Programming Distributed Memory Systems with MPI

2016 Ontario Summer School
on High Performance Computing

Scott Northrup
July 12-13
SciNet - Toronto

Intro to Message Passing Interface (MPI)

HPC Systems

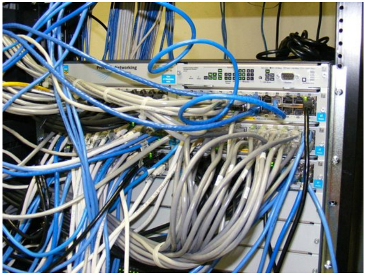
Architectures

- Clusters, or, distributed memory machines
  - A bunch of servers linked together by a network ("interconnect").
  - GigE, Infiniband, Cray Gemini/Aries, IBM BGQ Torus
- Symmetric Multiprocessor (SMP) machines, or, shared memory machines
  - These can all see the same memory, typically a limited number of cores.
  - IBM Pseries, Cray SMT, SGI Altix/UV
- Vector machines.
  - No longer dominant in HPC anymore.
  - Cray, NEC
- Accelerator (GPU, Cell, MIC, FPGA)
  - Heterogeneous use of standard CPU's with a specialized accelerator.
  - NVIDIA, AMD, Intel, Xilinx, Altera

Distributed Memory: Clusters

Simplest type of parallel computer to build

- Take existing powerful standalone computers
- And network them



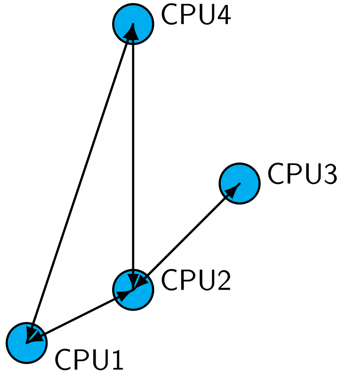
(source: http://flickr.com/photos/eurleif)

Distributed Memory: Clusters

Each node is independent!

Parallel code consists of programs running on separate computers, communicating with each other.

Could be entirely different programs.

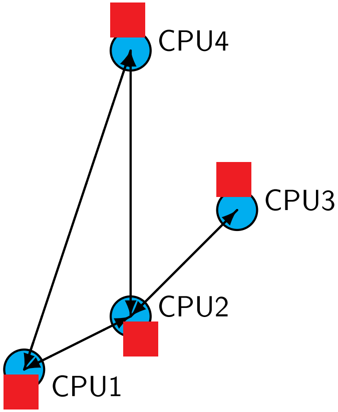


Distributed Memory: Clusters

Each node is independent!
Parallel code consists of programs running on separate computers, communicating with each other.
Could be entirely different programs.

Each node has own memory!
Whenever it needs data from another region, requests it from that CPU.

Usual model: “message passing”



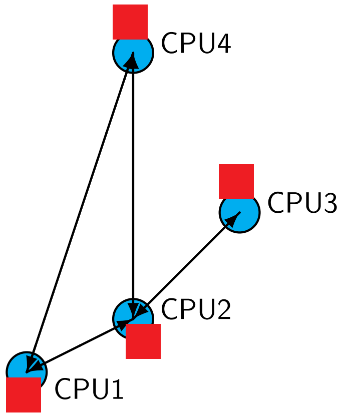
Clusters+Message Passing

Hardware:

Easy to build
(Harder to build well)
Can build larger and larger clusters relatively easily

Software:

Every communication has to be hand-coded: hard to program



HPC Programming Models

Languages

- serial
  - C, C++, Fortran
- threaded (shared memory)
  - OpenMP, pthreads
- message passing (distributed memory)
  - MPI, PGAS (UPC, Coarray Fortran)
- accelerator (GPU, Cell, MIC, FPGA)
  - CUDA, OpenCL, OpenACC

Task (function, control) Parallelism

Work to be done is decomposed across processors

- e.g. divide and conquer
- each processor responsible for some part of the algorithm
- communication mechanism is significant
- must be possible for different processors to be performing different tasks

Intro to Message Passing Interface (MPI)

Message Passing Interface (MPI)

What is it?

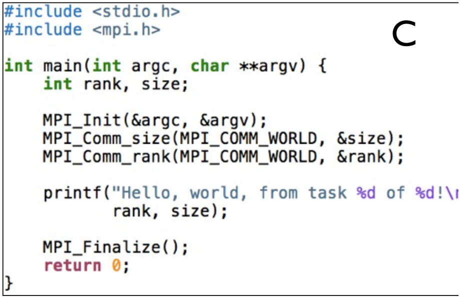
- An open standard library interface for message passing, ratified by the MPI Forum
- Version: 1.0 (1994), 1.1 (1995), 1.2 (1997), 1.3 (2008)
- Version: 2.0 (1997), 2.1 (2008), 2.2 (2009)
- Version: 3.0 (2012)

MPI Implementations

- OpenMPI (www.open-mpi.org)
  - OpenMPI 1.8.x
  - SciNet GPC: **`module load gcc openmpi`**
  - SciNet GPC: **`module load intel openmpi`**
- MPICH2 (www.mpich.org)
  - MPICH 3.x, MVAPICH2 2.x , IntelMPI 5.x
  - SciNet GPC: **`module load intel intelmpi`**

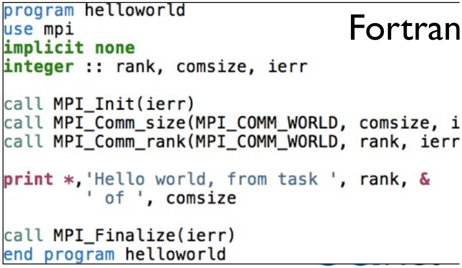
MPI is a **Library** for Message-Passing

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



C

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

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

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

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

    MPI_Finalize();
    return 0;
}
```

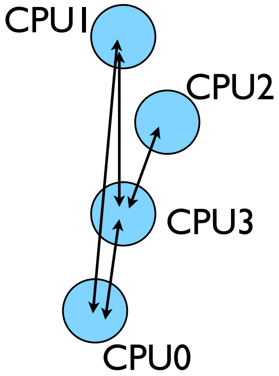


Fortran

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

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, i
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr

print *,'Hello world, from task ', rank, &
        ' of ', comsize

call MPI_Finalize(ierr)
end program helloworld
```

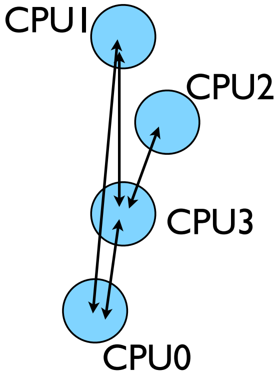
MPI is a Library for **Message-Passing**

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



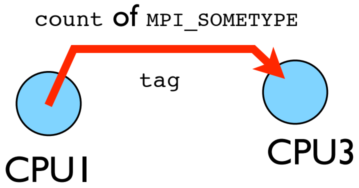
MPI is a Library for **Message-Passing**

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



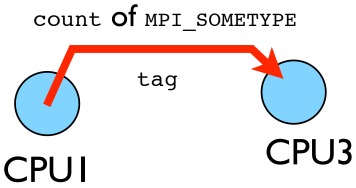
Messages

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



Messages

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



Size of MPI Library

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

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

SciNet Access

Access to SciNet

- Log into SciNet and get a copy of the source.
- Guest SciNet accounts available from instructor.

```
$ssh -Y USER@login.scinet.utoronto.ca
$ssh -Y gpc0[1-8]
$cd $SCRATCH
$cp -r /scinet/course/ssc2016/mpi .
$source mpi/setup
```

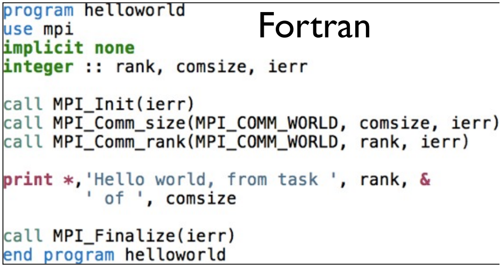
Submit a job

```
$qsub -l nodes=1:ppn=8,walltime=8:00:00 -I -X -q
teach
```

Hello World

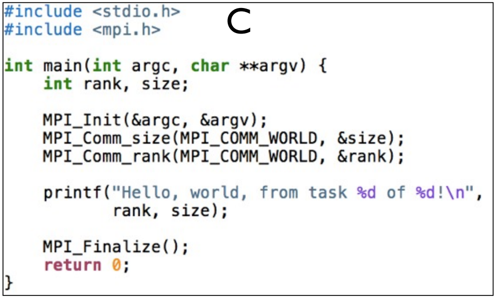
- The obligatory starting point
- cd mpi/mpi-intro
- Type it in, compile and run it together

Fortran



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

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

print *,'Hello world, from task ', rank, &
        ' of ', comsize

call MPI_Finalize(ierr)
end program helloworld
```

C



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

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

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

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

    MPI_Finalize();
    return 0;
}
```

edit hello-world.c or .f90

```
$ mpif90 hello-world.f90
         -o hello-world
```

or

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

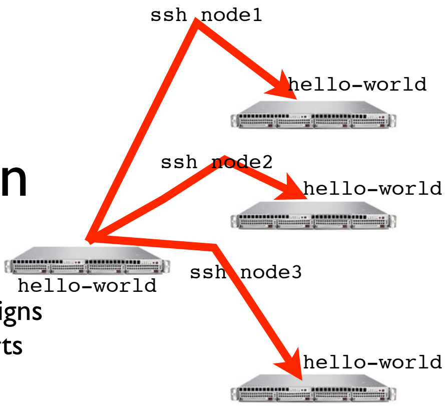
t

What mpicc/ mpif77 do

- Just wrappers for the system C, Fortran compilers that have the various -I, -L clauses in there automaticaly
- -v option (sharcnet) or --showme (OpenMPI) shows which options are being used

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

gcc -I/usr/local/include
 -pthread hello-world.c -o
hello-world -L/usr/local/lib
-lmpi -lopen-rte -lopen-pal
-ldl -Wl,--export-dynamic -lnsl
-lutil -lm -ldl
```

What mpirun does



- Launches n processes, assigns each an MPI rank and starts the program
- For multinode run, has a list of nodes, ssh's to each node and launches the program

Number of Processes

- Number of processes to use is almost always equal to the number of processors
- But not necessarily.
- On your nodes, what happens when you run this?

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

mpirun runs *any* program

- mpirun will start that process-launching procedure for any progam
- Sets variables somehow that mpi programs recognize so that they know which process they are

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

Example: “Hello World”

```
$mpirun -np 4 ./hello-world
Hello from task 2 of 4 world
Hello from task 1 of 4 world
Hello from task 0 of 4 world
Hello from task 3 of 4 world
```

Example: "Hello World"

```
$mpirun -np 4 ./hello-world
Hello from task 2 of 4 world
Hello from task 1 of 4 world
Hello from task 0 of 4 world
Hello from task 3 of 4 world
```

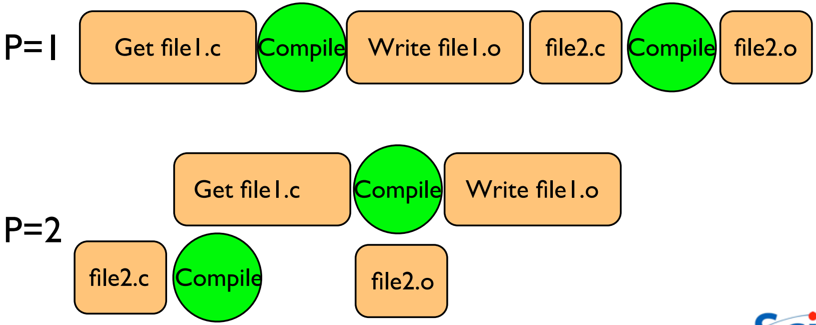
```
$mpirun -tag-output -np 4 ./hello-world
[1,3]<stdout>:Hello from task 3 of 4 world
[1,2]<stdout>:Hello from task 2 of 4 world
[1,0]<stdout>:Hello from task 0 of 4 world
[1,1]<stdout>:Hello from task 1 of 4 world
```

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

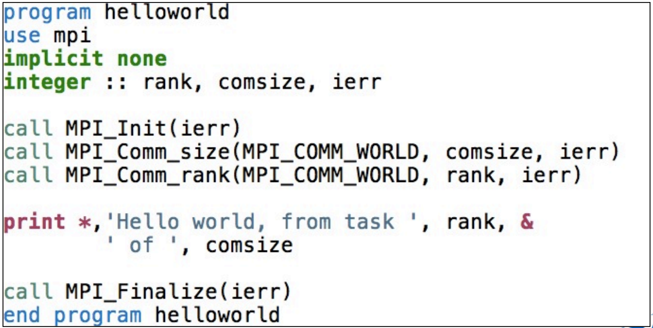
```
$ make
$ make -j 2
$ make -j
```

Overlapping Computation with I/O



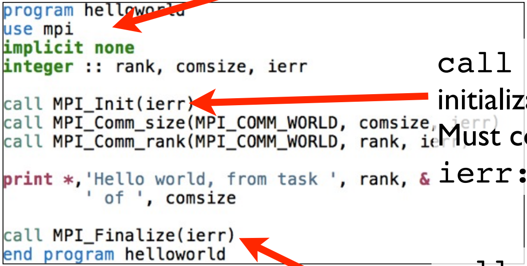
What the code does

- (FORTRAN version; C is similar)



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

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

print *,'Hello world, from task ', rank, &
        ' of ', comsize

call MPI_Finalize(ierr)
end program helloworld
```

`use mpi` : imports declarations for MPI function calls



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

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

print *,'Hello world, from task ', rank, &
        ' of ', comsize

call MPI_Finalize(ierr)
end program helloworld
```

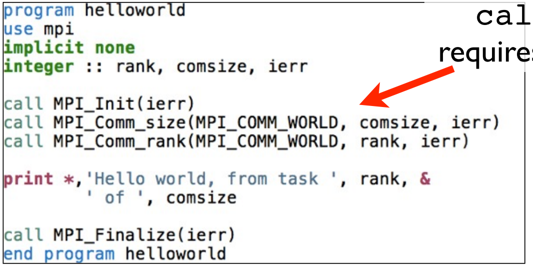
`call MPI_INIT(ierr)`: initialization for MPI library. Must come first.

`ierr`: Returns any error code.

`call MPI_FINALIZE(ierr)`: close up MPI stuff. Must come last.

`ierr`: Returns any error code.

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



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

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

print *,'Hello world, from task ', rank, &
        ' of ', comsize

call MPI_Finalize(ierr)
end program helloworld
```

