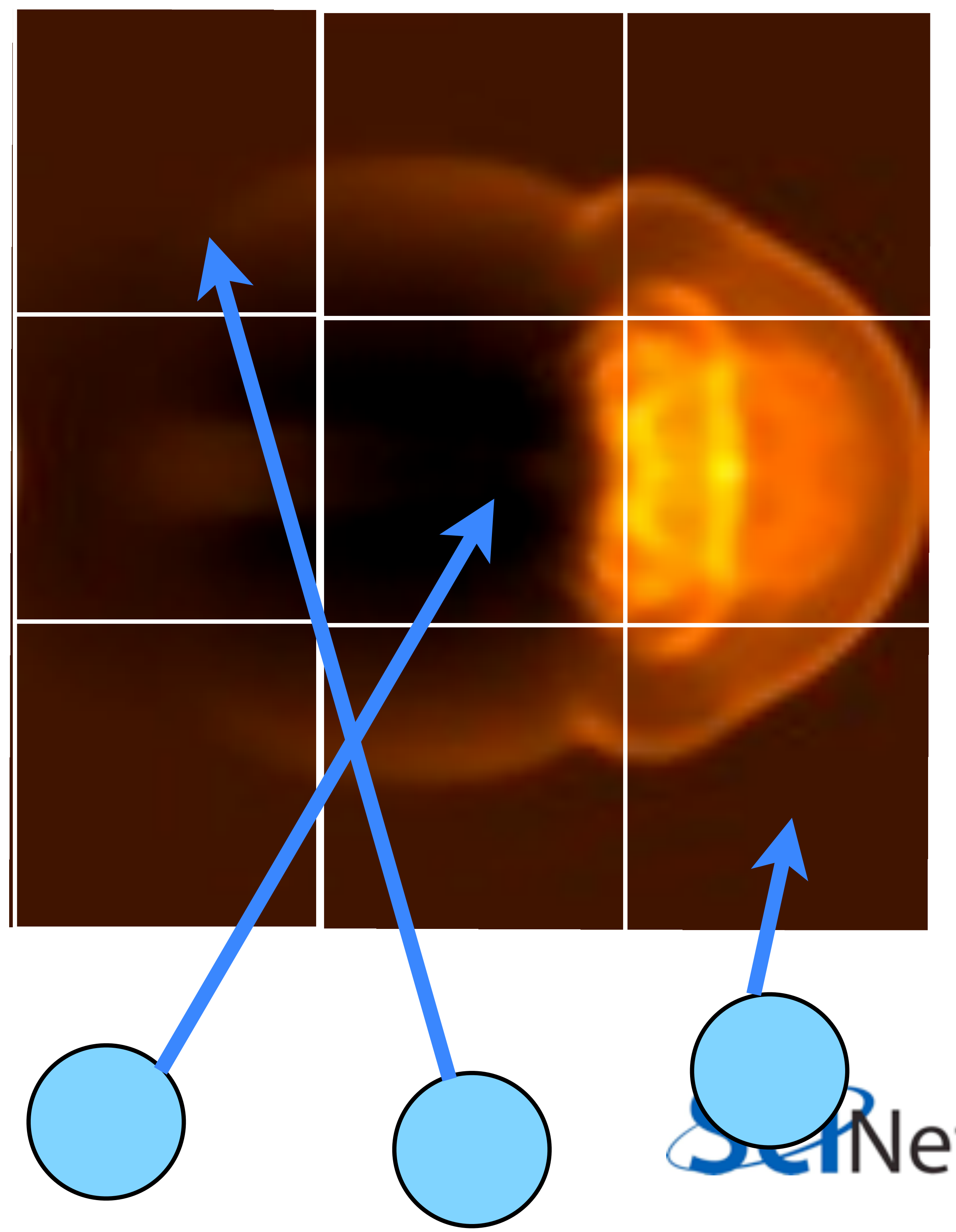
MPI-IO

- Would like the new, parallel version to still be able to write out single output files.
- But at no point does a single processor have entire domain...



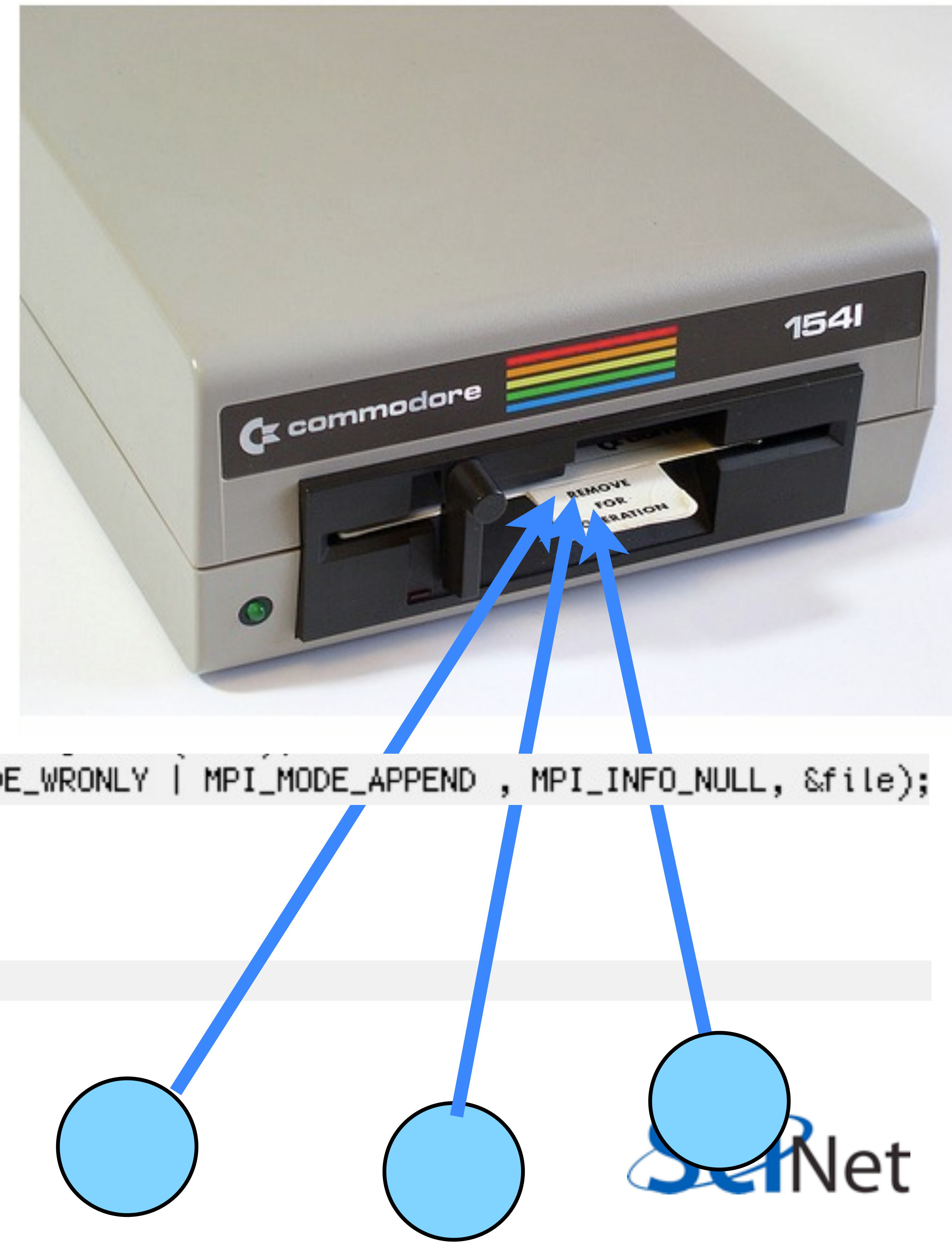
Parallel I/O

- Each processor has to write its own piece of the domain..
- without overwriting the other.
- Easier if there is global coordination



MPI-IO

- Uses MPI to coordinate reading/writing to single file



```
ierr = MPI_File_open(MPI_COMM_WORLD,filename, MPI_MODE_WRONLY | MPI_MODE_APPEND , MPI_INFO_NULL, &file);
```

...stuff...

```
ierr = MPI_File_close(&file);
```

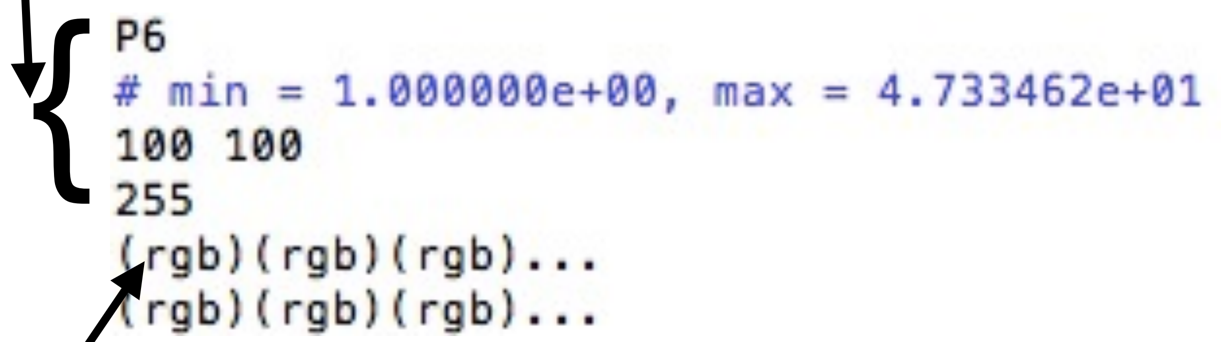
- Coordination -- *collective* operations.

PPM file format

- Simple file format
- Someone has to write a header, then each PE has to output only its 3-bytes pixels skipping everyone elses.

header -- ASCII characters

'P6', comments, height/width, max val

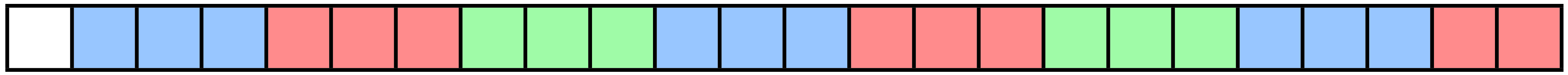


```
P6
# min = 1.000000e+00, max = 4.733462e+01
100 100
255
(rgb)(rgb)(rgb)...
(rgb)(rgb)(rgb)...
```

row by row triples of bytes: each pixel = 3 bytes

MPI-IO File View

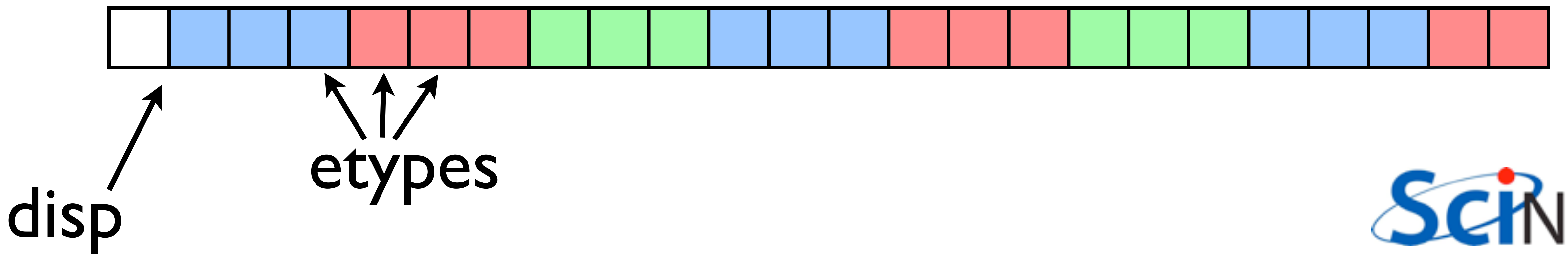
- Each process has a view of the file that consists of only of the parts accessible to it.
- For writing, hopefully non-overlapping!
- Describing this - how data is laid out in a file - is very similar to describing how data is laid out in memory...



MPI-IO File View

- int MPI_File_set_view(

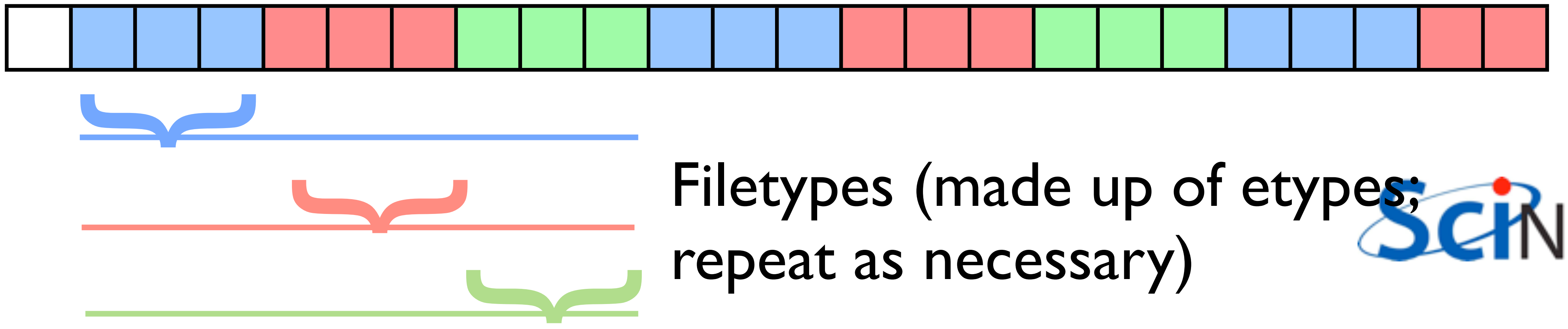
```
    MPI_File fh,
    MPI_Offset disp,           /* displacement in bytes from start */
    MPI_Datatype etype,        /* elementary type */
    MPI_Datatype filetype,     /* file type; prob different for each proc */
    char *datarep,             /* 'native' or 'internal' */
    MPI_Info info)             /* MPI_INFO_NULL for today */
```



MPI-IO File View

- int MPI_File_set_view(

```
    MPI_File fh,
    MPI_Offset disp,          /* displacement in bytes from start */
    MPI_Datatype etype,       /* elementary type */
    MPI_Datatype filetype,    /* file type; prob different for each proc */
    char *datarep,            /* 'native' or 'internal' */
    MPI_Info info)            /* MPI_INFO_NULL */
```



Filetypes (made up of etypes; repeat as necessary)

MPI-IO File Write

- int MPI_File_write_all(
    MPI_File fh,
    void *buf,
    int count,
    MPI_Datatype datatype,
    MPI_Status *status)

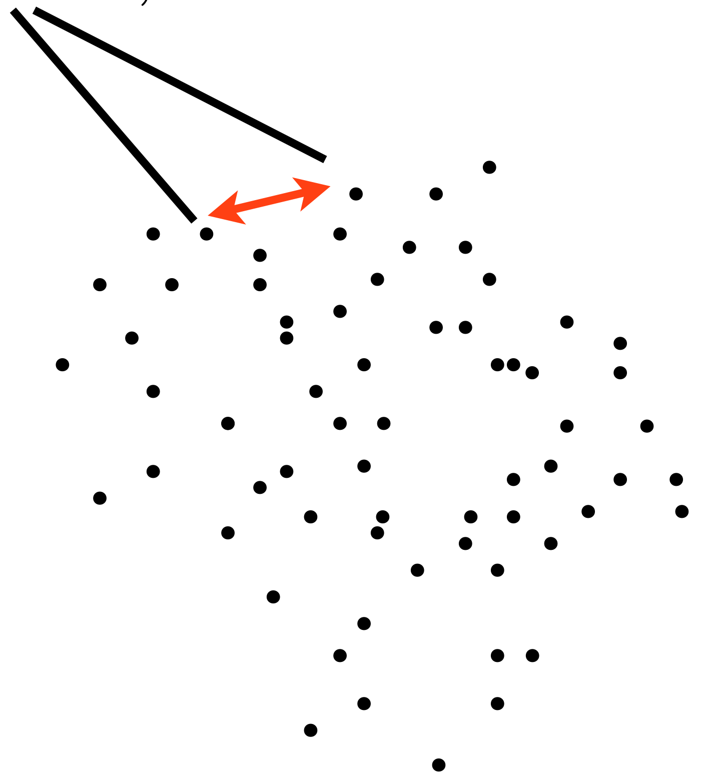
Writes (_all: collectively) to part of file within view.

Hands On

- Implement the ppm routines collectively using the subarray type.

