Building and Running a Parallel Application…
…Continued

Steve Lantz
Computing and Information Science 4205
www.cac.cornell.edu/~slantz

A Course Project to Meet Your Goals!

Assignment due 2/6:

- **Propose a problem in parallel computing that you would like to solve as an outcome of this course**
- **It should involve the following elements:**
  - **Designing a parallel program (due at the end of week 5)**
  - **Writing a proof-of-principle code (due at the end of week 7)**
  - **Verifying that your code works (due at the end of week 8)**
- **It should not be so simple that you can look it up in a book**
- **It should not be so hard that it's equivalent to a Ph.D. thesis project**
- **You will be able to seek help from me and your classmates!**
- **Take this as an opportunity to work on something you care about**

Which Technique Should You Choose?

MPI

- Code will run on distributed- and/or shared-memory systems
- Functional or nontrivial data parallelism within a single application

OpenMP

- Code will run on shared-memory systems
- Parallel constructs are simple, e.g., independent loop iterations
- Want to parallelize a serial code using OpenMP directives to (say) gcc
- Want to create a hybrid by adding OpenMP directives to an MPI code

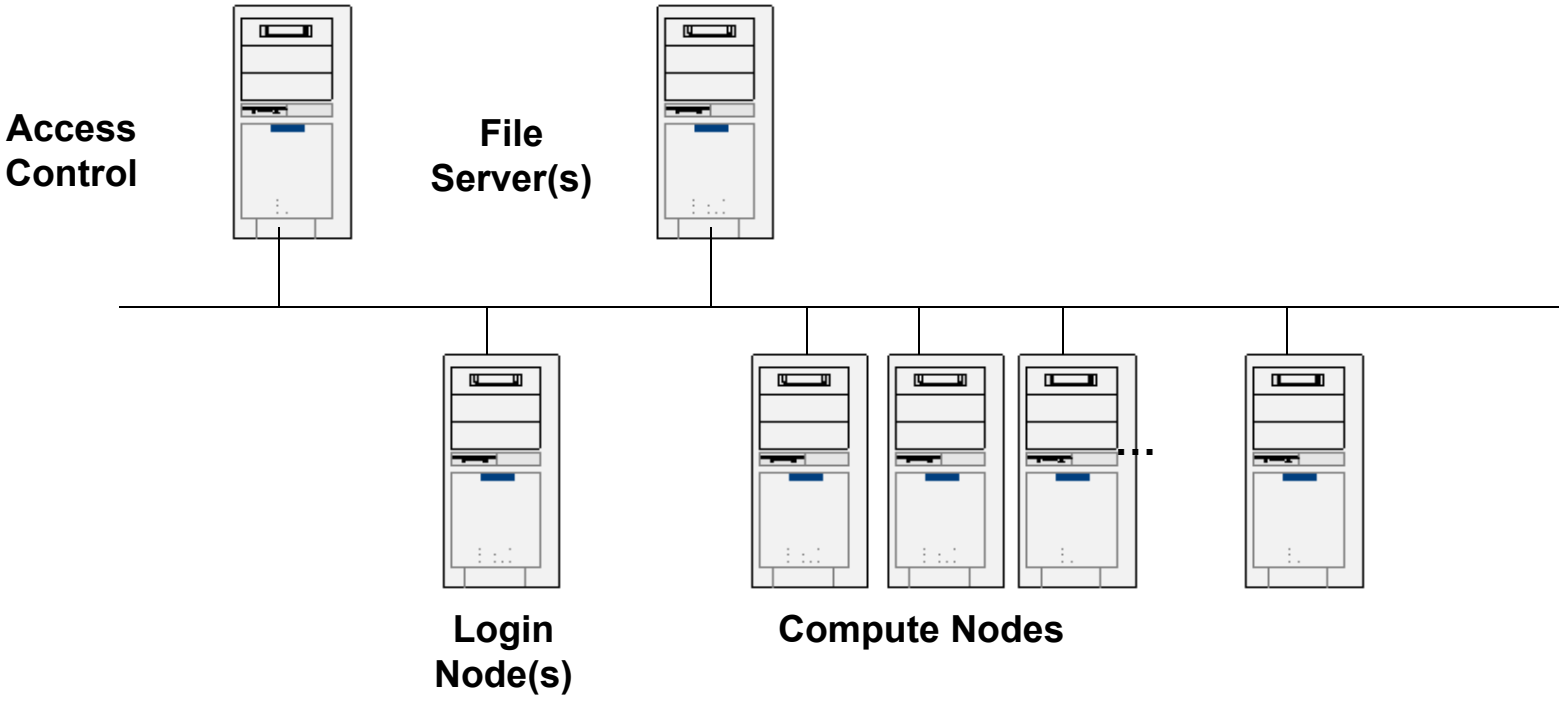
Task-Oriented Parallelism (Grid style)

