# 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

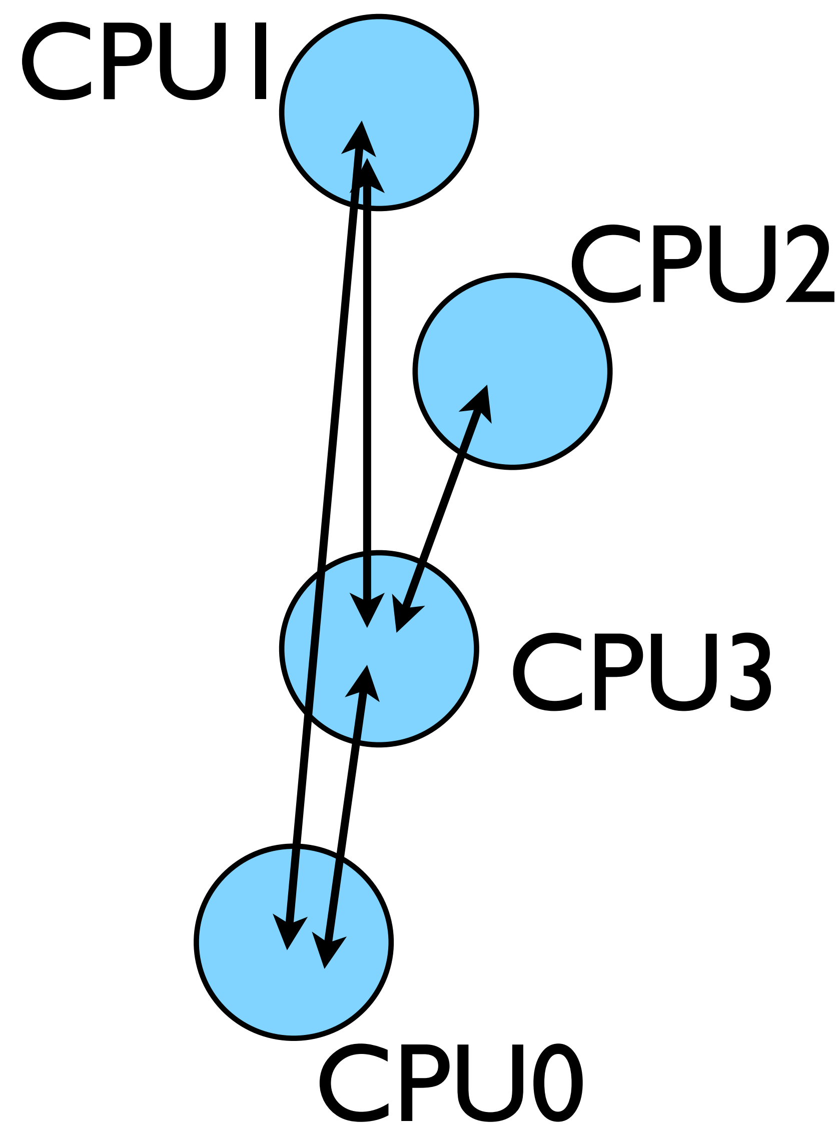
```c
#include <stdio.h>
#include <mpi.h>

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
```

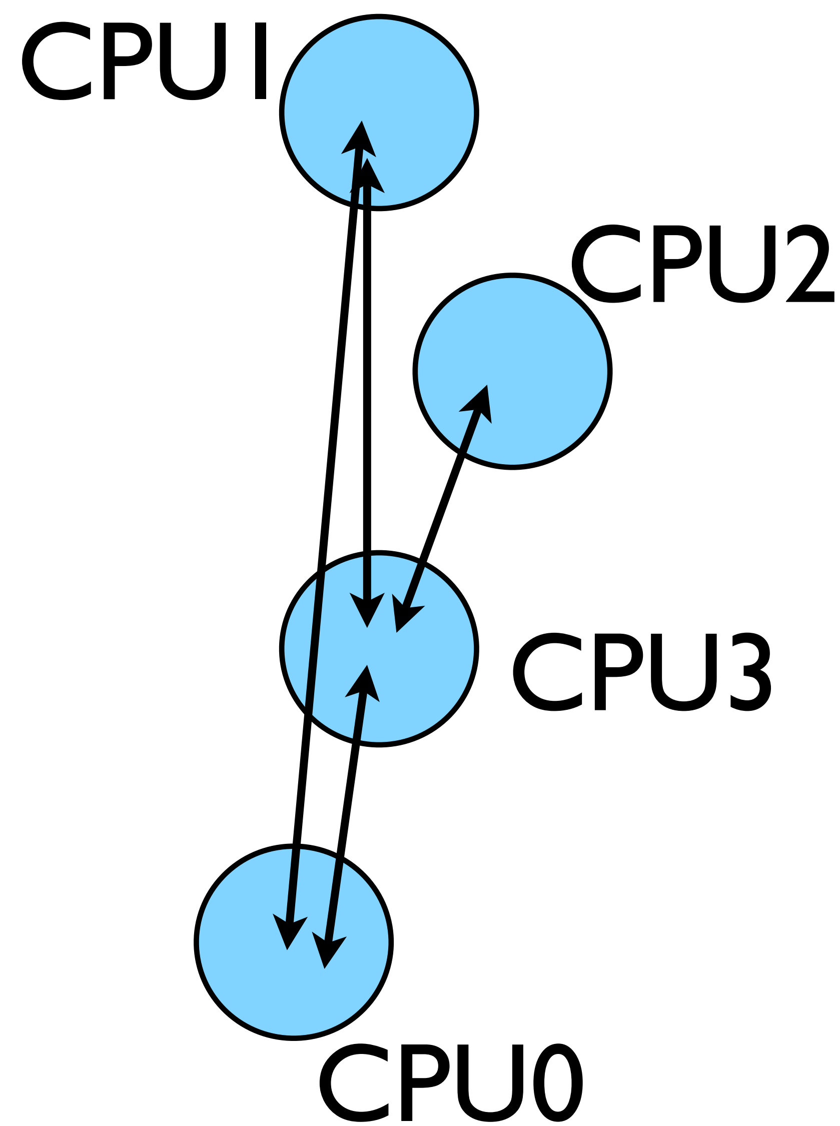
# MPI is a Library for **Message-Passing**

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



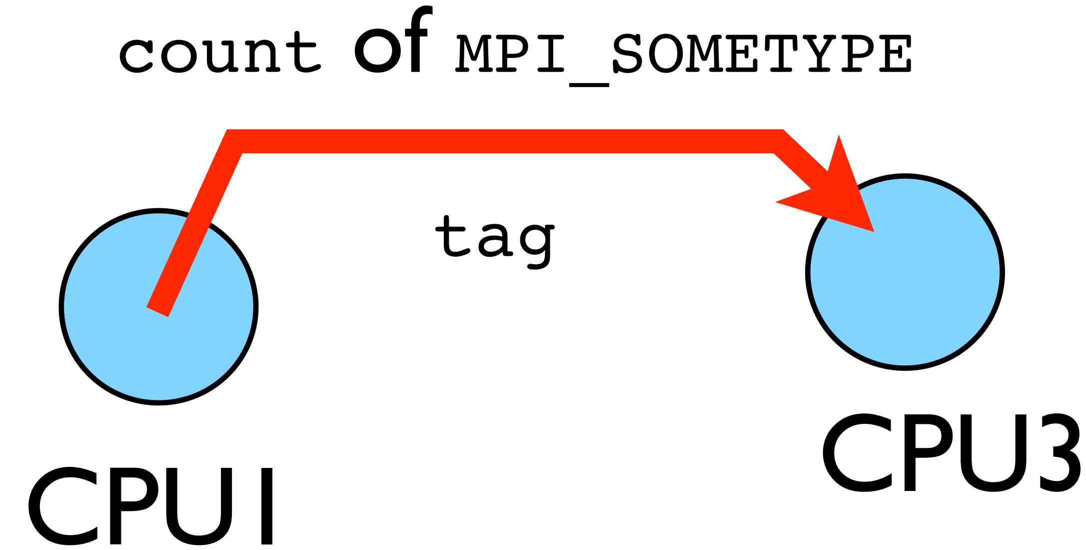
# MPI is a Library for **Message-Passing**