MPI Basics

Basic MPI Components

- `#include <mpi.h>` : MPI library details
- `MPI_Init(&argc, &argv);` : MPI Intialization, must come first
- `MPI_Finalize()` : Finializes MPI, must come last
- `ierr` : Returns error code

MPI Basics

Basic MPI Components

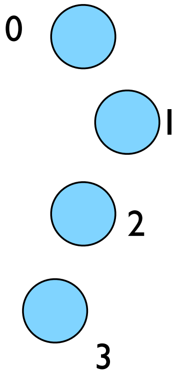
- `#include <mpi.h>` : MPI library details
- `MPI_Init(&argc, &argv);` : MPI Intialization, must come first
- `MPI_Finalize()` : Finializes MPI, must come last
- `ierr` : Returns error code

Communicator Components

- `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`
- `MPI_Comm_size(MPI_COMM_WORLD, &size)`

Communicators

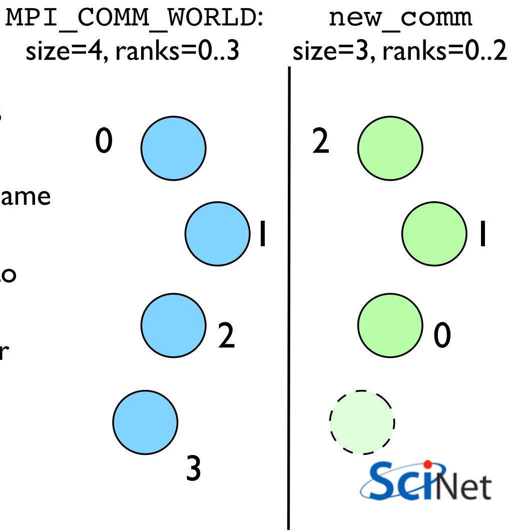
- MPI groups processes into communicators.
- Each communicator has some size -- number of tasks.
- Each task has a rank 0..size-1
- Every task in your program belongs to `MPI_COMM_WORLD`



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

Communicators

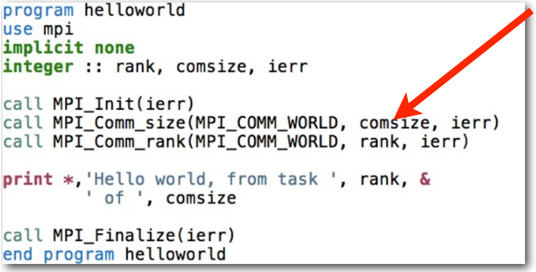
- Can create our own communicators over the same tasks
- May break the tasks up into subgroups
- May just re-order them for some reason



MPI Basics

Communicator Components

- `MPI_COMM_WORLD` :
  Global Communicator
- `MPI_Comm_rank(MPI_COMM_WORLD, &rank)` :
  Get current tasks rank
- `MPI_Comm_size(MPI_COMM_WORLD, &size)` :
  Get communicator size



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

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

print *,'Hello world, from task ', rank, &
        ' of ', comsize

call MPI_Finalize(ierr)
end program helloworld
```

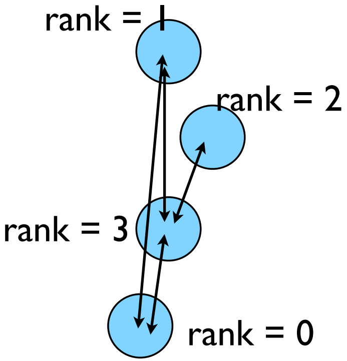
`call MPI_COMM_RANK,`
`call MPI_COMM_SIZE:`

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

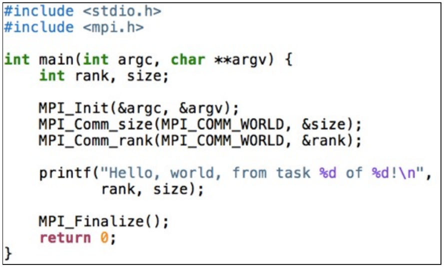
put answers in `rank` and
`size`

Rank and Size much more important in MPI than OpenMP

- In OpenMP, compiler assigns jobs to each thread; don't need to know which one you are.
- MPI: processes determine amongst themselves which piece of puzzle to work on, then communicate with appropriate others.

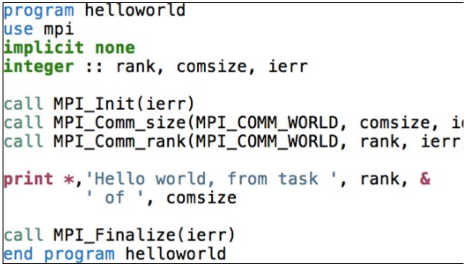


C



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

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

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

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

    MPI_Finalize();
    return 0;
}
```

Fortran



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

call MPI_Init(ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, i
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr

print *,'Hello world, from task ', rank, &
        ' of ', comsize

call MPI_Finalize(ierr)
end program helloworld
```

- `#include <mpi.h>` vs `use mpi`
- C - functions **return** ierr;
- Fortran - **pass** ierr
- MPI_Init

SciNet

Intro to Message Passing Interface (MPI)

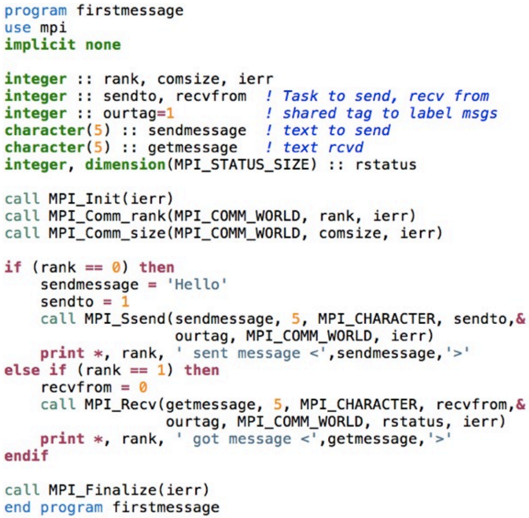
Our first real MPI program - but no Ms are P'ed!

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

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

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

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

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

Fortran version

- Let's fix this
- mpif90 -o firstmessage firstmessage.f90
- mpirun -np 2 ./ firstmessage
- FORTRAN - MPI_CHARACTER



```fortran
program firstmessage
use mpi
implicit none

integer :: rank, comsize, ierr
integer :: sendto, recvfrom  ! Task to send, recv from
integer :: ourtag=1          ! shared tag to label msgs
character(5) :: sendmessage  ! text to send
character(5) :: getmessage   ! text rcvd
integer, dimension(MPI_STATUS_SIZE) :: rstatus

call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, comsize, ierr)

if (rank == 0) then
    sendmessage = 'Hello'
    sendto = 1
    call MPI_Ssend(sendmessage, 5, MPI_CHARACTER, sendto,&
                   ourtag, MPI_COMM_WORLD, ierr)
    print *, rank, ' sent message <',sendmessage,'>'
else if (rank == 1) then
    recvfrom = 0
    call MPI_Recv(getmessage, 5, MPI_CHARACTER, recvfrom,&
                  ourtag, MPI_COMM_WORLD, rstatus, ierr)
    print *, rank, ' got message <',getmessage,'>'
endif

call MPI_Finalize(ierr)
end program firstmessage
```

C - Send and Receive

```
MPI_Status status;

ierr = MPI_Ssend(sendptr, count, MPI_TYPE, destination,
                 tag, Communicator);

ierr = MPI_Recv(rcvptr, count, MPI_TYPE, source, tag,
                 Communicator, status);
```

Fortran - Send and Receive

```
integer status(MPI_STATUS_SIZE)

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

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

Special Source/Dest: MPI_PROC_NULL

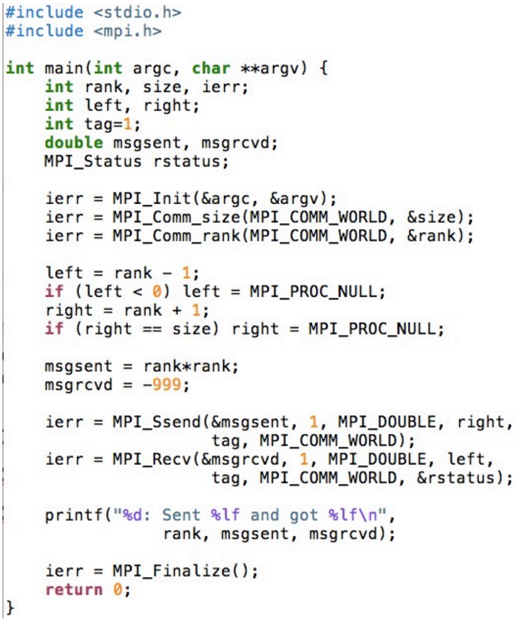
`MPI_PROC_NULL` basically ignores the relevant operation; can lead to cleaner code.

Special Source: MPI_ANY_SOURCE

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

More complicated example:

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

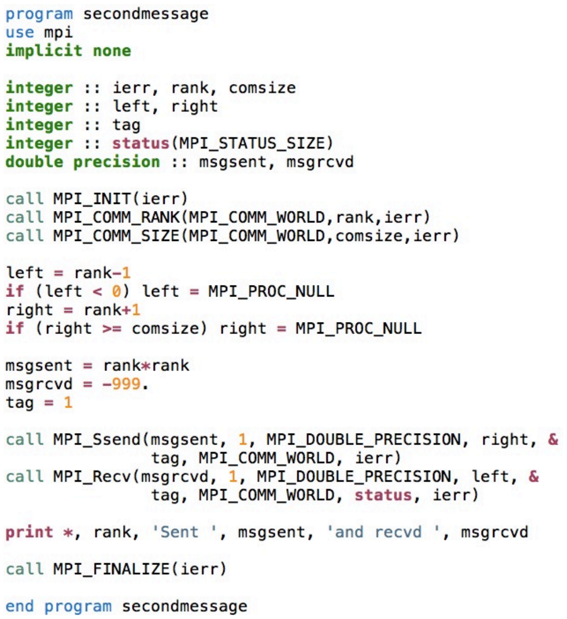


```c
#include <stdio.h>
#include <mpi.h>

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

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

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

    msgsent = rank*rank;
    msgrcvd = -999;

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

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

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

t

More complicated example:

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



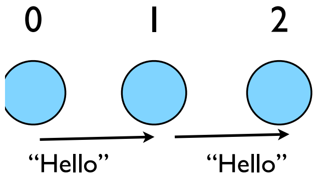
```
program secondmessage
use mpi
implicit none

integer :: ierr, rank, comsize
integer :: left, right
integer :: tag
integer :: status(MPI_STATUS_SIZE)
double precision :: msgsent, msgrcvd

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

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

msgsent = rank*rank
msgrcvd = -999.
tag = 1

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

print *, rank, 'Sent ', msgsent, 'and recvd ', msgrcvd

call MPI_FINALIZE(ierr)

end program secondmessage
```

Compile and run

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

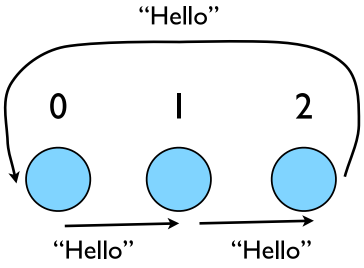
```
$ mpirun -np 4 ./secondmessage
3: Sent 9.000000 and got 4.000000
0: Sent 0.000000 and got -999.000
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
```

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

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

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

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

    msgsent = rank*rank;
    msgrcvd = -999;

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

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

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

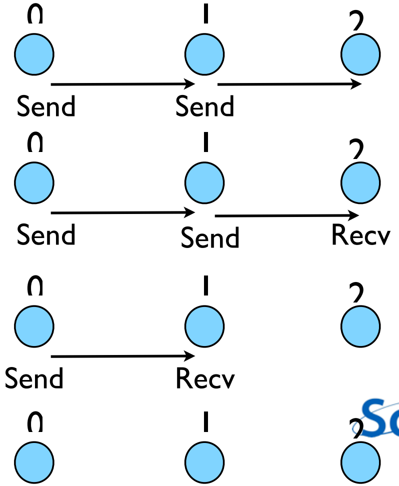


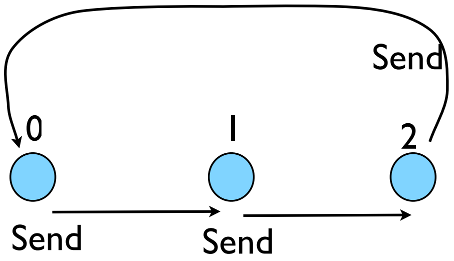
Implement periodic boundary conditions

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



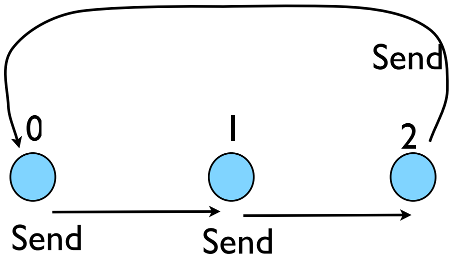
```fortran
left = rank-1
if (left < 0) left = comsize-1
right = rank+1
if (right >= comsize) right = 0

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



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**

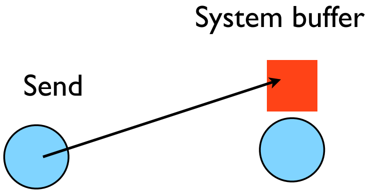
Big MPI
Lesson #1

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

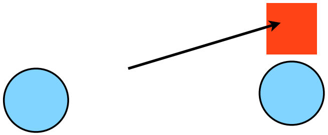
Different versions of SEND

- 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

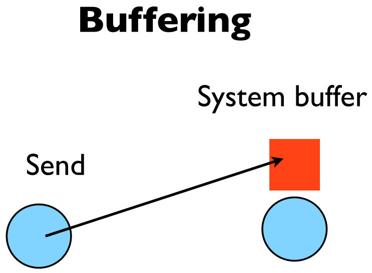


(Non) Blocking

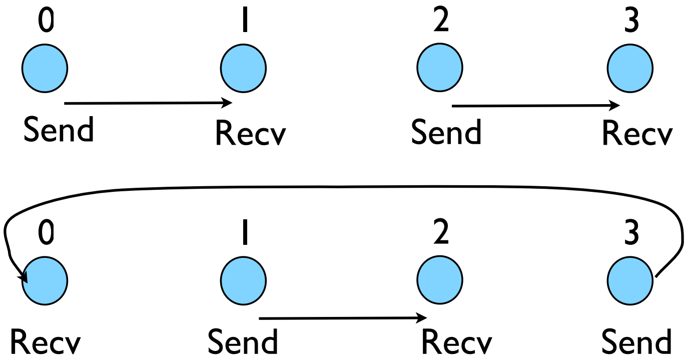


Buffering is dangerous!