- Parallelism is at the application-level, coarse-grained, scriptable
- Little communication or synchronization is needed

Running Programs in a Cluster Computing Environment

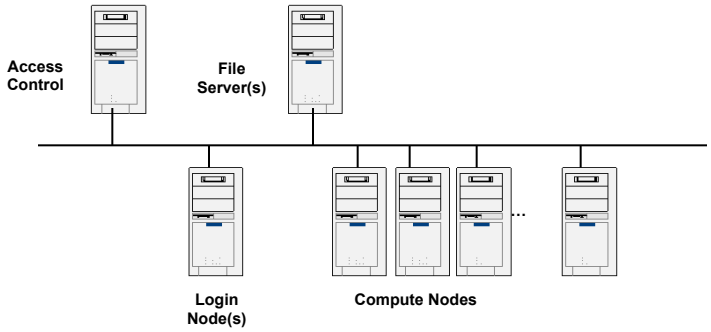
The Basics

- Login Nodes
- File Servers & Scratch Space
- Compute Nodes
- Batch Schedulers



Login Nodes

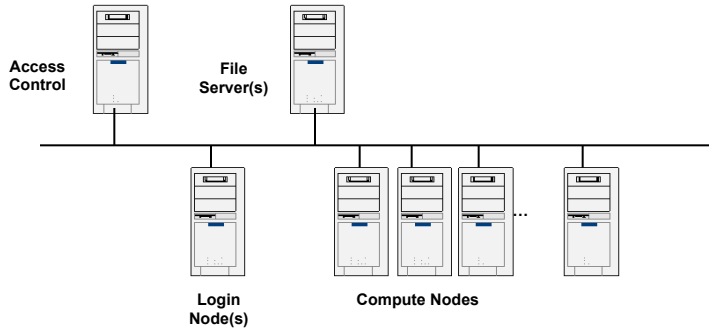
- **Develop, Compile & Link Parallel Programs**
- **Availability of Development Tools & Libraries**
- **Submit, Cancel & Check the Status of Jobs**



File Servers & Scratch Space

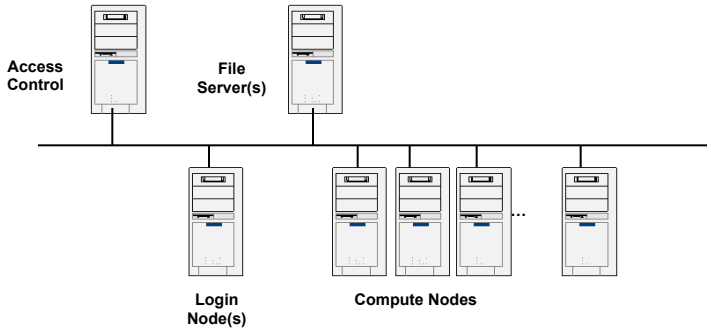
- **File Servers**
  - Store source code, batch scripts, executables, input data, output data
  - Should be used to stage executables and data to compute nodes
  - Should be used to store results from compute nodes when jobs complete
  - Normally backed up

- **Scratch Space**
  - Temporary storage space residing on compute nodes
  - Executables, input data and output data reside here during while the job is running
  - Not backed up and normally old files are deleted regularly



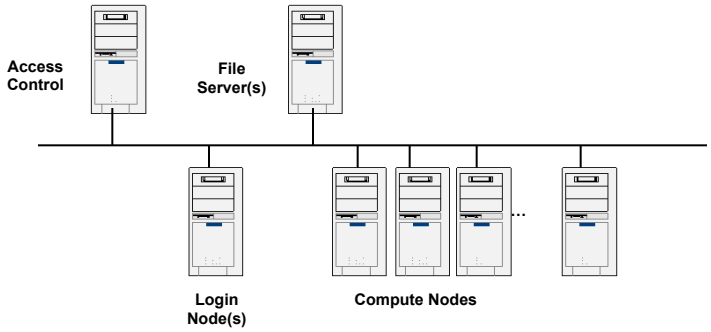
Compute Nodes

- **One or more used at a time to run batch jobs**
- **Have necessary software and run time libraries installed**
- **User only has access when their job is running**
  - (Note difference between batch and interactive jobs)