- 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



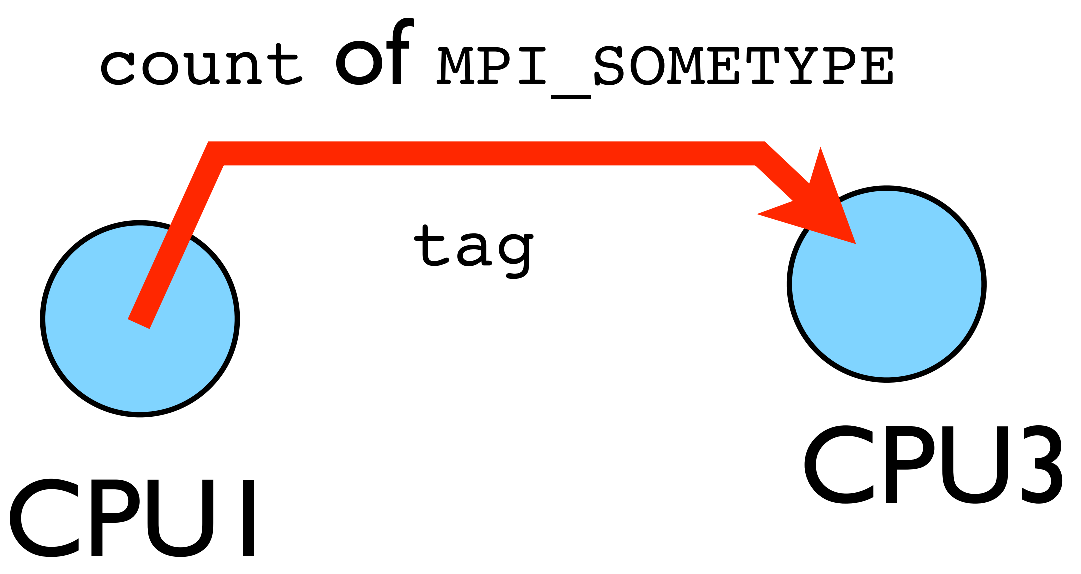
# Messages

- 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

```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

```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

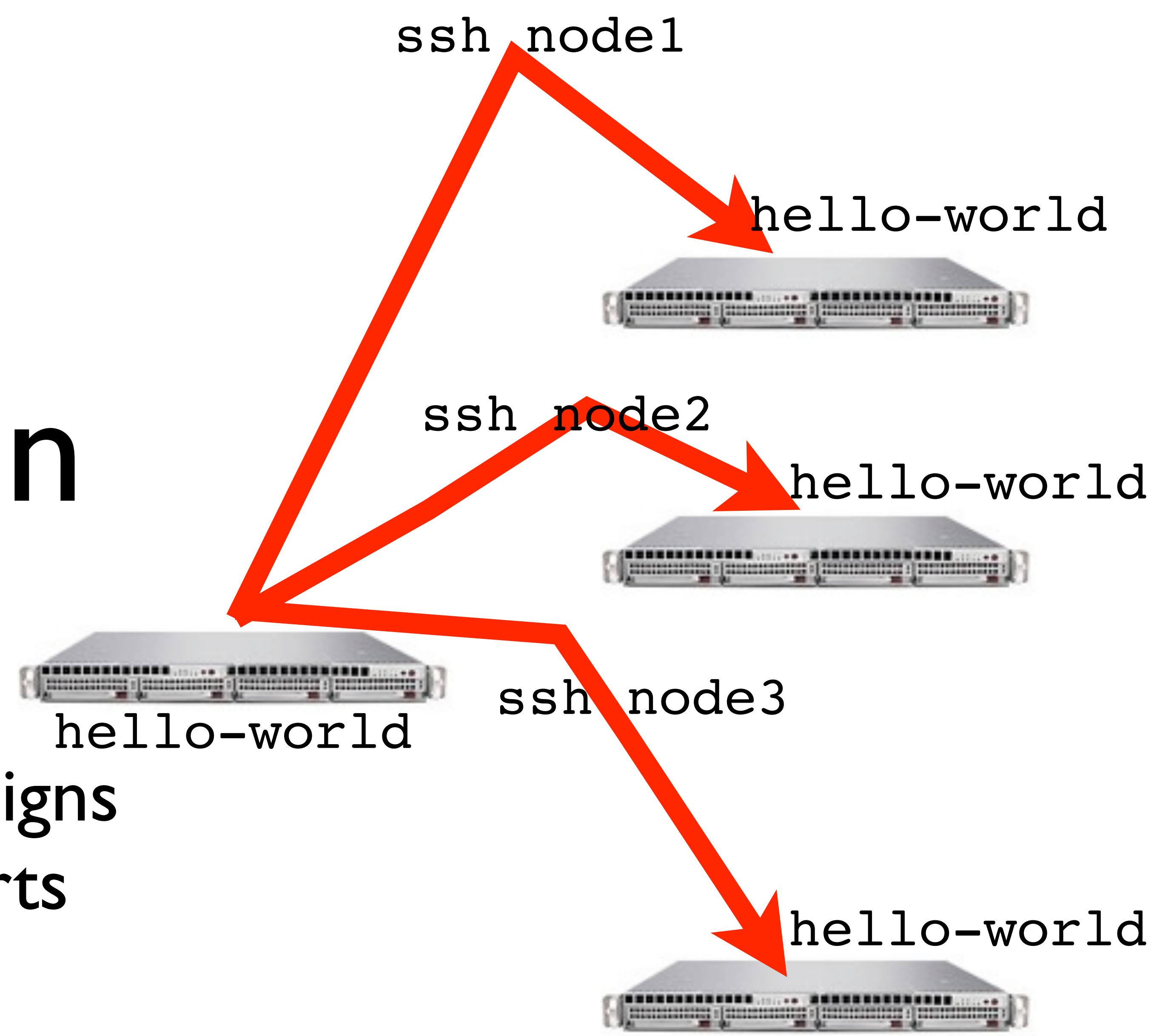
```
$ mpicc hello-world.c -o hello-world
$ mpirun -np 1 hello-world
$ mpirun -np 2 hello-world
```

# 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

```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,

call MPI_FINALIZE(ierr)

return
end
```

