An introduction to MPI

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

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

C

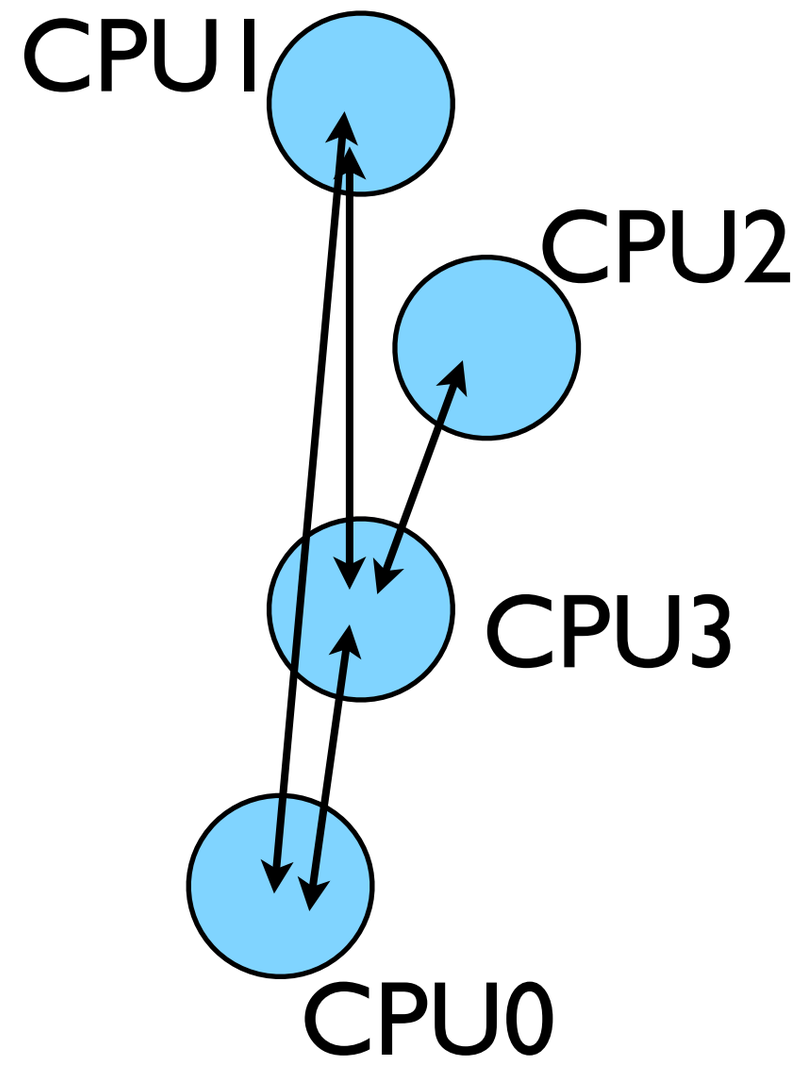
```c
#include <stdio.h>
#include <mpi.h>

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

        int rank, size;
        int ierr;

        ierr = MPI_Init(&argc, &argv);

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

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

        MPI_Finalize();

        return 0;
}
```

Fortran

```fortran
program hellompiworld
include "mpif.h"

integer rank, size
integer ierr

call MPI_INIT(ierr)

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

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

call MPI_FINALIZE(ierr)

return
end
```

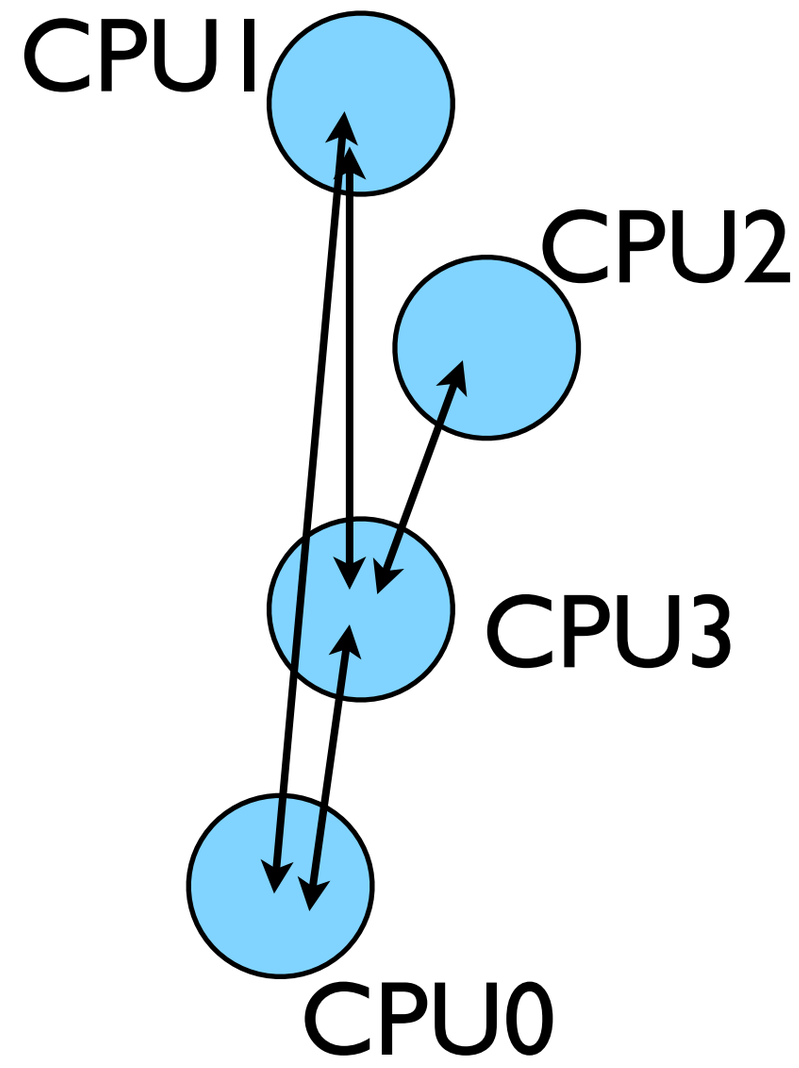
MPI is a Library for **Message-Passing**

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



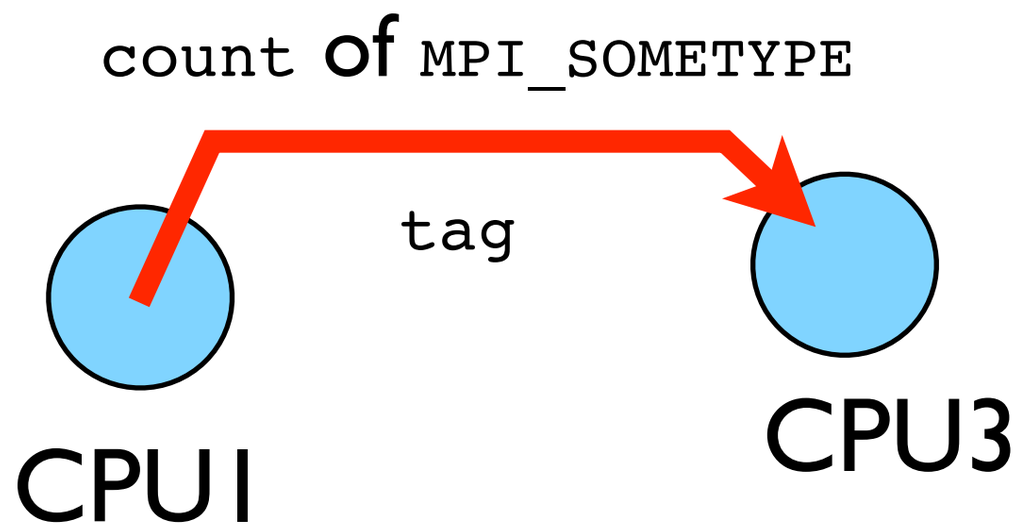
MPI is a Library for **Message-Passing**

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



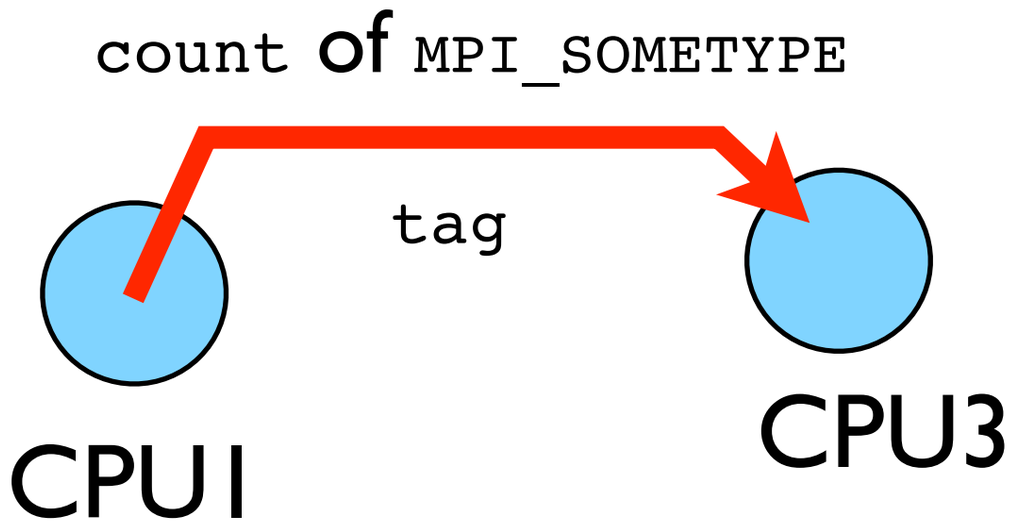
Messages

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



Messages

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



Size of MPI Library

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

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

Hello World

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

Fortran

```
program hellompiworld
include "mpif.h"

integer :: rank, comsize
integer :: ierr

call MPI_INIT(ierr)

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

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

call MPI_FINALIZE(ierr)

return
end
```

C

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

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

        int rank, size;
        int ierr;

        ierr = MPI_Init(&argc, &argv);

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

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

        MPI_Finalize();

        return 0;
}
```

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
```

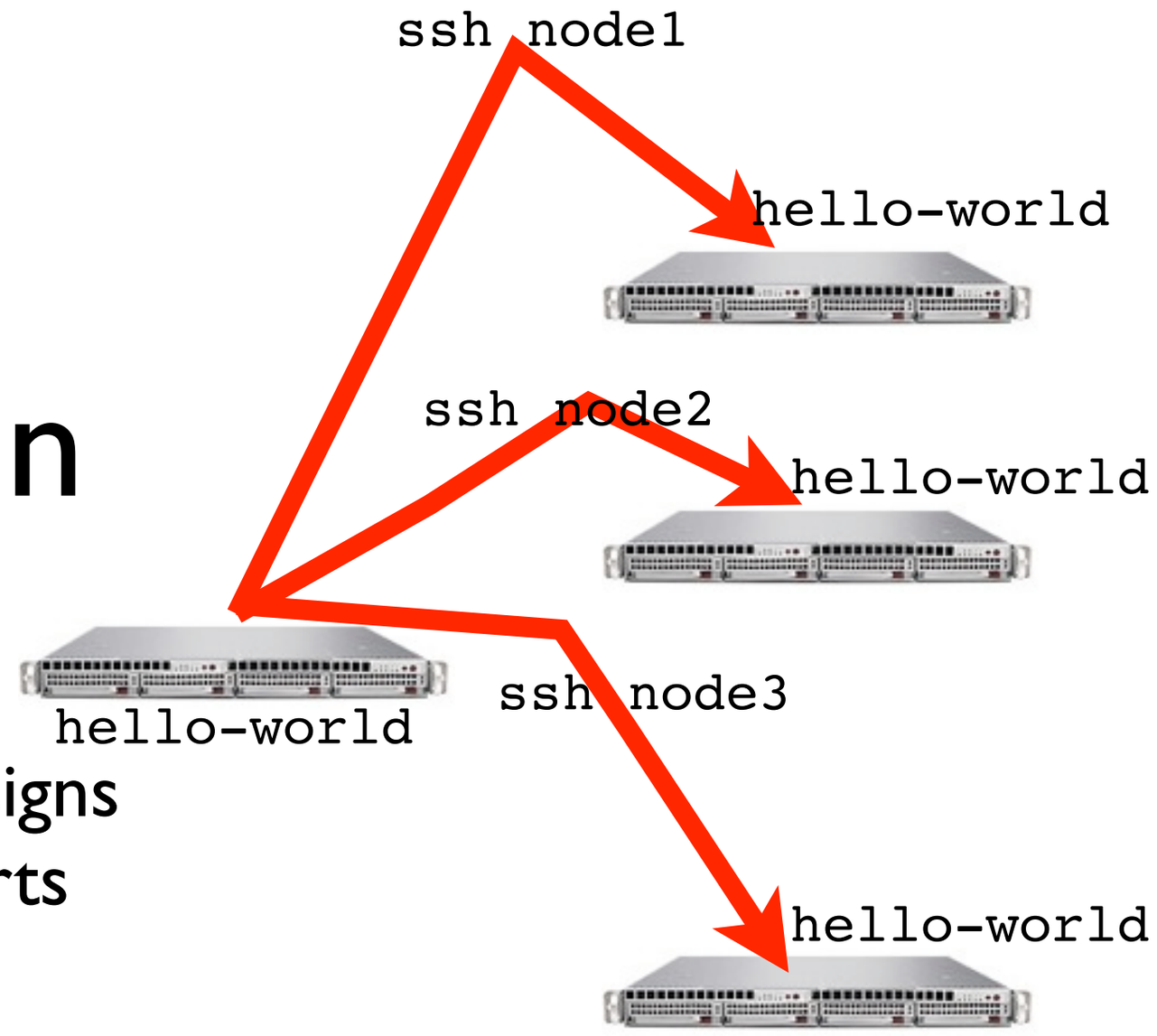
What mpicc/ mpif77 do

- Just wrappers for the system C, Fortran compilers that have the various -I, -L clauses in there automaticaly
- --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
```

What the code does

```
program hellompiworld
include "mpif.h"

integer :: rank, comsize
integer :: ierr

call MPI_INIT(ierr)

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

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

call MPI_FINALIZE(ierr)

return
end
```

- (FORTRAN version; C is similar)

`include “mpif.h”`: imports declarations for MPI function calls

```
program hellompiworld
include "mpif.h"

integer :: rank, comsize
integer :: ierr

call MPI_INIT(ierr)

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

print *, "Hello from task ",rank," of ", comsize,

call MPI_FINALIZE(ierr)

return
end
```

`call MPI_INIT(ierr)`:
initialization for MPI library.
Must come first.
`ierr:` Returns any error code.

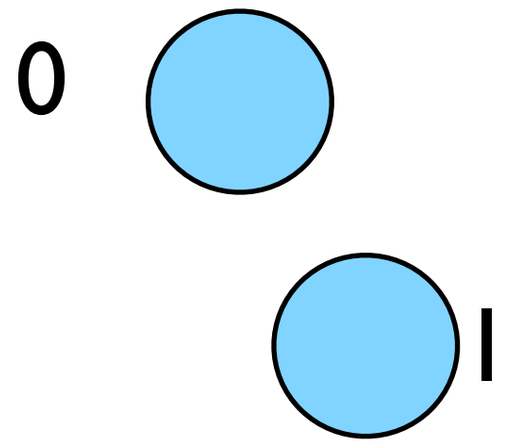
`call MPI_FINALIZE(ierr)`:
close up MPI stuff.
Must come last.
`ierr:` Returns any error code.

