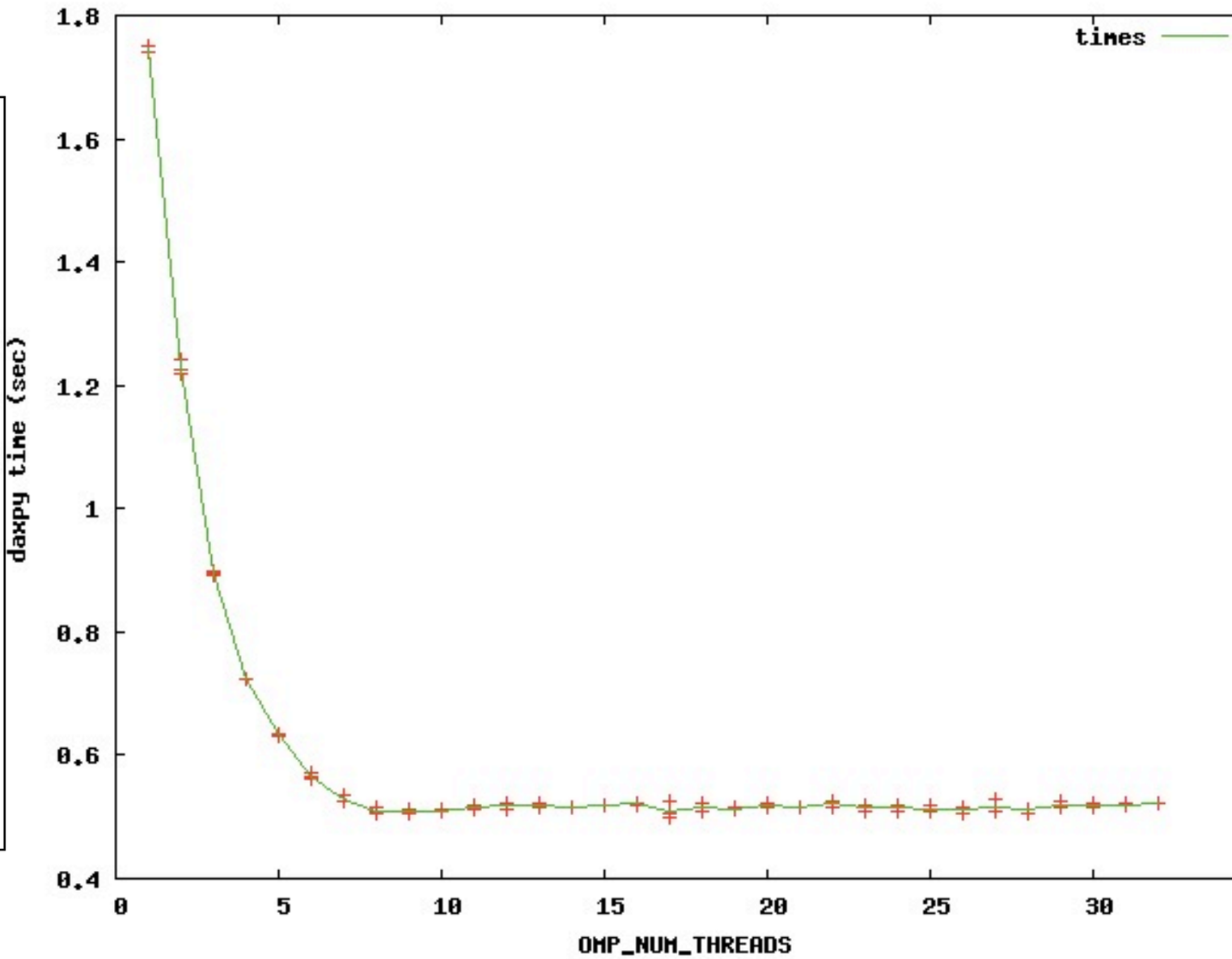
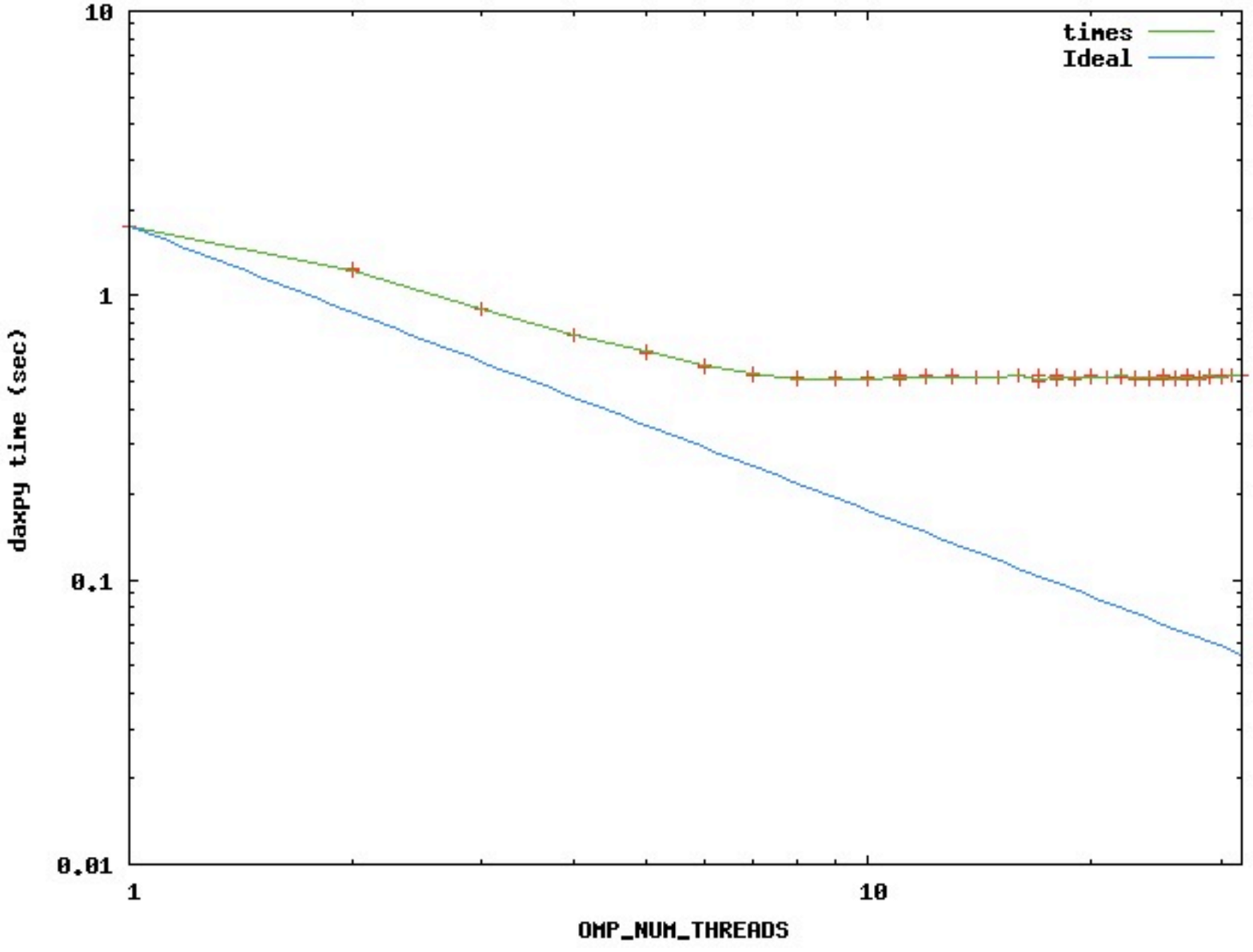
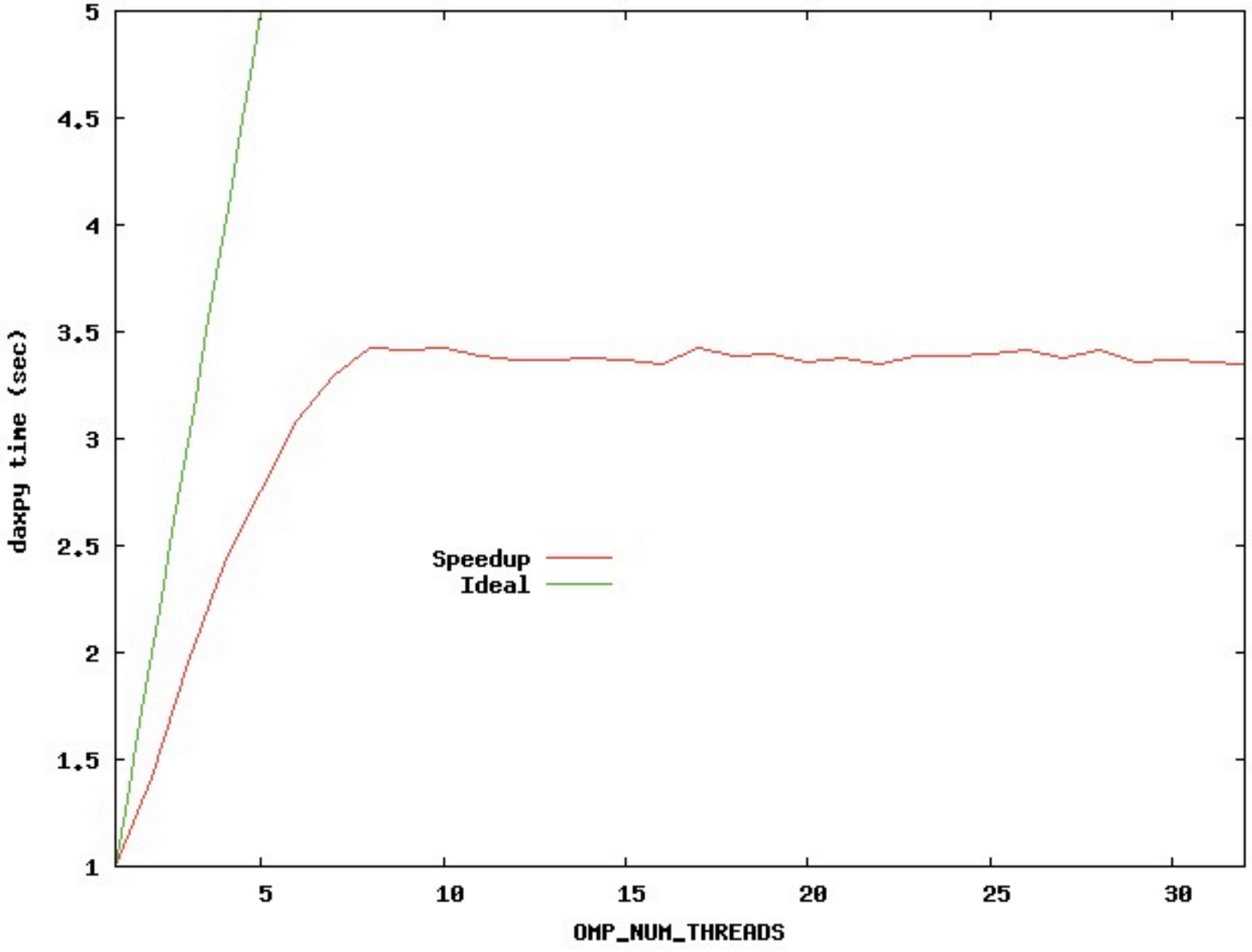
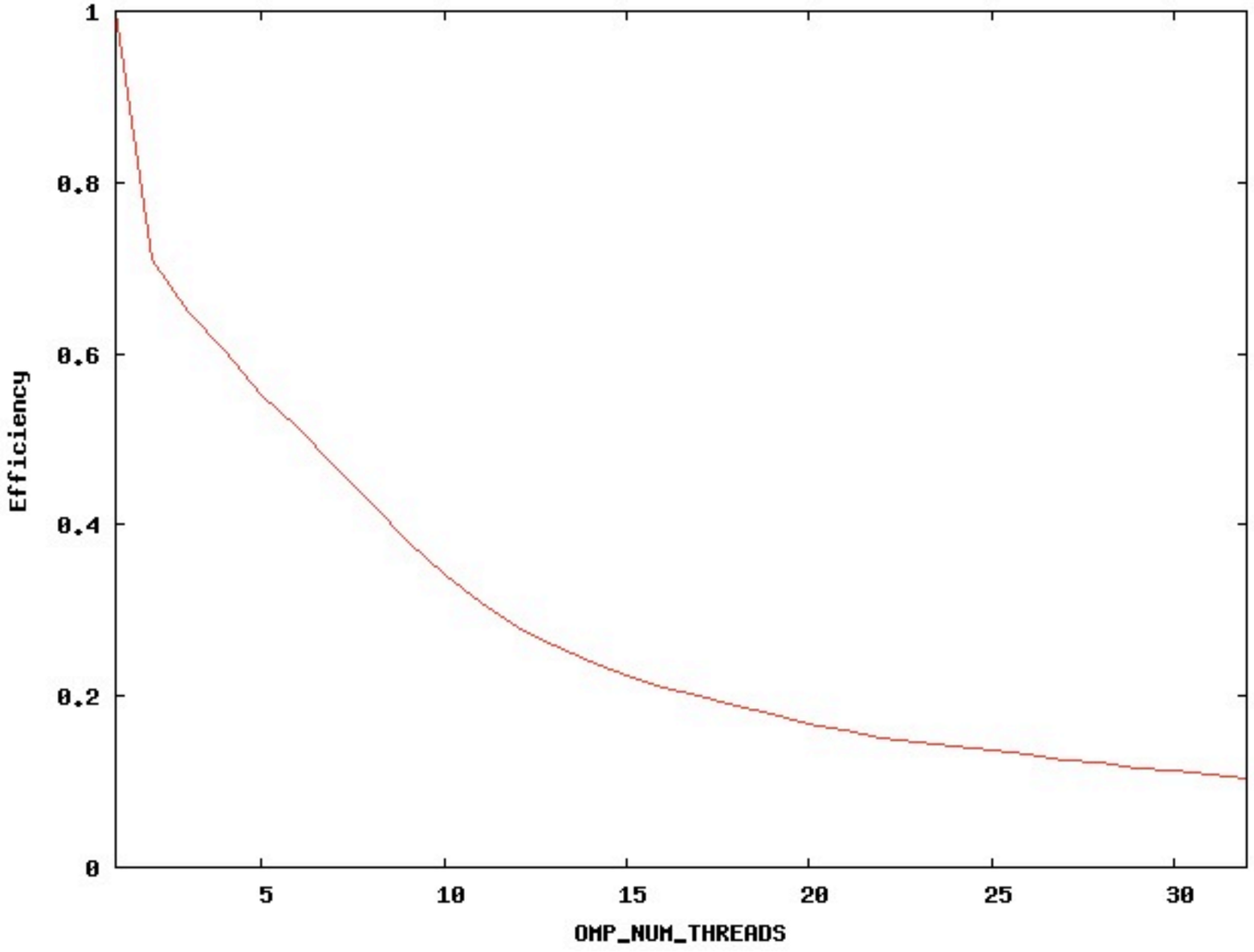
```
void daxpy(long n,double a,double *x,
                  double *y,double*z) {

   #pragma omp parallel \
               default(none) \
               shared(x,y,z,n,a)
   {
    #pragma omp for
    for (long i=0;i<n;i++) {
      x[i] = (double)i*(double)i;
      y[i] = (double)(i+1.)*(i-1.);
    }

    #pragma omp for
    for (long i=0;i<n;i++)
      z[i] += a*x[i] + y[i];
   }
}
```