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

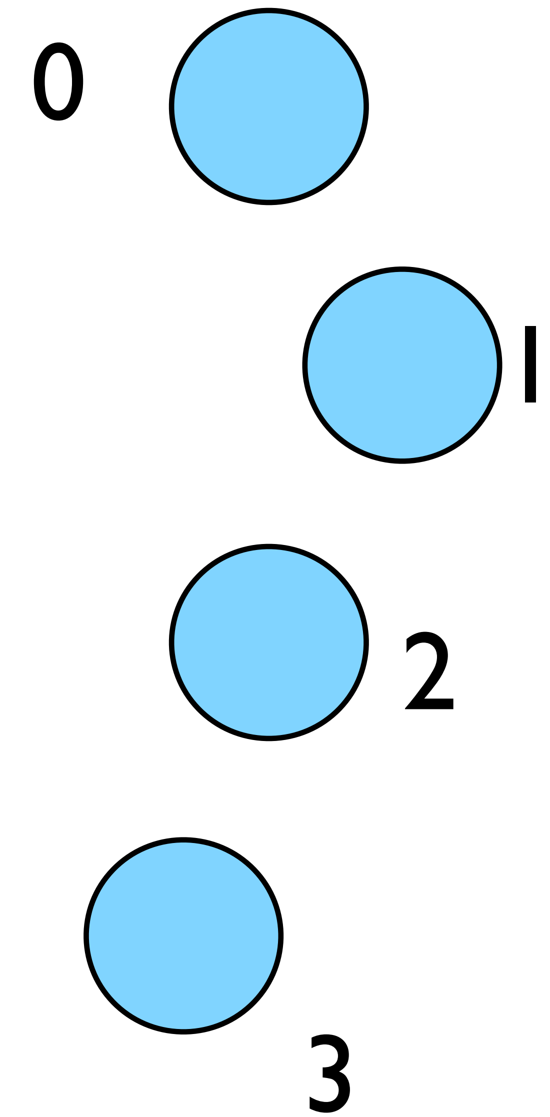
`call MPI_FINALIZE(ierr)`: close up MPI stuff. Must come last.
`ierr`: Returns any error code.

```
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

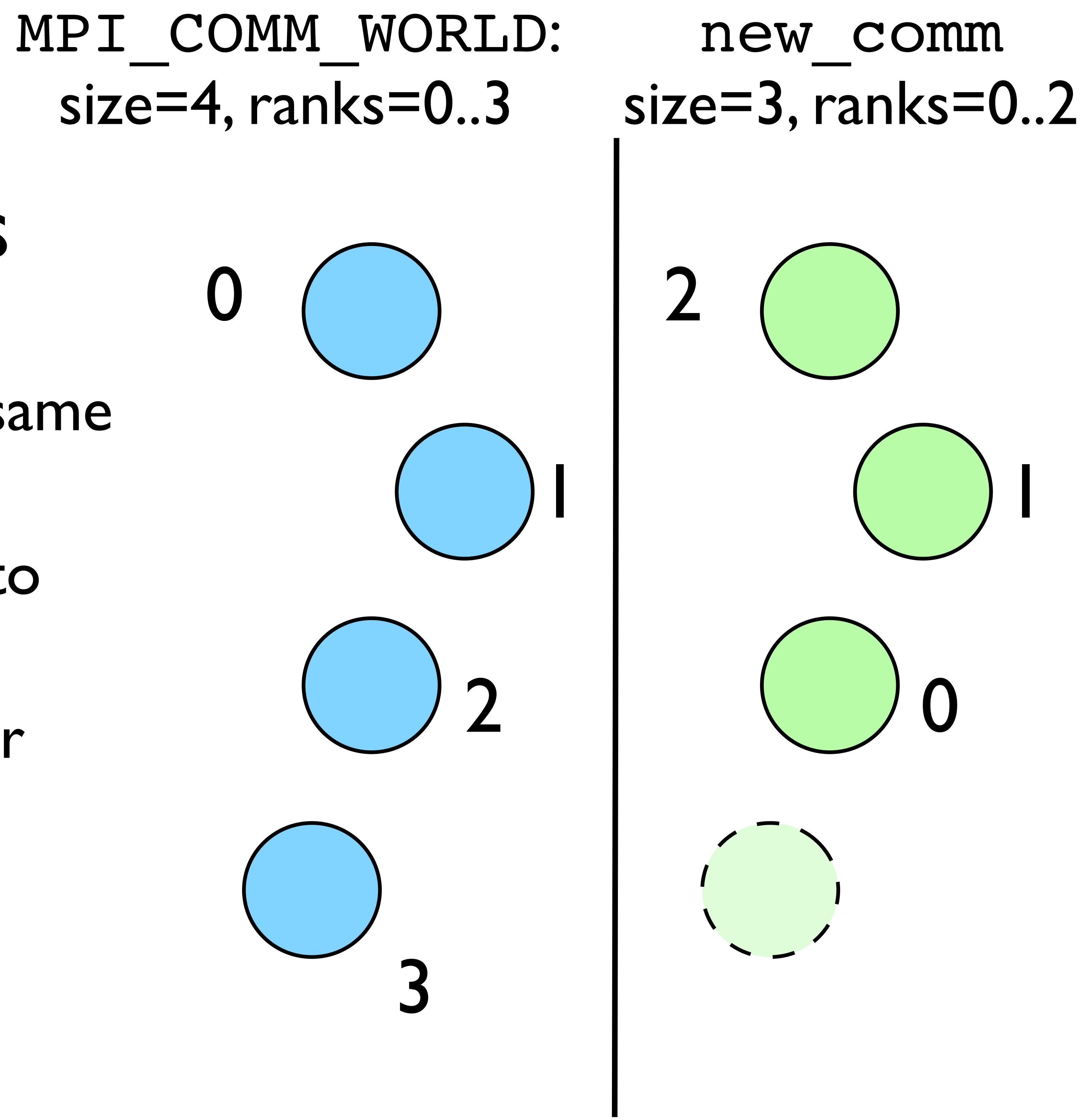
- MPI groups processes into communicators.
- Each communicator has some size -- number of tasks.
- Each task has a rank 0..size-1
- Every task in your program belongs to `MPI_COMM_WORLD`




`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



`call MPI_COMM_RANK`,
`call MPI_COMM_SIZE`:

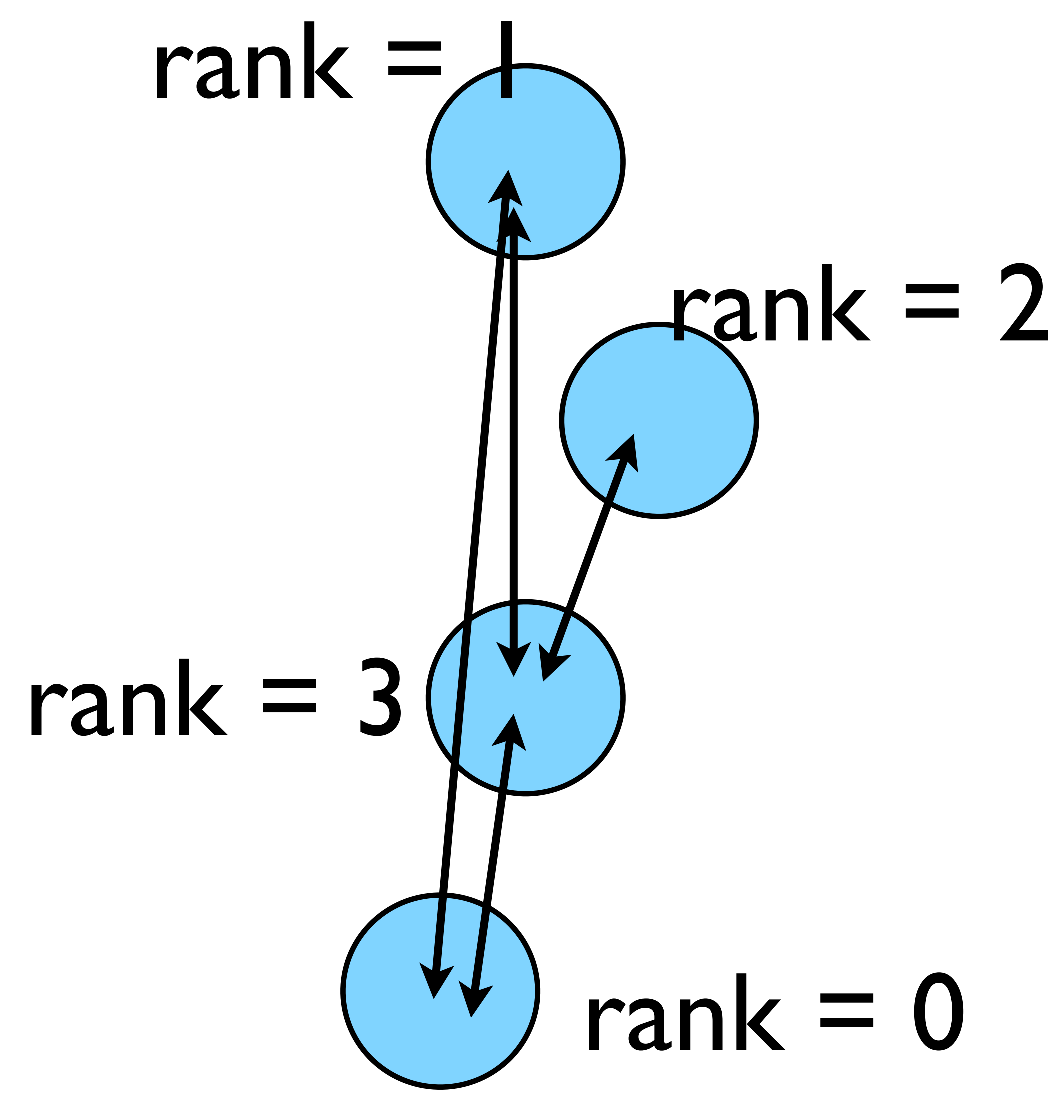
get the size of communicator, the current tasks's rank within communicator.

put answers in `rank` and `size`

```
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
```

# 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;
        int ierr;

        ierr = MPI_Init(&argc, &argv);

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

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

        MPI_Finalize();

        return 0;
}
```

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

integer :: rank, 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

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

      integer :: rank, comsize      ! standard MPI stuff
      integer :: ierr
      integer :: sendto, recvfrom   ! PE # to send, recv from
      integer,parameter :: ourtag=1 ! shared label for messages
      character(5) :: sendmessage   ! buffer for sending, recieving
      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, reciever
          print *,' FAIL: only one task'
          call MPI_Abort(MPI_COMM_WORLD,1)
      endif