N-Body Dynamics

$$F_{1,2} = -\frac{G m_1 m_2}{r_{1,2}^2}\hat{\mathbf{r}}$$

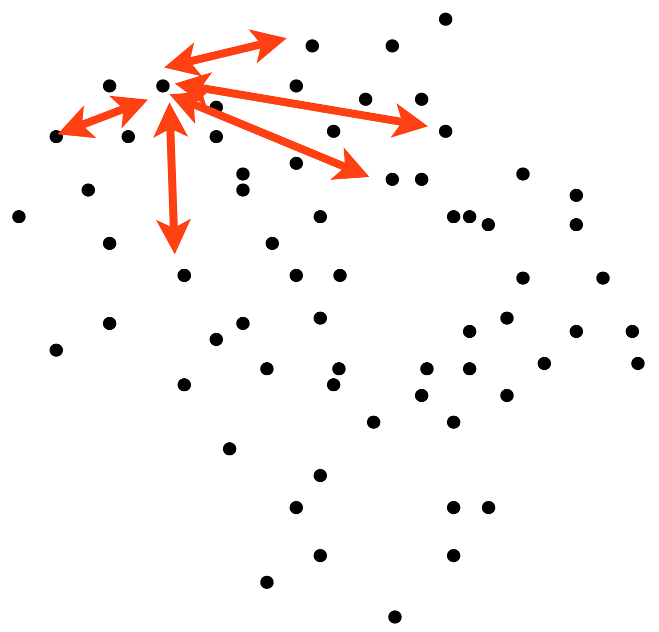


N-Body dynamics

- N interacting bodies
- Pairwise forces; here, Gravity.
- (here, stars in a cluster; could be molecular dynamics, economic agents...)

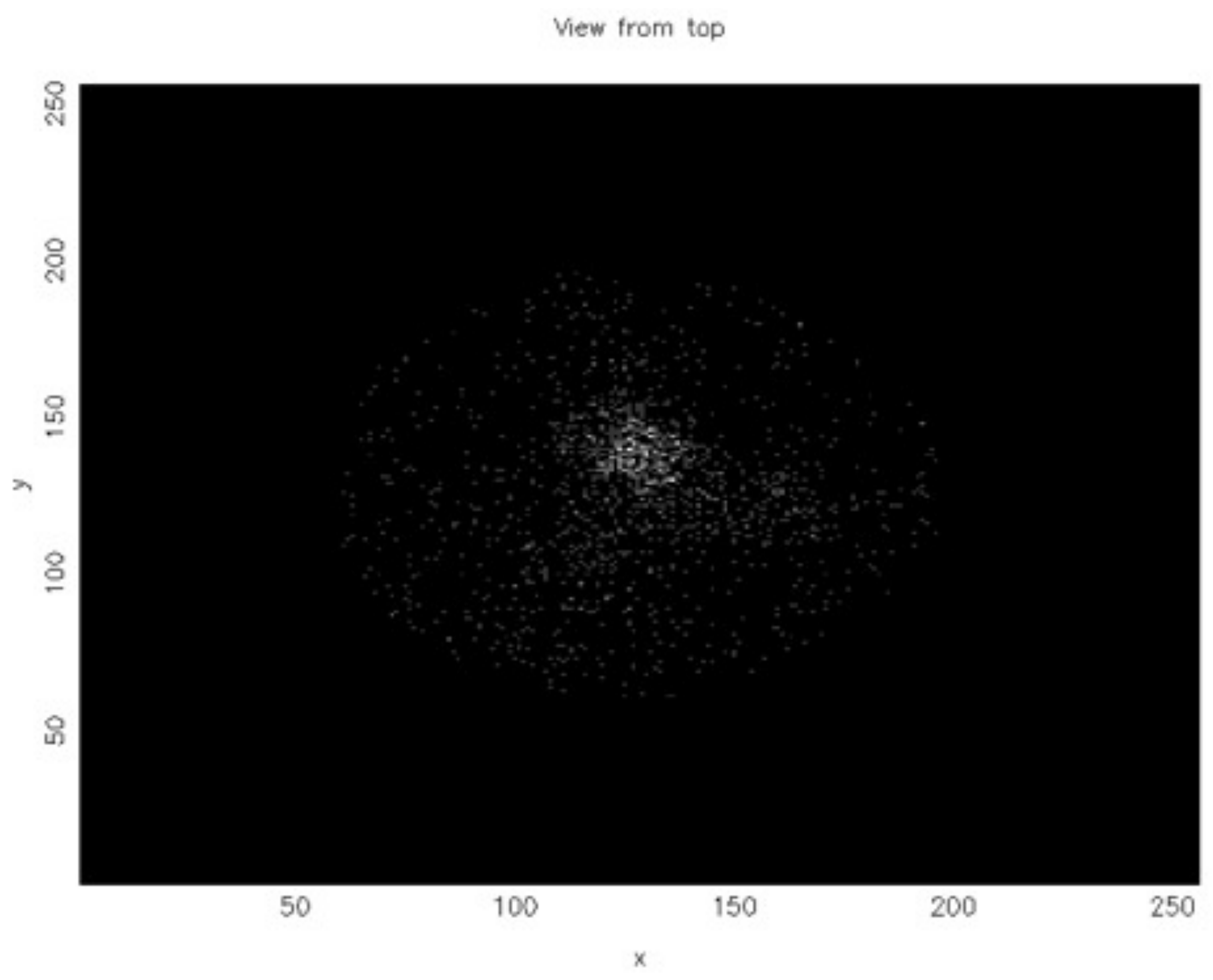
N-Body dynamics

- N interacting bodies
- Pairwise forces; here, Gravity.
- (here, stars in a cluster; could be molecular dynamics, economic agents...)



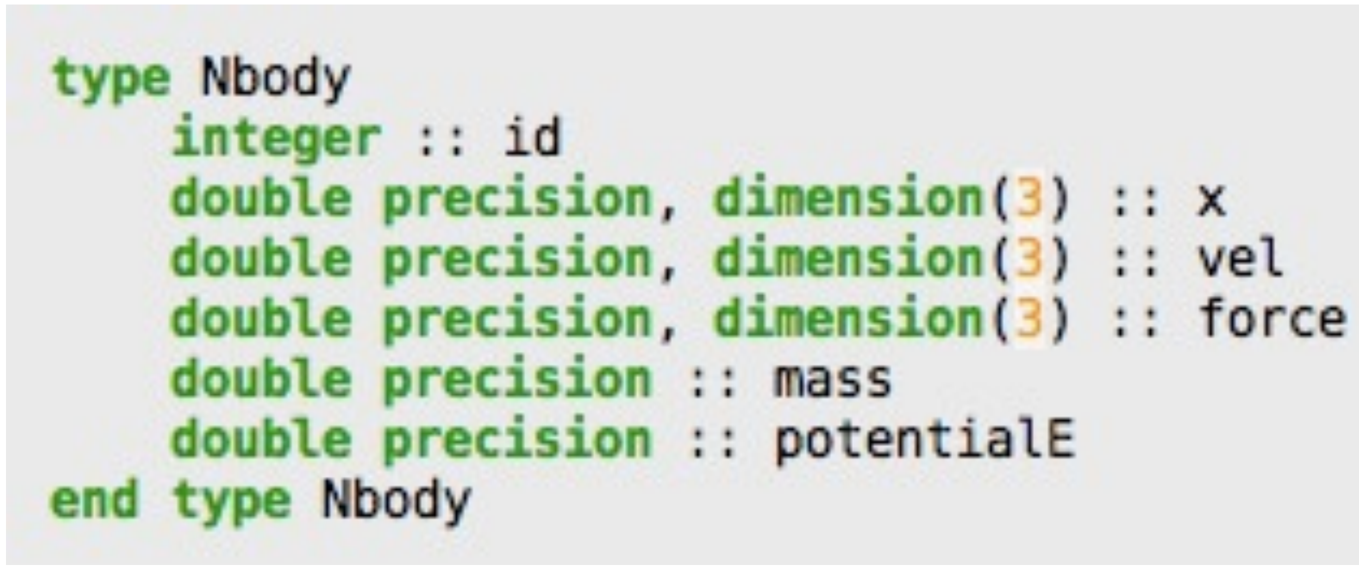
nbody

- `cd ~/mpi/nbodyc`
- `make`
- `./nbodyc`



A Particle type

- Everything based on a array of structures ('derived data types')

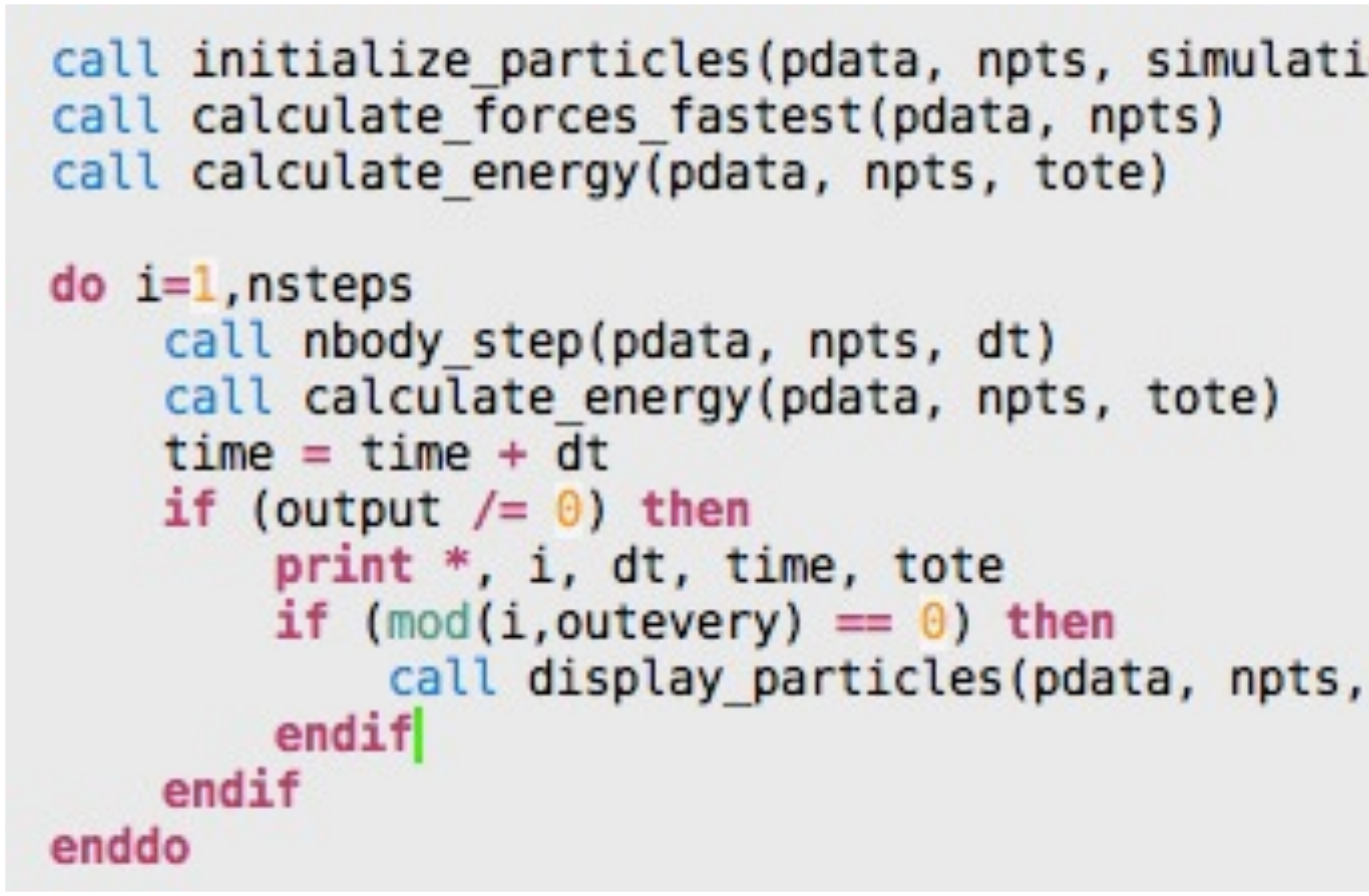


```
type Nbody
    integer :: id
    double precision, dimension(3) :: x
    double precision, dimension(3) :: vel
    double precision, dimension(3) :: force
    double precision :: mass
    double precision :: potentialE
end type Nbody
```

`nbody.f90`, line 5

Main loop

- nbody_step - calls calculate forces, updates positions.
- calculate energy (diagnostic)
- display particles.

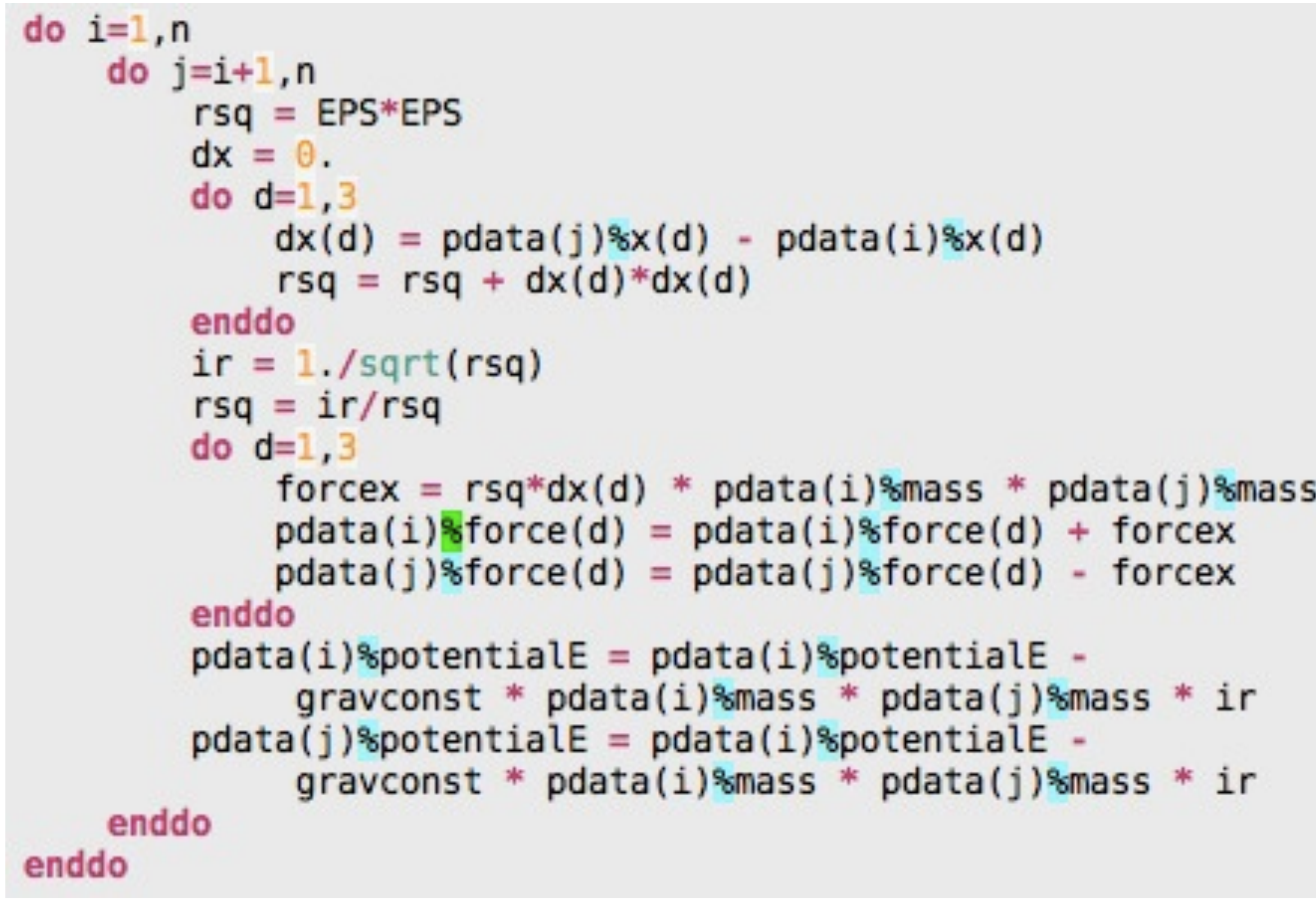


```
call initialize_particles(pdata, npts, simulati
call calculate_forces_fastest(pdata, npts)
call calculate_energy(pdata, npts, tote)

do i=1,nsteps
    call nbody_step(pdata, npts, dt)
    call calculate_energy(pdata, npts, tote)
    time = time + dt
    if (output /= 0) then
        print *, i, dt, time, tote
        if (mod(i,outevery) == 0) then
            call display_particles(pdata, npts,
        endif
    endif
enddo
```

`nbody.f90`, line 35

Calculate Forces

- For each particle i
- Foreach other particle j>i
- Calculate distance (most expensive!)
- Increment force
- Increment potential energy