SciNet compute • calcul CANADA


10
1
0.1
0.01
daxpy time (sec)
1
10
OMP_NUM_THREADS
times
Ideal


daxpy time (sec)
5
4.5
4
3.5
3
2.5
2
1.5
1
5
10
15
20
25
30
OMP_NUM_THREADS
Speedup
Ideal



SciNet
compute • calcul
CANADA

Distributed Memory Computing with MPI

Scientific Computing III
High Performance Scientific Computing
Feb 2012

SciNet
compute • calcul
CANADA

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;
}
```

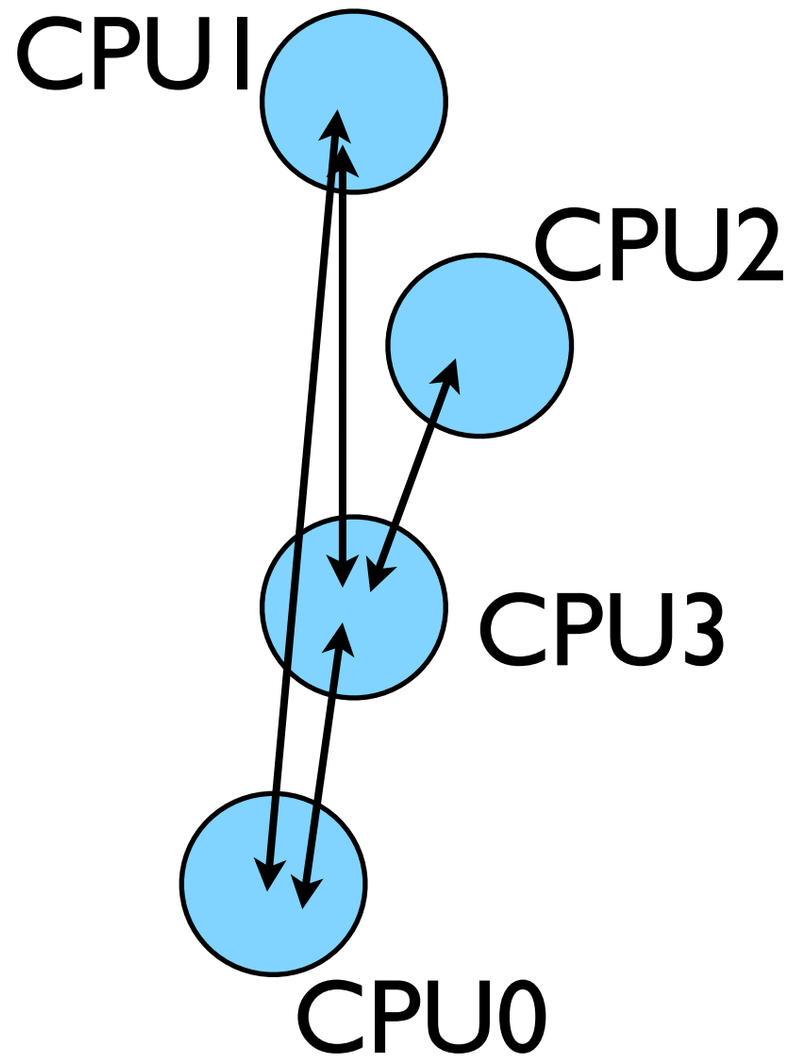
Fortran

```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.

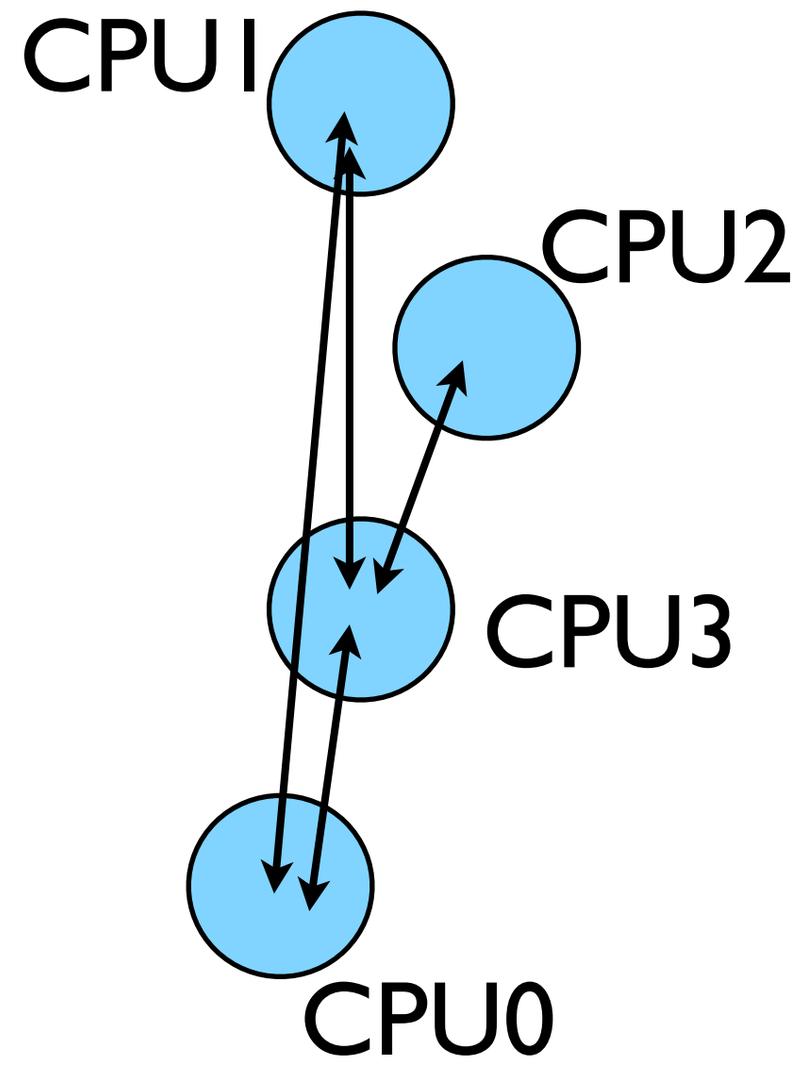
- Each message involves a function call from each of the programs.



SciNet
compute • calcul
CANADA

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

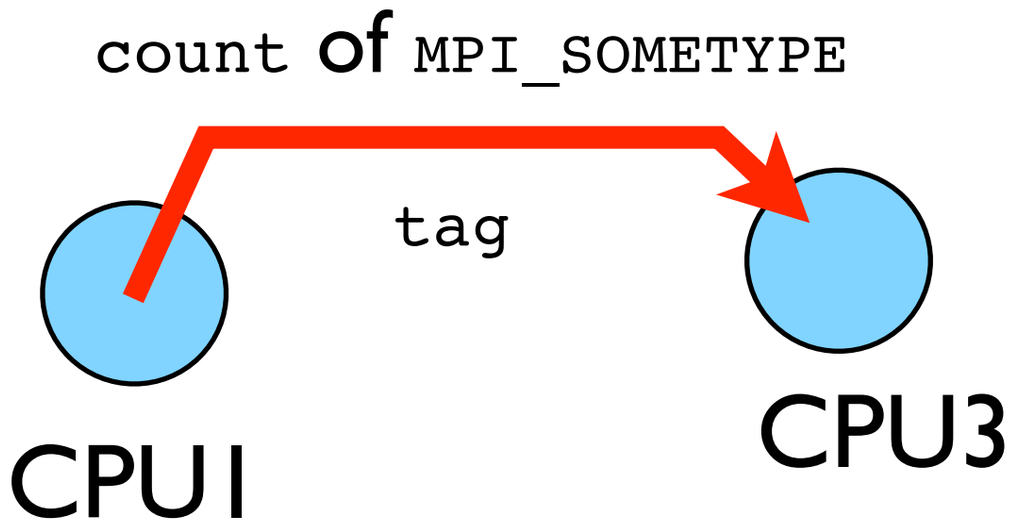
- Three basic sets of functionality:
- Pairwise communications via messages
- Collective operations via messages
- Efficient routines for getting data from memory into messages and vice versa



SciNet compute • calcul CANADA

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



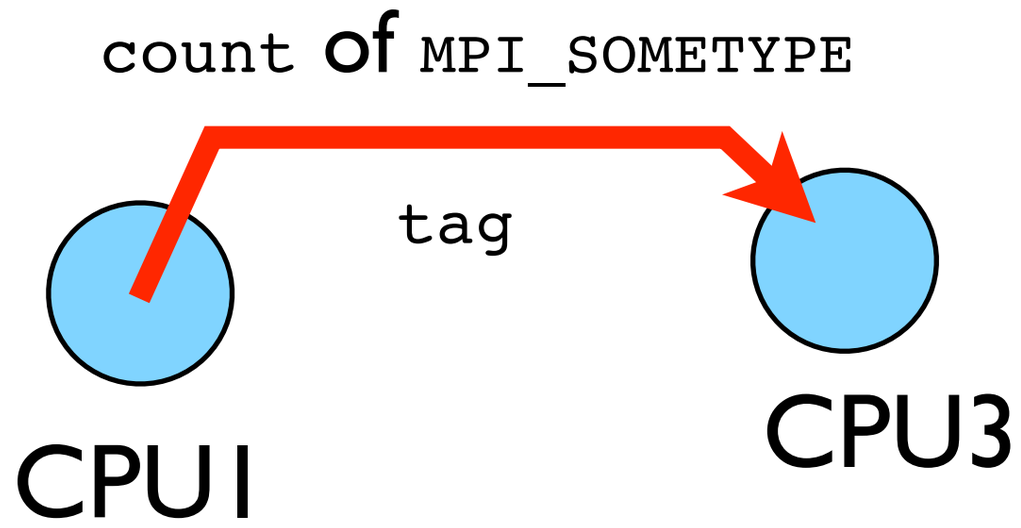
SciNet compute • calcul CANADA

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.



