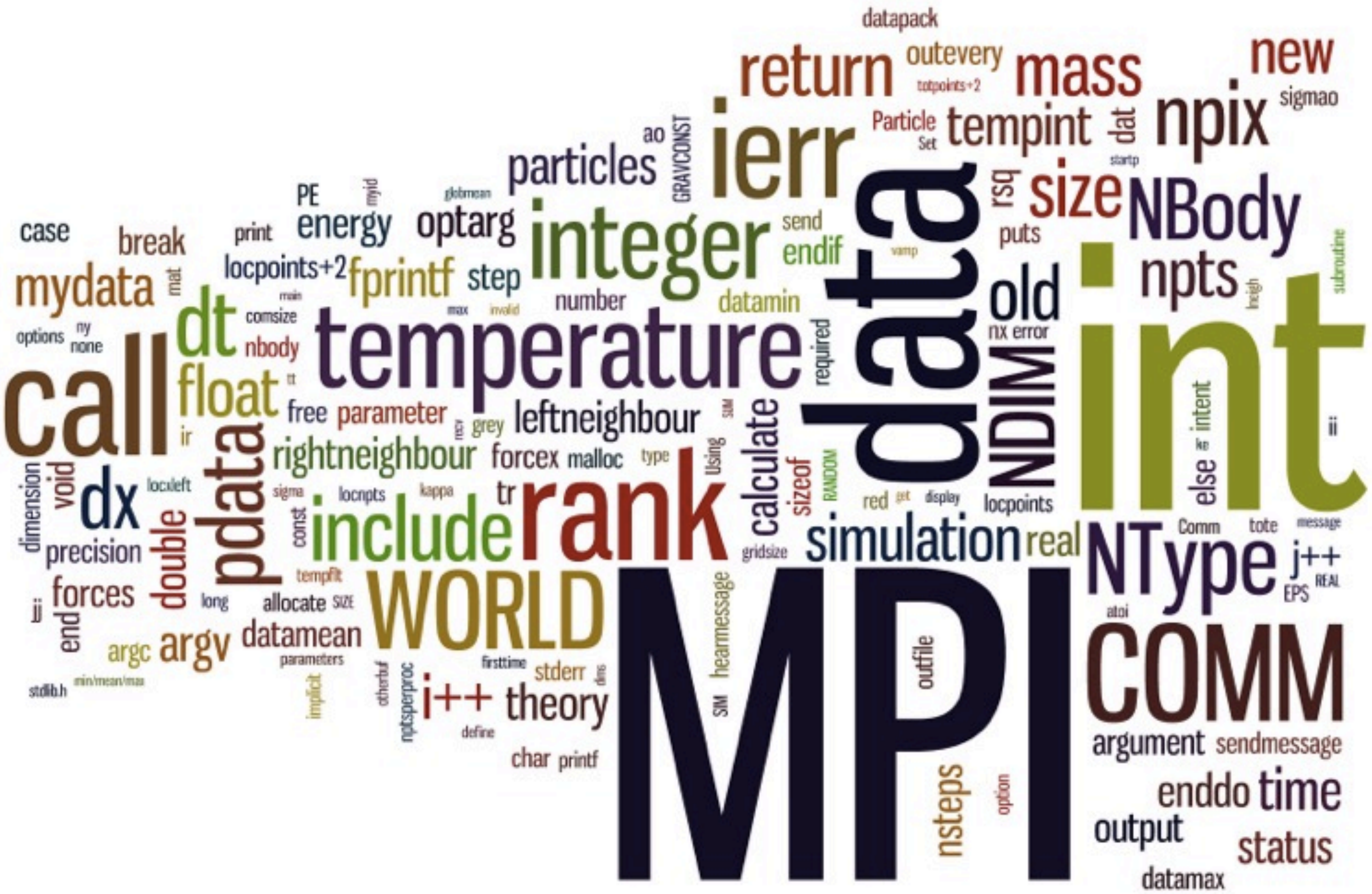
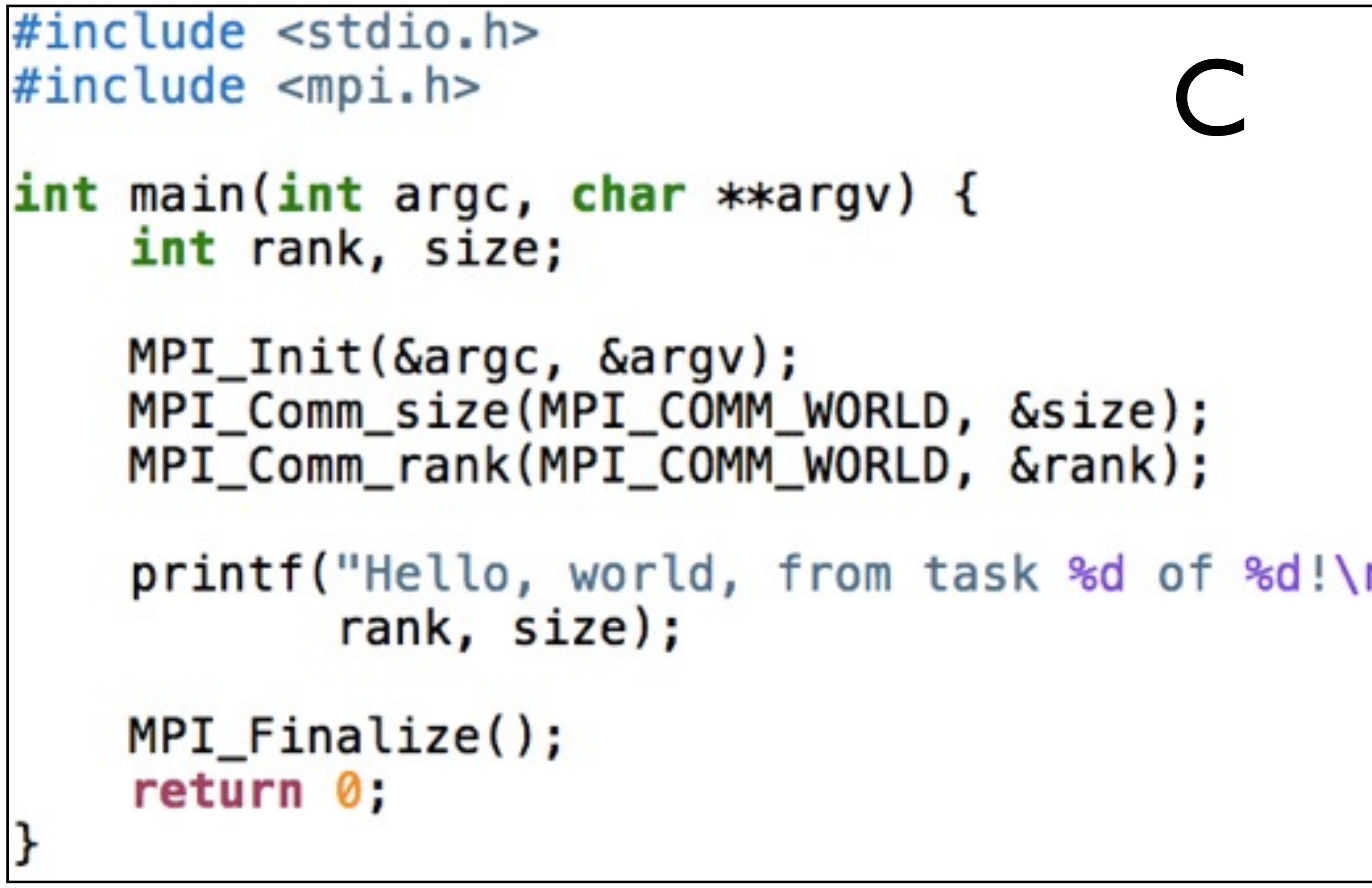
An introduction to MPI



(word cloud of all the MPI hydro code written for this course: http://www.wordle.net)

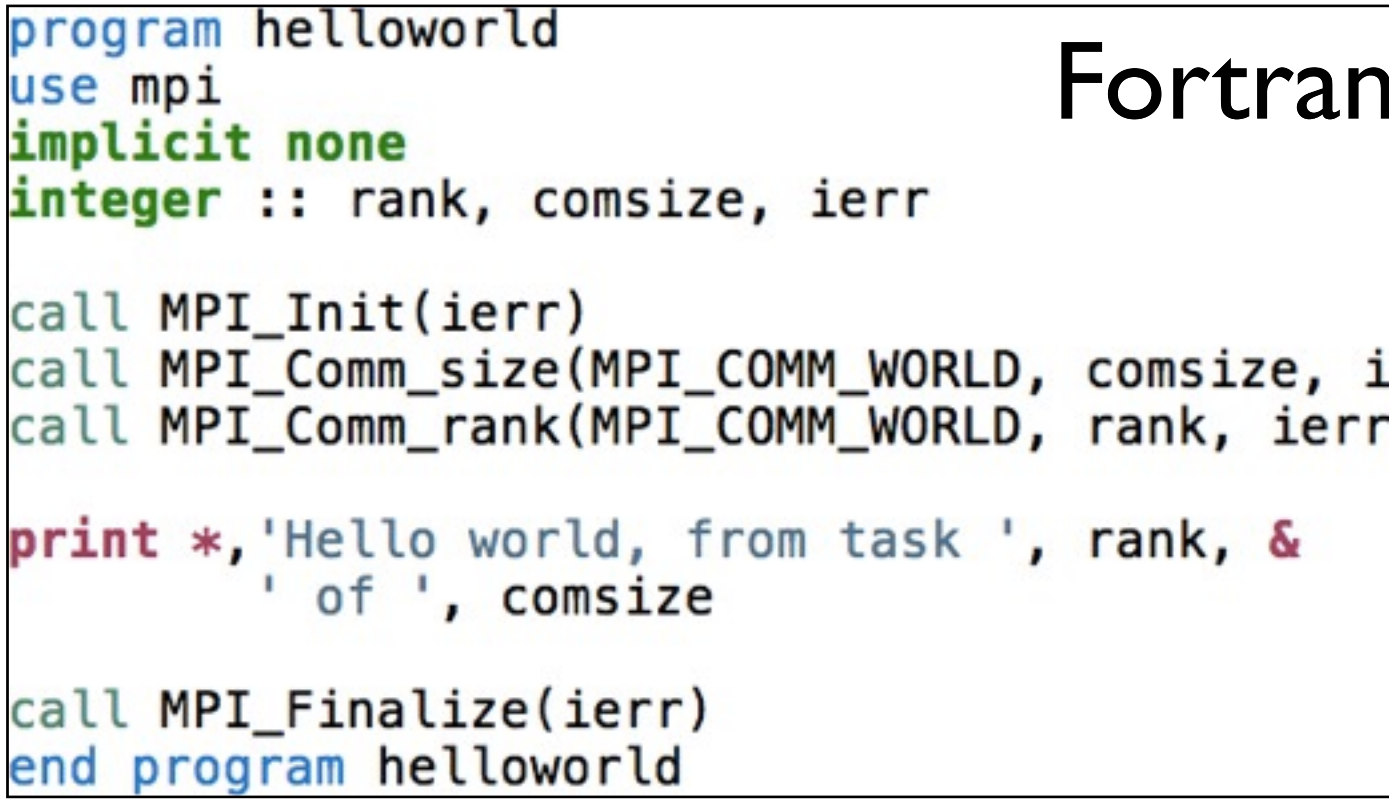
MPI is a **Library** for Message-Passing

- 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;
}
```

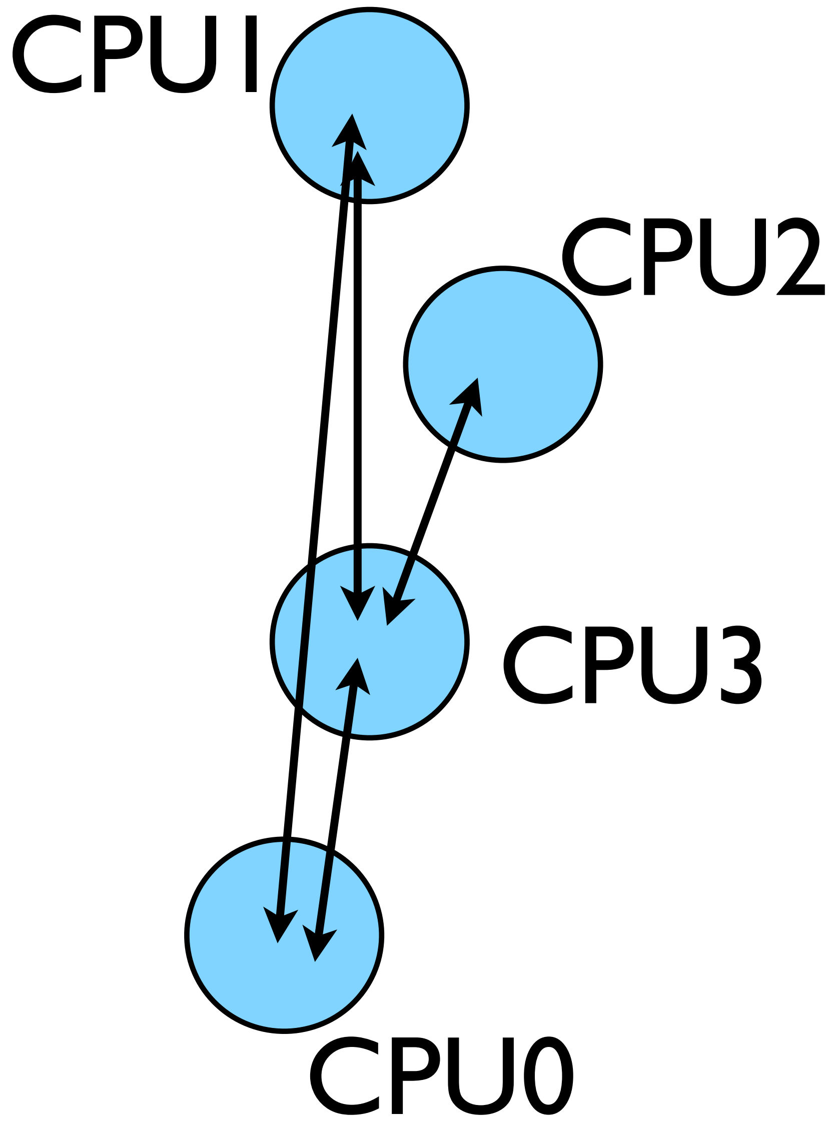


Fortran

```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
```

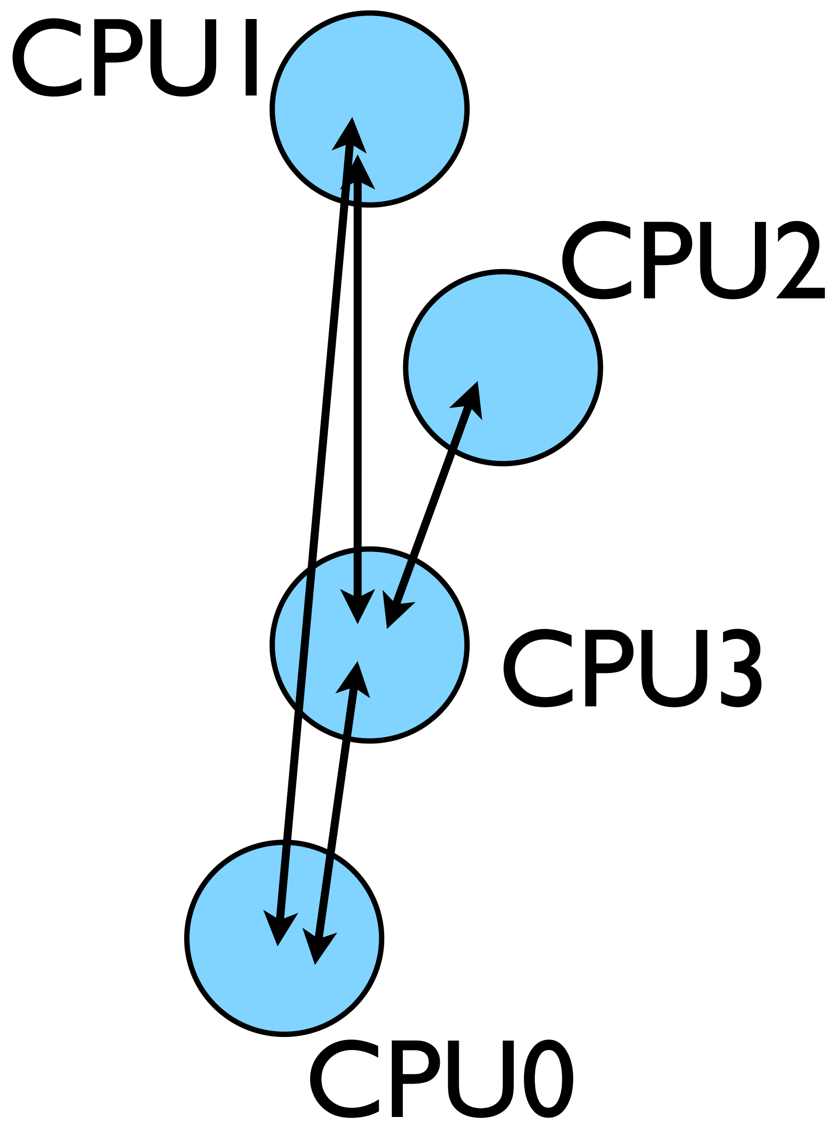
MPI is a Library for **Message-Passing**

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



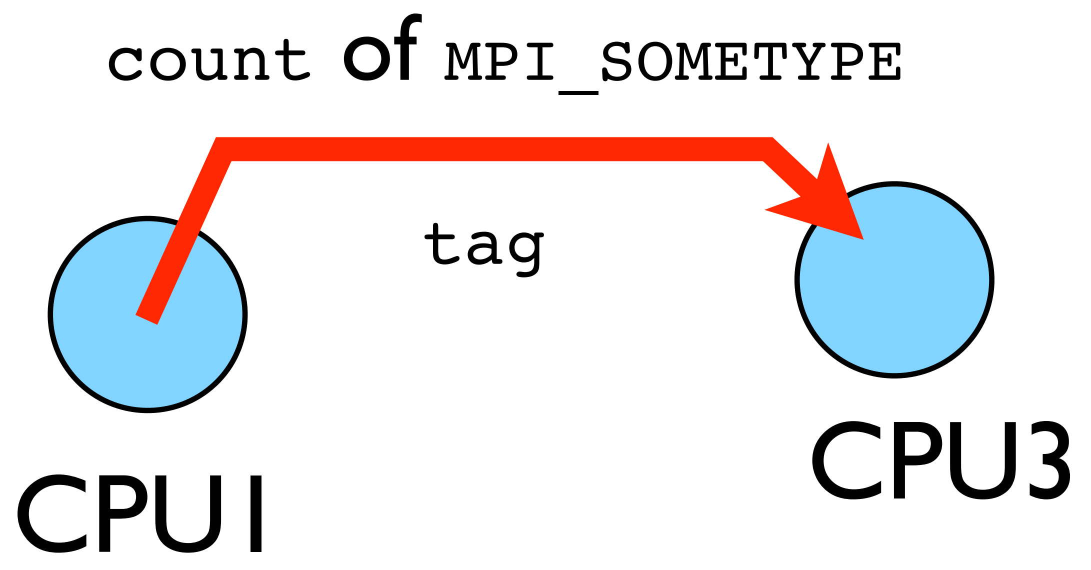
MPI is a Library for **Message-Passing**

- 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



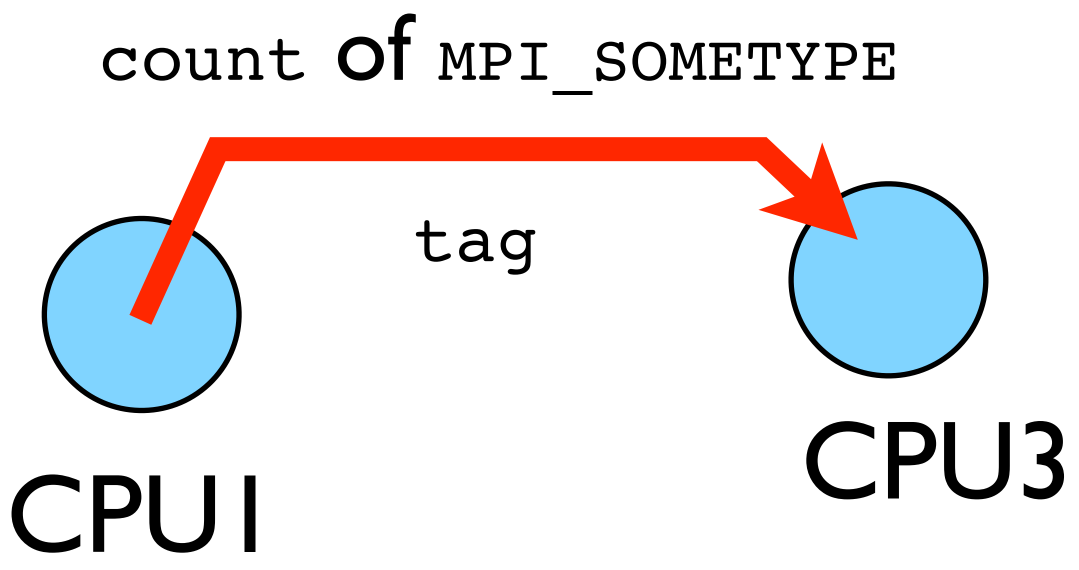
Messages

- 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.



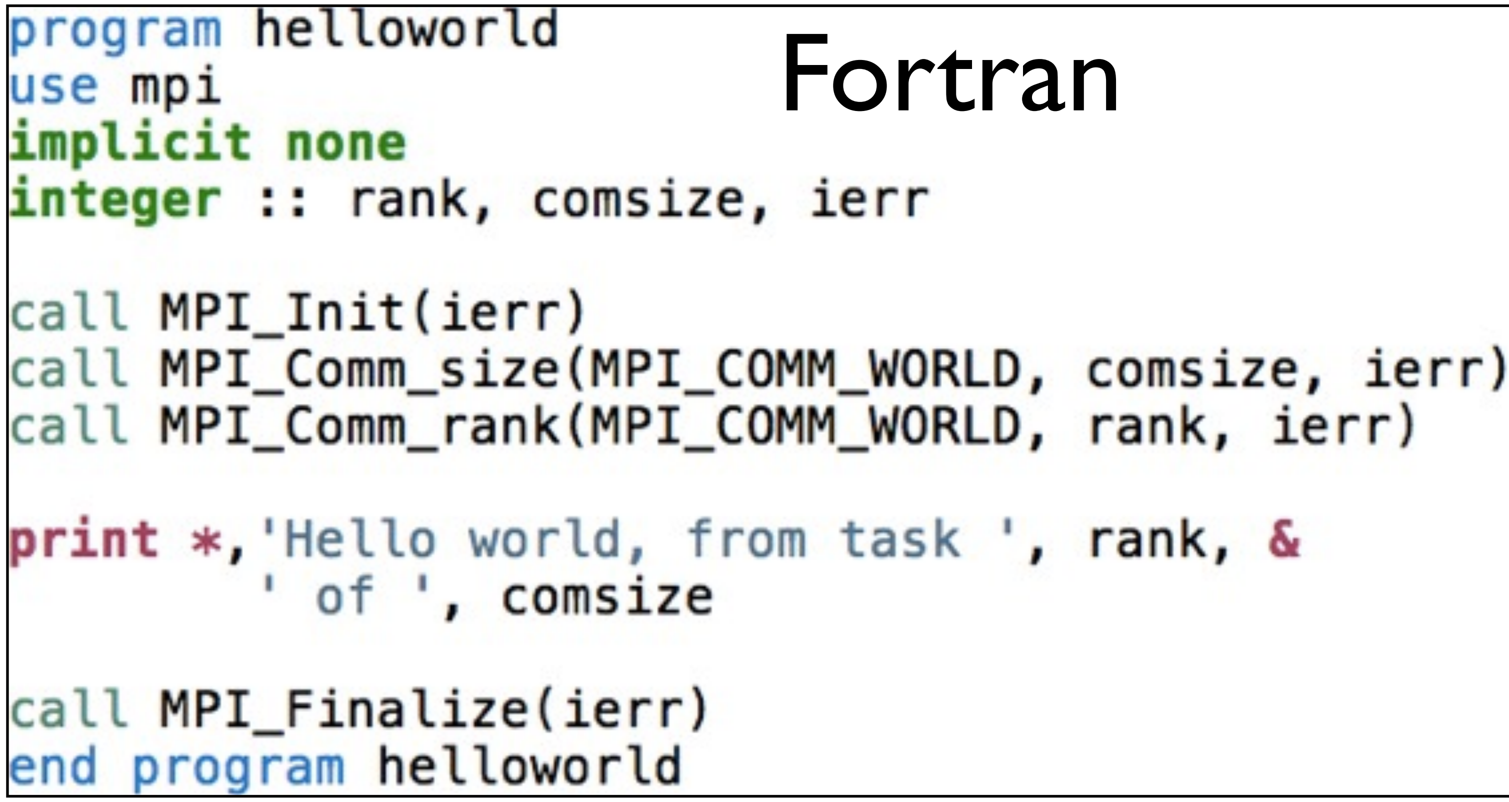
Size of MPI Library

- 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()
```