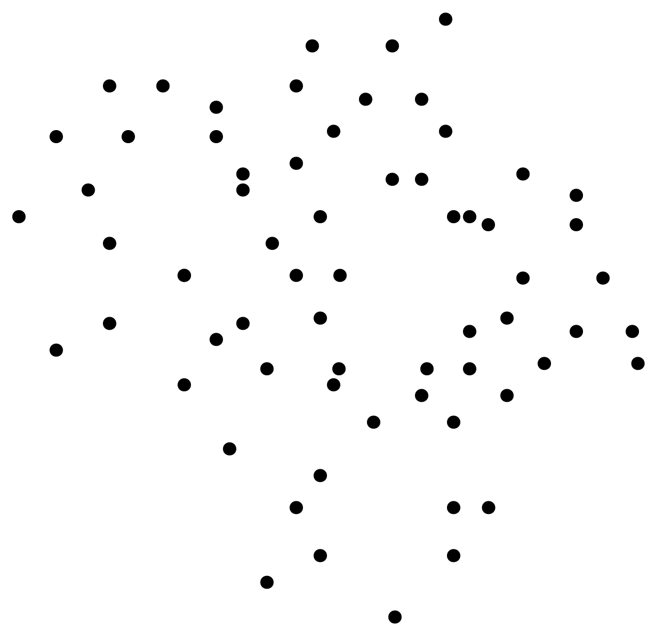


```
do i=1,n
    do j=i+1,n
        rsq = EPS*EPS
        dx = 0.
        do d=1,3
            dx(d) = pdata(j)%x(d) - pdata(i)%x(d)
            rsq = rsq + dx(d)*dx(d)
        enddo
        ir = 1./sqrt(rsq)
        rsq = ir/rsq
        do d=1,3
            forcex = rsq*dx(d) * pdata(i)%mass * pdata(j)%mass
            pdata(i)%force(d) = pdata(i)%force(d) + forcex
            pdata(j)%force(d) = pdata(j)%force(d) - forcex
        enddo
        pdata(i)%potentialE = pdata(i)%potentialE -
             gravconst * pdata(i)%mass * pdata(j)%mass * ir
        pdata(j)%potentialE = pdata(i)%potentialE -
             gravconst * pdata(i)%mass * pdata(j)%mass * ir
    enddo
enddo
```

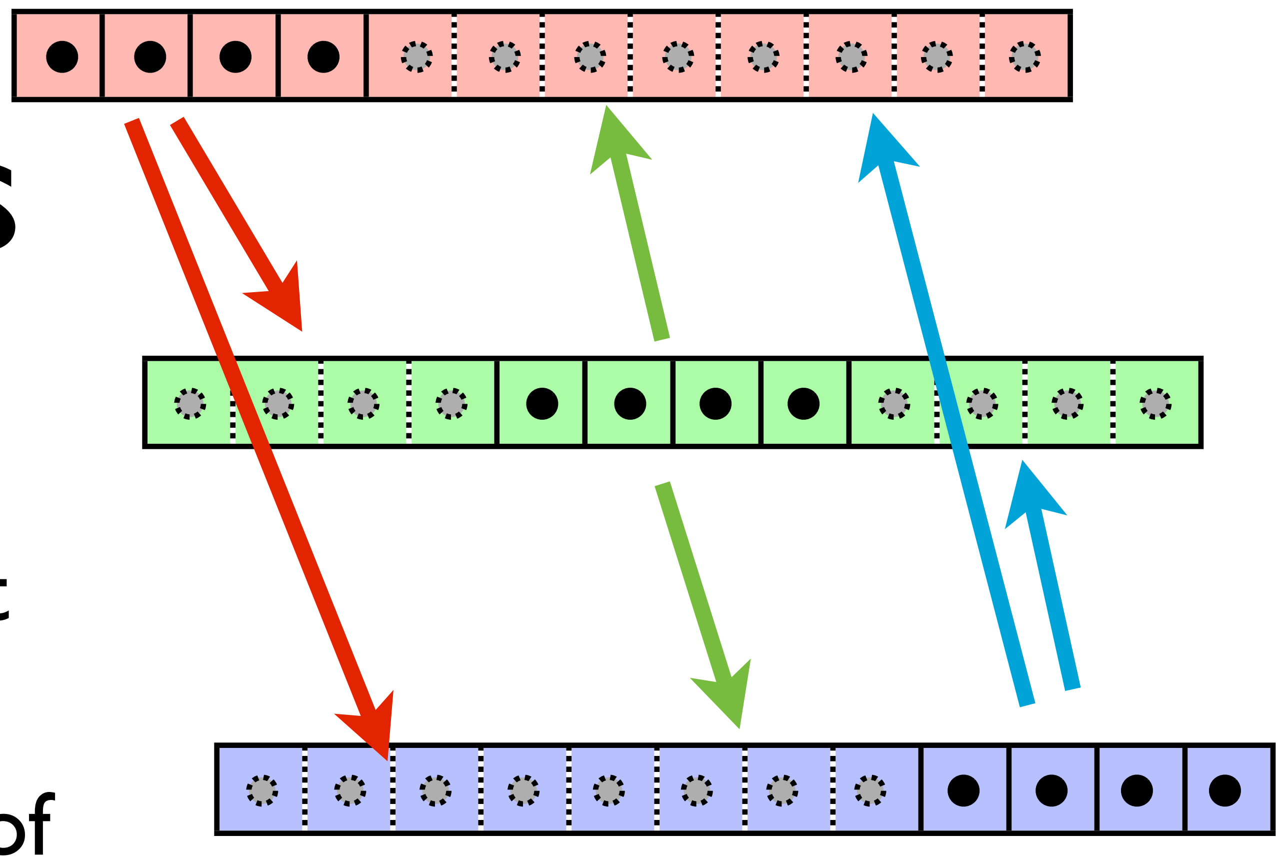
`nbody.f90`, line 100

Decomposing onto different processors

- Direct summation (N^2) - each particle needs to know about all other particles
- Limited locality possible
- Inherently a difficult problem to parallelize in distributed memory



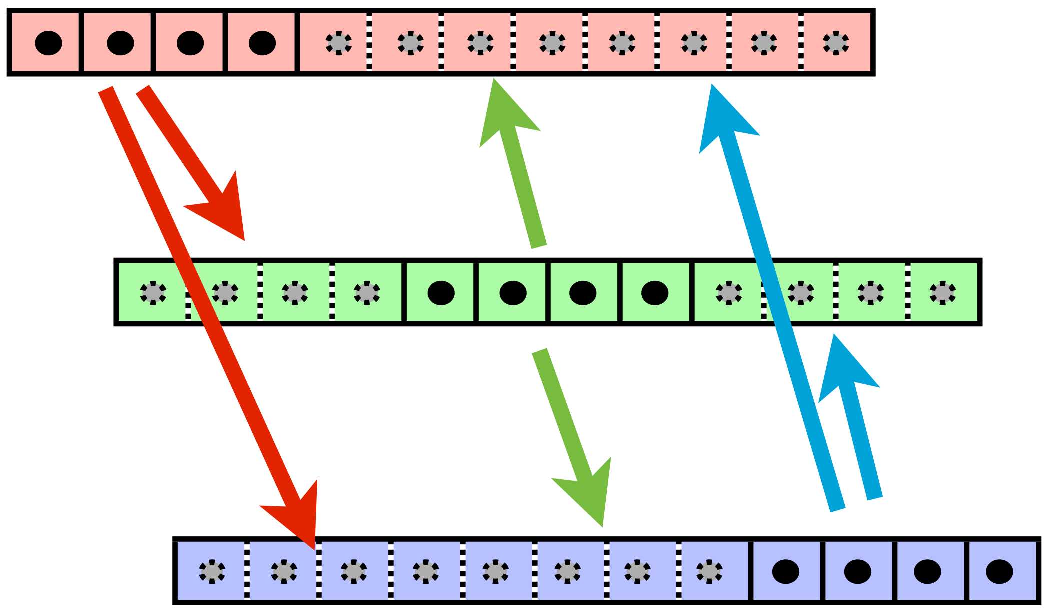
First go: Everyone sees everything



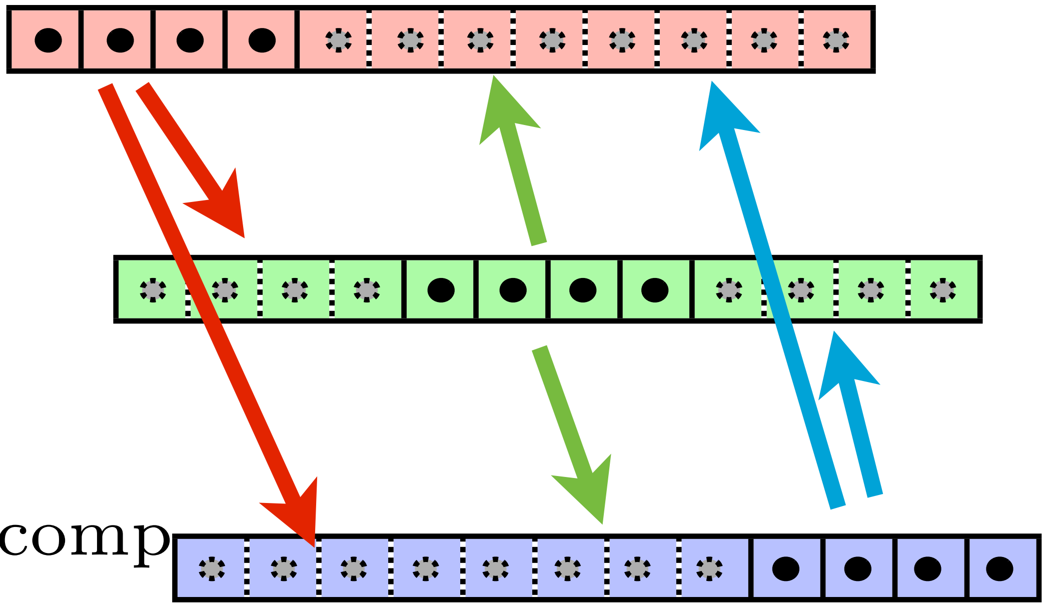
- Distribute the work, but not the data
- Everyone has complete set of particle data
- Just work on our own particles
- Send everyone our particles' data afterwards

Terrible Idea (I)

- Requires the entire problem to fit in the memory of each node.
- In general, you can't do that ($10^{10\text{-}11}$ particle simulation)
- No good for MD, astrophysics but could be useful in other areas (few bodies, complicated interactions) - agent-based simulation
- Best approach depends on your problem



Terrible Idea (II)



$$T_{\mathrm{comp}} \sim c_{\mathrm{grav}} \left(\frac{N}{P}\right) N C_{\mathrm{comp}}$$

$$= c_{\mathrm{grav}} \frac{N^2}{P} C_{\mathrm{comp}}$$

$$T_{\mathrm{comm}} \sim c_{\mathrm{particle}} \frac{N}{P} (P-1) C_{\mathrm{comm}}$$

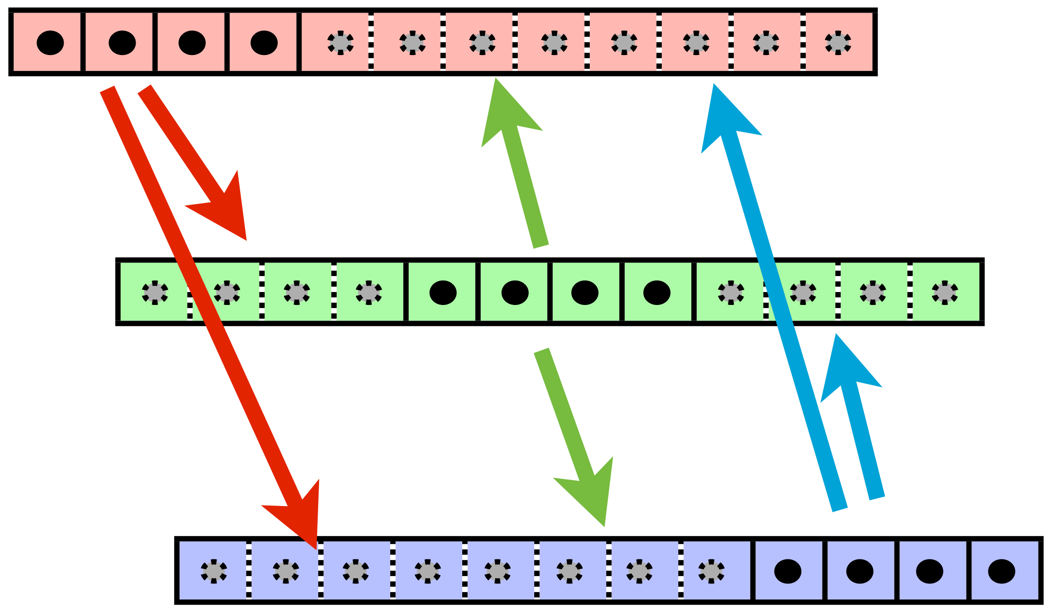
$$\approx c_{\mathrm{particle}} N C_{\mathrm{comm}}$$

$$\frac{T_{\mathrm{comm}}}{T_{\mathrm{comp}}} \approx \frac{c_{\mathrm{particle}}}{c_{\mathrm{grav}}} \frac{1}{N} P \frac{C_{\mathrm{comm}}}{C_{\mathrm{comp}}}$$

Since N is fixed, as P goes up, this fraction gets worse and worse

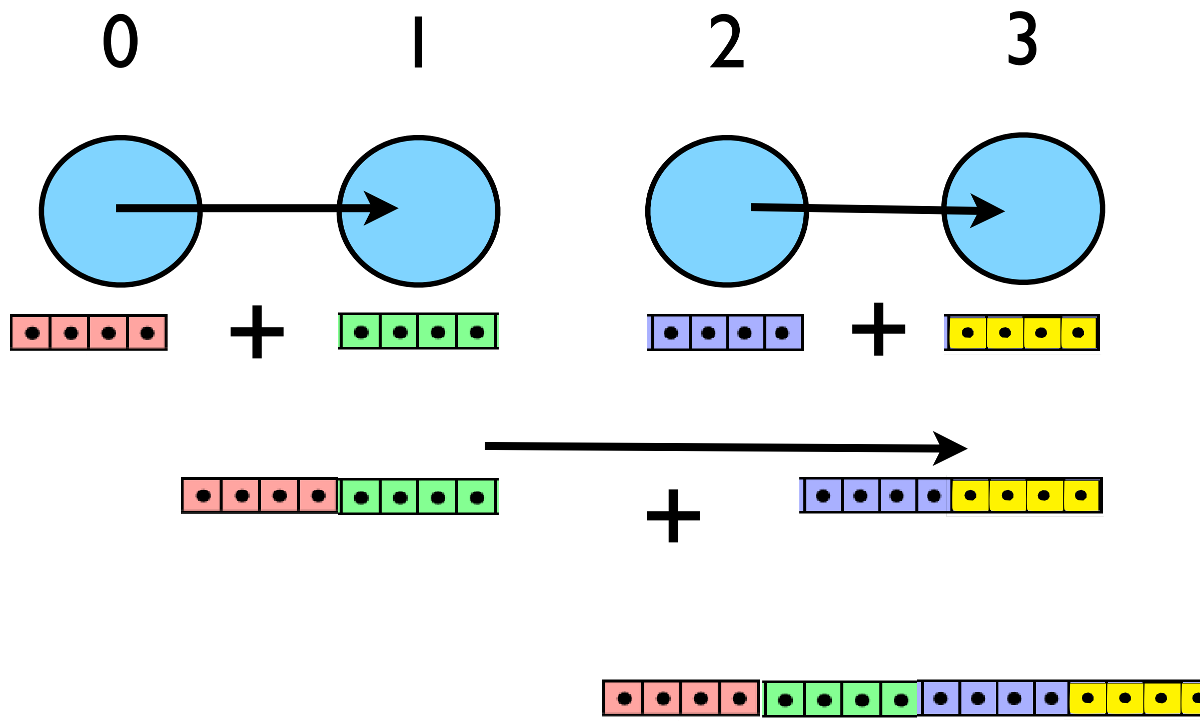
Terrible Idea (III)

- Wastes computation.
- Proc 0 and Proc 2 both calculate the force between particle 1 and particle 11.