SciNet compute • calcul CANADA

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()
```

SciNet compute • calcul CANADA

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()
```

SciNet compute • calcul CANADA

```
$ ssh -Y login.scinet.utoronto.ca
$ ssh -Y gpc0x
```

```
$ git clone /scinet/course/sc3/hw2
$ cd hw2
$ source ./setup
```

SciNet
compute • calcul
CANADA

Hello World

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

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

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

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

    MPI_Finalize();
    return 0;
}
```

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

edit hello-world.c

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

SciNet compute • calcul CANADA

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

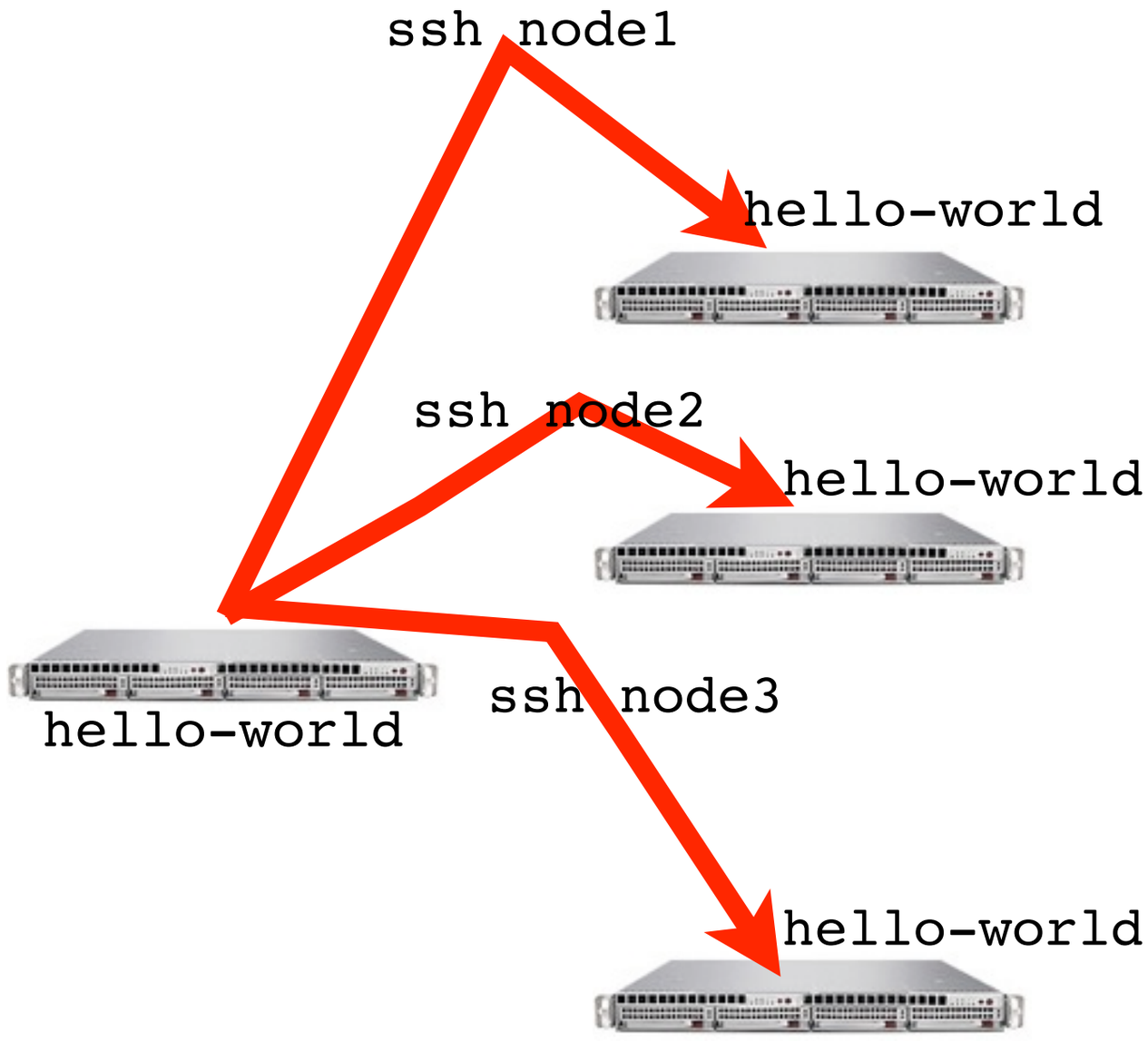
```
$ mpicc --showme hello-world.c
-o hello-world

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
```

SciNet compute • calcul CANADA

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



SciNet compute • calcul CANADA

Number of Processes?

- Number of processes to use is almost always equal to the number of processors

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

- But not necessarily.

- On your nodes, what happens when you run this?

SciNet compute • calcul CANADA

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
```

SciNet compute • calcul CANADA

What the code does

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

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

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

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

    MPI_Finalize();
    return 0;
}
```

SciNet compute • calcul CANADA

`#include <mpi.h>` : imports declarations for MPI function calls

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

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

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

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

    MPI_Finalize();
    return 0;
}
```

`MPI_Init()`: initialization for MPI library.
Must come first.

`MPI_Finalize()`: close up MPI stuff.
Must come last.

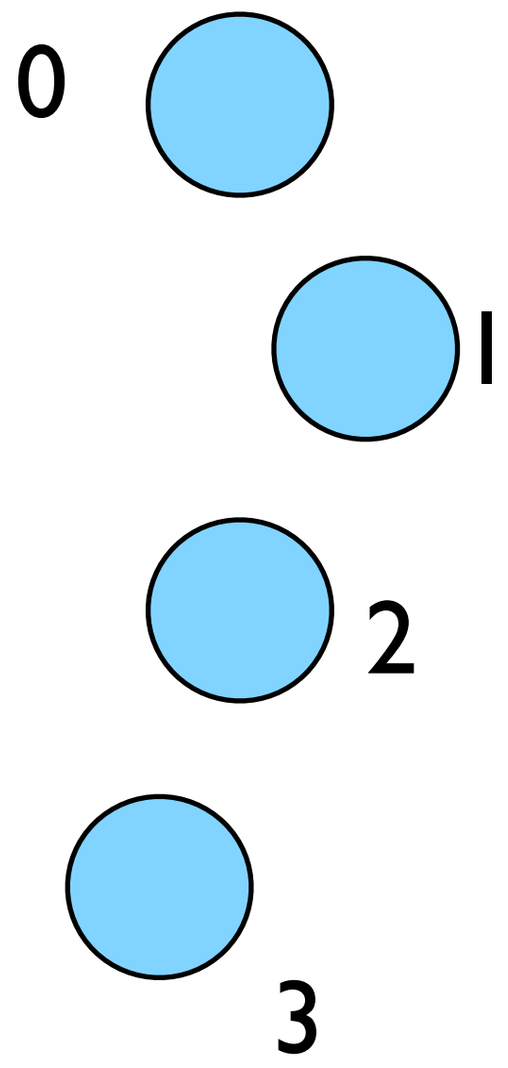
SciNet compute • calcul CANADA

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

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

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

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

    MPI_Finalize();
    return 0;
}
```

`MPI_Comm_rank`, `size`:
require a little more exposition.

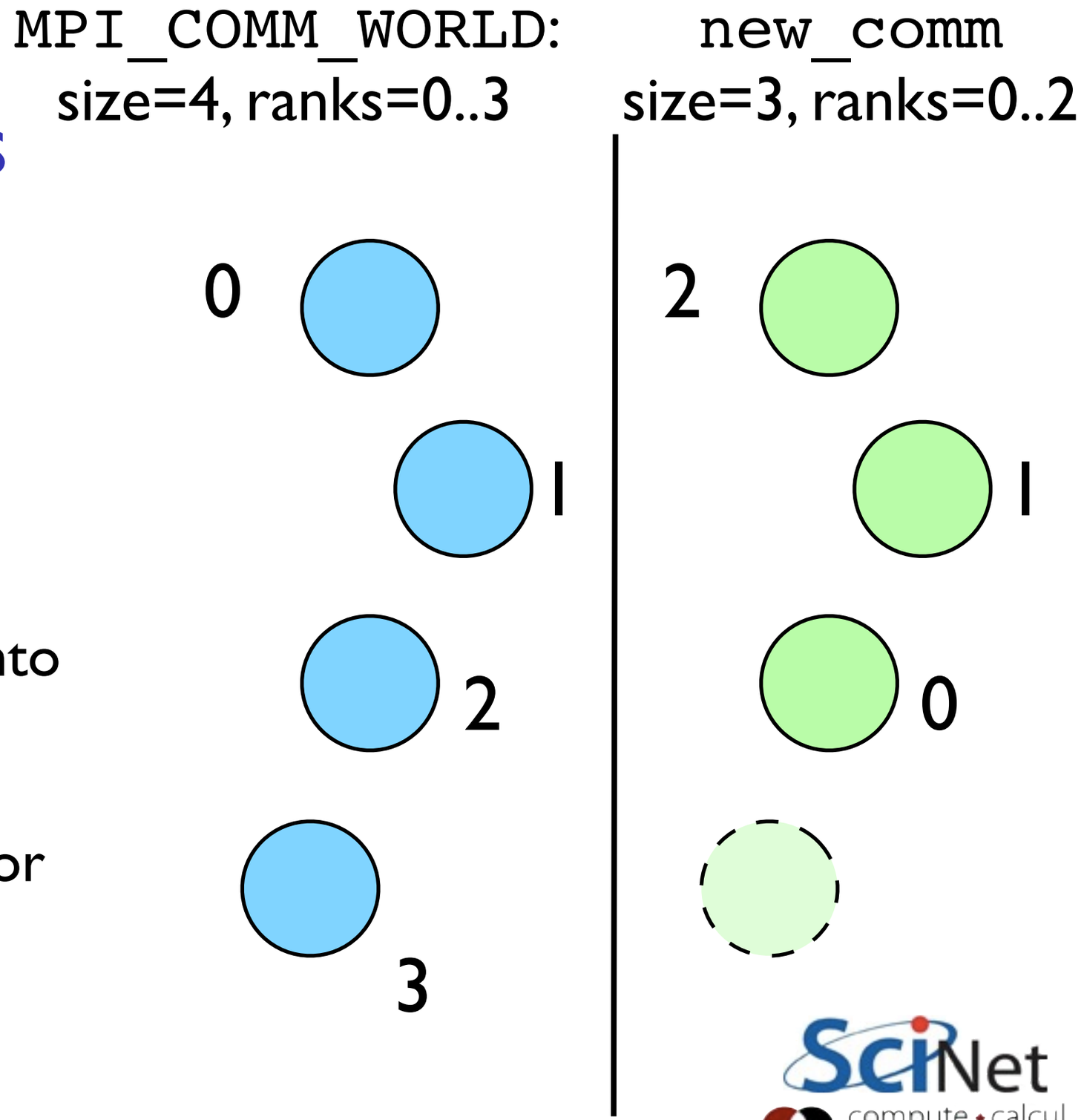
Communicators

- 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`



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



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

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

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

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

    MPI_Finalize();
    return 0;
}
```

`MPI_COMM_RANK`, `MPI_COMM_SIZE`:

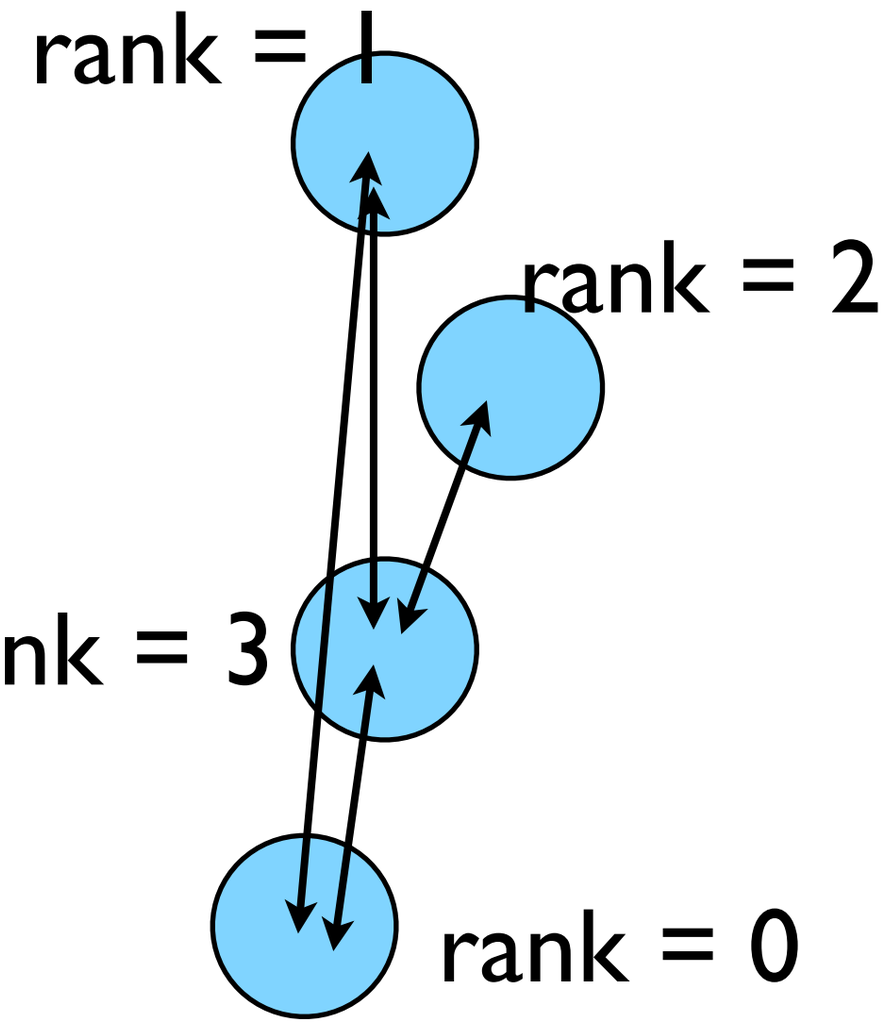
get the size of communicator, the current tasks's rank within communicator.

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.