- 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?

```
program fourthmessage
implicit none
include 'mpif.h'

    integer :: ierr, rank, comsize
    integer :: left, right
    integer :: tag
    integer :: status(MPI_STATUS_SIZE)
    double precision :: msgsent, msgrcvd

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

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

    msgsent = rank*rank
    msgrcvd = -999.
    tag = 1

    if (mod(rank,2) == 0) then
        call MPI_Ssend(msgsent, 1, MPI_DOUBLE_PRECISION, right, &
                       tag, MPI_COMM_WORLD, ierr)
        call MPI_Recv(msgrcvd, 1, MPI_DOUBLE_PRECISION, left, &
                       tag, MPI_COMM_WORLD, status, ierr)
    else
        call MPI_Recv(msgrcvd, 1, MPI_DOUBLE_PRECISION, left, &
                       tag, MPI_COMM_WORLD, status, ierr)
        call MPI_Ssend(msgsent, 1, MPI_DOUBLE_PRECISION, right, &
                       tag, MPI_COMM_WORLD, ierr)
    endif
    print *, rank, 'Sent ', msgsent, 'and recvd ', msgrcvd

    call MPI_FINALIZE(ierr)

end program fourthmessage
```

Evens send first

Then odds

fourthmessage.f90

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

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

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

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

    msgsent = rank*rank;
    msgrcvd = -999;

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

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

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

Evens send first

Then odds

SciNet

fourthmessage.c

Something new: Sendrecv

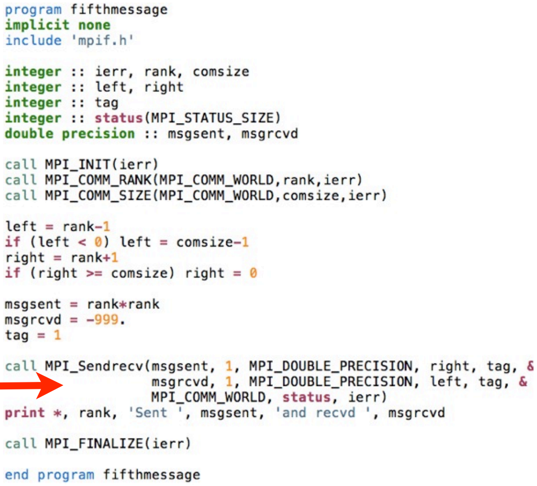
- A blocking send and receive built in together
- Lets them happen simultaneously
- Can automatically pair the sends/recvs!
- dest, source does not have to be same; nor do types or size.

```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

Something new: Sendrecv

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



```
program fifthmessage
implicit none
include 'mpif.h'

integer :: ierr, rank, comsize
integer :: left, right
integer :: tag
integer :: status(MPI_STATUS_SIZE)
double precision :: msgsent, msgrcvd

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

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

msgsent = rank*rank
msgrcvd = -999.
tag = 1

call MPI_Sendrecv(msgsent, 1, MPI_DOUBLE_PRECISION, right, tag, &
                  msgrcvd, 1, MPI_DOUBLE_PRECISION, left, tag, &
                  MPI_COMM_WORLD, status, ierr)
print *, rank, 'Sent ', msgsent, 'and recvd ', msgrcvd

call MPI_FINALIZE(ierr)

end program fifthmessage
```

fifthmessage.f90

Sendrecv = Send + Recv

C syntax

Send Args

```
MPI_Status status;

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

Recv Args

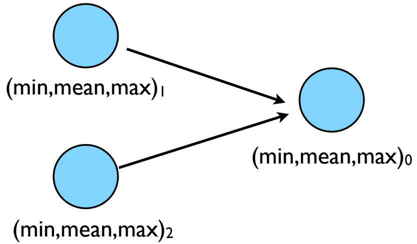
FORTRAN syntax

```
integer status(MPI_STATUS_SIZE)

call MPI_SENDRECV(sendptr, count, MPI_TYPE, destination,tag,
                     recvptr, count, MPI_TYPE, source, tag,
                     Communicator, status, ierr)
```

Why are there two different tags/types/counts?

Intro to Message Passing Interface (MPI)

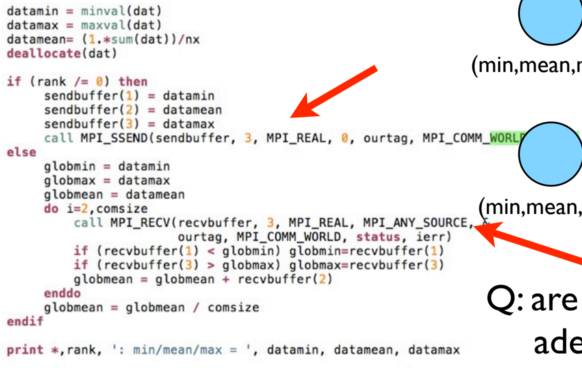
Min, Mean, Max of numbers



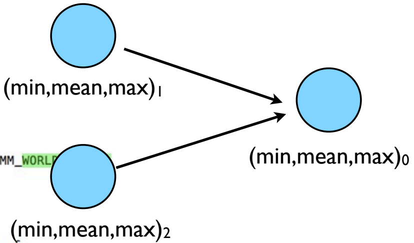
- Lets try some code that calculates the min/mean/max of a bunch of random numbers -1..1. Should go to -1,0,+1 for large N.
- Each gets their partial results and sends it to some node, say node 0 (why node 0?)
- ~/mpi/mpi-intro/minmeanmax.{c,f90}
- How to MPI it?

```fortran
      program randomdata
      implicit none
      integer,parameter :: nx=1500
      real, allocatable :: dat(:)

      integer :: i
      real    :: datamin, datamax, datamean

!
! random data
!
      allocate(dat(nx))
      call random_seed(put=[(i,i=1,8)])
      call random_number(dat)
      dat = 2*dat - 1.

!
! find min/mean/max
!
      datamin = minval(dat)
      datamax = maxval(dat)
      datamean= (1.*sum(dat))/nx

      deallocate(dat)

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

      return
      end
```

```
/*
 * generate random data
 */

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

/*
 * find min/mean/max
 */

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


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

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



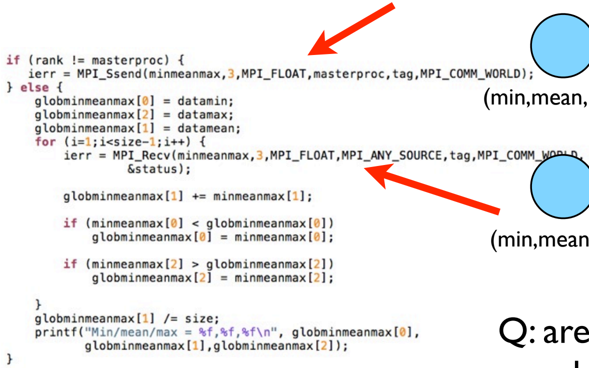
```fortran
datamin = minval(dat)
datamax = maxval(dat)
datamean= (1.*sum(dat))/nx
deallocate(dat)

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

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



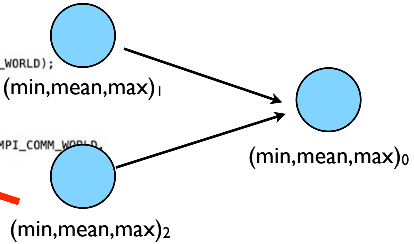
Q: are these sends/recvd adequately paired?

minmeanmax-mpi.f90

SciNet



```
if (rank != masterproc) {
   ierr = MPI_Ssend(minmeanmax,3,MPI_FLOAT,masterproc,tag,MPI_COMM_WORLD);
} else {
    globminmeanmax[0] = datamin;
    globminmeanmax[2] = datamax;
    globminmeanmax[1] = datamean;
    for (i=1;i<size-1;i++) {
        ierr = MPI_Recv(minmeanmax,3,MPI_FLOAT,MPI_ANY_SOURCE,tag,MPI_COMM_WORLD,
                 &status);

        globminmeanmax[1] += minmeanmax[1];

        if (minmeanmax[0] < globminmeanmax[0])
            globminmeanmax[0] = minmeanmax[0];

        if (minmeanmax[2] > globminmeanmax[2])
            globminmeanmax[2] = minmeanmax[2];

    }
    globminmeanmax[1] /= size;
    printf("Min/mean/max = %f,%f,%f\n", globminmeanmax[0],
           globminmeanmax[1],globminmeanmax[2]);
}
```



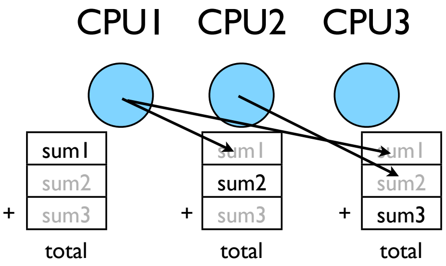
Q: are these sends/recvd adequately paired?

minmeanmax-mpi.c

SciNet

Inefficient!

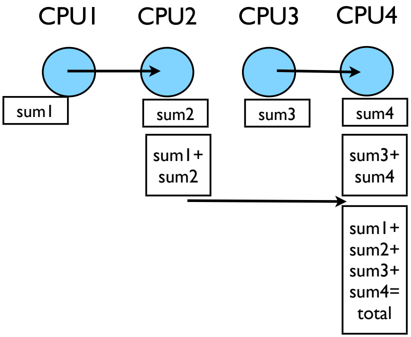
- Requires (P-1) messages, 2(P-1) if everyone then needs to get the answer.



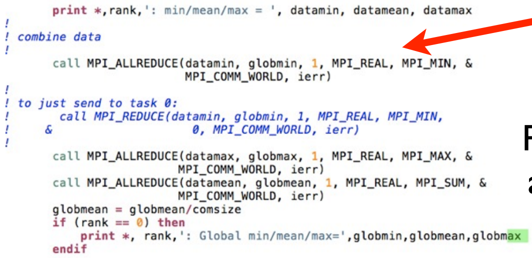
Better Summing

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

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



Reduction; works for a variety of operators (+,*,min,max...)



```
      print *,rank,': min/mean/max = ', datamin, datamean, datamax
!
! combine data
!
      call MPI_ALLREDUCE(datamin, globmin, 1, MPI_REAL, MPI_MIN, &
                         MPI_COMM_WORLD, ierr)
!
! to just send to task 0:
!       call MPI_REDUCE(datamin, globmin, 1, MPI_REAL, MPI_MIN,
!     &                           0, MPI_COMM_WORLD, ierr)
!
      call MPI_ALLREDUCE(datamax, globmax, 1, MPI_REAL, MPI_MAX, &
                        MPI_COMM_WORLD, ierr)
      call MPI_ALLREDUCE(datamean, globmean, 1, MPI_REAL, MPI_SUM, &
                        MPI_COMM_WORLD, ierr)
      globmean = globmean/comsize
      if (rank == 0) then
          print *, rank,': Global min/mean/max=',globmin,globmean,globmax
      endif
```

MPI_Reduce and MPI_Allreduce

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

minmeanmax-allreduce.f

SciNet

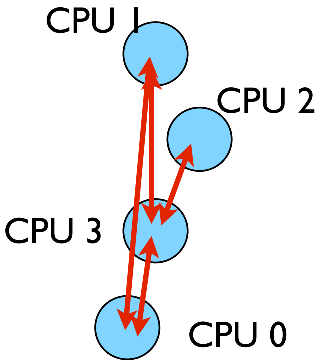
MPI Collectives

```
ierr = MPI_Allreduce(sendptr, rcvptr, count,
MPI_TYPE, MPI_OP, Communicator);
```

- `sendptr/rcvptr`: pointer to buffers
- `count`: number of elements in ptr
- `MPI_TYPE`: one of MPI_DOUBLE, MPI_FLOAT, MPI_INT, MPI_CHAR, etc.
- `MPI_OP`: one of MPI_SUM, MPI_PROD, MPI_MIN, MPI_MAX, etc.
- `Communicator`: MPI_COMM_WORLD or user created

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'



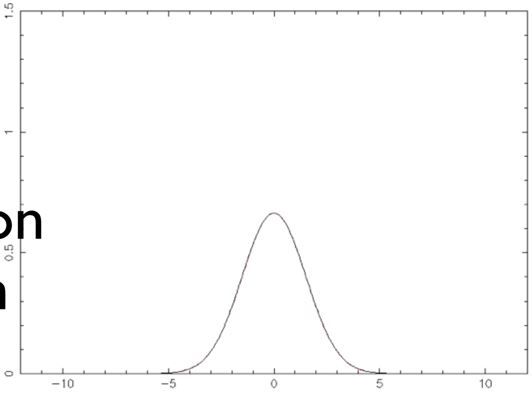
Intro to Message Passing Interface (MPI)

Scientific MPI Example

MPI “Real” problems

- Finite Difference Stencils
- Time-Marching Method
- Domain Decomposition
- Load Balancing
- Global Norms
- BC’s

1d diffusion equation

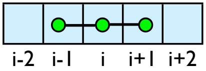


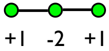
```
cd mpi/diffusion .
make diffusionf or make diffusionc
./diffusionf or ./diffusionc
```

Discretizing Derivatives

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

- Done by finite differencing the discretized values
- Implicitly or explicitly involves interpolating data and taking derivative of the interpolant
- More accuracy - larger ‘stencils’

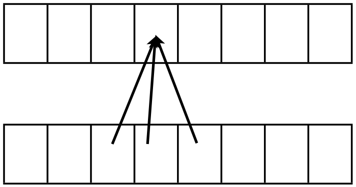




Diffusion Equation

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

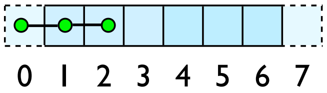
- Simple 1d PDE
- Each timestep, new data for T[i] requires old data for T[i+1],T[i],T[i-1]



