Overview of Intro to MPI class

Course Handout: (last update 14 February 2011)

These notes may be found at http://www.dartmouth.edu/~rc/classes/intro_mpi/src. The online version has many links to additional information and may be more up to date than the printed notes

---

Intro to MPI Course Overview

Topics Covered in this Course:

- Parallel programming models
- What is MPI?
- Different versions of MPI
- Different approaches to parallelizing via message passing
- Basic MPI routines
- How to compile and run MPI programs
- Some MPI examples
- MPI resources

Examples Used in this class

(1)

Advantages of Parallel Programming

- Need to solve larger problems
  - more memory intensive
  - more computation
  - more data intensive
- Parallel programming provides
  - more CPU resources
  - more memory resources
  - solve problems that were not possible with serial program
  - solve problems more quickly

Parallel Computer Architectures

Two Basic Architectures

- Distributed Memory (ex. Compute cluster)
  - collection of nodes which have multiple cores
  - each node uses its own local memory
  - work together to solve a problem
  - communicate between nodes and cores via messages
  - nodes are networked together
- Shared Memory Computer
  - multiple cores
  - share a global memory space
  - cores can efficiently exchange/share data

(3)

Parallel Programming Models

- Directives-based parallel programming language
  - OpenMP (most widely used)
  - High Performance Fortran (HPF)
  - directives tell processor how to distribute data and work across the processors
  - directives appear as comments in the serial code
  - implemented on shared memory architectures

- Message Passing (MPI)
  - pass messages to send/receive data between processes
  - each process has its own local variables
  - can be used on either shared or distributed memory architectures
  - outgrowth of PVM software

(4)

Pros and Cons of OpenMP/MPI

- Pros of OpenMP
  - easier to program and debug than MPI
  - directives can be added incrementally - gradual parallelization
  - can still run the program as a serial code
  - serial code statements usually don't need modification
  - code is easier to understand and maybe more easily maintained
- Cons of OpenMP
  - can only be run in shared memory computers
  - requires a compiler that supports OpenMP
  - mostly used for loop parallelization

- Pros of MPI
  - runs on either shared or distributed memory architectures
  - can be used on a wider range of problems than OpenMP
  - each process has its own local variables
  - distributed memory computers are less expensive than large shared memory computers
- Cons of MPI
  - requires more programming changes to go from serial to parallel version
  - can be harder to debug
  - performance is limited by the communcation network between the nodes

(5)

Parallel Programming Issues

- Goal is to reduce execution time
  - computation time
  - idle time - waiting for data from other processors
  - communication time - time the processors take to send and receive messages
- Load Balancing
  - divide the work equally among the available processors
- Minimizing Communication
  - reduce the number of messages passed
  - reduce amount of data passed in messages
- Where possible - overlap communication and computation
- Many problems scale well to only a limited number of processors

(6)

Problem Decomposition

Two kinds of decompositions:

- Domain decomposition
  - data divided into pieces of same size and mapped to different processors
  - processor works only on data assigned to it
  - communicates with other processors when necessary
  - examples of domain (data) decomposition
    - embarrassingly parallel applications (Monte Carlo simulations)
    - finte difference calculations
    - numerical integration
- Functional decomposition
  - used when pieces of data require different processing times
  - performance limited by the slowest process
  - program decomposed into a number of small tasks
  - tasks assigned to processors as they become available
  - implemented in a master/slave paradigm
  - examples of functional decomposition
    - surface reconstruction from a finite element mesh
    - searching images or data bases

(7)

What is MPI?

- MPI stands for Message Passing Interface
- library of functions (C/C++) or subroutines (Fortran)
- History
  - Early message passing Argonne's P4 and Oak Ridge PVM in 1980s
  - MPI-1 completed in May 1994
  - MPI-2 completed in 1998
    - parallel I/O
    - C++/F90 bindings
    - dynamic process management
  - full MPI-2 implementations only recently
- MPI-2 features gradually added to MPI implementations

Differences Between Versions of MPI

- Examples of Different Implementations
  - MPICH - developed by Argonne Nationa Labs (freeware)
  - MPI/LAM - developed by Indiana, OSC, Notre Dame (freeware)
  - MPI/Pro - commerical product
  - Apple's X Grid
  - OpenMPI - MPI-2 compliant, thread safe
- Similiarities in Various Implementations
  - source code compatibility (except parallel I/O)
  - programs should compile and run as is
  - support for heterogeneous parallel architectures
    - clusters, groups of workstations, SMP computers, grids
- Difference in Various Implementations
  - - commands for compiling and linking
  - - how to launch an MPI program
  - - parallel I/O (from MPI-2)
  - -debugging
- Programming Approaches
  - SPMD - Single Program Multiple Data (same program on all processors)
  - MPMD- Multiple Program Multiple Data (different programs on different processors)