SciNet compute • calcul CANADA

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

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

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

int main(int argc, char **argv) {
    int rank, size, ierr;
    int sendto, recvfrom;  /* task to send, recv from */
    int ourtag=1;          /* shared tag to label msgs*/
    char sendmessage[]="Hello";       /* text to send */
    char getmessage[6];             /* text to recieve */
    MPI_Status rstatus;       /* MPI_Recv status info */

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

    if (rank == 0) {
        sendto = 1;
        ierr = MPI_Ssend(sendmessage, 6, MPI_CHAR, sendto,
                         ourtag, MPI_COMM_WORLD);
        printf("%d: Sent message <%s>\n", rank, sendmessage);
    } else if (rank == 1) {
        recvfrom = 0;
        ierr = MPI_Recv(getmessage, 6, MPI_CHAR, recvfrom,
                        ourtag, MPI_COMM_WORLD, &rstatus);
        printf("%d: Got message <%s>\n", rank, getmessage);
    }
    ierr = MPI_Finalize();
    return 0;
}
```

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

- mpirun -np 2 ./firstmessage.py

```python
#!/usr/bin/env python

from mpi4py import MPI

def main():
    comm = MPI.COMM_WORLD
    nprocs = comm.Get_size()
    rank   = comm.Get_rank()

    if rank == 0:
        msg = "Hello"
        comm.ssend(msg, dest=1, tag=10)
        print rank, 'Sent message: ', msg
    elif rank == 1:
        rcvmsg = comm.recv(source=0, tag=10)
        print rank, 'Got message: ', rcvmsg


if __name__ == "__main__":
    main()
```

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);
```

Special Source/Dest: MPI_PROC_NULL

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

Special Source: MPI_ANY_SOURCE

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

More complicated example:

- Let's look at secondmessage.c

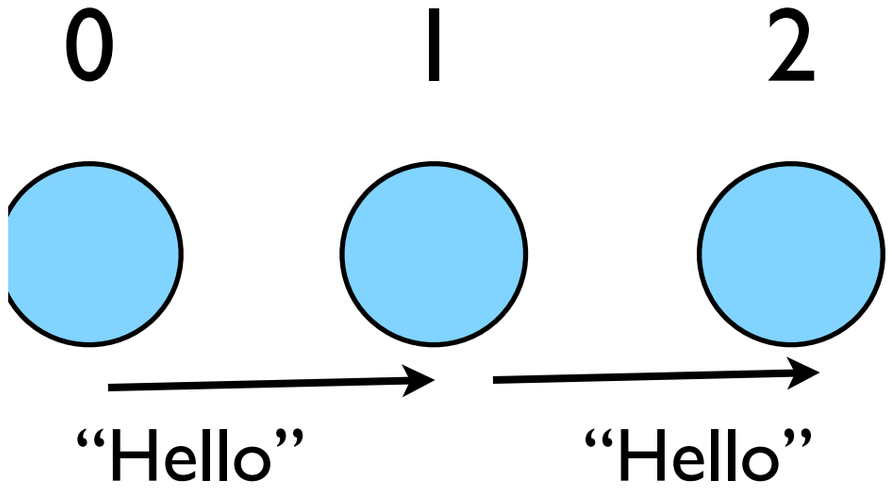
```c
#include <stdio.h>
#include <mpi.h>

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

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

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

    msgsent = rank*rank;
    msgrcvd = -999;

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

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

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

Compile and run

- mpicc -o secondmessage secondmessage.c
- mpirun -np 4 ./secondmessage

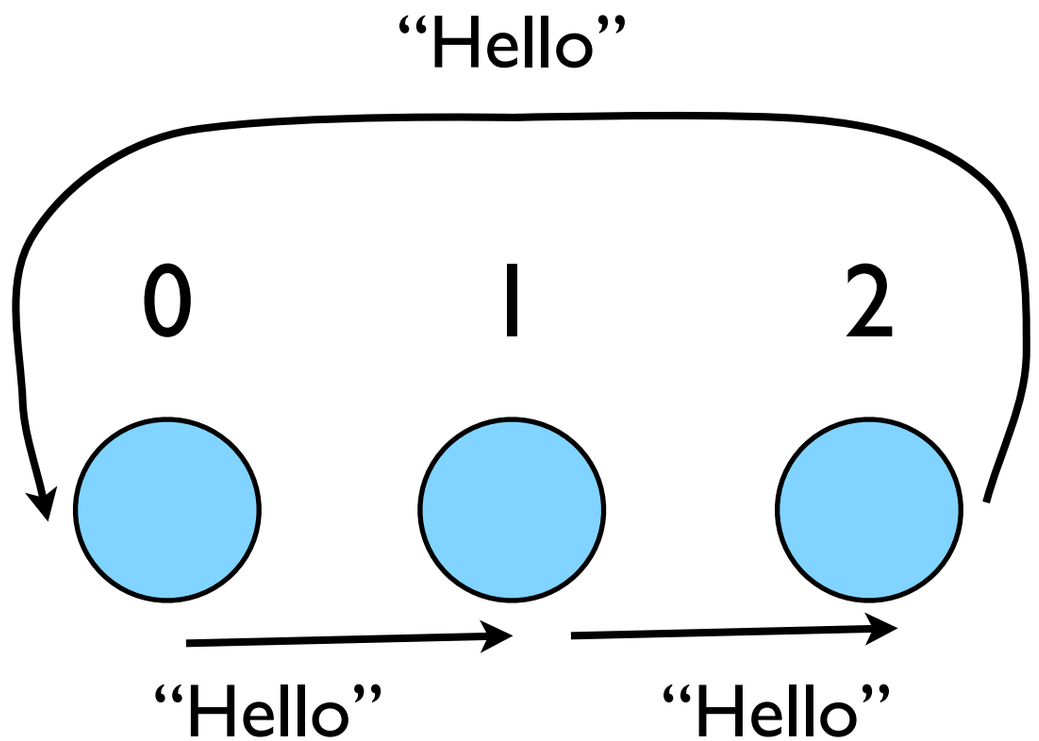
```
ljdursi|segfault.local> mpirun -np 4 ./secondmessage
            3 : got message <Hello>.
            2 : sent message <Hello>.
            2 : got message <Hello>.
            1 : sent message <Hello>.
            0 : sent message <Hello>.
            1 : got message <Hello>.
```

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

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

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

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

    msgsent = rank*rank;
    msgrcvd = -999;

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

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

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

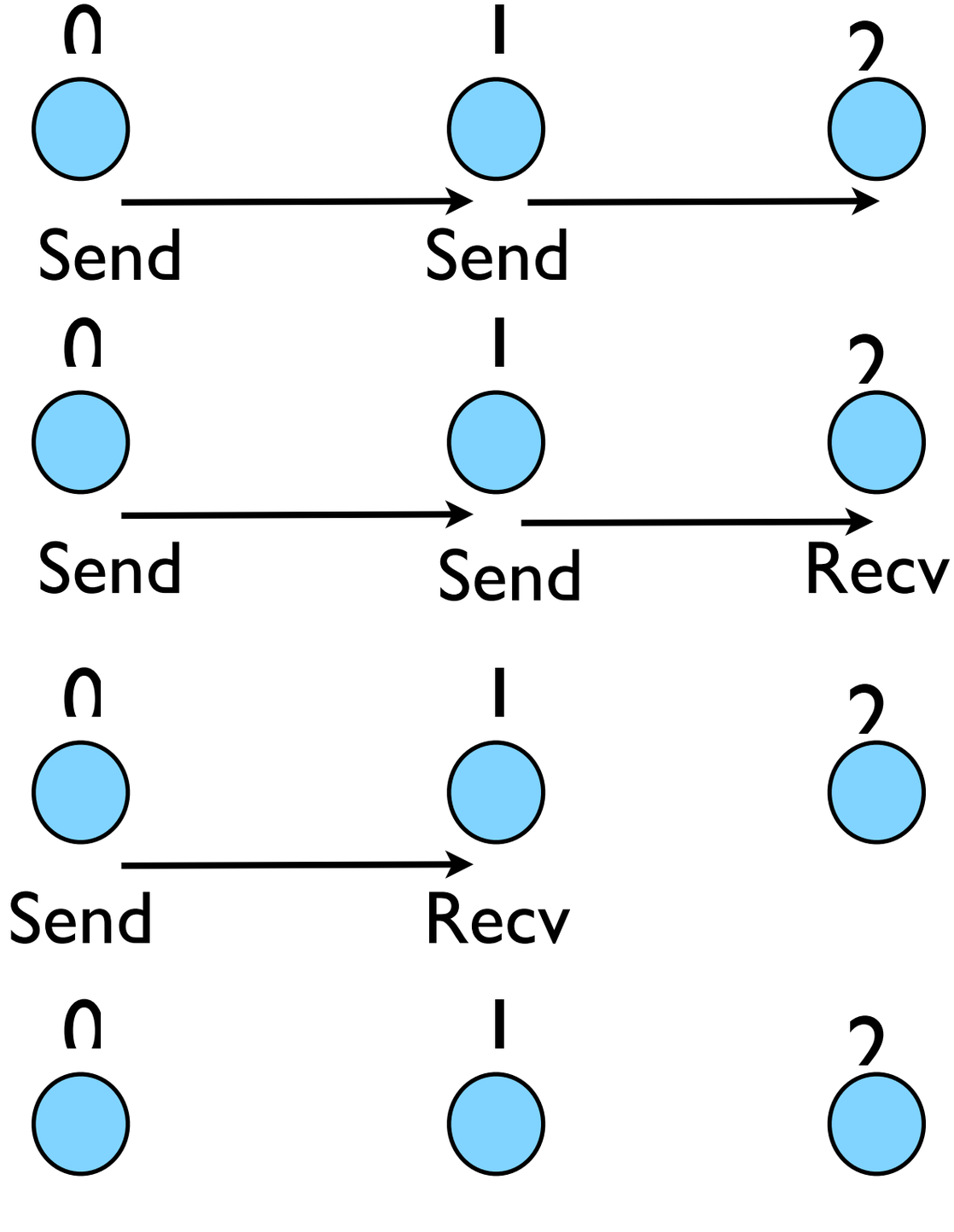


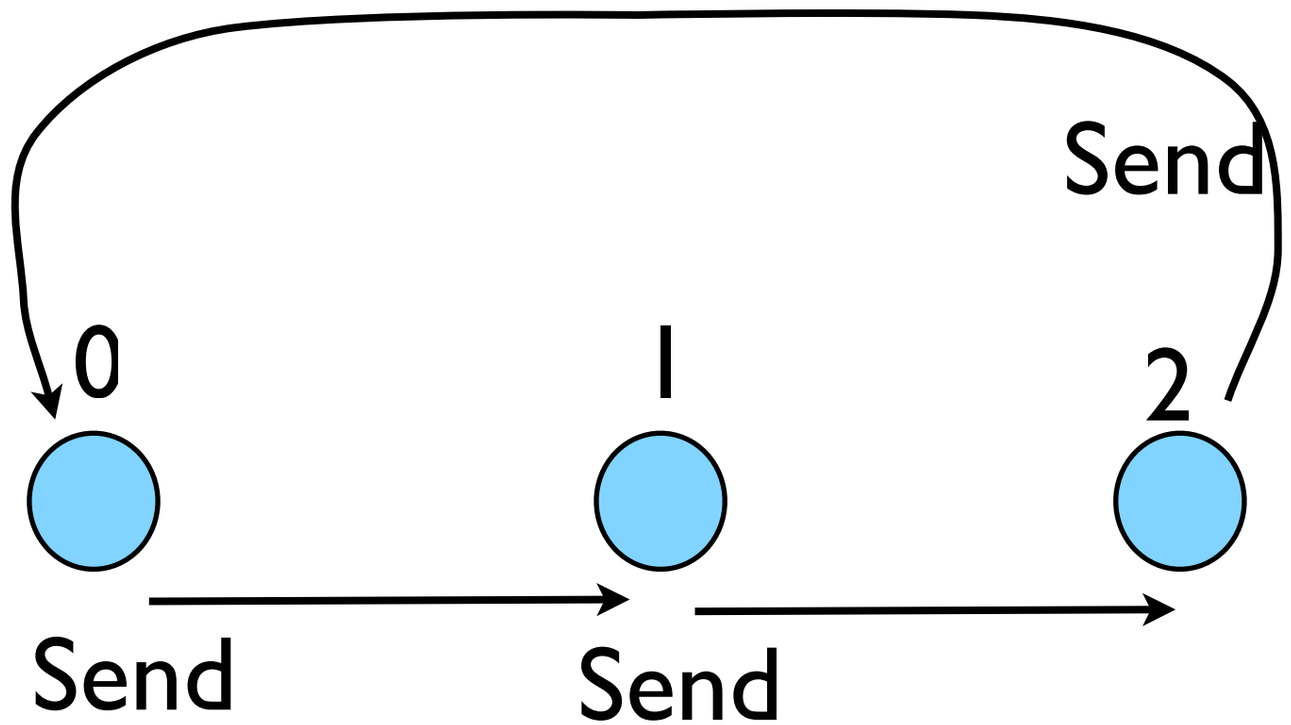
Implement periodic boundary conditions

- cp secondmessage.c thirdmessage.c
- edit so it `wraps around'
- mpicc thirdmessage.c -o thirdmessage
- mpirun -np 3 thirdmessage



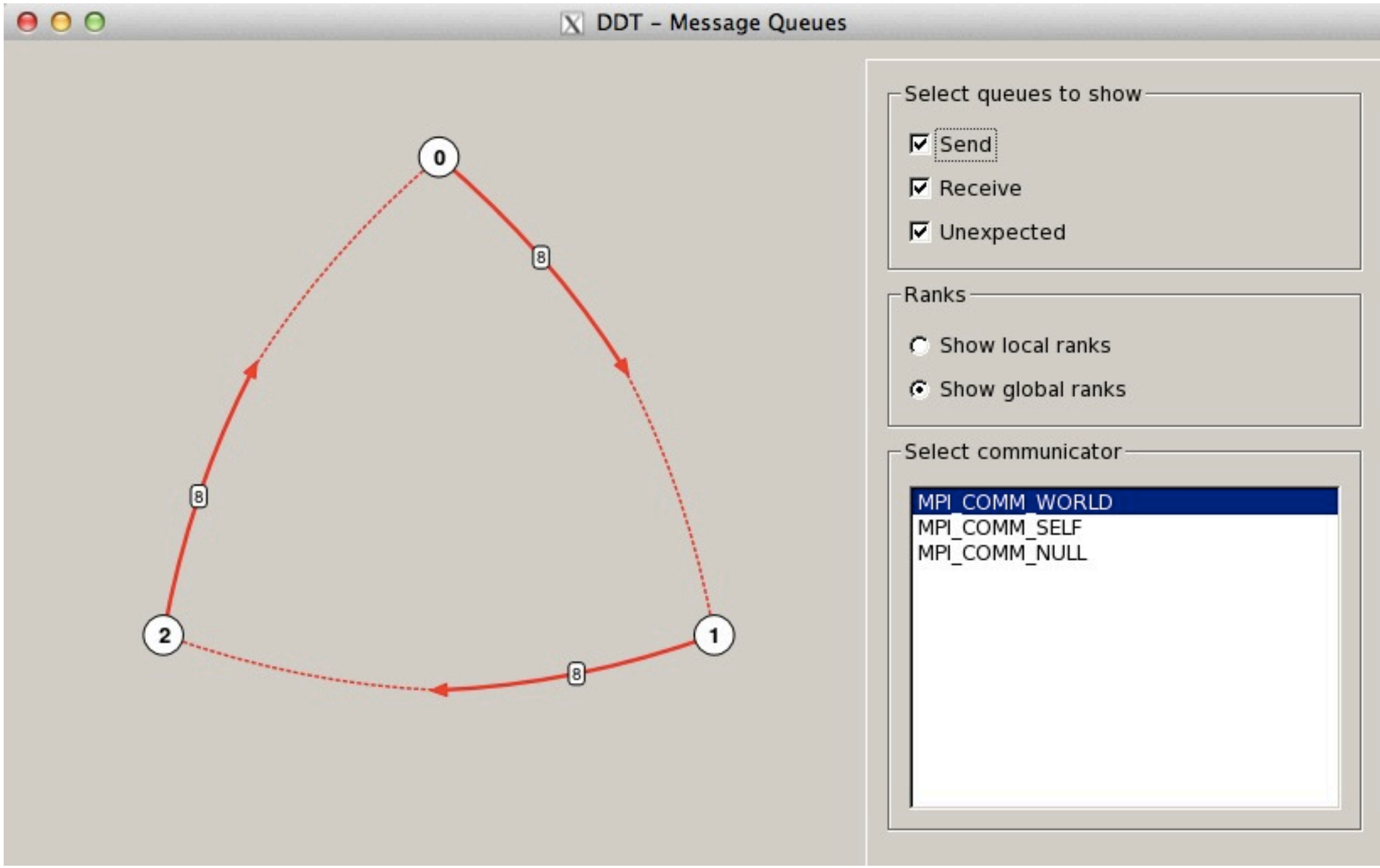
```
left = rank-1
if (left < 0) left = comsize-1
right = rank+1
if (right >= comsize) right = 0

