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

An introduction to MPI

SciNet Parallel Scientific Computing Course

Aug 31 - Sept 4, 2009

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

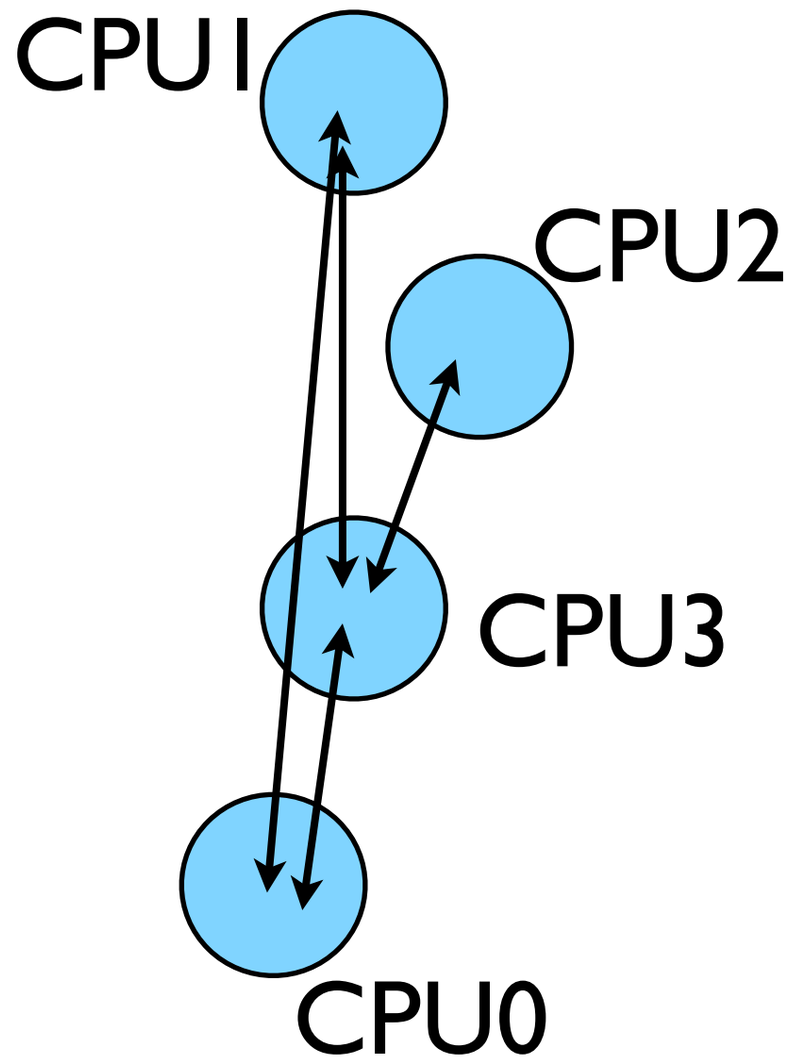
```c
#include <stdio.h>
#include <mpi.h>

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;
}
```

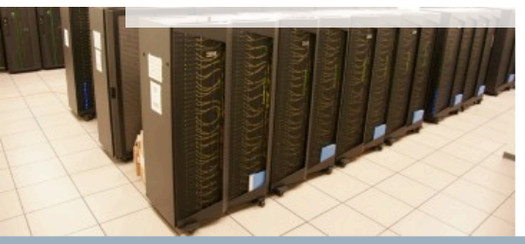
Fortran

```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**

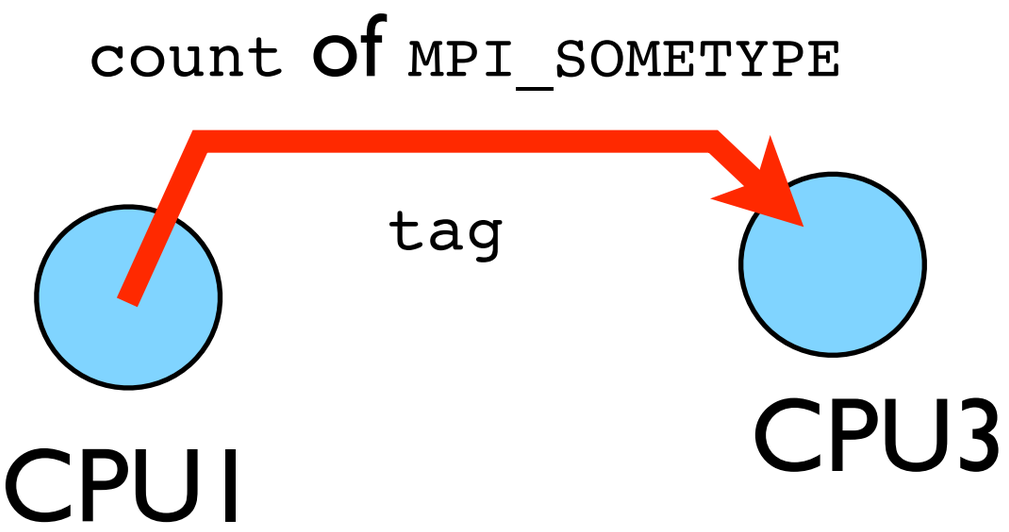
- Communication/coordination between tasks done by sending and receiving messages.
- Each message involves a function call from each of the programs.

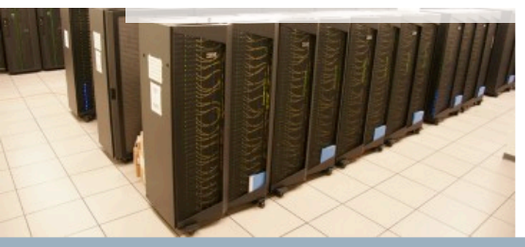




Messages

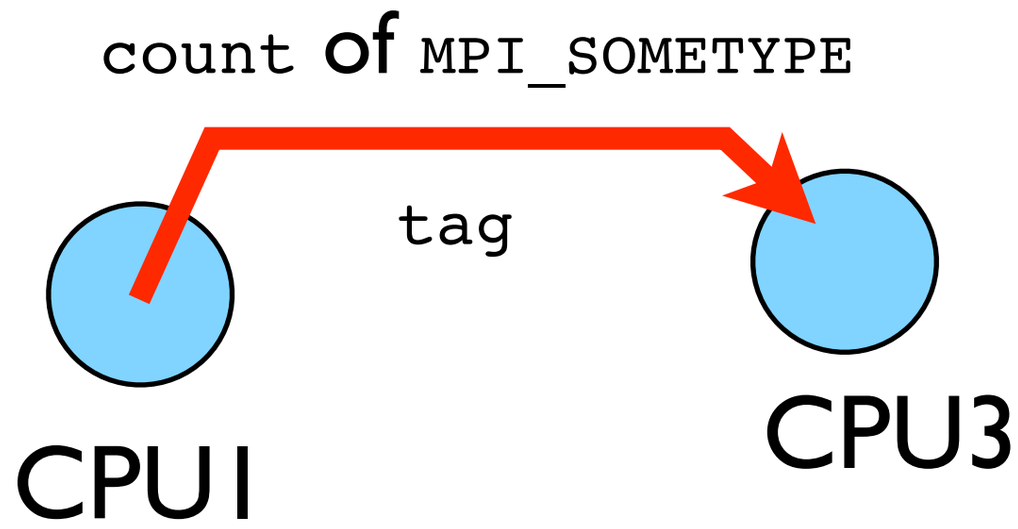
- Messages have a **sender** and a **receiver**
- When you are sending a message, don't need to specify sender (it's the current processor),
- A sent message has to be actively received by the receiving process

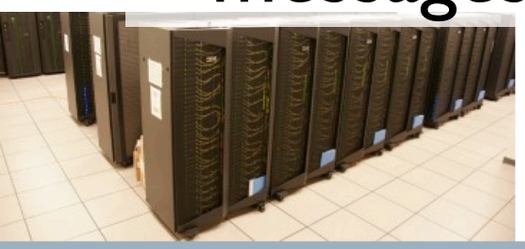




Messages

- 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 6-10, use more as needed.

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

Hello World

- The obligatory starting point
- Let's compile and run it together

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
```