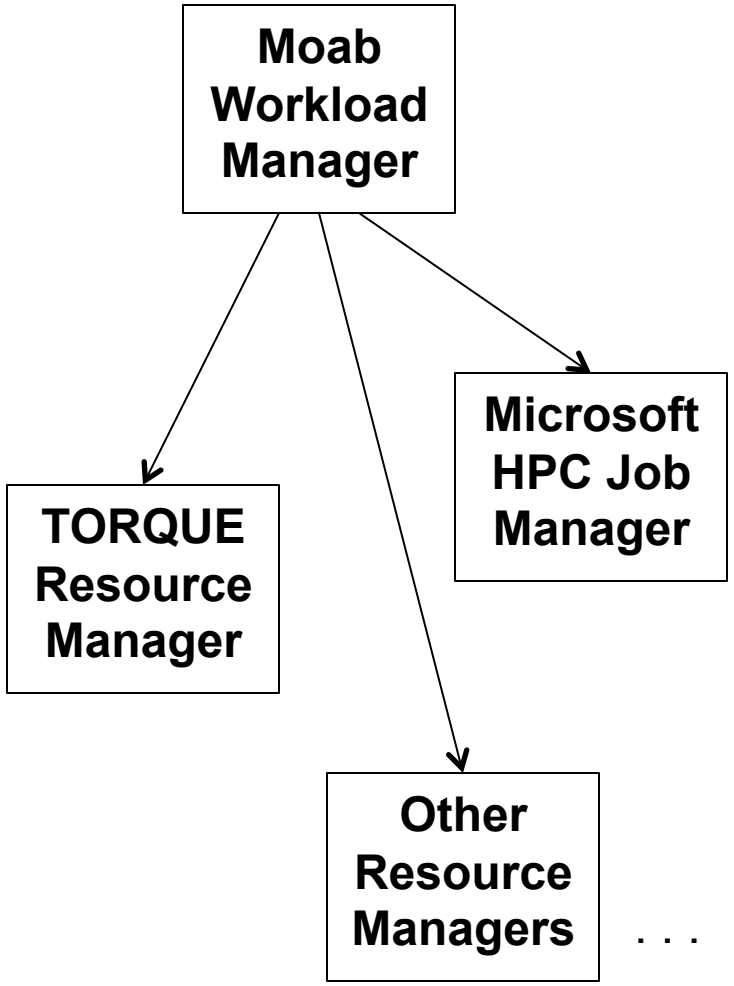
Batch Schedulers

- **Decide when jobs run and must stop based on requested resources**
- **Run jobs on compute nodes for users *as* the users**
- **Enforce local usage policies**
  - Who has access to what resources
  - How long jobs can run
  - How many jobs can run
- **Ensure resources are in working order when jobs complete**
- **Different types**
  - High Performance
  - High Throughput

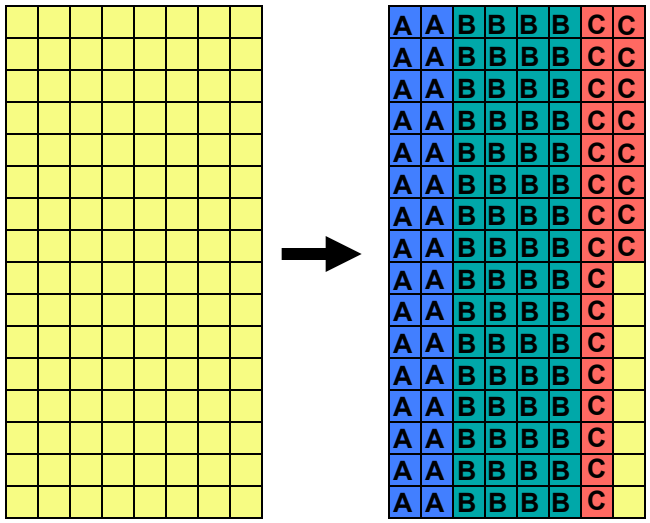


Next-Generation Job Scheduling: Workload Manager and Resource Managers

- **Moab Workload Manager (from Cluster Resources, Inc.) does overall job scheduling**
  - Manages multiple resources by utilizing the resources' own management software
  - More sophisticated than a cluster batch scheduler; e.g., Moab can make advanced reservations
- **TORQUE or other resource managers control subsystems**
  - Subsystems can be distinct clusters or other resources
  - For clusters, the typical resource manager is batch scheduler
  - Torque is based on OpenPBS (Portable Batch System)