call MPI_Ssend(msgsent, 1, MPI_DOUBLE_PRECISION, right, &
               tag, MPI_COMM_WORLD, ierr)
call MPI_Recv(msgrcvd, 1, MPI_DOUBLE_PRECISION, left, &
               tag, MPI_COMM_WORLD, status, ierr)
```






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

call MPI_Ssend(msgsent, 1, MPI_DOUBLE_PRECISION, right, &
               tag, MPI_COMM_WORLD, ierr)
call MPI_Recv(msgrcvd, 1, MPI_DOUBLE_PRECISION, left, &
               tag, MPI_COMM_WORLD, status, ierr)
```

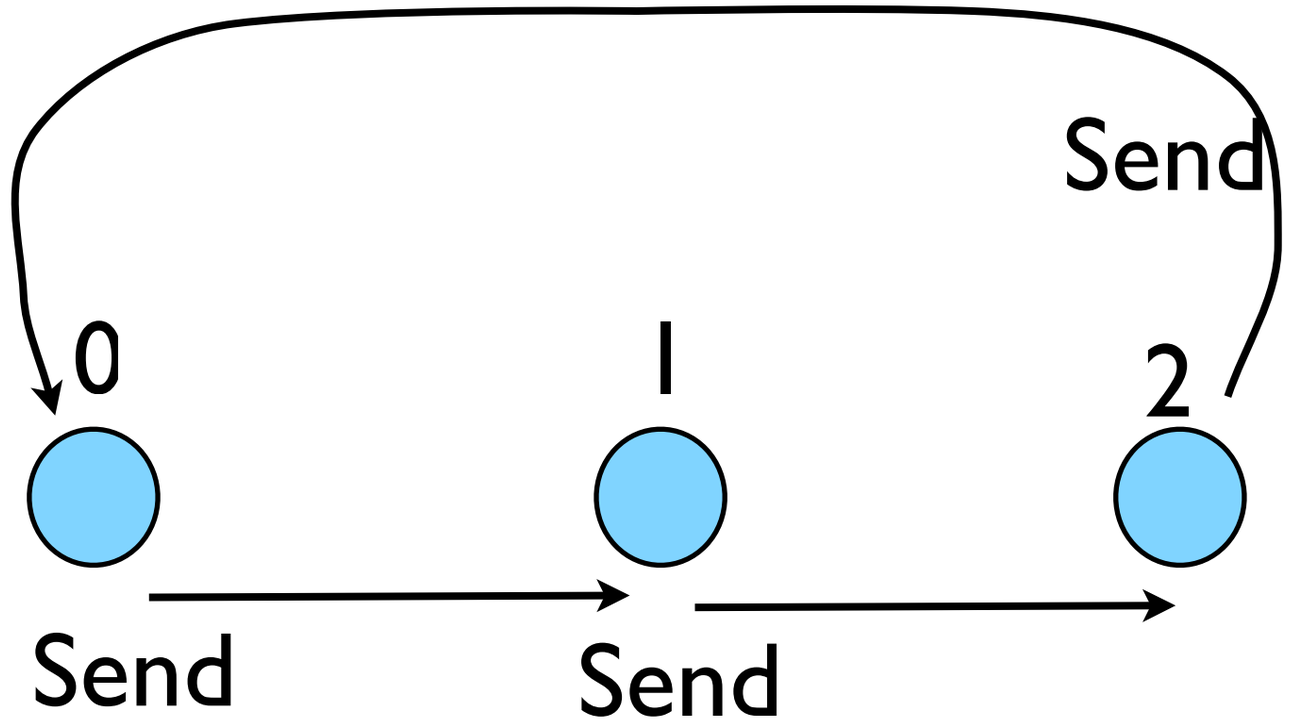
0,1,2




```
$ ddt -np 3 ./thirdmessage
[run; then pause]
[View -> Show Message Queues]
```

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.



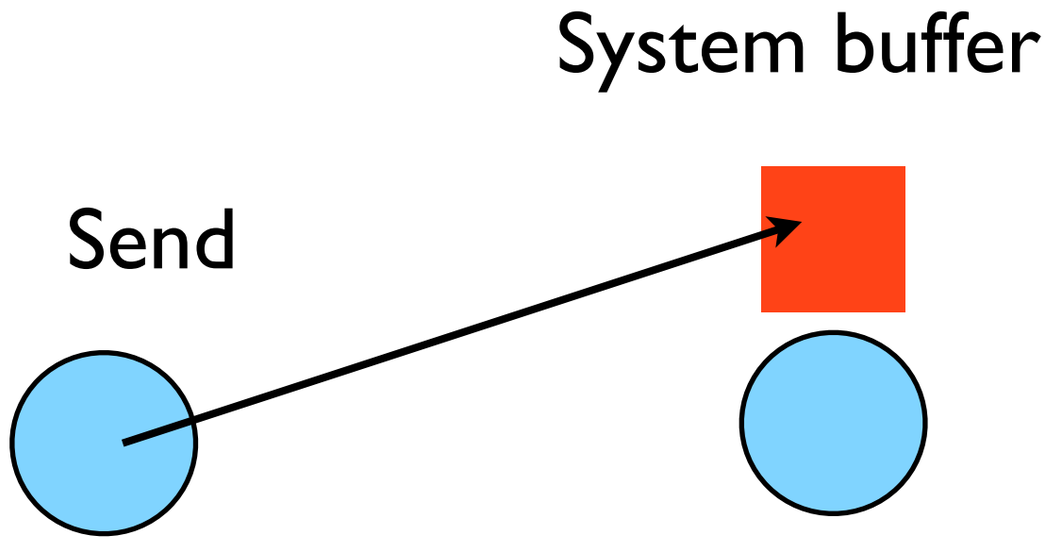
Big MPI
Lesson #1

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

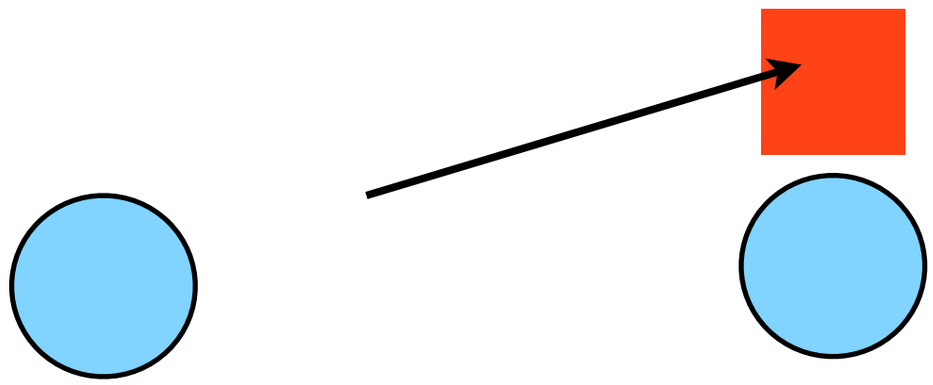
Different versions of SEND

- 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



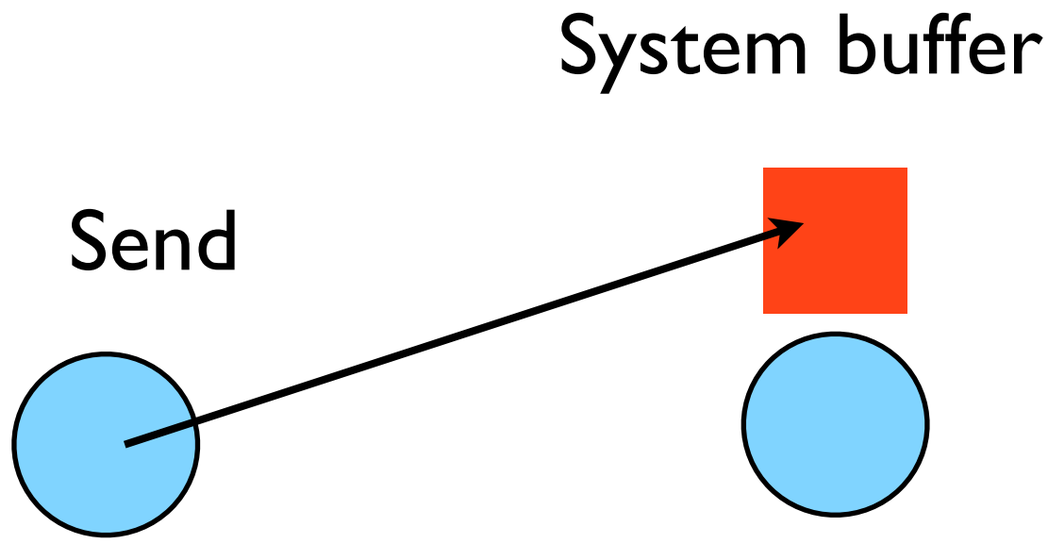
(Non) Blocking



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



Buffering is dangerous!

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

Buffering

Cisco Blog > High Performance Computing Networking

Top 10 reasons why buffered sends are evil

February 13, 2012 at 5:00 am PST

Jeff Squyres

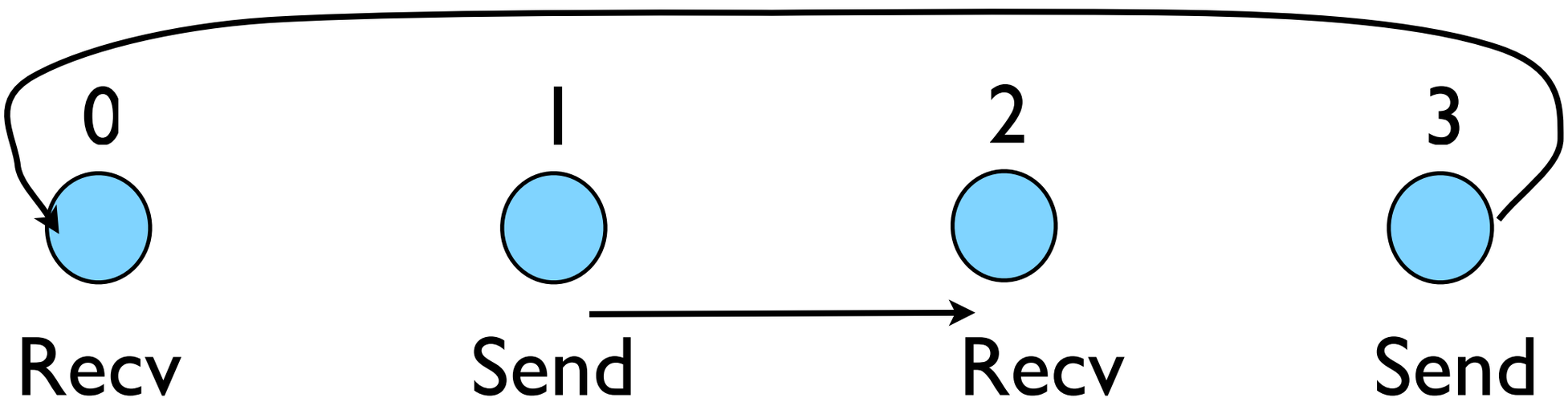
643 4 Tweet 1 Like

I made an offhand remark in my last entry about how MPI buffered sends are evil. In a comment on that entry, @brockpalen asked me why.