      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(hearmessage, 5, MPI_CHARACTER, recvfrom, &
                        ourtag, MPI_COMM_WORLD, status, ierr)
          print *, rank, ': got message <',hearmessage,'>.'
      endif

      call MPI_FINALIZE(ierr)

      end
```

# 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}

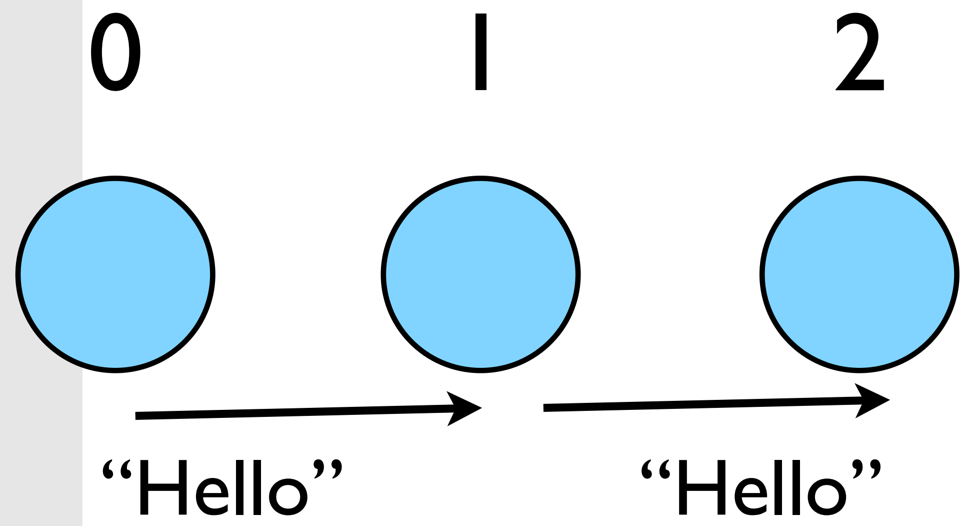
```fortran
program hellompiworld
implicit none
include "mpif.h"

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, recieving
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, reciever
    print *,' FAIL: only one task'
    call MPI_Abort(MPI_COMM_WORLD,1)
endif

leftneighbour  = rank - 1
rightneighbour = rank + 1

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

call MPI_FINALIZE(ierr)

end
```

# Compile and run

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

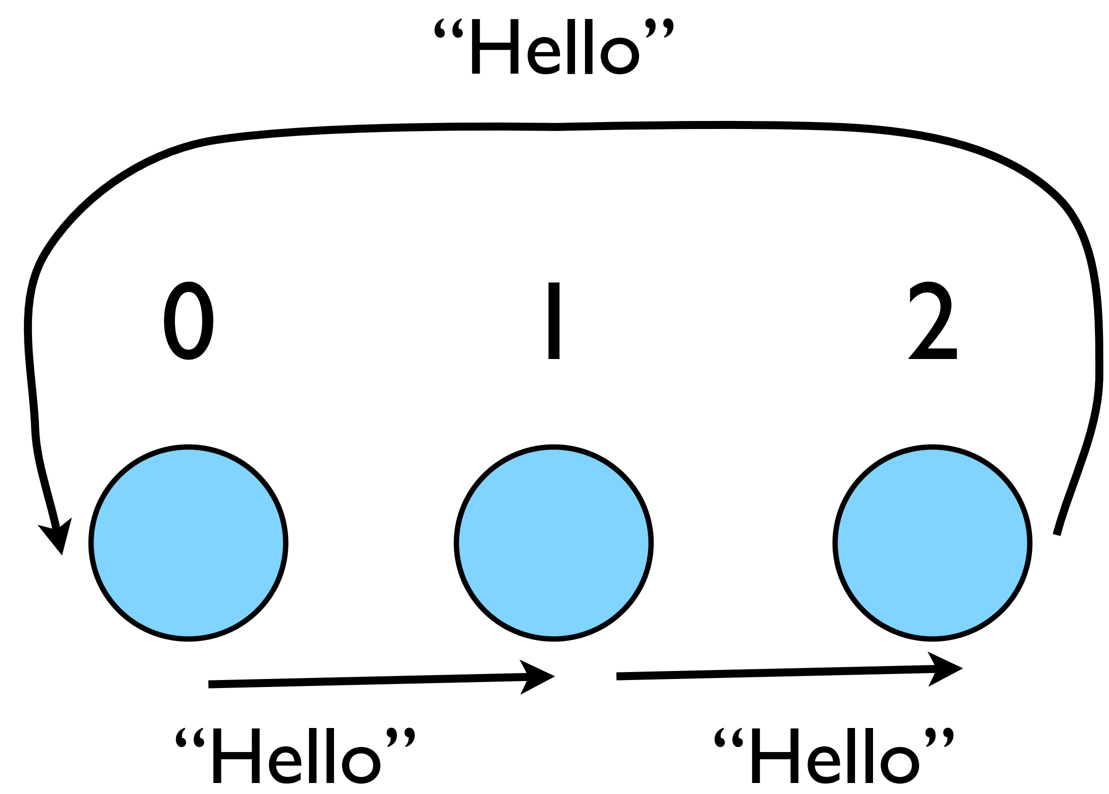
```
ljdursi|segfault.local> mpirun -np 4 ./secondmessage
           3 : got message <Hello>.
           2 : sent message <Hello>.
           2 : got message <Hello>.
           1 : sent message <Hello>.
           0 : sent message <Hello>.
           1 : got message <Hello>.
```