Hello World MPI Examples

4 most used MPI functions/subroutines

- MPI_Init
- MPI_Comm_Rank
- MPI_Comm_Size
- MPI_Finalize

```c
/* C Example */
#include <stdio.h>
#include <mpi.h>


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

  MPI_Init (&argc, &argv);      /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);        /* get current process id */
  MPI_Comm_size (MPI_COMM_WORLD, &size);        /* get number of processes */
  printf( "Hello world from process %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```

```fortran
c  Fortran example
   program hello
   include 'mpif.h'
   integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)

   call MPI_INIT(ierror)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
   call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
   print*, 'node', rank, ': Hello world'
   call MPI_FINALIZE(ierror)
   end
```

(10)

How to Compile MPI Programs

MPICH shell script command to compile and link programs

- mpicc
- mpiCC
- mpif77
- mpif90

Command line options:

- -help: help on command line options
- -compile_info: show commands and options used to compile a program
- -link_info: show commands and options used to link a program
- -cc : used with mpicc to change the compiler used
- -CC: used with mpiCC to change the compiler used
- -fc: used with mpif77 to change the compiler used
- -f90: used with mpif90 to change the compiler used

```
andes:~> mpicc -o hello_world_c hello_world.c
andes:~>
andes:~> mpicc -compile_info
gcc -DUSE_STDARG -DHAVE_STDLIB_H=1 -DHAVE_STRING_H=1 -DHAVE_UNISTD_H=1 \
-DHAVE_STDARG_H=1 -DUSE_STDARG=1 -DMALLOC_RET_VOID=1 -c -I/usr/local/mpich-1.2.7p1/include


andes:~> mpif90 -o hello_world_f hello_world.f
andes:~>
andes:~> mpif90 -compile_info
ln -s /usr/local/mpich-1.2.7p1/include/mpif.h mpif.h
ifort -c -I/usr/local/mpich-1.2.7p1/include/f90choice
rm -f mpif.h
```

How to Run MPICH-1 Programs Interactively

MPICH-1 **mpirun** command starts up MPI programs

where do the processes run:

- SMP computer - any processors that are available
- Distributed Computer
  - (default) nodes in file /MPICH_DIR/share/machines.ARCH (for example: **machines.LINUX**)
  - use the machinefile option to mpirun to specify nodes to run on
  - always runs one process on node where mpirun comand was executed (unless **-nolocal** mpirun option used)

Use **mpirun -help** to see command line options

On SMP computer line andes:

```
andes:~> mpirun -np 4 hello_world_c
Hello world from process 1 of 4
Hello world from process 2 of 4
Hello world from process 3 of 4
Hello world from process 0 of 4
```

On Discovery cluster interactive nodes t01,t02,t03:

```
[sas@t01]$ cat machine.file
t01
t01
t01
t01
t02
t02
t02
t02

[sas@t01]$ mpirun -np 4 -machinefile machine.file  hello_world_c
Hello world from process 0 of 4
Hello world from process 1 of 4
Hello world from process 3 of 4
Hello world from process 2 of 4
```

```
[sas@t01]$ mpirun -np 8 -t hello_world_c
Procgroup file:
t01.bw01.dartmouth.edu 0 /home/sas/Classes/intro_mpi/hello_world_c
t01 1 /home/sas/Classes/intro_mpi/hello_world_c
t01 1 /home/sas/Classes/intro_mpi/hello_world_c
t01 1 /home/sas/Classes/intro_mpi/hello_world_c
t01 1 /home/sas/Classes/intro_mpi/hello_world_c
t02 1 /home/sas/Classes/intro_mpi/hello_world_c
t02 1 /home/sas/Classes/intro_mpi/hello_world_c
t02 1 /home/sas/Classes/intro_mpi/hello_world_c
t02 1 /home/sas/Classes/intro_mpi/hello_world_c
/home/sas/Classes/intro_mpi/hello_world_c -p4pg /home/sas/Classes/intro_mpi/PI11653 \
 -p4wd /home/sas/Classes/intro_mpi
```

How to Run MPICH-2 Programs Interactively

There are 3 commands necessary to run MPICH-2 programs

- **mpdboot** - boot the MPICH2 engine wth the correct number of mpd processes
- **mpiexec** - start up the MPIC2 program
- **mpdallexit** - close down the MPICH2 engine and all mpd processes

mpd - MPI process manager daemon

- In the directory where you will be running your MPI program, create a file called `mpd.hosts` which will contain the name(s) of the computers and the number of processes on each computer where you will be running your program.
- If you are going to be running it on all 4 cores of each of two computers called t01 and t02, the file will contain the following:

```
t01:4
t02:4
```

- Then issue the mpdboot command to boot the MPICH2 engine with the correct number of mpd processes:
  - **mpdboot -f mpd.hosts -n 2**
  - *note: 2 is the number of nodes you will be using*
- Next start `your MPI program` using the `mpiexec` command:
  - **mpiexec -np 8 mpi_program_name [program argume**
- You can run the `mpiexec` command as many time as needed and when you are finished, issue the following command to close down the MPICH2 engine and all of the mpd processes:
  - **mpdallexit**

On computer t01:

```
t01:~> mpiexec -n 8 hello_world_c
Hello world from process 1 of 8
Hello world from process 6 of 8
Hello world from process 0 of 8
Hello world from process 3 of 8
Hello world from process 4 of 8
Hello world from process 2 of 8
Hello world from process 5 of 8
Hello world from process 7 of 8
```

How to Run MPICH-1 Programs in a Batch Queue

Example of Submitting to discovery PBS batch queue

Here is an example of a script that you can use to submit a job to the PBS batch queue. Use the command **qsub** to submit the file to the PBS batch queue. Use the command **qstat job#** where the job number is the number that was returned by the **qsub** command. Use the **showq** command to see the jobs in the PBS batch queue.

The batch queue determines which processors your job runs on.