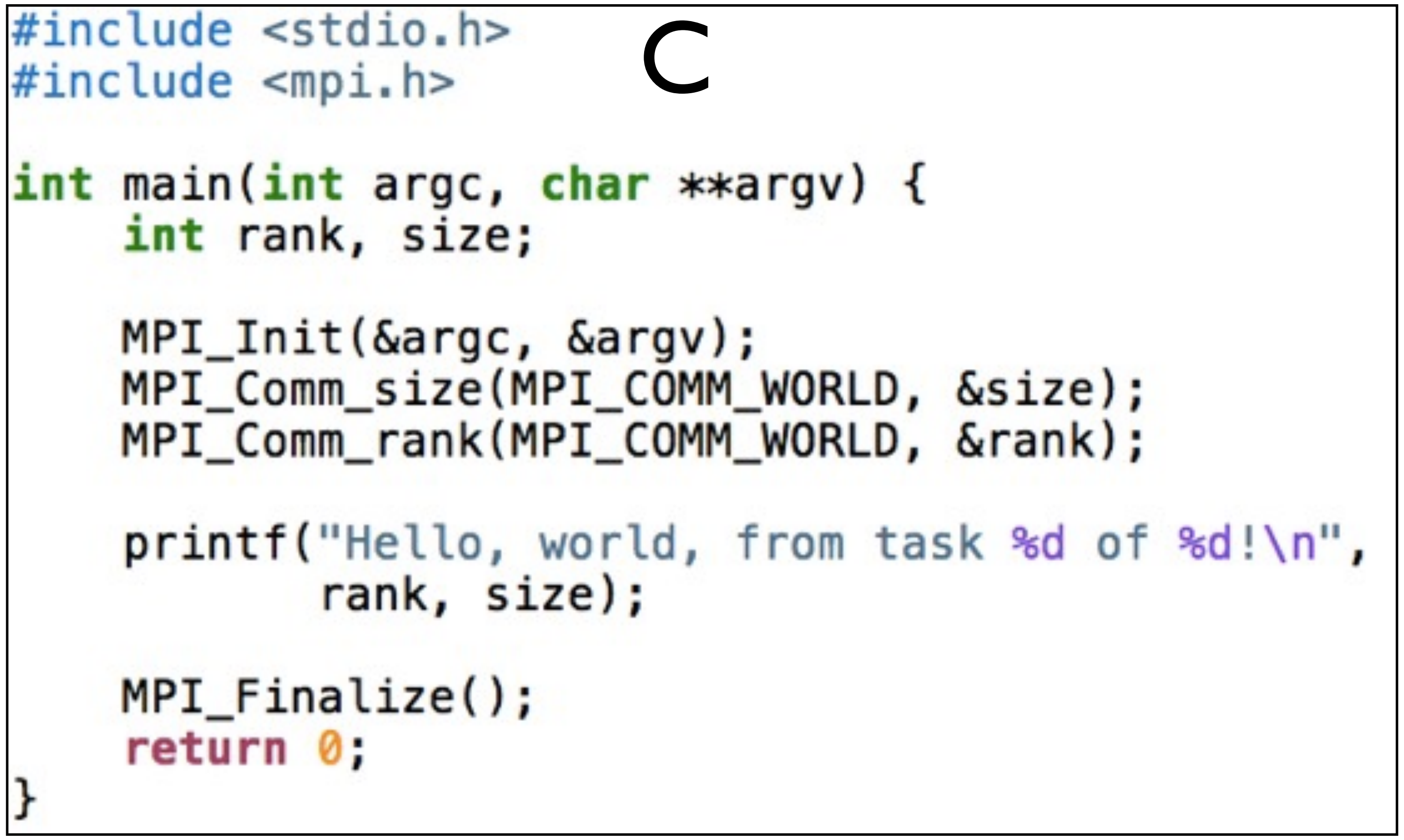
Hello World

- The obligatory starting point
- cd ~/ppp/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

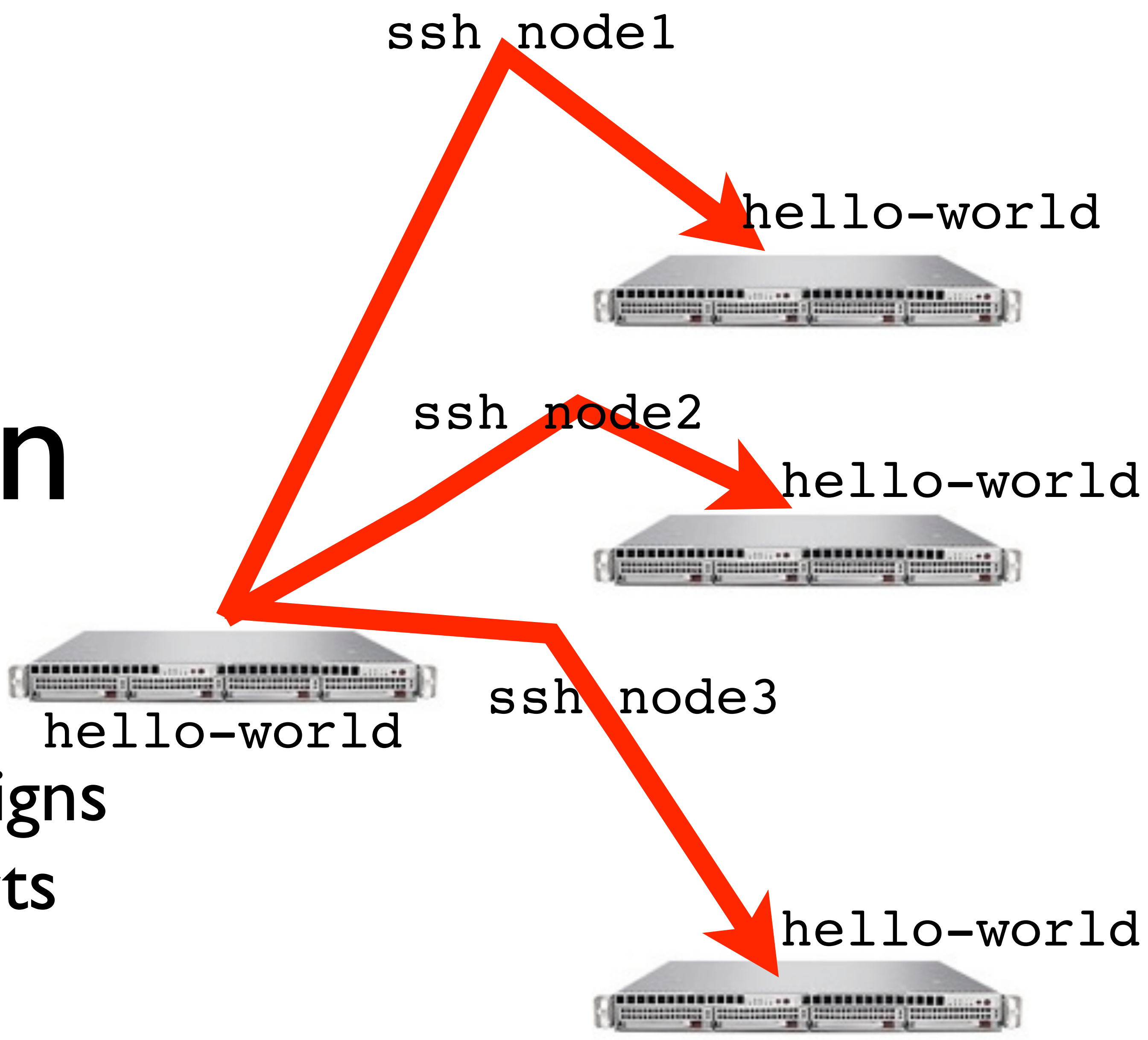
```
$ mpicc hello-world.c
         -o hello-world
$ mpirun -np 1 hello-world
$ mpirun -np 2 hello-world
$ mpirun -np 8 hello-world
```

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

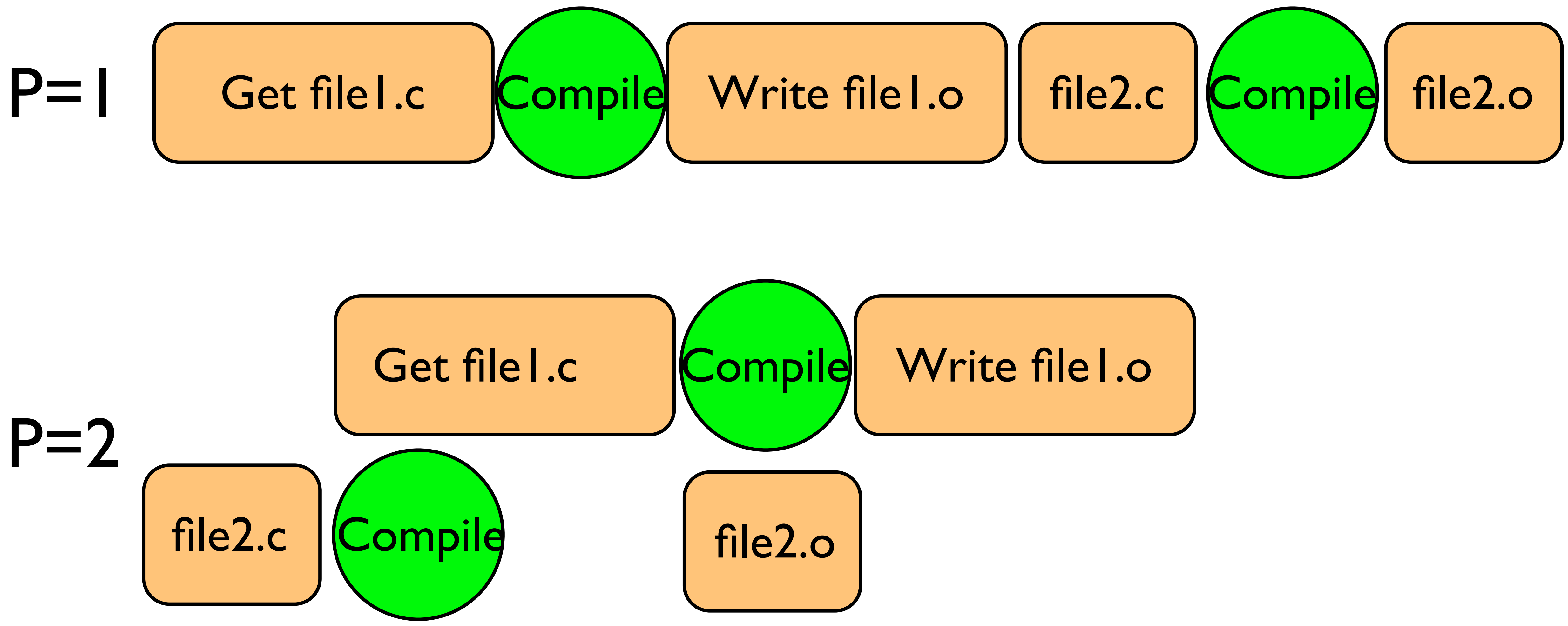
```
$ hostname
$ mpirun -np 4 hostname
$ ls
$ mpirun -np 4 ls
```

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

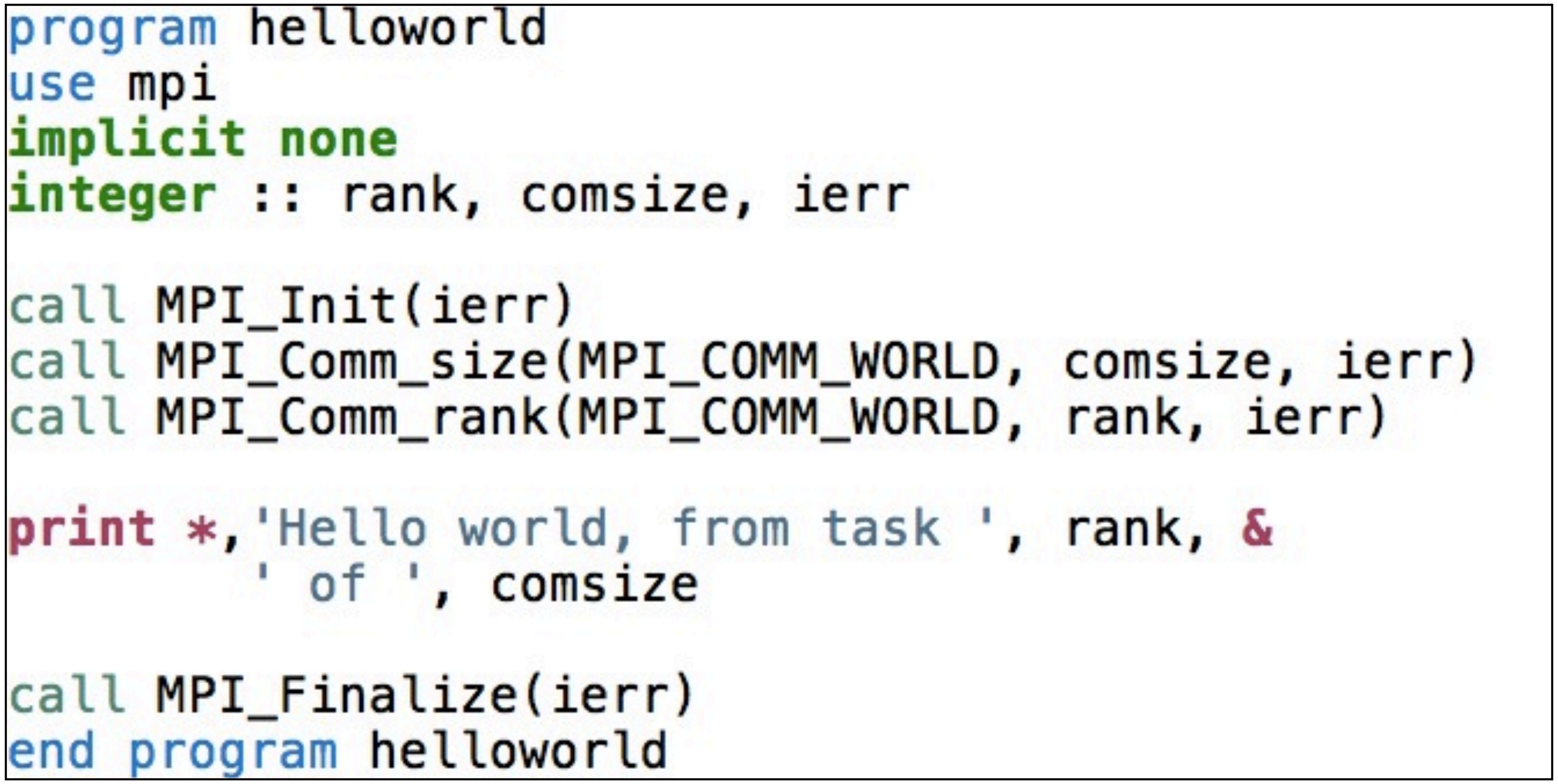
```
$ make
$ make -j 2
$ make -j
```

Overlapping Computation with I/O



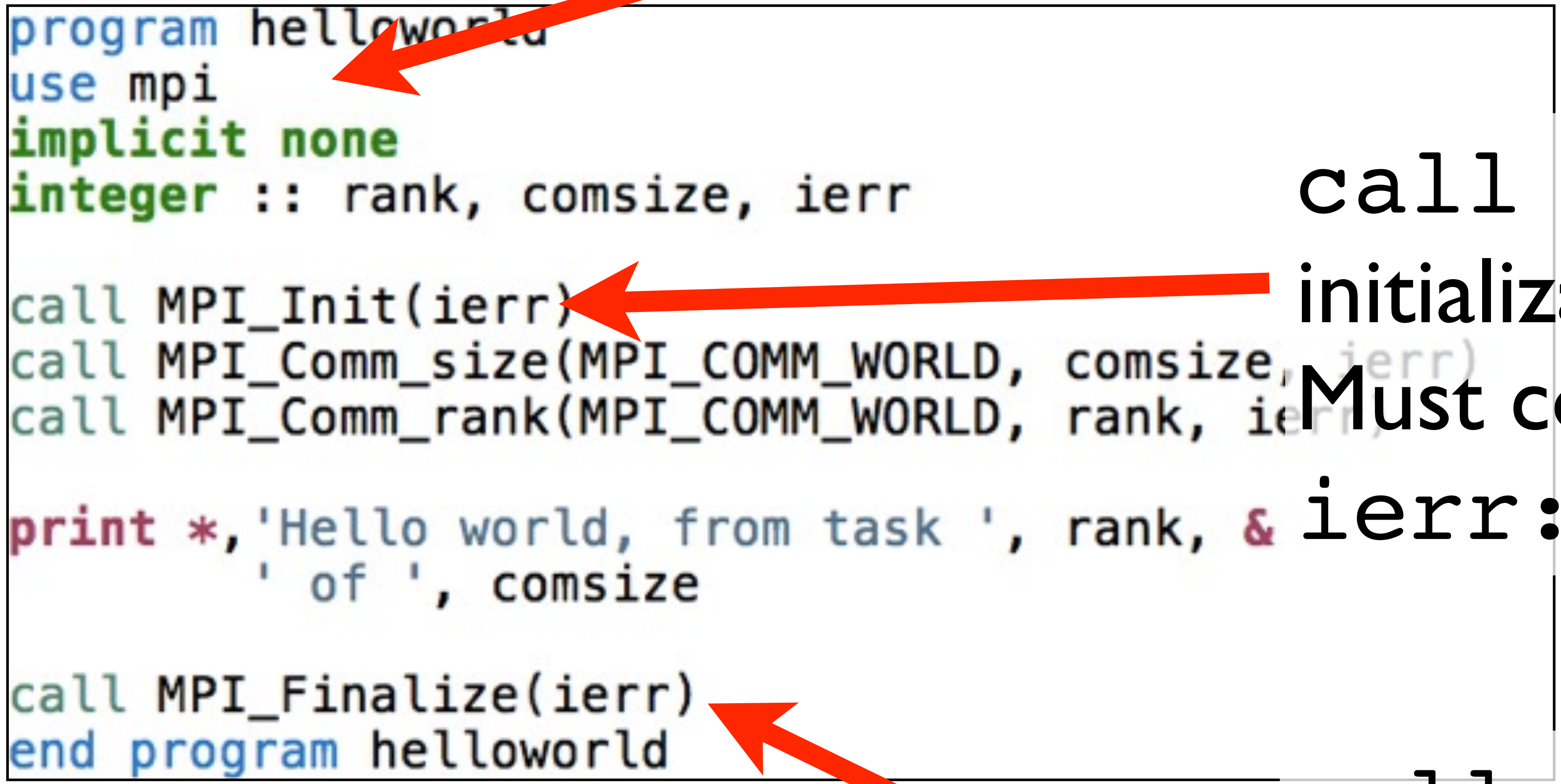
What the code does

- (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

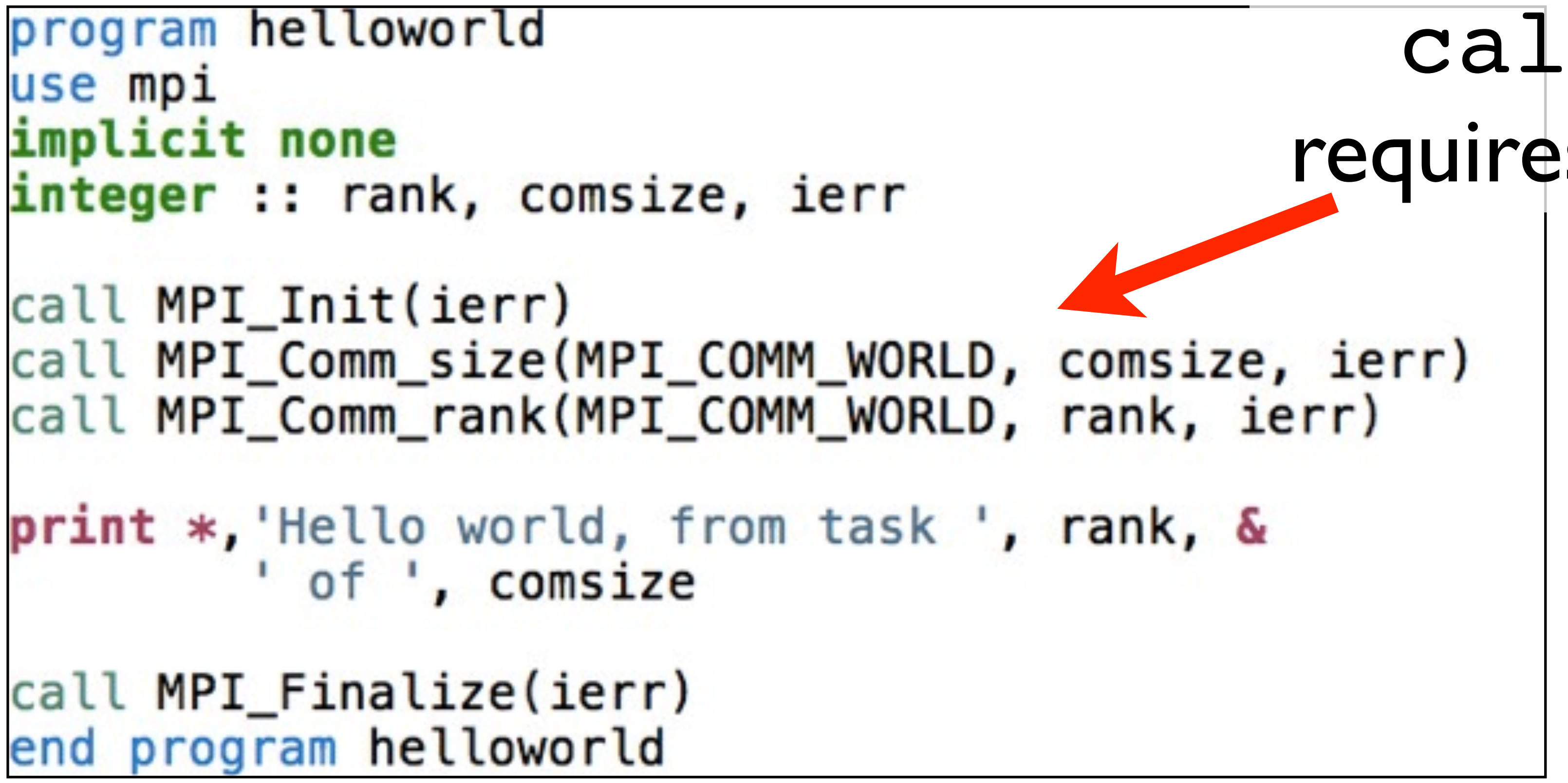


```
program helloworld
use mpi
implicit none
integer :: rank, comsize, ierr

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.

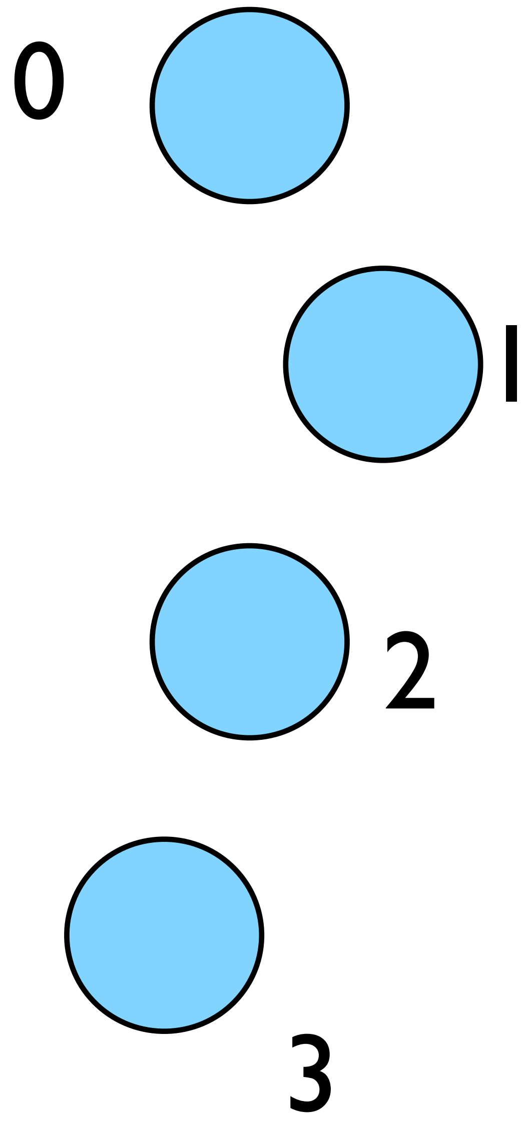
`call MPI_COMM_RANK`,
`call MPI_COMM_SIZE`:
requires a little more exposition.