Can address (II) a little

- Collecting everyone's data is like a global sum
- (Concatenation is the sort of operation that allows reduction)
- GATHER operation
- Send back the results: ALLGATHER
- 2 (P-1) vs P^2 messages, but length differs



Avg Message Length

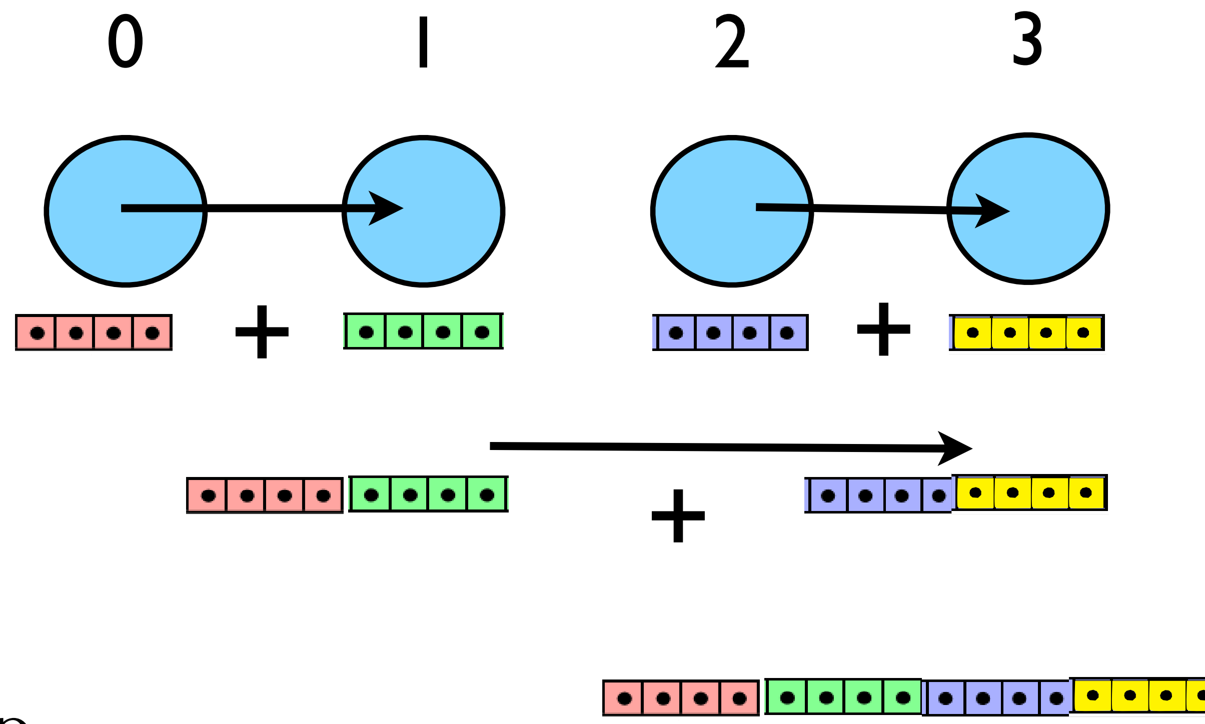
$=(N/2 \log_2 P)/(P-1)$

$\sim N + N/P \log_2(P)$

Total sent ~

$2 N \log_2(P)$ vs $N P$

Can address (I) a little

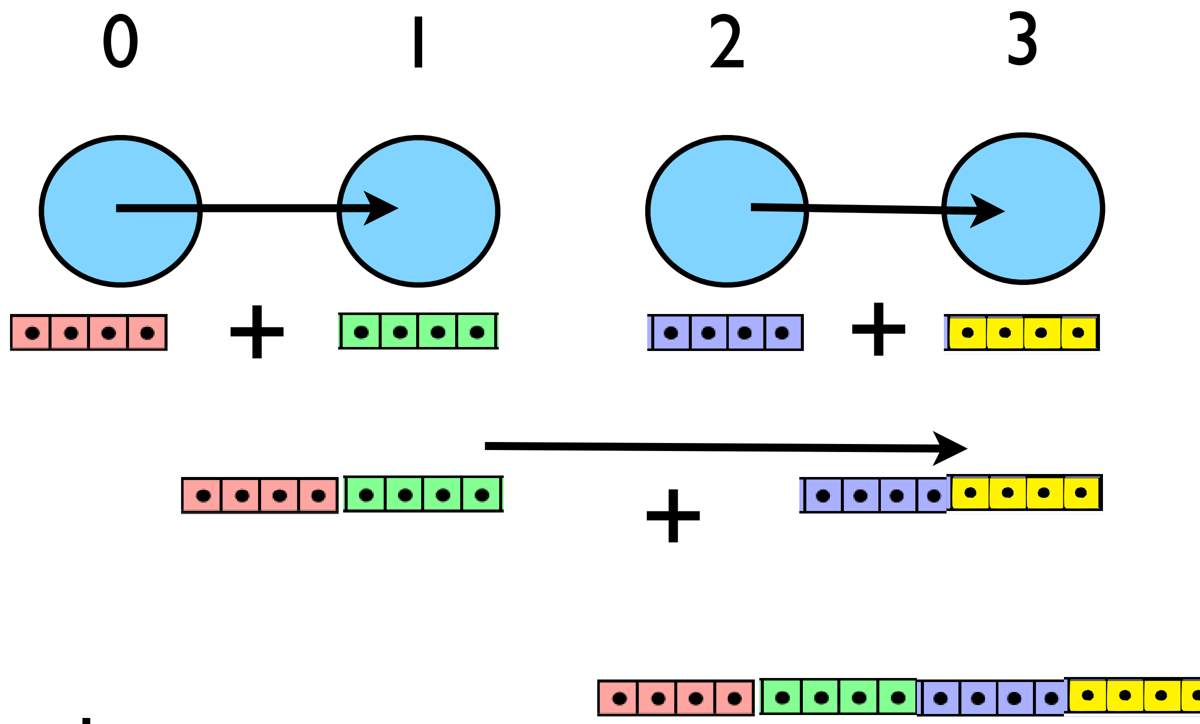


$$T_{\text{comp}} = c_{\text{grav}} \frac{N^2}{P} C_{\text{comp}}$$

$$T_{\text{comm}} \sim c_{\text{particle}} 2N \frac{\log_2 P}{P} C_{\text{comm}}$$

$$\frac{T_{\text{comm}}}{T_{\text{comp}}} \approx \frac{c_{\text{particle}}}{c_{\text{grav}}} \frac{2}{N} \log_2(P) \frac{C_{\text{comm}}}{C_{\text{comp}}}$$

Another collective operation



Stuff you're sending

How Much

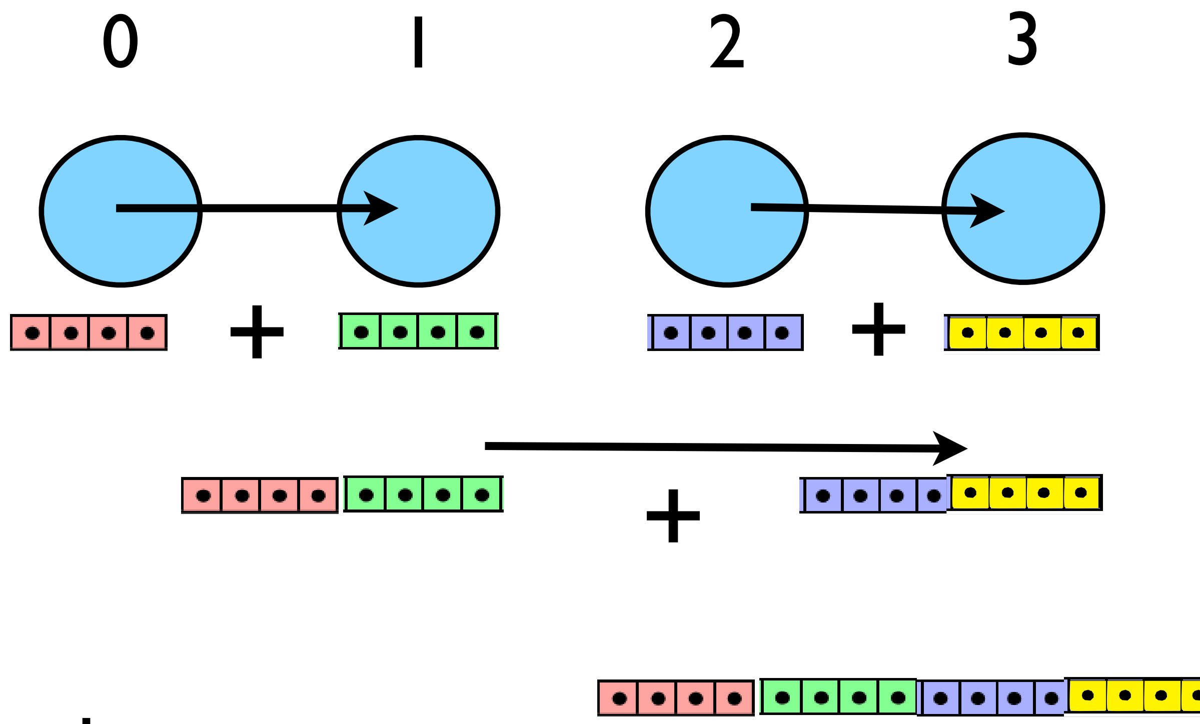
What Type

```
int MPI_Gather (void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm);
```

Place you're receiving

Who's getting all the data

Another collective operation



Stuff you're sending

How Much

What Type

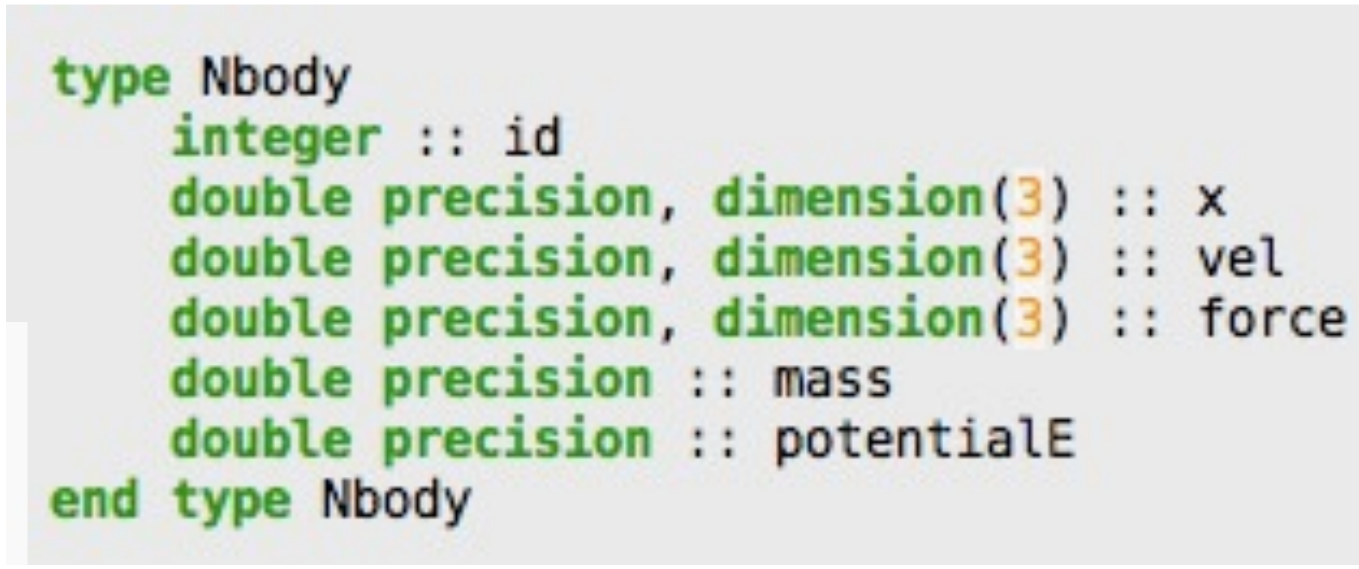
```
MPI_GATHER (sendbuf, INTEGER sendcnt, INTEGER sendtype,
               recvbuf, INTEGER recvcount, INTEGER recvtype,
               INTEGER root, INTEGER comm, INTEGER ierr);
```

Place you're receiving

Who's getting all the data

But what data type should we use?

- Not just a multiple of a single data type
- Contiguous, vector, subarray types won't do it.



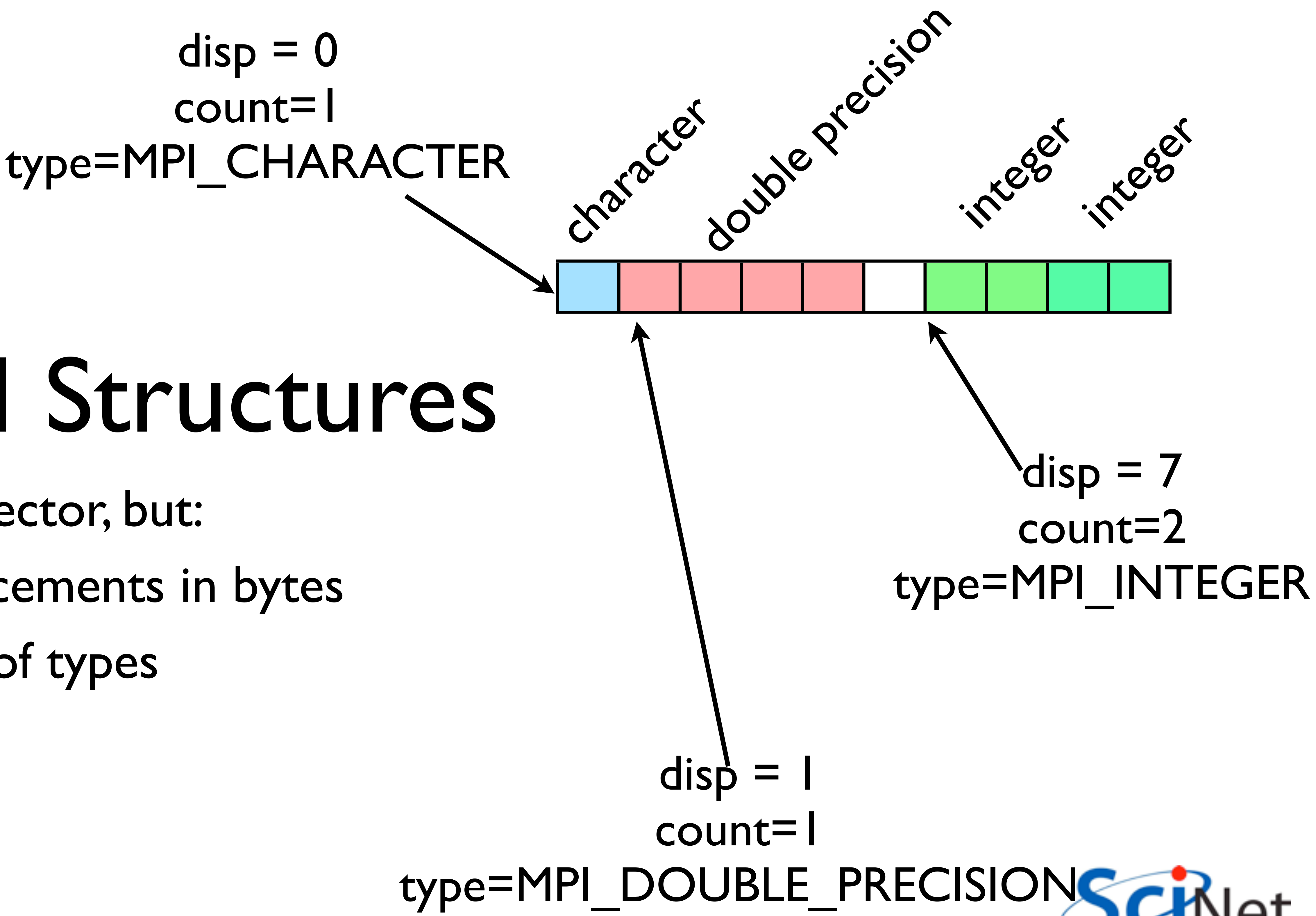
```
type Nbody
    integer :: id
    double precision, dimension(3) :: x
    double precision, dimension(3) :: vel
    double precision, dimension(3) :: force
    double precision :: mass
    double precision :: potentialE
end type Nbody
```