```
discovery:~> cat sample_pbs_script
# start up bash script as if it were a login shell
#!/bin/bash -l
# declare a name for this job to be sample_job
#PBS -N hello_world
# request the default queue for this job
#PBS -q default
```

```
# request a total of 8 processors for this job (2 nodes and 4 processors per node)
#PBS -l nodes=2:ppn=4
# request 1 hour of wall time
#PBS -l walltime=1:00:00
# specify your email address so mail can be sent to when the job is finished
#PBS -M John.Smith@dartmouth.edu
# have email sent to you when the job is finished
#PBS -m ae
#change to the directory where you submitted the job
cd $PBS_O_WORKDIR
#include the full path to the name of your MPI program
/opt/mpiexec/0.84/bin/mpiexec -comm p4 /home/sas/Classes/intro_mpi/hello_world_c

discovery:~> qsub sample_pbs_script
40952.endeavor.bw01.dartmouth.edu

discovery:qstat 40952
Job id           Name             User             Time Use S Queue
---------------- ---------------- ---------------- -------- - -----
40952.endeavor   hello_world      sas                     0 R default

discovery: showq
ACTIVE JOBS--------------------
JOBNAME            USERNAME      STATE  PROC   REMAINING            STARTTIME

40950                jsmith    Running     2 18:16:25:07  Mon Oct 24 09:33:45
40952                   sas    Running     8 20:20:00:00  Wed Oct 26 13:08:38
.
.
.
   108 active jobs         204 of 428 processors in use by local jobs
                            49 of 91 nodes active
```

How to Run MPICH2 Programs in a Batch Queue

Example of Submitting to discovery PBS batch queue

Here is an example of a script that you can use to submit a job to the PBS batch queue. Use the command **qsub** to submit the file to the PBS batch queue. Use the command **qstat job#** where the job number is the number that was returned by the **qsub** command. Use the **showq** command to see the jobs in the PBS batch queue.

The batch queue determines which processors your job runs on.

```
discovery:~> cat sample_pbs_script
# start up bash script as if it were a login shell
#!/bin/bash -l
# declare a name for this job to be sample_job
#PBS -N hello_world
# request the default queue for this job
#PBS -q default
# request a total of 8 processors for this job (2 nodes and 4 processors per node)
#PBS -l nodes=2:ppn=4
# request 1 hour of wall time
#PBS -l walltime=1:00:00
# specify your email address so mail can be sent to when the job is finished
#PBS -M John.Smith@dartmouth.edu
# have email sent to you when the job is finished
#PBS -m ae
#change to the directory where you submitted the job
cd $PBS_O_WORKDIR
#include the full path to the name of your MPI program
/opt/mpiexec/0.84/bin/mpiexec -comm mpich2-pmi /home/sas/Classes/intro_mpi/hello_world_c

discovery:~> qsub sample_pbs_script
40952.endeavor.bw01.dartmouth.edu

discovery:qstat 40952
Job id           Name             User             Time Use S Queue
---------------- ---------------- ---------------- -------- - -----
40952.endeavor   hello_world      sas                     0 R default

discovery: showq
ACTIVE JOBS--------------------
JOBNAME            USERNAME      STATE  PROC   REMAINING            STARTTIME