```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;
}
```

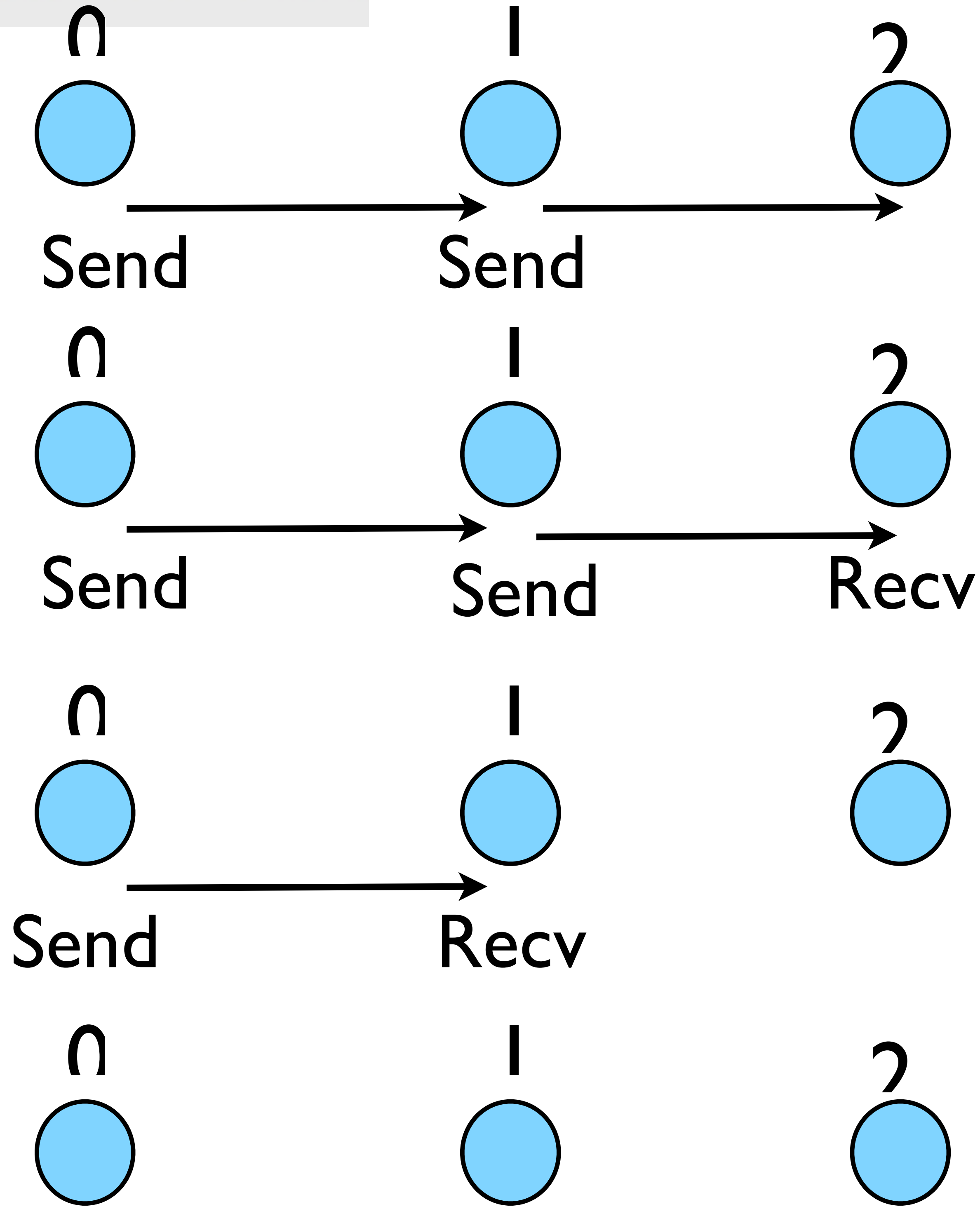


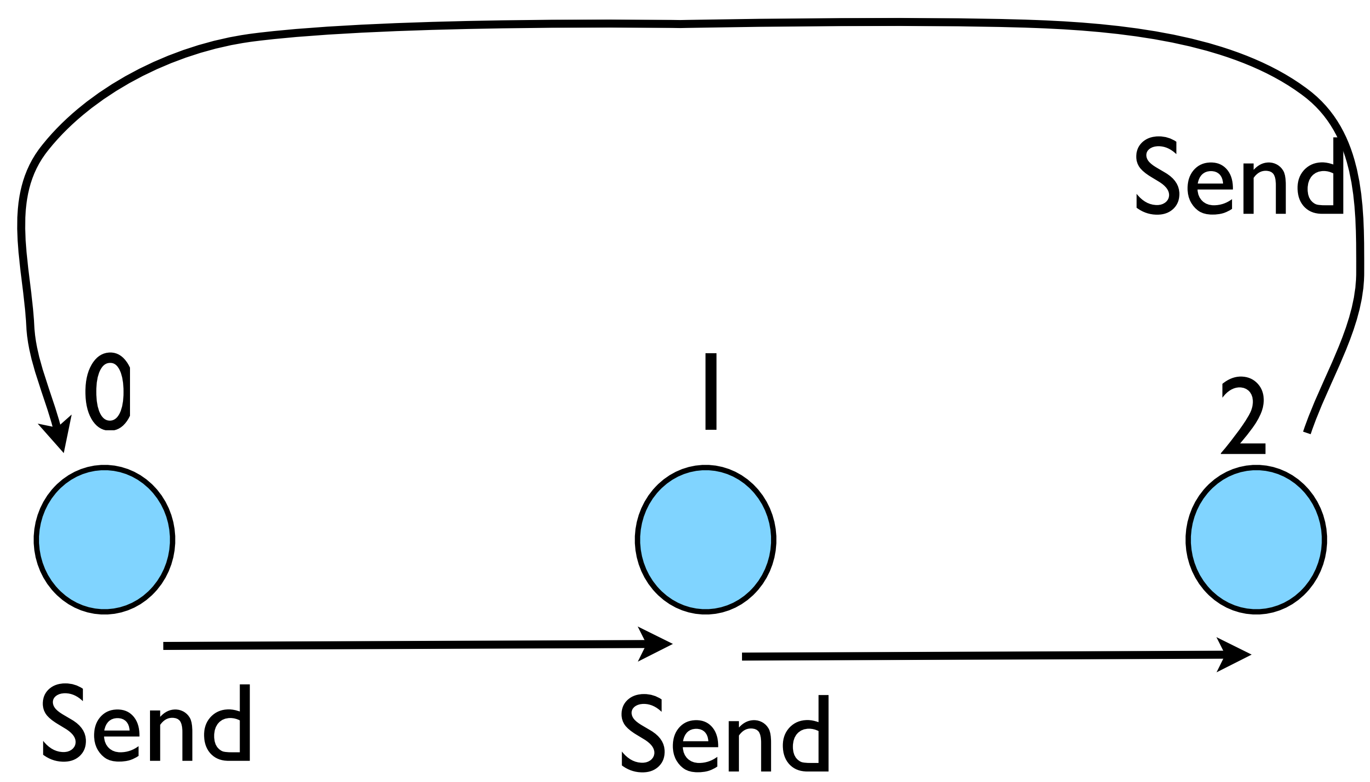
# Implement periodic boundary conditions

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



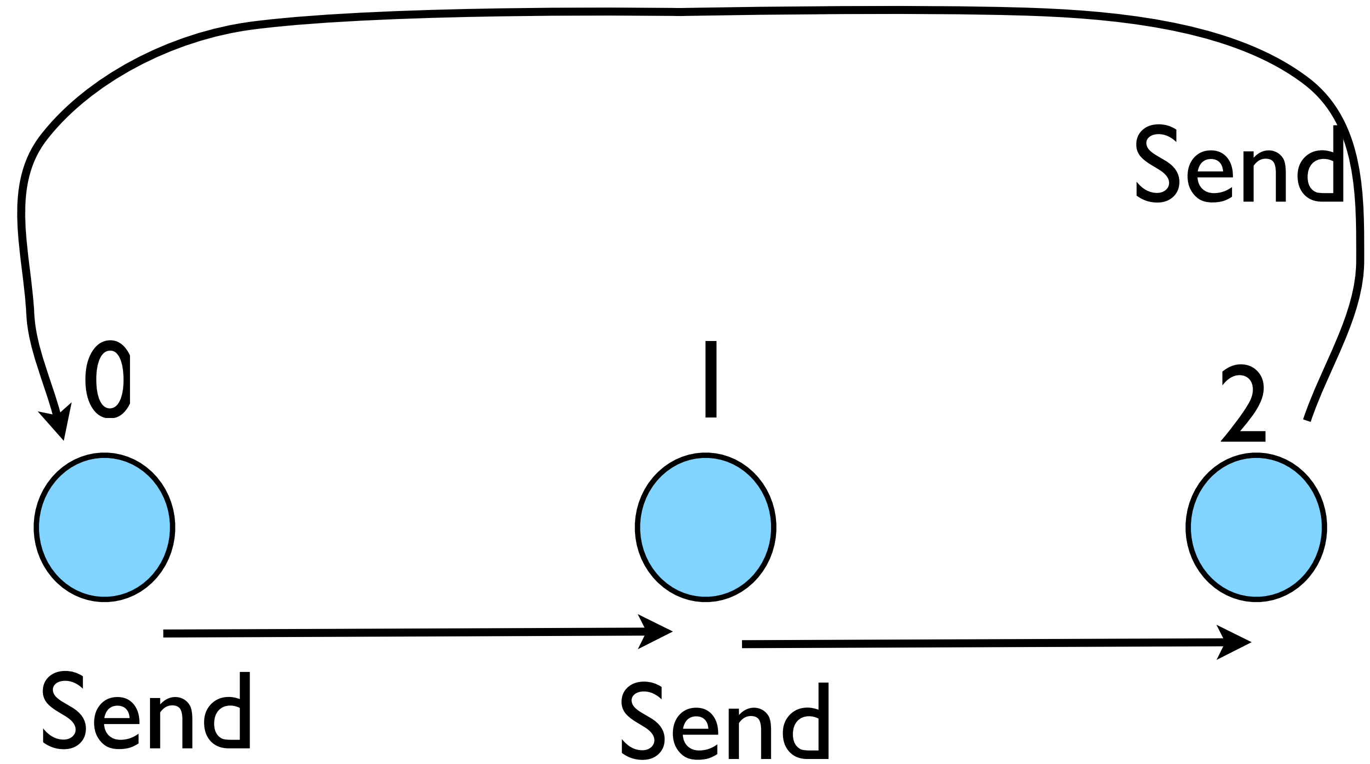
```
if (rightneighbour < size) {
    ierr = MPI_Ssend(sendmessage, 6, MPI_CHAR, rightneighbour,
                     OURTAG, MPI_COMM_WORLD);
    printf("%d: Sent message <%s> to %d \n", rank, sendmessage, rightneighbour);
}
if (leftneighbour >= 0) {
    ierr = MPI_Recv(hearmessage, 6, MPI_CHAR, leftneighbour,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