Guardcells

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

Global Domain

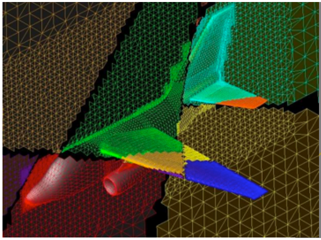


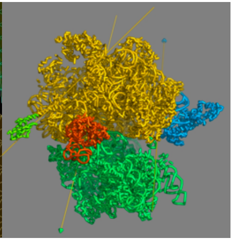
ng = 1

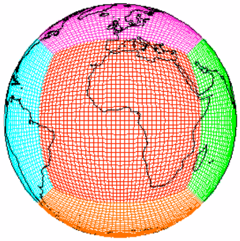
loop from ng, N - 2 ng

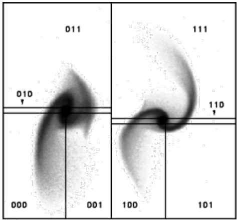
Domain Decomposition

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


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


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


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



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

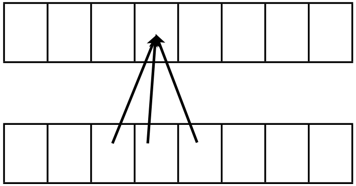
SciNet

Implement a diffusion equation in MPI

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

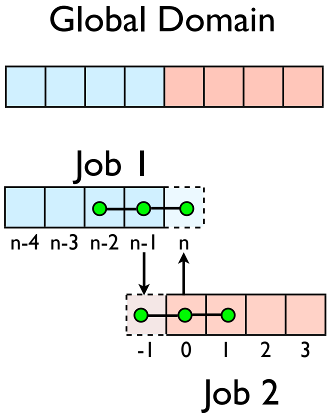
$$T_i^{n+1} = T_i^n + \frac{D\Delta t}{\Delta x^2}\left(T_{i+1}^n - 2T_i^n + T_{i-1}^n\right)$$

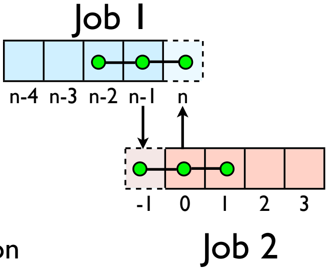
- Need one neighboring number per neighbor per timestep



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

SciNet

Hands-on: MPI diffusion

- cp diffusionf.f90 diffusionf-mpi.f90 or
- cp diffusionc.c diffusionc-mpi.c or
- Make an MPI-ed version of diffusion equation
- (Build: `make diffusionf-mpi` or `make diffusionc-mpi`)
- Test on 1..8 procs

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

C syntax

```
MPI_Status status;

ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_{size,rank}(Communicator, &{size,rank});
ierr = MPI_Send(sendptr, count, MPI_TYPE, destination,
                 tag, Communicator);
ierr = MPI_Recv(rcvptr, count, MPI_TYPE, source, tag,
                 Communicator, &status);
ierr = MPI_Sendrecv(sendptr, count, MPI_TYPE, destination,tag,
                    recvptr, count, MPI_TYPE, source, tag,
                    Communicator, &status);
ierr = MPI_Allreduce(&mydata, &globaldata, count, MPI_TYPE,
                     MPI_OP, Communicator);

Communicator -> MPI_COMM_WORLD
MPI_Type -> MPI_FLOAT, MPI_DOUBLE, MPI_INT, MPI_CHAR...
MPI_OP -> MPI_SUM, MPI_MIN, MPI_MAX,...
```

FORTRAN syntax

```
integer status(MPI_STATUS_SIZE)

call MPI_INIT(ierr)
call MPI_COMM_{SIZE,RANK}(Communicator, {size,rank},ierr)
call MPI_SSEND(sendarr, count, MPI_TYPE, destination,
                 tag, Communicator)
call MPI_RECV(rcvarr, count, MPI_TYPE, destination,tag,
                 Communicator, status, ierr)
call MPI_SENDRECV(sendptr, count, MPI_TYPE, destination,tag,
                    recvptr, count, MPI_TYPE, source, tag,
                    Communicator, status, ierr)
call MPI_ALLREDUCE(&mydata, &globaldata, count, MPI_TYPE,
                     MPI_OP, Communicator, ierr)


Communicator -> MPI_COMM_WORLD
MPI_Type -> MPI_REAL, MPI_DOUBLE_PRECISION,
           MPI_INTEGER, MPI_CHARACTER
MPI_OP -> MPI_SUM, MPI_MIN, MPI_MAX,...
```

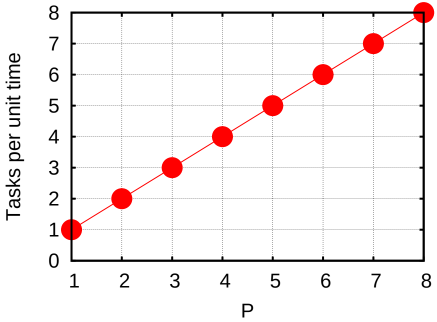
Intro to Message Passing Interface (MPI)

Scaling — Throughput

- How a problem's throughput scales as processor number increases ("strong scaling").
- In this case, linear scaling:

$$\mathbf{H} \propto \mathbf{P}$$

- This is Perfect scaling.

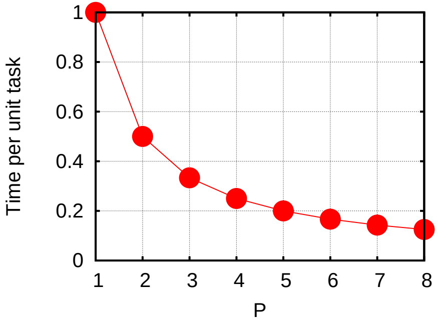


Scaling – Time

- How a problem's timing scales as processor number increases.
- Measured by the time to do one unit. In this case, inverse linear scaling:

$$\mathbf{T \propto 1/P}$$

- Again this is the ideal case, or "embarrassingly parallel".

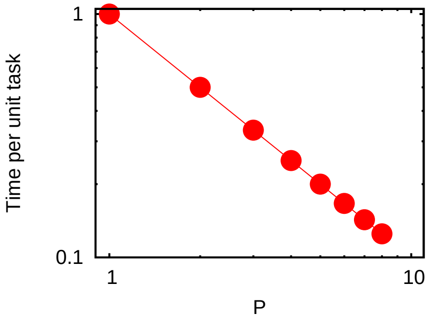


Scaling – Time

- How a problem's timing scales as processor number increases.
- Measured by the time to do one unit. In this case, inverse linear scaling:

$$\mathbf{T \propto 1/P}$$

- Again this is the ideal case, or "embarrassingly parallel".

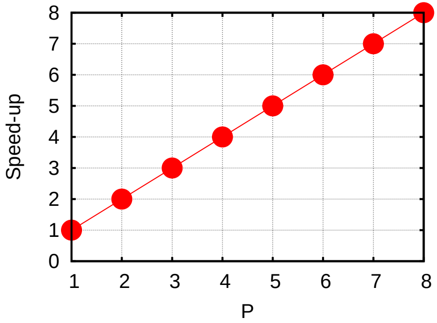


Scaling – Speedup

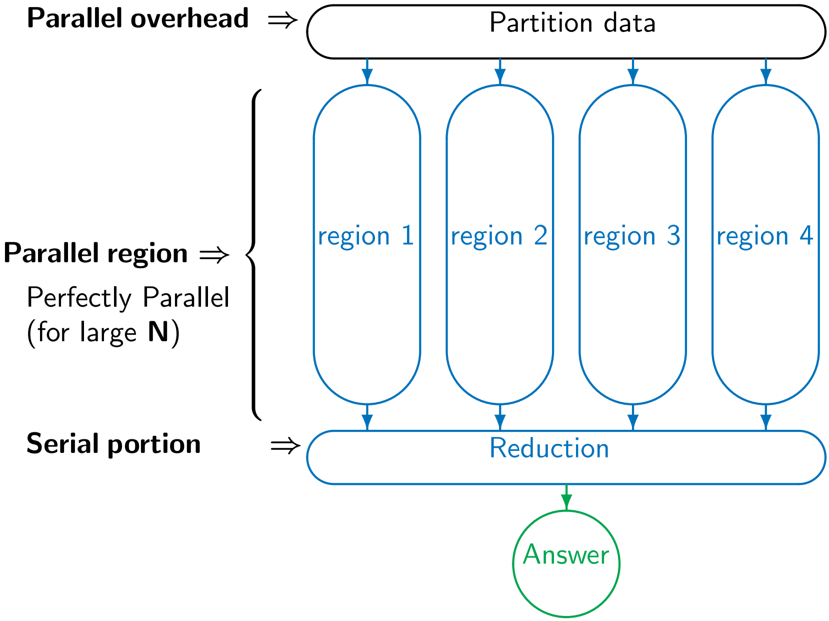
- How much faster the problem is solved as processor number increases.
- Measured by the serial time divided by the parallel time

$$\mathbf{S = \frac{T_{serial}}{T(P)} \propto P}$$

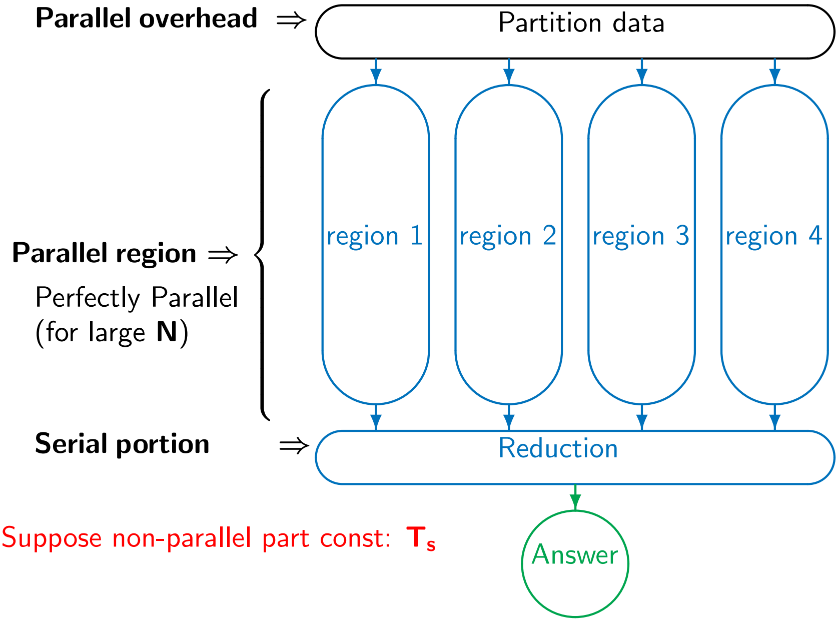
- For embarrassingly parallel applications: Linear speed up.



Serial Overhead



Serial Overhead



Suppose non-parallel part const: $\mathbf{T_s}$

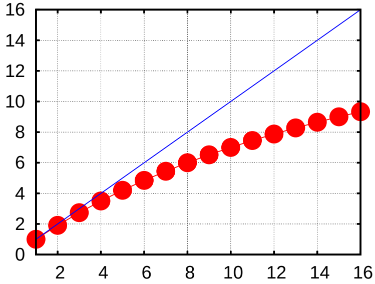
Amdahl's law

Speed-up (without parallel overhead):

$$\mathbf{S = \frac{NT_1 + T_s}{\frac{NT_1}{P} + T_s}}$$

or, calling $\mathbf{f = T_s/(T_s + NT_1)}$ the serial fraction,

$$\mathbf{S = \frac{1}{f + (1-f)/P}}$$



(for $\mathbf{f = 5\%}$)

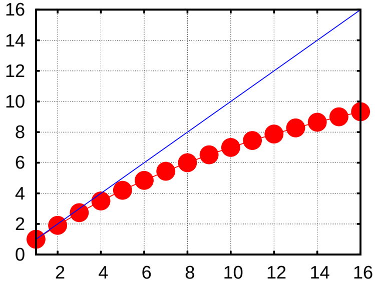
Amdahl's law

Speed-up (without parallel overhead):

$$\mathbf{S} = \frac{\mathbf{NT_1 + T_s}}{\frac{\mathbf{NT_1}}{\mathbf{P}} + \mathbf{T_s}}$$

or, calling $\mathbf{f = T_s/(T_s + NT_1)}$ the serial fraction,

$$\mathbf{S} = \frac{\mathbf{1}}{\mathbf{f + (1 - f)/P}} \quad \xrightarrow{\mathbf{P \to \infty}} \quad \frac{\mathbf{1}}{\mathbf{f}}$$



Serial part dominates asymptotically.

Speed-up limited, no matter size of **P**.

And this is the overly optimistic case!

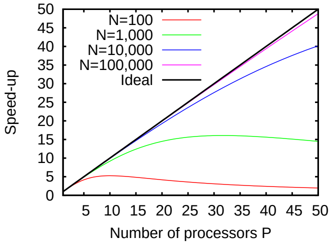
(for $\mathbf{f = 5\%}$)

Trying to beat Amdahl's law

Scale up!

The larger **N**, the smaller the serial fraction:

$$\mathbf{f(P) = \frac{P}{N}}$$

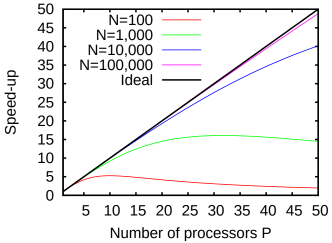


Trying to beat Amdahl's law

Scale up!

The larger **N**, the smaller the serial fraction:

$$\mathbf{f(P) = \frac{P}{N}}$$



Weak scaling: Increase problem size while increasing **P**

$$\mathbf{Time_{weak}(P) = Time(N = n \times P, P)}$$

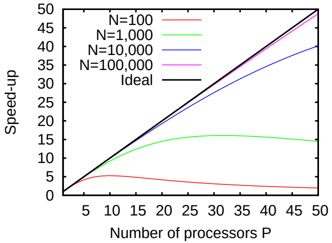
Good weak scaling means this time approaches a constant for large **P**.

Trying to beat Amdahl's law

Scale up!