40950                jsmith    Running     2 18:16:25:07  Mon Oct 24 09:33:45
40952                   sas    Running     8 20:20:00:00  Wed Oct 26 13:08:38
.
.
.
   108 active jobs         204 of 428 processors in use by local jobs
                            49 of 91 nodes active
```

Basic MPI Functions (Subroutines) and Data types

MPI Communicator (**MPI_Comm MPI_COM_WORLD)**

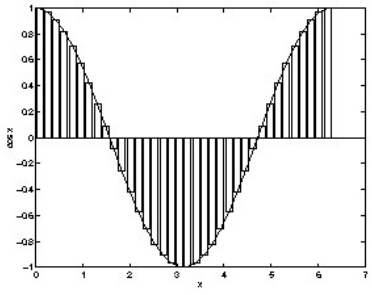
- all of your processes
- defined when you call MPI_init
- all MPI communication calls have a communicator argument
- defined in include file
- C: typedef in mpi.h
- Fortran: integer in mpif.h

Function Purpose	**C Function Call**	**Fortran Subroutine Call**
Initialize MPI	`int MPI_Init(int *argc, char **argv)`	`integer ierror` `call MPI_Init(ierror)`
Determine number of processes within a communicator	`int MPI_Comm_size(MPI_Comm comm, int *size)`	`integer comm,size,ierror` `call MPI_Comm_Size(comm,size,ierror)`
Determine processor rank within a communicator	`int MPI_Comm_rank(MPI_Comm comm, int *rank)`	`integer comm,rank,ierror` `call MPI_Comm_Rank(comm,rank,ierror)`
Exit MPI (must be called last by all processors)	`int MPI_Finalize()`	`CALL MPI_Finalize(ierror)`

Send a message	`int MPI_Send (void *buf,int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`	`<type> buf(*)`
		`integer count, datatype,dest,tag`
		`integer comm, ierror`
		`call MPI_Send(buf,count, datatype, dest, tag, comm, ierror)`

Receive a message	`int MPI_Recv (void *buf,int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`	`<type> buf(*)`
		`integer count, datatype, source,tag`
		`integer comm, status, ierror`
		`call MPI_Recv(buf,count, datatype, source, tag, comm, status, ierror)`

Numerical Integration Example



Example 1. Numerical Integration by Mid-Point rule

$$\int_a^b \cos(x)dx = \sum_{i=0}^{p-1}\sum_{j=0}^{n-1}\int_{a_{ij}}^{a_{ij}+h}\cos(x)dx$$
$$\simeq \sum_{i=0}^{p-1}\left[\sum_{j=0}^{n-1}\cos(a_{ij}+\frac{h}{2})h\right]$$

```
where p = # of processes
      n = number of intervals per process
      a = lower limit of integration
      b = upper limit of integration
      h = (b-a)/(n*p)
      aij = a +[ i*n +j]h
```

```
/* C Example */
#include <mpi.h>
#include <math.h>
#include <stdio.h>
float fct(float x)
{
      return cos(x);
}
/* Prototype */
float integral(float a, int n, float h);
void main(argc,argv)
int argc;
char *argv[];
{
/*********************************************************************
 *                                                                   *
 * This is one of the MPI versions on the integration example        *
 * It demonstrates the use of :                                      *
 *                                                                   *
 * 1) MPI_Init                                                       *
 * 2) MPI_Comm_rank                                                  *
 * 3) MPI_Comm_size                                                  *
 * 4) MPI_Recv                                                       *
 * 5) MPI_Send                                                       *
 * 6) MPI_Finalize                                                   *
 *                                                                   *
 *********************************************************************/
      int n, p, i, j, ierr,num;
      float h, result, a, b, pi;
      float my_a, my_range;

      int myid, source, dest, tag;
      MPI_Status status;
      float my_result;

      pi = acos(-1.0);  /* = 3.14159... */
      a = 0.;           /* lower limit of integration */
      b = pi*1./2.;     /* upper limit of integration */
      n = 100000;          /* number of increment within each process */

      dest = 0;         /* define the process that computes the final result */
      tag = 123;        /* set the tag to identify this particular job */

/* Starts MPI processes ... */

      MPI_Init(&argc,&argv);              /* starts MPI */
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* get current process id */
      MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number of processes */

      h = (b-a)/n;    /* length of increment */
      num = n/p;        /* number of intervals calculated by each process*/
      my_range = (b-a)/p;
      my_a = a + myid*my_range;
      my_result = integral(my_a,num,h);

      printf("Process %d has the partial result of %f\n", myid,my_result);

      if(myid == 0) {
        result = my_result;
        for (i=1;i<p;i++) {
          source = i;           /* MPI process number range is [0,p-1] */
          MPI_Recv(&my_result, 1, MPI_REAL, source, tag,
                        MPI_COMM_WORLD, &status);
          result += my_result;
        }
        printf("The result =%f\n",result);
```

```
      }
      else
        MPI_Send(&my_result, 1, MPI_REAL, dest, tag,
                      MPI_COMM_WORLD);      /* send my_result to intended dest.
                      */
      MPI_Finalize();                       /* let MPI finish up ... */
}
float integral(float a, int n, float h)
{
      int j;
      float h2, aij, integ;

      integ = 0.0;                 /* initialize integral */
      h2 = h/2.;
      for (j=0;j<n;j++) {             /* sum over all "j" integrals */
        aij = a + j*h;        /* lower limit of "j" integral */
        integ += fct(aij+h2)*h;
      }
      return (integ);
}
```

Numerical Integration Example using a MPI Reduction Function

```
where p = # of processes
      n = number of intervals per process
      a = lower limit of integration
      b = upper limit of integration
      h = (b-a)/(n*p)
      aij = a +[ i*n +j]h
```