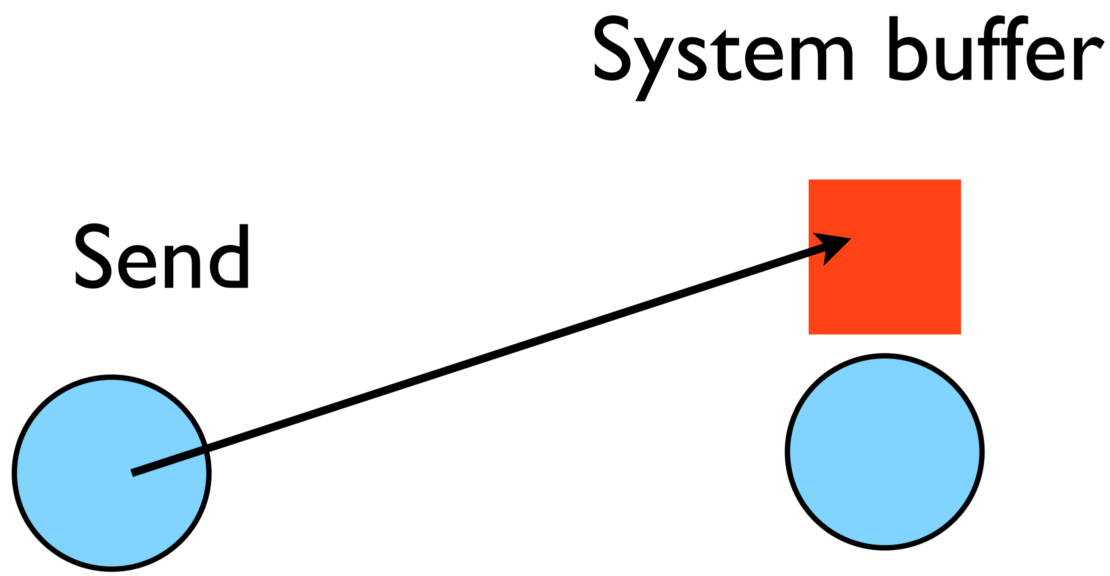
# Big MPI
# Lesson #1

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

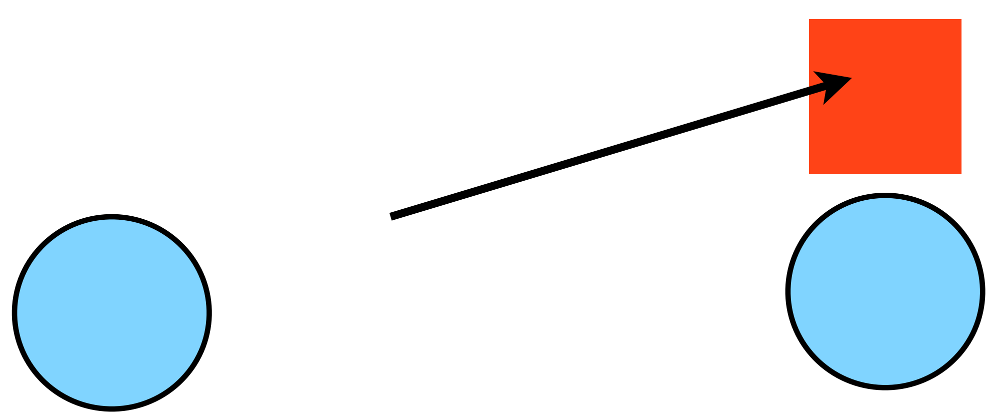
# Different versions of SEND

- 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**

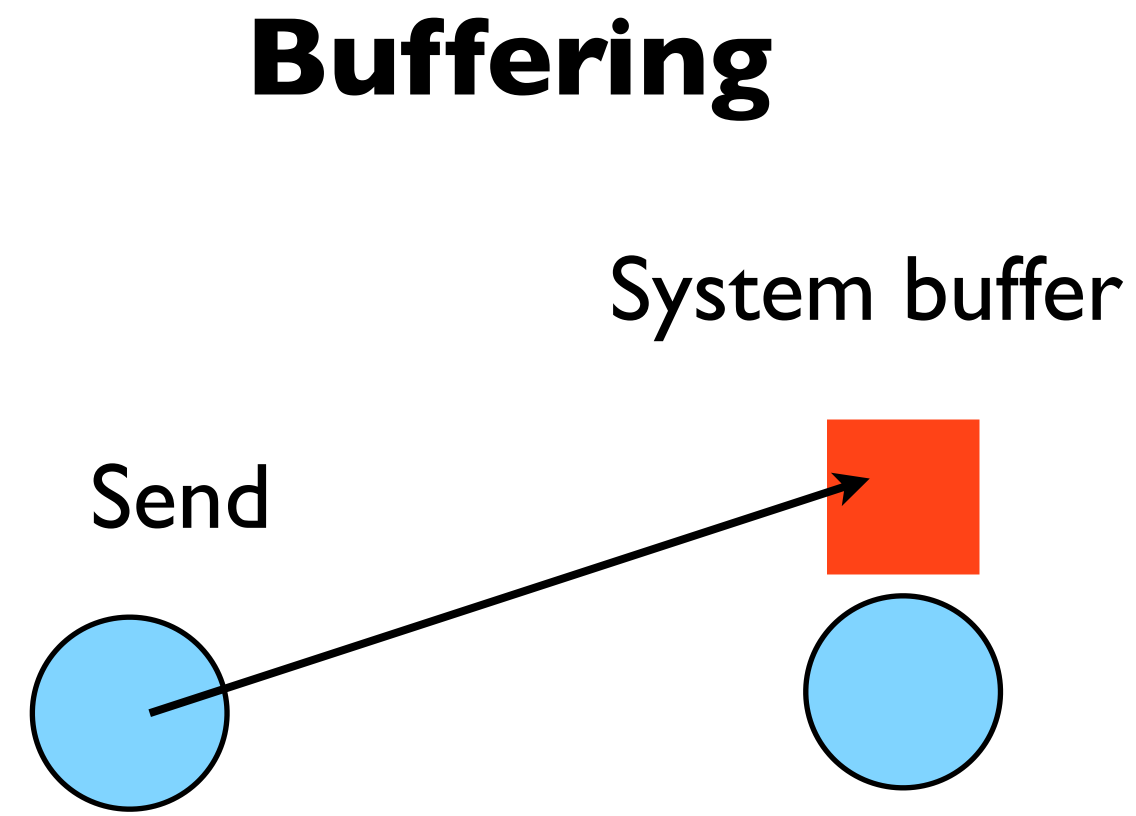



**(Non) Blocking**

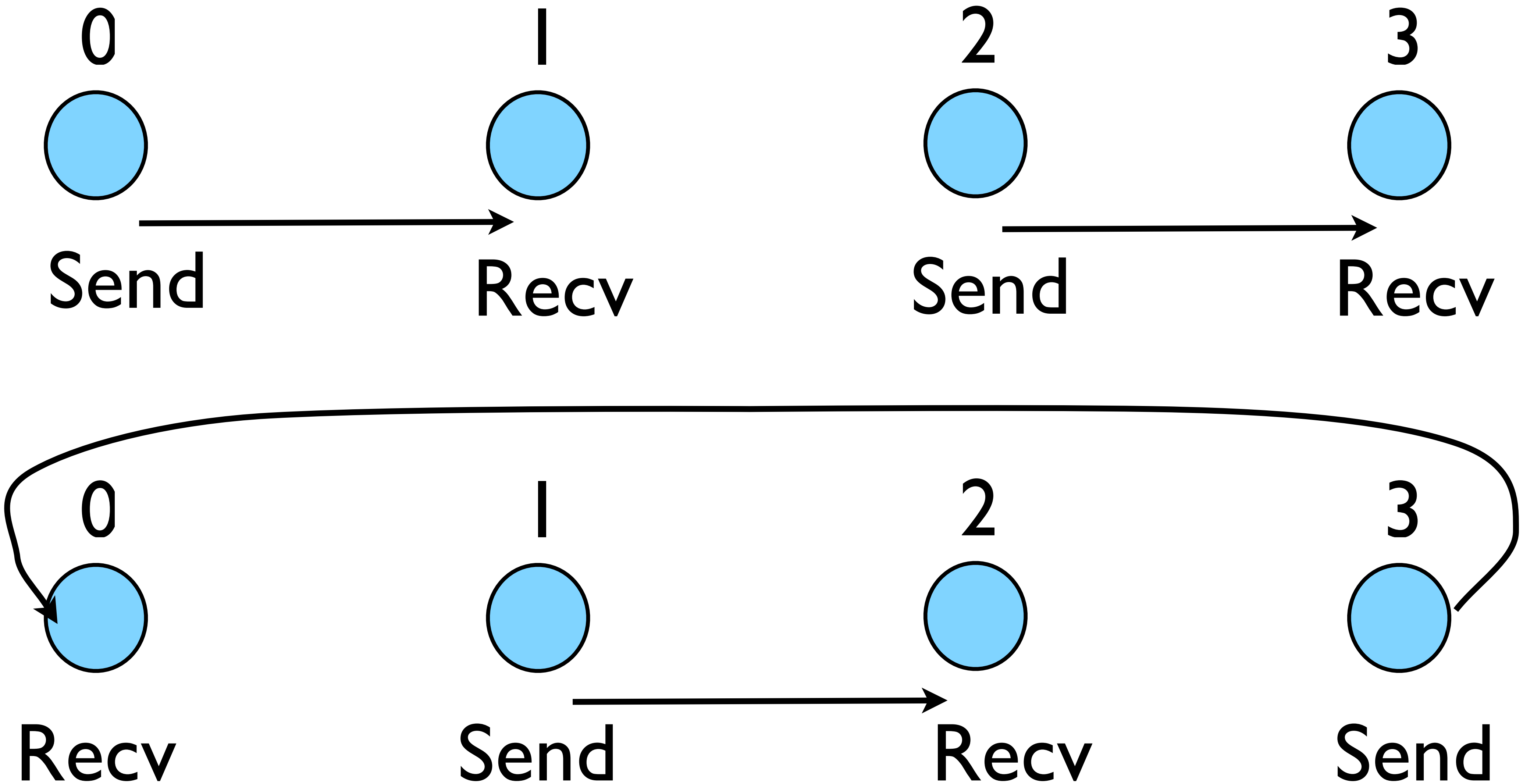


# Buffering is dangerous!

- 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)



# 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 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.

```fortran
      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)

      sendmessage = 'Hello'
      call MPI_SENDRECV(sendmessage, 5, MPI_CHARACTER, rightneighbour, ourtag,
                        hearmessage, 5, MPI_CHARACTER, leftneighbour,  ourtag,