```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

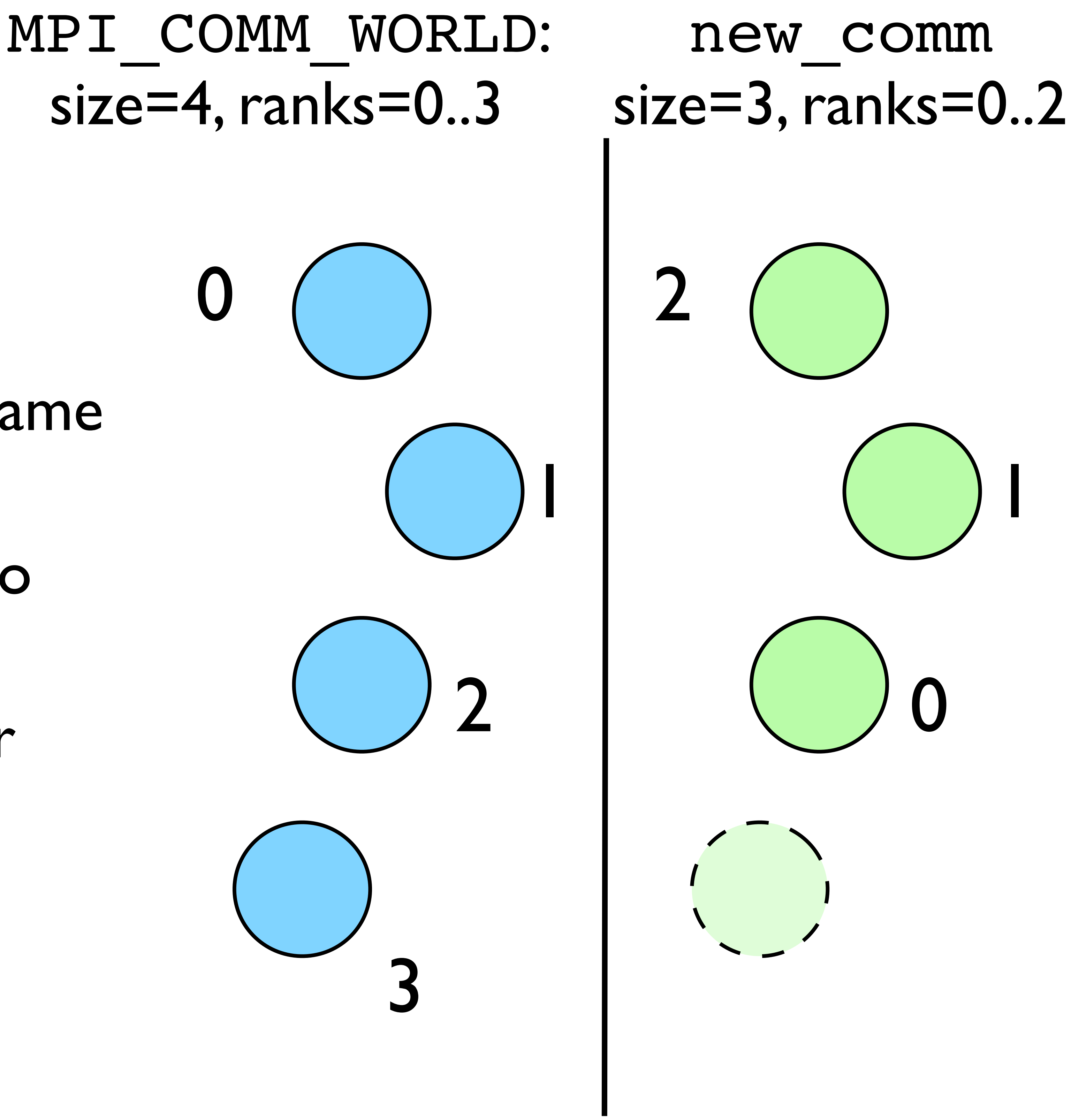
- MPI groups processes into communicators.
- Each communicator has some size -- number of tasks.
- Each task has a rank 0..size-1
- Every task in your program belongs to `MPI_COMM_WORLD`

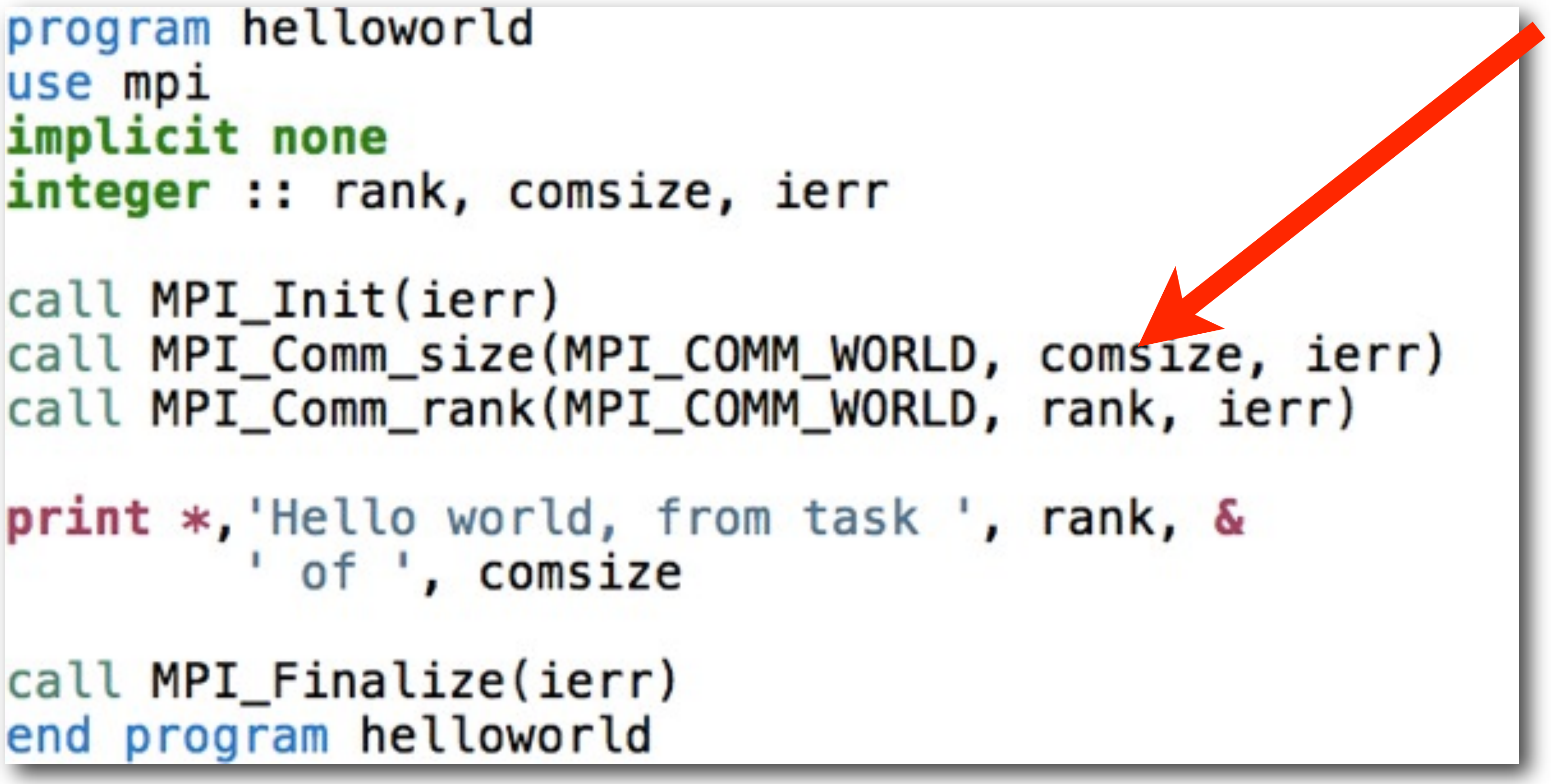


`MPI_COMM_WORLD`:
size=4, ranks=0..3

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





```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
```

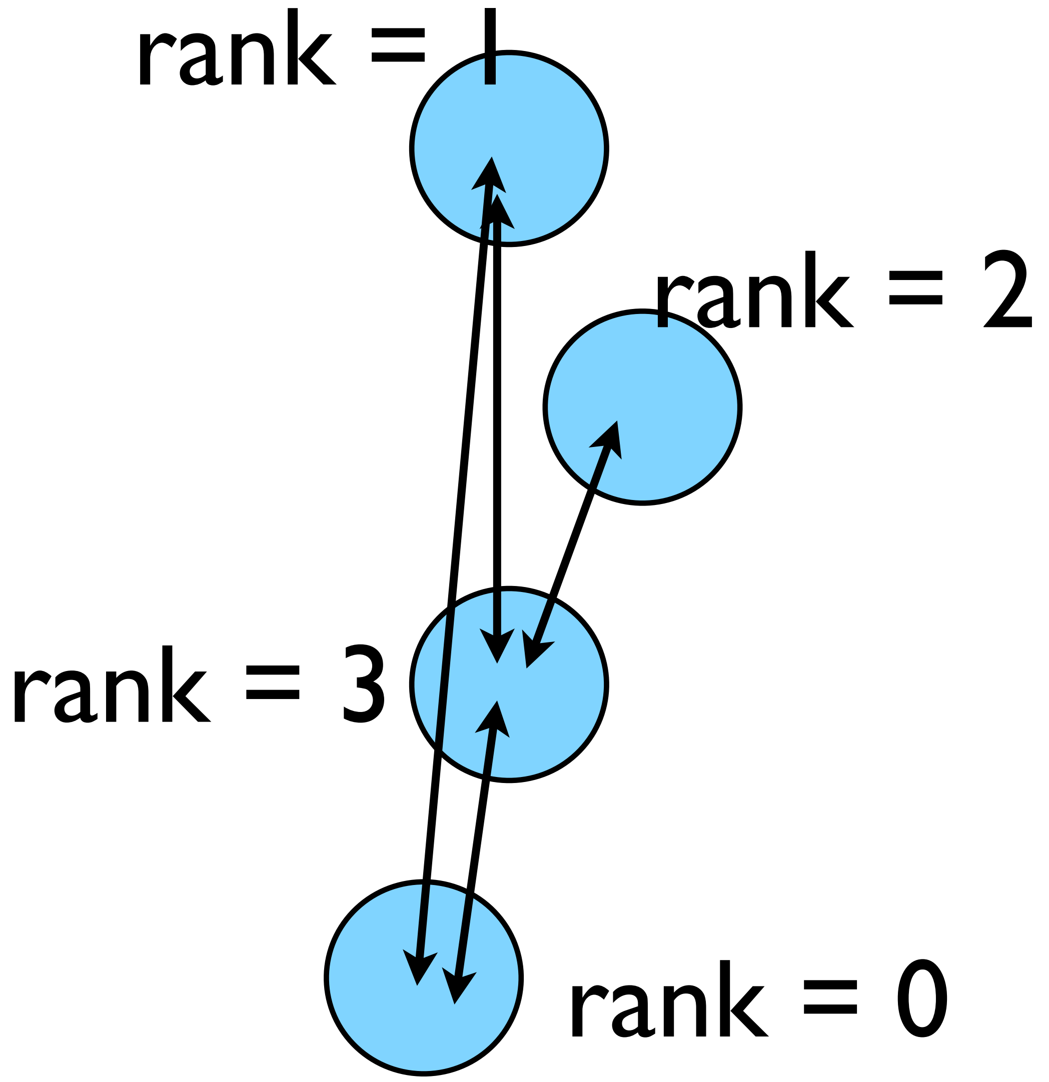
`call MPI_COMM_RANK,`
`call MPI_COMM_SIZE:`

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

put answers in `rank` and
`size`

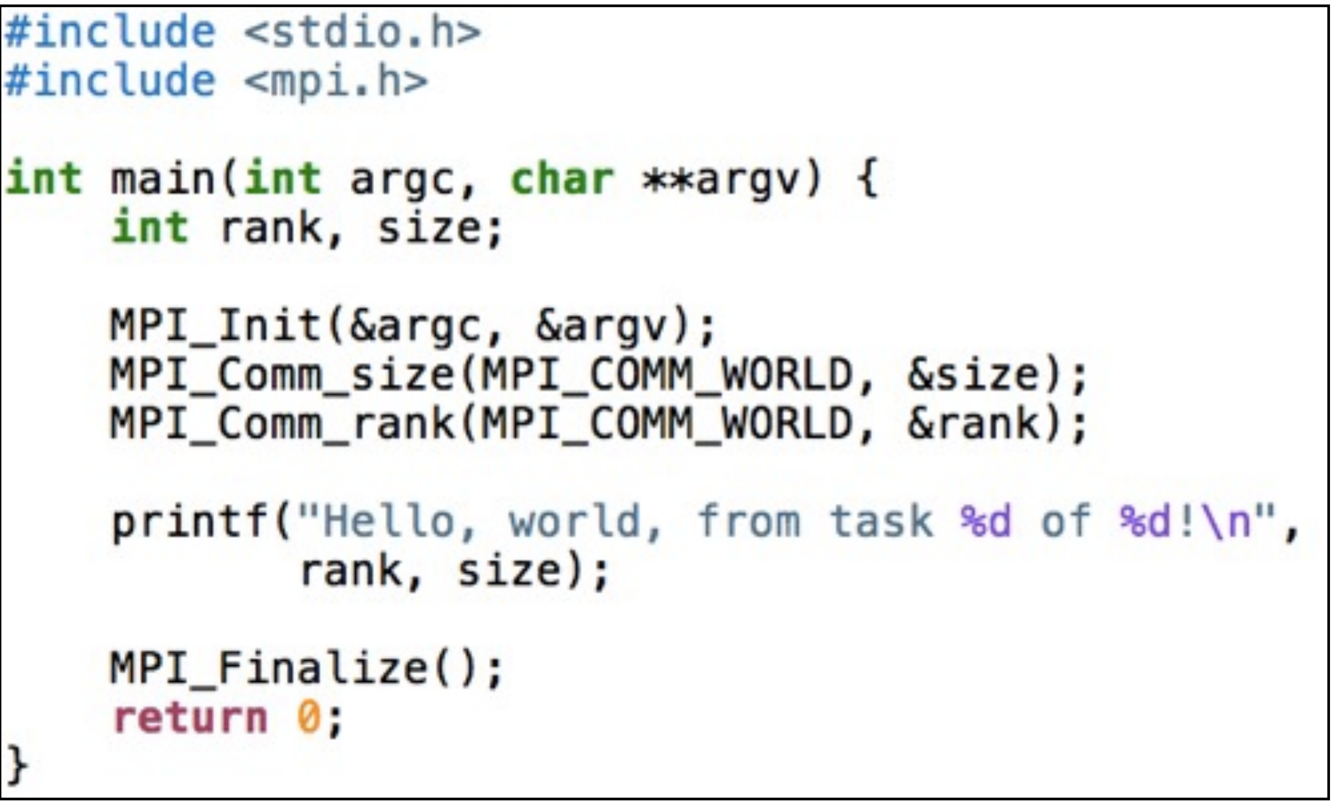
Rank and Size much more important in MPI than OpenMP

- 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.

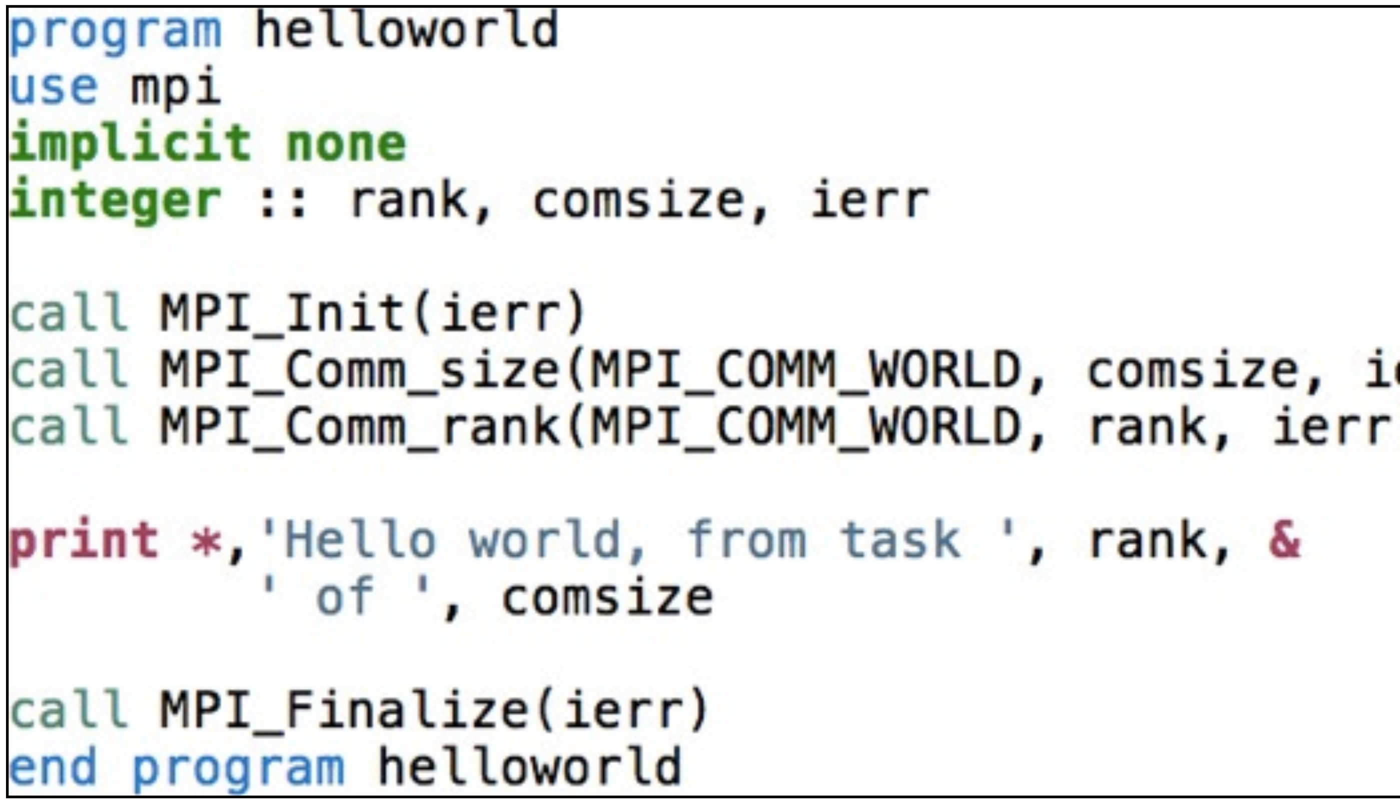


C

Fortran



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

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

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

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

    MPI_Finalize();
    return 0;
}
```

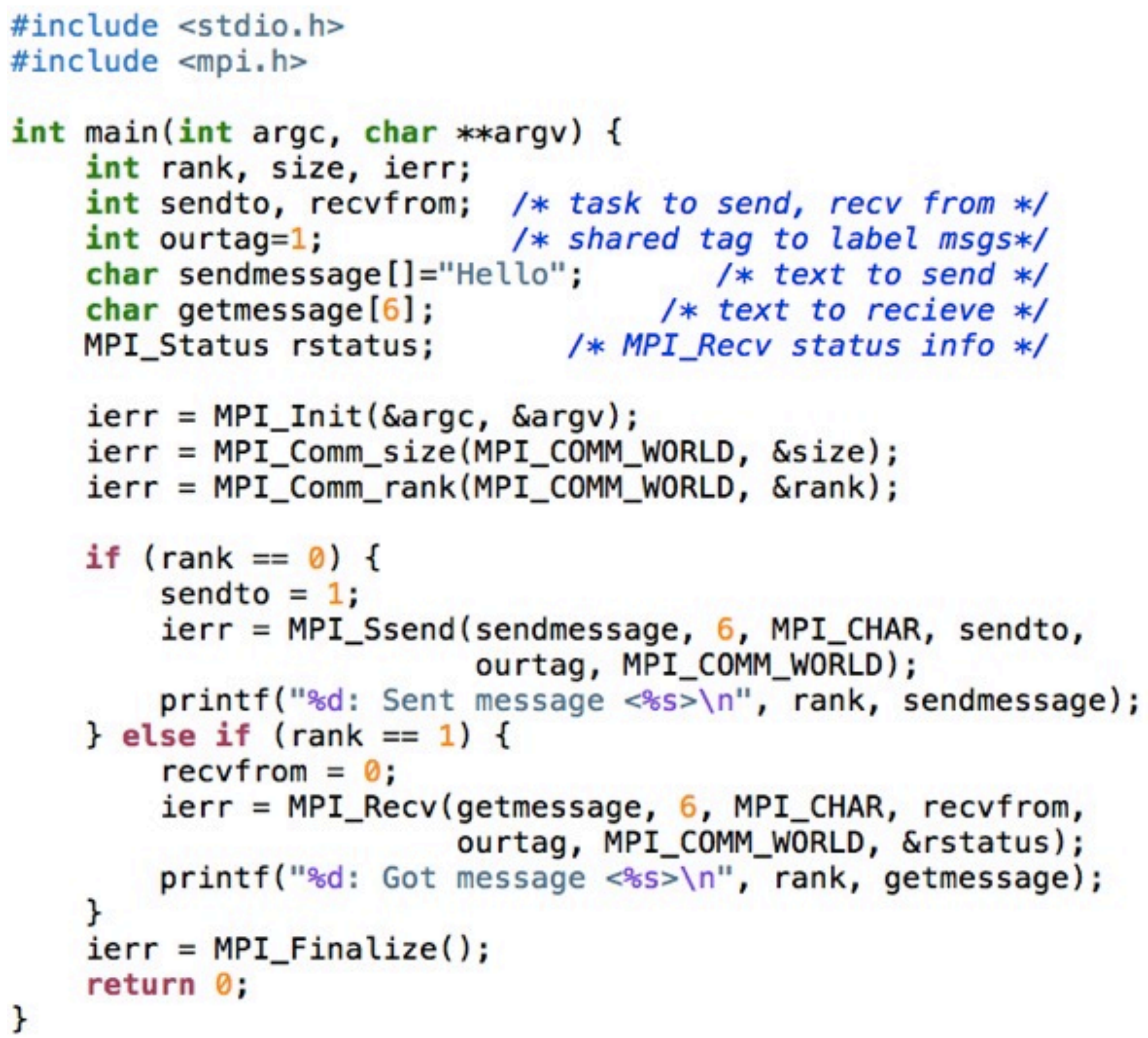


```fortran
program helloworld
use mpi
implicit none
integer :: rank, comsize, ierr