```
/* C Example */
#include <mpi.h>
#include <math.h>
#include <stdio.h>
float fct(float x)
{
      return cos(x);
}
/* Prototype */
float integral(float a, int n, float h);
void main(argc,argv)
int argc;
char *argv[];
{
/*********************************************************************
 *                                                                   *
 * This is one of the MPI versions on the integration example        *
 * It demonstrates the use of :                                      *
 *                                                                   *
 * 1) MPI_Init                                                       *
 * 2) MPI_Comm_rank                                                  *
 * 3) MPI_Comm_size                                                  *
 * 4) MPI_Reduce                                                       *
 * 5) MPI_Finalize                                                   *
 *                                                                   *
 *********************************************************************/
      int n, p, i, j, ierr,num;
      float h, result, a, b, pi;
      float my_a, my_range;

      int myid, source, dest, tag;
      MPI_Status status;
      float my_result;

      pi = acos(-1.0);  /* = 3.14159... */
      a = 0.;           /* lower limit of integration */
      b = pi*1./2.;     /* upper limit of integration */
      n = 100000;          /* number of increment within each process */

      dest = 0;         /* define the process that computes the final result */
      tag = 123;        /* set the tag to identify this particular job */

/* Starts MPI processes ... */

      MPI_Init(&argc,&argv);              /* starts MPI */
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* get current process id */
      MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number of processes */

      h = (b-a)/n;    /* length of increment */
      num = n/p;        /* number of intervals calculated by each process*/
      my_range = (b-a)/p;
      my_a = a + myid*my_range;
      my_result = integral(my_a,num,h);

      printf("Process %d has the partial result of %f\n", myid,my_result);

/* Use an MPI sum reduction to collect the results */
      MPI_Reduce(&my_result, &result,1,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD);

      MPI_Finalize();                      /* let MPI finish up ... */
}
float integral(float a, int n, float h)
{
      int j;
      float h2, aij, integ;

      integ = 0.0;                /* initialize integral */
      h2 = h/2.;
      for (j=0;j<n;j++) {             /* sum over all "j" integrals */
        aij = a + j*h;        /* lower limit of "j" integral */
        integ += fct(aij+h2)*h;
      }
      return (integ);
}
```

Details of Message Passing

For a Communication to Succeed

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Receiver's buffer must be large enough

MPI Basic Datatypes - C

MPI Data type	C Data Type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNISGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI Basic Datatypes Fortran

MPI Data type	Fortran Data Type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_BYTE	
MPI_PACKED	

(19)

MPI Wildcarding and Use of the Status Handle

Wildcarding

- Receiver can wildcard
- To receive from any source
  - MPI_ANY_SOURCE
- To receive with any tag
  - MPI_ANY_TAG
- Actual source and tag are returned in the receiver's status parameter
- Example:
  - MPI_Recv(buffer,count,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,status)

Using the Status Handle

- Information from a receive is returned from MPI_Recv in **status handle**

Information	**C**	**Fortran**
source	status.MPI_SOURCE	status(MPI_SOURCE)
tag	status.MPI_TAG	status(MPI_TAG)
count	MPI_Get_count(status,datatype,&count)	MPI_GET_COUNT(status,datatype,count,ierr)

MPI Point-to-Point Communication Modes

Send Modes	MPI function	Completion Condition
Synchronous send	MPI_Ssend	only completes when the receive has completed
Buffered send	MPI_Bsend	always completes (unless an error occurs) irrespective of the receiver
**Standard send	MPI_Send	message sent (receive state unknown)
Ready send	MPI_Rsend	may be used only when the a matching receive has already been posted

Receive Mode	MPI function	Completion Condition
Receive	MPI_Recv	Complete when a message has arrived

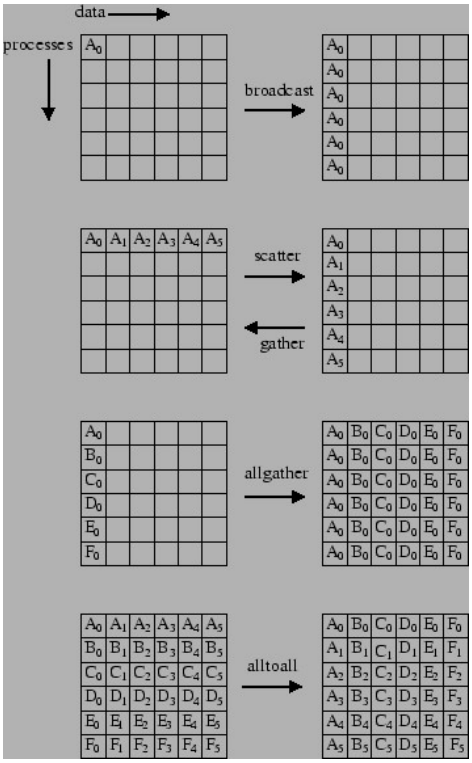
Blocking and Non-blocking Communication

- blocking send or receive
  - call does not return until the operation has been completed
  - allows you to know when it is safe to use the data received or reuse the data sent
- non-blocking send or receive
  - call returns immediately without knowing if the operation has been completed
  - allows you to overlap other computation while testing for the operation to be completed
  - less possibility of deadlocking code
  - used with MPI_Wait or MPI_Test

Type of Commmunication	MPI Function	
blocking send	MPI_Send	
non-blocking send	MPI_Isend	
blocking receive	MPI_Recv	
non-blocking receive	MPI_Irecv	

MPI Collective Communication Routines

MPI_Function	Function Description
MPI_Bcast	Broadcast a message from on process to all others
MPI_Barrier	blocks until all processes have reached this routine
MPI_Reduce	Reduce values from all processes to a single value (add,mult, min, max, etc.)
MPI_Gather	Gathers together values from a group of processes
MPI_Scatter	Sends data from process to the other processes in a group
MPI_Allgather	Gathers data from all tasks and distributes it to all
MPI_Allreduce	Reduces values from all processes and distributes the result back to all processes



MPI Task Parallelism Example

- Master process hands out tasks to worker processes
- Workers are assigned new tasks after they complete previous task
- Matrix multiplication example
  - master process does no work
  - task is to multiply a vector times a matrix
  - initially each process receives the vector and one row of the matrix
  - processes receive additional rows to compute after they send back answer for current row