C

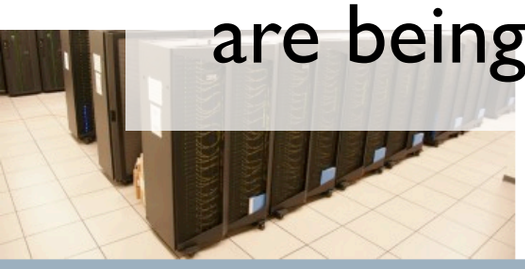
```
#include <stdio.h>
#include <mpi.h>

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;
}
```

```
$ cd ~/pca/src/mpi-intro
edit hello-world.c or .f
$ mpif77 hello-world.f -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

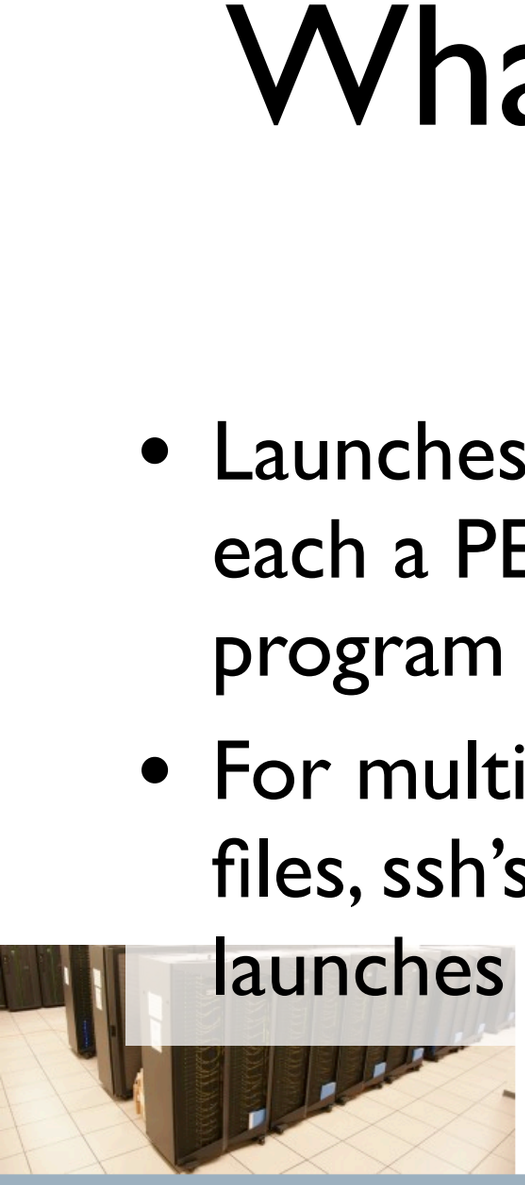
- Just wrappers for the system C, Fortran compilers that have the various -I, -L clauses in there automaticaly
- --showme option shows (for this MPI version) which flags are being used

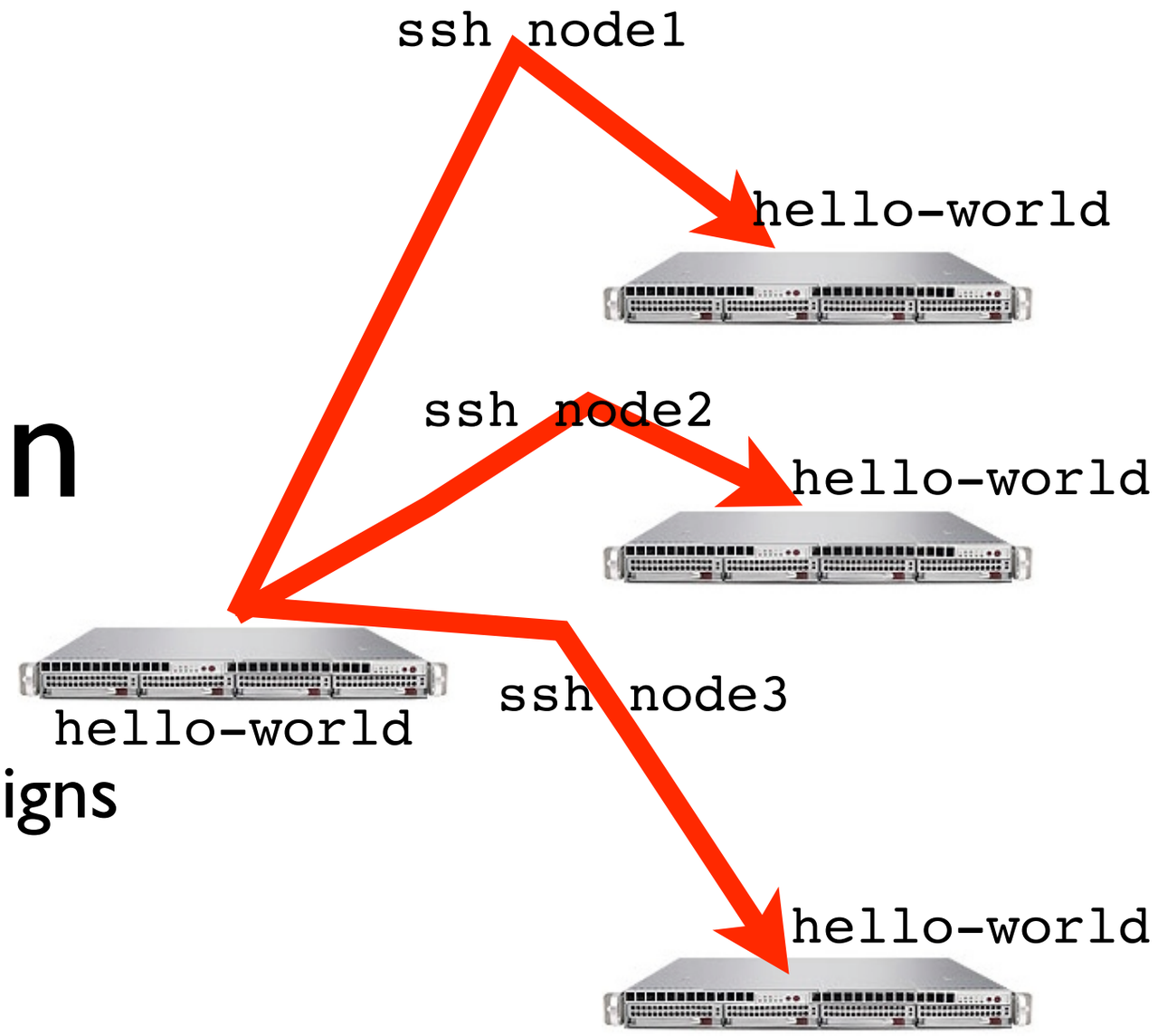
```
$ mpicc --showme hello-
world.c -o hello-world

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 a PE and starts the program
- For multinode run, has a list of files, 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?

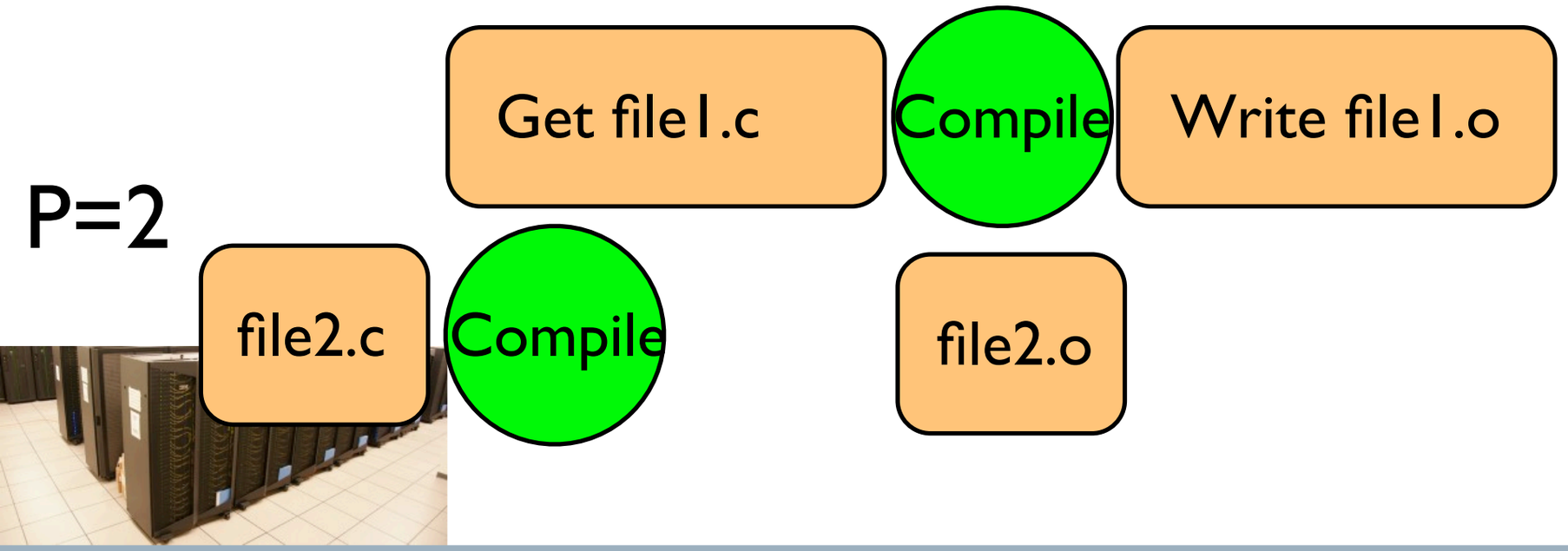
```
$ mpirun -np 24 hello-world
```

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

```
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
```

- (FORTRAN version; C is similar)

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

```
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
```

`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.

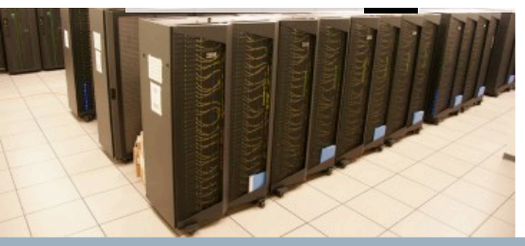
`ierr:` Returns any error code.

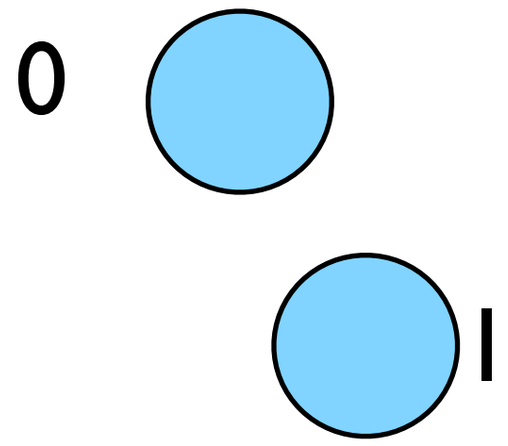
```
program hellompiworld
include "mpif.h"

integer rank, 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
```

`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 belongs to `MPI_COMM_WORLD`