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

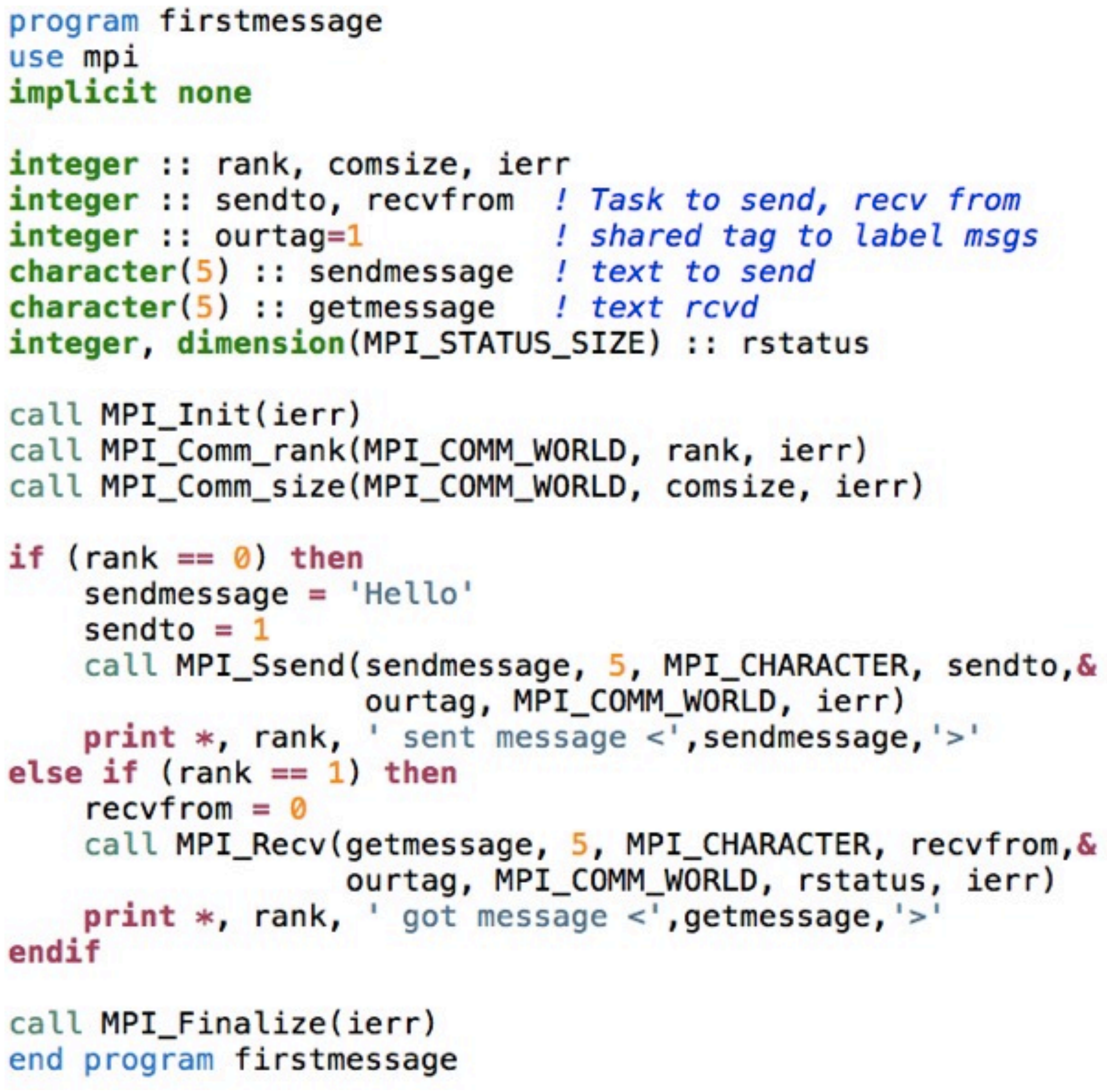
- Let's fix this
- mpicc -o firstmessage firstmessage.c
- mpirun -np 2 ./firstmessage
- Note: C - MPI_CHAR



```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)

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

Special Source/Dest: MPI_PROC_NULL

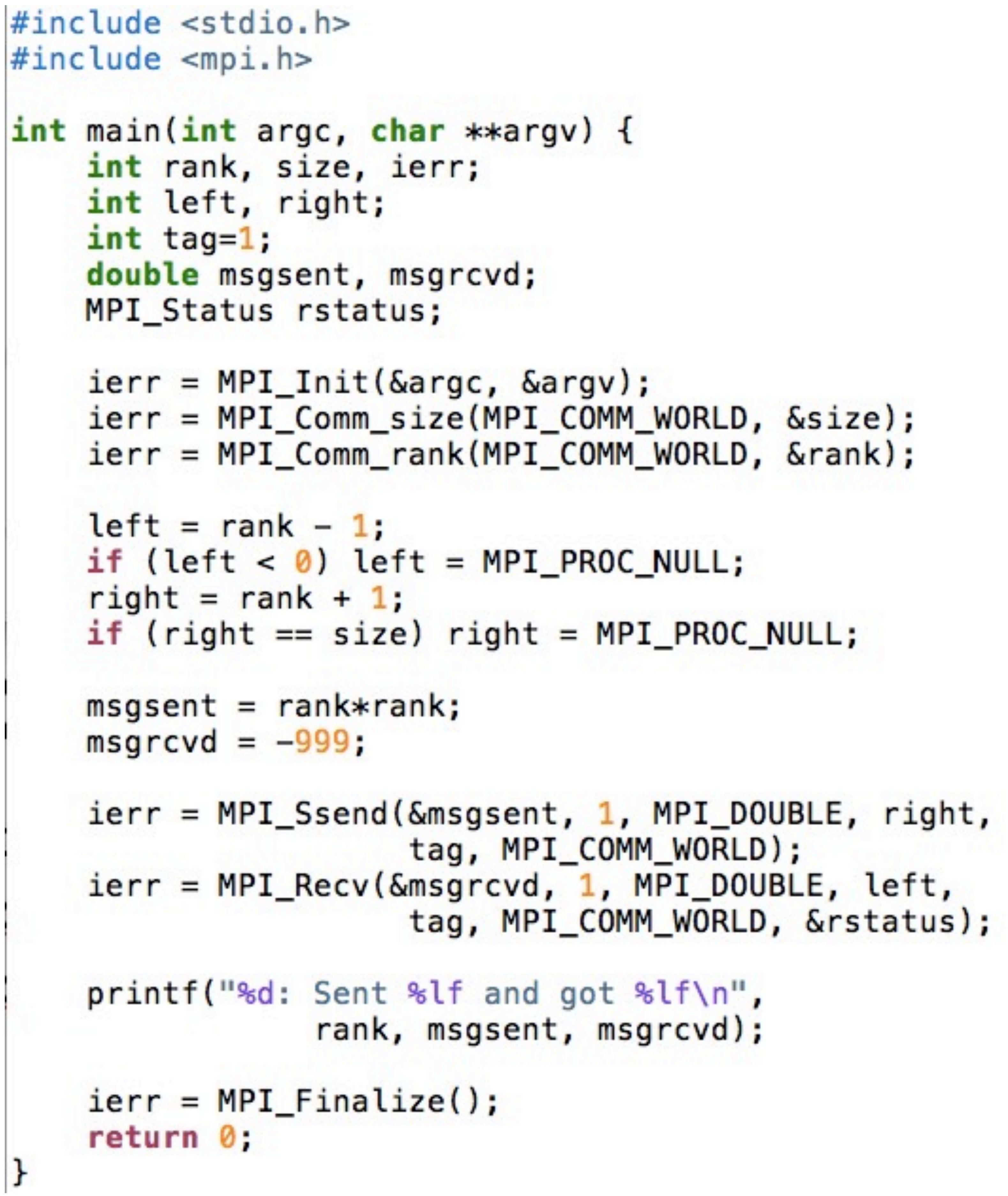
`MPI_PROC_NULL` basically ignores the relevant operation; can lead to cleaner code.

Special Source: MPI_ANY_SOURCE

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

More complicated example:

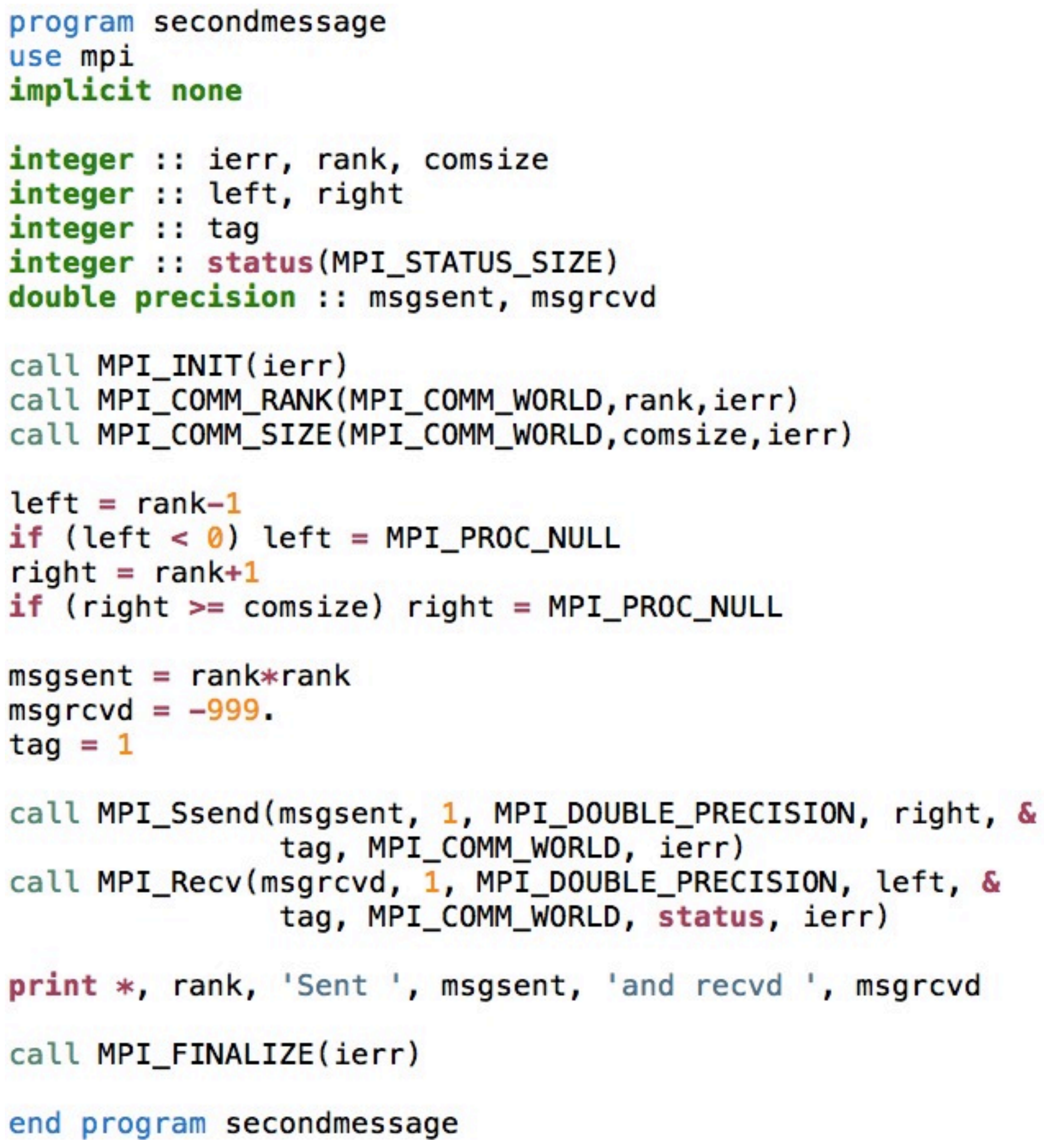
- Let's look at secondmessage.f90, secondmessage.c



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

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

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

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

    msgsent = rank*rank;
    msgrcvd = -999;

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

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

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

More complicated example:

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



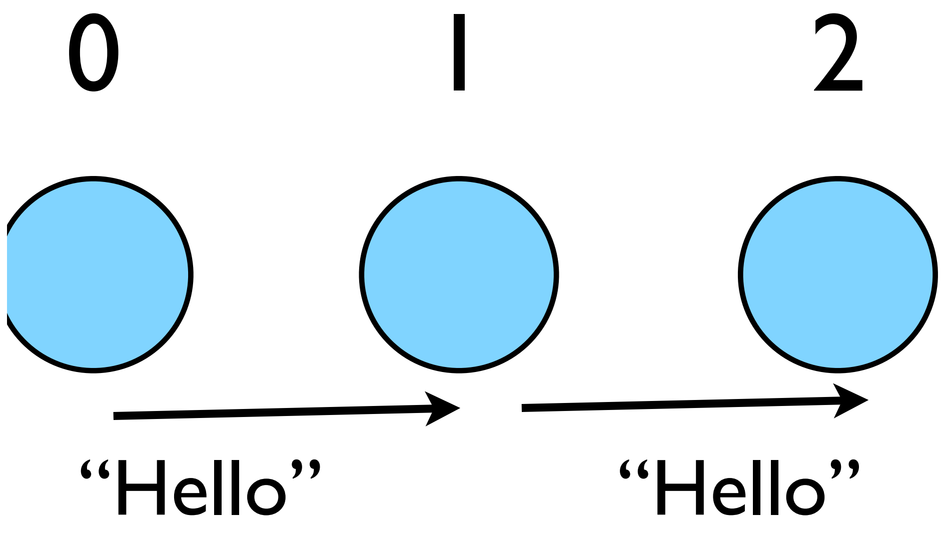
```fortran
program secondmessage
use mpi
implicit none

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

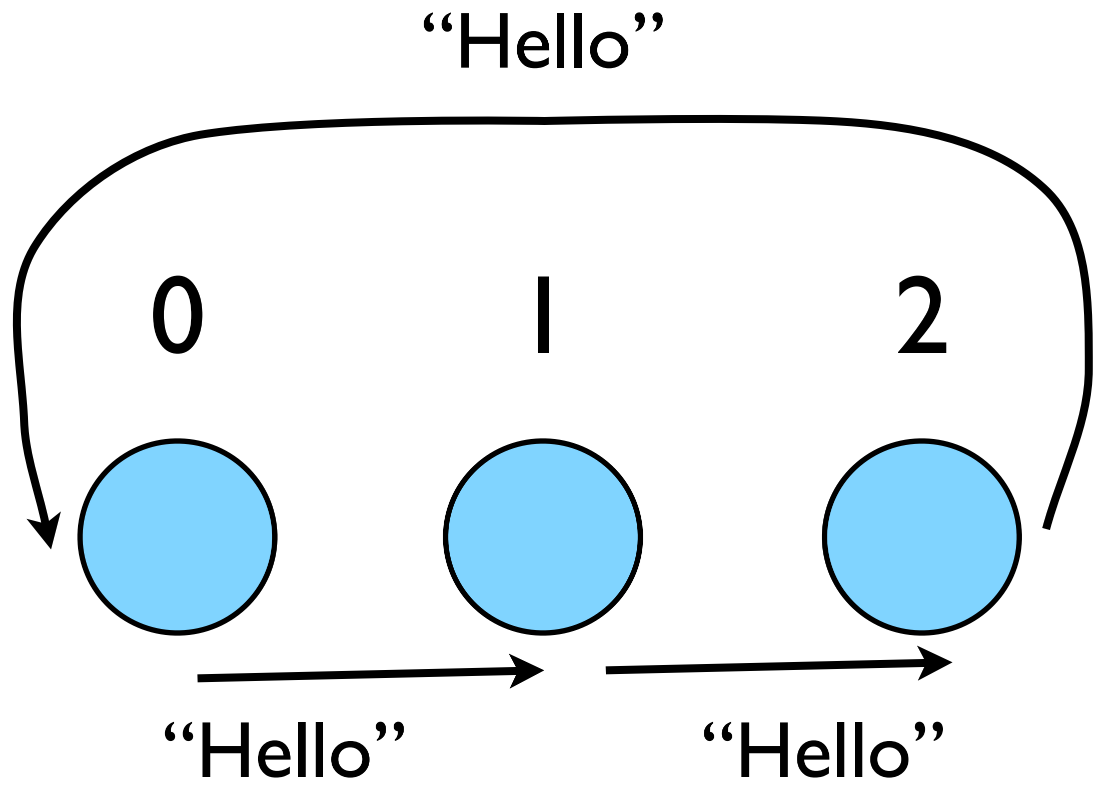
```
$ mpirun -np 4 ./secondmessage
3: Sent 9.000000 and got 4.000000
0: Sent 0.000000 and got -999.000
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
```

```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;
}
```