I gave a brief explanation in a comment reply, but the subject is enough to warrant its own entry.

So here it is — my top 10 reasons why MPI_BSEND (and its two variants) are evil:

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?

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

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

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

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

    msgsent = rank*rank;
    msgrcvd = -999;

    if (rank % 2 == 0) {
        ierr = MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right,
                         tag, MPI_COMM_WORLD);
        ierr = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left,
                         tag, MPI_COMM_WORLD, &rstatus);
    } else {
        ierr = MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left,
                         tag, MPI_COMM_WORLD, &rstatus);
        ierr = MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right,
                         tag, MPI_COMM_WORLD);
    }

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

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

Evens send first

Then odds

fourthmessage.c

Something new: Sendrecv

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

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

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

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

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

    msgsent = rank*rank;
    msgrcvd = -999;

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

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

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

fifthmessage.c

Sendrecv = Send + Recv

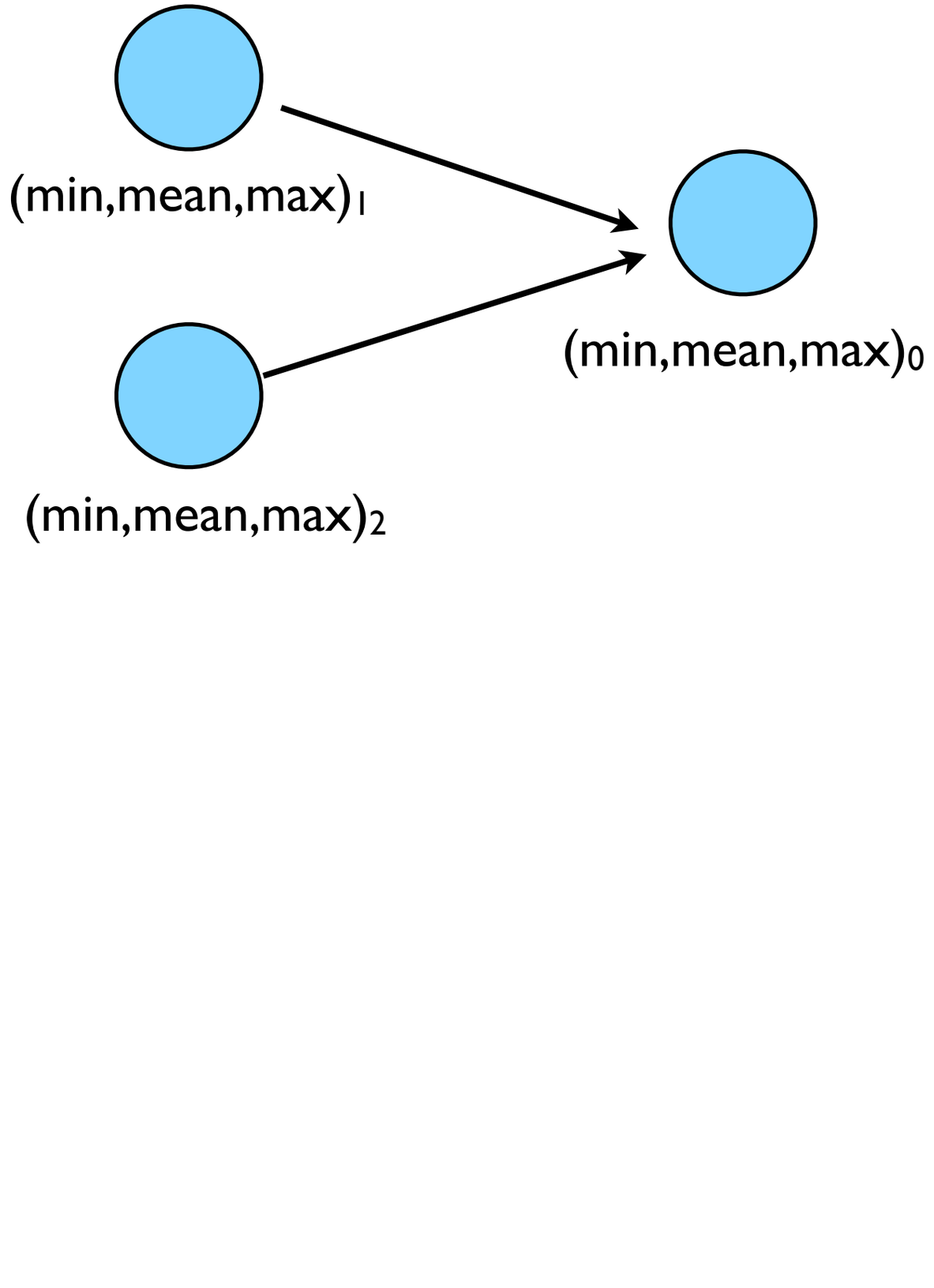
```
MPI_Status status;

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

Send Args

Recv Args

Why are there two different tags/types/counts?

Min, Mean, Max of numbers

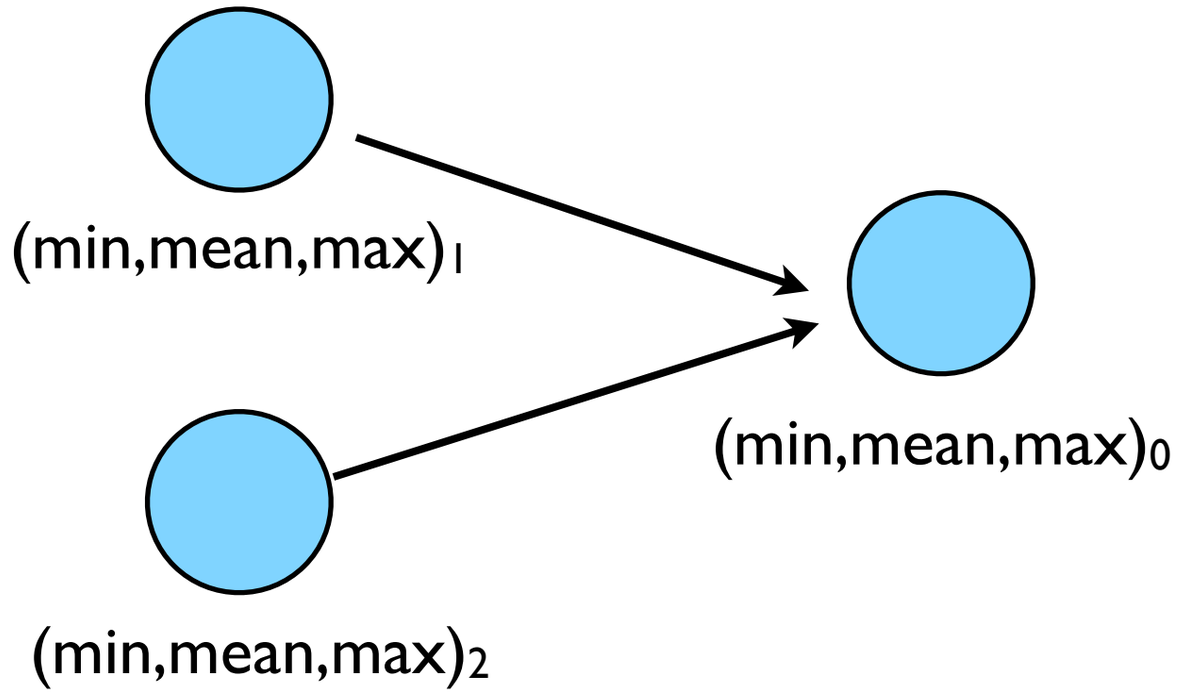


- Lets try some code that calculates the min/mean/max of a bunch of random numbers -1..1. Should go to -1,0,+1 for large N.
- Each gets their partial results and sends it to some node, say node 0 (why node 0?)
- ~ppp/mpi-intro/minmeanmax.{c,f90}
- How to MPI it?

```
// generate random data

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

// find min/mean/max

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

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

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

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

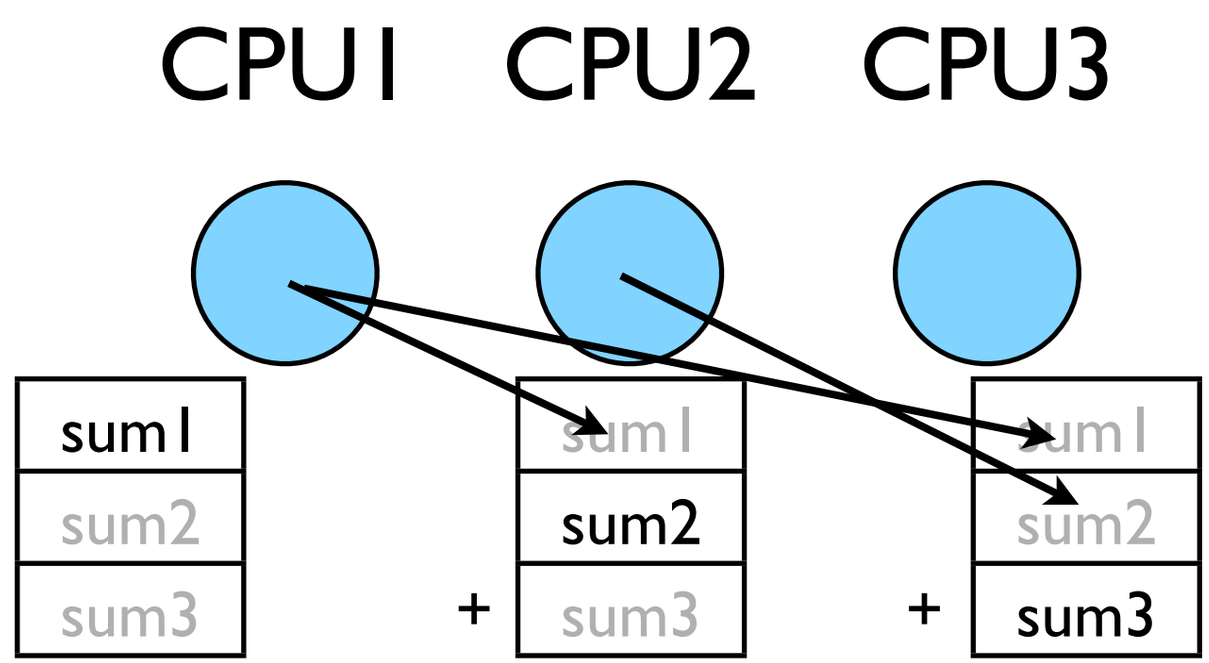


Q: are these sends/recvd adequately paired?

minmeanmax-mpi.f90

Inefficient!

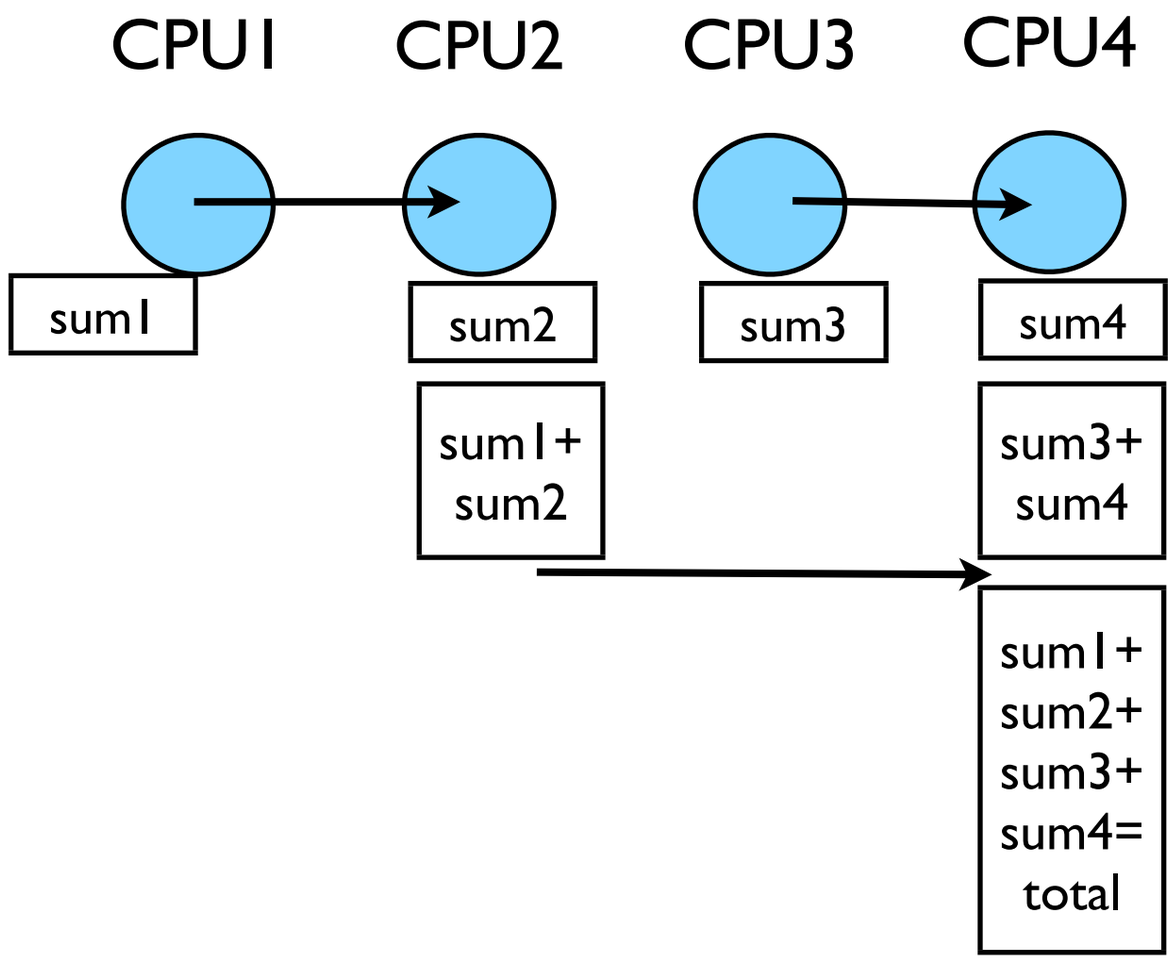
- Requires (P-1) messages,
  2(P-1) if everyone then needs to get the answer.