```
program hellompiworld
include "mpif.h"

integer :: rank, comsize
integer :: ierr

call MPI_INIT(ierr)

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

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

call MPI_FINALIZE(ierr)

return
end
```

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

Communicators

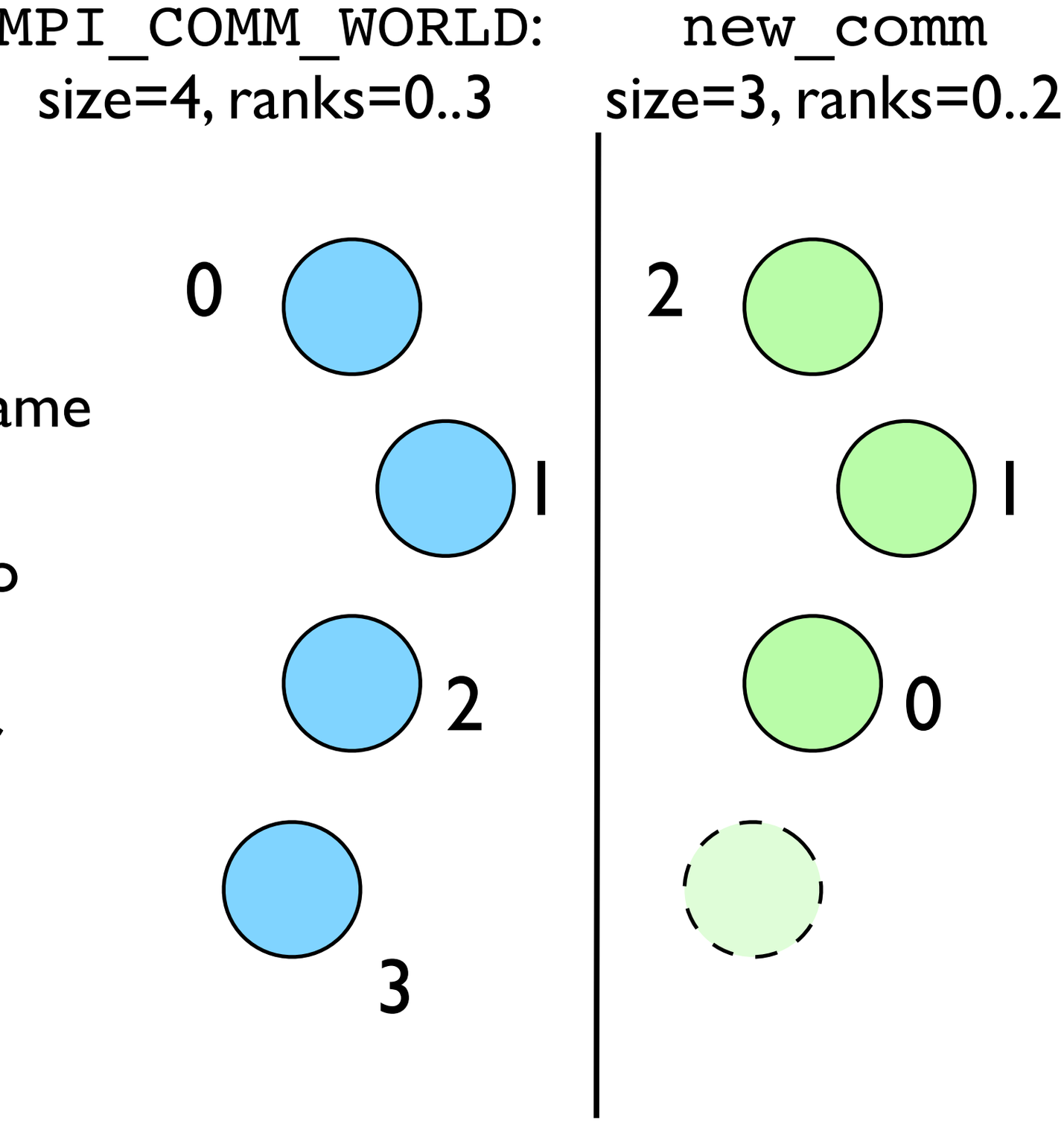
- MPI groups processes into communicators.
- Each communicator has some size -- number of tasks.
- Each task has a rank 0..size-1
- Every task in your program belongs to `MPI_COMM_WORLD`



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

Communicators

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



```
program hellompiworld
include "mpif.h"

integer :: rank, comsize
integer :: ierr

call MPI_INIT(ierr)

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

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

call MPI_FINALIZE(ierr)

return
end
```

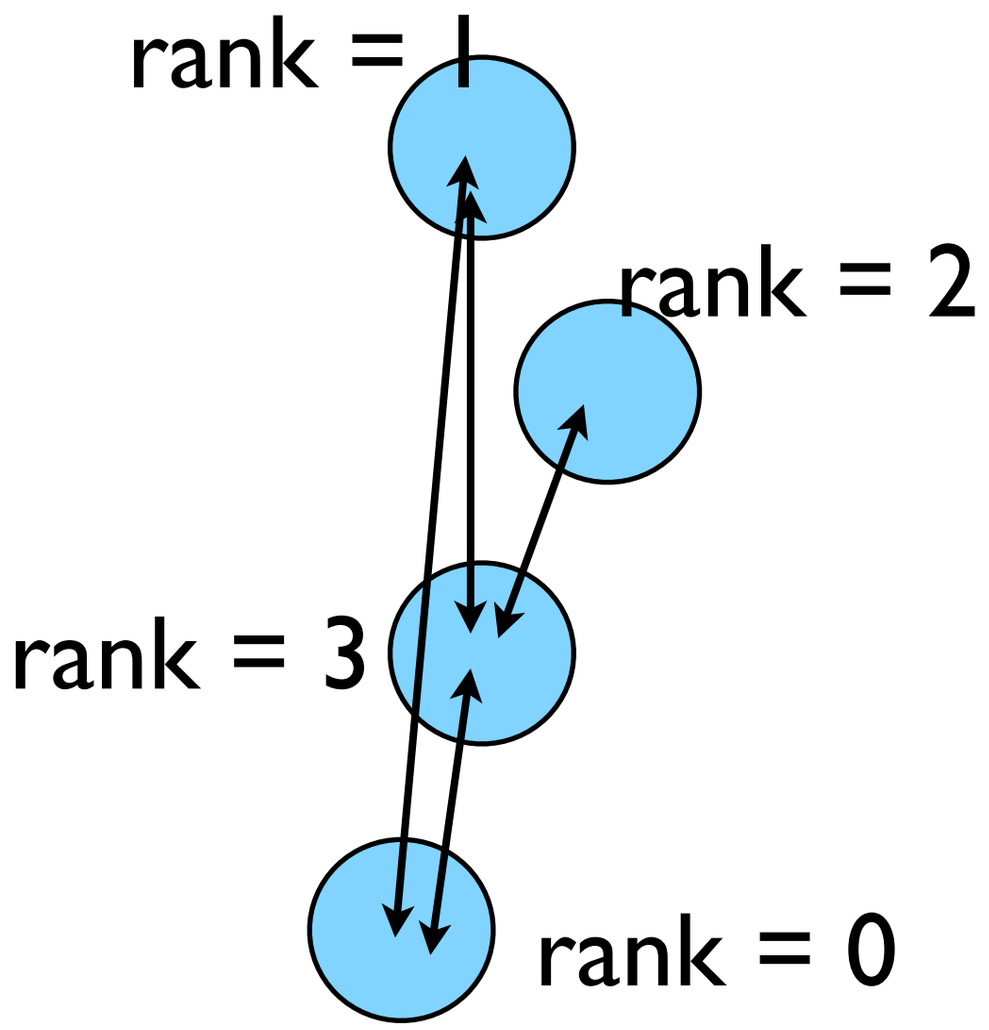
`call MPI_COMM_RANK,`
`call MPI_COMM_SIZE:`

get the size of communicator, the current tasks's rank within 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;
        int ierr;

        ierr = MPI_Init(&argc, &argv);

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

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

        MPI_Finalize();

        return 0;
```

Fortran

```fortran
program hellompiworld
include "mpif.h"

integer :: rank, comsize
integer :: ierr

call MPI_INIT(ierr)

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

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

call MPI_FINALIZE(ierr)

return
end
```

- Fortran: All caps (convention)
- C - functions **return** ierr;
- Fortran - **pass** ierr
- MPI_Init

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

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

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

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

    int rank, size;          /* the usual MPI stuff */
    int ierr;
    char hearmessage[6];     /* we recieve into here, and */
    char sendmessage[]="Hello"; /* send from here.*/
    int sendto;              /* PE # we send to */
    int recvfrom;            /* PE # we recv from */
    const int OURTAG=1;      /* shared tag to label messages */
    MPI_Status status;       /* recieve status info */

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

    if (size < 2) {
        fprintf(stderr,"FAIL: only one task\n");
        MPI_Abort(MPI_COMM_WORLD,1);
    }

    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);
    }

    if (rank == 1) {
        recvfrom = 0;
        ierr = MPI_Recv(hearmessage, 6, MPI_CHAR, recvfrom,
                         OURTAG, MPI_COMM_WORLD, &status);
        printf("%d: Recieved message <%s>\n", rank, hearmessage);
    }

    MPI_Finalize();

    return 0;
}
```

Fortran version

- Let's fix this
- cp hello-world.f firstmessage.f90
- mpif77 -o firstmessage firstmessage.f90
- mpirun -np 2 ./ firstmessage
- FORTRAN - MPI_CHARACTER

```
1         program hellompiworld
2         implicit none
3         include "mpif.h"
4
5         integer :: rank, comsize      ! standard MPI stuff
6         integer :: ierr
7         integer :: sendto, recvfrom   ! PE # to send, recv from
8         integer,parameter :: ourtag=1 ! shared label for messages
9         character(5) :: sendmessage   ! buffer for sending, recieving
10        character(5) :: hearmessage   ! messages
11        integer, dimension(MPI_STATUS_SIZE) :: status  ! rcv status
12
13        call MPI_INIT(ierr)
14        call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
15        call MPI_COMM_SIZE(MPI_COMM_WORLD, comsize, ierr)
16
17        if (comsize .le. 1) then      ! need at least a sender, reciever
18            print *,' FAIL: only one task'
19            call MPI_Abort(MPI_COMM_WORLD,1)
20        endif
21
22        if (rank == 0) then
23            sendmessage = 'Hello'
24            sendto = 1
25            call MPI_SSEND(sendmessage, 5, MPI_CHARACTER, sendto, &
26                           ourtag, MPI_COMM_WORLD, ierr)
27            print *, rank, ': sent message <',sendmessage,'>.'
28        else if (rank == 1) then
29            recvfrom = 0
30            call MPI_RECV(hearmessage, 5, MPI_CHARACTER, recvfrom, &
31                          ourtag, MPI_COMM_WORLD, status, ierr)
32            print *, rank, ': got message <',hearmessage,'>.'
33        endif
34
35        call MPI_FINALIZE(ierr)
36
37        end
38
```

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:

- cp firstmessage.{c,f90} secondmessage.{c,f90}

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

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

    int rank, size;         /* the usual MPI stuff */
    int ierr;
    char hearmessage[6];    /* we recieve into here, and */
    char sendmessage[]="Hello"; /* send from here.*/
    int leftneighbour, rightneighbour;
    const int OURTAG=1;     /* shared tag to label messages */
    MPI_Status status;      /* recieve status info */

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

    if (size < 2) {         /* need at least a sender, reciever */
        fprintf(stderr,"FAIL: only one task\n");
        MPI_Abort(MPI_COMM_WORLD,1);
    }

    leftneighbour = rank-1;
    rightneighbour = rank+1;

    if (rightneighbour < size) {
        ierr = MPI_Ssend(sendmessage, 6, MPI_CHAR, rightneighbour,
                         OURTAG, MPI_COMM_WORLD);
        printf("%d: Sent message <%s> to %d \n", rank, sendmessage, rightneighbour);
    }
    if (leftneighbour >= 0) {
        ierr = MPI_Recv(hearmessage, 6, MPI_CHAR, leftneighbour,
                         OURTAG, MPI_COMM_WORLD, &status);
        printf("%d: Recieved message <%s> from %d\n", rank, hearmessage, leftneighbour);
    }

    MPI_Finalize();

    return 0;
}
```

More complicated example:

- cp firstmessage.{c,f90} secondmessage.{c,f90}

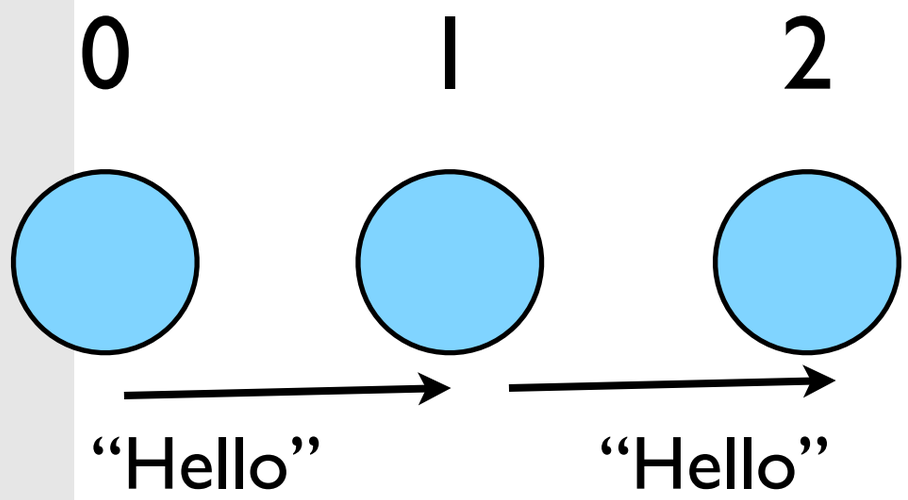
```
1       program hellompiworld
2       implicit none
3       include "mpif.h"
4
5       integer :: rank, comsize      ! standard MPI stuff
6       integer :: ierr
7       integer :: leftneighbour, rightneighbour
8       integer,parameter :: ourtag=1 ! shared label for messages
9       character(5) :: sendmessage   ! buffer for sending, recieving
10      character(5) :: hearmessage   ! messages
11      integer, dimension(MPI_STATUS_SIZE) :: status  ! rcv status
12
13      call MPI_INIT(ierr)
14      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
15      call MPI_COMM_SIZE(MPI_COMM_WORLD, comsize, ierr)
16
17      if (comsize .le. 1) then      ! need at least a sender, reciever
18          print *,' FAIL: only one task'
19          call MPI_Abort(MPI_COMM_WORLD,1)
20      endif
21
22      leftneighbour  = rank - 1
23      rightneighbour = rank + 1
24
25      if (rightneighbour < comsize) then
26          sendmessage = 'Hello'
27          call MPI_SSEND(sendmessage, 5, MPI_CHARACTER, rightneighbour, &
28                         ourtag, MPI_COMM_WORLD, ierr)
29          print *, rank, ': sent message <',sendmessage,'> to ', rightneighbour
30      endif
31      if (leftneighbour >= 0) then
32          call MPI_RECV(hearmessage, 5, MPI_CHARACTER, leftneighbour, &
33                        ourtag, MPI_COMM_WORLD, status, ierr)
34          print *, rank, ': got message <',hearmessage,'> from ', leftneighbour
35      endif
36
37      call MPI_FINALIZE(ierr)
38
39      end
```