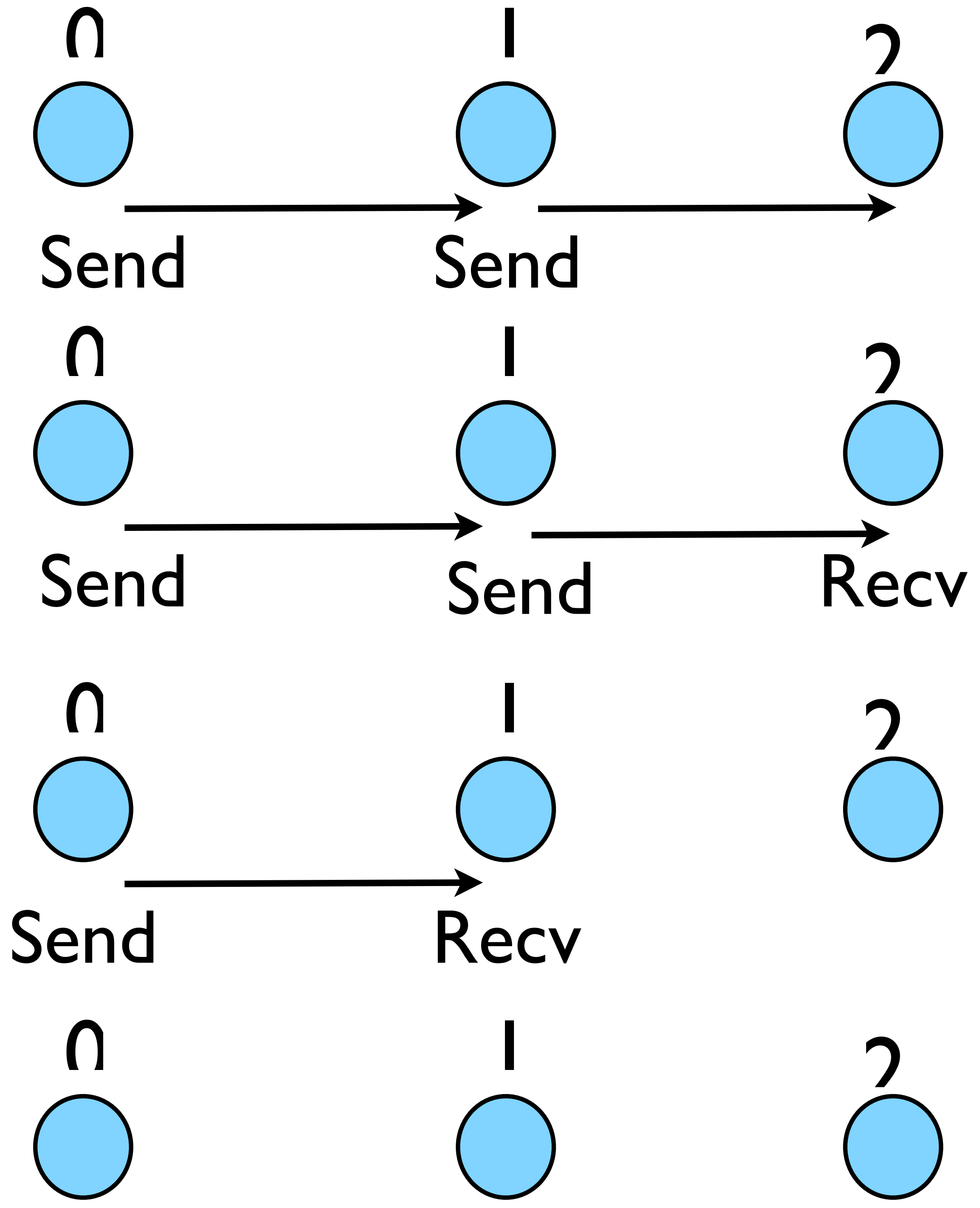


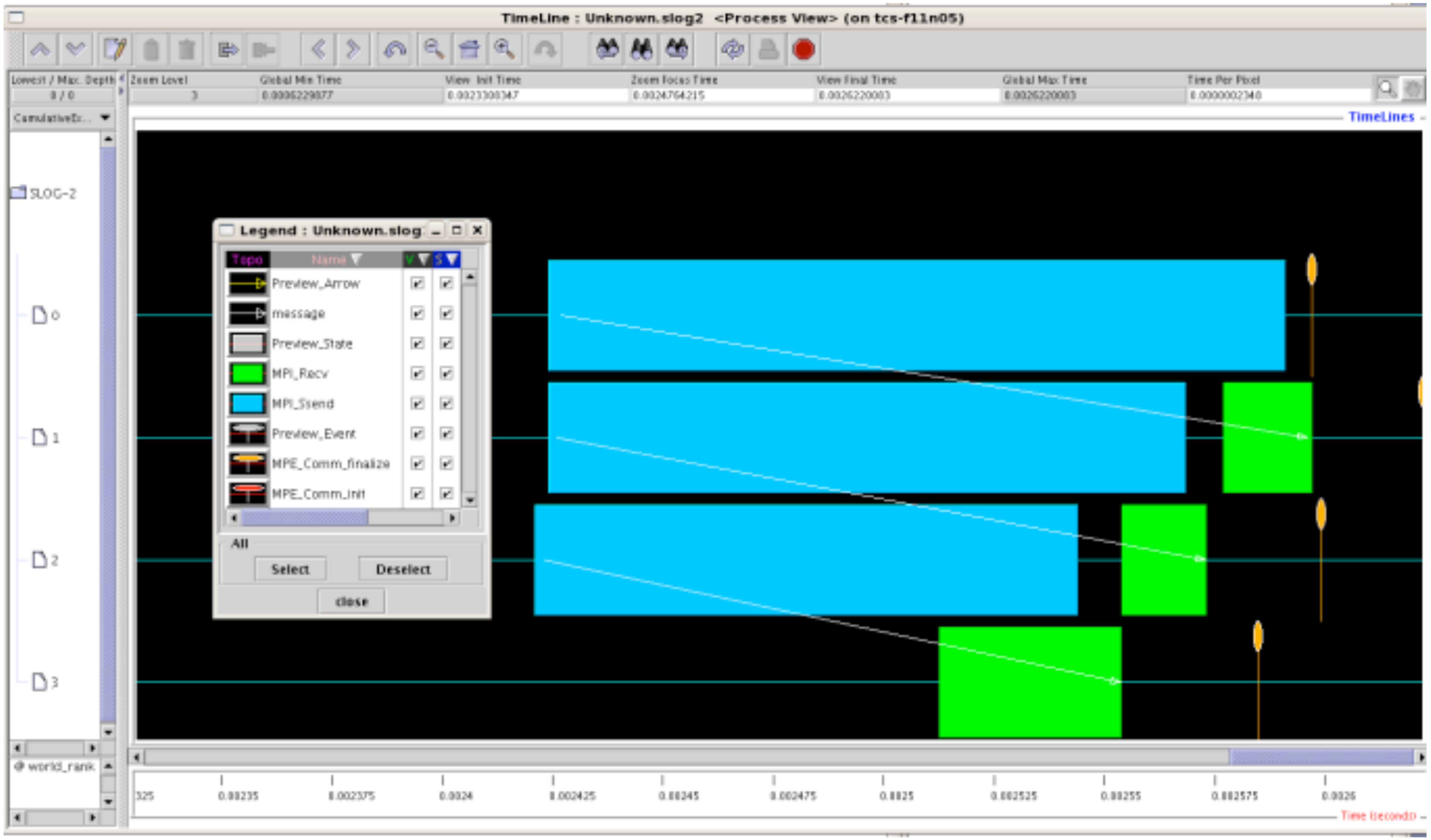
Implement periodic boundary conditions

- 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)
```

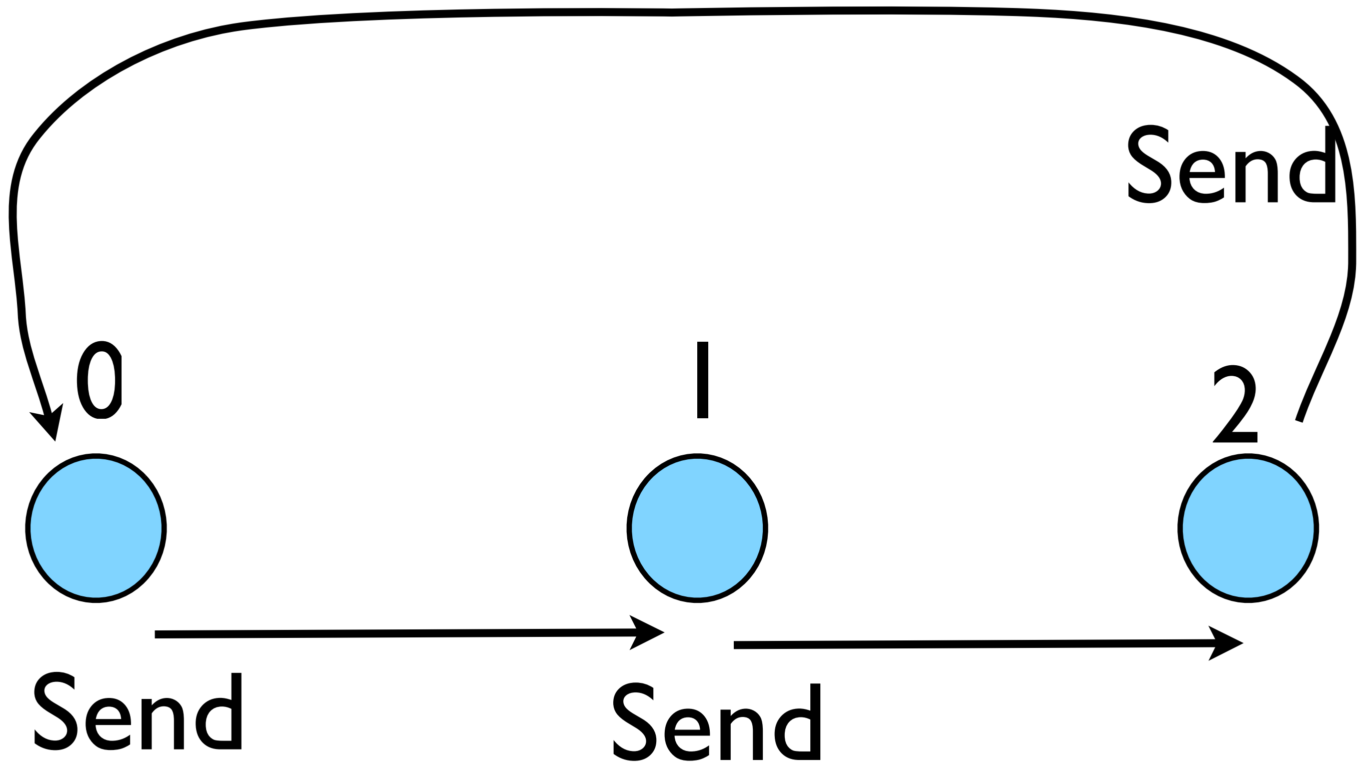



TimeLine : Unknown.slog2 <Process View> (on tcs-f11n05)

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	✓	✓

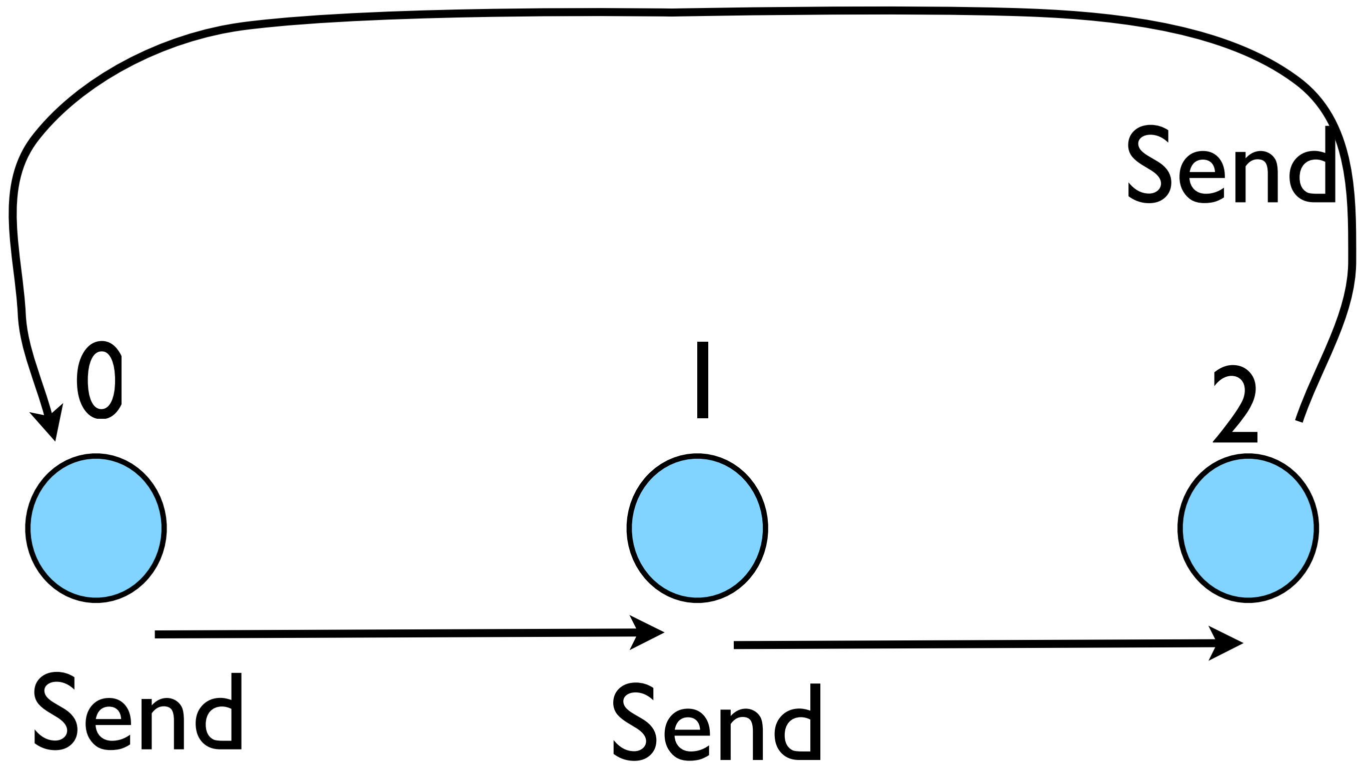


```
left = rank-1
if (left < 0) left = comsize-1
right = rank+1
if (right >= comsize) right = 0

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.



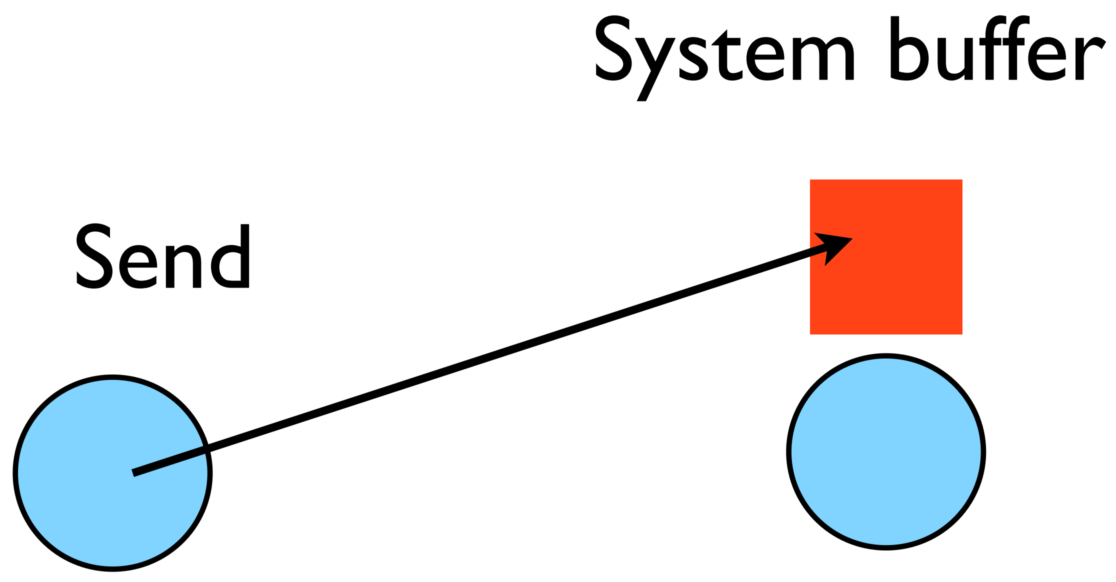
Big MPI Lesson #1

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

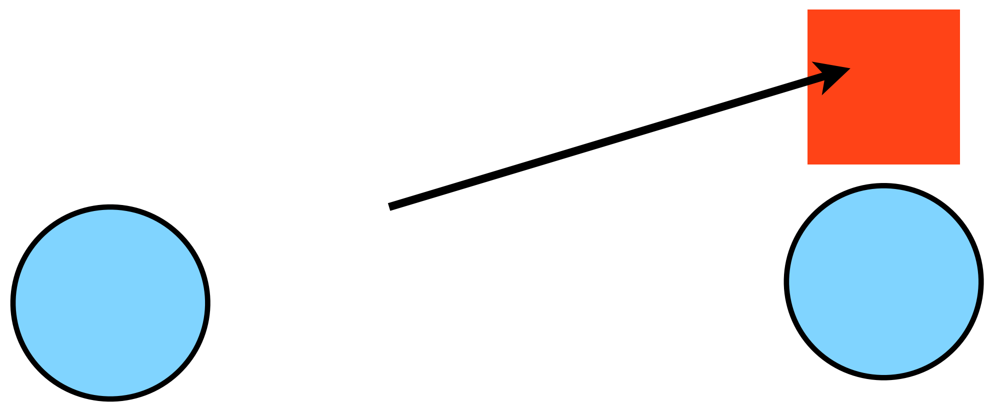
Different versions of SEND

- 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

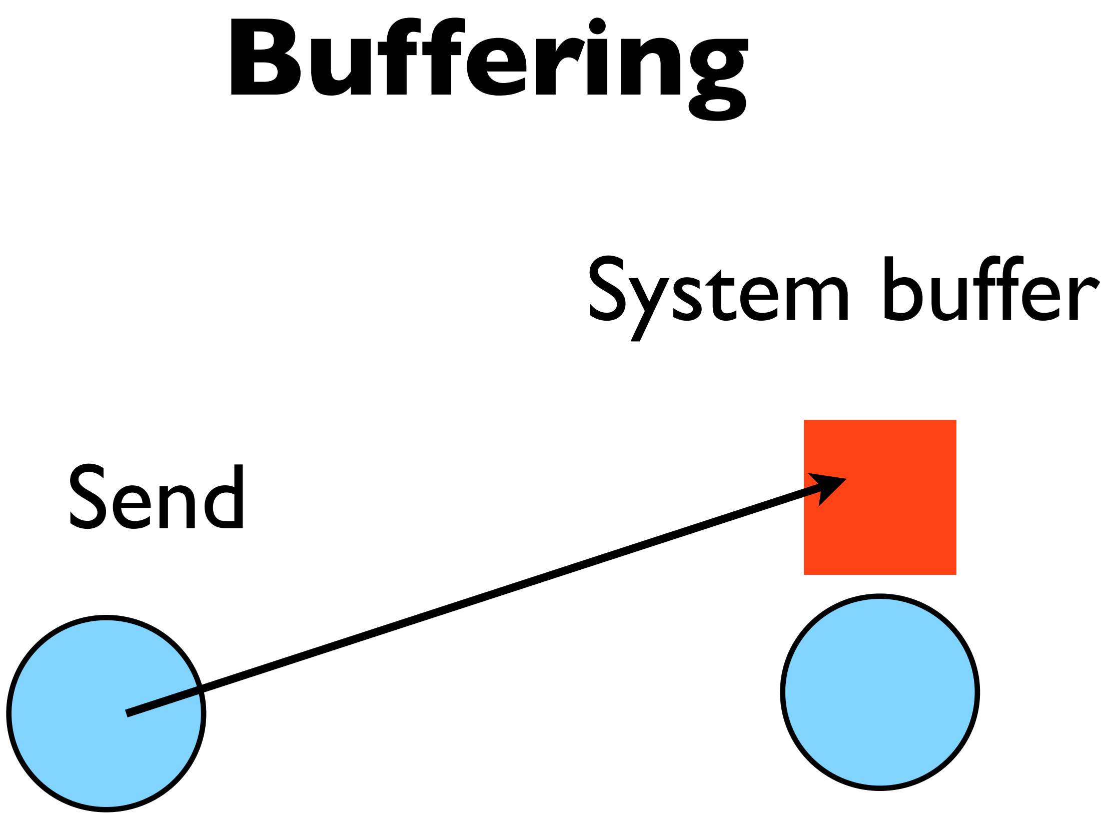


(Non) Blocking

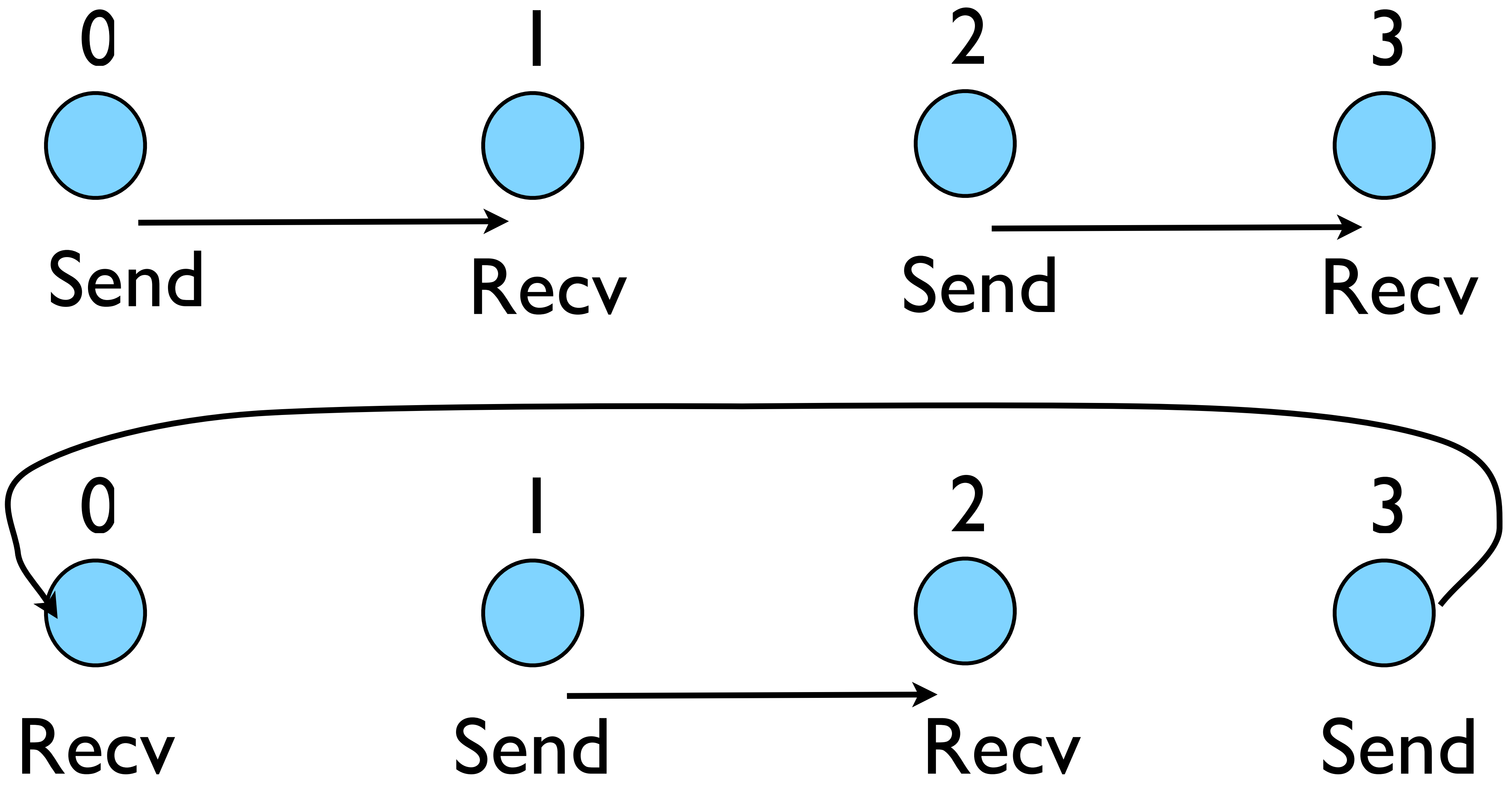


Buffering is dangerous!

- 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)



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?

```fortran
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);
```

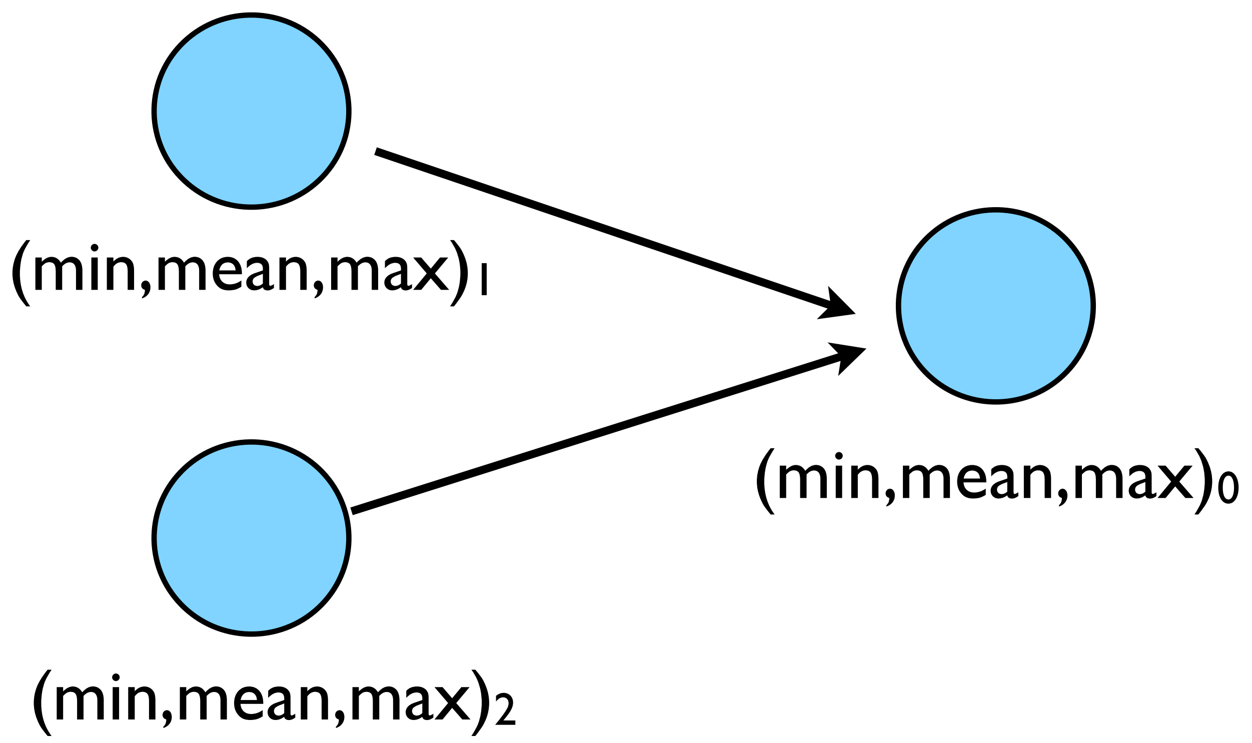
Recv Args

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?

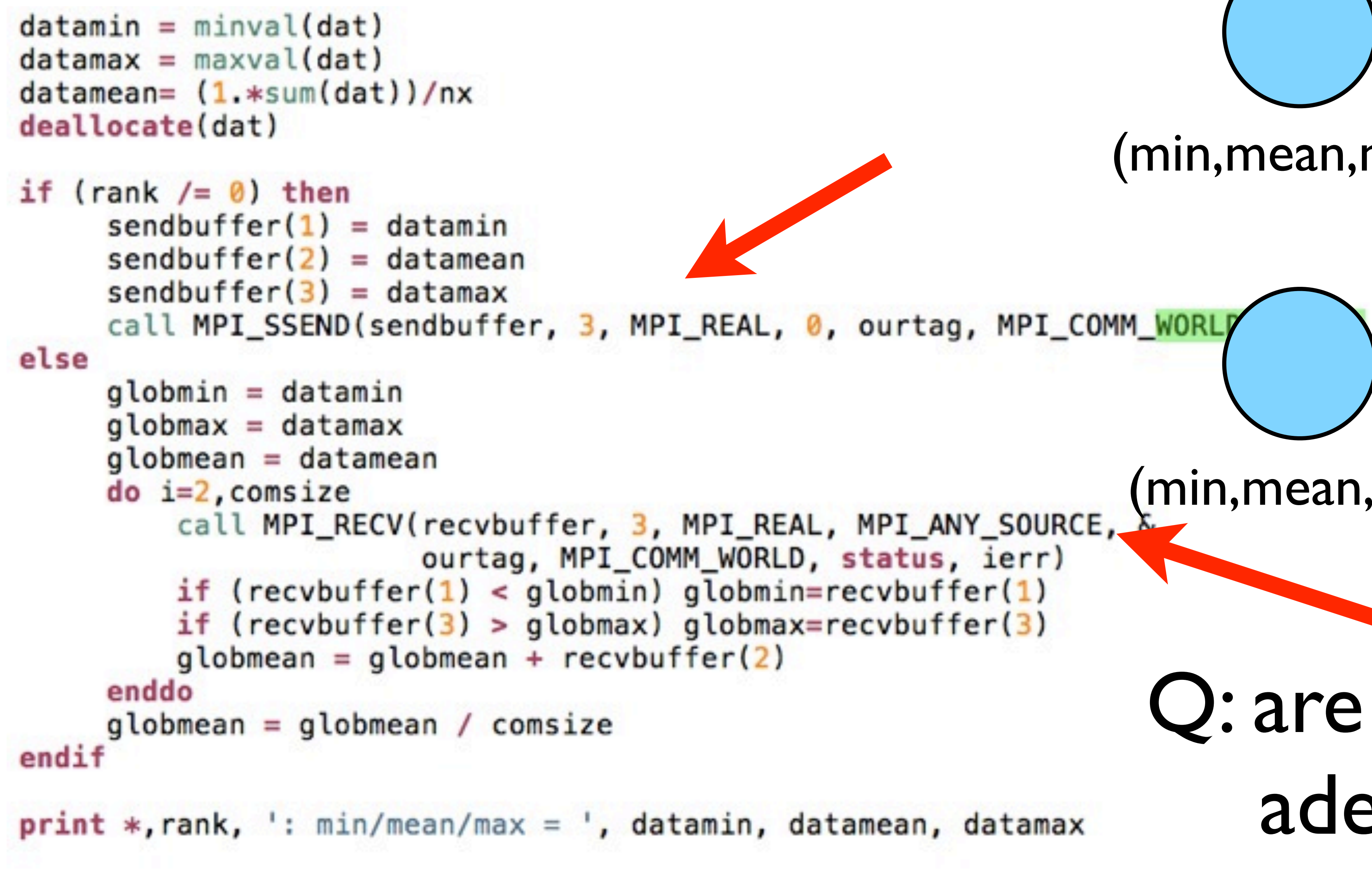
Min, Mean, Max of numbers



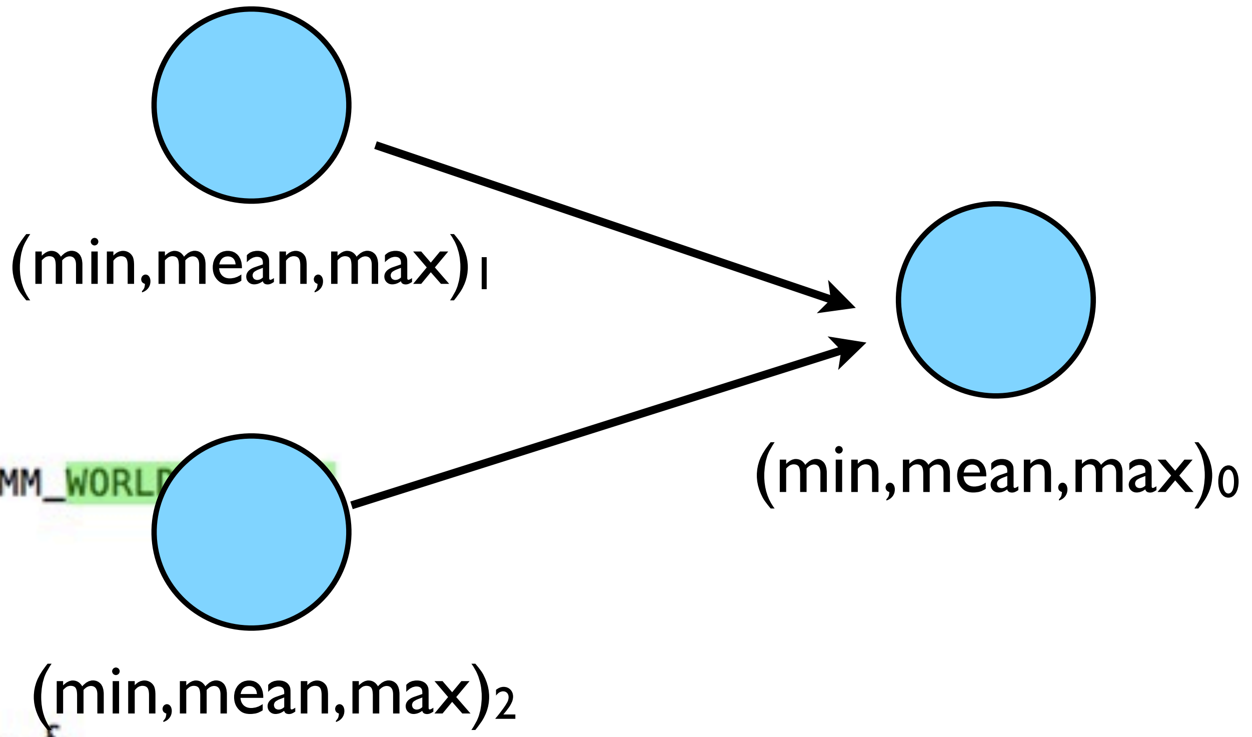
- Lets try some code that calculates the min/mean/max of a bunch of random numbers -1..1. Should go to -1,0,+1 for large N.
- Each gets their partial results and sends it to some node, say node 0 (why node 0?)
- ~/ppp/mpi-intro/minmeanmax.{c,f90}
- How to MPI it?

```fortran
      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);
```