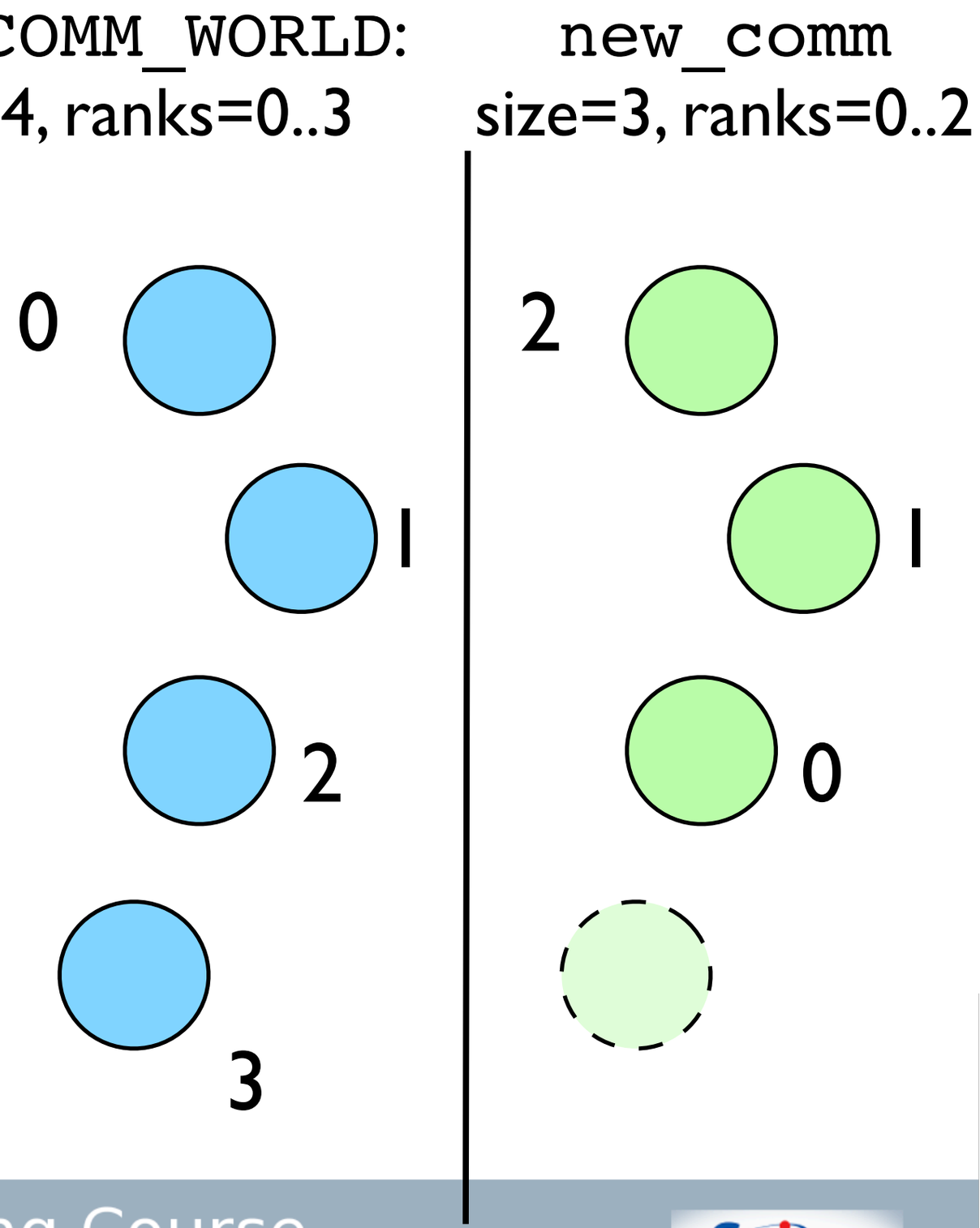




`MPI_COMM_WORLD`:
size=4, ranks=0..3

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



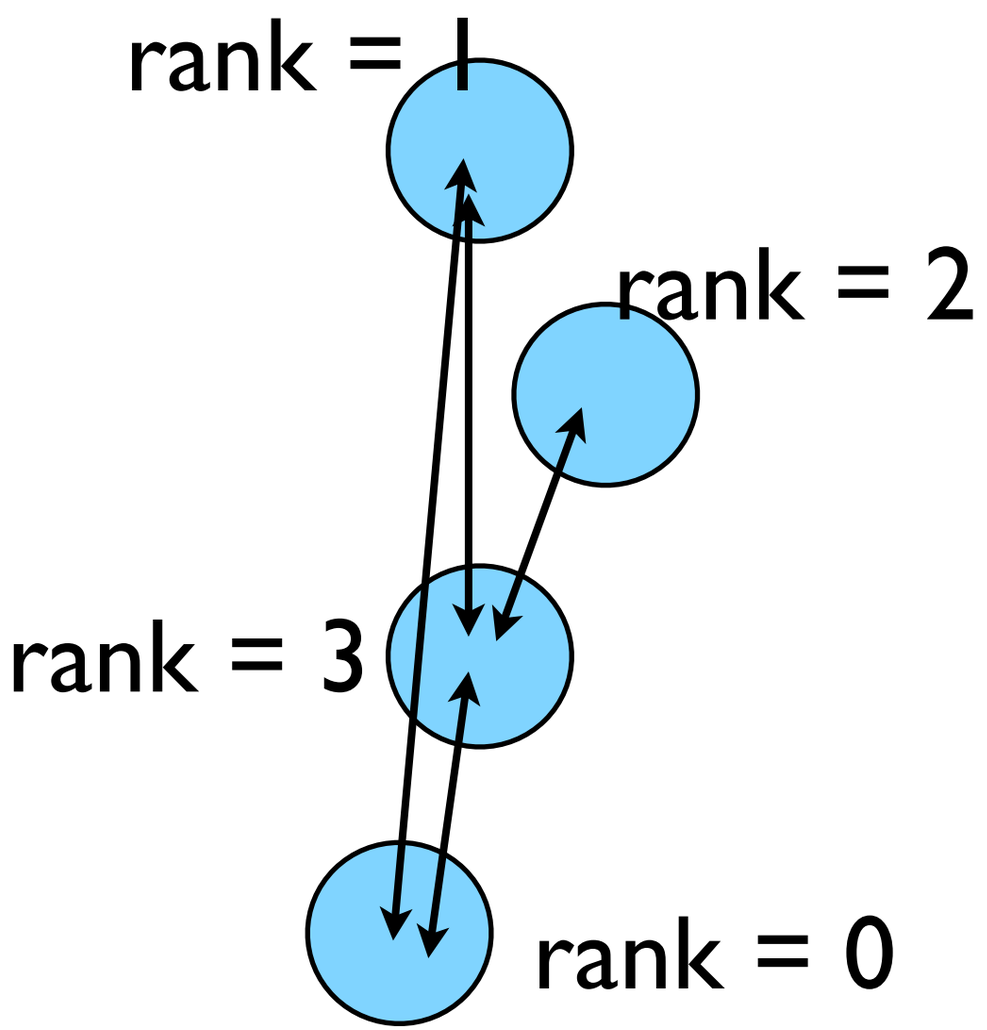
```
program hellompiworld
include "mpif.h"

integer rank, 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
```

`call MPI_COMM_RANK`, `call MPI_COMM_SIZE`: get the size of `MPI_COMM_WORLD` and the current tasks's rank within `MPI_COMM_WORLD`

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;
}
```

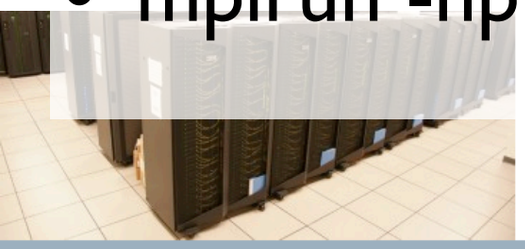
Fortran

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

integer rank, 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
```

- Fortran: All caps
- 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



```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 */
        char sendmessage[]="Hello";   /* send from here*/
        int sendto;             /* PE # we send to */
        int recvfrom;           /* PE # we recv from */
        const int OURTAG = 1;   /* our shared tag */
        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);

        assert(size > 1);       /* won't work otherwise */

        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.f
- mpif77 -o firstmessage firstmessage.f
- mpirun -np 2 ./ firstmessage

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

      integer rank, size
      integer ierr
      integer sendto
      integer recvfrom
      integer ourtag
      parameter (ourtag=1)
      character(5) hearmessage
      character(5) sendmessage
      integer status(MPI_STATUS_SIZE)

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

      if (size .le. 1) then
          print *, 'FAIL: only one task'
      else
          if (rank .eq. 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 .eq. 1) then
              recvfrom = 0
              call MPI_RECV(hearmessage, 5, MPI_CHARACTER, recvfrom,    &
     &                      ourtag, MPI_COMM_WORLD, status, ierr)
              print *, rank, ': got message <',hearmessage,'>.'
          endif
       endif
       call MPI_FINALIZE(ierr)

       return
       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, destination,tag,
                Communicator, status);
```

Fortran - Send and Receive

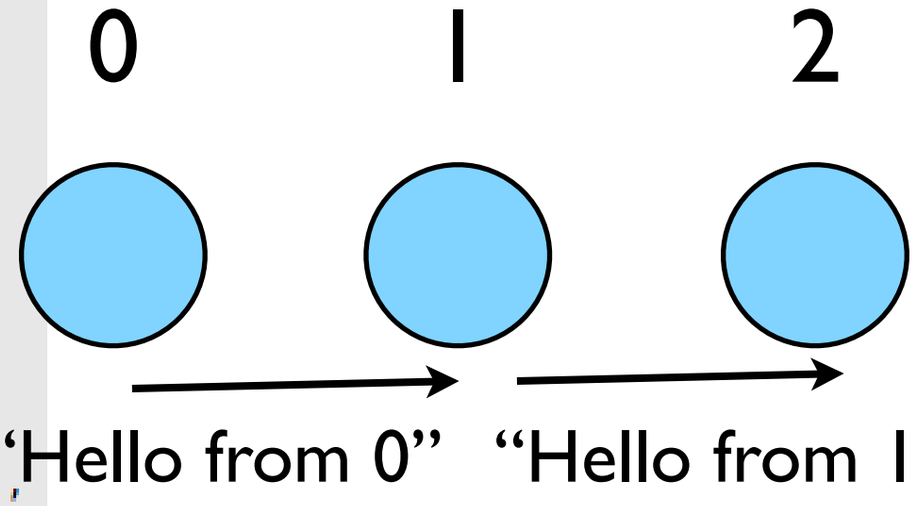
```
integer status(MPI_STATUS_SIZE)

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

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

More complicated example:

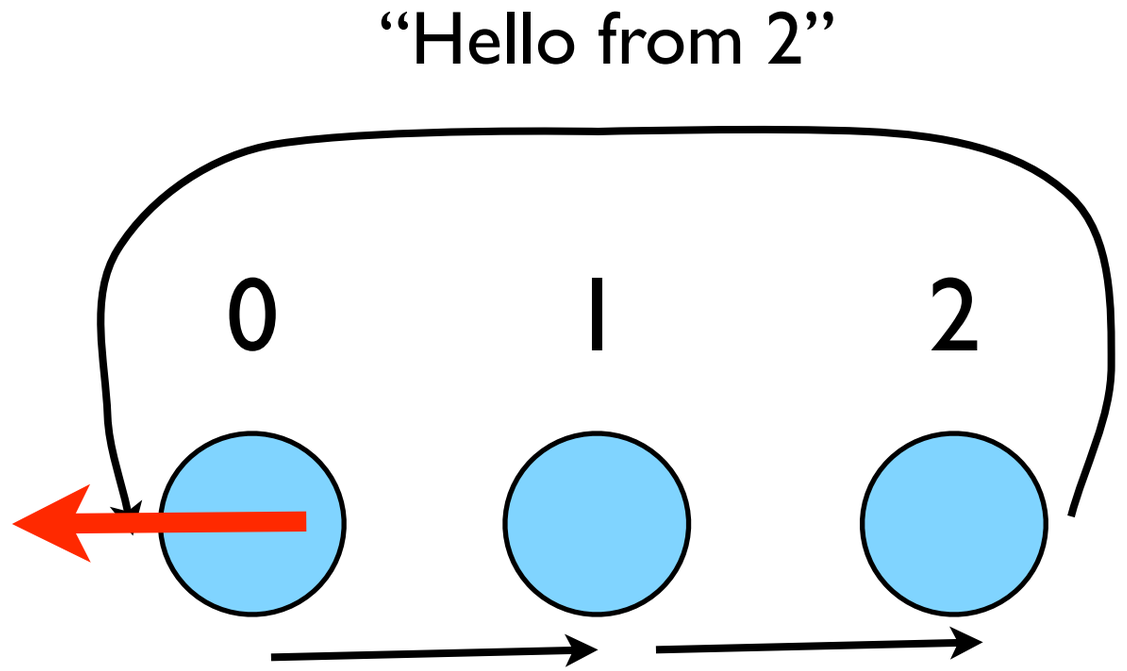
- make hello-world-send
- mpirun -np 4 hello-world-send

```
      program hellompiworld

      implicit none
      include "mpif.h"

      integer rank, size
      integer ierr
      integer leftneighbour
      integer rightneighbour
      integer tag
      integer status(MPI_STATUS_SIZE)
      character(15) greeting
      character(15) received

      call MPI_INIT(ierr)

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

      leftneighbour  = rank-1
      rightneighbour = rank+1

      if (leftneighbour.ge.0) print *,rank,': My left neighbor is ',
     &                            leftneighbour
      if (rightneighbour.lt.size)print *,rank,': My right neighbor is ',
     &                            rightneighbour
      print *, rank, ': lets say Hi to our right!'

      write(greeting,'(a i4)'), 'Hello from', rank
      tag = 1

      if (rightneighbour .lt. size) then
          call MPI_SSEND(greeting, 15, MPI_CHARACTER, rightneighbour,
     &                tag, MPI_COMM_WORLD, ierr)
      endif

      print *, rank, ": ...and see if we are greeted!"

      if (leftneighbour .ge. 0) then
          call MPI_RECV(received, 15, MPI_CHARACTER, leftneighbour,
     &                tag, MPI_COMM_WORLD, status, ierr)
          print *, rank, ": leftneighbor says <",received,">!"
      endif

      call MPI_FINALIZE(ierr)

      return
      end
```

```
integer leftneighbour
integer rightneighbour
integer tag
integer status(MPI_STATUS_SIZE)
character(15) greeting
character(15) received

call MPI_INIT(ierr)

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

leftneighbour  = rank-1
rightneighbour = rank+1

if (leftneighbour.ge.0) print *,rank,': My left neighbor is ',
&                         leftneighbour
if (rightneighbour.lt.size)print *,rank,': My right neighbor is ',
&                         rightneighbour
print *, rank, ': lets say Hi to our right!'