The larger **N**, the smaller the serial fraction:

$$\mathbf{f(P) = \frac{P}{N}}$$



Weak scaling: Increase problem size while increasing **P**

$$\mathbf{Time_{weak}(P) = Time(N = n \times P, P)}$$

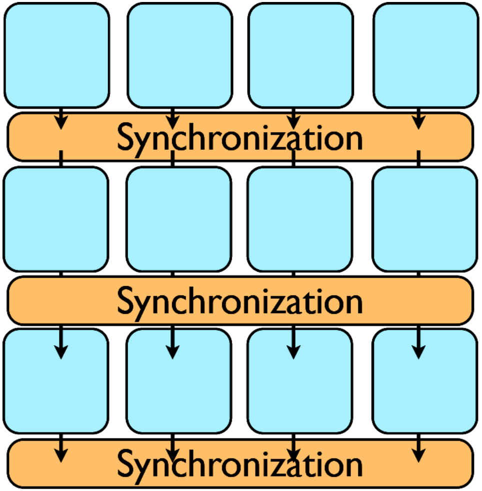
Good weak scaling means this time approaches a constant for large **P**.

Gustafson's Law

Any large enough problem can be efficiently parallelized (Efficiency$\longrightarrow$1).

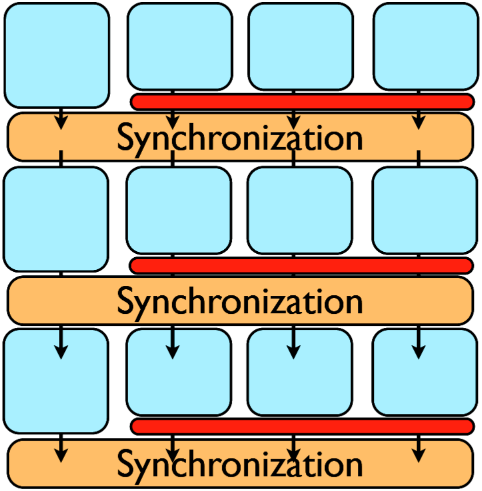
Synchronization Overhead

- Most problems are not purely concurrent.
- Some level of synchronization or exchange of information is needed between tasks.
- While synchronizing, nothing else happens: increases Amdahl's **f**.
- And synchronizations are themselves costly.



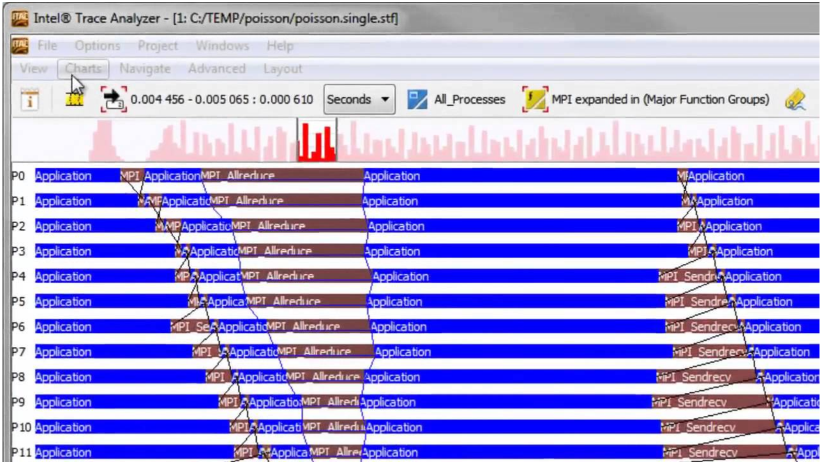
Load Balancing

- The division of calculations among the processors may not be equal.
- Some processors would already be done, while others are still going.
- Effectively using less than **P** processors: This reduces the efficiency.
- Aim for load balanced algorithms.



Intro to Message Passing Interface (MPI)

MPI: Blocking


Intel® Trace Analyzer - [1: C:/TEMP/poisson/poisson.single.stf]

0.004 456 - 0.005 065 : 0.000 610 Seconds All_Processes MPI expanded in (Major Function Groups)

Message Passing Interface (MPI)

Non-Blocking Communications

- Mechanism for overlapping/interleaving communications and useful computations
- Avoid deadlocks
- Can avoid system buffering, memory-to-memory copying and improve performance

Message Passing Interface (MPI)

Non-Blocking Communications

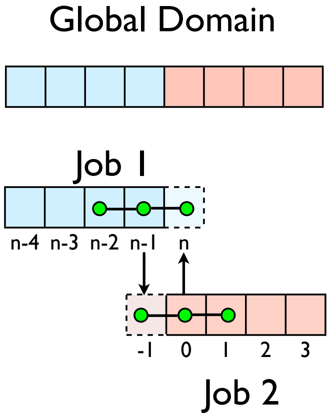
- Mechanism for overlapping/interleaving communications and useful computations
- Avoid deadlocks
- Can avoid system buffering, memory-to-memory copying and improve performance

Non-Blocking: MPI_Isend, MPI_Irecv

- Returns immediately, posting request to system to initiate communication.
- However, communication is not completed yet.
- Cannot tamper with the memory provided in these calls until the communication is completed.

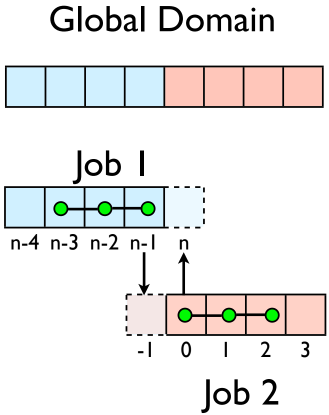
Diffusion: Had to wait for communications to compute

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



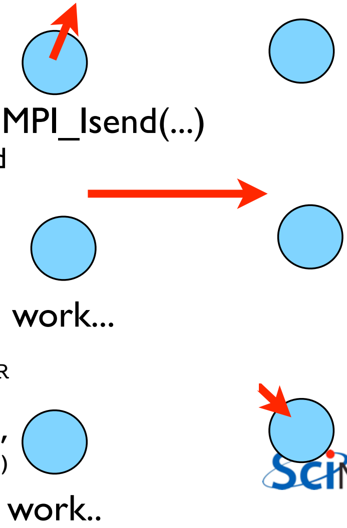
Diffusion: *Had* to wait?

- But inner zones could have been computed just fine
- Ideally, would do inner zones work while communications is being done; then go back and do end points.



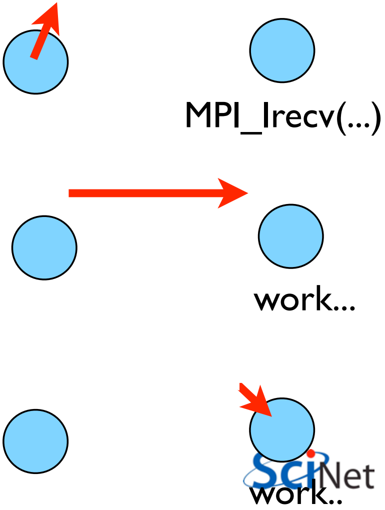
Nonblocking Sends

- Allows you to get work done while message is 'in flight'
- Must **not** alter send buffer until send has completed.
- C: `MPI_Isend( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request )` (**MPI_Request *request** in bold)
- FORTRAN: `MPI_ISEND(BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG, INTEGER COMM, INTEGER REQUEST,INTEGER IERROR)` (**INTEGER REQUEST** in bold)



Nonblocking Recv

- Allows you to get work done while message is 'in flight'
- Must **not** access recv buffer until recv has completed.
- C: `MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, ` **`MPI_Request *request`** ` )`
- FORTRAN: `MPI_IREV(BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER SOURCE,INTEGER TAG, INTEGER COMM, ` **`INTEGER REQUEST`**`,INTEGER IERROR)`



MPI: Non-Blocking Isend & Irecv

```
ierr = MPI_Isend(sendptr, count, MPI_TYPE,
destination,tag, Communicator, MPI_Request)
ierr = MPI_Irecv(rcvptr, count, MPI_TYPE,
source, tag,Communicator, MPI_Request)
```

- `sendptr/rcvptr`: pointer to message
- `count`: number of elements in ptr
- `MPI_TYPE`: one of MPI_DOUBLE, MPI_FLOAT, MPI_INT, MPI_CHAR, etc.
- `destination/source`: rank of sender/reciever
- `tag`: unique id for message pair
- `Communicator`: MPI_COMM_WORLD or user created
- `MPI_Request`: Identify comm operations

How to tell if message is completed?

- `int MPI_Wait(MPI_Request *request,MPI_Status *status);`
- `MPI_WAIT(INTEGER REQUEST,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER IERROR)`

- `int MPI_Waitall(int count,MPI_Request *array_of_requests, MPI_Status *array_of_statuses);`
- `MPI_WAITALL(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),INTEGER`

Also: MPI_Waitany, MPI_Test...

MPI: Wait & Waitall

- Will block until the communication(s) complete

```
ierr = MPI_Wait(MPI_Request *, MPI_Status *)
ierr = MPI_Waitall(count, MPI_Request *, MPI_Status
*)
```

- `MPI_Request`: Identify comm operation(s)
- `MPI_Status`: Status of comm operation(s)
- `count`: Number of comm operations(s)

MPI: Test

- Does not block, returns immediately
- Provides a mechanism for overlapping communication and computation

```
ierr = MPI_Test(MPI_Request *, flag, MPI_Status *)
```

- **`MPI_Request`**: Identify comm operation(s)
- **`MPI_Status`**: Status of comm operation(s)
- **`flag`**: true if comm complete; false if not sent/recv yet

Hands On

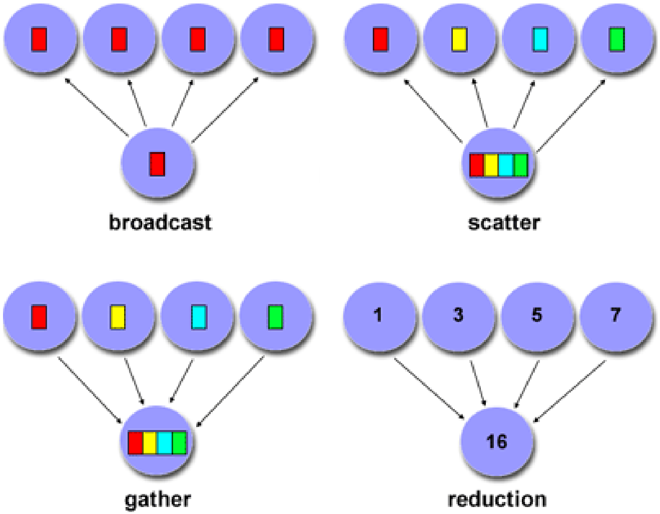
- In diffusion directory, cp diffusion{c,f}-mpi.{c,f90} to diffusion{c,f}-mpi-nonblocking.{c,f90}
- Change to do non-blocking IO; post sends/recvs, do inner work, wait for messages to clear, do end points

Intro to Message Passing Interface (MPI)

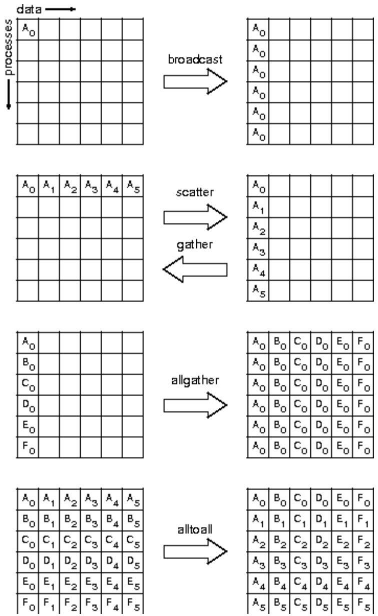
MPI Collectives

- All processes in a group participate in communication, by calling the same function with matching arguments.
- Types:
  - Synchronization: MPI_Barrier
  - Data Movement: MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Alltoall
  - Collective Computation: MPI_Allreduce
- Collective routines are blocking

MPI Collectives



MPI Collectives



MPI Collectives: Broadcast

- Broadcasts a message from process with rank “root” to all processes in group, including itself.
- Amount of data sent must be equal to amount of data received.

```
ierr = MPI_Bcast(void *buf, count, MPI_Type, root,
Comm)
```

- **buf**: buffer of data to send/recv
- **count**: number of elements in buf
- **MPI_TYPE**: one of MPI_DOUBLE, MPI_FLOAT, MPI_INT, MPI_CHAR, etc.
- **root**: “root” processor to send from
- **Communicator**: MPI_COMM_WORLD or user created

MPI Collectives: Scatter/Gather

- Scatter: Sends data from "root" to all processes in group.
- Gather: Recives data on "root" from all processes in group.

```
ierr = MPI_Scatter(void *send_buf, send_count,
MPI_Type, void *recv_buf, recv_count, MPI_Type,
root, Comm)
ierr = MPI_Gather(void *send_buf, send_count,
MPI_Type, void *recv_buf, recv_count, MPI_Type,
root, Comm)
```

- **`send_buf`**: buffer of data to send
- **`send_count`**: number of elements in send_buf
- **`MPI_TYPE`**: one of MPI_DOUBLE, MPI_INT, MPI_CHAR, etc.
- **`recv_buf`**: buffer of data to recv
- **`recv_count`**: number of elements in recv_buf
- **`root`**: "root" processor to send from
- **`Communicator`**: MPI_COMM_WORLD or user created

Example: Scatter/Gather

Scatter

- Simple Scatter example sending data from root to 4 procesors.

```
$cd mpi/collectives
$make
$mpirun -np 4 ./scatter
```

Example: Scatter/Gather

Scatter

- Simple Scatter example sending data from root to 4 procesors.

```
$cd mpi/collectives
$make
$mpirun -np 4 ./scatter
```

Gather

- Copy Scatter.c to Gather.c and reverse the process.
- Send from 4 processes and collect on root using MPI_Gather()

MPI Collectives: Barrier

- Blocks calling process until all group members have called it.
- Decreases performance. Try to avoid using it explicitly.

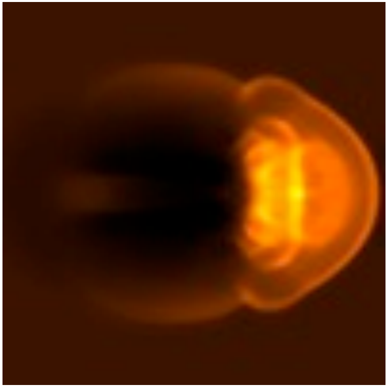
```
ierr = MPI_Barrier(Comm)
```

- **Communicator**: MPI_COMM_WORLD or user created

Intro to Message Passing Interface (MPI)

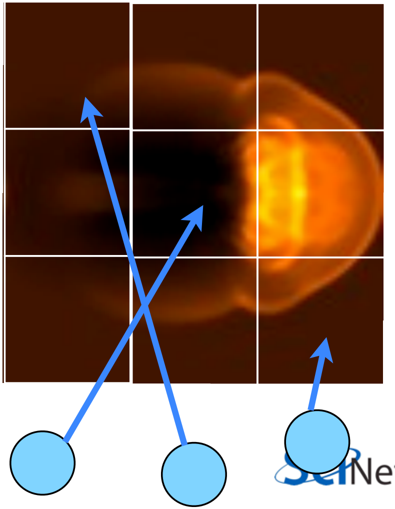
MPI-IO

- Would like the new, parallel version to still be able to write out single output files.
- But at no point does a single processor have entire domain...