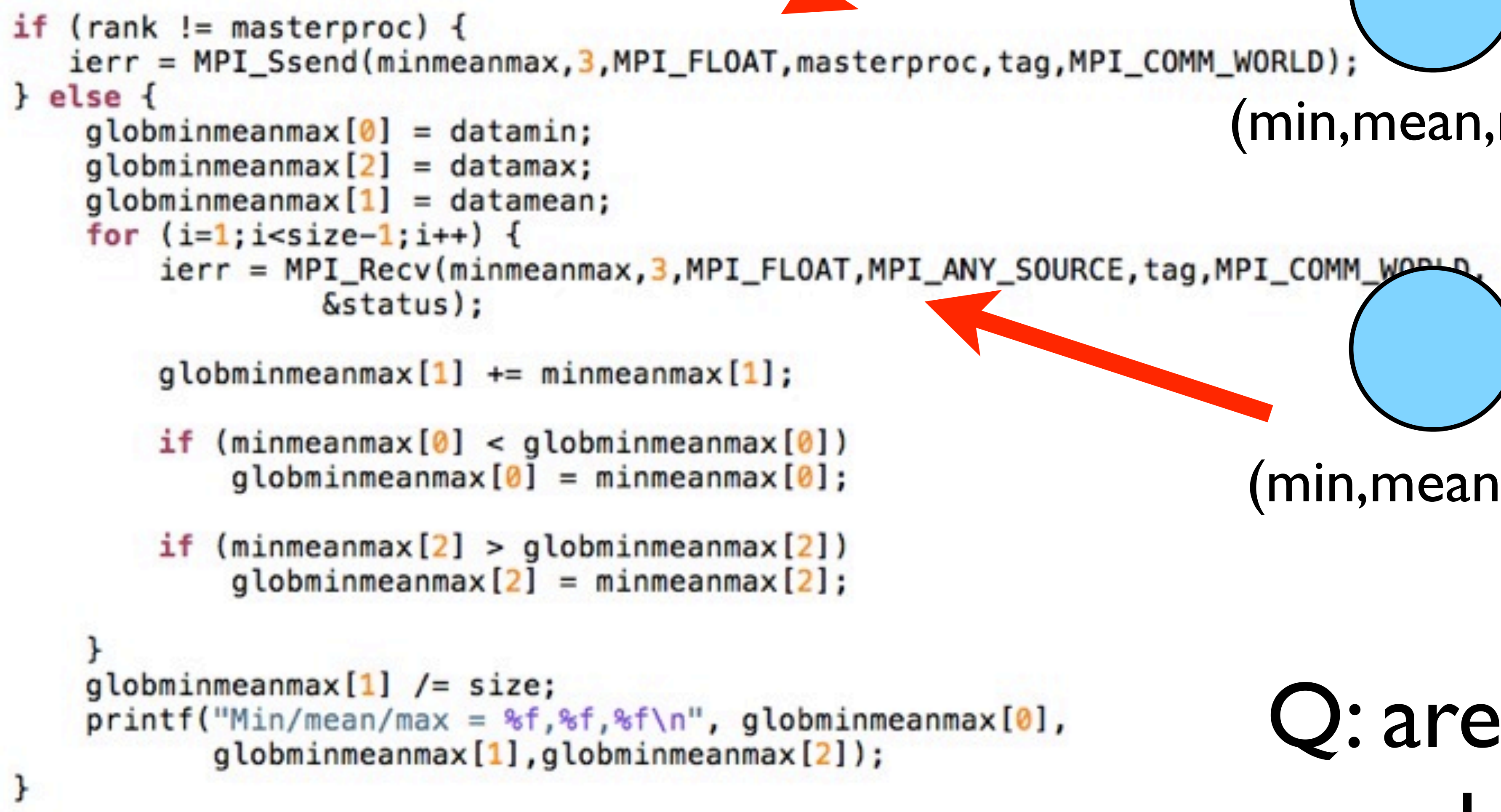
```fortran
datamin = minval(dat)
datamax = maxval(dat)
datamean= (1.*sum(dat))/nx
deallocate(dat)

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
```



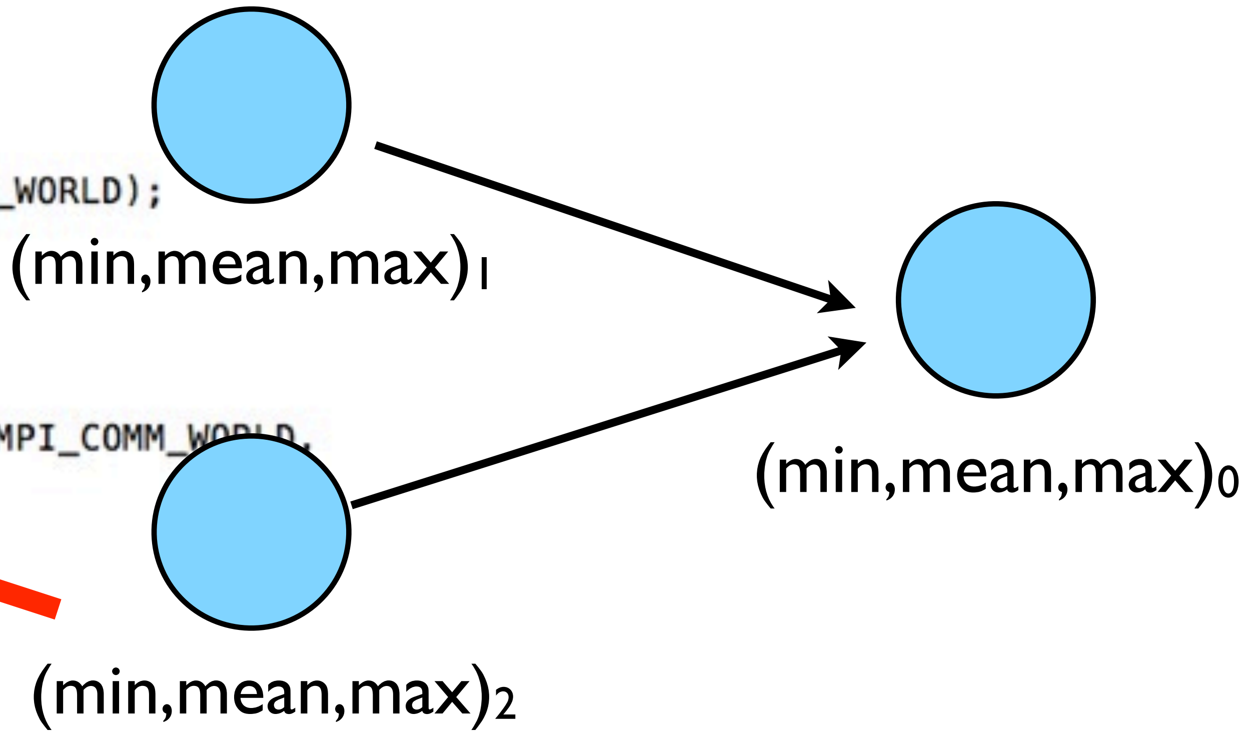
$(\text{min,mean,max})_1$
$(\text{min,mean,max})_0$
$(\text{min,mean,max})_2$

Q: are these sends/recvd adequately paired?

minmeanmax-mpi.f90



```
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]);
}
```

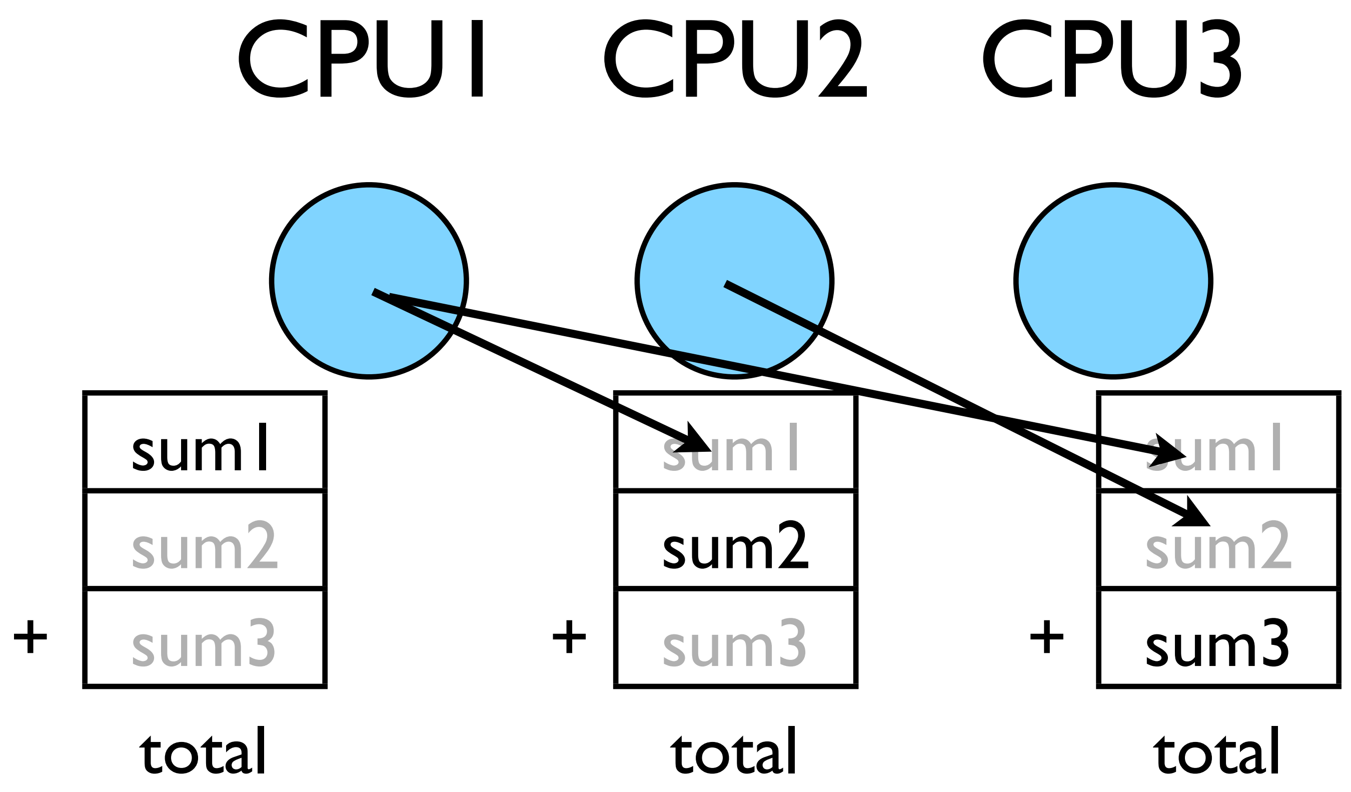


Q: are these sends/recvd adequately paired?

minmeanmax-mpi.c

Inefficient!

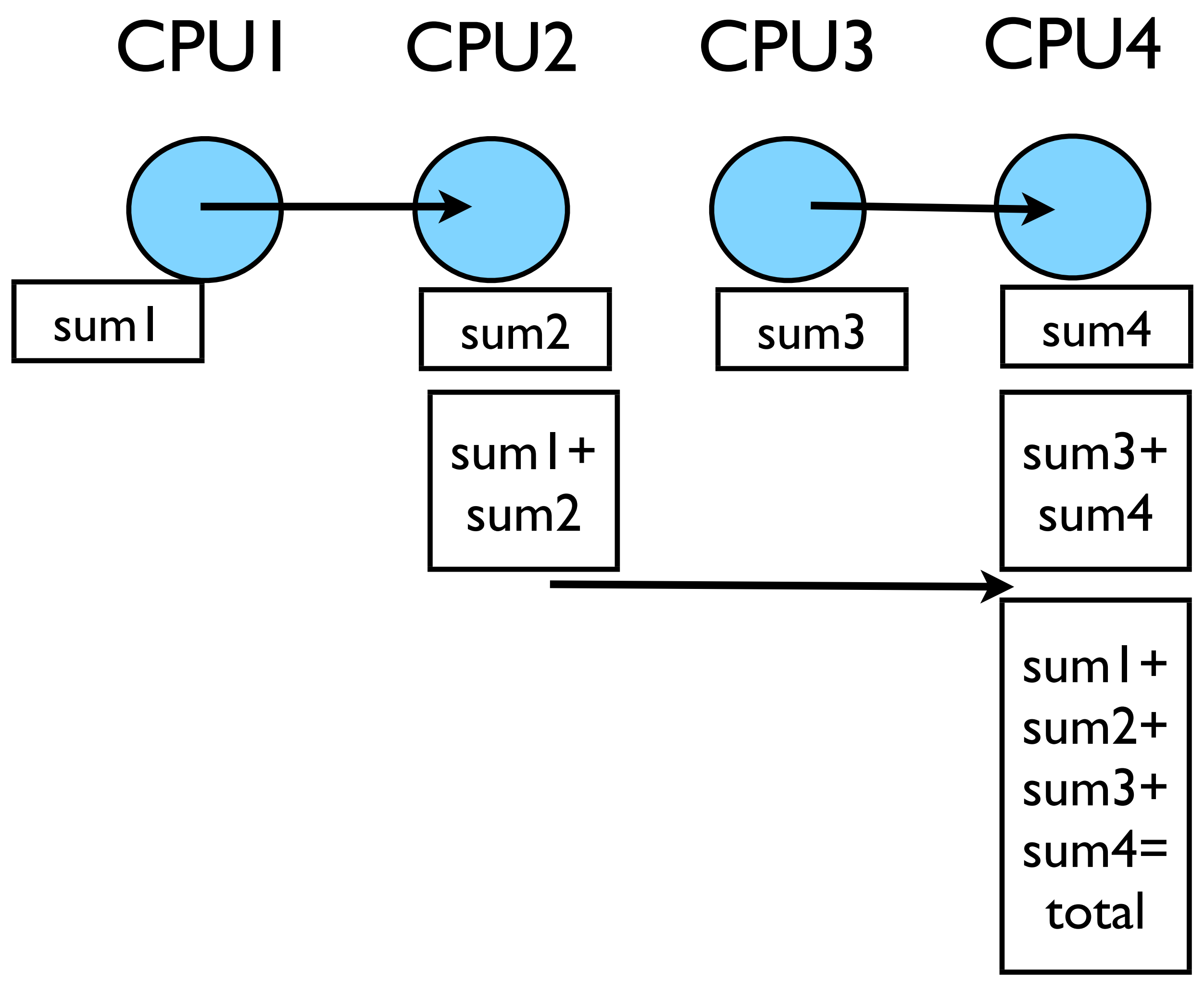
- Requires (P-1) messages, 2 (P-1) if everyone then needs to get the answer.