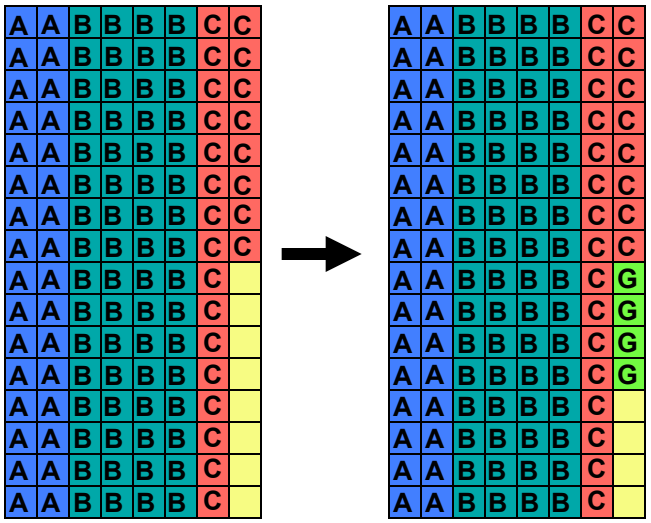


Backfill Scheduling Algorithm 1 of 3



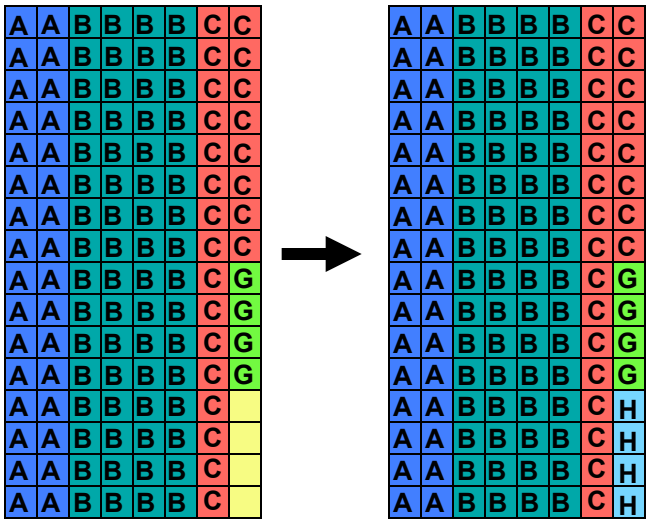
User Name	Number of Nodes	Number of Minutes	Job Status
User A	32	120	S
User B	64	60	S
User C	24	180	S
User D	32	120	W
User E	16	120	W
User F	10	480	W
User G	4	30	W
User H	4	120	W

Backfill Scheduling Algorithm 2 of 3



User Name	Number of Nodes	Number of Minutes	Job Status
User A	32	120	R
User B	64	60	R
User C	24	180	R
User D	32	120	W
User E	16	120	W
User F	10	480	W
User G	4	30	S
User H	4	120	W

Backfill Scheduling Algorithm 3 of 3



User Name	Number of Nodes	Number of Minutes	Job Status
User A	32	120	R
User B	64	60	R
User C	24	180	R
User D	32	120	W
User E	16	120	W
User F	10	480	W
User G	4	30	R
User H	4	120	S