Compile and run

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

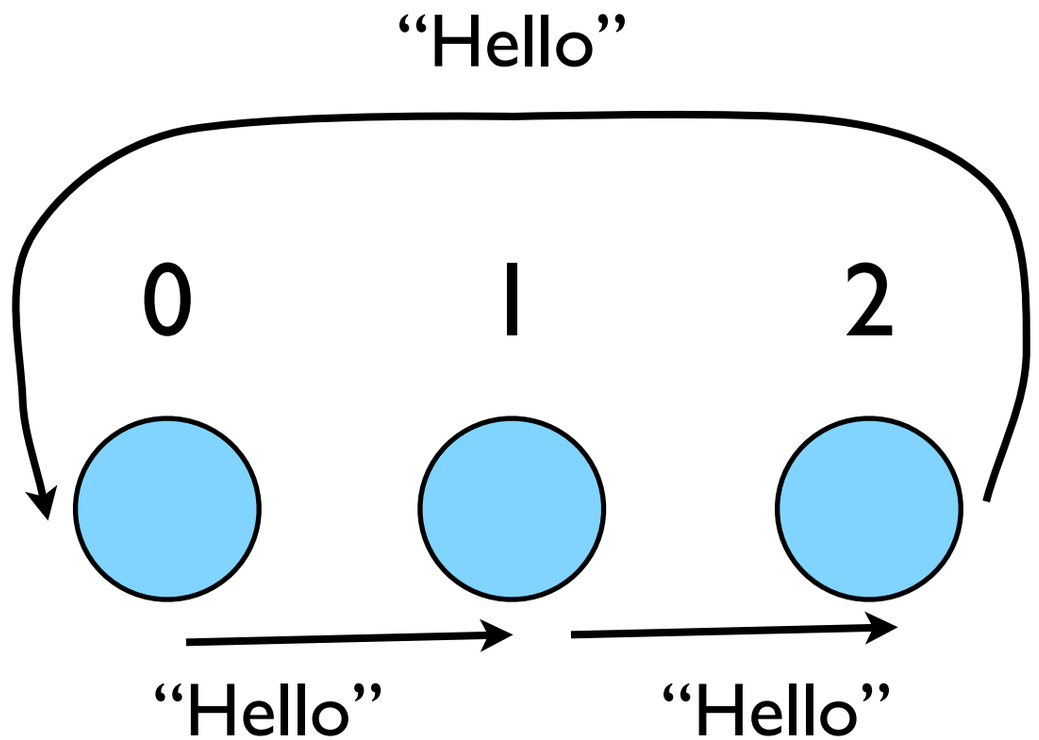
```
ljdursi|segfault.local> mpirun -np 4 ./secondmessage
            3 : got message <Hello>.
            2 : sent message <Hello>.
            2 : got message <Hello>.
            1 : sent message <Hello>.
            0 : sent message <Hello>.
            1 : got message <Hello>.
```

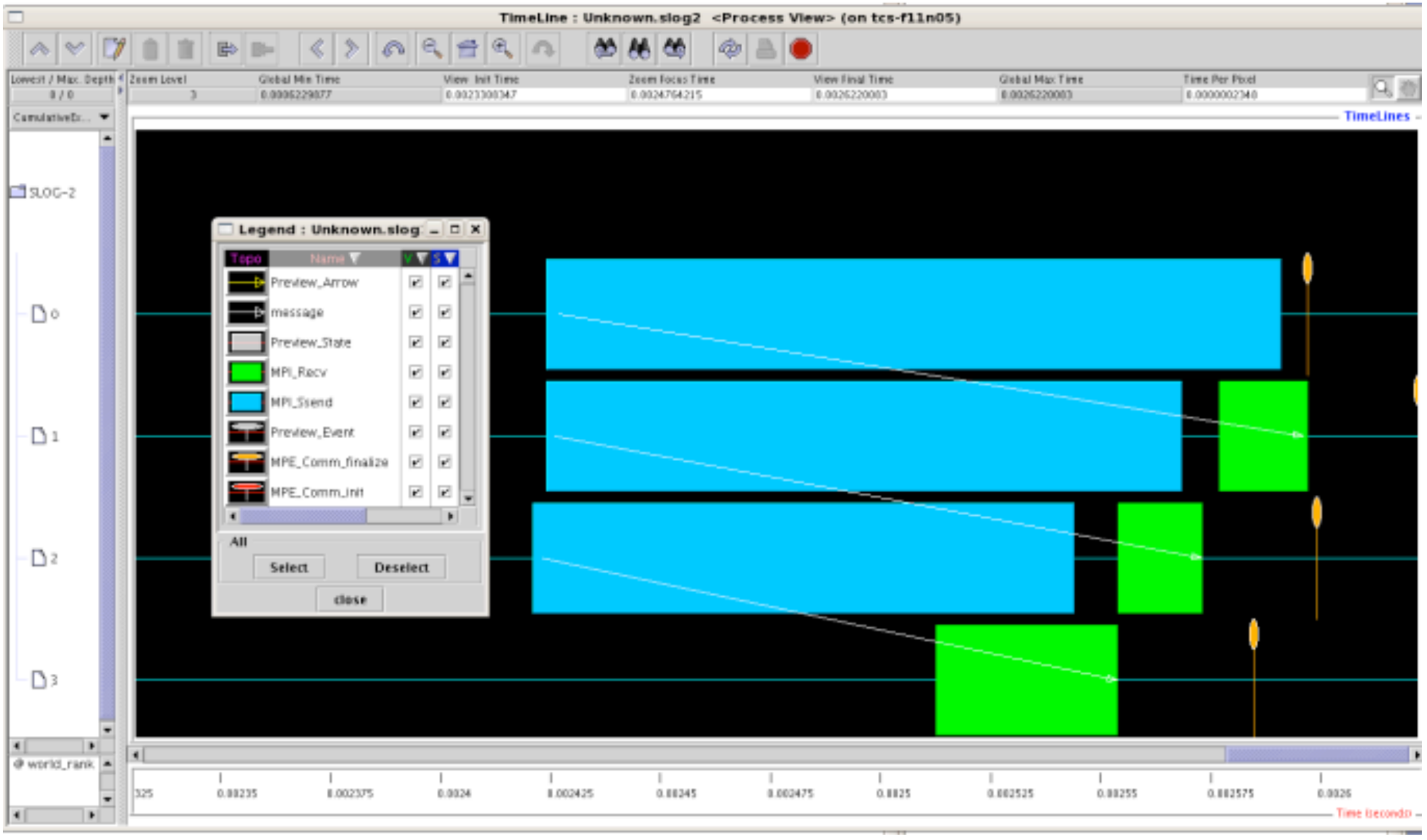
```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <mpi.h>

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

    int rank, size;          /* the usual MPI stuff */
    int ierr;
    char hearmessage[6];     /* we recieve into here, and */
    char sendmessage[]="Hello"; /* send from here.*/
    int leftneighbour, rightneighbour;
    const int OURTAG=1;      /* shared tag to label messages */
    MPI_Status status;       /* recieve status info */

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

    if (size < 2) {          /* need at least a sender, reciever */
        fprintf(stderr,"FAIL: only one task\n");
        MPI_Abort(MPI_COMM_WORLD,1);
    }

    leftneighbour = rank-1;
    rightneighbour = rank+1;

    if (rightneighbour < size) {
        ierr = MPI_Ssend(sendmessage, 6, MPI_CHAR, rightneighbour,
                         OURTAG, MPI_COMM_WORLD);
        printf("%d: Sent message <%s> to %d \n", rank, sendmessage, rightneighbour);
    }
    if (leftneighbour >= 0) {
        ierr = MPI_Recv(hearmessage, 6, MPI_CHAR, leftneighbour,
                         OURTAG, MPI_COMM_WORLD, &status);
        printf("%d: Recieved message <%s> from %d\n", rank, hearmessage, leftneighbour);
    }

    MPI_Finalize();

    return 0;
}
```



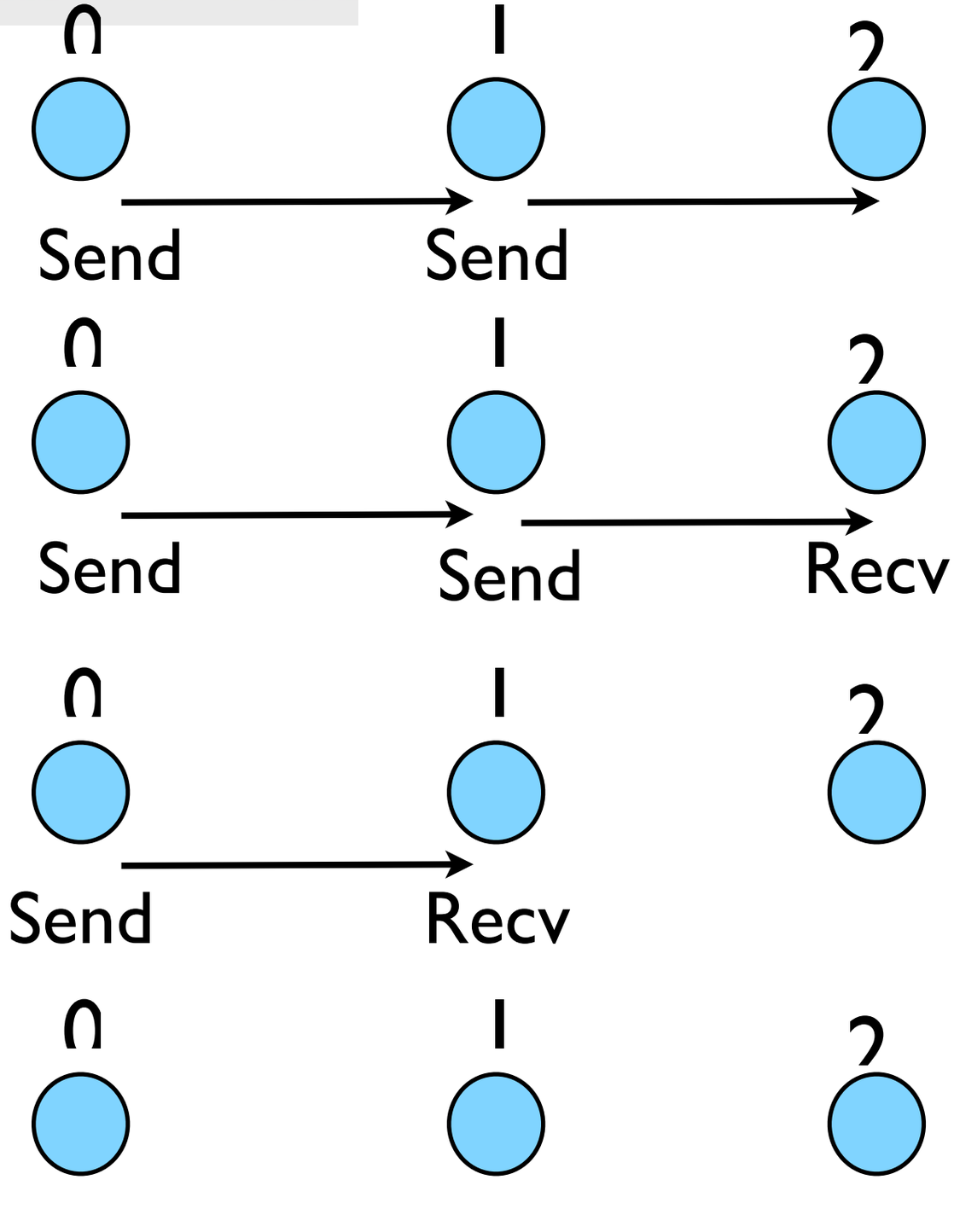
Implement periodic boundary conditions

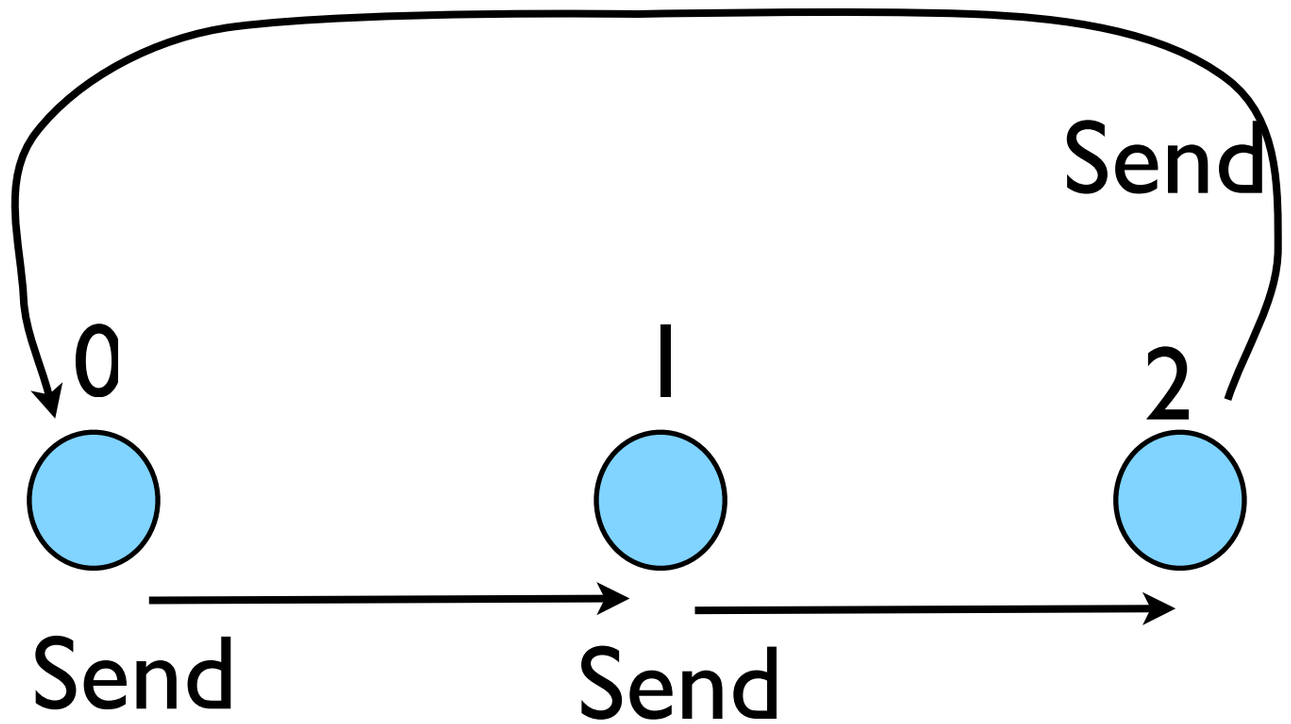
- cp secondmessage.{c,f90} thirdmessage.{c,f90}
- edit so it `wraps around'
- mpi{cc,f90} thirdmessage.{c,f90} -o thirdmessage
- mpirun -np 3 thirdmessage




TimeLine : Unknown.slog2 <Process View> (on tcs-f11n05)
Lowest / Max. Depth
0 / 0
Zoom Level
3
Global Min Time
0.0005229877
View Init Time
0.0023300347
Zoom Focus Time
0.0024764215
View Final Time
0.0026220003
Global Max Time
0.0026220003
Time Per Pixel
0.0000002340
TimeLines
SLOG-2
0
1
2
3
world_rank
Legend : Unknown.slog
Topo
Name
V
S
Preview_Arrow
message
Preview_State
MPI_Recv
MPI_Isend
Preview_Event
MPE_Comm_finalize
MPE_Comm_Init
All
Select
Deselect
close
0.00235
0.002375
0.0024
0.002425
0.00245
0.002475
0.0025
0.002525
0.00255
0.002575
0.0026
Time (seconds)

```c
if (rightneighbour < size) {
    ierr = MPI_Ssend(sendmessage, 6, MPI_CHAR, rightneighbour,
                     OURTAG, MPI_COMM_WORLD);
    printf("%d: Sent message <%s> to %d \n", rank, sendmessage, rightneighbour);
}
if (leftneighbour >= 0) {
    ierr = MPI_Recv(hearmessage, 6, MPI_CHAR, leftneighbour,
                     OURTAG, MPI_COMM_WORLD, &status);
    printf("%d: Recieved message <%s> from %d\n", rank, hearmessage, leftneighbour);
}
```