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
printf("Rank %d: Min/mean/max = %f,%f,%f\n",
            rank, datamin, datamean, datamax);

MPI_Reduce(&datamin, &globmin, 1, MPI_FLOAT,
                     MPI_MIN, masterproc, MPI_COMM_WORLD);
// To send to all: MPI_Allreduce(&datamin, &globmin,
//                     1, MPI_FLOAT, MPI_MIN, MPI_COMM_WOR
MPI_Reduce(&datamean, &globmean, 1, MPI_FLOAT,
                     MPI_SUM, masterproc, MPI_COMM_WORLD);

MPI_Reduce(&datamax, &globmax, 1, MPI_FLOAT,
                     MPI_MAX, masterproc, MPI_COMM_WORLD);
```

```c
globmean /= size;
if (rank == masterproc) {
    printf("Min/mean/max = %f,%f,%f\n", globmin,
                                        globmean, globmax);
}
```

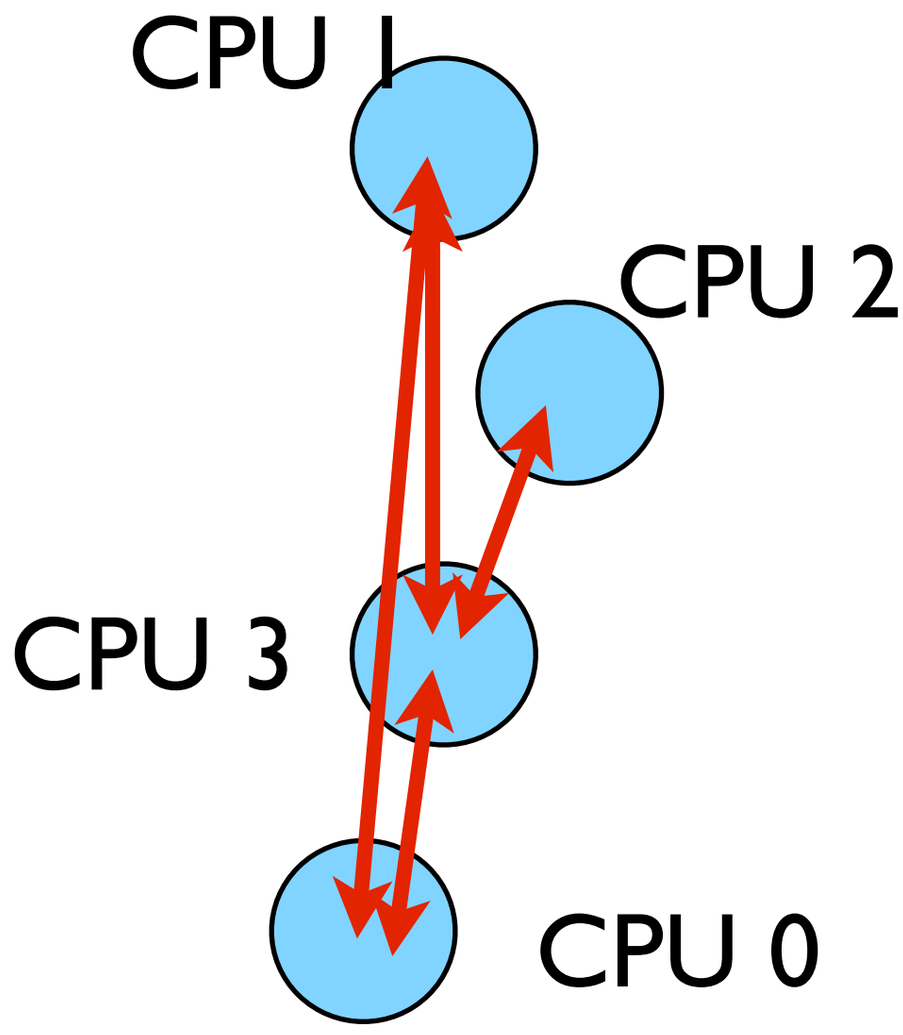
MPI_Reduce and MPI_Allreduce

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

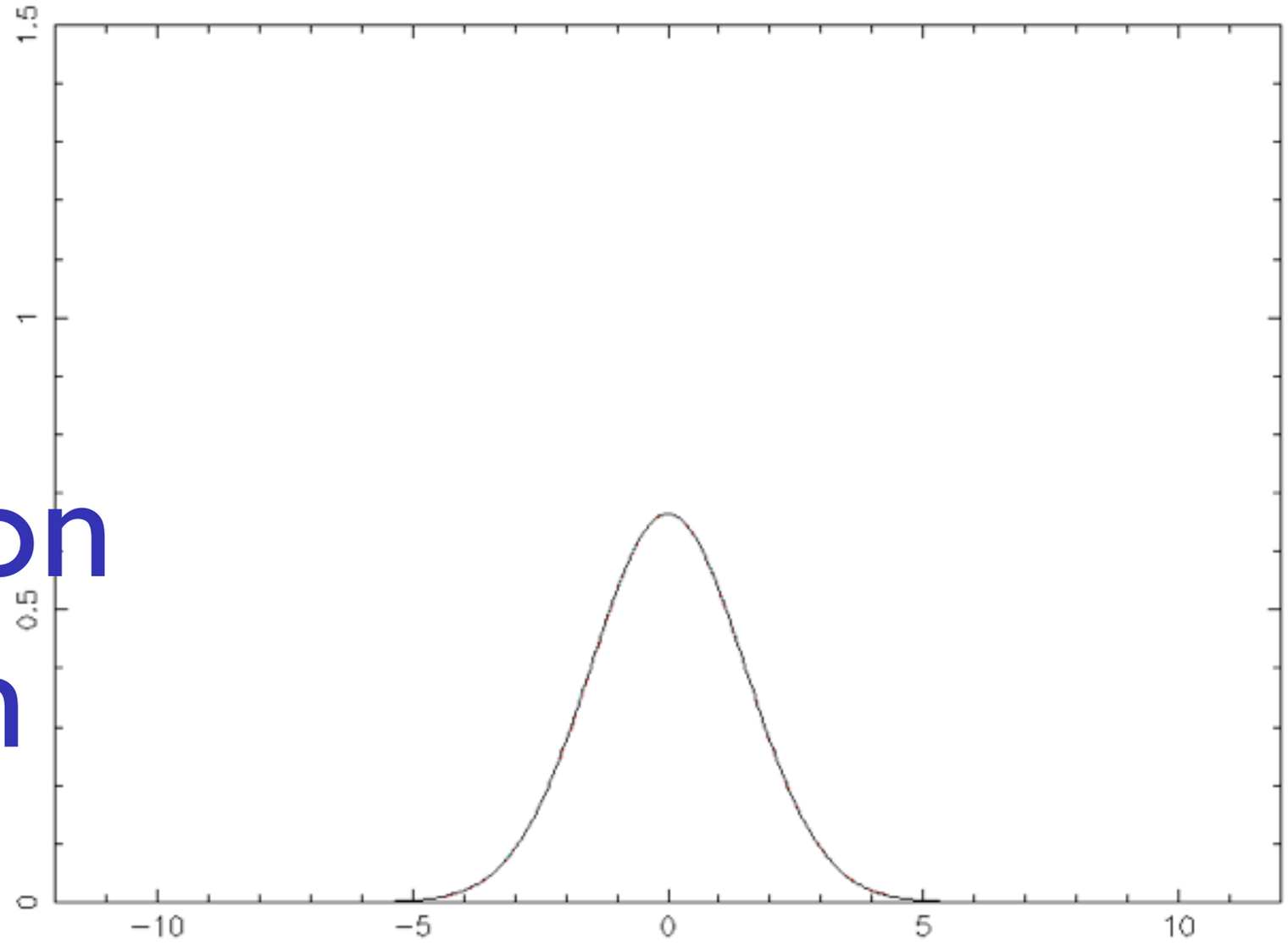
minmeanmax-allreduce.c

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

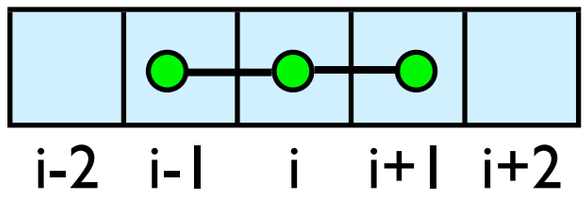


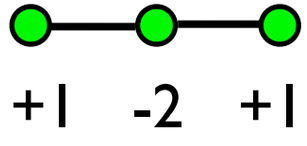