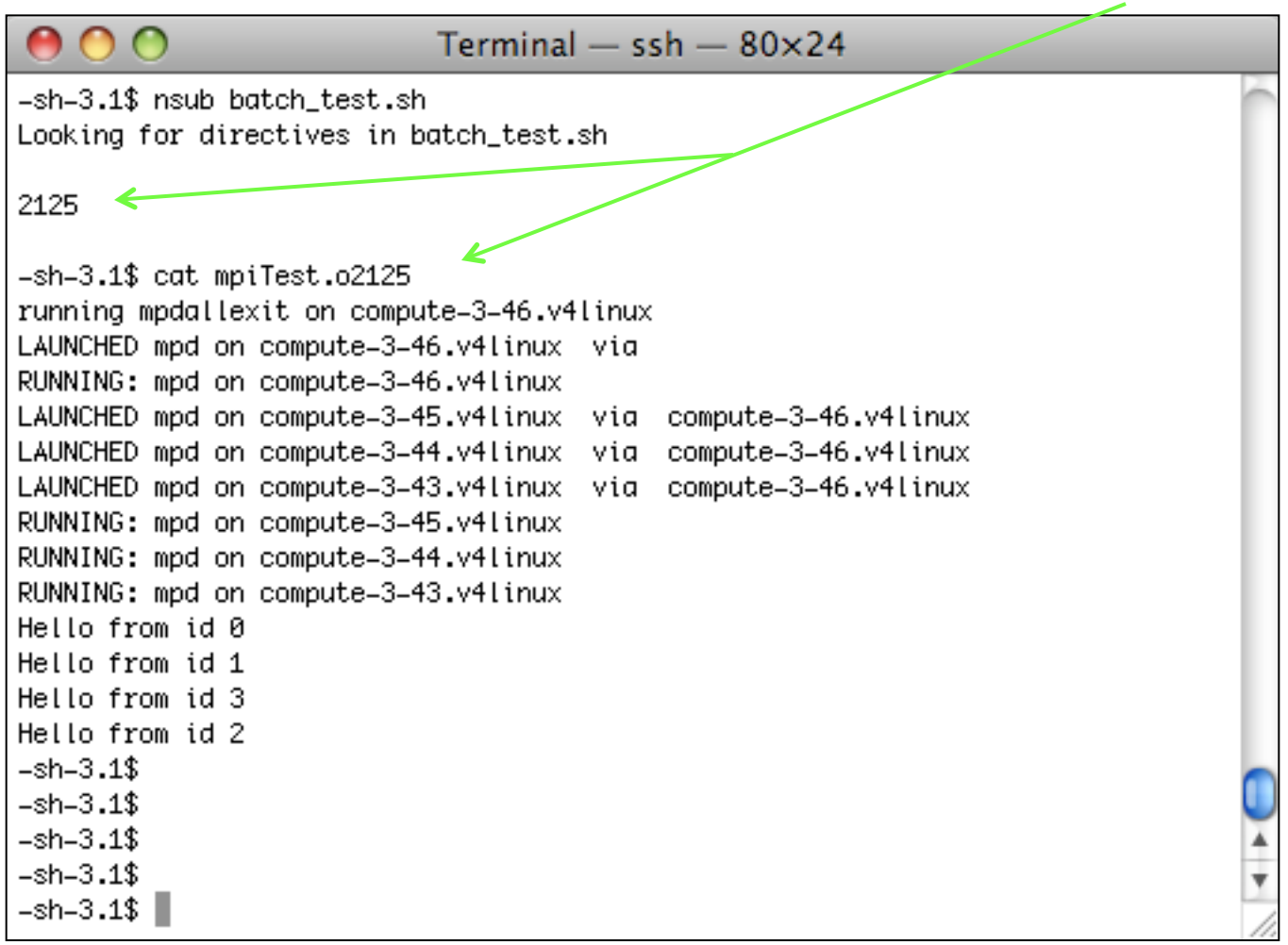
Batch Scripts

- **See examples in the CAC Web documentation at:** http://www.cac.cornell.edu/Documentation/batch/examples.aspx
- **Also refer to batch_test.sh on the course website…**

```
#!/bin/sh

#PBS -A xy44_0001
#PBS -l walltime=02:00,nodes=4:ppn=1
#PBS -N mpiTest
#PBS -j oe
#PBS -q v4

# Count the number of nodes
np=$(wc -l < $PBS_NODEFILE)

# Boot mpi on the nodes
mpdboot -n $np --verbose -r /usr/bin/ssh -f $PBS_NODEFILE

# Now execute
mpiexec -n $np $HOME/CIS4205/helloworld
mpdallexit
```

Submitting a Batch Job

- **nsub batch_test.sh …job number appears in name of output file**



Moab Batch Commands

• showq	Show status of jobs in the queues
• checkjob -A *jobid*	Get info on job *jobid*
• mjobctl -c *jobid*	Cancel job number *jobid*
• checknode hostname	Check status of a particular machine
• echo $PBS_NODEFILE	At runtime, see location of machines file
• showbf -u *userid* -A	Show available resources for *userid*

Available batch queues

- **v4 – primary batch queue for most work**
- **v4dev – development queue for testing/debugging**
- **v4-64g – queue for the high-memory (64GB/machine) servers**