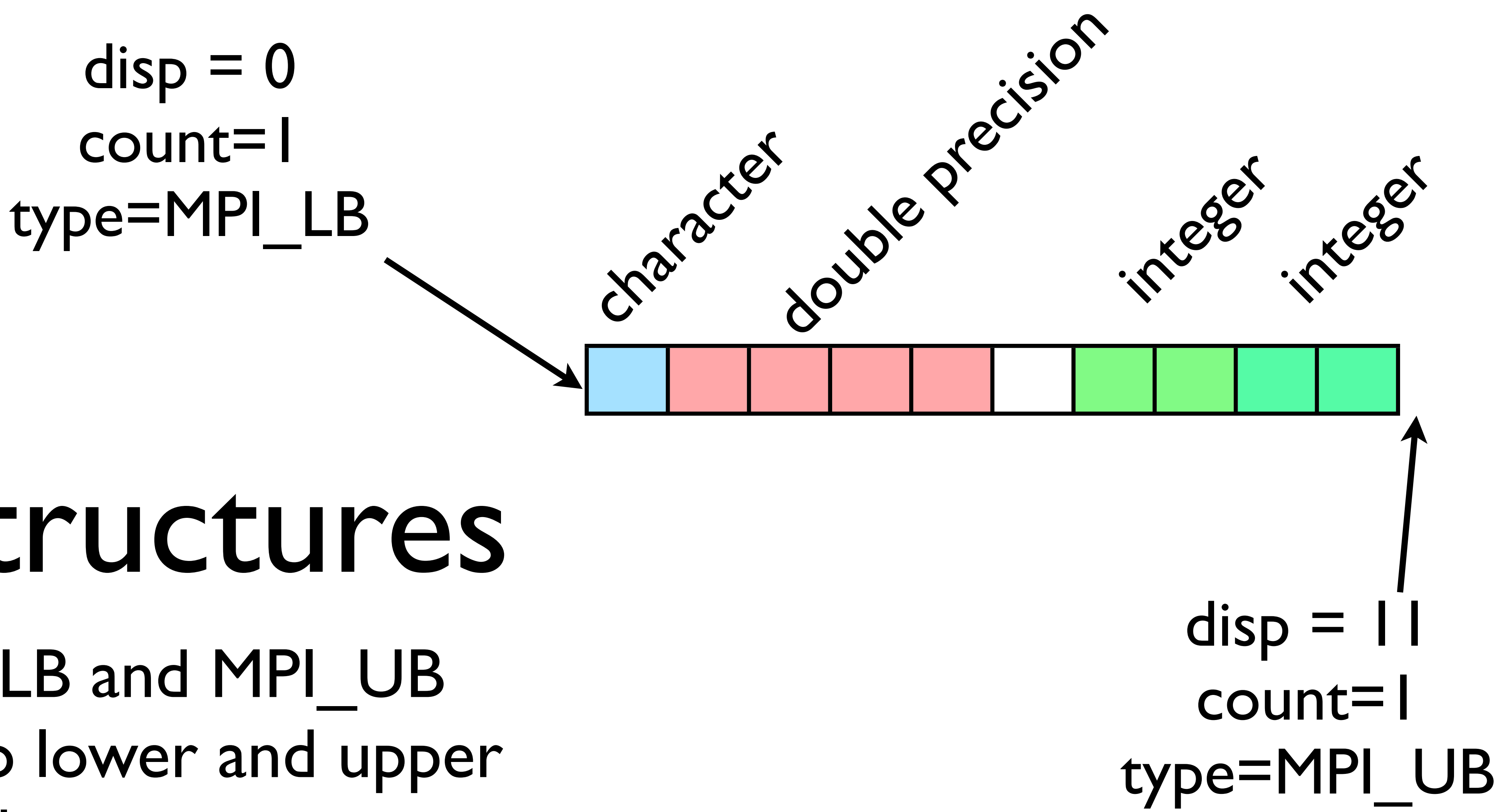
```
MPI_TYPE_CREATE_STRUCT(INTEGER COUNT, INTEGER ARRAY_OF_BLOCKLENGTHS(*),
            INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF DISPLACEMENTS(*),
            INTEGER ARRAY_OF_TYPES(*), INTEGER NEWTYPE, INTEGER IERROR)
```

```
int MPI_Type_create_struct(int count, int array_of_blocklengths[],
    MPI_Aint array_of_displacements[], MPI_Datatype array_of_types[],
    MPI_datatype *newtype);
```

MPI Structures

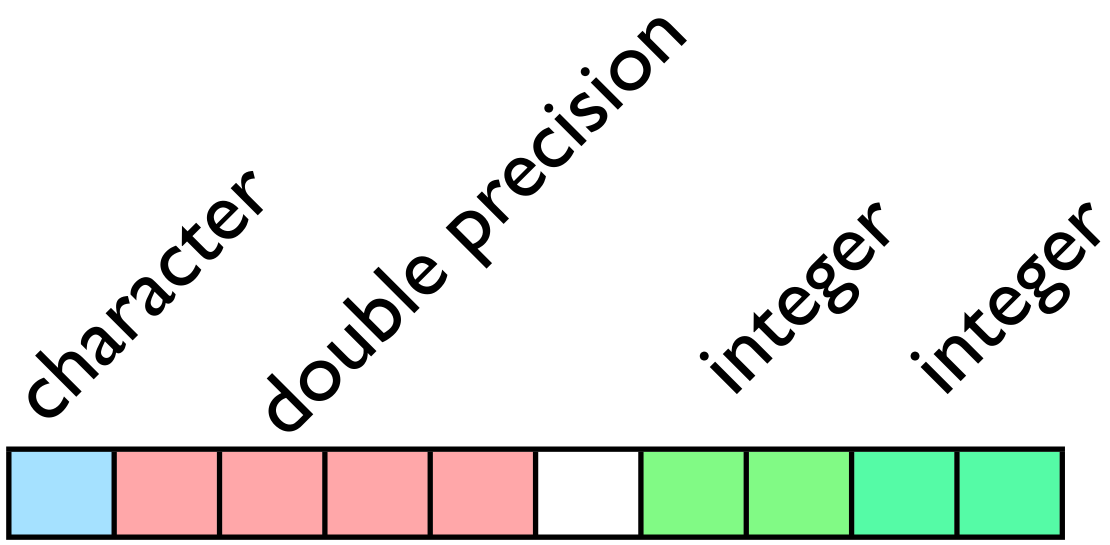
- Like vector, but:
- displacements in bytes
- array of types





MPI Structures

- Types MPI_LB and MPI_UB can point to lower and upper bounds of the structure, as well

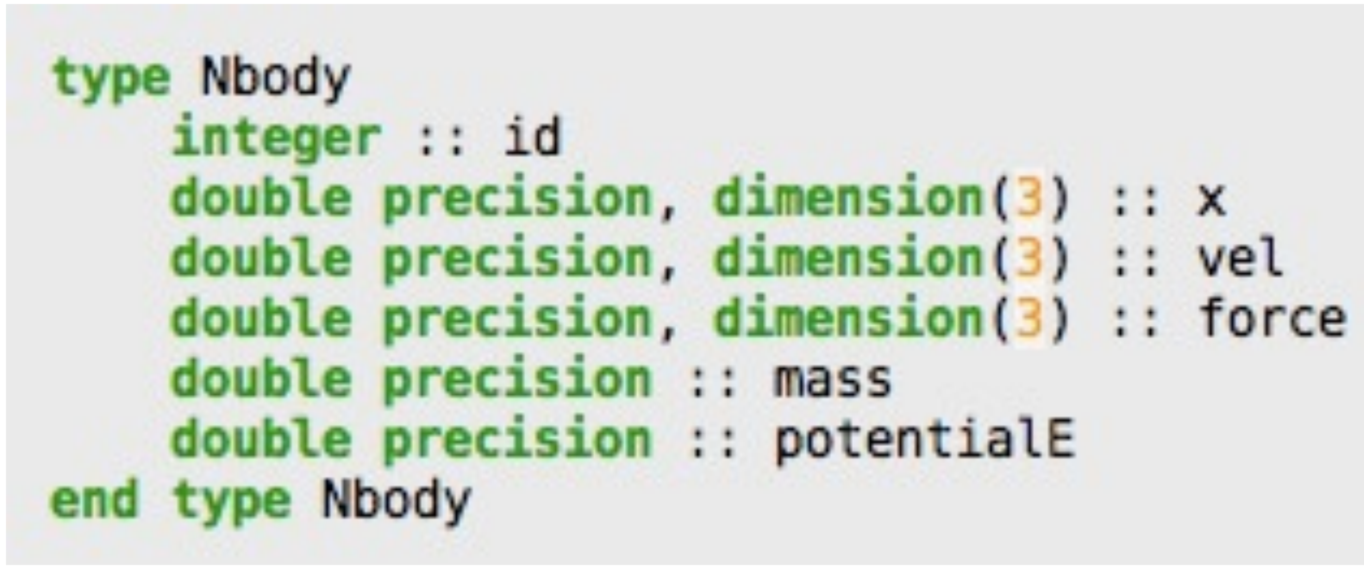


MPI Type Maps

- Complete description of this structure looks like:
  blocklens = (1,1,1,2,1)
  displacements = (0,0,1,6,10)
  types = (MPI_LB, MPI_CHARACTER, MPI_DOUBLE_PRECISION, MPI_INTEGER, MPI_UB)
- Note typemaps not unique; could write the integers out as two single integers with displacements 6, 8.

MPI Type Maps

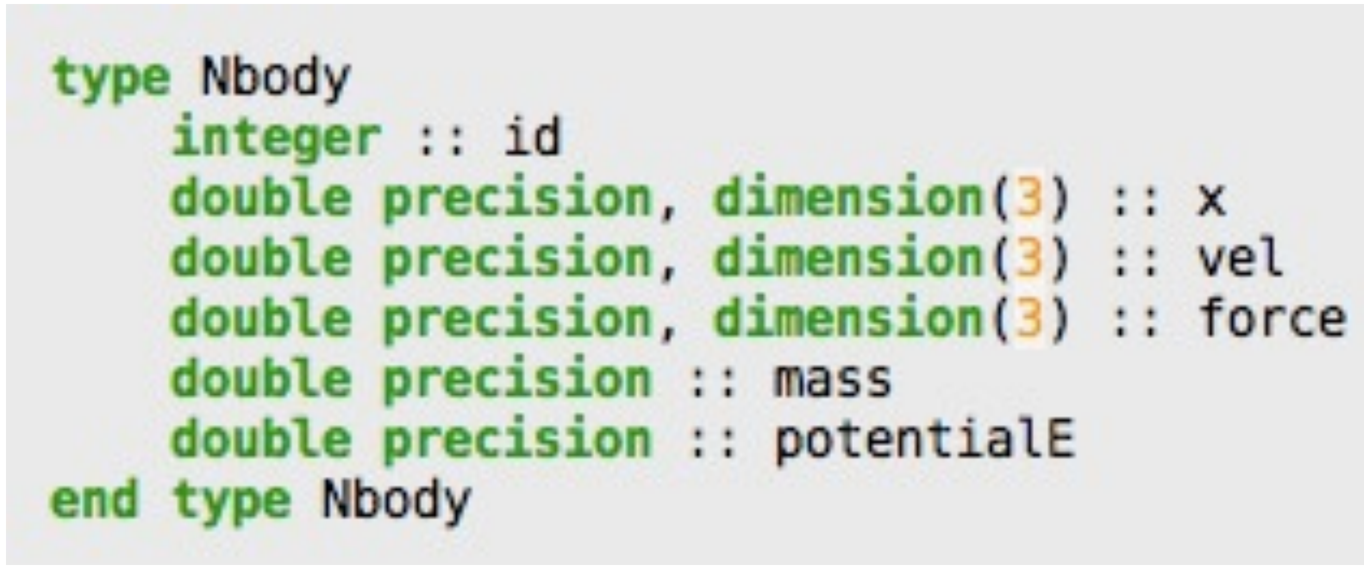
- What does type map look like for Nbody?



```fortran
type Nbody
    integer :: id
    double precision, dimension(3) :: x
    double precision, dimension(3) :: vel
    double precision, dimension(3) :: force
    double precision :: mass
    double precision :: potentialE
end type Nbody
```

MPI Type Maps

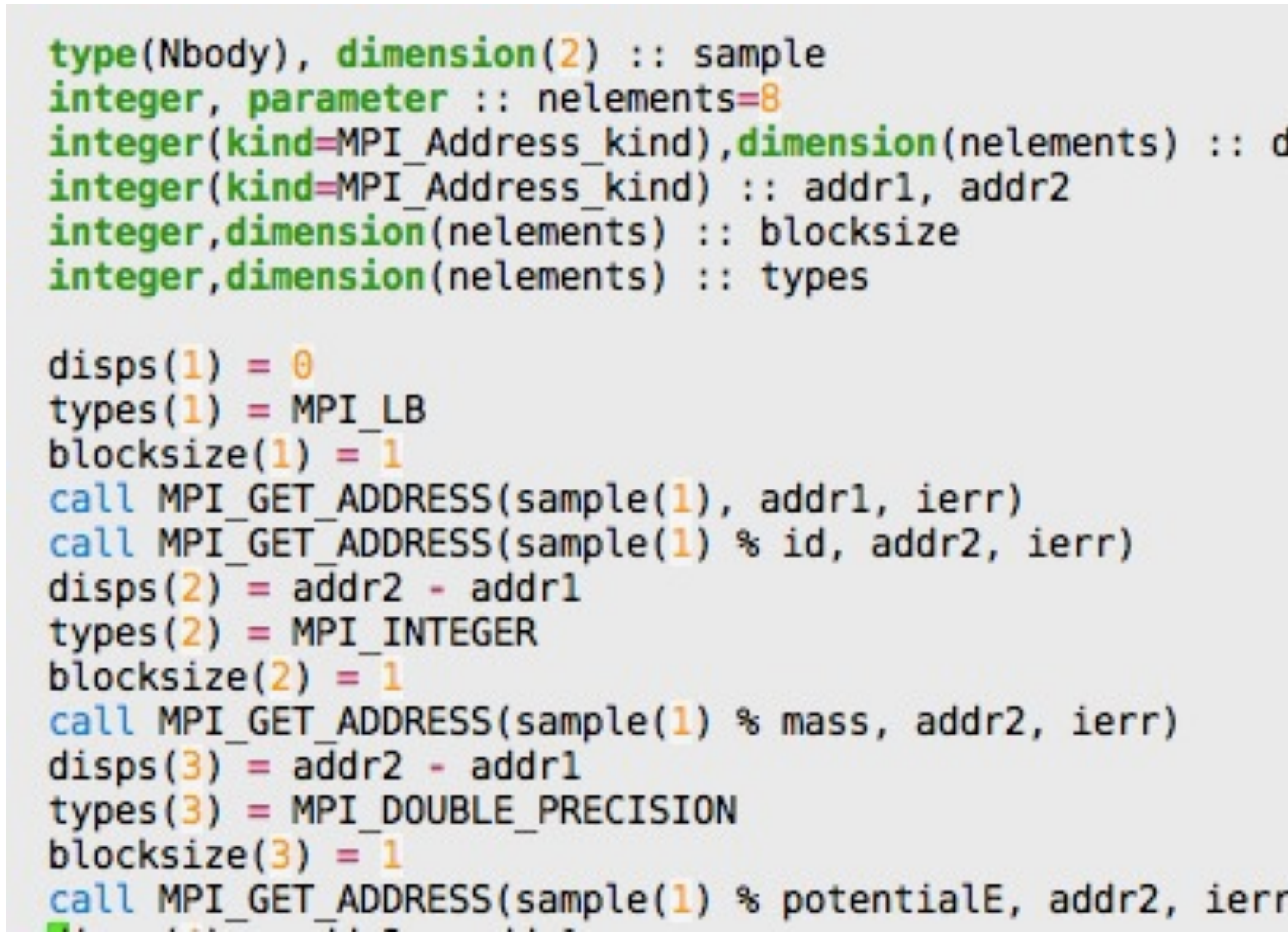
- What does type map look like for Nbody?
- How laid out in memory depends entirely on compiler, compiler options.
- alignment, padding...



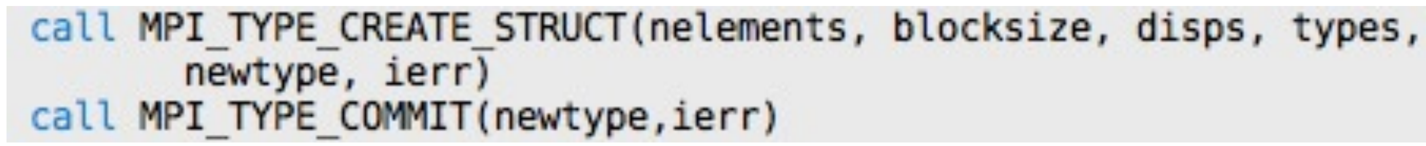
```
type Nbody
    integer :: id
    double precision, dimension(3) :: x
    double precision, dimension(3) :: vel
    double precision, dimension(3) :: force
    double precision :: mass
    double precision :: potentialE
end type Nbody
```

MPI Type Maps

- Use MPI_GET_ADDRESS to find addresses of different objects, and subtract the two to get displacements
- Build structure piece by piece.