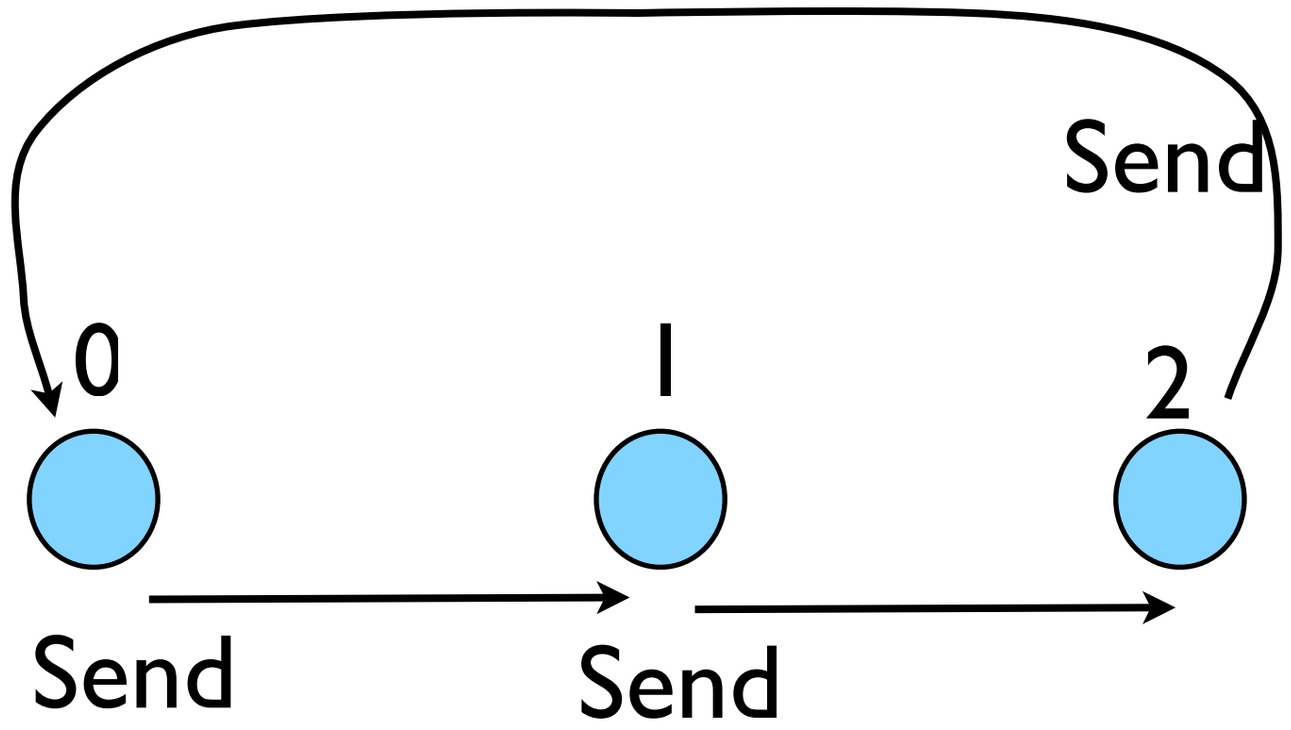



```
if (rightneighbour < size) {
    ierr = MPI_Ssend(sendmessage, 6, MPI_CHAR, rightneighbour,
                     OURTAG, MPI_COMM_WORLD);
    printf("%d: Sent message <%s> to %d \n", rank, sendmessage, rightneighbour);
}
if (leftneighbour >= 0) {
    ierr = MPI_Recv(hearmessage, 6, MPI_CHAR, leftneighbour,
                     OURTAG, MPI_COMM_WORLD, &status);
    printf("%d: Recieved message <%s> from %d\n", rank, hearmessage, leftneighbour);
}
```

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.



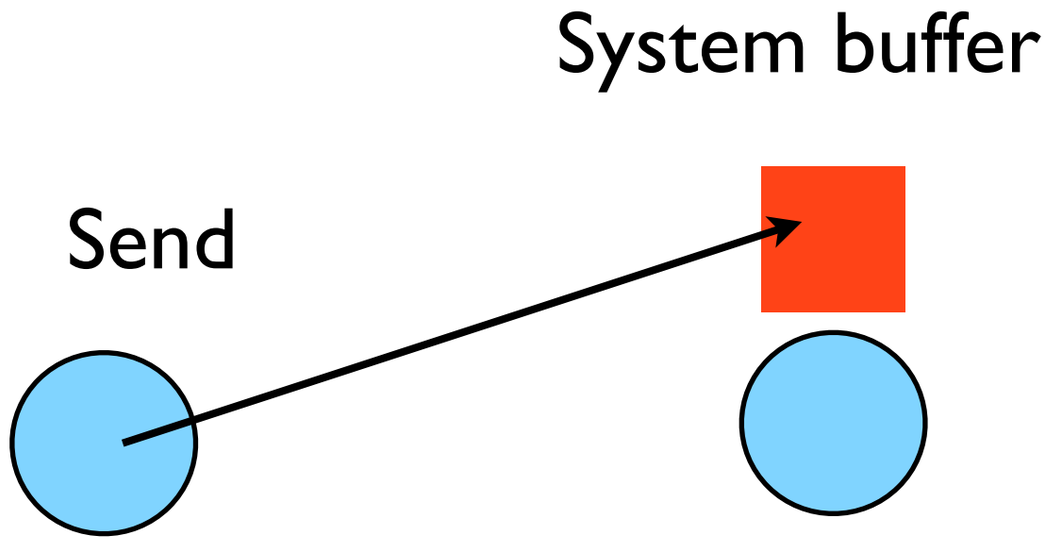
Big MPI
Lesson #1

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

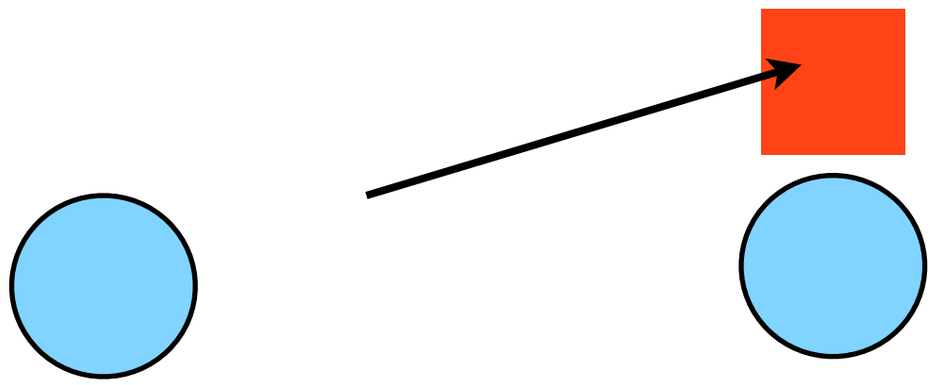
Different versions of SEND

- 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



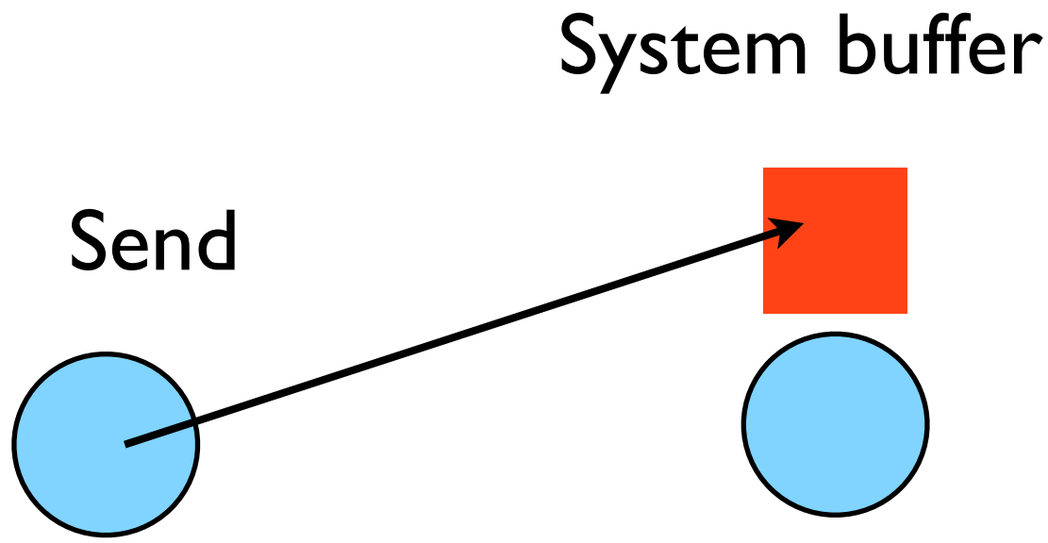
(Non) Blocking



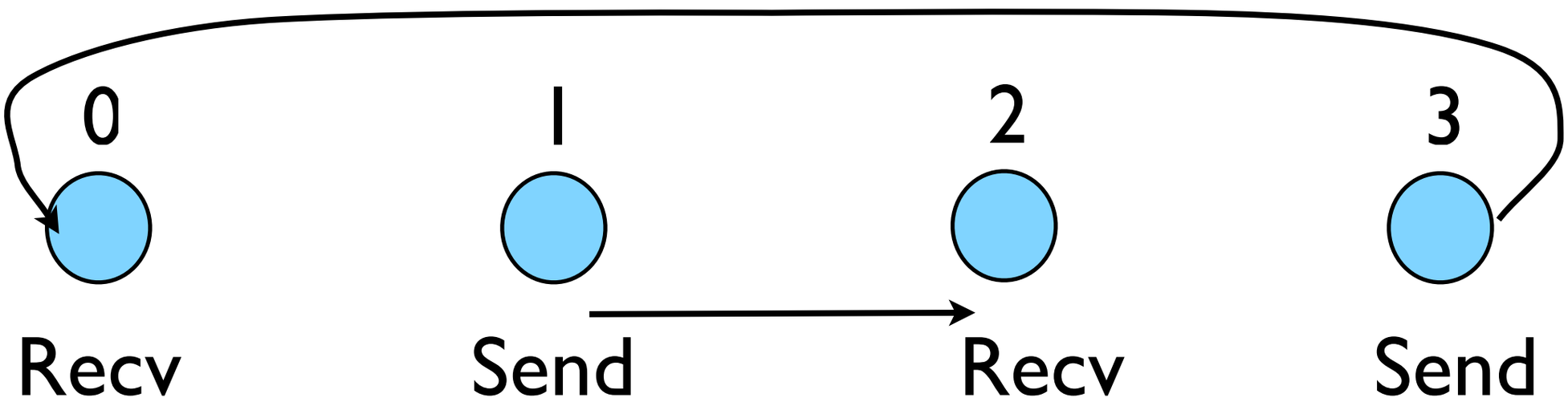
Buffering is dangerous!

- 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?

```f90
program hellompiworld
implicit none
include "mpif.h"

integer :: rank, comsize      ! standard MPI stuff
integer :: ierr
integer :: leftneighbour, rightneighbour
integer,parameter :: ourtag=1 ! shared label for messages
character(5) :: sendmessage   ! buffer for sending, receiving
character(5) :: hearmessage   ! messages
integer, dimension(MPI_STATUS_SIZE) :: status  ! rcv status

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

if (comsize .le. 1) then      ! need at least a sender, receiver
    print *,' FAIL: only one task'
    call MPI_Abort(MPI_COMM_WORLD,1)
endif

leftneighbour  = mod(rank - 1 + comsize,comsize)
rightneighbour = mod(rank + 1,comsize)

print *,'rank = ', rank, 'neighbours = ', leftneighbour, rightneighbour
sendmessage = 'Hello'

if (mod(rank,2) == 0) then
    call MPI_SSEND(sendmessage, 5, MPI_CHARACTER, rightneighbour, &
                   ourtag, MPI_COMM_WORLD, ierr)
    call MPI_RECV(hearmessage, 5, MPI_CHARACTER, leftneighbour, &
                  ourtag, MPI_COMM_WORLD, status, ierr)
else
    call MPI_RECV(hearmessage, 5, MPI_CHARACTER, leftneighbour, &
                  ourtag, MPI_COMM_WORLD, status, ierr)
    call MPI_SSEND(sendmessage, 5, MPI_CHARACTER, rightneighbour, &
                   ourtag, MPI_COMM_WORLD, ierr)
endif
print *, rank, ': sent message <',sendmessage,'> to ', rightneighbour
print *, rank, ': got message <',hearmessage,'> from ', leftneighbour

call MPI_FINALIZE(ierr)

end
```

Evens send first

Then odds

thirdmessage-fixed.f90

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

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

    int rank, size;         /* the usual MPI stuff */
    int ierr;
    char hearmessage[6];    /* we receive into here, and */
    char sendmessage[]="Hello"; /* send from here.*/
    int leftneighbour, rightneighbour;
    const int OURTAG=1;     /* shared tag to label messages */
    MPI_Status status;      /* receive status info */

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

    if (size < 2) {         /* need at least a sender, receiver */
        fprintf(stderr,"FAIL: only one task\n");
        MPI_Abort(MPI_COMM_WORLD,1);
    }

    leftneighbour = (rank-1 + size) % size;
    rightneighbour = (rank + 1) % size;

    if ((rank % 2) == 0) {
        ierr = MPI_Ssend(sendmessage, 6, MPI_CHAR, rightneighbour,
                         OURTAG, MPI_COMM_WORLD);
        ierr = MPI_Recv(hearmessage, 6, MPI_CHAR, leftneighbour,
                         OURTAG, MPI_COMM_WORLD, &status);
    } else {
        ierr = MPI_Recv(hearmessage, 6, MPI_CHAR, leftneighbour,
                         OURTAG, MPI_COMM_WORLD, &status);
        ierr = MPI_Ssend(sendmessage, 6, MPI_CHAR, rightneighbour,
                         OURTAG, MPI_COMM_WORLD);
    }
    printf("%d: Sent message <%s> to %d \n", rank, sendmessage, rightneighbour);
    printf("%d: Recieved message <%s> from %d\n", rank, hearmessage, leftneighbour);

    MPI_Finalize();

    return 0;
}
```

Evens send first

Then odds

thirdmessage-fixed.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.