                        MPI_COMM_WORLD, status, ierr)

      print *, rank, ': sent message <',sendmessage,'> to ', rightneighbour
      print *, rank, ': got message <',hearmessage,'> from ', leftneighbour

      call MPI_FINALIZE(ierr)

      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

Send Args

```
MPI_Status status;

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

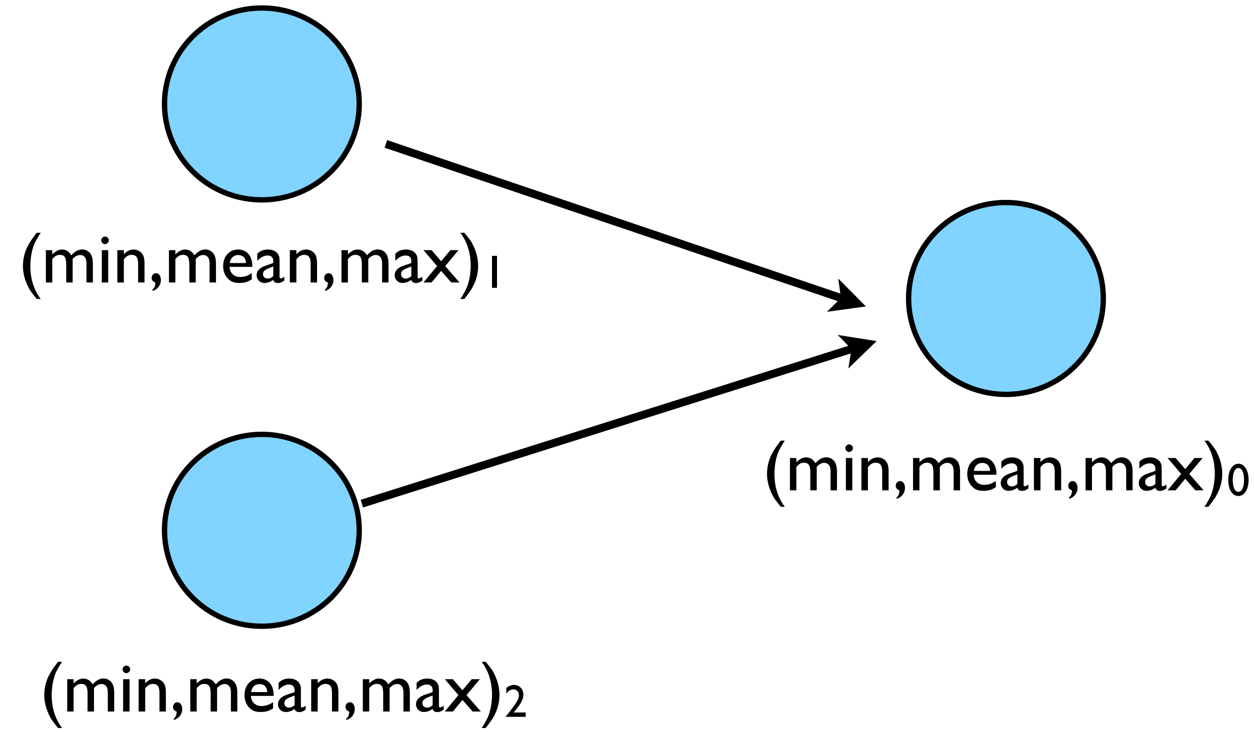
Recv Args

## 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?

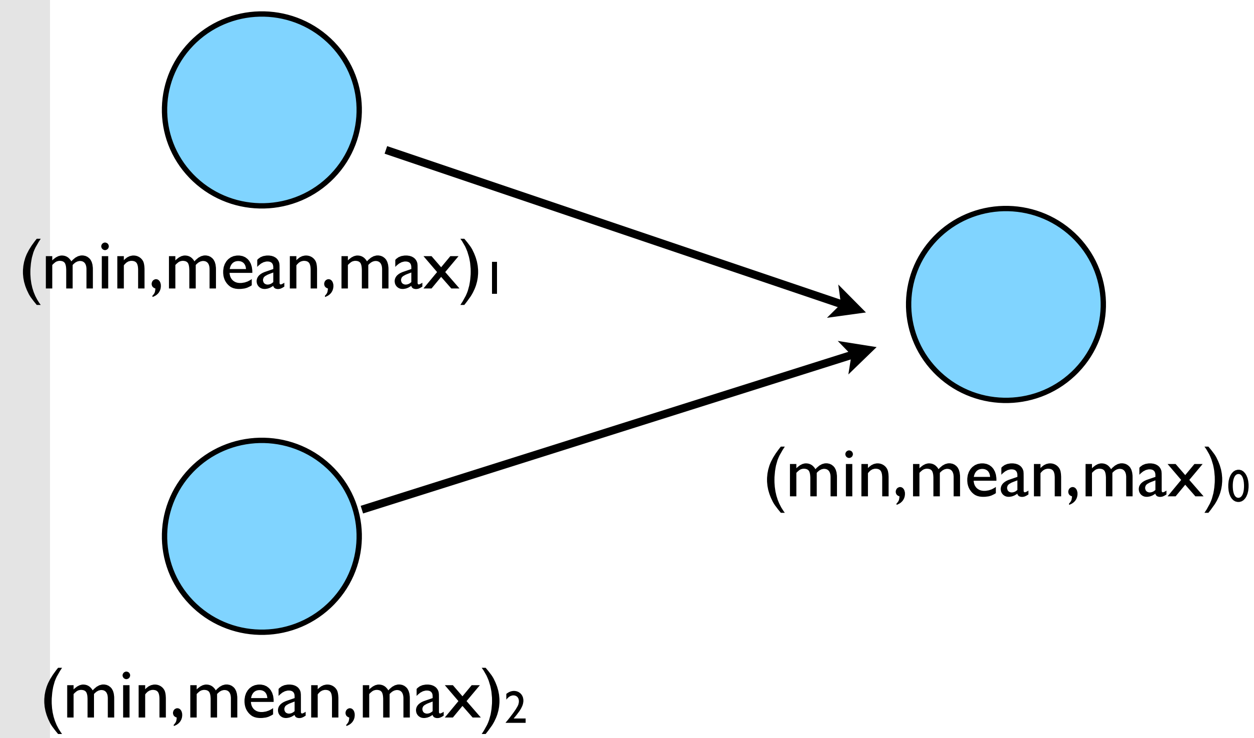
# Min, Mean, Max of numbers




- 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?)
- ~ljdursi/ss2010/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 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
```