More Than One MPI Process Per Node (ppn)

```
#!/bin/sh

#PBS -A xy44_0001
#PBS -l walltime=02:00,nodes=1:ppn=1
# CAC's batch manager always always resets ppn=1
# For a different ppn value, use -ppn in mpiexec
#PBS -N OneNode8processes
#PBS -j oe
#PBS -q v4

# Count the number of nodes
nnode=$(wc -l < $PBS_NODEFILE)
ncore=8
np=$(($ncore*$nnode))

# Boot mpi on the nodes
mpdboot -n $nnode --verbose -r /usr/bin/ssh -f $PBS_NODEFILE

# Now execute... note, in mpiexec, the -ppn flag must precede the -n flag
mpiexec -ppn $ncore -n $np $HOME/CIS4205/helloworld > $HOME/CIS4205/hifile
mpiexec -ppn $ncore -n $np hostname
mpdallexit
```

Linux Tips of the Day

- **Try gedit instead of vi or emacs for intuitive GUI text editing**
  - gedit requires X Windows
  - Must login with “ssh -X” and run an X server on your local machine
- **Try nano as a simple command-line text editor**
  - originated with the Pine email client for Unix (pico)
- **To retrieve an example from the course website, use wget:**

  wget http://www.cac.cornell.edu/~slantz/CIS4205/Downloads/batch_test.sh.txt
- **To create an animated gif, use Image Magick:**

  display -scale 200x200 *pgm mymovie.gif

Distributed Memory Programming
Using Basic MPI (Message Passing Interface)

The Basics
Helloworld.c

- MPI programs must include the MPI header file
- Include file is mpi.h for C, mpif.h for Fortran
- For Fortran 90/95, USE MPI from mpi.mod (perhaps compile mpi.f90)
- mpicc, mpif77, mpif90 already know where to find these files

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

void main(int argc, char **argv )
 {
  int myid, numprocs;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  printf("Hello from id %d\n", myid);
  MPI_Finalize();
 }
```

MPI_Init

- Must be the first MPI function call made by every MPI process
- (Exception: MPI_Initialized tests may be called head of MPI_Init)
- In C, MPI_Init also returns command-line arguments to all processes
- Note, arguments in MPI calls are generally pointer variables
- This aids Fortran bindings ("call by reference", not "call by value")

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

void main(int argc, char **argv )
 {
  int i;
  int myid, numprocs;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  for (i=0; i<argc; i++) printf("argv[%d]=%s\n",i,argv[i]);
  printf("Hello from id %d\n", myid);
  MPI_Finalize();
 }
```

MPI_Comm_rank

- After MPI is initialized, every process is part of a "communicator"
- MPI_COMM_WORLD is the name of this default communicator
- MPI_Comm_rank returns the number (rank) of the current process
- For MPI_COMM_WORLD, this is a number from 0 to (numprocs-1)
- It is possible to create other, user-defined communicators

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

void main(int argc, char **argv )
 {
  int i;
  int myid, numprocs;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  for (i=0; i<argc; i++) printf("argv[%d]=%s\n",i,argv[i]);
  printf("Hello from id %d\n", myid);
  MPI_Finalize();
 }
```

MPI_Comm_size

- **Returns the total number of processes in the communicator**

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

void main(int argc, char **argv )
 {
  int i;
  int myid, numprocs;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  for (i=0; i<argc; i++) printf("argv[%d]=%s\n",i,argv[i]);
  printf("Hello from id %d, %d or %d processes\n",myid,myid+1,numprocs);
  MPI_Finalize();
 }
```

MPI_Finalize

- Called when all MPI calls are complete
- Frees system resources used by MPI

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

void main(int argc, char **argv )
 {
  int i;
  int myid, numprocs;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  for (i=0; i<argc; i++) printf("argv[%d]=%s\n",i,argv[i]);
  printf("Hello from id %d, %d or %d processes\n",myid,myid+1,numprocs);
  MPI_Finalize();
 }
```

MPI_Send

MPI_Send(void *message, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)

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

void main(int argc, char **argv )
 {
  int i;
  int myid, numprocs;
  char sig[80];
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  for (i=0; i<argc; i++) printf("argv[%d]=%s\n",i,argv[i]);
  if (myid == 0)
   {
    printf("Hello from id %d, %d of %d processes\n",myid,myid+1,numprocs);
    for(i=1; i<numprocs; i++)
     {
      MPI_Recv(sig,sizeof(sig),MPI_CHAR,i,0,MPI_COMM_WORLD,&status);
      printf("%s",sig);
     }
   }
  else
   {
    sprintf(sig,"Hello from id %d, %d of %d processes\n",myid,myid+1,numprocs);
    MPI_Send(sig,sizeof(sig),MPI_CHAR,0,0,MPI_COMM_WORLD);
   }
  MPI_Finalize();
 }
```