```
c**************************************************************************
c     matmul.f - matrix - vector multiply, simple self-scheduling version
c**************************************************************************

      Program Matmult

c#########################################################################
c#
c# This is an MPI example of multiplying a vector times a matrix
c# It demonstrates the use of :
c#
c# * MPI_Init
c# * MPI_Comm_rank
c# * MPI_Comm_size
c# * MPI_Bcast
c# * MPI_Recv
c# * MPI_Send
c# * MPI_Finalize
c# * MPI_Abort
c#
c#########################################################################
```

```
      include 'mpif.h'
      integer MAX_ROWS, MAX_COLS, rows, cols
      parameter (MAX_ROWS = 1000, MAX_COLS = 1000, MAX_PROCS =32)
      double precision a(MAX_ROWS,MAX_COLS), b(MAX_COLS), c(MAX_COLS)
      double precision buffer(MAX_COLS), ans
      integer procs(MAX_COLS), proc_totals(MAX_PROCS)
      integer myid, master, numprocs, ierr, status(MPI_STATUS_SIZE)
      integer i, j, numsent, numrcvd, sender, job(MAX_ROWS)
      integer rowtype, anstype, donetype

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      if (numprocs .lt. 2) then
         print *, "Must have at least 2 processes!"
         call MPI_ABORT( MPI_COMM_WORLD, 1 )
         stop
      else if (numprocs .gt. MAX_PROCS) then
         print *, "Must have 32 processes or less."
         call MPI_ABORT( MPI_COMM_WORLD, 1 )
         stop
      endif
      print *, "Process ", myid, " of ", numprocs, " is alive"
      rowtype  = 1
      anstype  = 2
      donetype = 3
      master   = 0
      rows     = 100
      cols     = 100
      if ( myid .eq. master ) then
c        master initializes and then dispatches
c        initialize a and b
         do 20 i = 1,cols
            b(i) = 1
            do 10 j = 1,rows
               a(i,j) = i
 10         continue
 20      continue
         numsent = 0
         numrcvd = 0
c        send b to each other process
         call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
     $        MPI_COMM_WORLD, ierr)
c        send a row to each other process
         do 40 i = 1,numprocs-1
            do 30 j = 1,cols
               buffer(j) = a(i,j)
 30         continue
            call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, i,
     $           rowtype, MPI_COMM_WORLD, ierr)
            job(i)  = i
            numsent = numsent+1
 40      continue
            do 70 i = 1,rows
            call MPI_RECV(ans, 1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,
     $           anstype, MPI_COMM_WORLD, status, ierr)
            sender = status(MPI_SOURCE)
            c(job(sender)) = ans
            procs(job(sender))= sender
            proc_totals(sender+1) =  proc_totals(sender+1) +1
            if (numsent .lt. rows) then
               do 50 j = 1,cols
                 buffer(j) = a(numsent+1,j)
 50            continue
               call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, sender,
     $              rowtype, MPI_COMM_WORLD, ierr)
               job(sender) = numsent+1
               numsent     = numsent+1
            else
            call MPI_SEND(1, 1, MPI_INTEGER, sender, donetype,
     $           MPI_COMM_WORLD, ierr)
            endif
 70      continue
c        print out the answer
          do 80 i = 1,cols
            write(6,809) i,c(i),procs(i)
 809        format('c(',i3,') =',f8.2,' computed by proc #',i3)
 80      continue
         do 81  i=1,numprocs
            write(6,810) i-1,proc_totals(i)
 810        format('Total answers computed by processor #',i2,' were ',i3)
 81        continue
      else
c        compute nodes receive b, then compute dot products until done message
         call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
     $        MPI_COMM_WORLD, ierr)
 90      call MPI_RECV(buffer, cols, MPI_DOUBLE_PRECISION, master,
     $        MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
         if (status(MPI_TAG) .eq. donetype) then
            go to 200
         else
            ans = 0.0
            do 100 i = 1,cols
               ans = ans+buffer(i)*b(i)
 100        continue
            call MPI_SEND(ans, 1, MPI_DOUBLE_PRECISION, master, anstype,
     $           MPI_COMM_WORLD, ierr)
            go to 90
         endif
      endif
 200  call MPI_FINALIZE(ierr)
```