```
1         program hellompiworld
2         implicit none
3         include "mpif.h"
4
5         integer :: rank, comsize      ! standard MPI stuff
6         integer :: ierr
7         integer :: leftneighbour, rightneighbour
8         integer,parameter :: ourtag=1 ! shared label for messages
9         character(5) :: sendmessage   ! buffer for sending, receiving
10        character(5) :: hearmessage   ! messages
11        integer, dimension(MPI_STATUS_SIZE) :: status  ! rcv status
12
13        call MPI_INIT(ierr)
14        call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
15        call MPI_COMM_SIZE(MPI_COMM_WORLD, comsize, ierr)
16
17        if (comsize .le. 1) then      ! need at least a sender, receiver
18            print *,' FAIL: only one task'
19            call MPI_Abort(MPI_COMM_WORLD,1)
20        endif
21
22        leftneighbour  = mod(rank - 1 + comsize,comsize)
23        rightneighbour = mod(rank + 1,comsize)
24
25        sendmessage = 'Hello'
26        call MPI_SENDRECV(sendmessage, 5, MPI_CHARACTER, rightneighbour, ourtag,
27                          hearmessage, 5, MPI_CHARACTER, leftneighbour,  ourtag,
28                          MPI_COMM_WORLD, status, ierr)
29
30        print *, rank, ': sent message <',sendmessage,'> to ', rightneighbour
31        print *, rank, ': got message <',hearmessage,'> from ', leftneighbour
32
33        call MPI_FINALIZE(ierr)
34
35        end
```

fourthmessage.f90

Something new: Sendrecv

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

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

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

    int rank, size;          /* the usual MPI stuff */
    int ierr;
    char hearmessage[6];     /* we receive into here, and */
    char sendmessage[]="Hello"; /* send from here.*/
    int leftneighbour, rightneighbour;
    const int OURTAG=1;      /* shared tag to label messages */
    MPI_Status status;       /* receive status info */

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

    if (size < 2) {          /* need at least a sender, receiver */
        fprintf(stderr,"FAIL: only one task\n");
        MPI_Abort(MPI_COMM_WORLD,1);
    }

    leftneighbour = (rank-1 + size) % size;
    rightneighbour = (rank + 1) % size;

    ierr = MPI_Sendrecv(sendmessage, 6, MPI_CHAR, rightneighbour, OURTAG,
                        hearmessage, 6, MPI_CHAR, leftneighbour, OURTAG,
                        MPI_COMM_WORLD, &status);

    printf("%d: Sent message <%s> to %d \n", rank, sendmessage, rightneighbour);
    printf("%d: Recieved message <%s> from %d\n", rank, hearmessage, leftneighbour);

    MPI_Finalize();

    return 0;
}
```

fourthmessage.c

Sendrecv = Send + Recv

C syntax

```
MPI_Status status;

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

Send Args

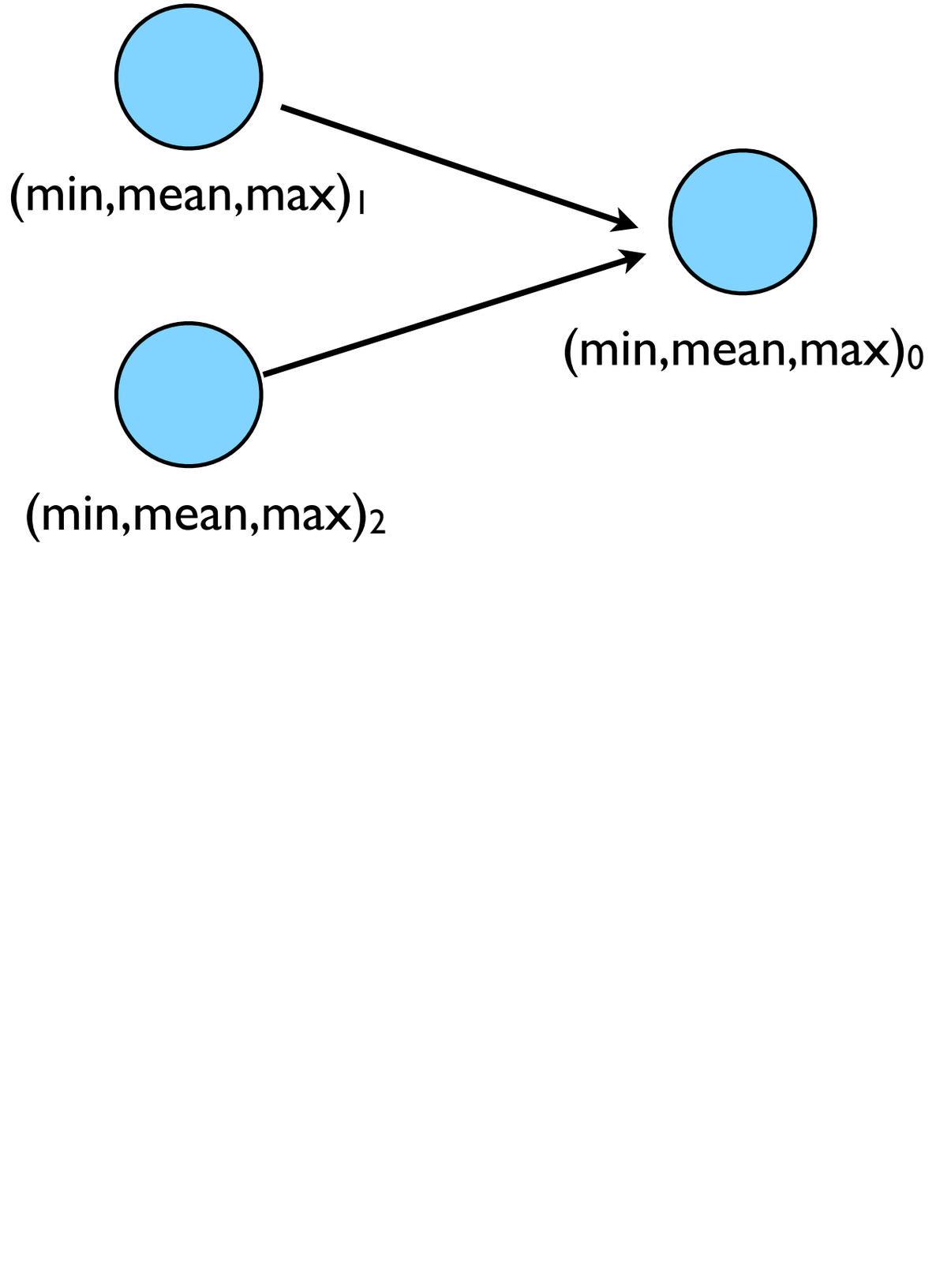
Recv Args

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?

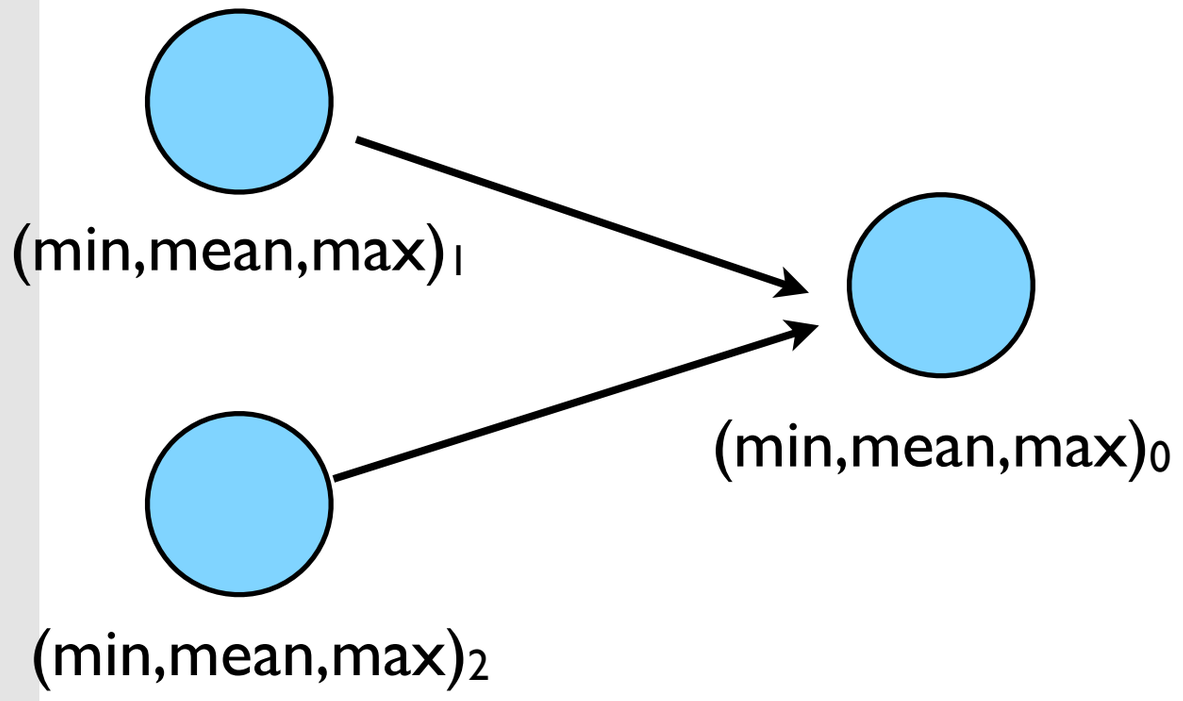
Min, Mean, Max of numbers



- 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?)
- ~ljdursi/ss2010/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 srand(0)
      do i=1,nx
         dat(i) = 2*rand(0)-1.
      enddo

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

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

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

      return
      end
```

```
33 c
34 c find min/mean/max
35 c
36       datamin = 1e+19
37       datamax =-1e+19
38       datamean = 0
39
40       do i=1,nx
41          do j=1,ny
42              if (dat(i,j) .lt. datamin) datamin = dat(i,j)
43              if (dat(i,j) .gt. datamax) datamax = dat(i,j)
44              datamean = datamean + dat(i,j)
45          enddo
46       enddo
47       datamean = datamean/(1.*nx*ny)
48
49       print *,myid,': min/mean/max = ', datamin, datamean, datamax
50 c
51 c combine data
52 c
53       if (myid .ne. 0) then
54           datapack(1) = datamin
55           datapack(2) = datamean
56           datapack(3) = datamax
57           call MPI_SSEND(datapack,3,MPI_REAL,0,1,MPI_COMM_WORLD,ierr)
58       else
59           globmin  = datamin
60           globmax  = datamax
61           globmean = datamean
62           do proc=1,nprocs-1
63                call MPI_RECV(datapack, 3, MPI_REAL, MPI_ANY_SOURCE, 1,
64      &                      MPI_COMM_WORLD, status, ierr)
65                if (datapack(1) .lt. globmin) globmin=datapack(1)
66                globmean = globmean + datapack(2)
67                if (datapack(3) .gt. globmax) globmax=datapack(3)
68           enddo
69           globmean = globmean/nprocs
70           print *,'Global min/mean/max=',globmin,globmean,globmax
71       endif
72
73       call MPI_FINALIZE(ierr)
74       return
75       end
76
77
```

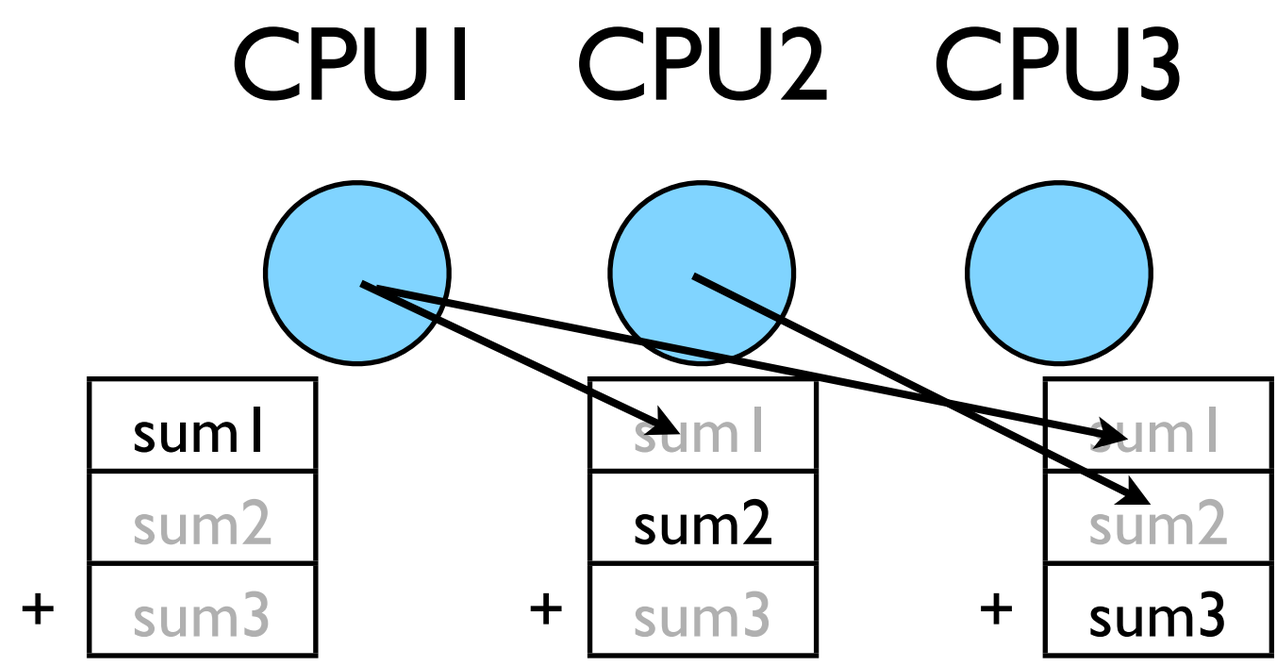


Q: are these sends/recvd adequately paired?

minmeanmax-mpi.f

Inefficient!

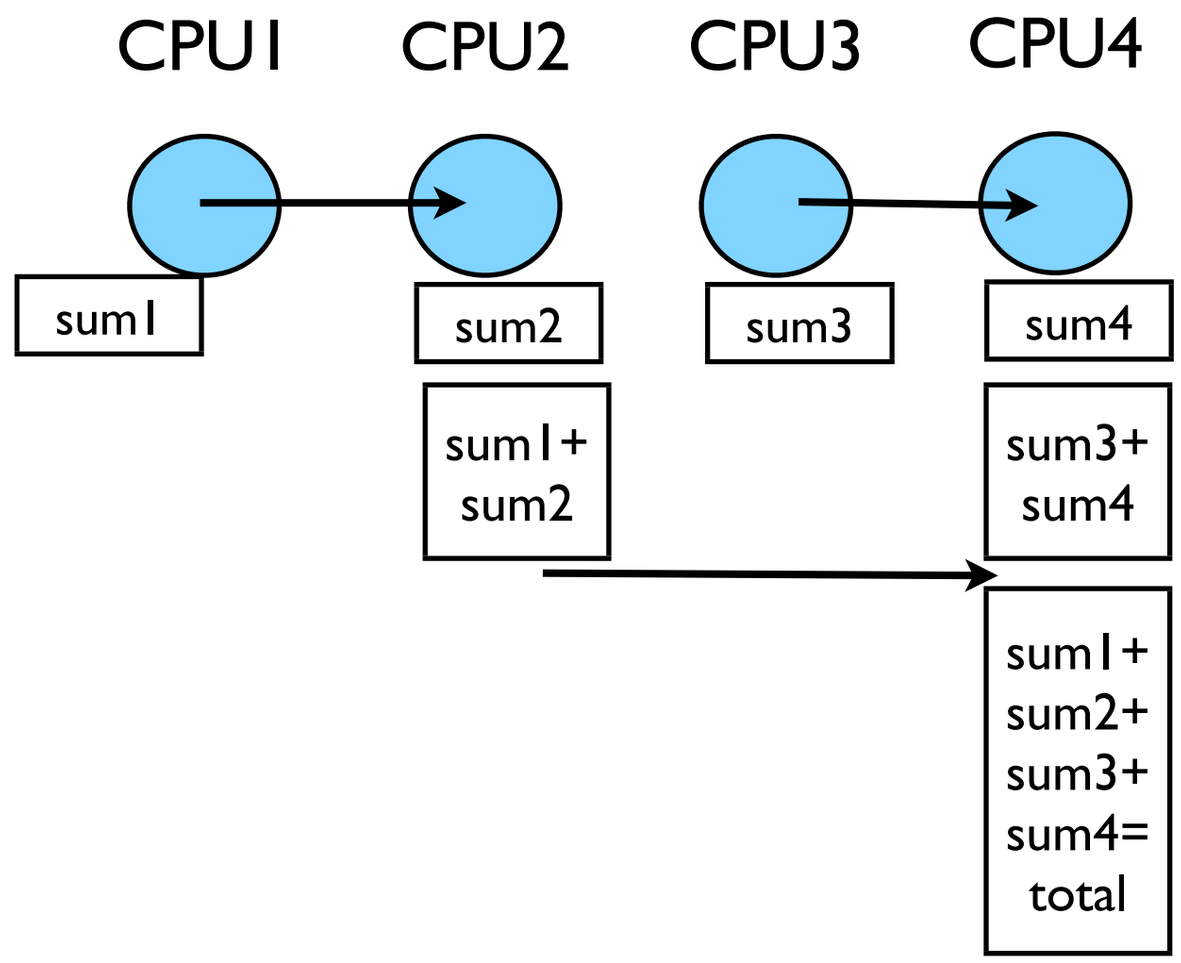
- Requires (P-1) messages, 2 (P-1) if everyone then needs to get the answer.