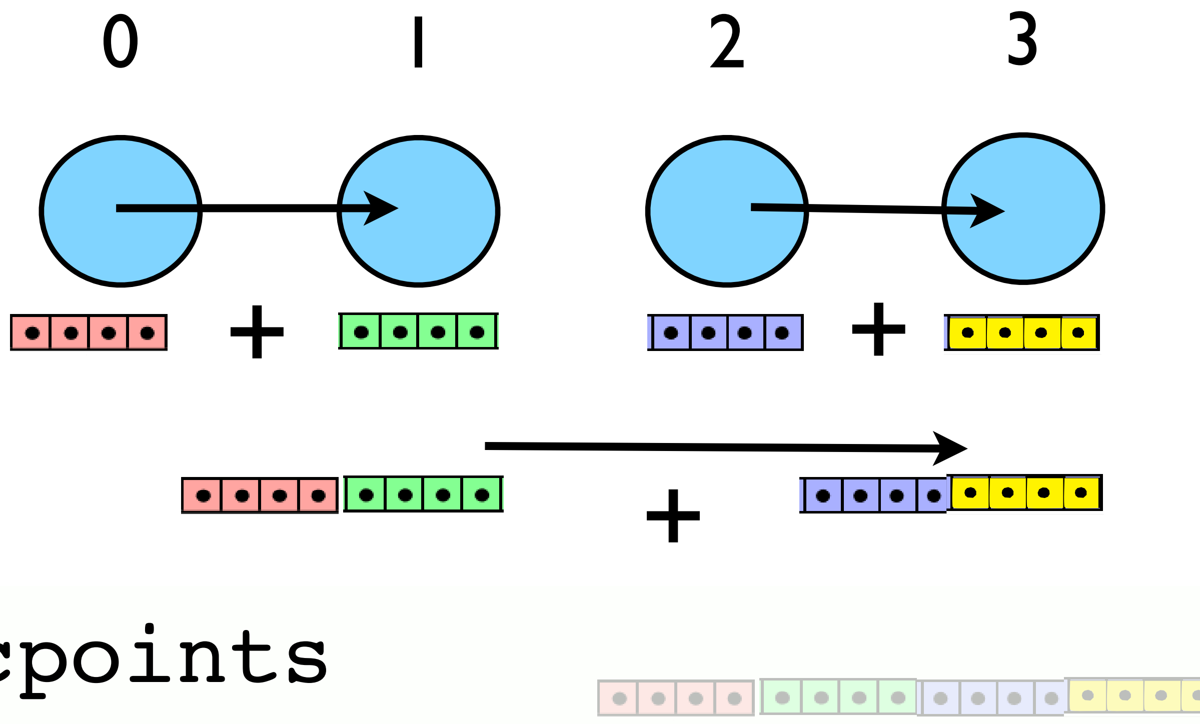


```
type(Nbody), dimension(2) :: sample
integer, parameter :: nelements=8
integer(kind=MPI_Address_kind),dimension(nelements) :: d
integer(kind=MPI_Address_kind) :: addr1, addr2
integer,dimension(nelements) :: blocksize
integer,dimension(nelements) :: types

disps(1) = 0
types(1) = MPI_LB
blocksize(1) = 1
call MPI_GET_ADDRESS(sample(1), addr1, ierr)
call MPI_GET_ADDRESS(sample(1) % id, addr2, ierr)
disps(2) = addr2 - addr1
types(2) = MPI_INTEGER
blocksize(2) = 1
call MPI_GET_ADDRESS(sample(1) % mass, addr2, ierr)
disps(3) = addr2 - addr1
types(3) = MPI_DOUBLE_PRECISION
blocksize(3) = 1
call MPI_GET_ADDRESS(sample(1) % potentialE, addr2, ierr
```



```
call MPI_TYPE_CREATE_STRUCT(nelements, blocksize, disps, types,
        newtype, ierr)
call MPI_TYPE_COMMIT(newtype,ierr)
```

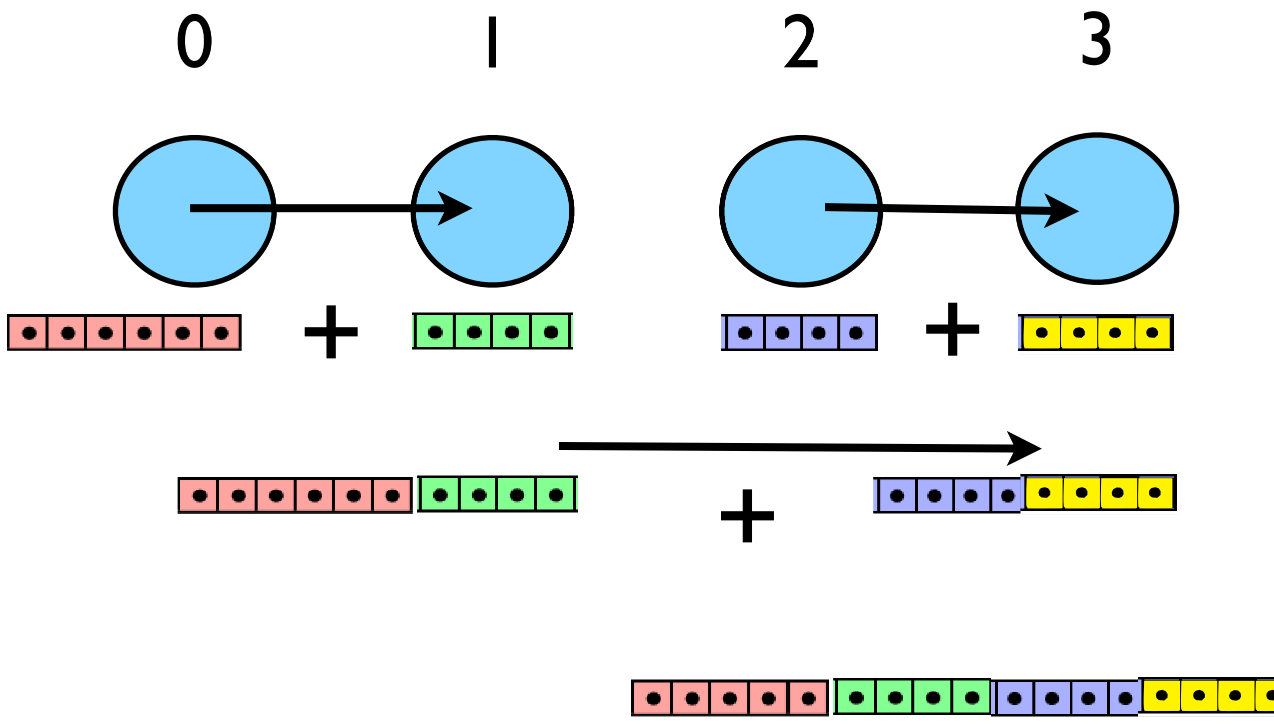
Another collective operation



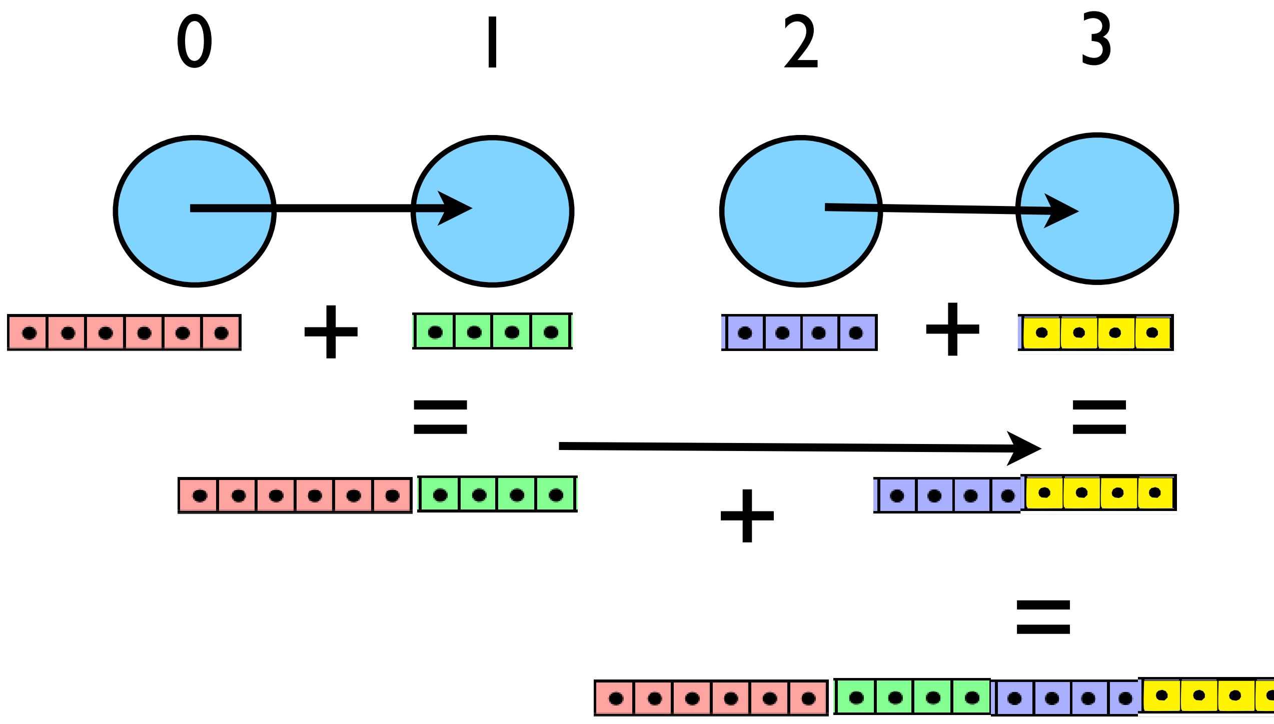
```
integer :: startp, endp, locpoints
integer :: ptype
type(Nbody), dimension(N) :: pdata

call MPI_Allgather(pdata(startp), locpoints, ptype,
             pdata, locpoints, ptype,
             MPI_COMM_WORLD, ierr)
```

What if not same # of particles?

- When everyone has same # of particles, easy to figure out where one processor's piece goes in the global array
- Otherwise, need to know how many each has and where their chunk should go in the global array



What if not same # of particles?

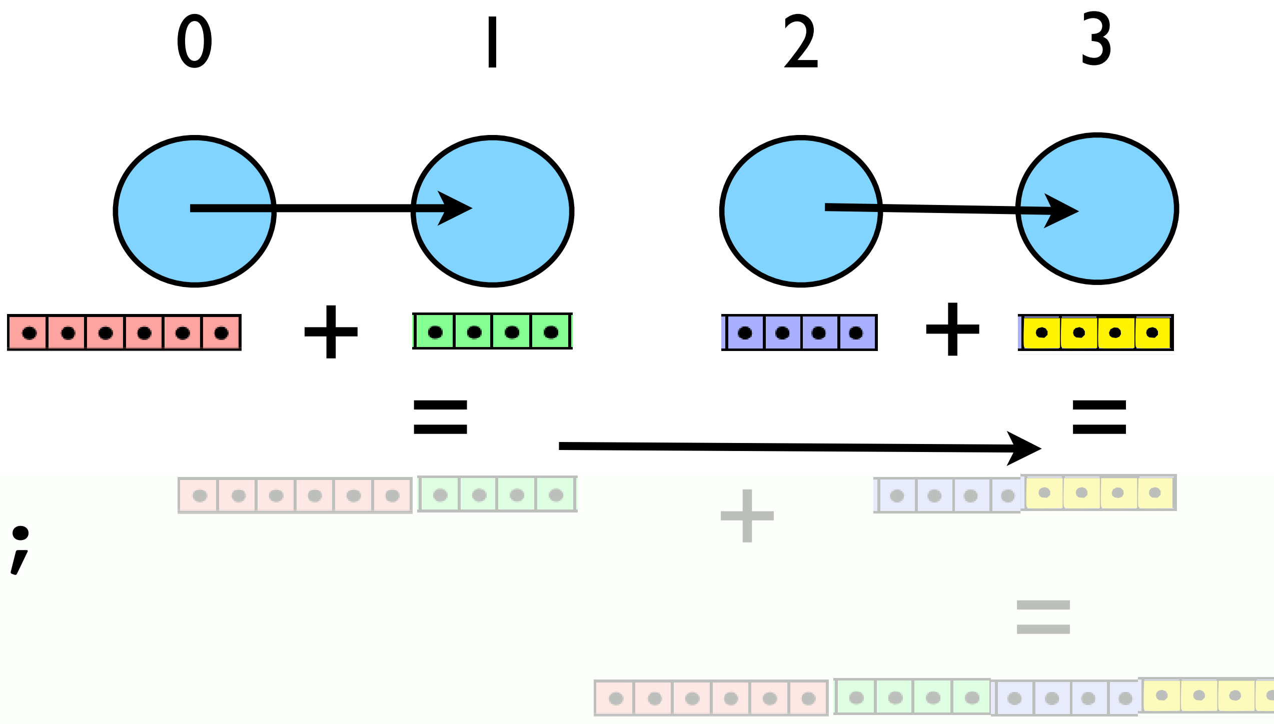


```
int MPI_Allgatherv ( void *sendbuf, int sendcount, MPI_Datatype sendtype,
                     void *recvbuf, int *recvcounts, int *displs,
                    MPI_Datatype recvtype, MPI_Comm comm )
```

Array of counts; eg {6,4,4,4}

Where they should go; eg {0,6,10,14}

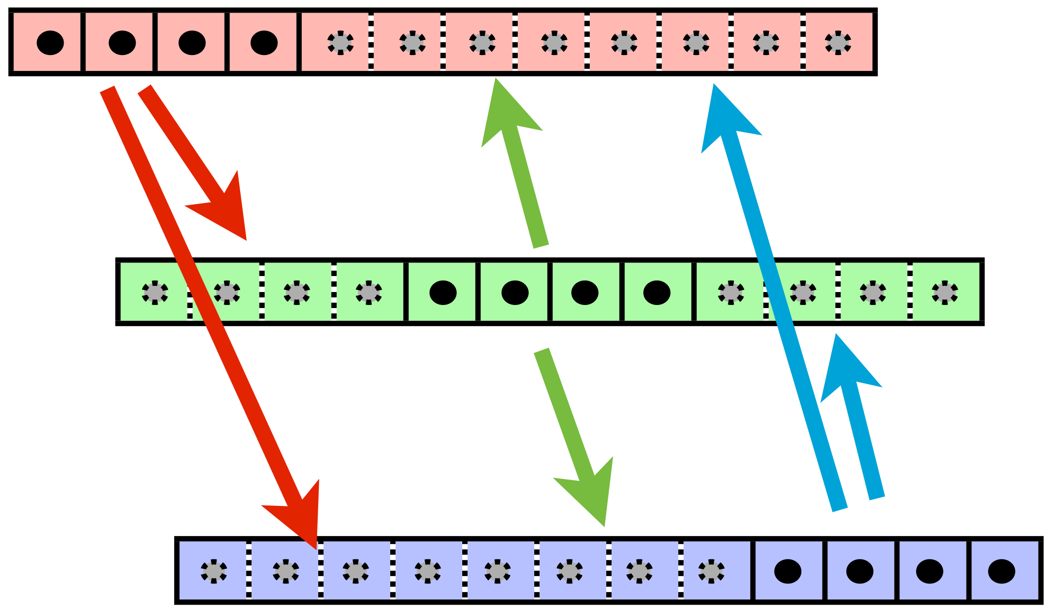
How would we get this data? Allgather!



```
int counts[size], disp[size];
int mystart=..., mynump=...;

MPI_Allgather(&mynump, 1, MPI_INT,
              counts, 1, MPI_INT, MPI_COMM_WORLD);
disp[i]=0;
for (i=1;i<size;i++) disp[i]=disp[i-1]+counts[i];

MPI_Allgatherv(&(data[mystart]), mynump, MPI_Particle,
             data, counts, disp, MPI_Particle,
               MPI_COMM_WORLD);
```