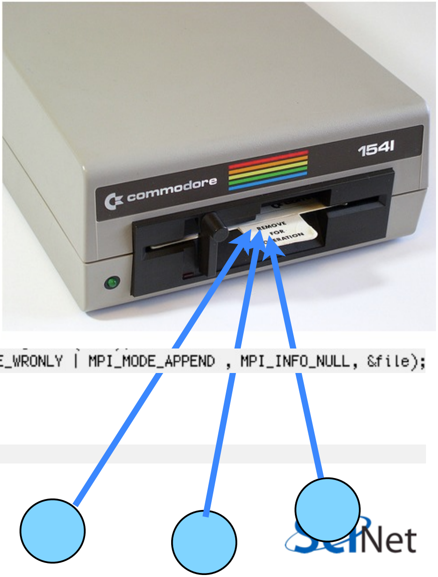
Parallel I/O

- Each processor has to write its own piece of the domain..
- without overwriting the other.
- Easier if there is global coordination



MPI-IO

- Uses MPI to coordinate reading/writing to single file



```
ierr = MPI_File_open(MPI_COMM_WORLD,filename, MPI_MODE_WRONLY | MPI_MODE_APPEND , MPI_INFO_NULL, &file);
```

...stuff...

```
ierr = MPI_File_close(&file);
```

- Coordination -- *collective* operations.

MPI-IO: Example

```
{
  ...
  MPI_Offset offset = (msgsize*rank);
  MPI_File file;
  MPI_Status stat;

  MPI_File_open(MPI_COMM_WORLD, "helloworld.txt",
  MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL,
  &file);

  MPI_File_seek(file, offset, MPI_SEEK_SET);
  MPI_File_write(file, msg, msgsize, MPI_CHAR,
  &stat);
  MPI_File_close(&file);
  ...
}
```

MPI-IO: Example

```
{
  ...
  MPI_Offset offset = (msgsize*rank);
  MPI_File file;
  MPI_Status stat;

  MPI_File_open(MPI_COMM_WORLD, "helloworld.txt",
  MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL,
  &file);

  //Collective Coordinated Write
  MPI_File_write_at_all(file, offset, msg, msg-
  size, MPI_CHAR, &stat);
  MPI_File_close(&file);
  ...
}
```

MPI-IO: MPI_File_open

- MPI_File_open

```
ierr = MPI_File_open(communicator, filename, mode,
MPI_Info, MPI_File);
ierr = MPI_File_close(MPI_File);
```

- `communicator`: MPI_COMM_WORLD or user created
- `char * filename`: character string filename
- `int mode`: Access modes, MPI_MODE_CREATE, MPI_MODE_WRONLY, MPI_MODE_RDWR, etc.
- `MPI_Info`: extra info or MPI_INFO_NULL
- `MPI_File`: MPI file handle

MPI-IO: MPI_File_write_at_all

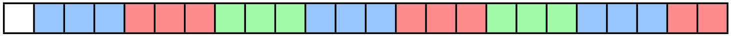
- Collective operation across all Comm processors

```
ierr = MPI_File_write_at_all(MPI_File,
MPI_Offset,buffer, count, MPI_Type, MPI_Status)
```

- `MPI_File`: MPI file handle
- `MPI_Offset`: MPI file offset location
- `void * buffer`: buffer of data to write
- `int count`: number of elements in ptr
- `MPI_TYPE`: one of MPI_FLOAT, MPI_INT, MPI_CHAR, etc.
- `MPI_Request`: Identify comm operations

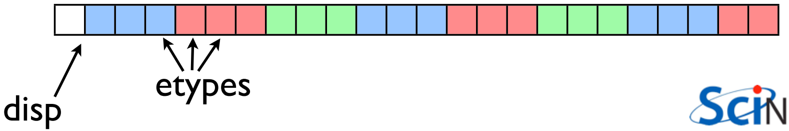
MPI-IO File View

- Each process has a view of the file that consists of only of the parts accessible to it.
- For writing, hopefully non-overlapping!
- Describing this - how data is laid out in a file - is very similar to describing how data is laid out in memory...



MPI-IO File View

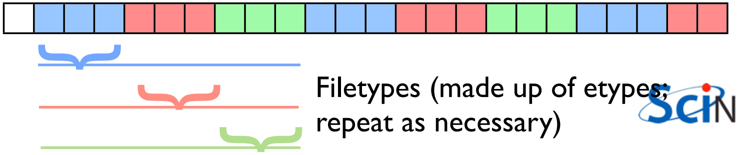
- int MPI_File_set_view(
    MPI_File fh,
    MPI_Offset disp, /* displacement in *bytes* from start */
    MPI_Datatype etype, /* elementary type */
    MPI_Datatype filetype, /* file type; prob different for each proc */
    char *datarep, /* 'native' or 'internal' */
    MPI_Info info) /* MPI_INFO_NULL for today */



MPI-IO File View

- int MPI_File_set_view(

```
    MPI_File fh,
    MPI_Offset disp,           /* displacement in bytes from start */
    MPI_Datatype etype,        /* elementary type */
    MPI_Datatype filetype,     /* file type; prob different for each proc */
    char *datarep,             /* 'native' or 'internal' */
    MPI_Info info)             /* MPI_INFO_NULL */
```



Filetypes (made up of etypes; repeat as necessary)

MPI-IO File Write

- int MPI_File_write_all(
  MPI_File fh,
  void *buf,
  int count,
  MPI_Datatype datatype,
  MPI_Status *status)

Writes (_all: collectively) to part of file within view.

Example: MPI-IO

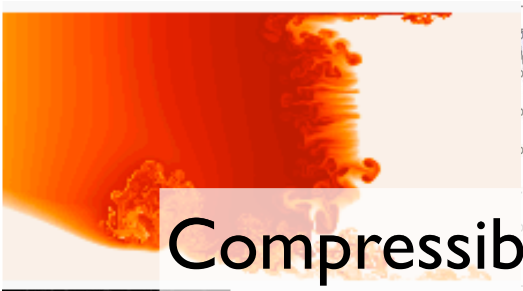
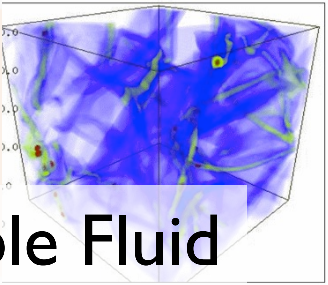
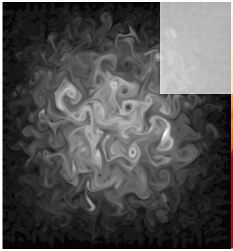
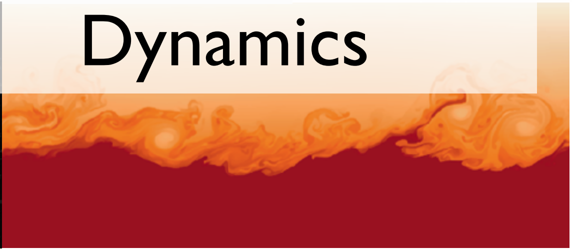
MPI-IO Example

- Simple Example showing MPI writing to a single file.

```
$cd mpi/mpiio
$make
$mpirun -np 4 ./sine
$./dosineplot
```

Anything wrong with this code?

Intro to Message Passing Interface (MPI)

Compressible Fluid Dynamics

Equations of Hydrodynamics

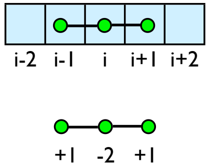
- Density, momentum, and energy equations
- Supplemented by an equation of state - pressure as a function of dens, energy

$$
\begin{aligned}
\frac{\partial}{\partial t}\rho + \nabla \cdot (\rho \mathbf{v}) &= 0 \\
\frac{\partial}{\partial t}(\rho \mathbf{v}) + \nabla \cdot (\rho \mathbf{v}\mathbf{v}) &= -\nabla p \\
\frac{\partial}{\partial t}(\rho E) + \nabla \cdot ((\rho E + p)\, \mathbf{v}) &= 0
\end{aligned}
$$

Discretizing Derivatives

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

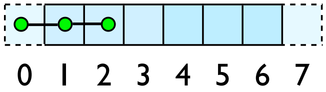
- Done by finite differencing the discretized values
- Implicitly or explicitly involves interpolating data and taking derivative of the interpolant
- More accuracy - larger 'stencils'



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 0th point in domain
- Fill guard cells with values such that the required boundary conditions are met

Global Domain

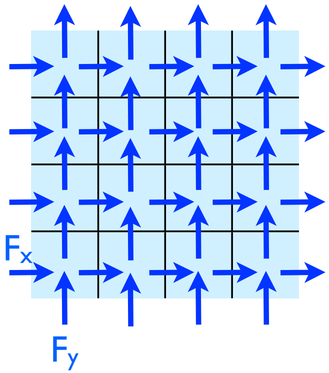


ng = 1

loop from ng, N - 2 ng

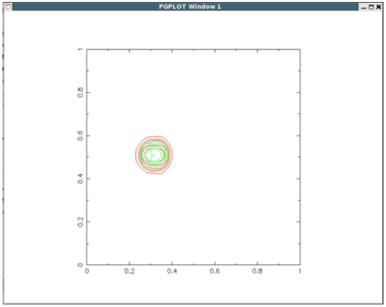
Finite Volume Method

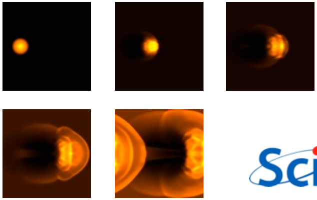
- Conservative; very well suited to high-speed flows with shocks
- At each timestep, calculate fluxes using interpolation/finite differences, and update cell quantities.
- Use conserved variables -- eg, momentum, not velocity.



Single-Processor hydro code

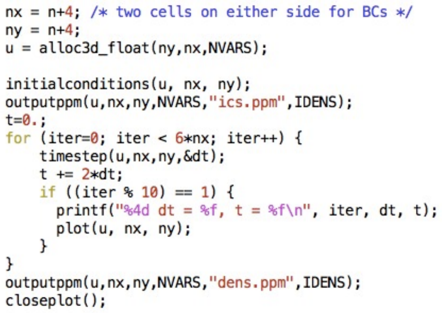
- `cd hydro{c,f};  make`
- `./hydro 100`
- Takes options:
  - number of points to write
- Outputs image (ppm) of initial conditions, final state (plots density)
- display ics.ppm
- display dens.ppm





Single-Processor hydro code

- Set initial conditions
- Loop, calling *timestep()* and maybe some output routines (*plot()* - contours)
- At beginning and end, save an image file with *outputppm()*
- All data stored in array *u*.

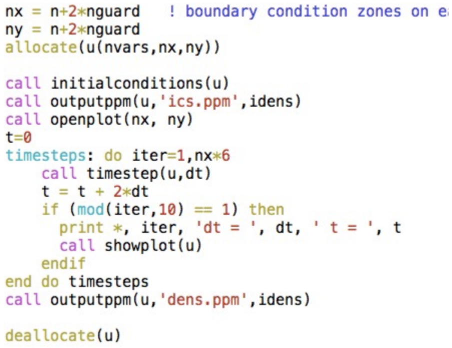


```
nx = n+4; /* two cells on either side for BCs */
ny = n+4;
u = alloc3d_float(ny,nx,NVARS);

initialconditions(u, nx, ny);
outputppm(u,nx,ny,NVARS,"ics.ppm",IDENS);
t=0.;
for (iter=0; iter < 6*nx; iter++) {
    timestep(u,nx,ny,&dt);
    t += 2*dt;
    if ((iter % 10) == 1) {
      printf("%4d dt = %f, t = %f\n", iter, dt, t);
      plot(u, nx, ny);
    }
}
outputppm(u,nx,ny,NVARS,"dens.ppm",IDENS);
closeplot();
```

hydro.c

Single-Processor hydro code

- Set initial conditions
- Loop, calling *timestep()* and maybe some output routines (*plot()* - contours)
- At beginning and end, save an image file with *outputppm()*
- All data stored in array *u*.

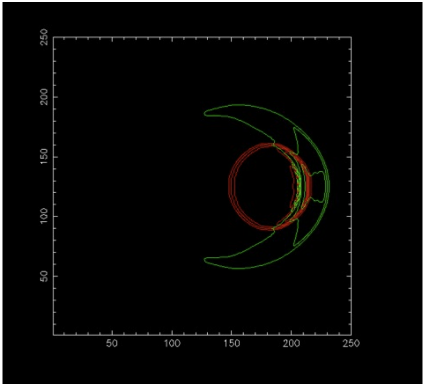


```
nx = n+2*nguard    ! boundary condition zones on e
ny = n+2*nguard
allocate(u(nvars,nx,ny))

call initialconditions(u)
call outputppm(u,'ics.ppm',idens)
call openplot(nx, ny)
t=0
timesteps: do iter=1,nx*6
    call timestep(u,dt)
    t = t + 2*dt
    if (mod(iter,10) == 1) then
      print *, iter, 'dt = ', dt, ' t = ', t
      call showplot(u)
    endif
end do timesteps
call outputppm(u,'dens.ppm',idens)

deallocate(u)
```

hydro.f90 SciNet

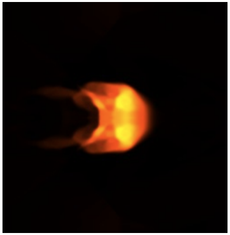
Plotting to screen

- plot.c, plot.f90
- Every 10 timesteps
- Find min, max of pressure, density
- Plot 5 contours of density (red) and pressure (green)
- pgplot library (old, but works).



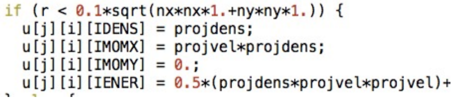
Plotting to file

- ppm.c, ppm.f90
- PPM format -- binary (w/ ascii header)
- Find min, max of density
- Calculate r,g,b values for scaled density (black = min, yellow = max)
- Write header, then data.