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...)

```
c
c find min/mean/max
c
      datamin = 1e+19
      datamax =-1e+19
      datamean = 0

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

      print *,myid,': min/mean/max = ', datamin, datamean, datamax
c
c combine data
c
      call MPI_ALLREDUCE(datamin, globmin, 1, MPI_REAL, MPI_MIN,
     &                   MPI_COMM_WORLD, ierr)
c
c to just send to task 0:
c      call MPI_REDUCE(datamin, globmin, 1, MPI_REAL, MPI_MIN,
c     &                   0, MPI_COMM_WORLD, ierr)
c      etc.
c
      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/nprocs
      print *, myid,': Global min/mean/max=',globmin,globmean,globmax

      call MPI_FINALIZE(ierr)
      return
      end
```

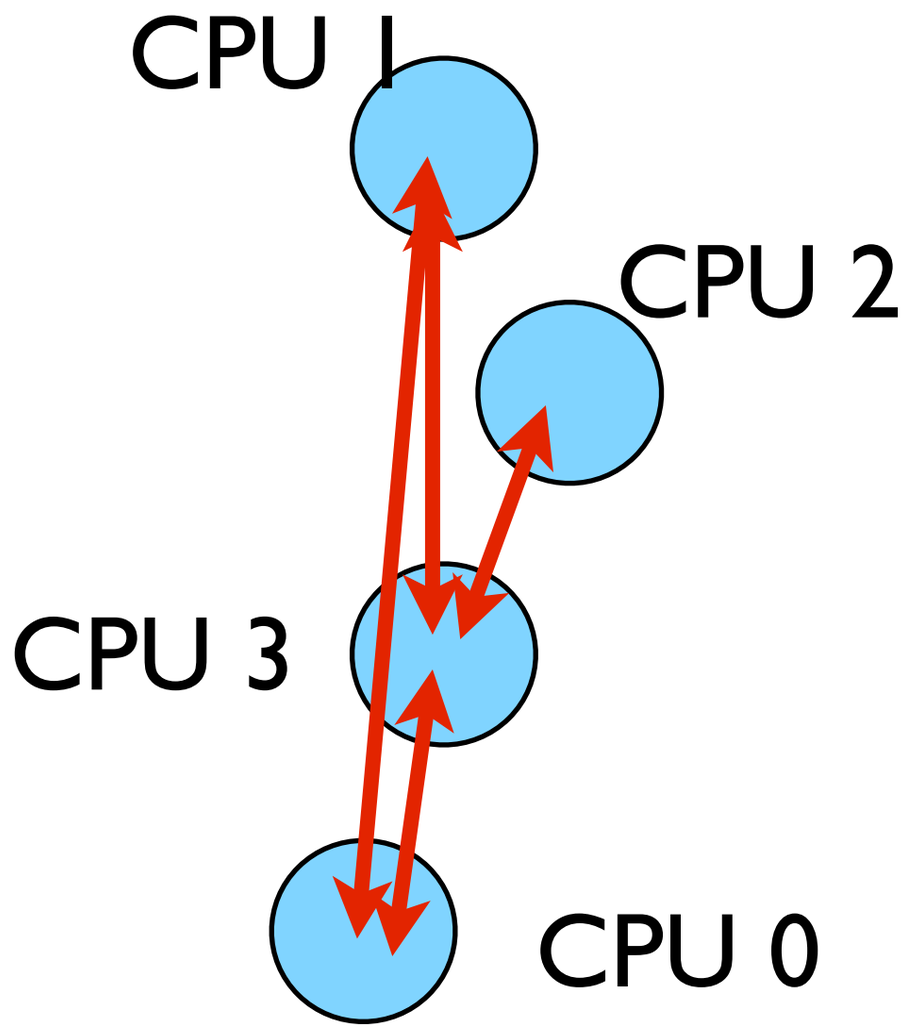
MPI_Reduce and MPI_Allreduce

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

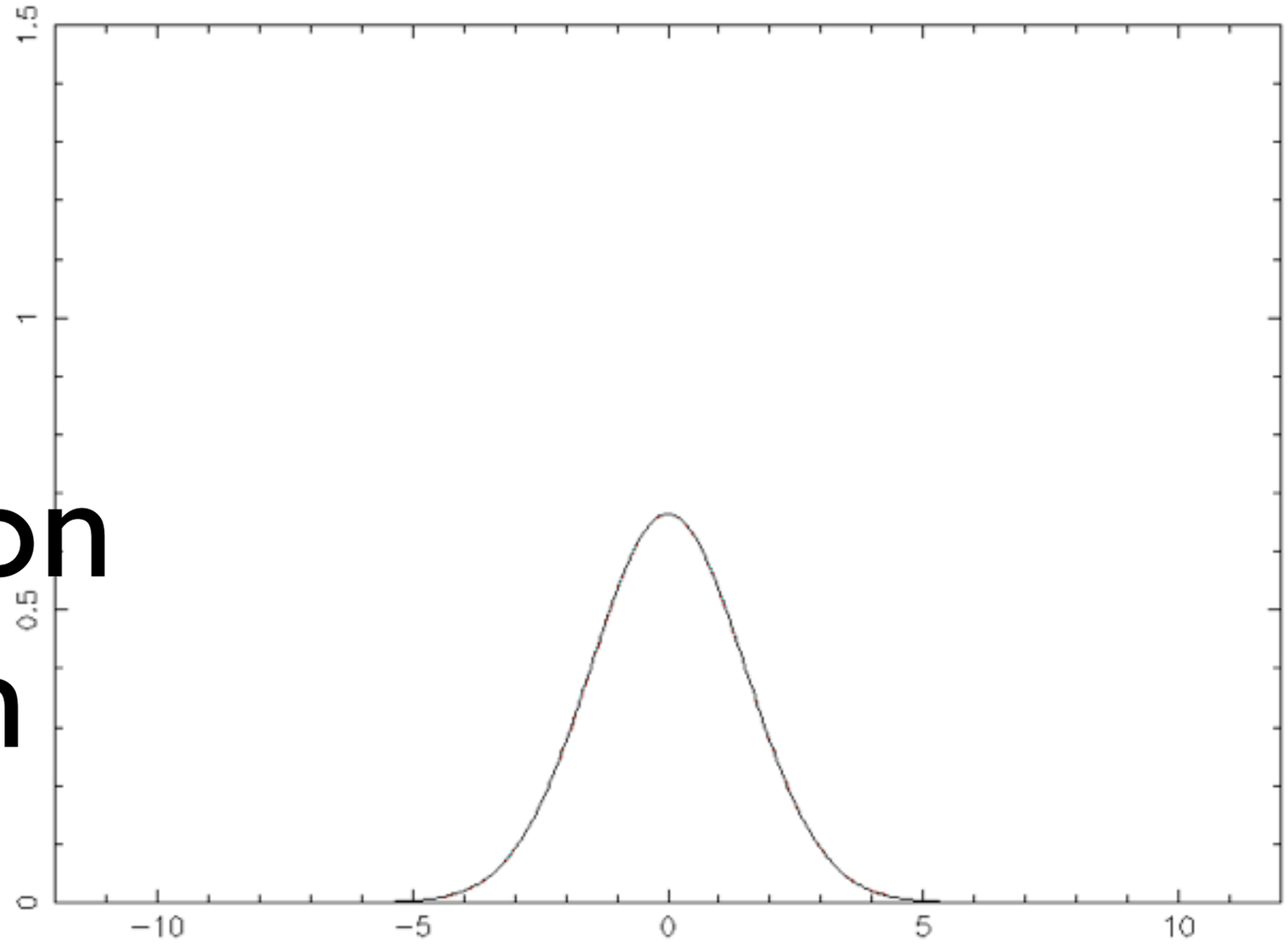
minmeanmax-allreduce.f

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

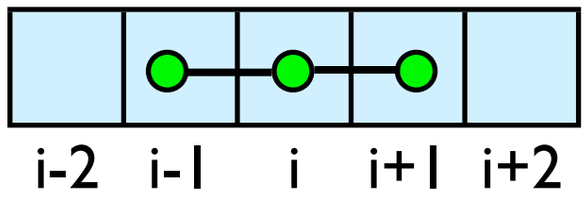


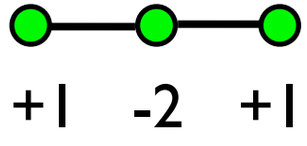

```
cd ~/intro-ppp/diffusion
make diffusionf or make diffusionc
./diffusionf or ./diffusionc
```

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

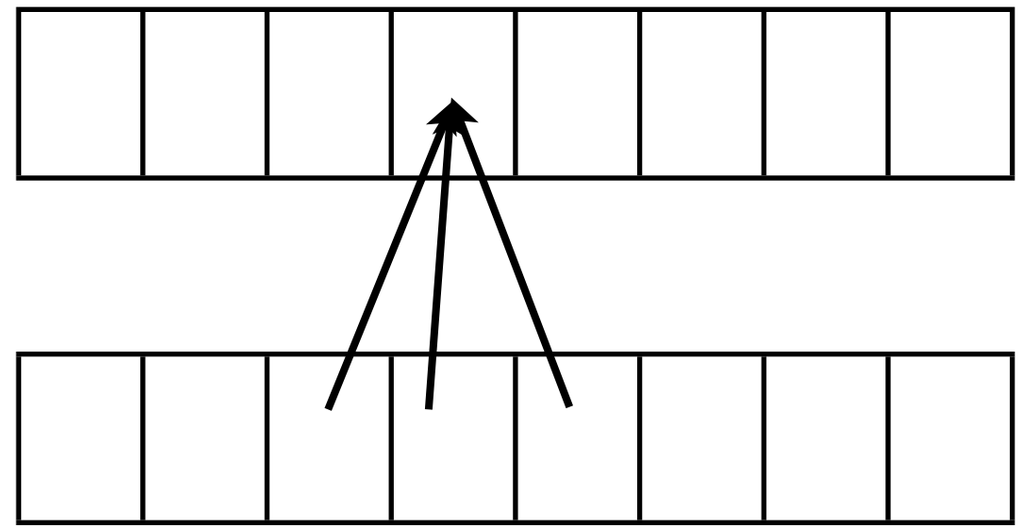
$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2}$$

$$\frac{\partial T_i^{(n)}}{\partial t} \approx \frac{T_i^{(n)} + T_i^{(n-1)}}{\Delta t}$$

$$\frac{\partial T_i^{(n)}}{\partial x} \approx \frac{T_{i+1}^{(n)} - 2T_i^{(n)} + T_{i-1}^{(n)}}{\Delta x^2}$$

$$T_i^{(n+1)} \approx T_i^{(n)} + \frac{D\Delta t}{\Delta x^2}\left(T_{i+1}^{(n)} - 2T_i^{(n)} + T_{i-1}^{(n)}\right)$$

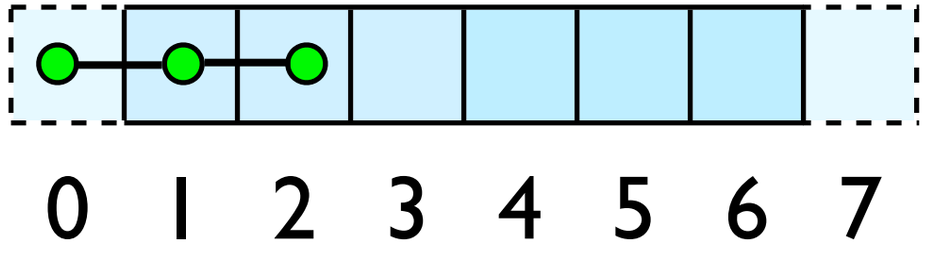
- Simple 1d PDE
- Each timestep, new data for T[i] requires old data for T[i+1],T[i],T[i-1]



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

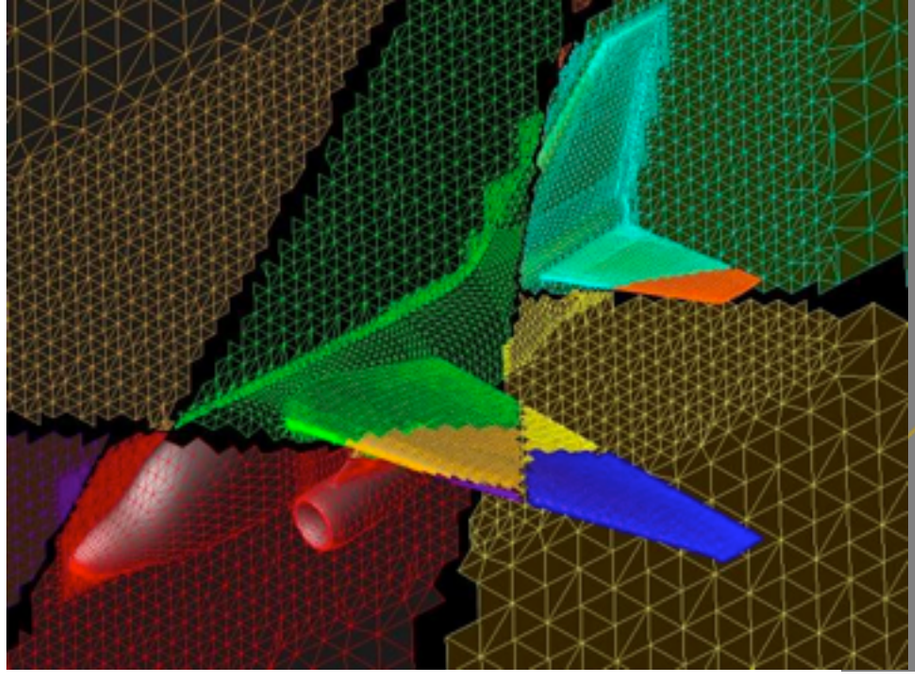


ng = 1

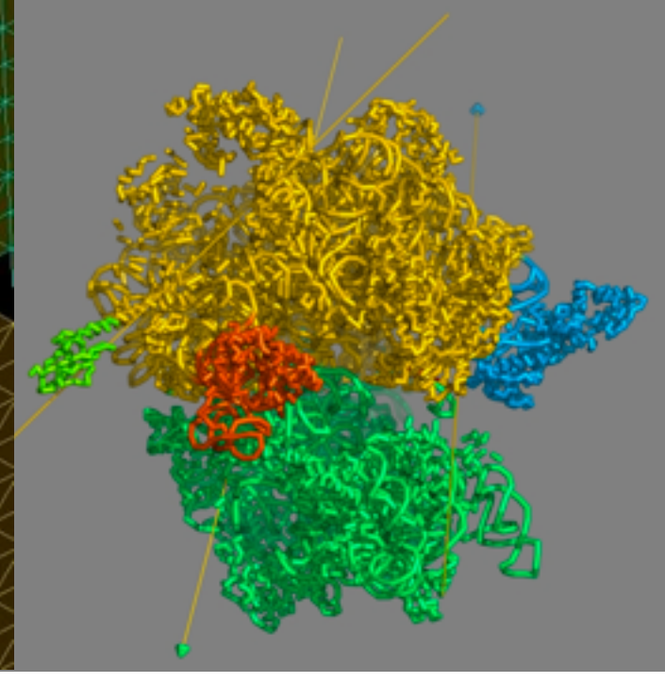
loop from ng, N - 2 ng

Domain Decomposition

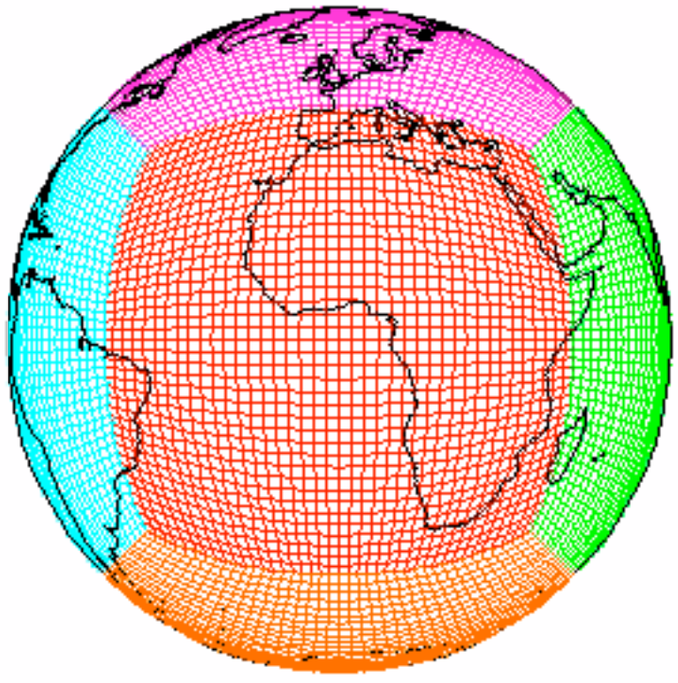
- A very common approach to parallelizing on distributed memory computers
- Maintain Locality; need local data mostly, this means only surface data needs to be sent between processes.