```
      stop
      end
```

MPI Array Search Example 1

- Master process
  - reads the data from a file into an array
  - broadcasts data array to other processes
- Compute processes (includes master)
  - search data array for value 11
  - Notify other processes when they have found it
  - Other processes stop their search when they receive notificiation

```
/* array_search1.c - array searching example where each process is looking for a specific
   number and notifies the other processes when it finds it. Uses a non-blocking receive.
*/
#include <mpi.h>
#include <stdio.h>

main(int argc, char* argv[]) {
  int rank,size;
  MPI_Status status;
  MPI_Request request;
  int done,myfound,inrange,nvalues;
  int b[400];
  int i,j,dummy;
  FILE *infile;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  myfound=0;
  if (rank==0) {
    infile=fopen("data","r");
    for(i=0;i<400;++i) {
      fscanf(infile,"%d",&b[i]);
    }
  }
  MPI_Bcast(b,400,MPI_INT,0,MPI_COMM_WORLD);
  MPI_Irecv(&dummy,1,MPI_INT,MPI_ANY_SOURCE,86,MPI_COMM_WORLD,&request);
  MPI_Test(&request,&done,&status);
  nvalues=400/size;
  i=rank*nvalues;
  inrange=(i<=((rank+1)*nvalues-1) && i>=rank*nvalues);
  while(!done && inrange) {
      if (b[i]==11) {
        dummy=123;
        for(j=0;j<size;++j) {
          MPI_Send(&dummy,1,MPI_INT,j,86,MPI_COMM_WORLD);
        }
        printf("P:%d found it at global index %d\n",rank,i);
        myfound=1;
      }
      MPI_Test(&request,&done,&status);
      ++i;
      inrange=(i<=((rank+1)*nvalues-1) && i>=rank*nvalues);
  }
  if (!myfound) {
    printf("P:%d stopped at global index %d\n",rank,i-1);
  }
  MPI_Finalize();
}
```

MPI Array Search Example 2

- Master process
  - reads the data from a file into an array
  - scatters data array to other processes
  - scatter sends a portion of the array to each process
- Compute processes (includes master)
  - search data array for value 11
  - Notify other processes when they have found it
  - Other processes stop their search when they receive notificiation

```
/* array_search2.c - array searching example where each process is looking for a specific
   number and notifies the other processes when it finds it. Uses a non-blocking receive.
*/

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

main(int argc, char* argv[]) {
  int rank,size;
  MPI_Status status;
  MPI_Request request;
  int done,myfound, nvalues;
  int b[400];
  int *sub;
  int i,j,dummy,index;
  FILE *infile;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  myfound=0;
  if (rank==0) {
    infile=fopen("data","r");
    for(i=0;i<400;++i) {
      fscanf(infile,"%d",&b[i]);
    }
  }
  nvalues=400/size;
  sub = (int *)malloc(nvalues *sizeof(int));
  MPI_Scatter(b,nvalues,MPI_INT,sub,nvalues,MPI_INT,0,MPI_COMM_WORLD);
  MPI_Barrier(MPI_COMM_WORLD);  /* Not needed, put in for fun */
  MPI_Irecv(&dummy,1,MPI_INT,MPI_ANY_SOURCE,86,MPI_COMM_WORLD,&request);
  MPI_Test(&request,&done,&status);
  i=0;
  while(!done && i<nvalues) {
    if (sub[i]==11) {
      dummy=123;
      for(j=0;j<=3;++j) {
    MPI_Send(&dummy,1,MPI_INT,j,86,MPI_COMM_WORLD);
      }
      printf("P:%d found it at local index %d\n",rank,i);
      myfound=1;
    }
    MPI_Test(&request,&done,&status);
    ++i;
  }
  if (!myfound) {
    if (i==0)
      index=0;
    else
      index=i-1;
    printf("P:%d stopped at local index %d\n",rank,index);
  }

}
```

(25)

Avoiding MPI Deadlock or Race Conditions

Deadlock or race conditions occur when the message passing cannot be completed.

Consider the following and assume that the MPI_Send does not complete until the correspondingMPI_Recv is posted and visa versa. The MPI_Send commands will never be completed and the program will deadlock.