MPI_Datatype
Datatypes for C

MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

MPI_Recv

MPI_Recv(void *message, int count, MPI_Datatype dype ,int source, int tag, MPI_Comm comm, MPI_Status *status)

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

void main(int argc, char **argv )
 {
  int i;
  int myid, numprocs;
  char sig[80];
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  for (i=0; i<argc; i++) printf("argv[%d]=%s\n",i,argv[i]);
  if (myid == 0)
   {
    printf("Hello from id %d, %d of %d processes\n",myid,myid+1,numprocs);
    for(i=1; i<numprocs; i++)
     {
      MPI_Recv(sig,sizeof(sig),MPI_CHAR,i,0,MPI_COMM_WORLD,&status);
      printf("%s",sig);
     }
   }
  else
   {
    sprintf(sig,"Hello from id %d, %d of %d processes\n",myid,myid+1,numprocs);
    MPI_Send(sig,sizeof(sig),MPI_CHAR,0,0,MPI_COMM_WORLD);
   }
  MPI_Finalize();
 }
```

MPI_Status
Status Record

- **MPI_Recv blocks until a message is received or an error occurs**
- **Once MPI_Recv returns the status record can be checked**
  - status->MPI_SOURCE (where the message came from)
  - status->MPI_TAG (the tag value, user-specified)
  - status->MPI_ERROR (error condition, if any)

```
printf("Hello from id %d, %d of %d processes\n",myid,myid+1,numprocs);
for(i=1; i<numprocs; i++)
 {
  MPI_Recv(sig,sizeof(sig),MPI_CHAR,i,0,MPI_COMM_WORLD,&status);
  printf("%s",sig);
  printf("Message source = %d\n",status.MPI_SOURCE);
  printf("Message tag = %d\n",status.MPI_TAG);
  printf("Message Error condition = %d\n",status.MPI_ERROR);
 }
```

Watch Out for Deadlocks!

- **Deadlocks occur when the code waits for a condition that will never happen**

- **Remember MPI Send and Receive work like channels in Foster's Design Methodology**
  - Sends are asynchronous (the call returns immediately after sending)
  - Receives are synchronous (the call blocks until the receive is complete)

- **A common MPI deadlock happens when 2 processes are supposed to exchange messages and they both issue an MPI_Recv before doing an MPI_Send**

MPI_Wtime & MPI_Wtick

- Used to measure performance (i.e., to time a portion of the code)
- MPI_Wtime returns number of seconds since a point in the past
- Nothing more than a simple wallclock timer, but it is perfectly portable between platforms and MPI implementations
- MPI_Wtick returns the resolution of MPI_Wtime in seconds
- Generally this return value will be some small fraction of a second

MPI_Wtime & MPI_Wtick
example

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
for (i=0; i<argc; i++) printf("argv[%d]=%s\n",i,argv[i]);
if (myid == 0)
 {
  printf("Hello from id %d, %d of %d processes\n",myid,myid+1,numprocs);
  for(i=1; i<numprocs; i++)
   {
    MPI_Recv(sig,sizeof(sig),MPI_CHAR,i,0,MPI_COMM_WORLD,&status);
    printf("%s",sig);
   }
  start = MPI_Wtime();
  for (i=0; i<100; i++)
   {
    a[i] = i;
    b[i] = i * 10;
    c[i] = i + 7;
    a[i] = b[i] * c[i];
   }
  end = MPI_Wtime();
  printf("Our timers precision is %.20f seconds\n",MPI_Wtick());
  printf("This silly loop took %.5f seconds\n",end-start);
```

MPI_Barrier
MPI_Barrier(MPI_Comm comm)

- **A mechanism to force synchronization amongst all processes**
- **Useful when you are timing performance**
  - Assume all processes are performing the same calculation
  - You need to ensure they all start at the same time
- **Also useful when you want to ensure that all processes have completed an operation before any of them begin a new one**

```
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
result = run_big_computation();
MPI_Barrier(MPI_COMM_WORLD);
end = MPI_Wtime();
printf("This big computation took %.5f seconds\n",end-start);
```