write(greeting,'(a i4)'), 'Hello from', rank
tag = 1

if (rightneighbour .lt. size) then
    call MPI_SSEND(greeting, 15, MPI_CHARACTER, rightneighbour,
&              tag, MPI_COMM_WORLD, ierr)
endif

print *, rank, ": ...and see if we are greeted!"

if (leftneighbour .ge. 0) then
    call MPI_RECV(received, 15, MPI_CHARACTER, leftneighbour,
&              tag, MPI_COMM_WORLD, status, ierr)
    print *, rank, ": leftneighbor says <",received,">!"
endif

call MPI_FINALIZE(ierr)

return
end
```



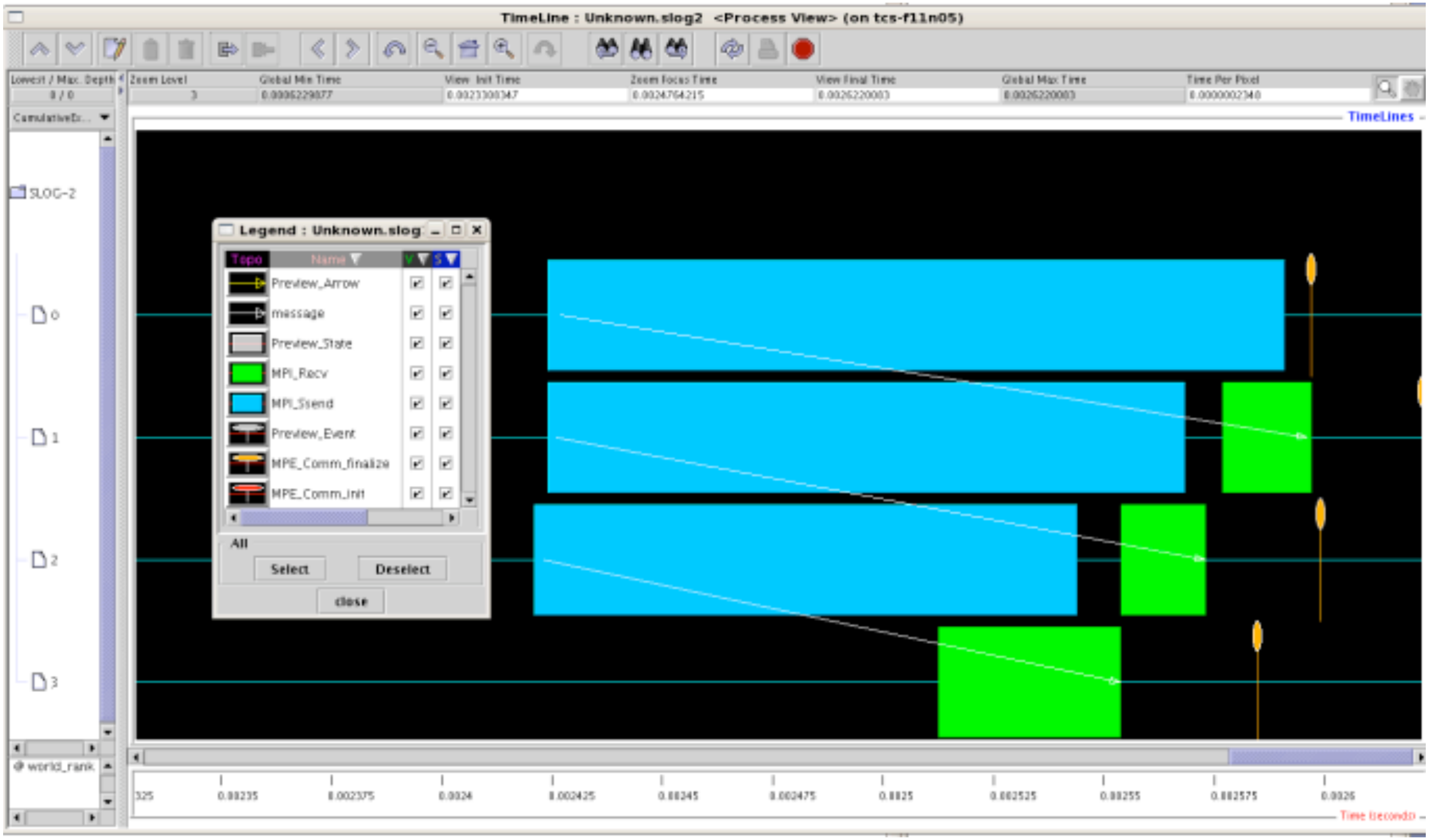
Implement periodic boundary conditions



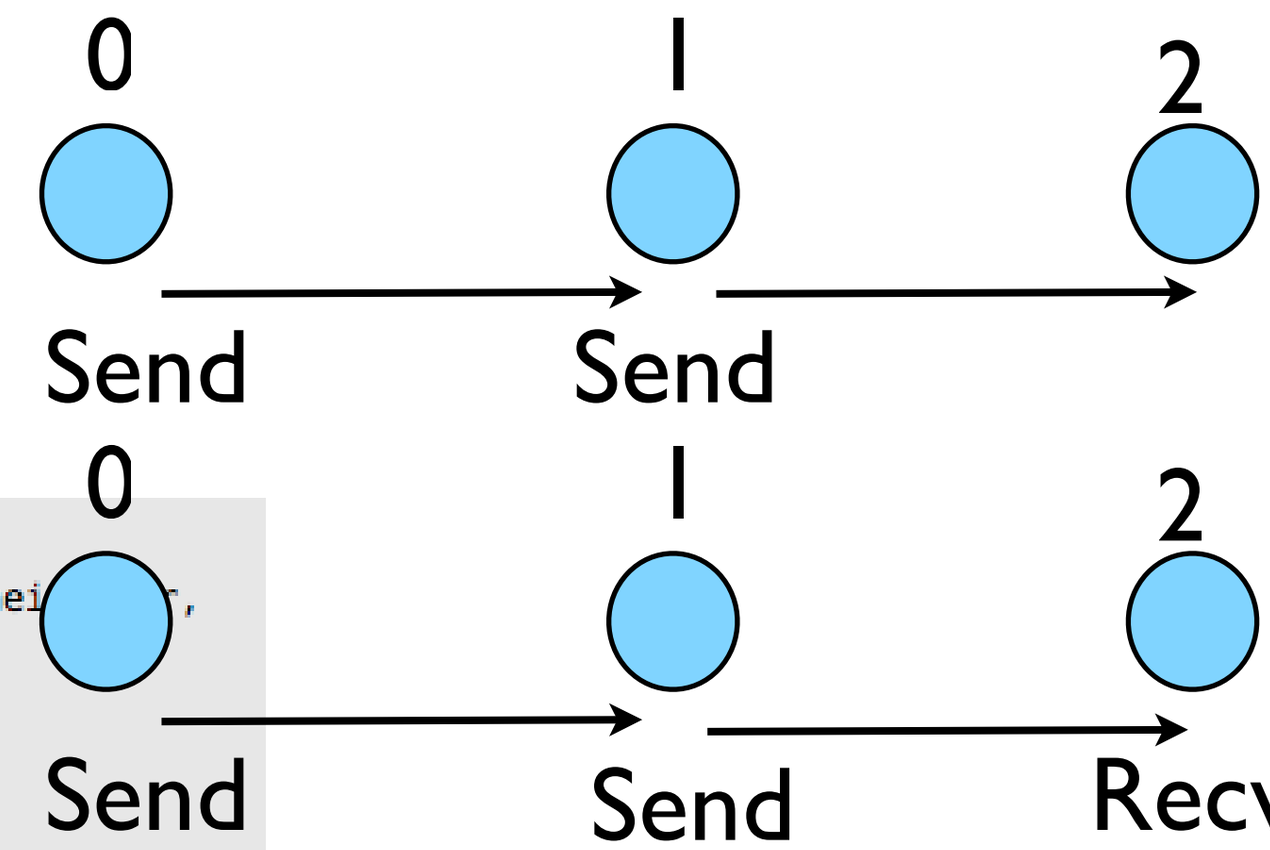
"Hello from 0" "Hello from 1"

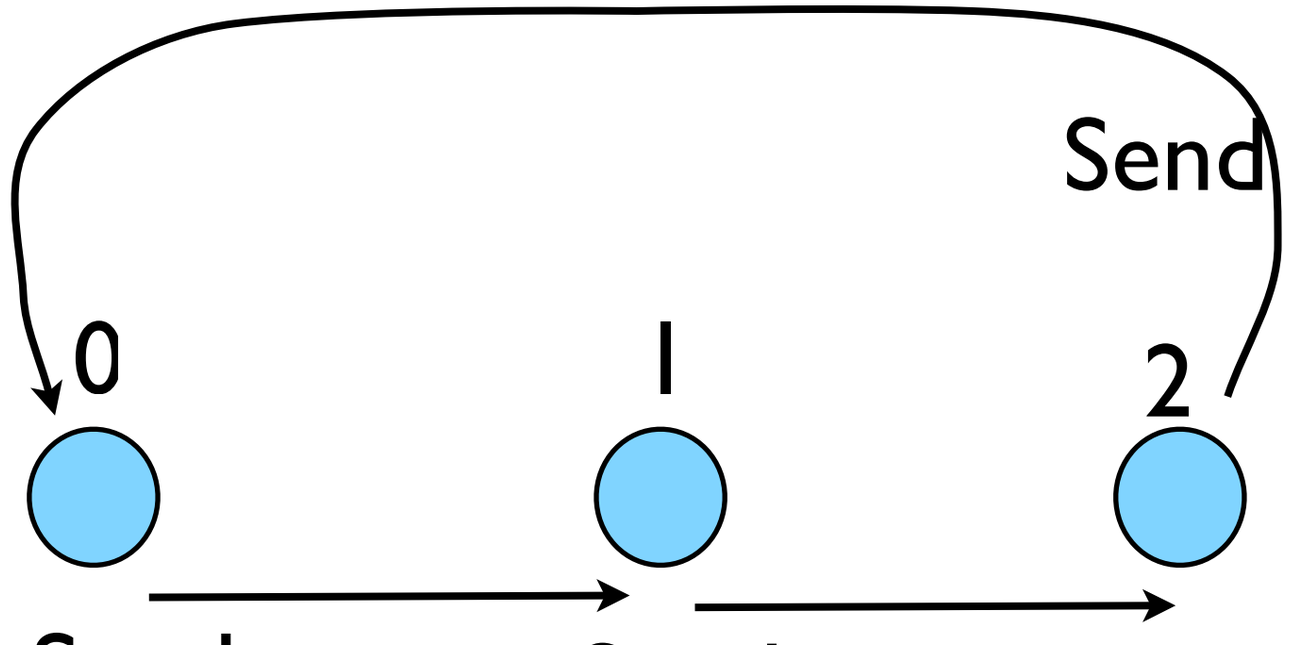
- cp hello-world-send.c hello-world-send-around.c
- edit so it `wraps around'
- make hello-world-send-around
- mpirun -np 3 hello-world-send-around

- make hello-world-send-around
- mpirun -np 3 hello-world-send-around


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
TimeLines
SLOG-2
0
1
2
3
world_rank
Legend : Unknown.slog
Topo
Name
V
S
Preview_Arrow
message
Preview_State
MPI_Recv
MPI_Send
Preview_Event
MPE_Comm_finalize
MPE_Comm_Init
All
Select
Deselect
close
Time (seconds)