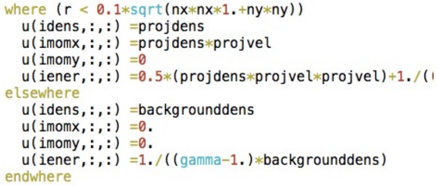
Data structure

- *u* : 3 dimensional array containing each variable in 2d space
- eg, u[j][i][IDENS]
- or u(idens, i, j)



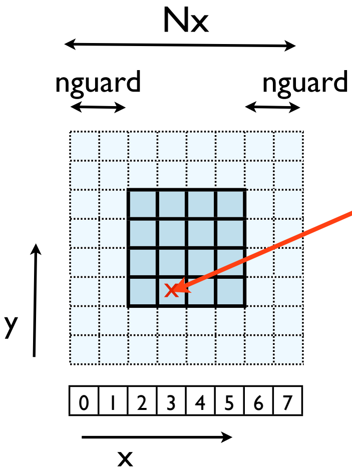
```
if (r < 0.1*sqrt(nx*nx*1.+ny*ny*1.)) {
  u[j][i][IDENS] = projdens;
  u[j][i][IMOMX] = projvel*projdens;
  u[j][i][IMOMY] = 0.;
  u[j][i][IENER] = 0.5*(projdens*projvel*projvel)+
```

solver.c (initialconditions)



```
where (r < 0.1*sqrt(nx*nx*1.+ny*ny))
  u(idens,:,:) =projdens
  u(imomx,:,:) =projdens*projvel
  u(imomy,:,:) =0
  u(iener,:,:) =0.5*(projdens*projvel*projvel)+1./(
elsewhere
  u(idens,:,:) =backgrounddens
  u(imomx,:,:) =0.
  u(imomy,:,:) =0.
  u(iener,:,:) =1./((gamma-1.)*backgrounddens)
endwhere
```

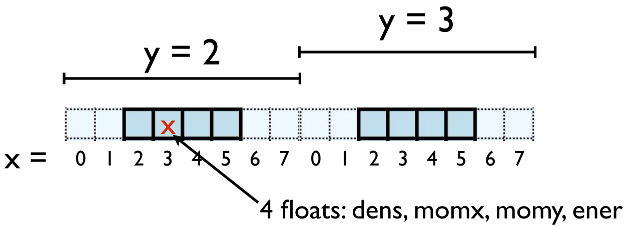
solver.f90 (initialconditions)



```
u[2][3][DENSVAR];
u[2][3][MOMXVAR];
u[2][3][MOMYVAR];
u[2][3][ENERVAR];
```

```
u(idens,4,3)
u(imomx,4,3)
u(imomy,4,3)
u(iener,4,3)
```

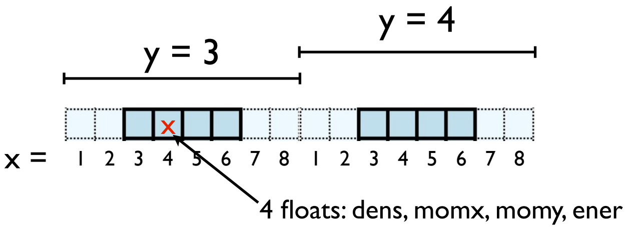
Laid out in memory (C)



y = 2
y = 3
x = 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
x
4 floats: dens, momx, momy, ener

Same way as in an image file
(one horizontal row at a time)

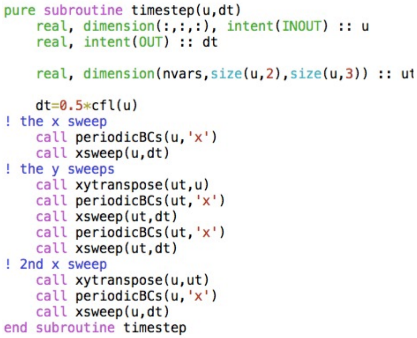
Laid out in memory (FORTRAN)



Same way as in an image file
(one horizontal row at a time)

Timestep routine

- Apply boundary conditions
- X sweep, Y sweep
- Transpose entire domain , so Y sweep is just an X sweep
- (unusual approach! But has advantages. Like matrix multiply.)
- Note - dt calculated each step (minimum across domain.)

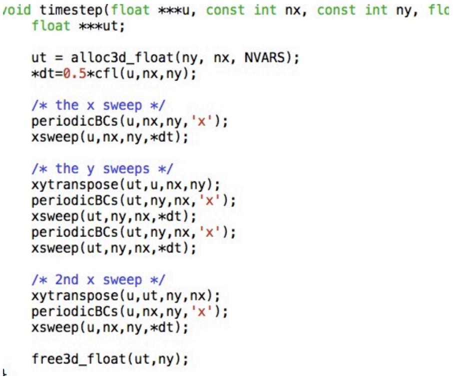


```fortran
pure subroutine timestep(u,dt)
    real, dimension(:,:,:), intent(INOUT) :: u
    real, intent(OUT) :: dt

    real, dimension(nvars,size(u,2),size(u,3)) :: ut

    dt=0.5*cfl(u)
! the x sweep
    call periodicBCs(u,'x')
    call xsweep(u,dt)
! the y sweeps
    call xytranspose(ut,u)
    call periodicBCs(ut,'x')
    call xsweep(ut,dt)
    call periodicBCs(ut,'x')
    call xsweep(ut,dt)
! 2nd x sweep
    call xytranspose(u,ut)
    call periodicBCs(u,'x')
    call xsweep(u,dt)
end subroutine timestep
```

timestep
solver.f90

Timestep routine

- Apply boundary conditions
- X sweep, Y sweep
- Transpose entire domain , so Y sweep is just an X sweep
- (unusual approach! But has advantages. Like matrix multiply.)
- Note - dt calculated each step (minimum across domain.)



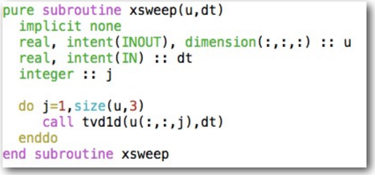
```
oid timestep(float ***u, const int nx, const int ny, flo
    float ***ut;

    ut = alloc3d_float(ny, nx, NVARS);
    *dt=0.5*cfl(u,nx,ny);

    /* the x sweep */
    periodicBCs(u,nx,ny,'x');
    xsweep(u,nx,ny,*dt);

    /* the y sweeps */
    xytranspose(ut,u,nx,ny);
    periodicBCs(ut,ny,nx,'x');
    xsweep(ut,ny,nx,*dt);
    periodicBCs(ut,ny,nx,'x');
    xsweep(ut,ny,nx,*dt);

    /* 2nd x sweep */
    xytranspose(u,ut,ny,nx);
    periodicBCs(u,nx,ny,'x');
    xsweep(u,nx,ny,*dt);

    free3d_float(ut,ny);
```

timestep
solver.c

SciNet

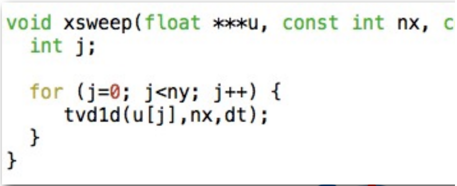
Xsweep routine

- Go through each x "pencil" of cells
- Do 1d hydrodynamics routine on that pencil.



```
pure subroutine xsweep(u,dt)
  implicit none
  real, intent(INOUT), dimension(:,:,:) :: u
  real, intent(IN) :: dt
  integer :: j

  do j=1,size(u,3)
     call tvd1d(u(:,:,j),dt)
  enddo
end subroutine xsweep
```

xsweep
solver.f90



```
void xsweep(float ***u, const int nx, c
  int j;

  for (j=0; j<ny; j++) {
     tvd1d(u[j],nx,dt);
  }
}
```

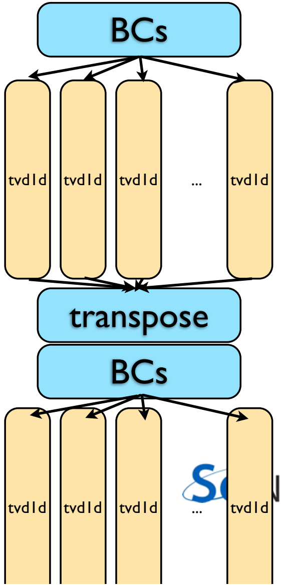
xsweep
solver.c

SciNet

What do data dependancies look like for this?

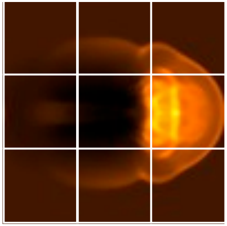
Data dependencies

- Previous timestep must be completed before next one started.
- Within each timestep,
- Each tvd1d "pencil" can be done independently
- All must be done before transpose, BCs



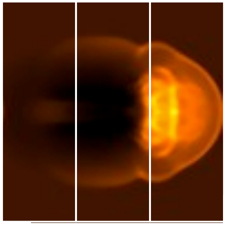
MPIing the code

- Domain decomposition



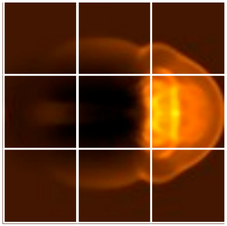
MPIing the code

- Domain decomposition
- For simplicity, for now we'll just implement decomposition in one direction, but we will design for full 2d decomposition



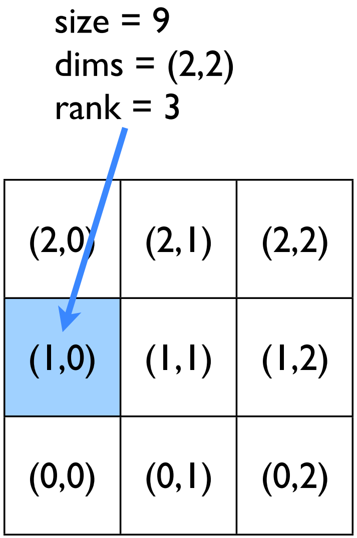
MPIing the code

- Domain decomposition
- We can do as with diffusion and figure out out neighbours by hand, but MPI has a better way...



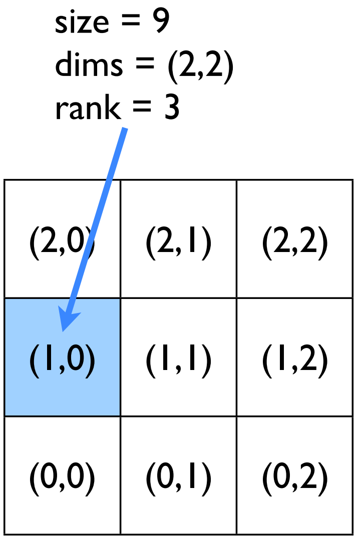
Create new communicator with new topology

- MPI_Cart_create (MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)

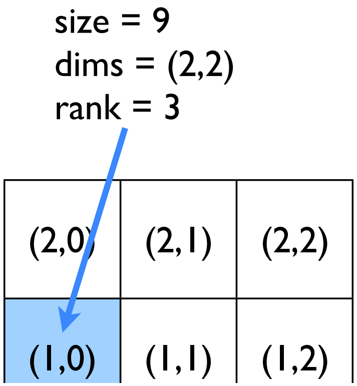


Create new communicator with new topology

- MPI_Cart_create (
  integer comm_old,
  integer ndims,
  integer [dims],
  logical [periods],
  integer reorder,
  integer comm_cart,
  integer ierr)

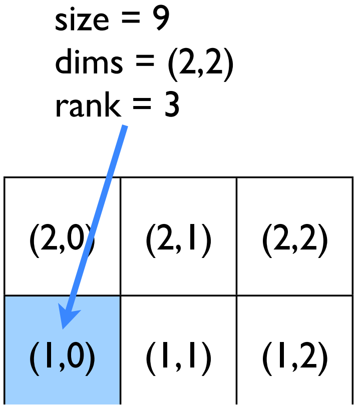


Create new communicator with new topology



```
C
ierr = MPI_Cart_shift(MPI_COMM new_comm, int dim,
       int shift, int *left, int *right)
ierr = MPI_Cart_coords(MPI_COMM new_comm, int rank,
       int ndims, int *gridcoords)
```

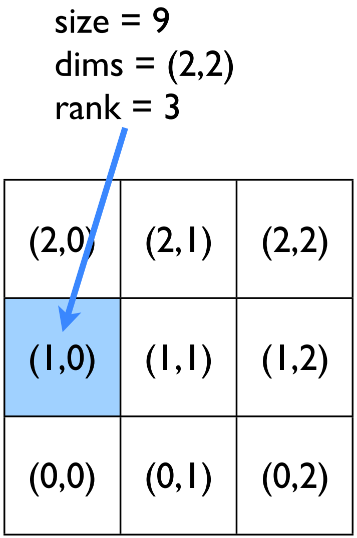
Create new communicator with new topology



```
FORTRAN
call MPI_Cart_shift(integer new_comm, dim, shift,
       left, right, ierr)
call MPI_Cart_coords(integer new_comm, rank,
       ndims, [gridcoords], ierr)
```

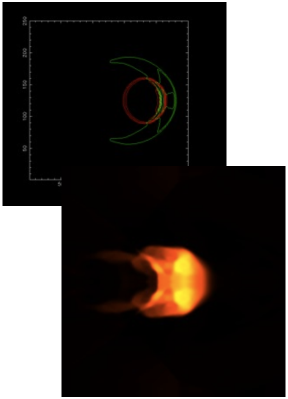
Let's try starting to do this together

- In a new directory:
- add mpi_init, _finalize, comm_size.
- mpi_cart_create
- rank on *new* communicator.
- neighbours
- Only do part of domain



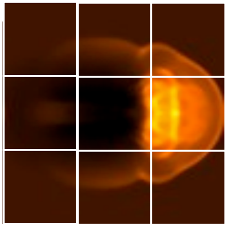
Next

- File IO - have each process write its own file so don't overwrite
- Coordinate min, max across processes for contours, images.
- Coordinate min in cfl routine.



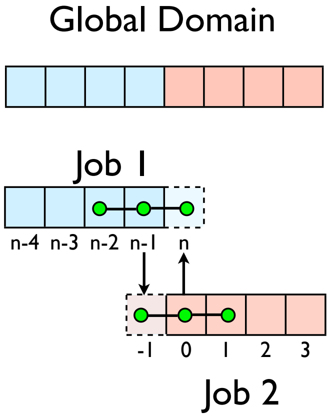
MPIing the code

- Domain decomposition
- Lots of data - ensures locality
- How are we going to handle getting non-local information across processors?