```
 if (rank == 0) {
      MPI_Send(..., 1, tag, MPI_COMM_WORLD);
      MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);
 } else if (rank == 1) {
      MPI_Send(..., 0, tag, MPI_COMM_WORLD);
      MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);
 }
```

There are a couple of ways to fix this problem. One way is to reverse the order of one of the send/receive pairs:

```
 if (rank == 0) {
            MPI_Send(..., 1, tag, MPI_COMM_WORLD);
            MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);
 } else if (rank == 1) {
            MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);
            MPI_Send(..., 0, tag, MPI_COMM_WORLD);
}
```

Another way is to make the send be a non-blocking one (MPI_Isend)

```
if (rank == 0) {
            MPI_Isend(..., 1, tag, MPI_COMM_WORLD, &req);
            MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);
            MPI_Wait(&req, &status);
 } else if (rank == 1) {
            MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);
            MPI_Send(..., 0, tag, MPI_COMM_WORLD);
}
```

MPI Error Handling

- MPI provides two predefined error handlers
  - **MPI_ERRORS_ARE_FATAL** (the default): causes MPI to abort
  - **MPI_ERRORS_RETURN**: causes MPI to return an error values instead of aborting
- MPI Error Handling Functions
  - **MPI_Errhandler_set**: set the error handler
  - **MPI_Error_class**: convert an error code into an error class
  - **MPI_Error_string**: returns a string for a given error code

```
/* Numerical Integration Example that uses MPI error checking functions */


#include <mpi.h>
#include <math.h>
#include <stdio.h>
float fct(float x)
{
      return cos(x);
}
/* Prototype */
float integral(float a, int n, float h);
void main(argc,argv)
int argc;
char *argv[];
{
/*********************************************************************
 *                                                                   *
 * This is one of the MPI versions of numerical integration          *
 * It demonstrates the use of :                                      *
 *                                                                   *
 * 1) MPI_Init                                                       *
 * 2) MPI_Comm_rank                                                  *
 * 3) MPI_Comm_size                                                  *
 * 4) MPI_Recv                                                       *
 * 5) MPI_Send                                                       *
 * 6) MPI_Finalize                                                   *
 * 7) MPI_Errhandler_set                                             *
 * 8) MPI_Error_class                                                *
 * 9) MPI_Error_string                                               *
 *                                                                   *
 *********************************************************************/
      int n, p, i, j, ierr,num,errclass,resultlen;
      float h, result, a, b, pi;
      float my_a, my_range;
      char err_buffer[MPI_MAX_ERROR_STRING];

      int myid, source, dest, tag;
      MPI_Status status;
      float my_result;

      pi = acos(-1.0);  /* = 3.14159... */
      a = 0.;           /* lower limit of integration */
      b = pi*1./2.;     /* upper limit of integration */
      n = 100000;          /* number of increment within each process */

      dest = 10;         /* define the process that computes the final result */
      tag = 123;        /* set the tag to identify this particular job */

/* Starts MPI processes ... */

      MPI_Init(&argc,&argv);              /* starts MPI */
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* get current process id */
      MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number of processes */

      /*Install a new error handler */

      MPI_Errhandler_set(MPI_COMM_WORLD,MPI_ERRORS_RETURN); /* return info about
      errors */

      h = (b-a)/n;    /* length of increment */
      num = n/p;       /* number of intervals calculated by each process*/
      my_range = (b-a)/p;
      my_a = a + myid*my_range;
      my_result = integral(my_a,num,h);

      printf("Process %d has the partial result of %f\n", myid,my_result);

      if(myid == 0) {
        result = my_result;
        for (i=1;i<p;i++) {
          source = i;           /* MPI process number range is [0,p-1] */
          MPI_Recv(&my_result, 1, MPI_REAL, source, tag,
                       MPI_COMM_WORLD, &status);
          result += my_result;
        }
        printf("The result =%f\n",result);
      }
      else
        ierr= MPI_Send(&my_result, 1, MPI_REAL, dest, tag,
                      MPI_COMM_WORLD);      /* send my_result to intended dest. */
       if (ierr != MPI_SUCCESS) {
           MPI_Error_class(ierr,&errclass);
            if (errclass== MPI_ERR_RANK) {
```

```
             fprintf(stderr,"Invalid rank used in MPI send call\n");
            MPI_Error_string(ierr,err_buffer,&resultlen);
            fprintf(stderr,err_buffer);
            MPI_Finalize();             /* abort*/
            }
        }

      MPI_Finalize();                       /* let MPI finish up ... */
}
float integral(float a, int n, float h)
{
      int j;
      float h2, aij, integ;

      integ = 0.0;                 /* initialize integral */
      h2 = h/2.;
      for (j=0;j<n;j++) {             /* sum over all "j" integrals */
        aij = a + j*h;         /* lower limit of "j" integral */
        integ += fct(aij+h2)*h;
      }
      return (integ);
}
```

MPI Resources

Tutorials and On-line Classes

- NCSA On-line WebCT Intro to MPI and Intermediate Classes http://webct.ncsa.uiuc.edu:8900/webct/public/home.pl
- Argonne National Lab MPI Learning Page http://www-unix.mcs.anl.gov/mpi/learning.html
- LAM MPI Tutorials Page http://www.lam-mpi.org/tutorials/

Websites for Various Versions of MPI

- MPICH http://www-unix.mcs.anl.gov/mpi/mpich/
- LAM/MPI http://www.lam-mpi.org/
- OpenMPI http://www.open-mpi.org/

Table of Contents