Other stuff about the nbody code

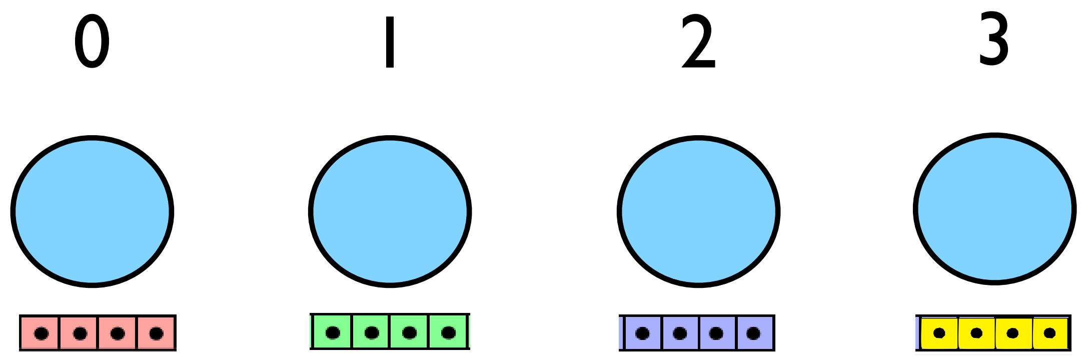
- At least plotting remains easy.
- Generally n-body codes keep track of things like global energy as a diagnostic
- We have a local energy we calculate on our particles;
- Should communicate that to sum up over all processors.
- Let's do this together



edit nbody-allgather.f90

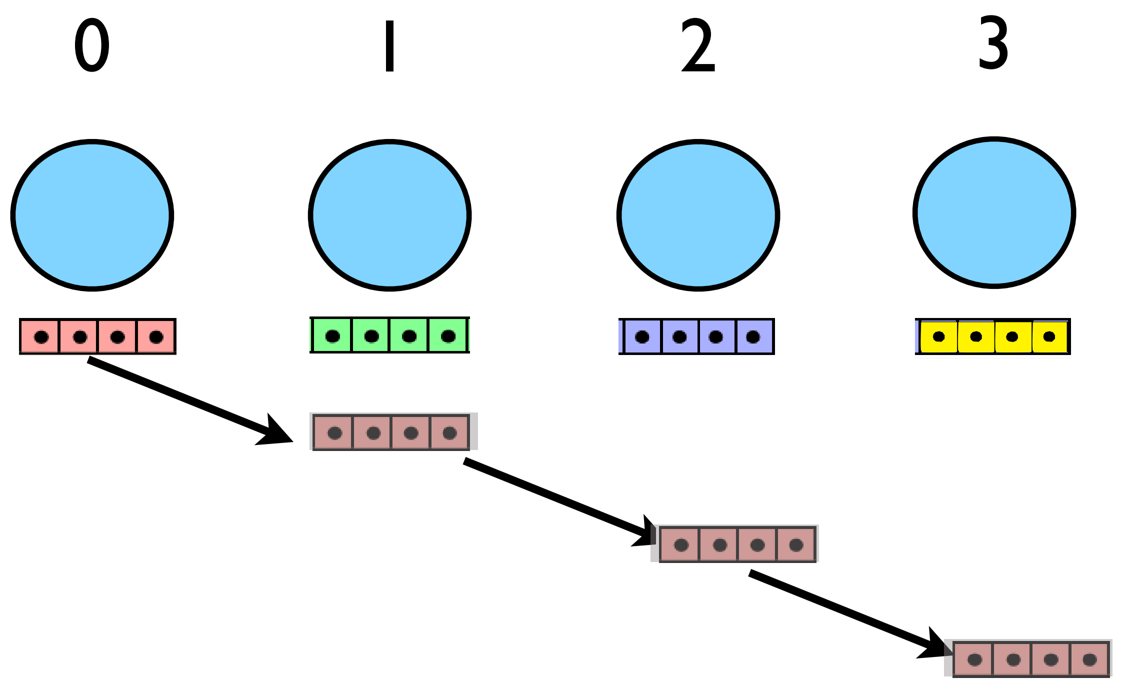
Problem (I) remains -- memory

- How do we avoid this?
- For direct summation, we need to be able to see all particles;
- But not necessarily at once.



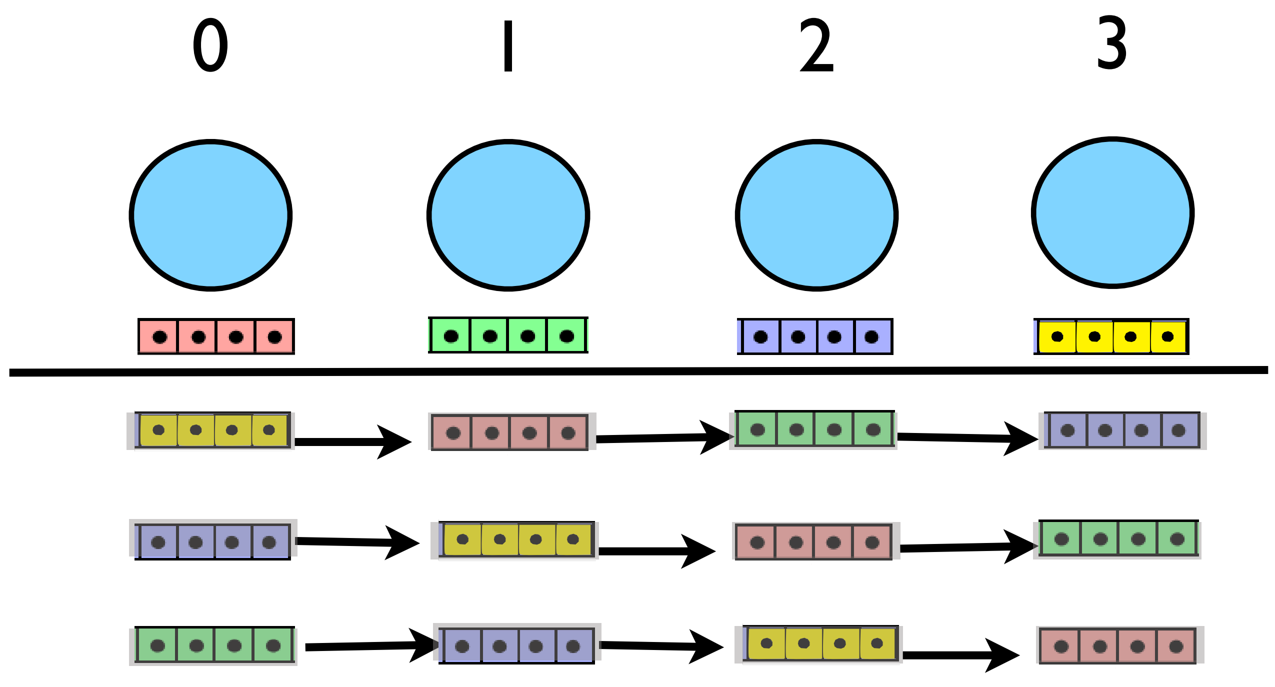
Pipeline

- 0 sends chunk of its particles to 1, which computes on it, then 2, then 3
- Then 1 does the same thing, etc.
- Size of chunk: tradeoff - memory usage vs. number of messages
- Let's just assume all particles go at once, and all have same # of particles (bookkeeping)

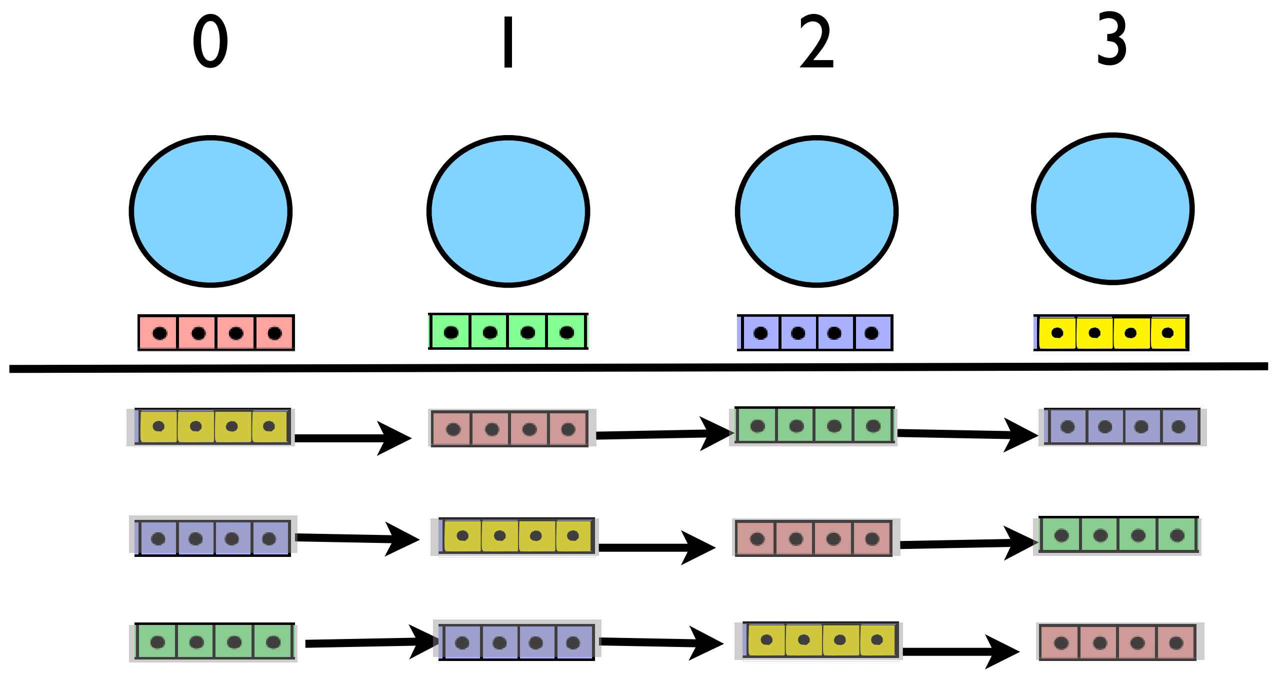


Pipeline

- No need to wait for 0s chunk to be done!
- Everyone sends their chunk forward, and keeps getting passed along.
- Compute local forces first, then start pipeline, and foreach (P-1) chunks compute the forces on your particles by theirs.



Pipeline



- Work unchanged

$$T_{\rm comp} \quad = \quad c_{\rm grav} \frac{N^2}{P} C_{\rm comp}$$

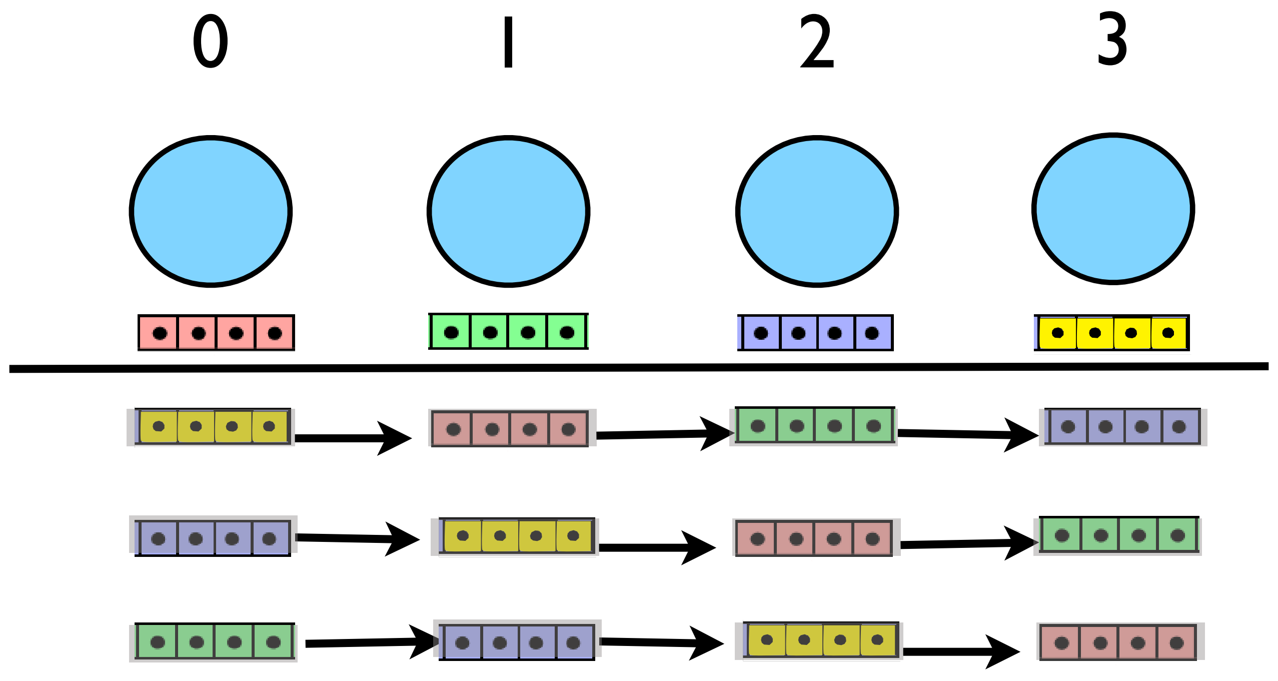
- Communication - each process sends (P-1) messages of length (N/P)

$$T_{\rm comm} \quad = \quad c_{\rm particle} (P-1) \frac{N}{P} C_{\rm comm} \quad \rightarrow c_{\rm particle} N C_{\rm comm}$$

$$\frac{T_{\rm comm}}{T_{\rm comp}} \quad \approx \quad \frac{c_{\rm particle}}{c_{\rm grav}} \frac{1}{N} P \frac{C_{\rm comm}}{C_{\rm comp}}$$

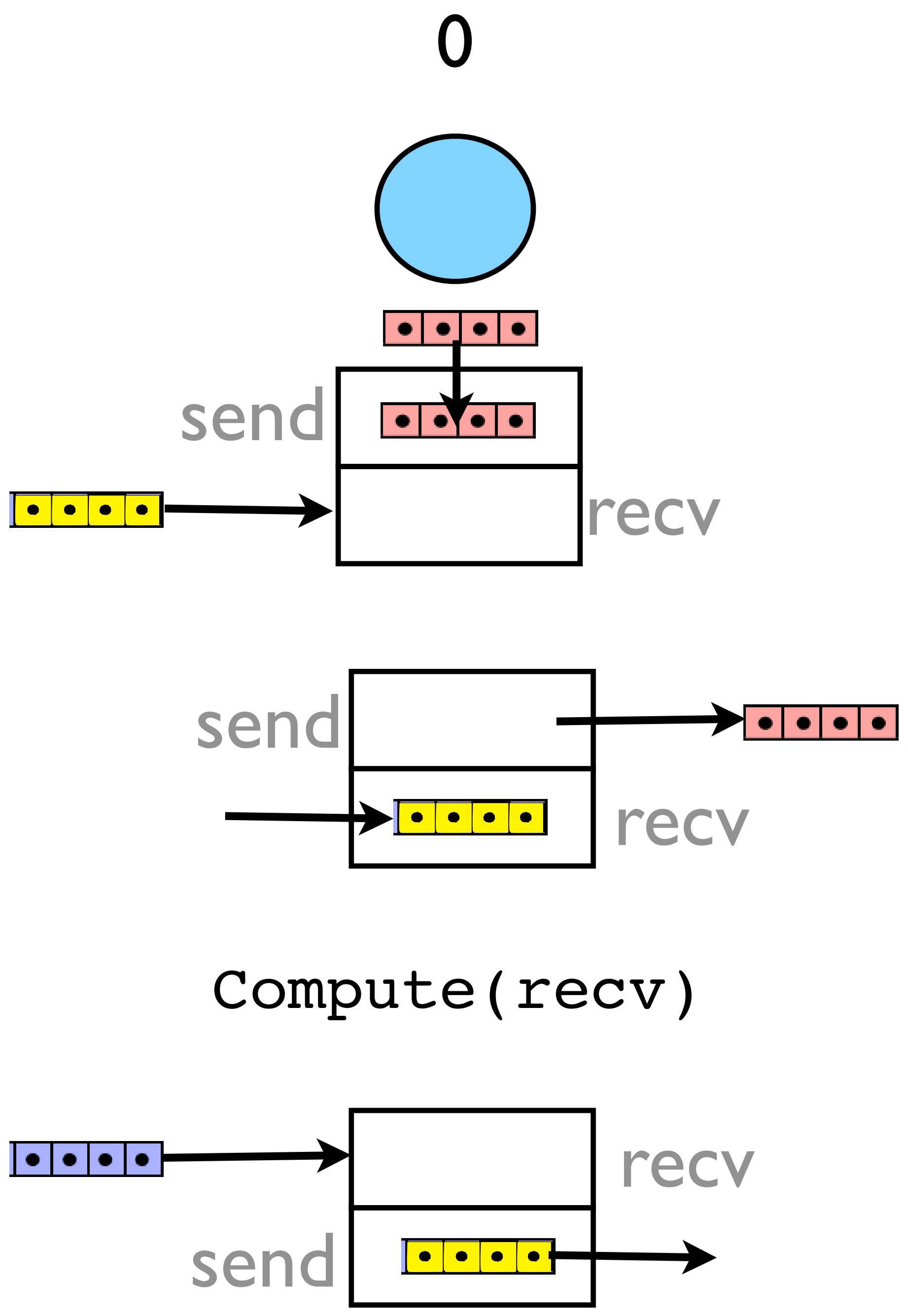
Pipeline

- Back to the first approach.
- *But* can do much bigger problems
- If we're filling memory, then N ~ P, and T_{comm}/T_{comp} is constant (yay!)
- With previous approach, maximum problem size is fixed by one processor's memory.