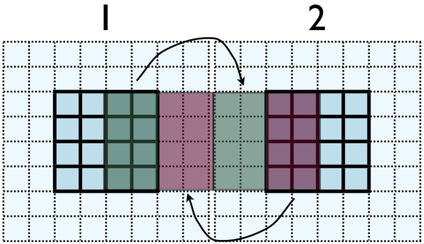
Guardcells

- 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



Guard cell fill

- When we're doing boundary conditions.
- Swap guardcells with neighbour.



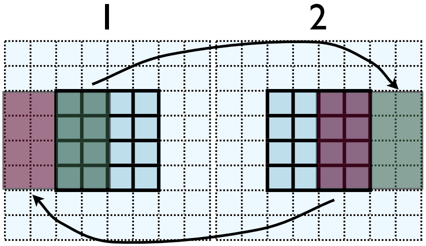
1: u(:, nx:nx+ng, ng:ny-ng)
→ 2: u(:,1:ng, ng:ny-ng)

2: u(:, ng+1:2*ng, ng:ny-ng)
→ 1: u(:, nx+ng+1:nx+2*ng, ng:ny-ng)

(ny-2*ng)*ng values to swap

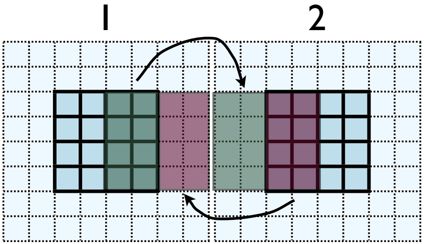
Cute way for Periodic BCs

- Actually make the decomposed mesh periodic;
- Make the far ends of the mesh neighbors
- Don't know the difference between that and any other neighboring grid
- Cart_create sets this up for us automatically upon request.



Implementing in MPI

- No different in principle than diffusion
- Just more values
- And more variables: dens, ener, imomx....
- Simplest way: copy all the variables into an NVARS*(ny-2*ng)*ng sized



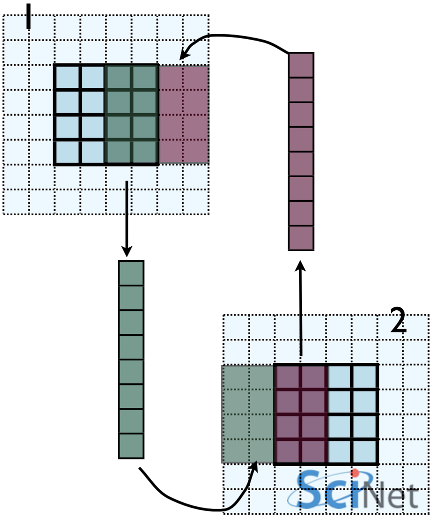
1: u(:, nx:nx+ng, ng:ny-ng)
→ 2: u(:,1:ng, ng:ny-ng)

2: u(:, ng+1:2*ng, ng:ny-ng)
→ 1: u(:, nx+ng+1:nx+2*ng, ng:ny-ng)

nvars*(ny-2*ng)*ng values to swap

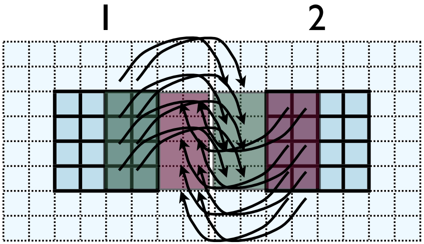
Implementing in MPI

- No different in principle than diffusion
- Just more values
- And more variables: dens, ener, temp....
- Simplest way: copy all the variables into an NVARS*(ny-2*ng)*ng sized



Implementing in MPI

- Even simpler way:
- Loop over values, sending each one, rather than copying into buffer.
- NVARS*nguard*(ny-2*nguard) latency hit.
- Would completely dominate communications cost.

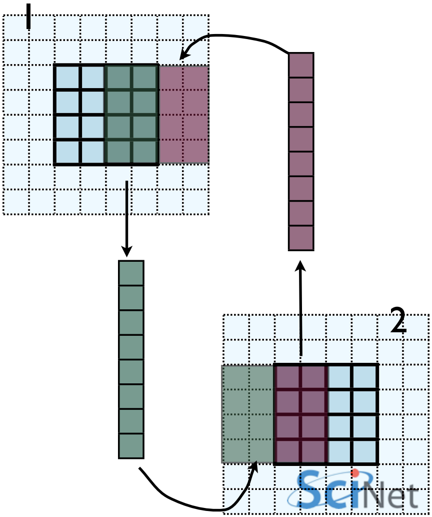


Implementing in MPI

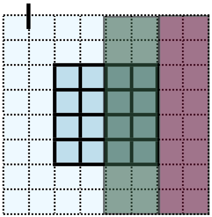
- Let's do this together
- solver.f90/solver.c; implement to bufferGuardcells
- When do we call this in timestep?

Implementing in MPI

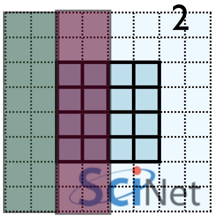
- This approach is simple, but introduces extraneous copies
- Memory bandwidth is already a bottleneck for these codes
- It would be nice to just point at the start of the guardcell data and have MPI read it from there.



Implementing in MPI

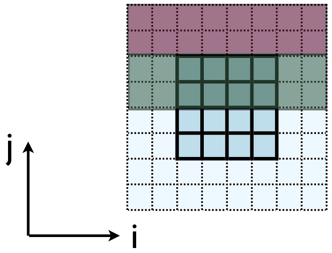


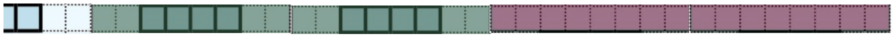
- Let me make one simplification for now; copy whole stripes
- This isn't necessary, but will make stuff simpler at first
- Only a cost of $2xNg^2 = 8$ extra cells (small fraction of ~200-2000 that would normally be copied)



Implementing in MPI

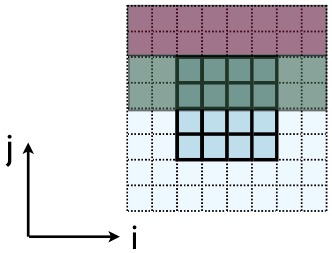
- Recall how 2d memory is laid out
- y-direction guardcells contiguous



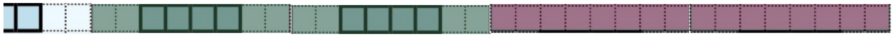


Implementing in MPI

- Can send in one go:

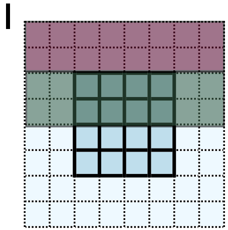


```
call MPI_Send(u(1,1,ny), nvars*nguard*ny, MPI_REAL, ....)
ierr = MPI_Send(&(u[ny][0][0]), nvars*nguard*ny, MPI_FLOAT, ....)
```



Implementing in MPI

- Creating MPI Data types.
- MPI_Type_contiguous: simplest case. Lets you build a string of some other type.

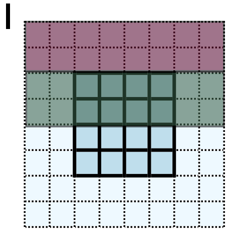


Count OldType &NewType

```
MPI_Datatype ybctype;

ierr = MPI_Type_contiguous(nvals*nguard*(ny), MPI_REAL, &ybctype);
ierr = MPI_Type_commit(&ybctype);

MPI_Send(&(u[ny][0][0]), 1, ybctype, ....)

ierr = MPI_Type_free(&ybctype);
```

Implementing in MPI

- Creating MPI Data types.
- MPI_Type_contiguous: simplest case. Lets you build a string of some other type.



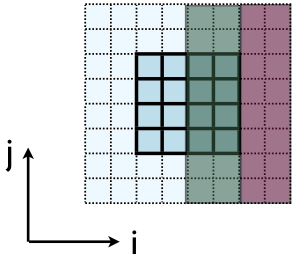
Count OldType NewType

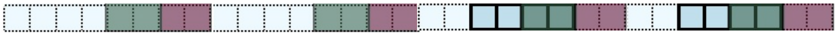
```
integer :: ybctype

call MPI_Type_contiguous(nvals*nguard*(ny), MPI_REAL, ybctype, ierr)
call MPI_Type_commit(ybctype, ierr)

MPI_Send(u(1,1,ny), 1, ybctype, ....)

call MPI_Type_free(ybctype, ierr)
```

SciNet

Implementing in MPI

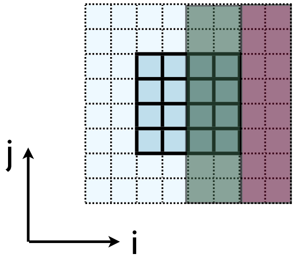
- Recall how 2d memory is laid out
- x gcs or boundary values *not* contiguous
- How do we do something like this for the x-direction?

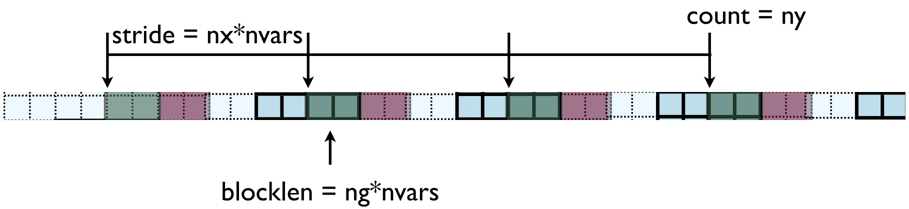




Implementing in MPI

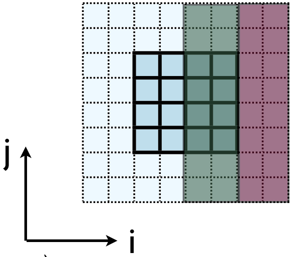
```
int MPI_Type_vector(
        int count,
        int blocklen,
        int stride,
        MPI_Datatype old_type,
        MPI_Datatype *newtype );
```

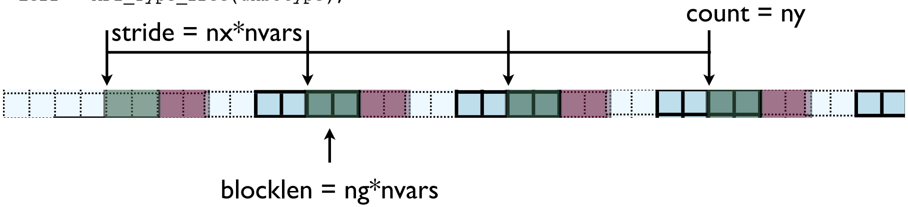




Implementing in MPI

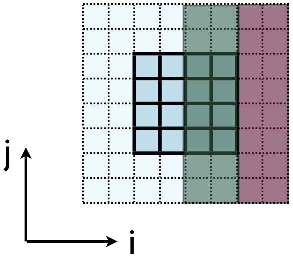
```
ierr = MPI_Type_vector(ny, nguard*nvars,
          nx*nvars, MPI_FLOAT, &xbctype);

ierr = MPI_Type_commit(&xbctype);

ierr = MPI_Send(&(u[0][nx][0]), 1, xbctype, ....)

ierr = MPI_Type_free(&xbctype);
```

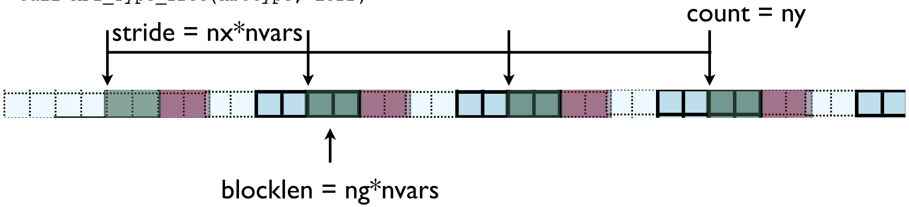




Implementing in MPI

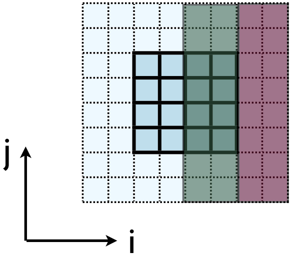
```
call MPI_Type_vector(ny, nguard*nvars,
        nx*nvars, MPI_REAL, xbctype, ierr)

call MPI_Type_commit(xbctype, ierr)

call MPI_Send(u(1,nx,1), 1, ybctype, ....)

call MPI_Type_free(xbctype, ierr)
```

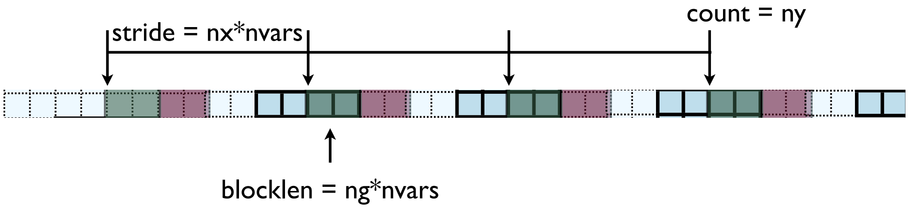




Implementing in MPI

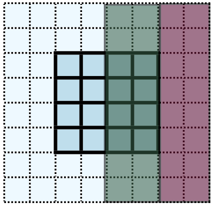
- Check: total amount of data = blocklen*count = ny*ng*nvars
- Skipped over stride*count = nx*ny*nvars





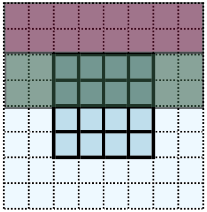
Implementing in MPI

- Hands-On: Implement X guardcell filling with types.
- Implement vectorGuardCells
- For now, create/free type each cycle through; ideally, we'd create/free these once.



In MPI, there's always more than one way..

- MPI_Type_create_subarray ; piece of a multi-dimensional array.
- *Much* more convenient for higher-dimensional arrays
- (Otherwise, need vectors of vectors of vectors...)



```
int MPI_Type_create_subarray(
      int ndims, int *array_of_sizes,
      int *array_of_subsizes,
      int *array_of_starts,
      int order,
      MPI_Datatype oldtype,
      MPI_Datatype &newtype);

call MPI_Type_create_subarray(
      integer ndims, [array_of_sizes],
      [array_of_subsizes],
      [array_of_starts],
      order, oldtype,
      newtype, ierr)
```