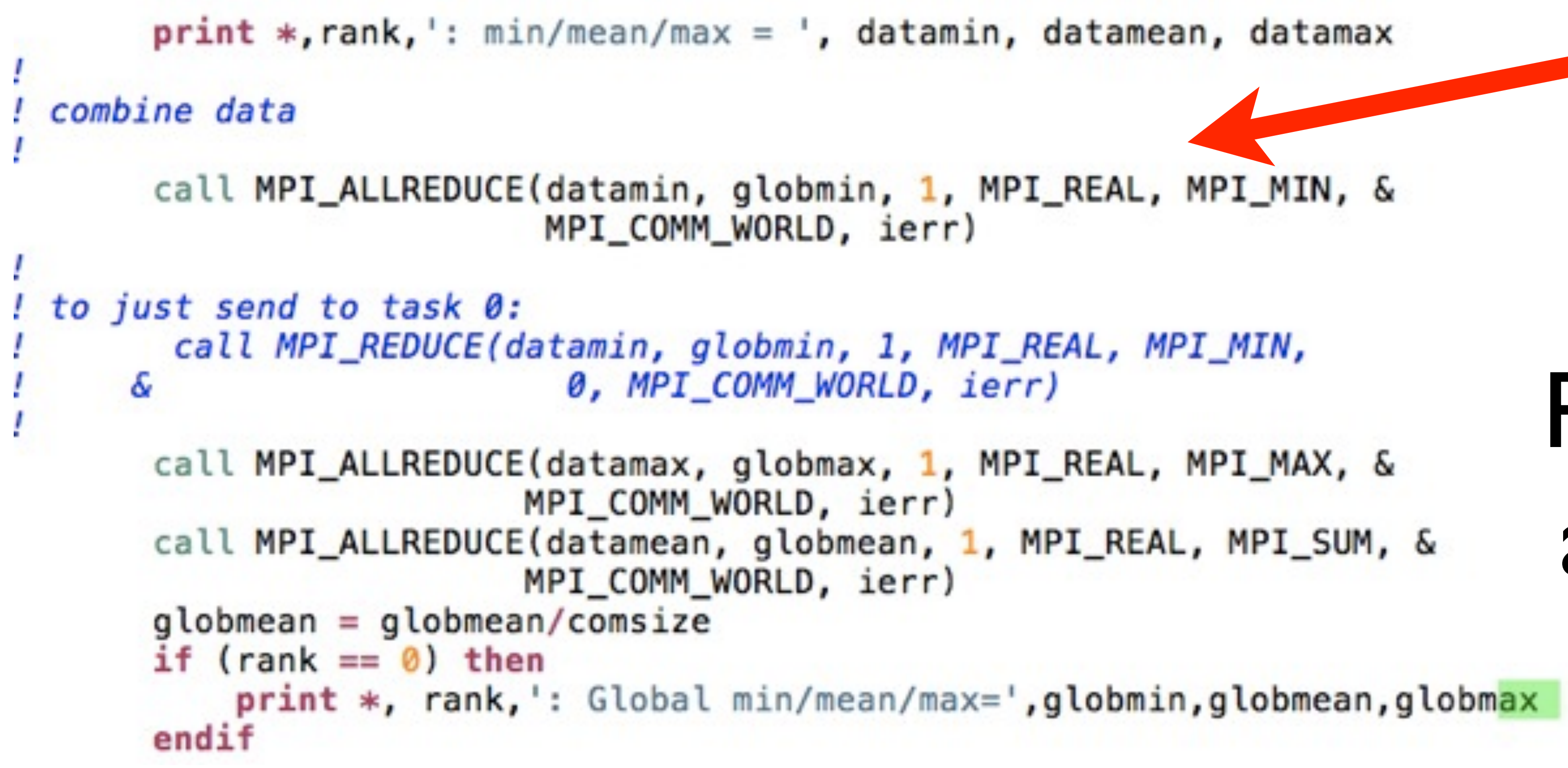
Better Summing

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

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



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
```

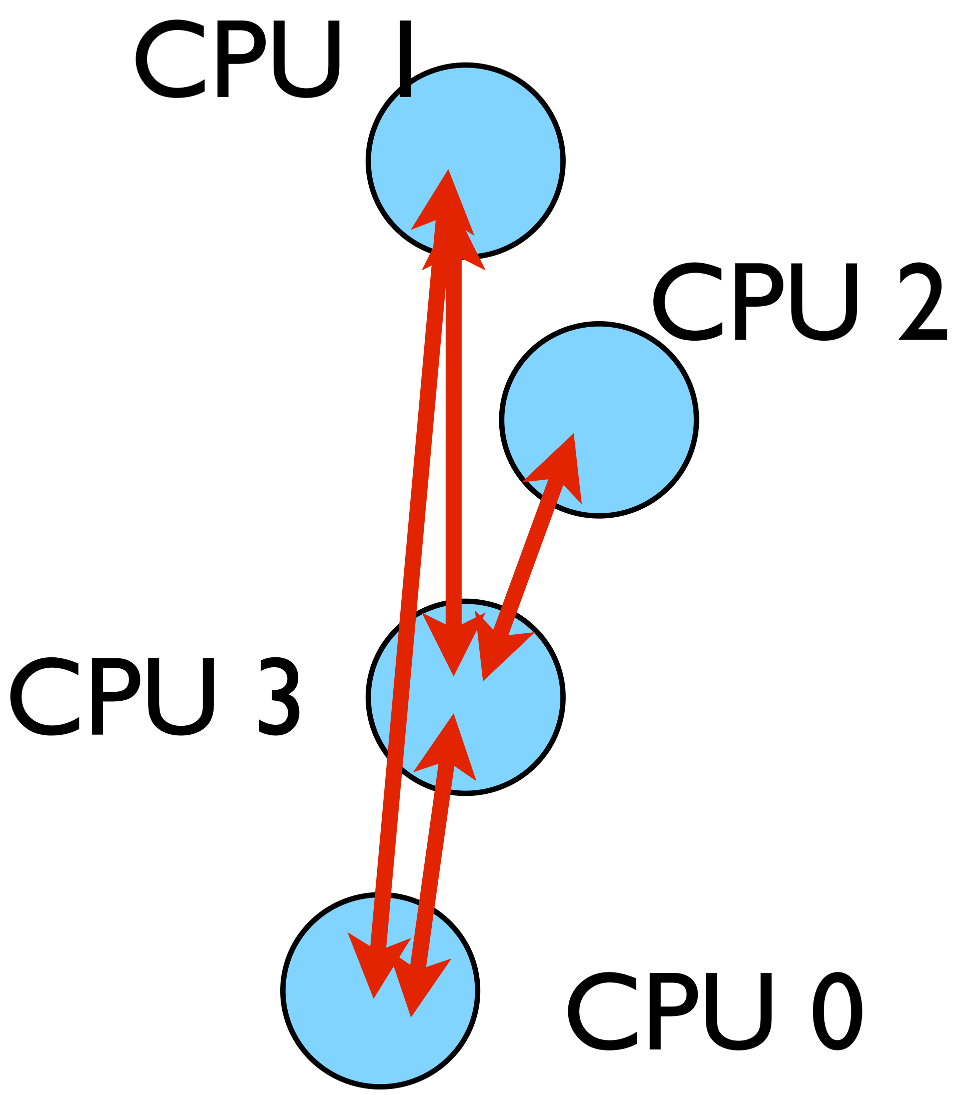
MPI_Reduce and MPI_Allreduce

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

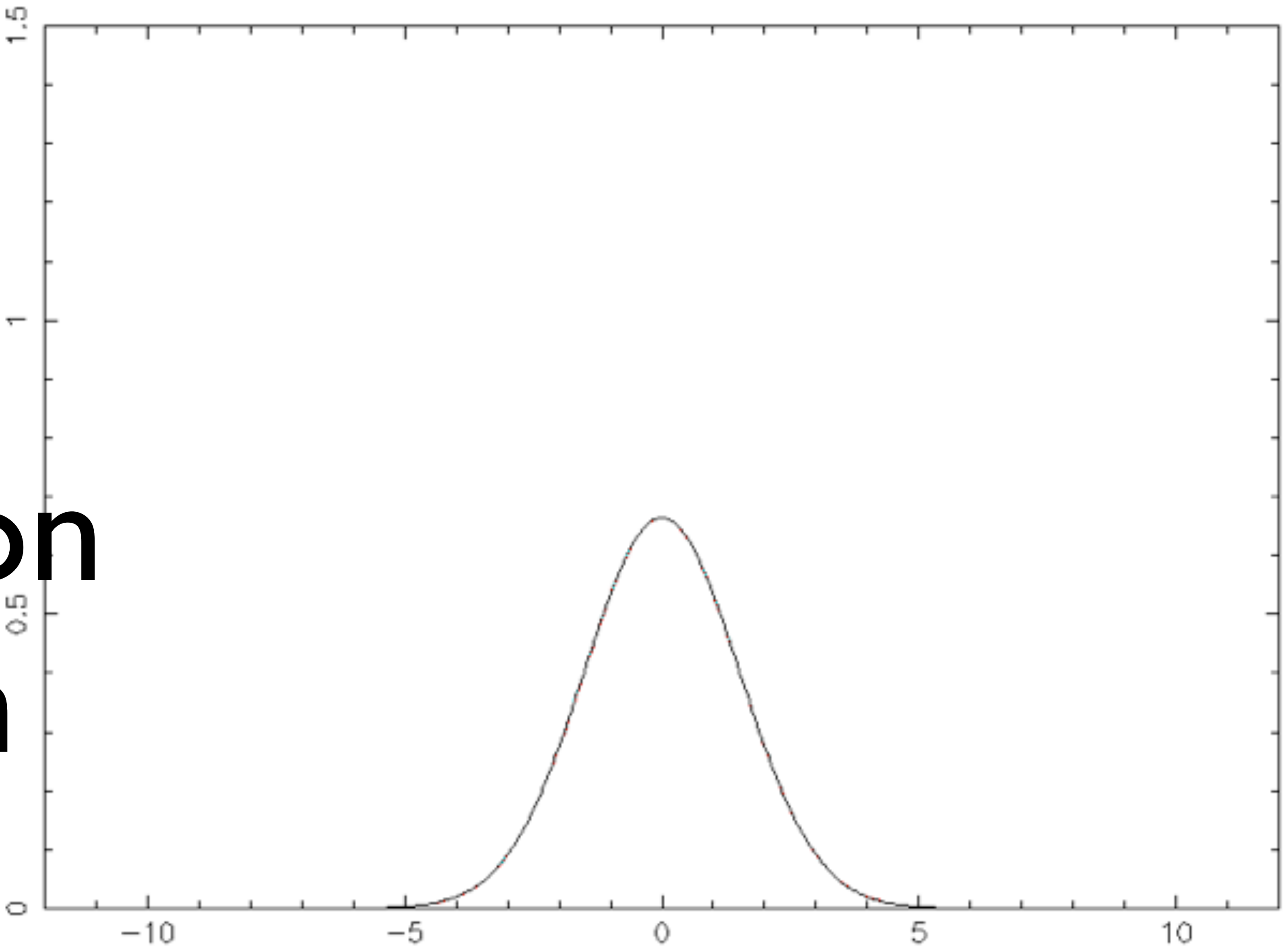
minmeanmax-allreduce.f

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

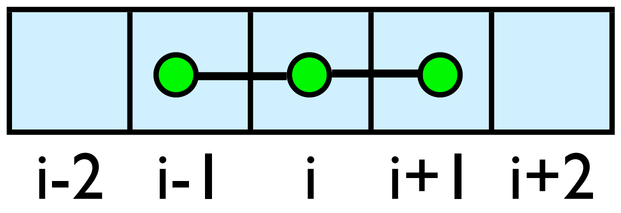


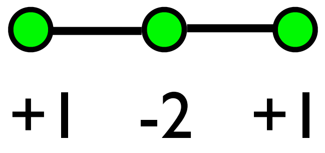
```
cp -R ~ljdursi/ppp/diffusion .
cd diffusion
make diffusionf or make diffusionc
./diffusionf or ./diffusionc
```

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'

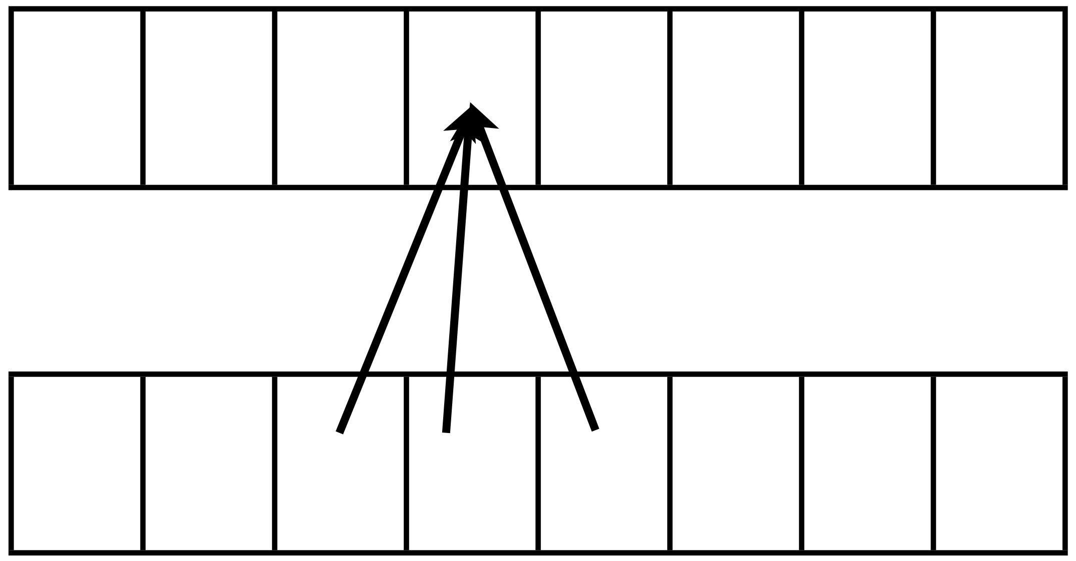




Diffusion Equation

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

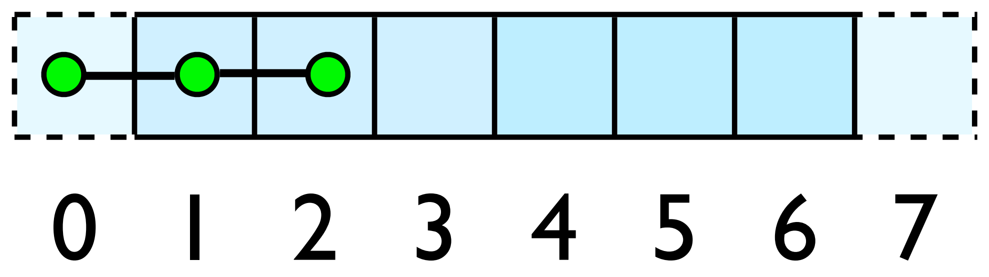
- Simple 1d PDE
- Each timestep, new data for T[i] requires old data for T[i+1],T[i],T[i-1]



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

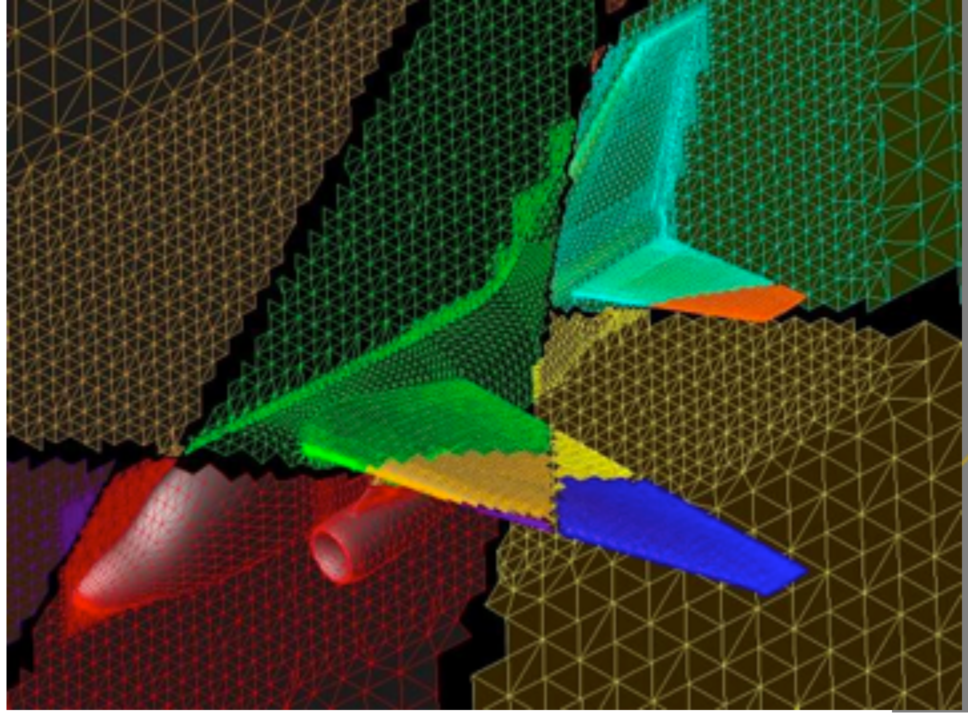


ng = 1

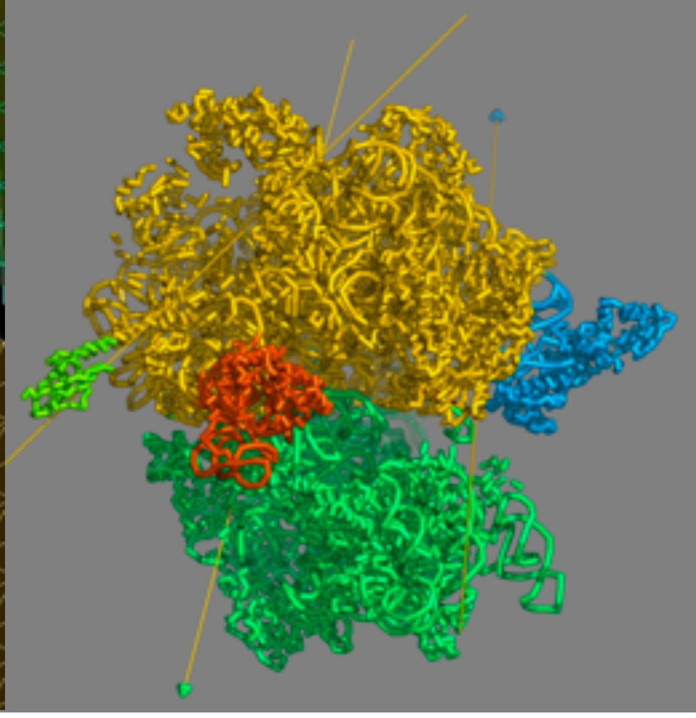
loop from ng, N - 2 ng

Domain Decomposition

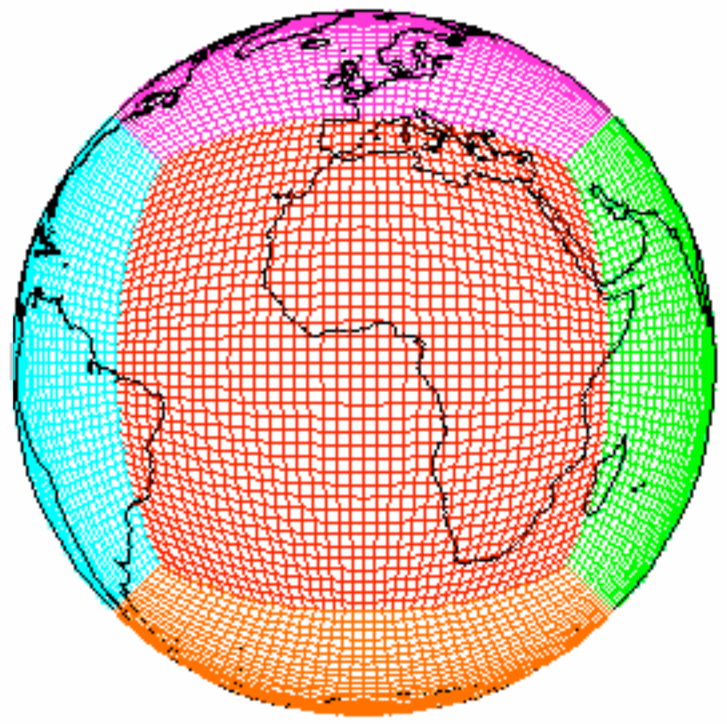
- A very common approach to parallelizing on distributed memory computers
- Maintain Locality; need local data mostly, this means only surface data needs to be sent between processes.



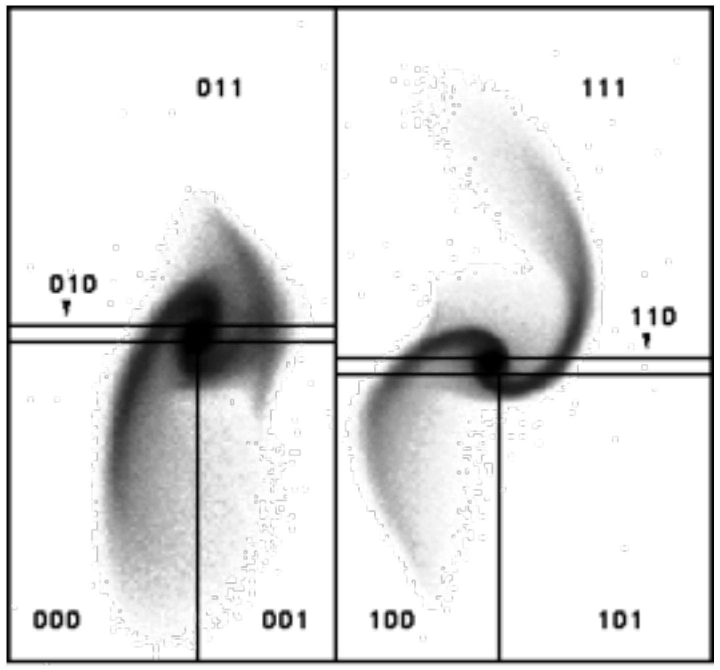
http://adg.stanford.edu/aa241 /design/compaero.html



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



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

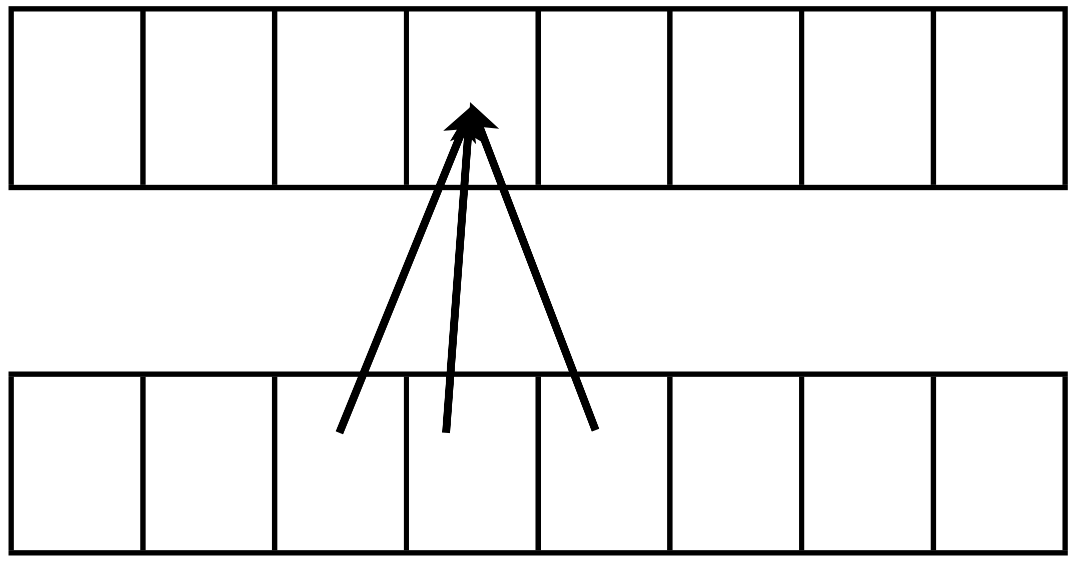


http://www.cita.utoronto.ca/~dubinski /treecode/node8.html

Implement a diffusion equation in MPI

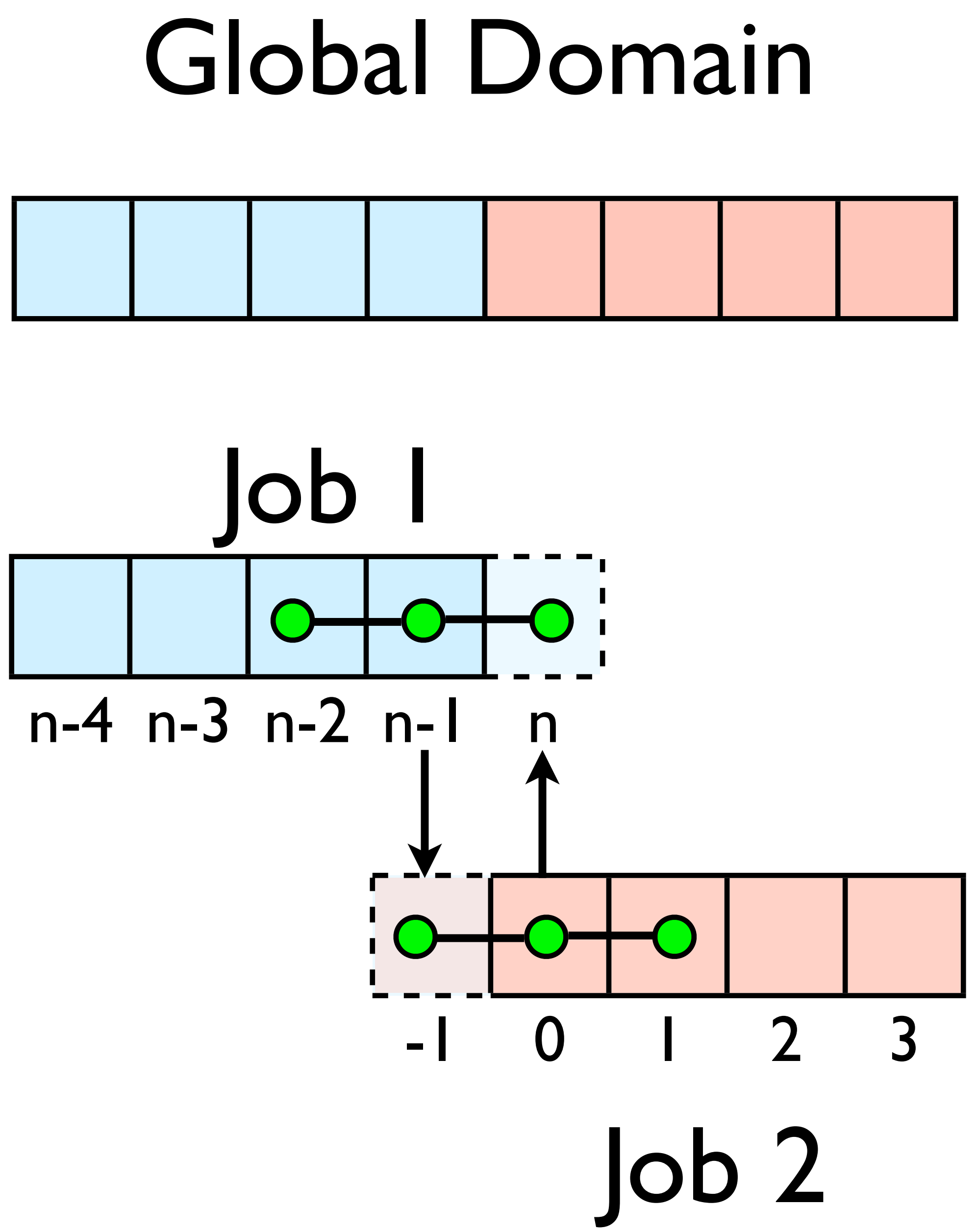
$$\begin{aligned} \frac{dT}{dt} &= D\frac{d^2T}{dx^2} \\ T_i^{n+1} &= T_i^n + \frac{D\Delta t}{\Delta x^2}\left(T_{i+1}^n - 2T_i^n + T_{i-1}^n\right) \end{aligned}$$

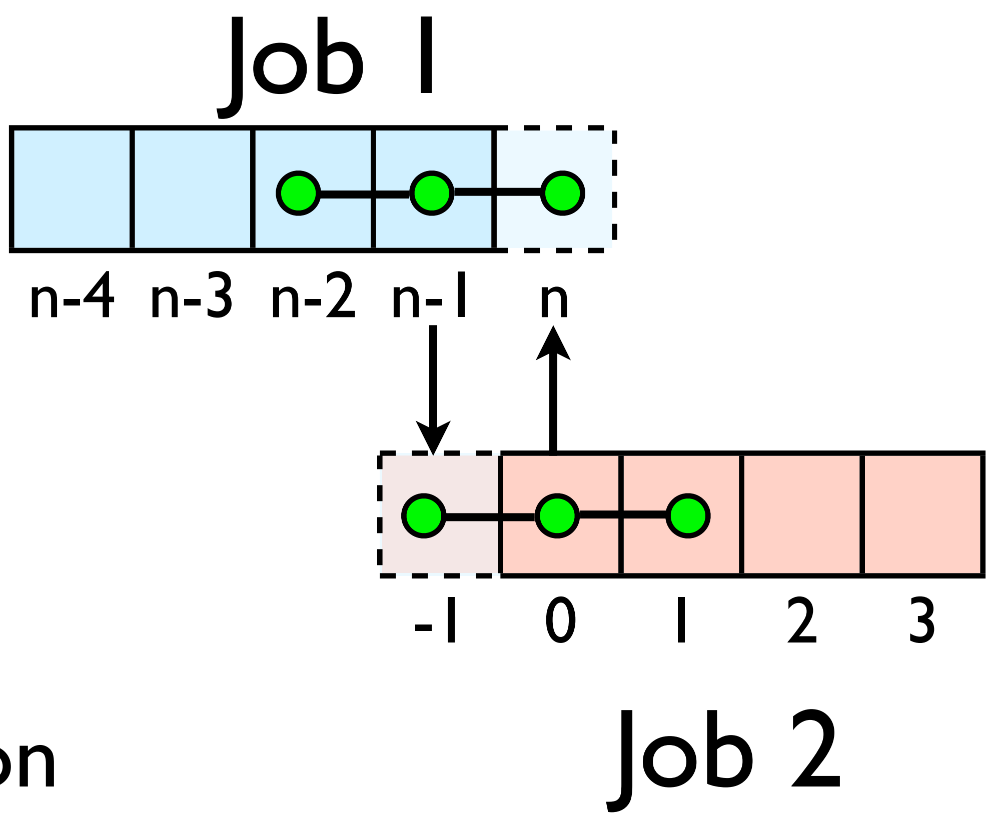
- Need one neighboring number per neighbor per timestep



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,...
```