```
 if (rightneighbour .lt. size) then
     call MPI_SSEND(greeting, 15, MPI_CHARACTER, rightnei     r,
&                tag, MPI_COMM_WORLD, ierr)
 endif

 print *, rank, ": ...and see if we are greeted!"

 if (leftneighbour .ge. 0) then
     call MPI_RECV(received, 15, MPI_CHARACTER, leftneighbour,
&                tag, MPI_COMM_WORLD, status, ierr)
     print *, rank, ": leftneighbor says <",received,">!"
 endif

 call MPI_FINALIZE(ierr)

 return
 end
```

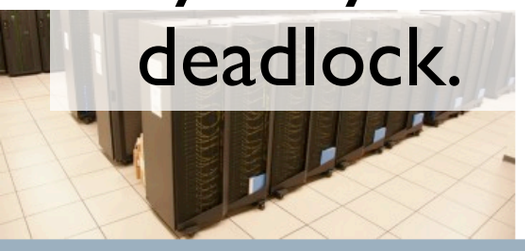


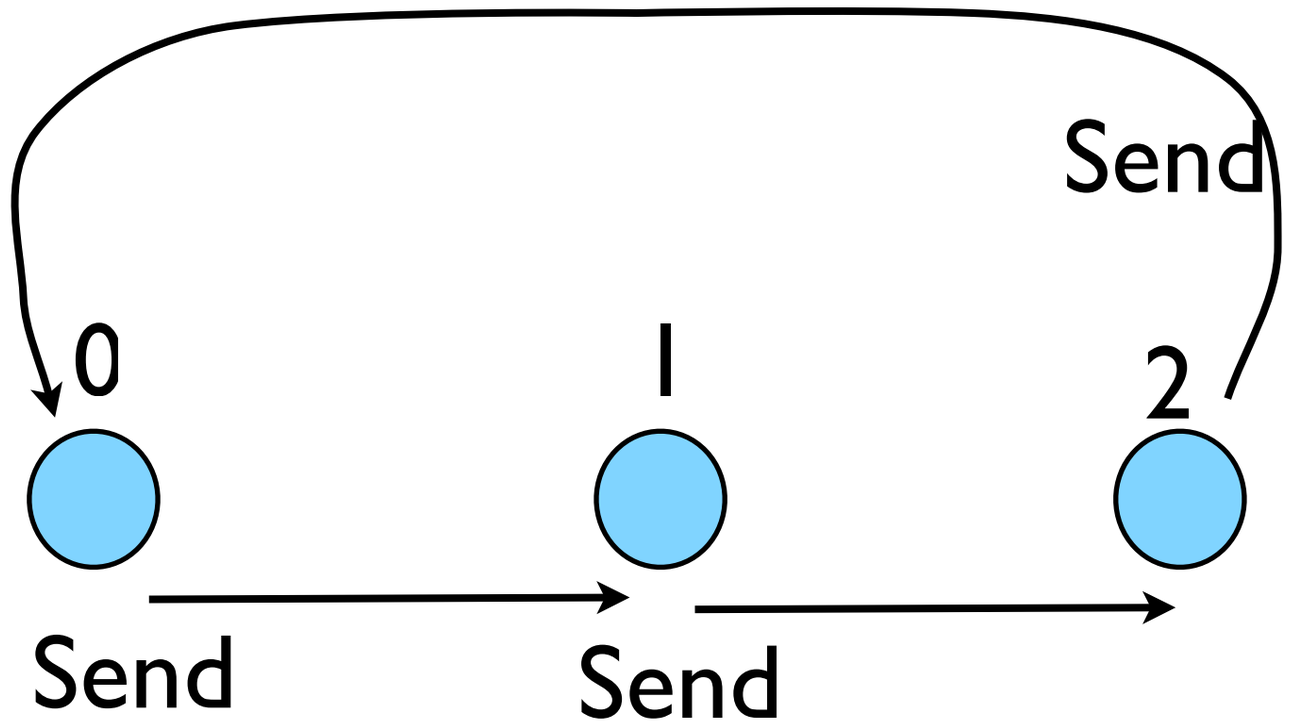



```
if (rightneighbour .lt. size) then
    call MPI_SSEND(greeting, 15, MPI_CHARACTER, rightneighbour,
&                  tag, MPI_COMM_WORLD, ierr)
endif

print *, rank, ": ...and see if we are greeted!"

if (leftneighbour .ge. 0) then
    call MPI_RECV(received, 15, MPI_CHARACTER, leftneighbour,
&                  tag, MPI_COMM_WORLD, status, ierr)
    print *, rank, ": leftneighbor says <",received,">!"
endif

call MPI_FINALIZE(ierr)

return
end
```

Deadlock

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





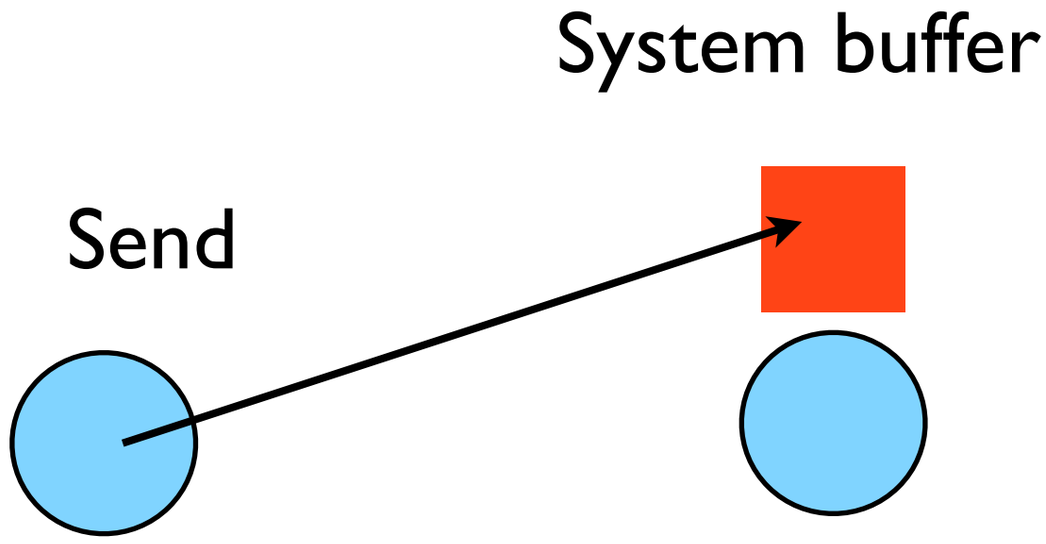
Big MPI
Lesson #1

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

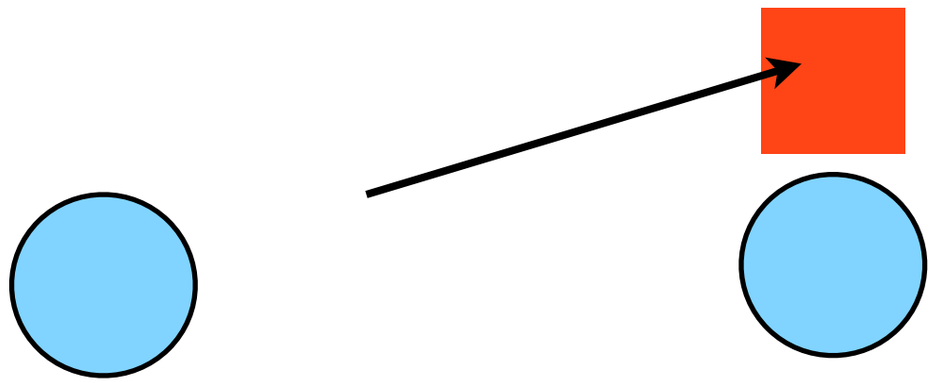
Different versions of SEND

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

Buffering



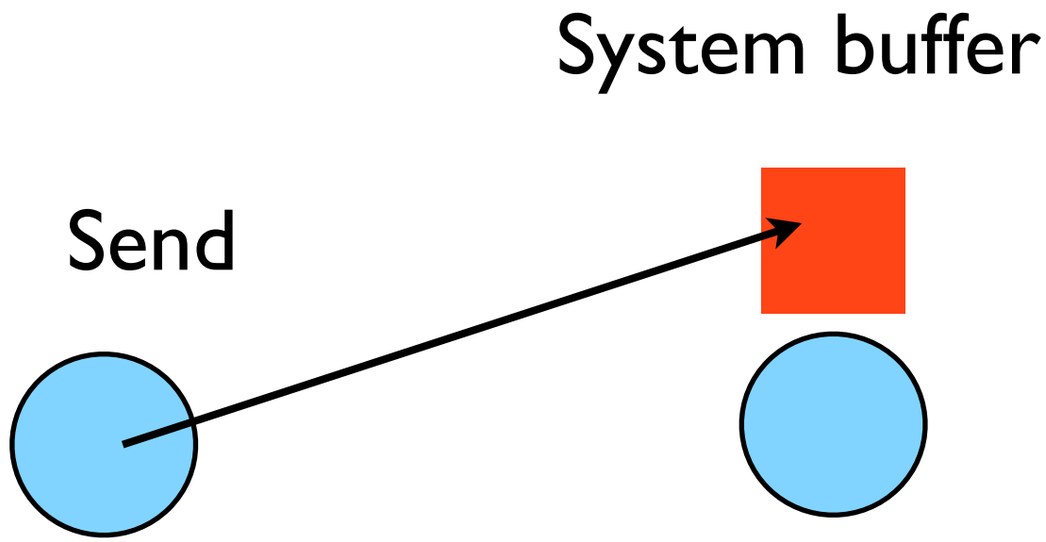
(Non) Blocking



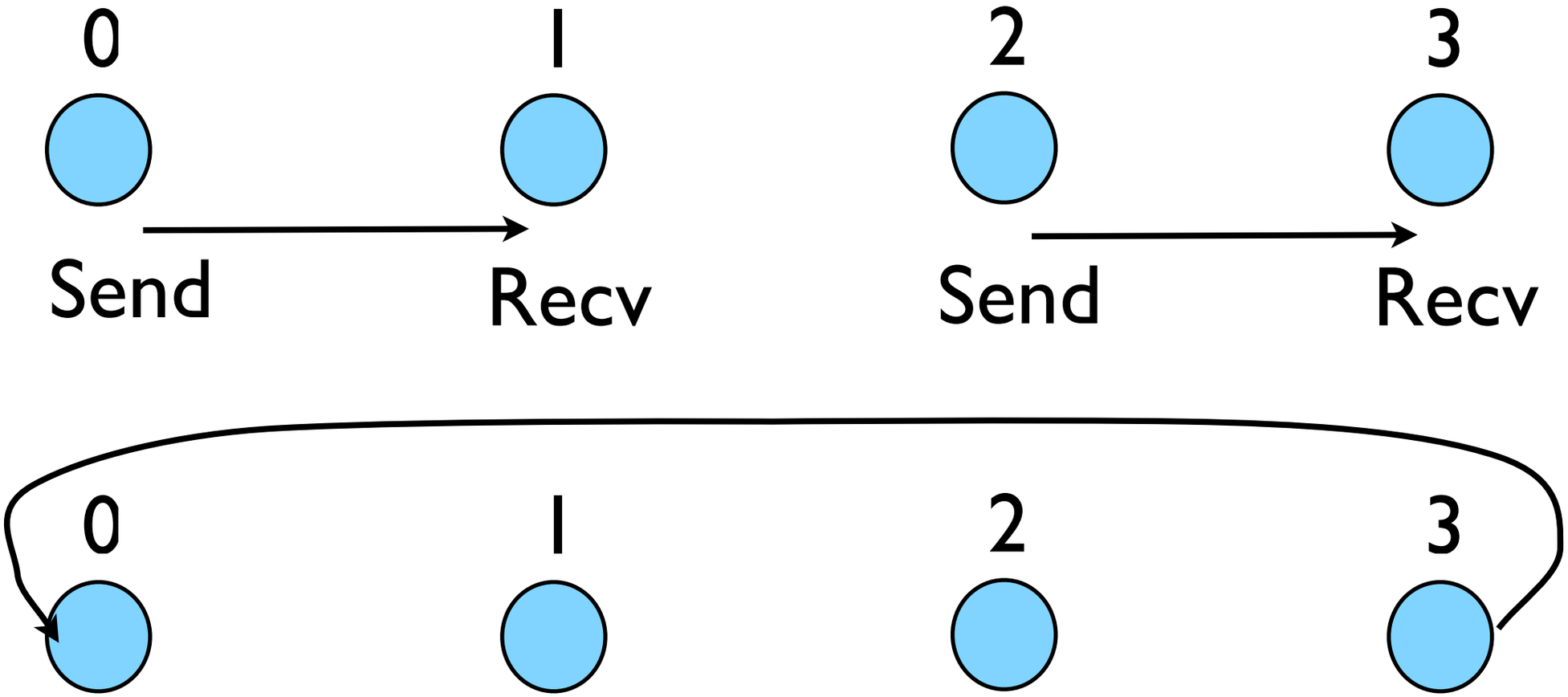
Buffering is dangerous!

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

Buffering



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



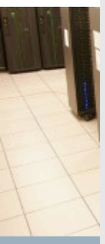
- First: evens send, odds receive
- Then: odds send, evens receive
- Will this work with an odd # of processes?
- How about 2? 1?