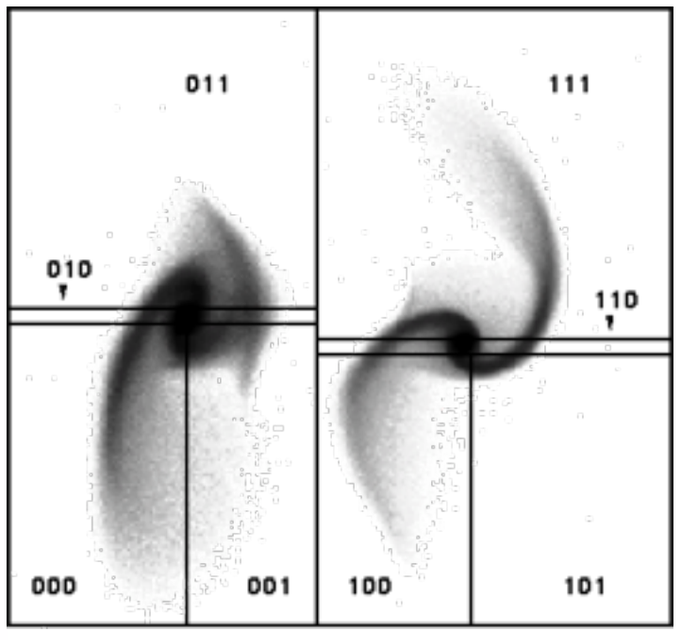
http://adg.stanford.edu/aa241/design/compaero.html



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



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



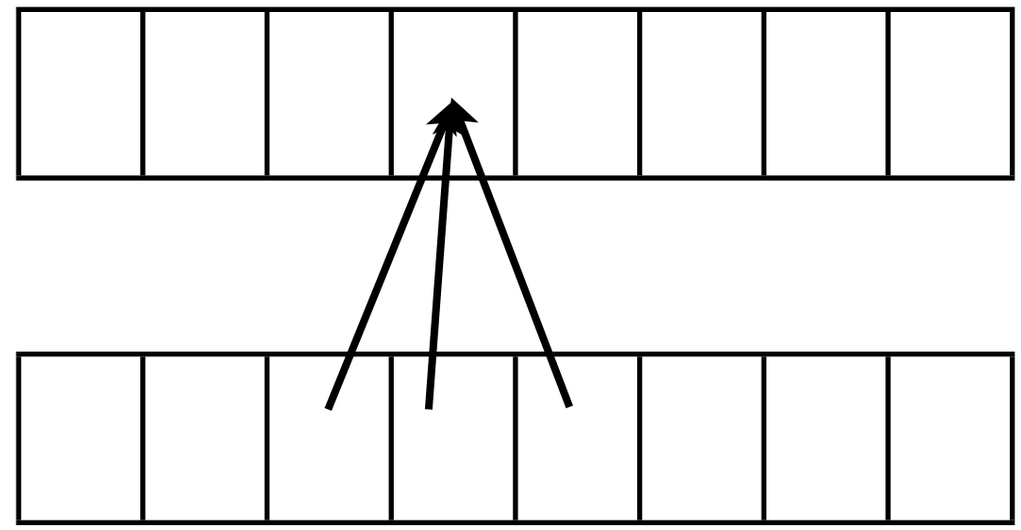
http://www.cita.utoronto.ca/~dubinski/treecode/node8.html

Implement a diffusion equation in MPI

$$\frac{dT}{dt} = D\frac{d^2T}{dx^2}$$

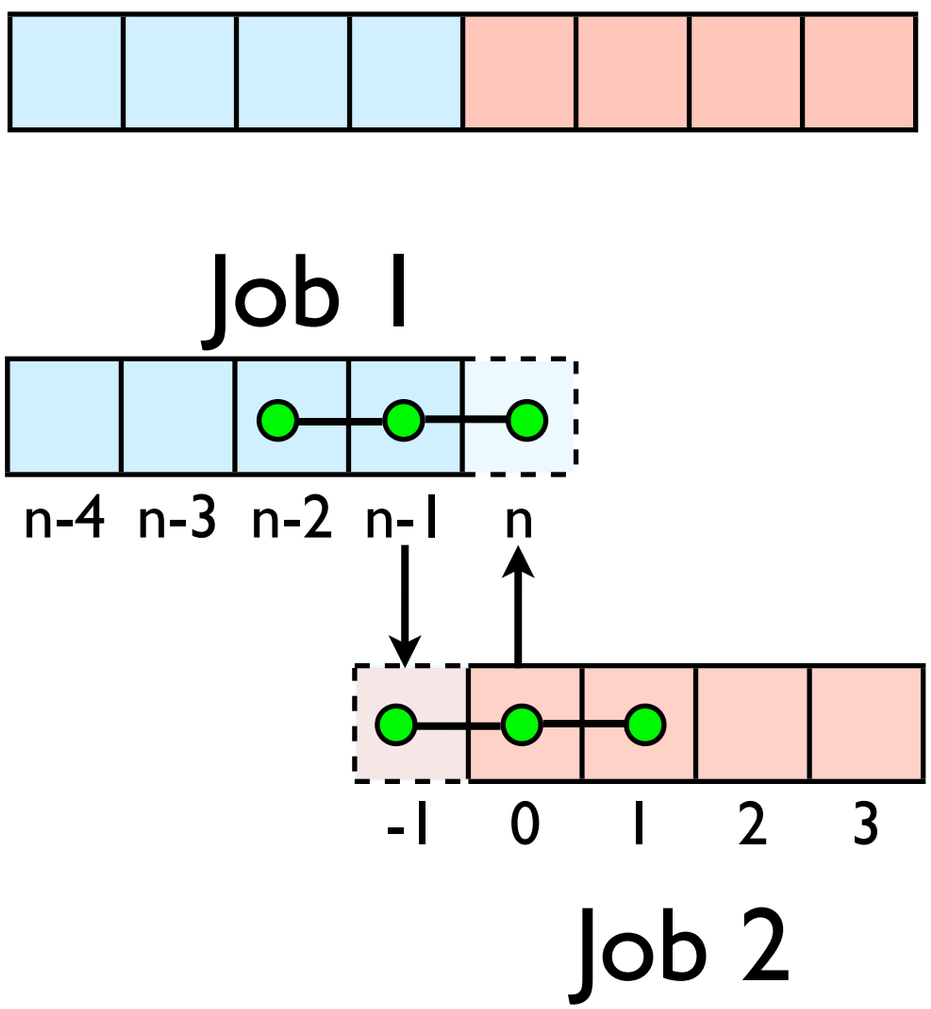
$$T_i^{n+1} = T_i^n + \frac{D\Delta t}{\Delta x^2}\left(T_{i+1}^n - 2T_i^n + T_{i-1}^n\right)$$

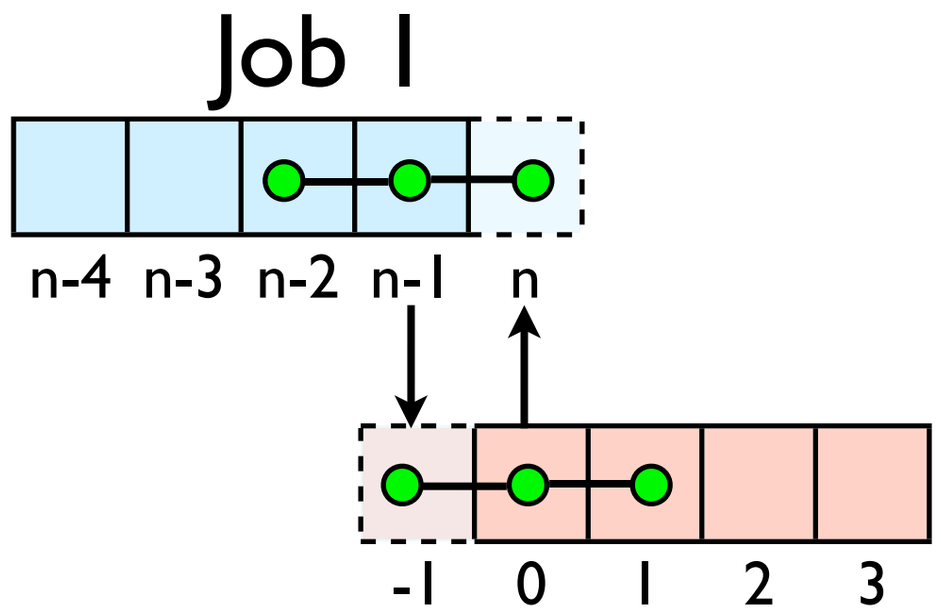
- Need one neighboring number per neighbor per timestep



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,...
```

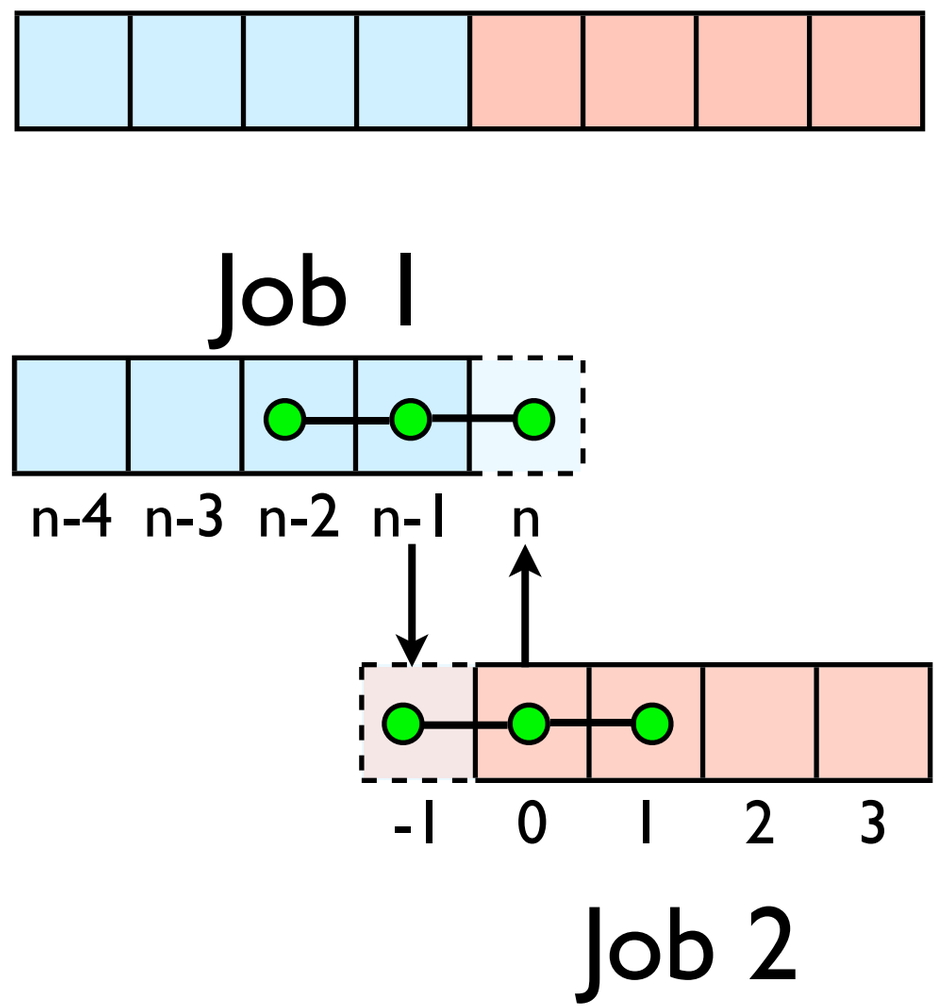
FORTRAN syntax

```
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

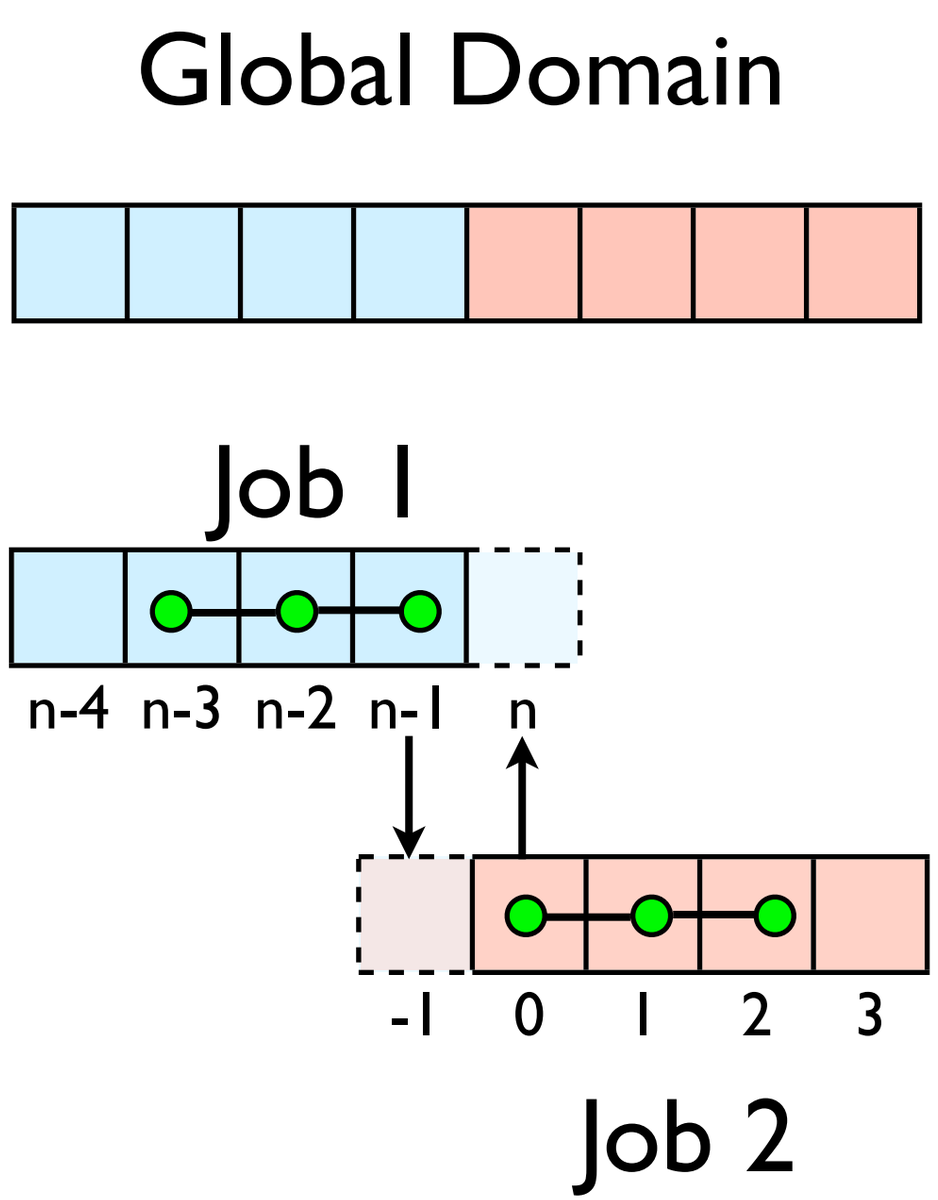
Diffusion: Had to wait for communications to compute