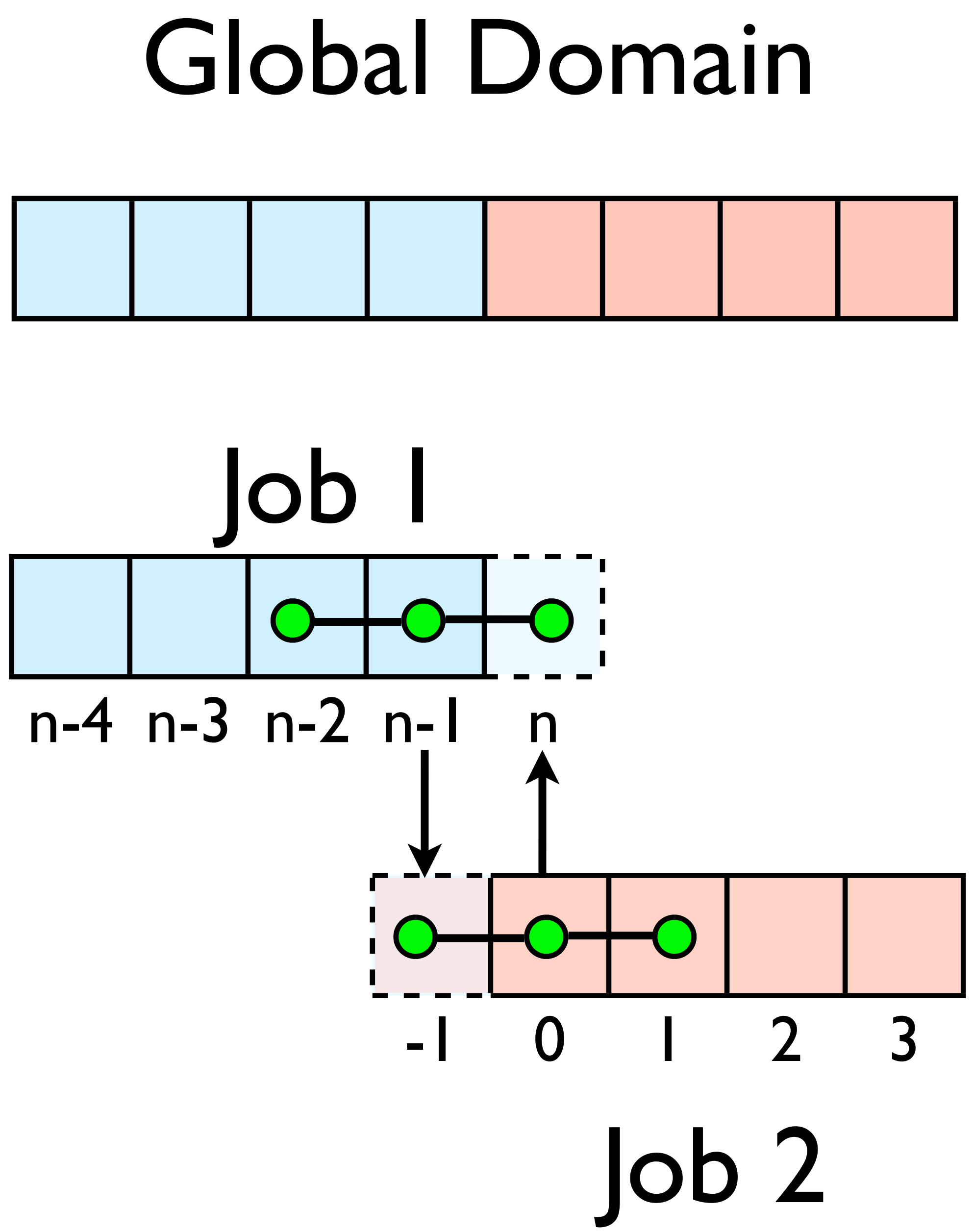
FORTRAN syntax

```
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,...
```

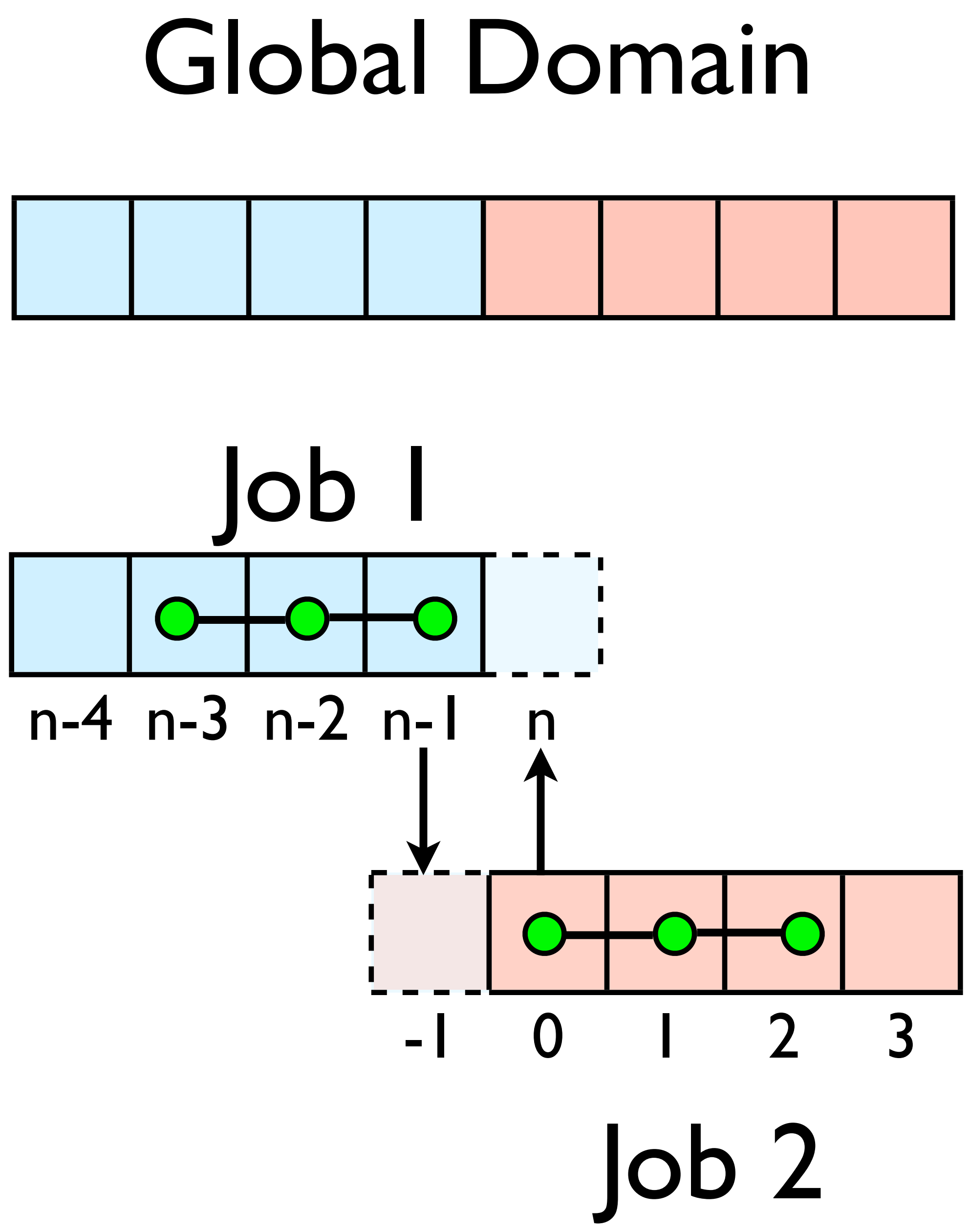
Non-blocking communications

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



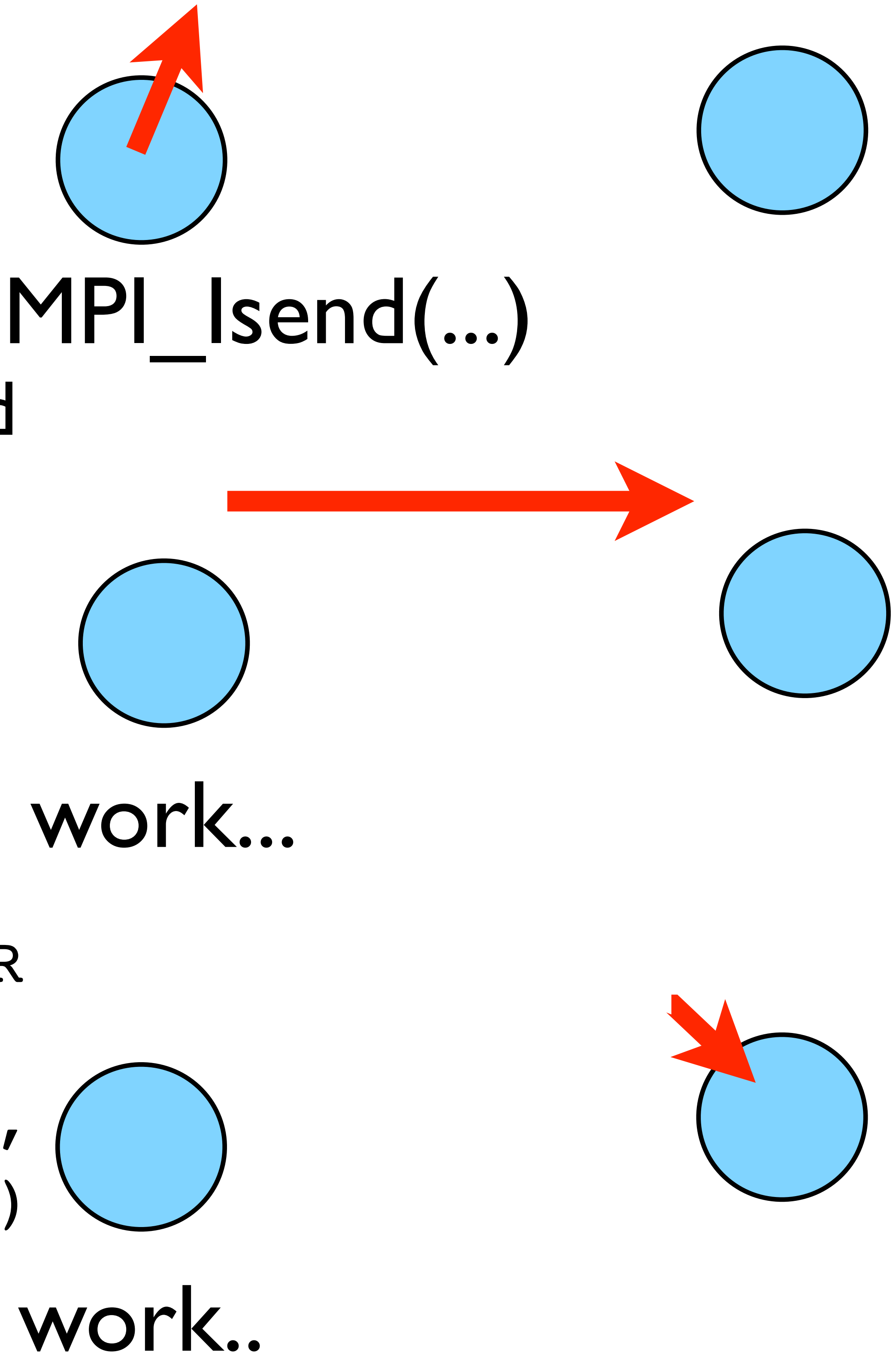
Diffusion: *Had* to wait?



- 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.

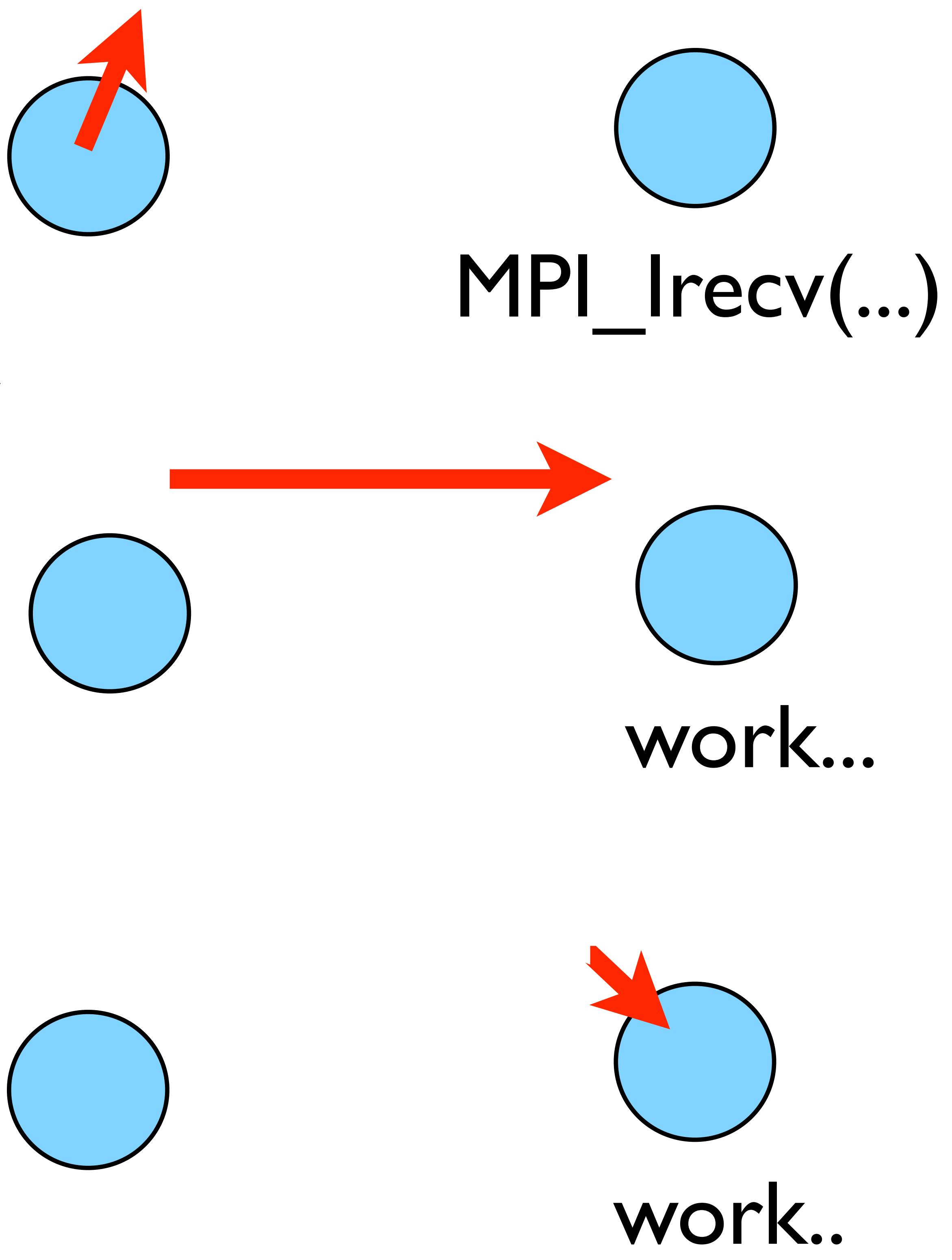
Nonblocking Sends

- 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)`

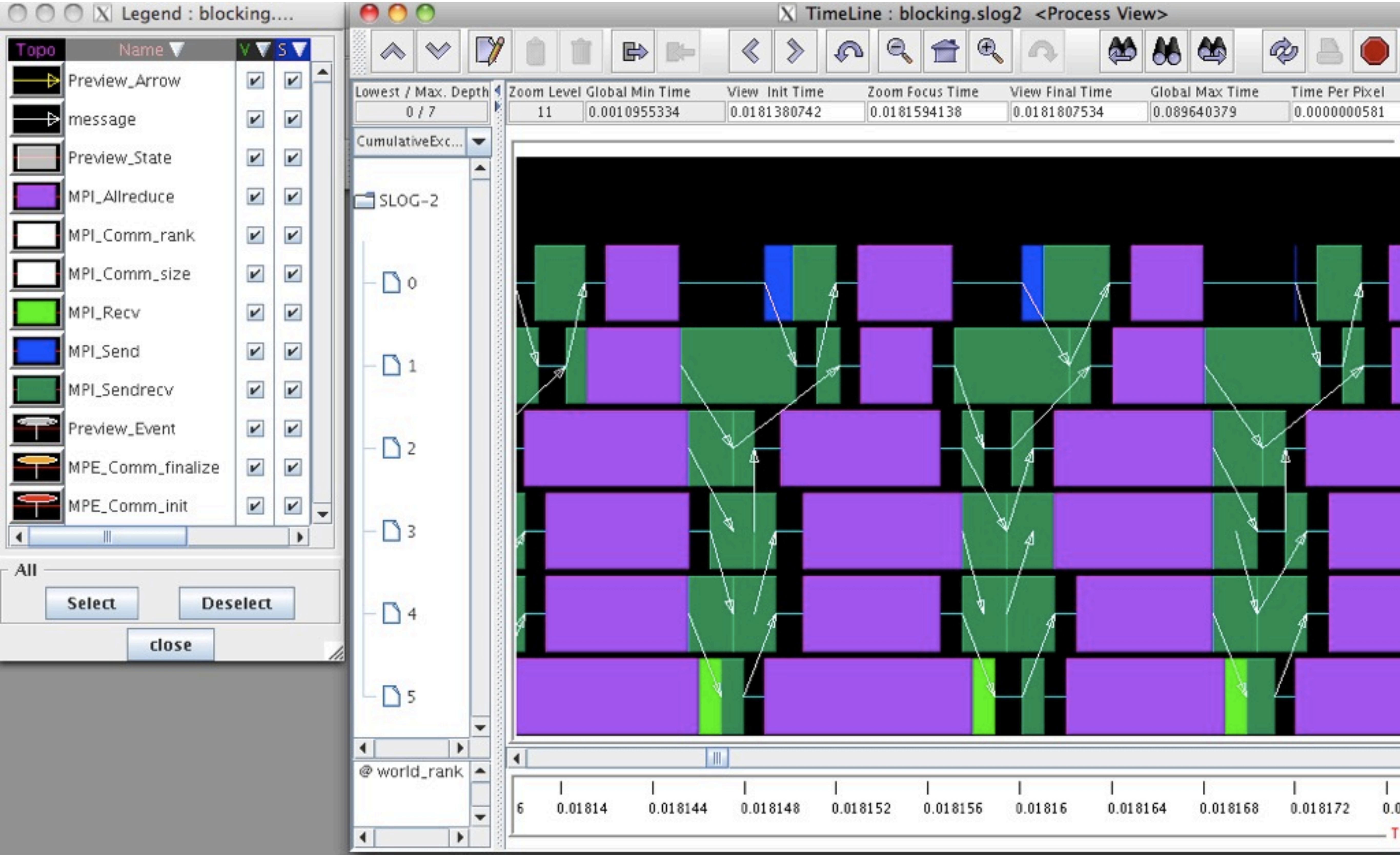


How to tell if message is completed?

- `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 IERROR)`

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	✓	✓

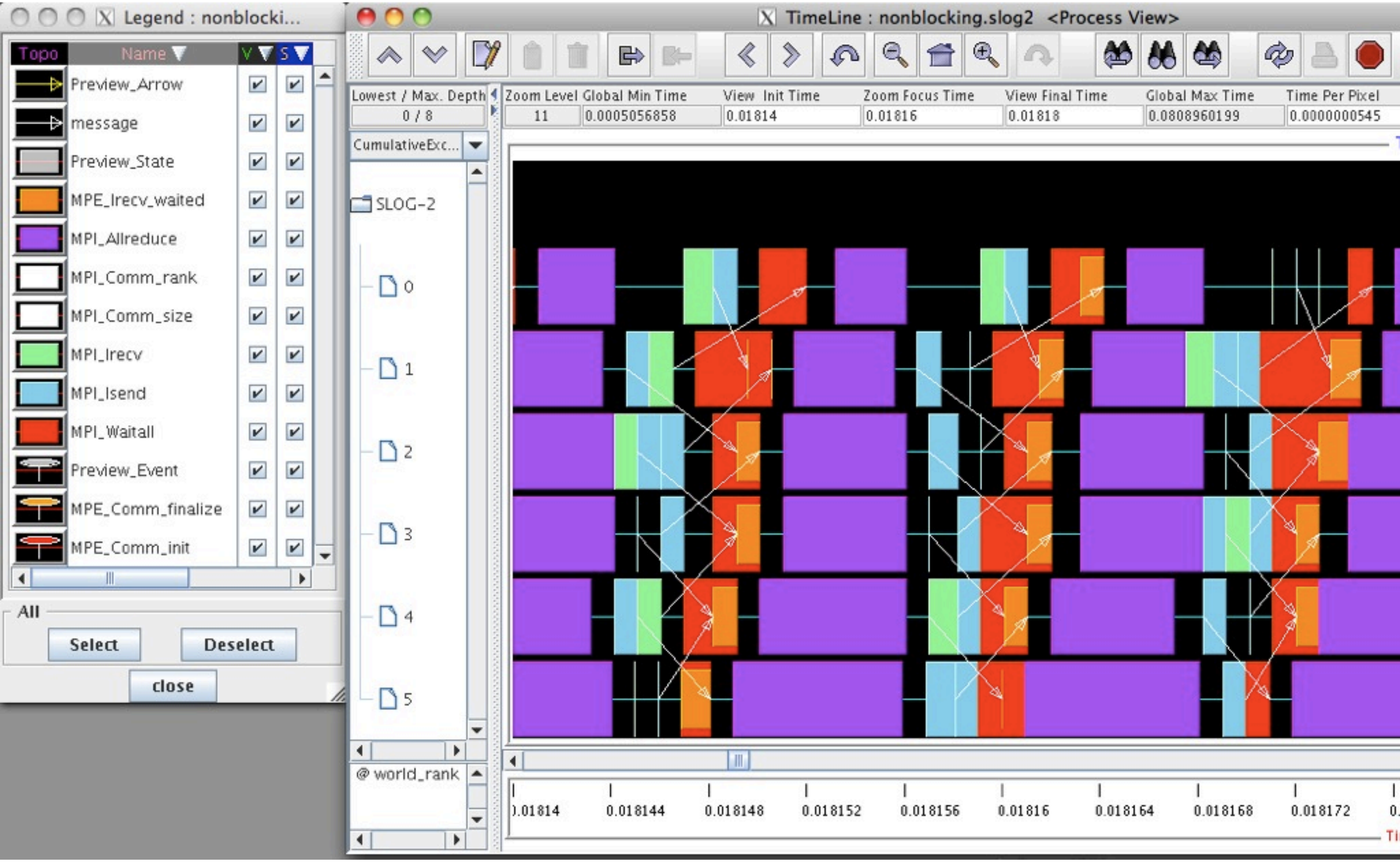
All

Select Deselect

close

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
0 / 8
Zoom Level
11
Global Min Time
0.0005056858
View Init Time
0.01814
Zoom Focus Time
0.01816
View Final Time
0.01818
Global Max Time
0.0808960199
Time Per Pixel
0.0000000545
CumulativeExc...
SLOG-2
0
1
2
3
4
5
@ world_rank
).01814
0.018144
0.018148
0.018152
0.018156
0.01816
0.018164
0.018168
0.018172

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