```
make diffusion
./diffusion
```

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

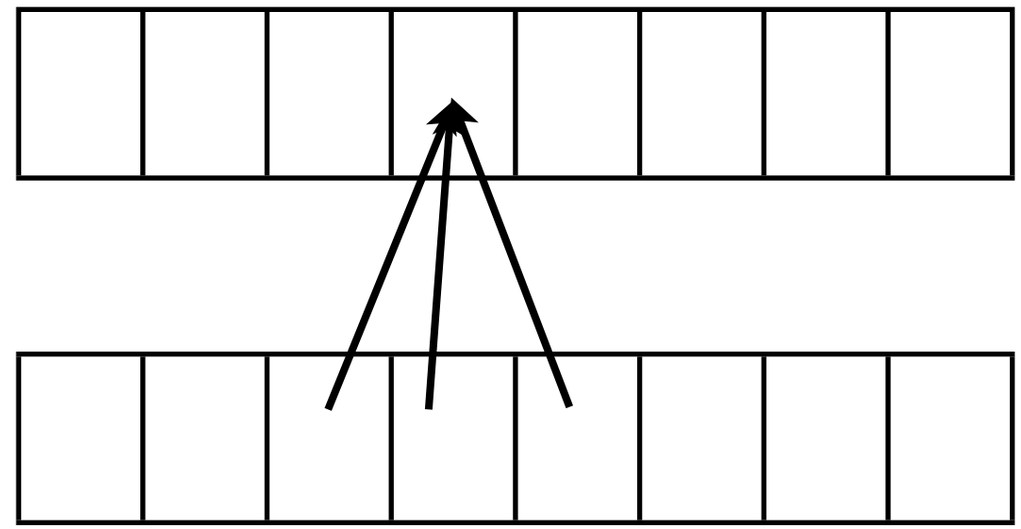
$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2}$$

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

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

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

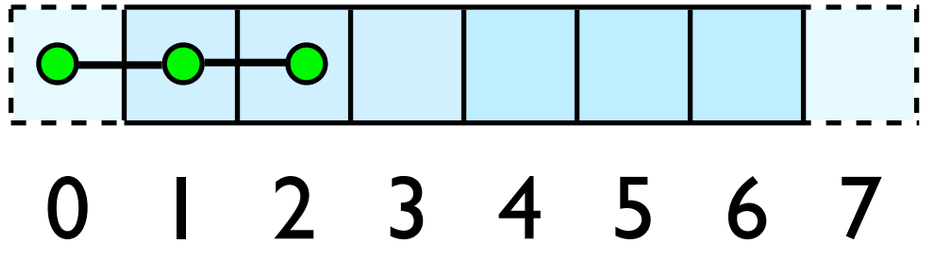
- Simple 1d PDE
- Each timestep, new data for T[i] requires old data for T[i+1],T[i],T[i-1]



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

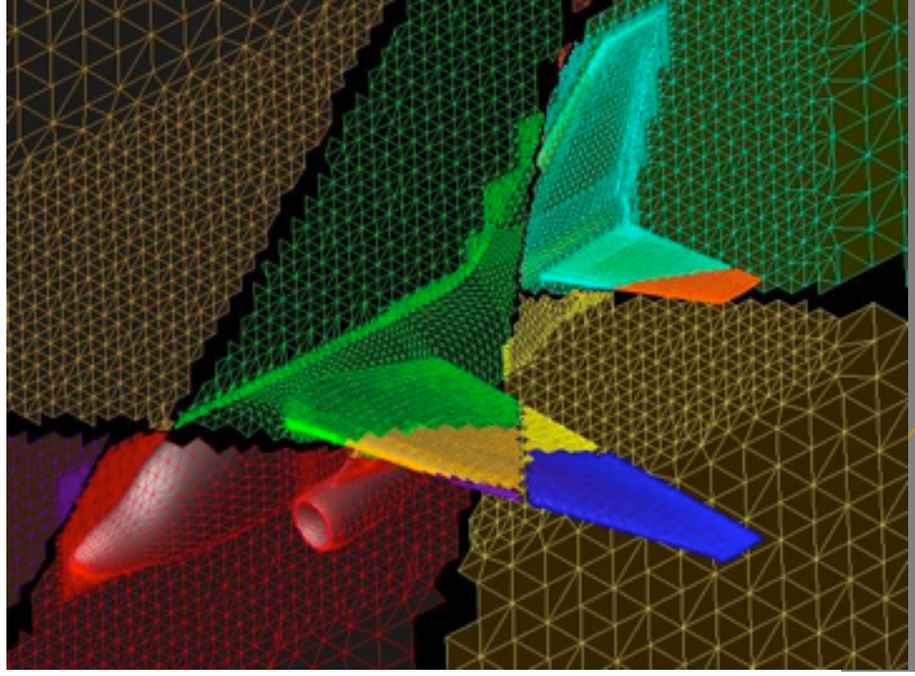


ng = 1

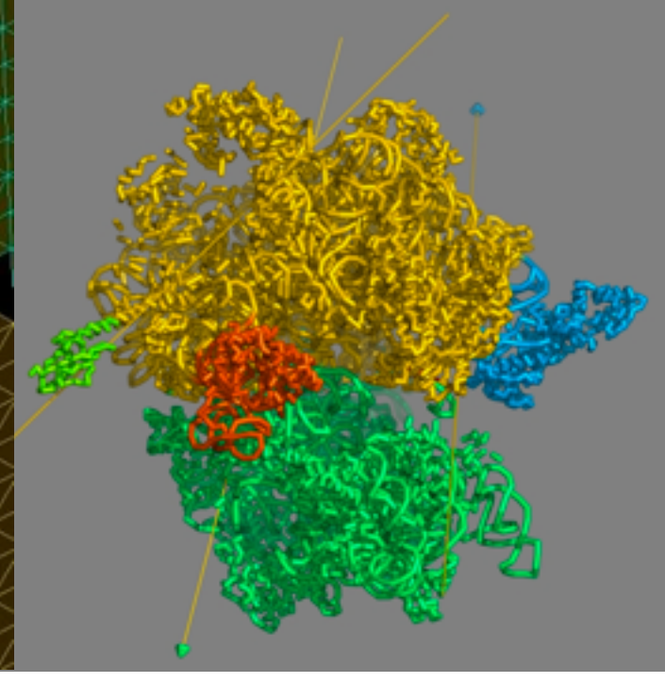
loop from ng, N - 2 ng

Domain Decomposition

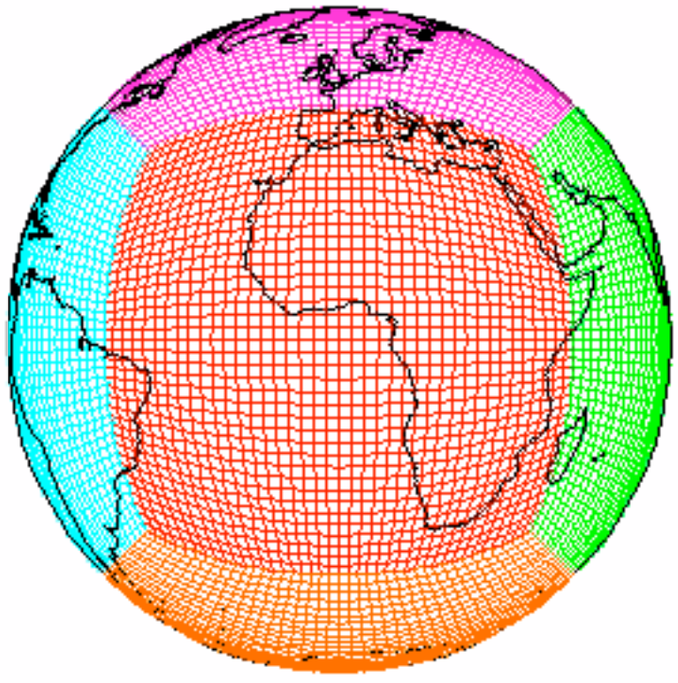
- A very common approach to parallelizing on distributed memory computers
- Maintain Locality; need local data mostly, this means only surface data needs to be sent between processes.



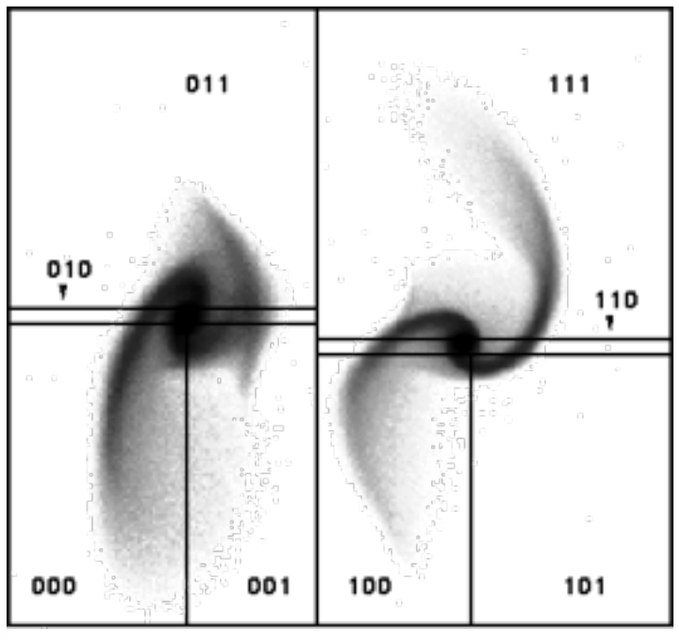
http://adg.stanford.edu/aa241/design/compaero.html



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



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



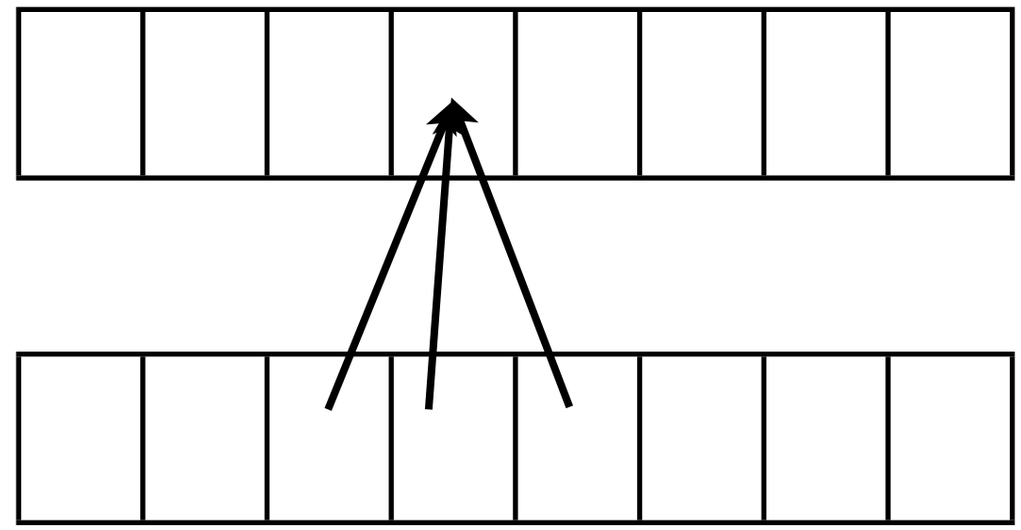
http://www.cita.utoronto.ca/~dubinski/treecode/node8.html

Implement a diffusion equation in MPI

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

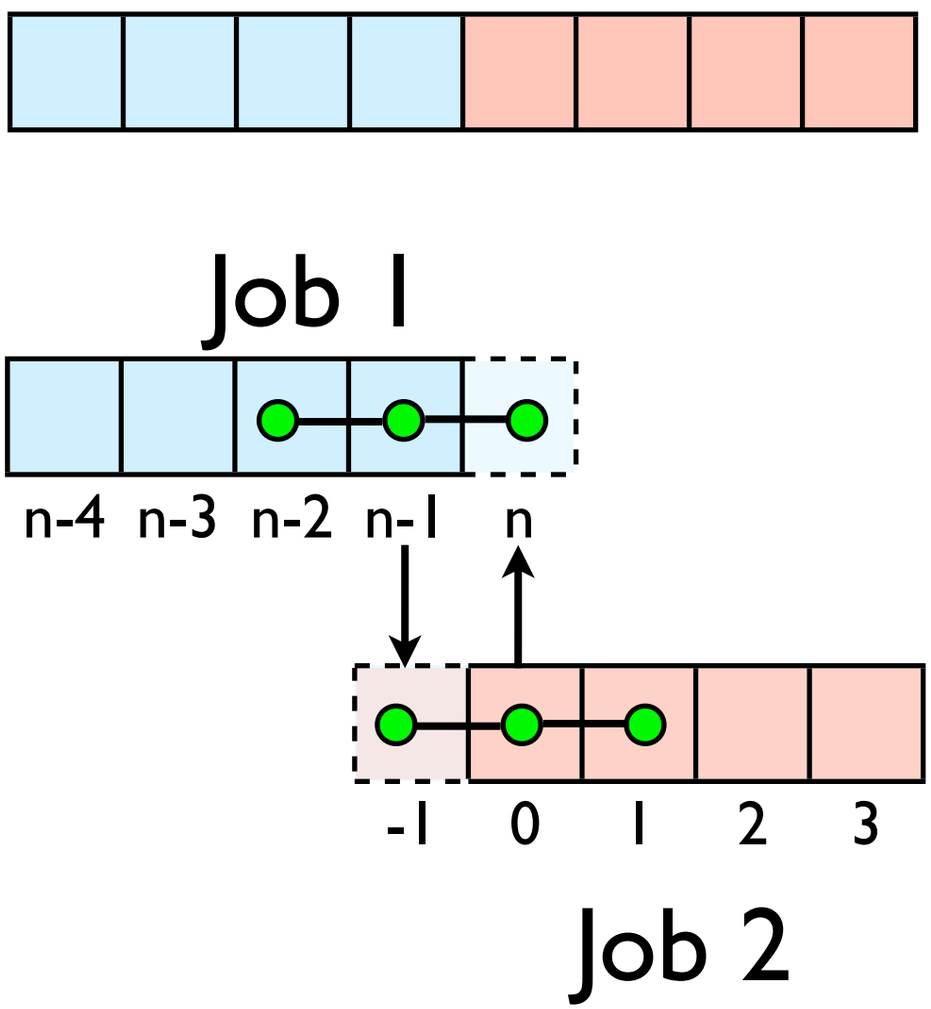
$$T_i^{n+1} = T_i^n + \frac{D\Delta t}{\Delta x^2}\left(T_{i+1}^n - 2T_i^n + T_{i-1}^n\right)$$

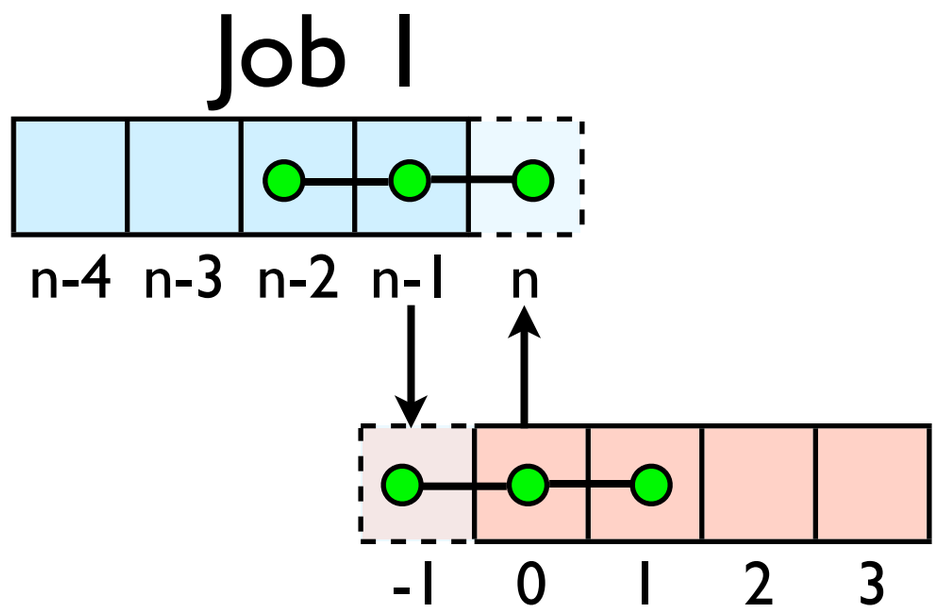
- Need one neighboring number per neighbor per timestep



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 diffusion.c diffusion-mpi.c
  or
- Make an MPI-ed version of diffusion equation
- (Build: `make diffusion-mpi`)
- Test on 1..8 procs
- Time on 1..8 (without DPGPLOT)

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

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,...
```