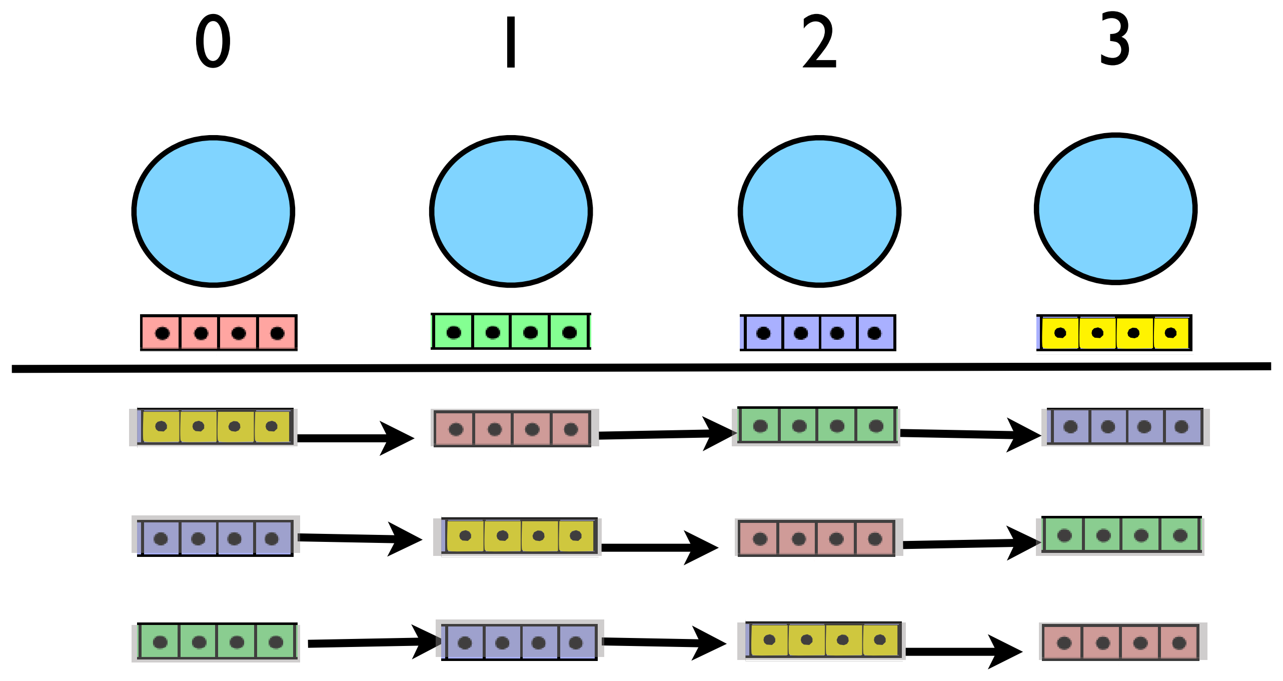
Pipeline

- Sending the messages: like one direction of the guardcell fills in the diffusion eqn; everyone sendrecv's.
- Periodic or else 0 would never see anyone elses particles!
- Copy your data into a buffer; send it, receive into another one.
- Compute on received data
- Swap send/recv and continue.



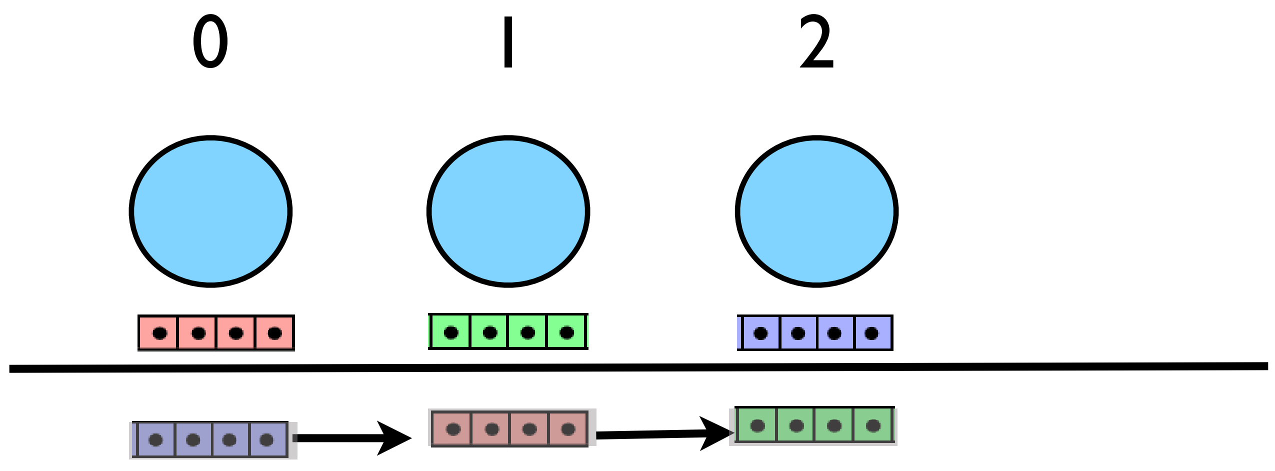
Pipeline

- Good: can do bigger problems!
- Bad: High communication costs, not fixable
- Bad x 2: still doing double work.



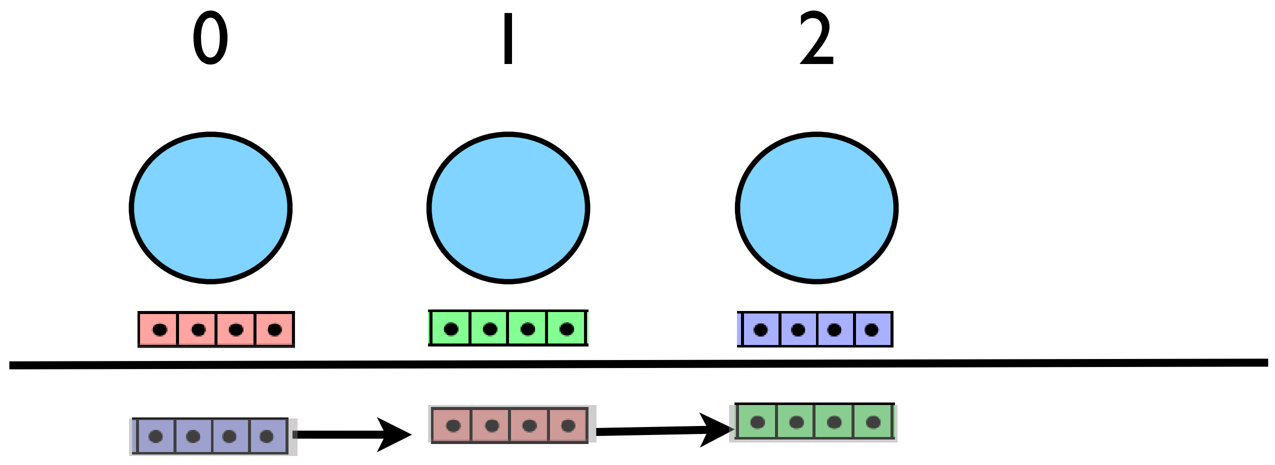
Pipeline

- Double work might be fixable
- We are sending whole particle structure when nodes only need x[NDIMS], mass.
- Option 1: we could only send chunk half way (for odd # procs); then every particle has seen every other
- If we update forces in both, then will have computed all non-local forces...)



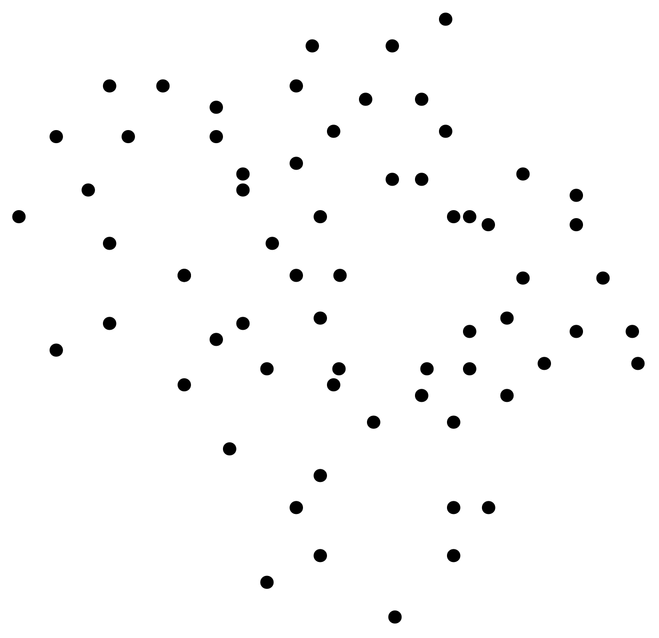
Pipeline

- Option 2: we could proceed as before, but only send the essential information
- Cut down size of message by a factor of 4/11
- Which is better?



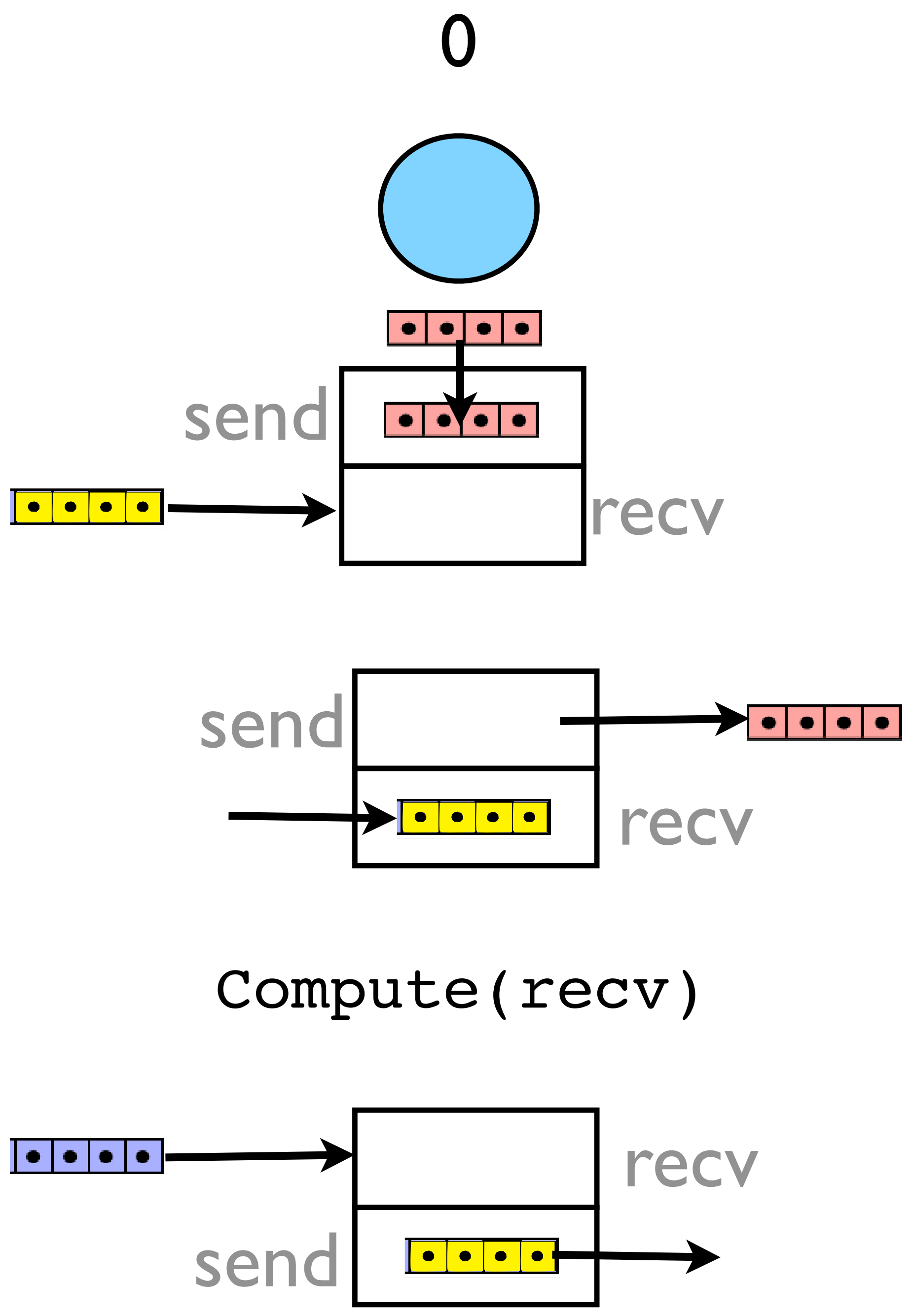
Displaying Data

- Now that no processor owns all of the data, can't make plots any more
- But the plot is small; it's a projection onto a 2d grid of the 3d data set.
- In general it's only data-sized arrays which are 'big'
- Can make it as before and Allreduce it



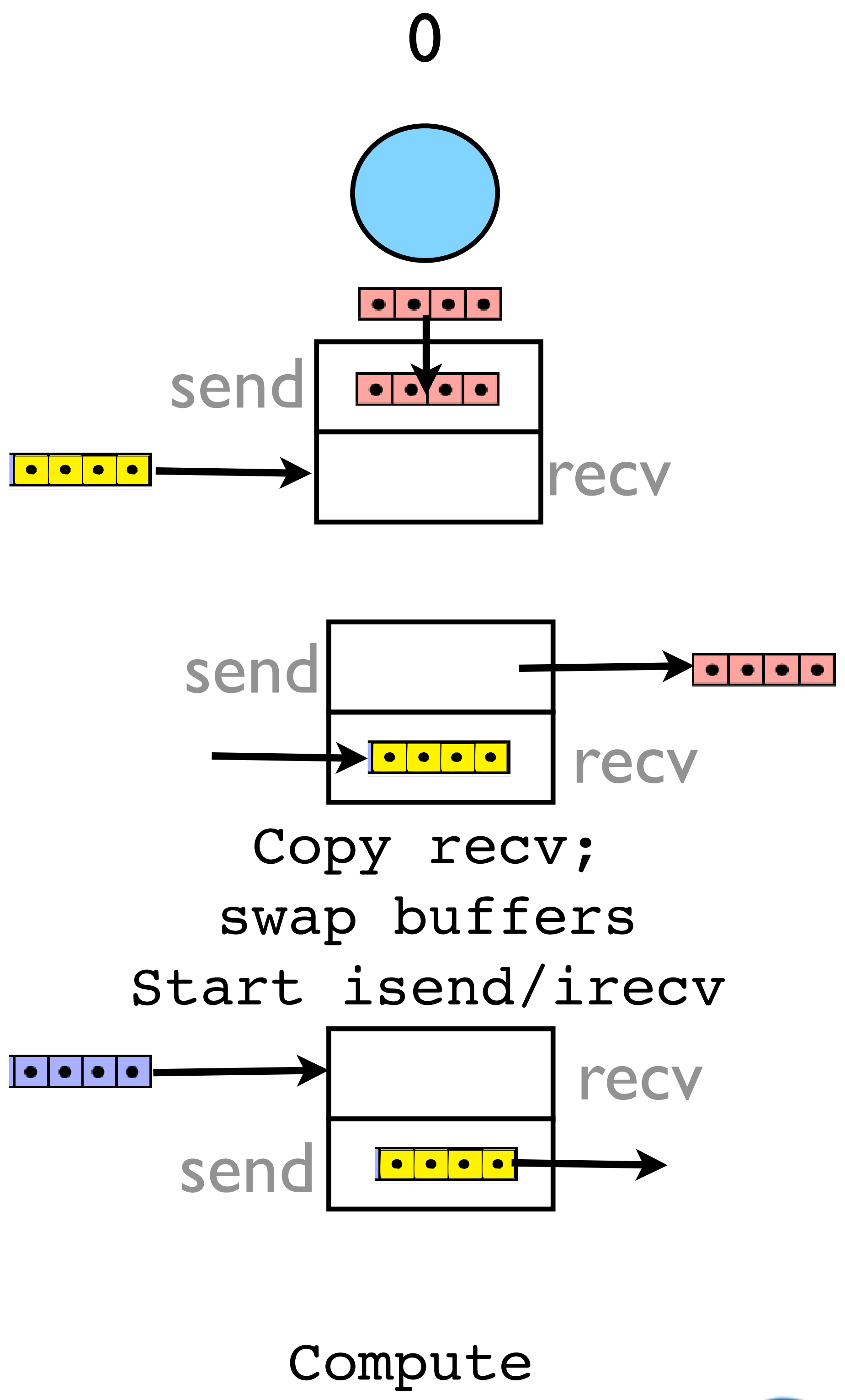
Overlapping Communication & Computation

- If only updating local forces, aren't changing the data in the pipeline at all.
- What we receive is what we send.
- Could issue send right away, but need to compute...



Overlapping Communication & Computation

- Now the communications will happen while we are computing
- Significant time savings! (~30% with 4 process)



Hands on

- Implement simplest pipeline (blocking)
- Try just doing one timestep, but calculating forces one block at a time
- Then sending blocks around
- Then non-blocking/double buffering