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



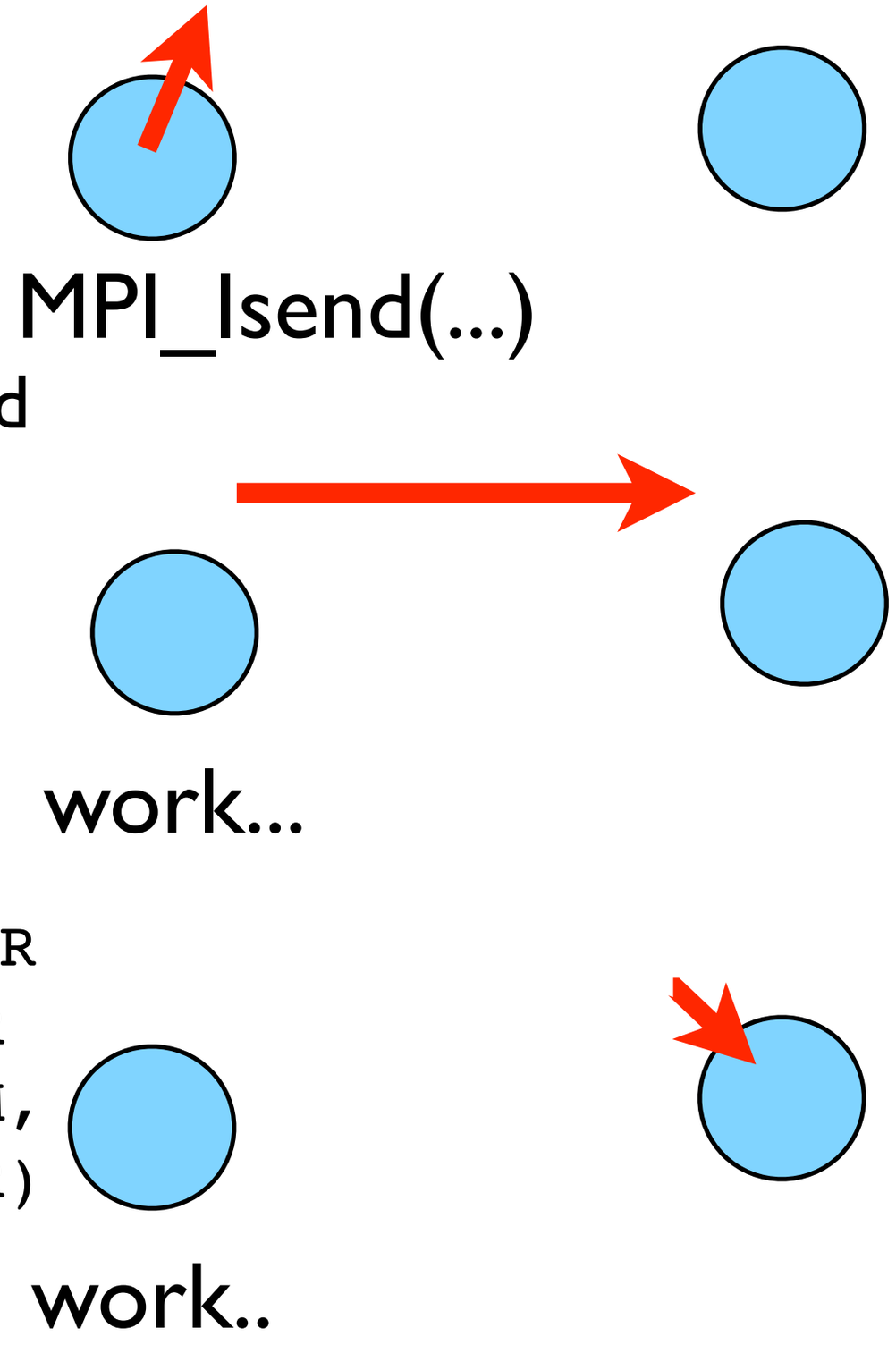
Diffusion: *Had* to wait?

- 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.



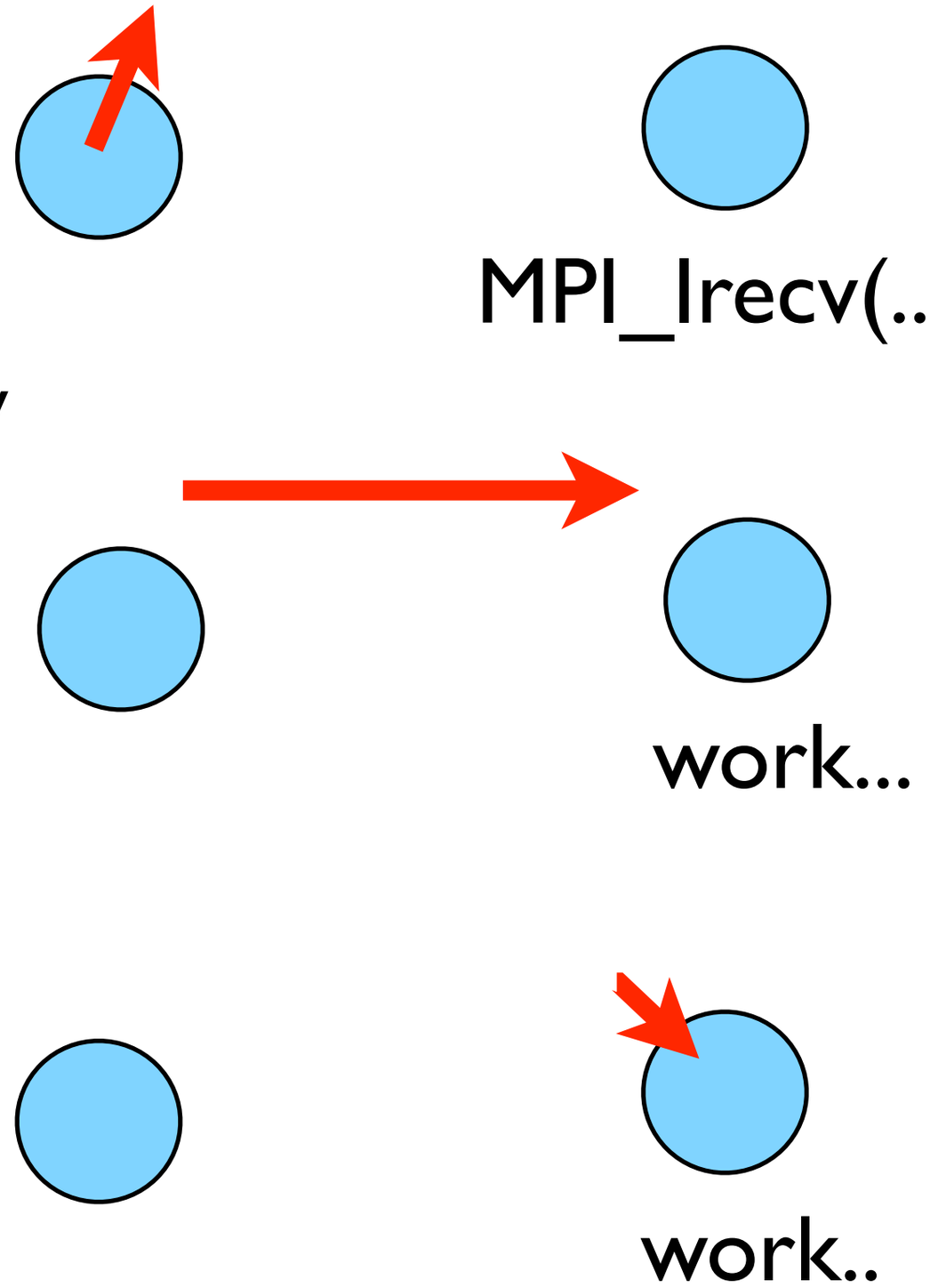
Nonblocking Sends

- 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

CumulativeExc...

SLOG-2

0

1

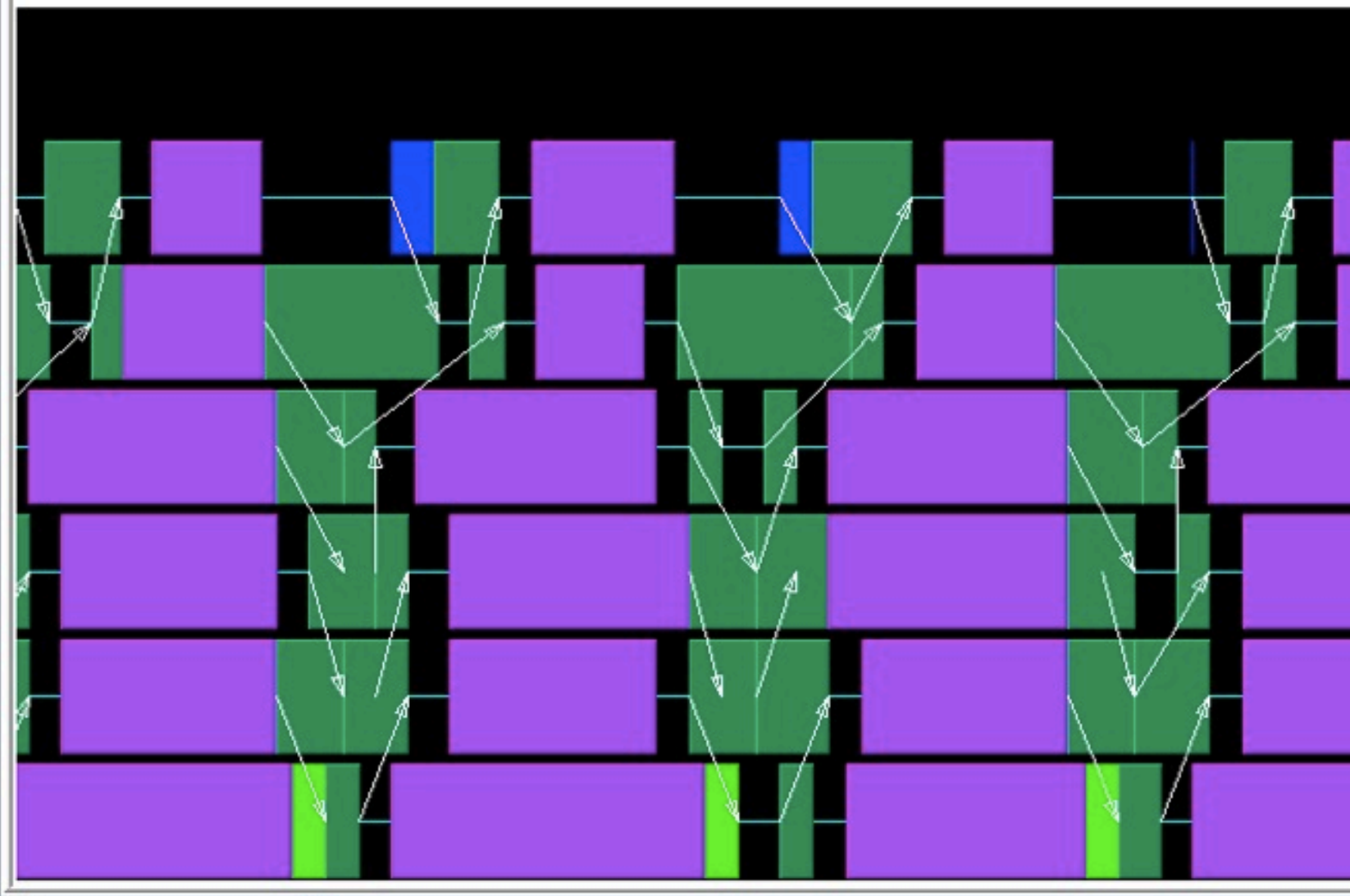
2

3

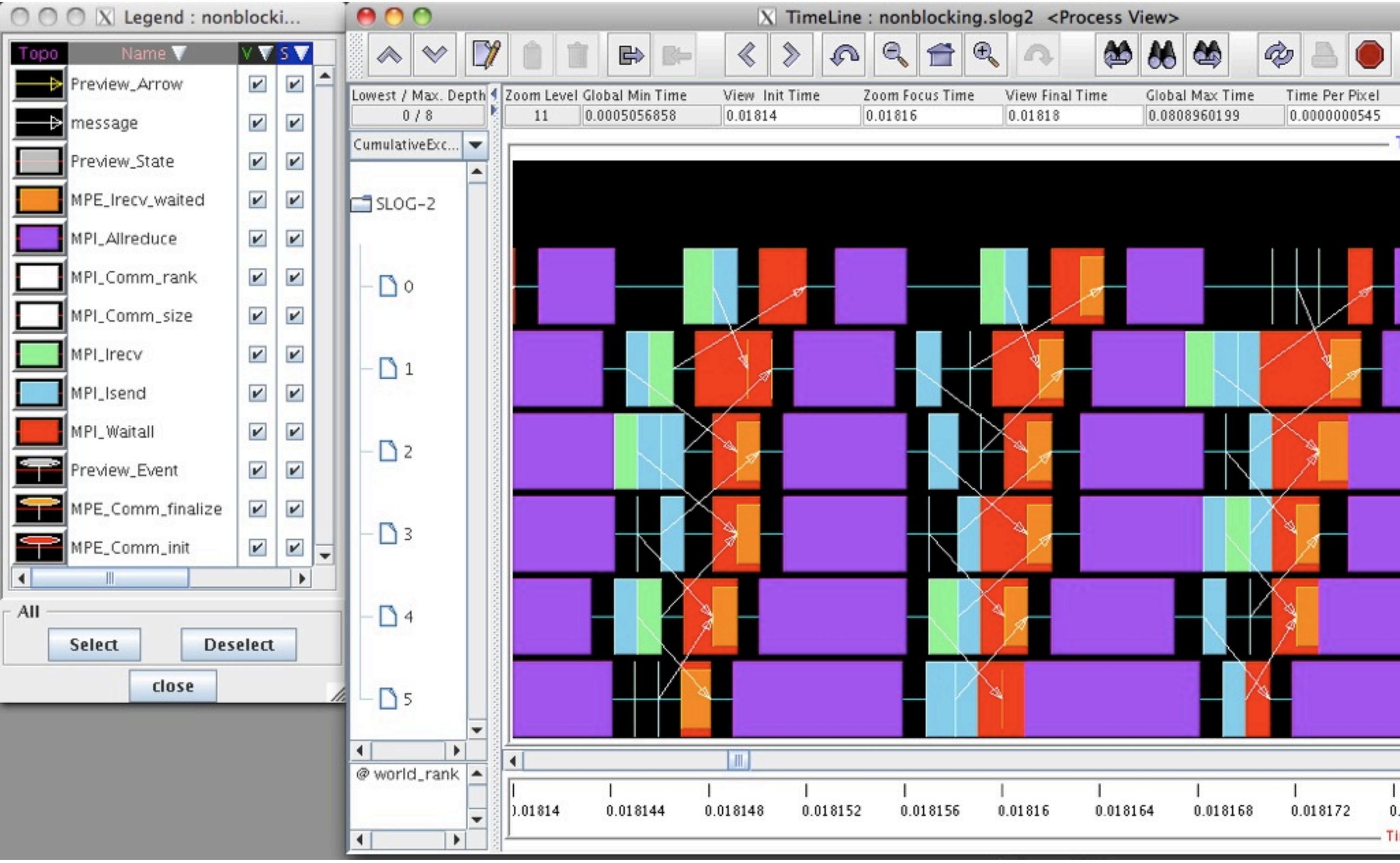
4

5

@ world_rank



6 0.01814 0.018144 0.018148 0.018152 0.018156 0.01816 0.018164 0.018168 0.018172 0.0


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
0.01814
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