```fortran
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
      if (myid .ne. 0) then
          datapack(1) = datamin
          datapack(2) = datamean
          datapack(3) = datamax
          call MPI_SSEND(datapack,3,MPI_REAL,0,1,MPI_COMM_WORLD,ierr)
      else
          globmin  = datamin
          globmax  = datamax
          globmean = datamean
          do proc=1,nprocs-1
             call MPI_RECV(datapack, 3, MPI_REAL, MPI_ANY_SOURCE, 1,
     &                     MPI_COMM_WORLD, status, ierr)
             if (datapack(1) .lt. globmin) globmin=datapack(1)
             globmean = globmean + datapack(2)
             if (datapack(3) .gt. globmax) globmax=datapack(3)
          enddo
          globmean = globmean/nprocs
          print *,'Global min/mean/max=',globmin,globmean,globmax
      endif

      call MPI_FINALIZE(ierr)
      return
      end
```

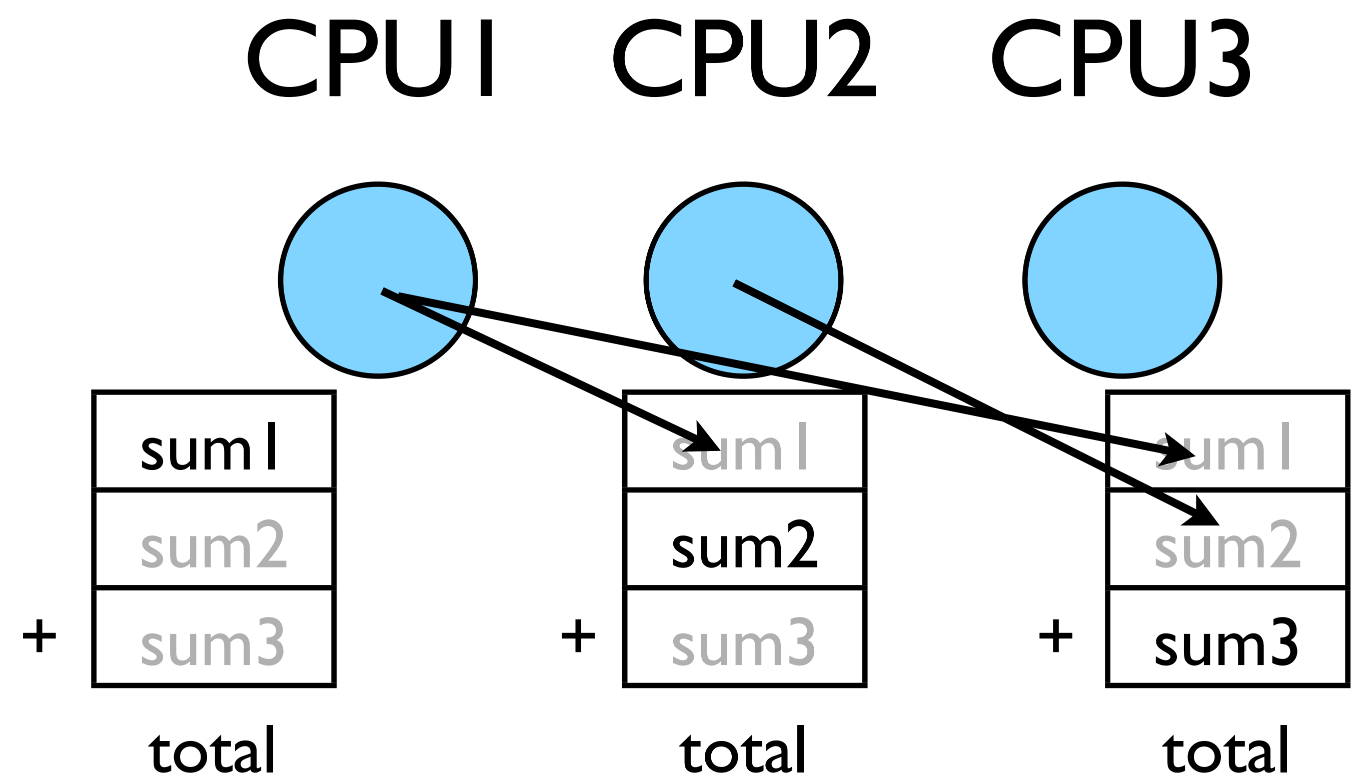



Q: are these sends/recvd adequately paired?

minmeanmax-mpi.f

# Inefficient!

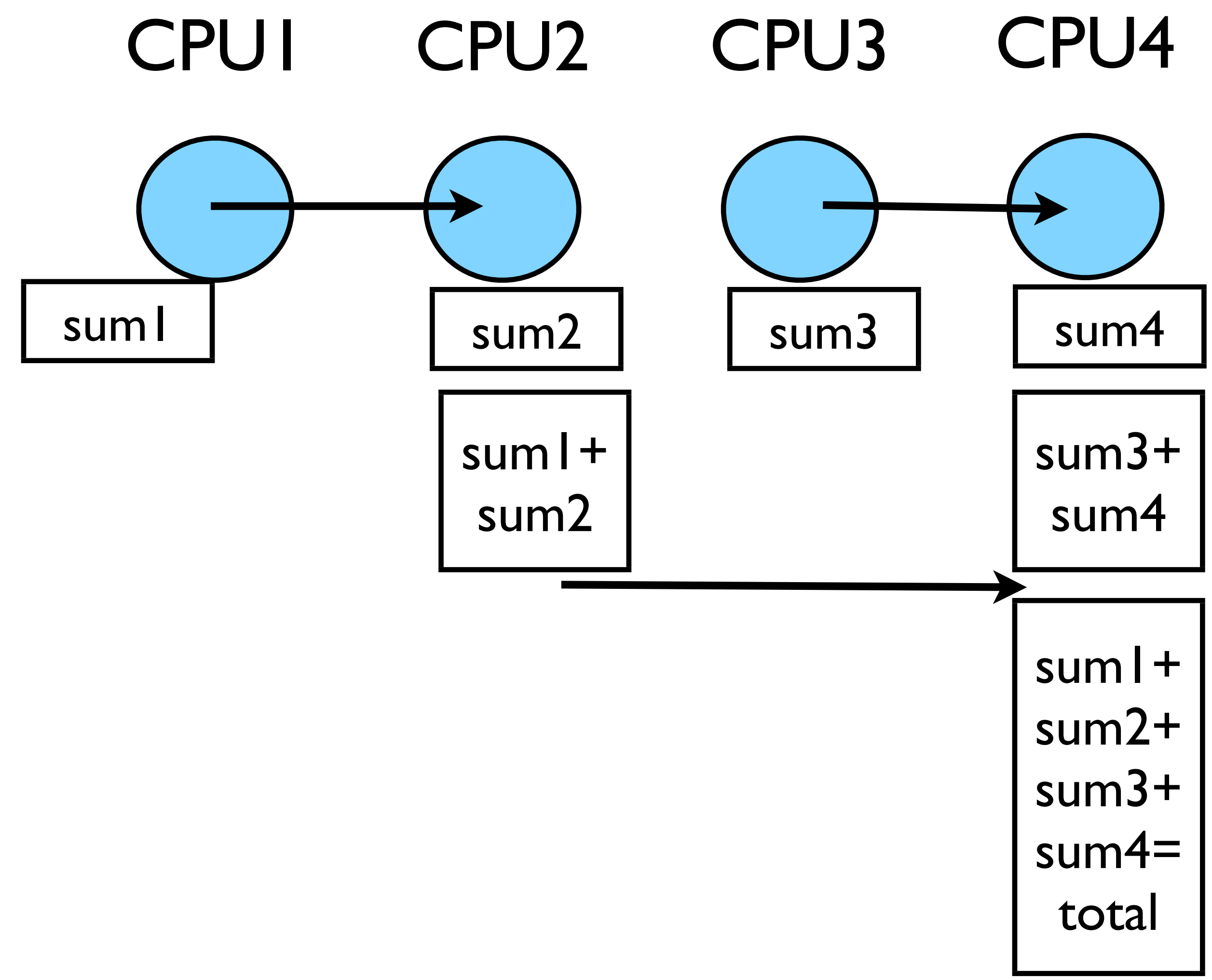
- Requires (P-1) messages, 2 (P-1) if everyone then needs to get the answer.



# 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...)

```fortran
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'