```
      program hellompiworld

      implicit none
      include "mpif.h"

      integer rank, size
      integer ierr
      integer leftneighbour
      integer rightneighbour
      integer tag
      integer status(MPI_STATUS_SIZE)
      character(15) greeting
      character(15) received

      call MPI_INIT(ierr)

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

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

      if (leftneighbour.ge.0)
     &            print *,rank,': My left neighbour is ', leftneighbour
      if (rightneighbour.lt.size)
     &            print *,rank,': My right neighbour is ',rightneighbour

      write(greeting,'(a i4)'), 'Hello from', rank
      tag = 1

      if (mod(rank,2) .eq. 0) then
            call MPI_SSEND(greeting, 15, MPI_CHARACTER,
     &                rightneighbour, tag, MPI_COMM_WORLD, ierr)
            call MPI_RECV(received, 15, MPI_CHARACTER,
     &                leftneighbour, tag, MPI_COMM_WORLD, status,
     &                ierr)
            print *, rank, ": leftneighbour says <",received,">!"
      else
            call MPI_RECV(received, 15, MPI_CHARACTER,
     &               leftneighbour, tag, MPI_COMM_WORLD, status,
     &               ierr)
            print *, rank, ": leftneighbour says <",received,">!"
            call MPI_SSEND(greeting, 15, MPI_CHARACTER,
     &                rightneighbour, tag, MPI_COMM_WORLD, ierr)
      endif

      call MPI_FINALIZE(ierr)

      return
      end
```

Evens send first

Then odds

hello-world-send-fixed.f



Something new: Sendrecv

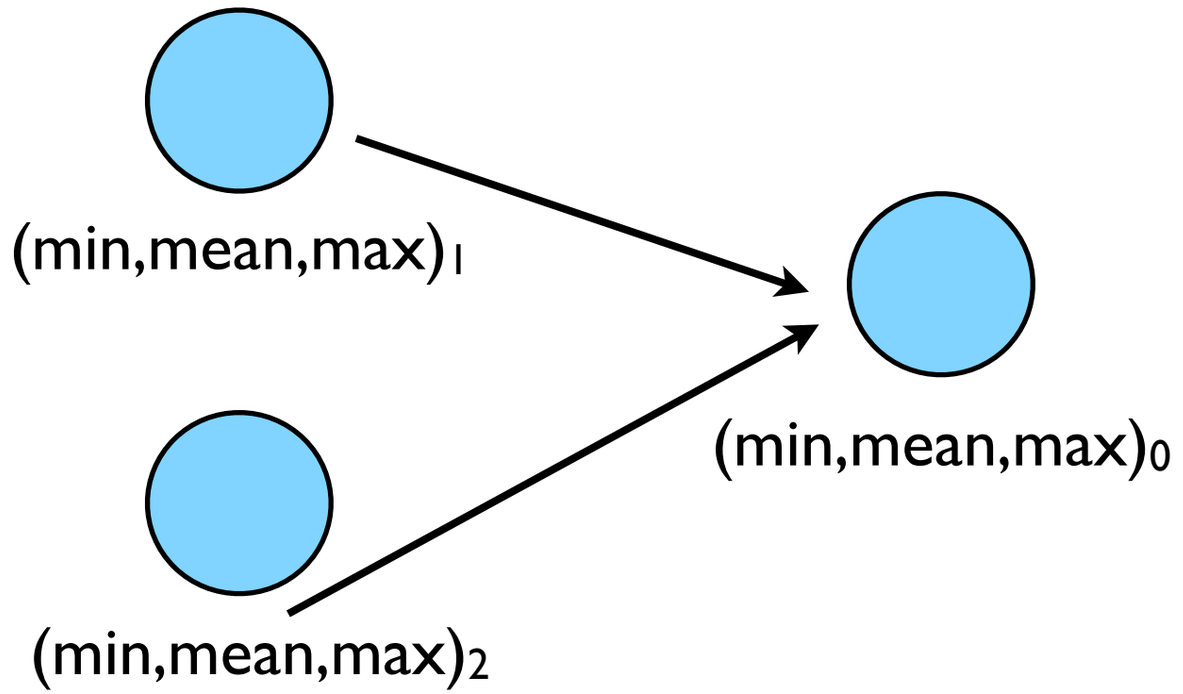
- A blocking send and receive built in together
- Lets them happen simultaneously
- Can automatically pair the sends/recvs!
- dest, source does not have to be same; nor do types or size.

```
      program hellompiworld

      implicit none
      include "mpif.h"

      integer rank, size
      integer ierr
      integer leftneighbour
      integer rightneighbour
      integer tag
      integer status(MPI_STATUS_SIZE)
      character(15) greeting
      character(15) received

      call MPI_INIT(ierr)

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

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

       if (leftneighbour.ge.0)
     &      print *,rank,': My left neighbour is ', leftneighbour
       if (rightneighbour.lt.size)
     &      print *,rank,': My right neighbour is ',rightneighbour

       write(greeting,'(a i4)'), 'Hello from', rank
       tag = 1

       call MPI_SENDRECV(greeting, 15, MPI_CHARACTER, rightneighbour,tag,
     &      received, 15, MPI_CHARACTER,leftneighbour, tag,
     &      MPI_COMM_WORLD, status, ierr)

       print *, rank, ": leftneighbour says <",received,">!"

       call MPI_FINALIZE(ierr)

       return
       end
```

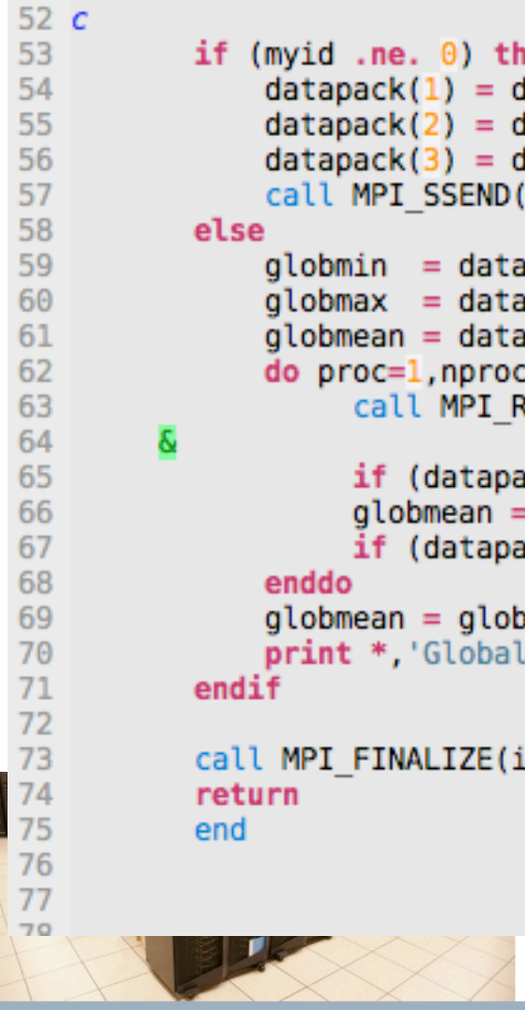
hello-world-sendrecv.f

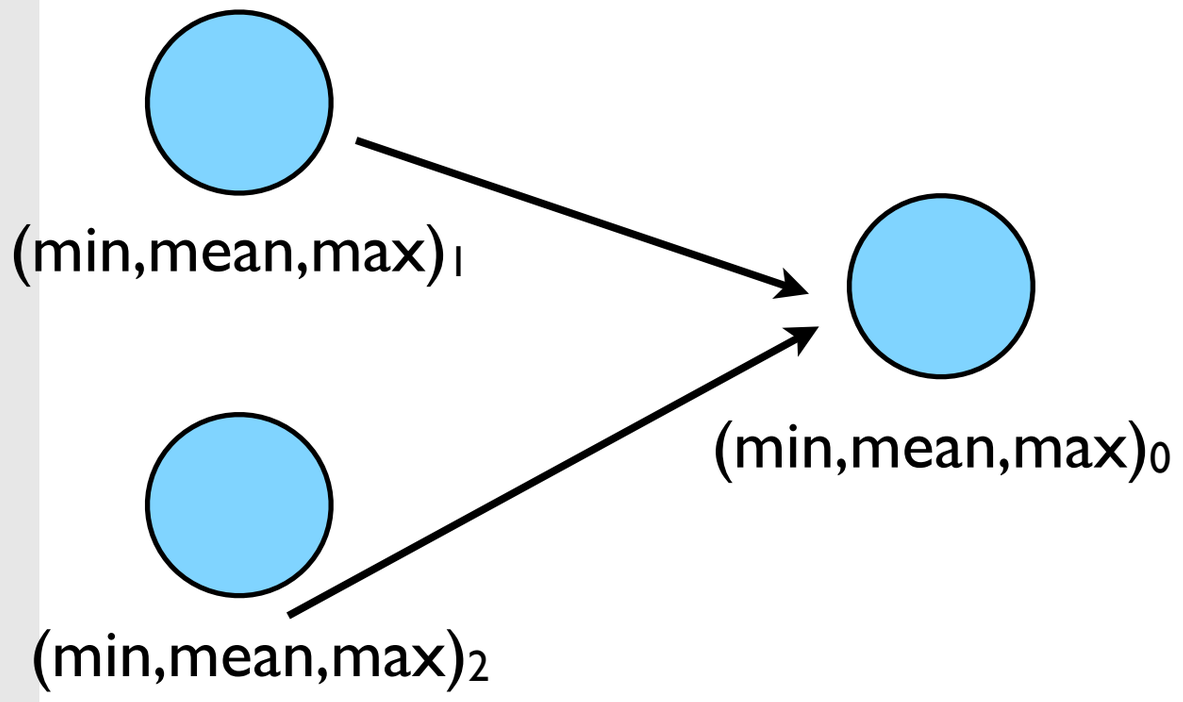
Min, Mean, Max of numbers

- 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?)



```
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-sendrecv.f

```
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

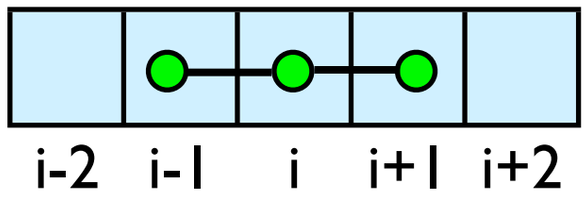
Performs a reduction and sends it to one node or everywhere

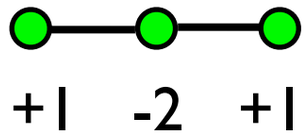
minmeanmax-allreduce.f

Discretizing Derivatives

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

- 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

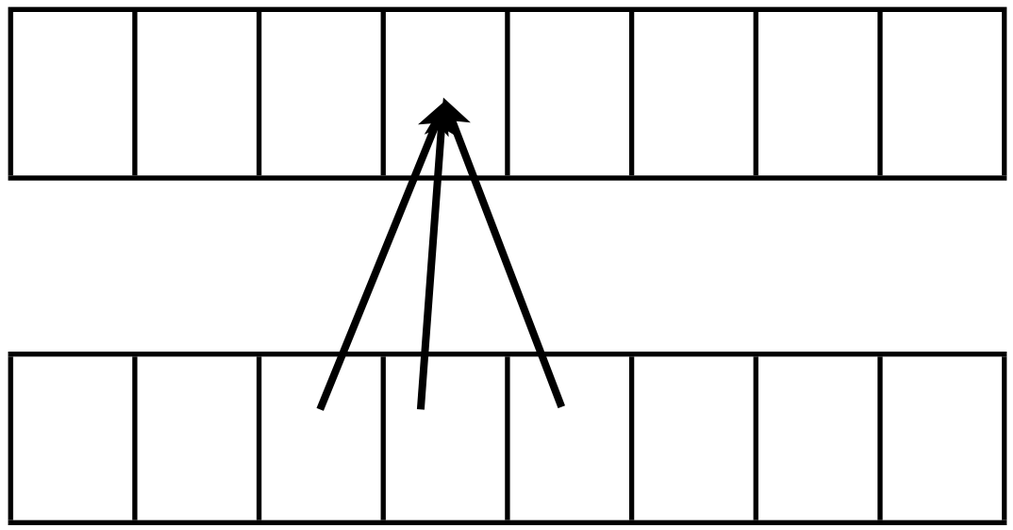
$$\frac{\partial T}{\partial t} = -D\frac{\partial^2 T}{\partial x^2}$$

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

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

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

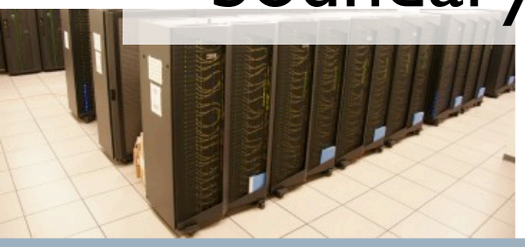
- Simple 1d ODE
- Each timestep, new data for T[i] requires old data for T[i+1],T[i],T[i-1]



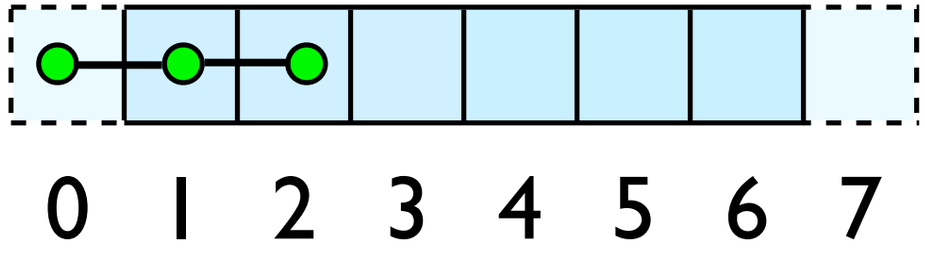
```
cd ~/pca/mpi-intro
make diffusion
./diffusion
```

Guardcells

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



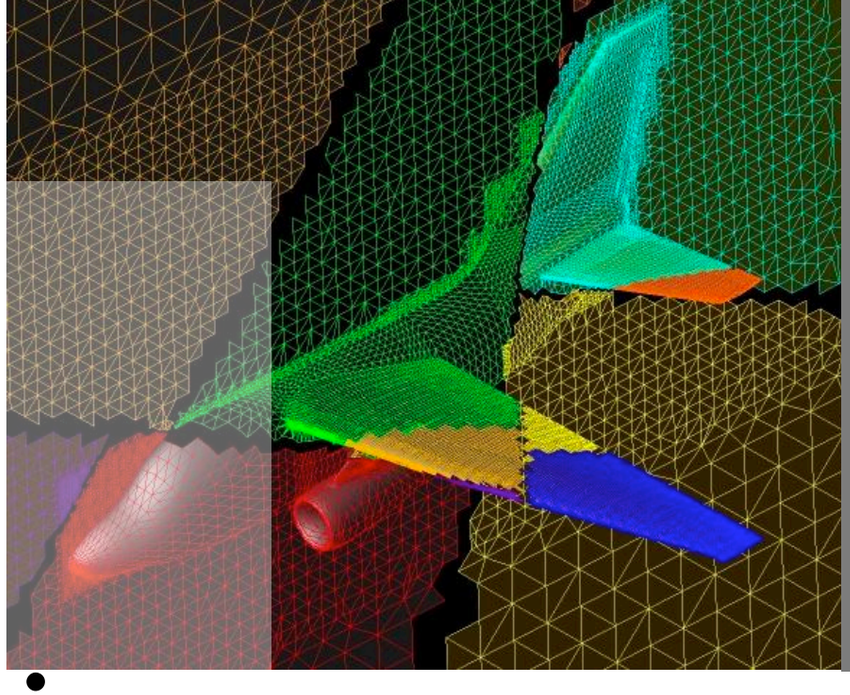
Global Domain



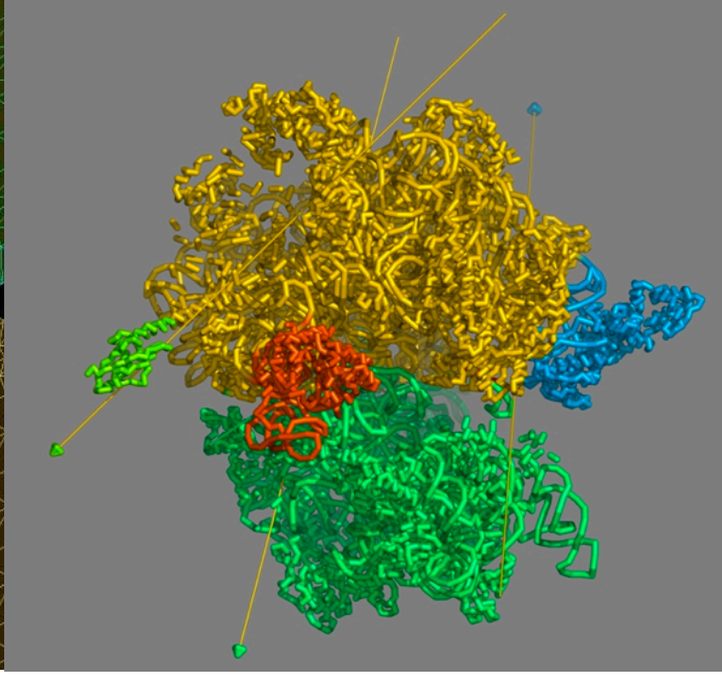
ng = 1

loop from ng, N - 2 ng

Domain Decomposition

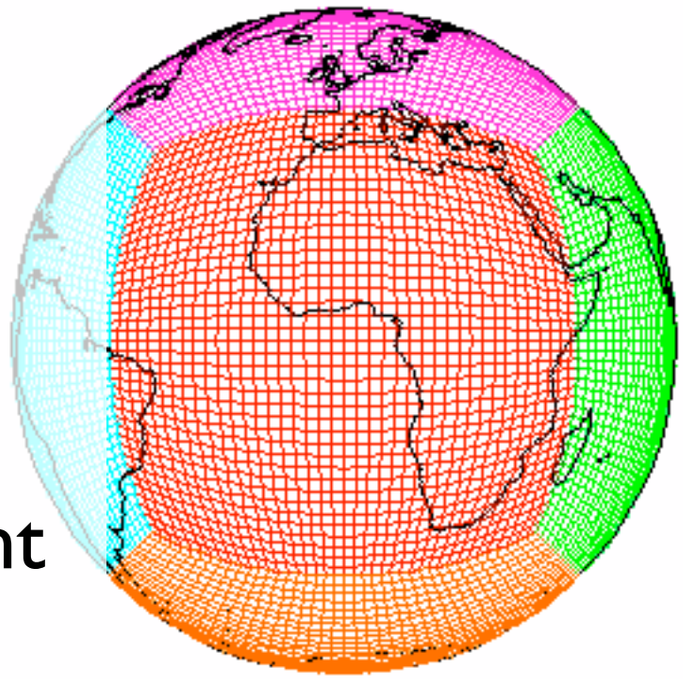


http://adg.stanford.edu/aa241/design/compaero.html

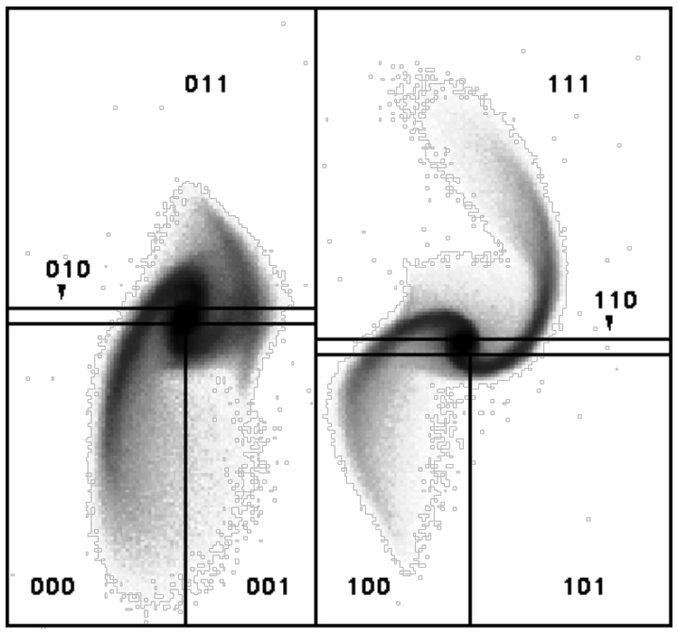


http://www.uea.ac.uk/cmp/research/cmpbio/Protein+Dynamics,+Structure+and+Function

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



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



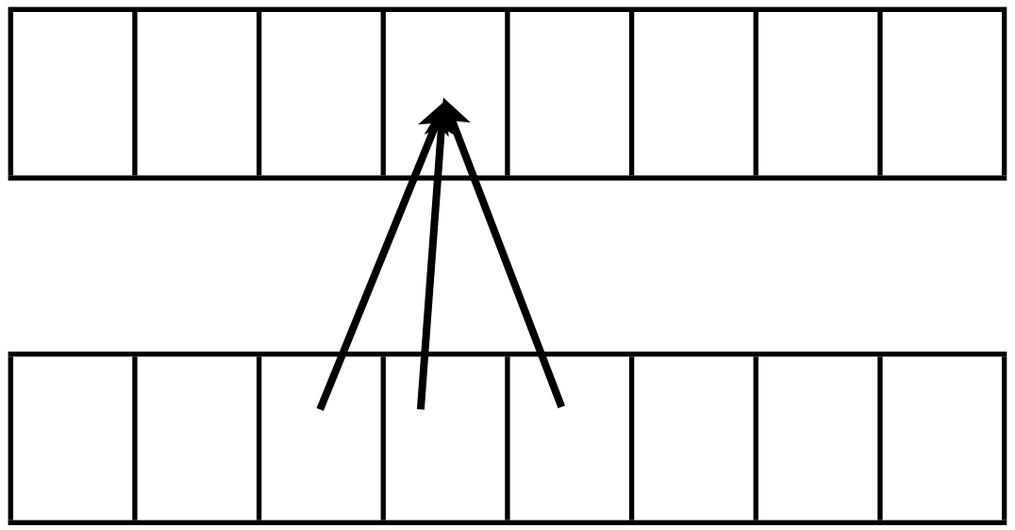
http://www.cita.utoronto.ca/~dubinski/treecode/node8.html

Implement a diffusion equation in MPI

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

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

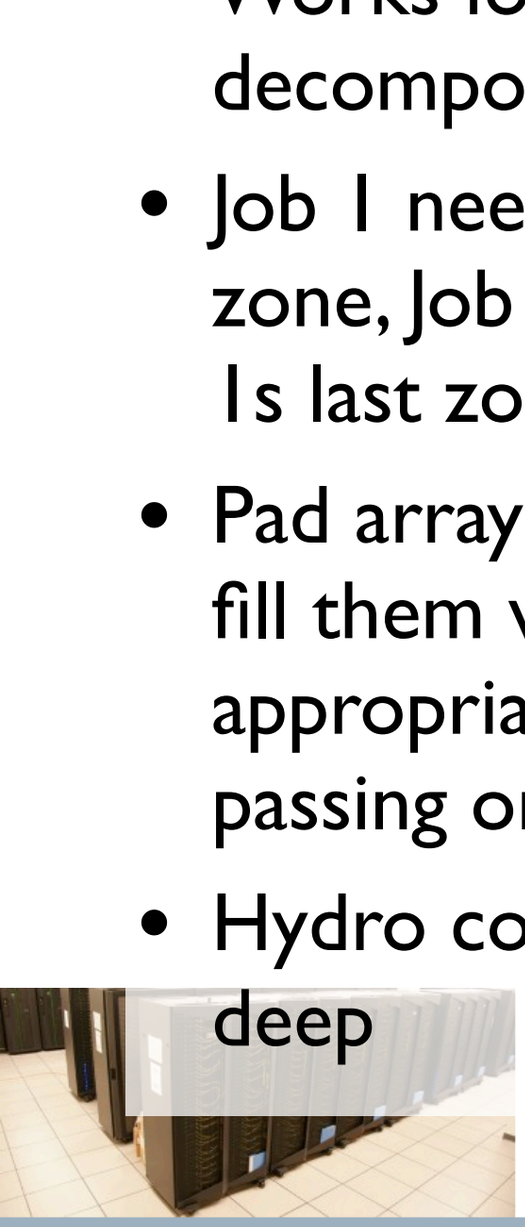
- Need one neighboring number per neighbor per timestep

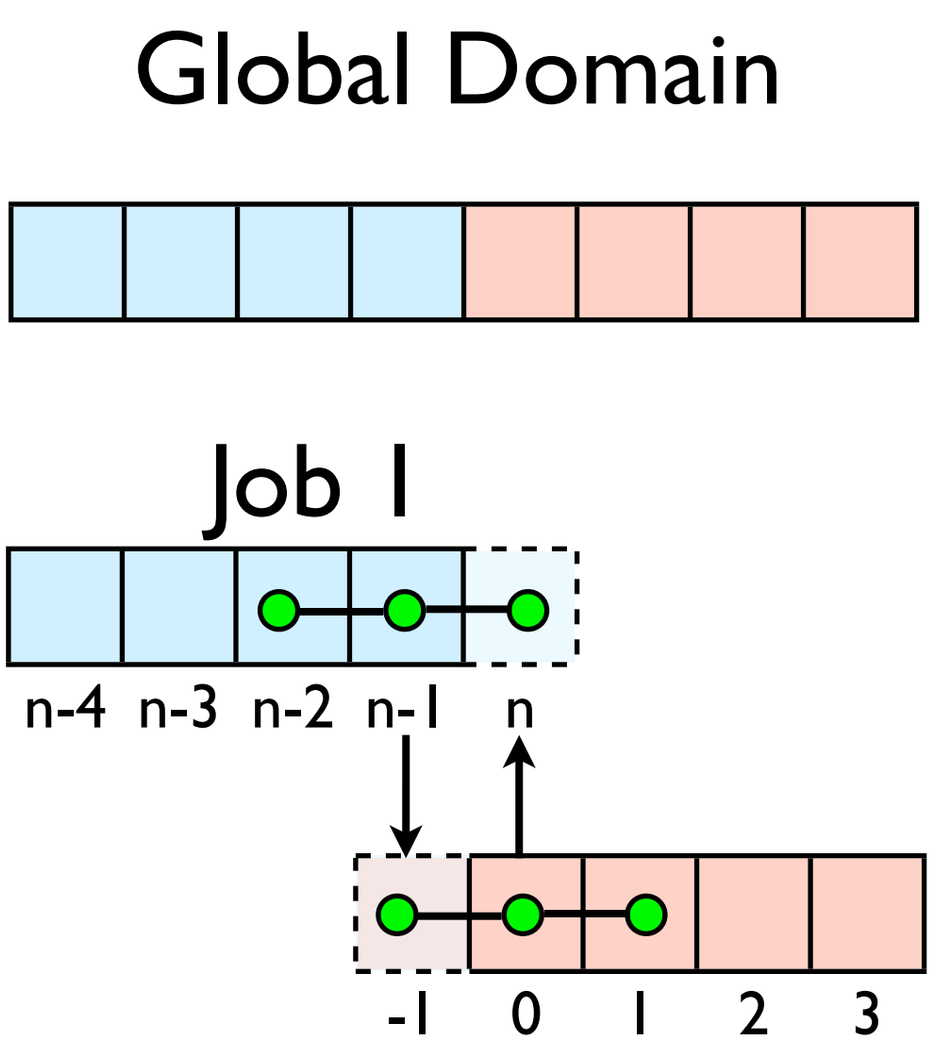


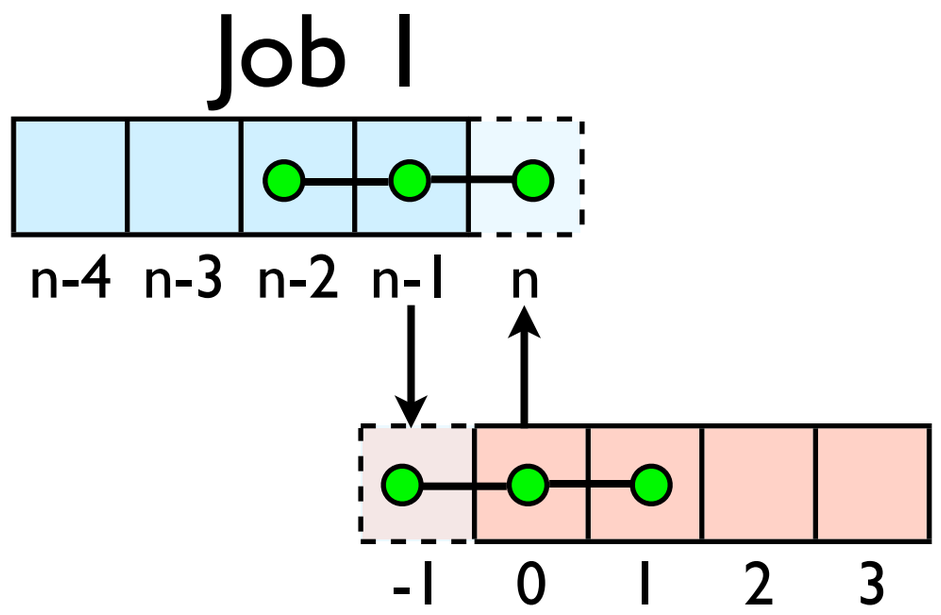
```
cd ~/pca/mpi-intro
make diffusion
./diffusion
```

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
  - Use even/odd trick from before; if even do rightmost
  - For simplicity, use periodic BCs -- task 0 is neighbors with task P-1

Hands-on Assignment I: MPI diffusion

- cp diffusion.f diffusion-mpi.f
- Make an MPI-ed version of diffusion equation
- (Sample executable in completedexecutables)
- Test on 1..8 procs
- Copy file to `~/hw-intrompi/`

- 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
- localxleft and localxright
- At end of step, exchange guardcells; use sendrecv

Hands-on Assignment II: MinMeanMax

- Run minmeanmax-sendrecv on 1,2,4 procs
- Test diffusion timing on 1,2,4,8 procs
- Put timing info in hw directory

```
$ cd mpi-intro
$ make clean
$ make minmeanmax-sendrecv
$ time -p mpirun -np 1 ./
minmeanmax-sendrecv
$ make diffusion-mpi
$ time -p mpirun -np 1 ./diffusion-
mpi
```

Hands-on Assignment III: Use the Queue

- Copy files to assegai
- Log in
- Run minmeanmax-sendrecv on 1,2,4,8 nodes
- Test diffusion timing on 1,2,4,8 nodes
- Put results in hw directory

From gpc01..04

```
$ cp ~/pca/scripts/sample.pbs .
$ make clean
$ make diffusion-mpi USEPGPLOT="" PGPLIBS=""

$ qstat
$ qsub sample.pbs
$ qstat
```

to change number of nodes, change
`#PBS -l nodes=2:ppn=8...`
to 1,2,4,8
and change
-np 16
as appropriate (8,16,32,64)

sample.pbs:

```
#!/bin/bash
# MOAB/Torque submission script for SciNet GPC (ethernet)
#
#PBS -l nodes=2:ppn=8,walltime=1:00:00
#PBS -N test

# DIRECTORY TO RUN - $PBS_O_WORDIR is directory job was submitted from
cd $PBS_O_WORKDIR

# EXECUTION COMMAND; -np nodes*ppn
mpirun -np 16 -hostfile $PBS_NODEFILE ./diffusion-mpi
```

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,...
```