Introduction to MPI (Message-Passing Interface)

What is MPI?

- MPI is a library. It specifies the names, calling sequences and the results of functions to be called from C programs, subroutines to be called from Fortran programs, and the classes and methods that make up the MPI C++ library.
- MPI is a specification, not a particular implementation. A correct MPI program should be able to run on all MPI implementations without change.
- MPI is targeted towards the message passing model.
- MPI standards can be obtained from the website: `http://www.mpi-forum.org/`

Basic MPI Concepts

- **Messages and buffers**. *Sending* and *receiving* messages are the two fundamental operations. Messages can be typed with a *tag* integer. Allows message buffers to be more complex than a simple buffer and address combination by giving options to the user to create their own data types.
- **Separating Families of Messages**. MPI programs can use the notion of *contexts* to separate messages in different parts of the code. Useful for writing libraries. The context are allocated at run time by the system in response to user (or library) requests.
- **Process Groups**. Processes belong to *groups*. Each process is *ranked* in its group with a linear numbering. Initially, all processes belong to one default group.

Basic MPI Concepts (contd.)

- **Communicators**. The notions of context and group are combined in a single object called a *communicator*, which becomes a argument to most point-to-point and collective operations. Thus the *destination* or *source* specified in send or receive operation always refers to the rank of the process in the group identified by a communicator. For example, consider the following blocking send and blocking receive operations in MPI.

  ```
  MPI_Send(address, count, datatype, destination, tag, comm)
  MPI_Recv(address, maxcount, datatype, source, tag, comm, status)
  ```

  The status object in the receive holds information about the actual message size, source and tag.

Other MPI Features

- **Collective Communications**. MPI provides two types of collective operations, performed by all the processes in the computation.
  - *Data movement*: Broadcast, scattering, gathering and others.
  - *Collective computation*: Reduction operations like minimum, maximum, sum, logical OR etc as well as user-defined operations.
  - *Groups*: Operations of creating and managing groups in a scalable manner. These can be used to control the scope of the above collective operations.
- **Virtual Topologies**. Allows processes to be conceptualized in an application-oriented topology. Grids and general graphs are supported.

Other MPI Features

- **Debugging and profiling**. MPI requires the availability of "hooks" that allow users to intercept MPI calls and thus define their own debugging and profiling mechanisms.
- **Support for libraries**. Explicit support for writing libraries that are independent of user-code and inter-operable with other libraries. Libraries can maintain arbitrary data, called *attributes*, associated with the communicators they allocate and can specify their own error handlers.
- **Support for heterogeneous networks**. MPI programs can run on a heterogeneous network without the user having to worry about data type conversions.
- **Processes and processors**. The MPI specification uses processes only. Thus the mapping of processes to processors is up to the implementation.

The Six Basic MPI Functions

MPI_Init	Initialize MPI
MPI_Comm_size	Find out how many processes there are
MPI_Comm_rank	Find out which process I am
MPI_Send	Send a message
MPI_Recv	Receive a message
MPI_Finalize	Terminate MPI

MPI History and Versions

- MPI Version 1.2 is the last MPI-1 specification. MPI-2 is the latest official MPI standard.
- Freely available MPI implementations (with access to source code):
  - MPICH 1.2.x. The latest version of MPI 1.2 implementation available for download from Argonne National Lab (`http://www-unix.mcs.anl.gov/mpi/mpich1/`).
  - MPICH 2. Almost full implementation of MPI-2. Available for download from `http://www-unix.mcs.anl.gov/mpi/mpich2/`. On Fedora Linux install via
    `yum install mpich2*`
    We will use MPICH2 in this course.
  - LAM MPI. Implements all of MPI 1.2 and much of MPI-2. Available for download from `http://www.lam-mpi.org/`.
  - Open MPI. Almost full implementation of MPI-2. Available from `http://www.open-mpi.org/`.

Hello World in MPI

```
/*lab/MPI/hello_world/spmd_hello_world.c */
/* appropriate header files */
#include <asm/param.h> /* for MAXHOSTNAMELEN */
#include <mpi.h>

int main(int argc, char **argv)
{
    int pid;
    int nproc;
    char hostname[MAXHOSTNAMELEN];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);

    gethostname(hostname, 100);
    printf("Hello! I am %d of %d running on %s.\n", pid, nproc, hostname);

    MPI_Finalize();
    exit(0);
}
```

MPI Functions Introduced in the Example

- `int MPI_Init(int *argc, char ***argv)`
- `int MPI_Comm_size(MPI_Comm comm, int *size)`
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
- `int MPI_Finalize()`

Using MPICH2 in the Linux Lab

- Log in to the head node `onyx`. Acquire nodes from the scheduler:
  `pbsget -4`
- Run your mpi program with the `mpiexec` command.
  `mpiexec -n 4 hello_world`
- Exit and release all allocated nodes with the command:
  `exit`

Using MPICH2 on your computer

- **Setting up MPICH2**. Assuming that it was installed via yum, there is no further setup required.
- **Running a MPI program**
  - Simply run your mpi program with the `mpiexec` command.
    `mpiexec -n 4 hello_world`

Building MPICH2 from source on your computer

- Download the tarball of the software from http://www-unix.mcs.anl.gov/mpi/mpich2/. Unpack it somewhere, say in /usr/local/src with the command:

```
cd /usr/local/src
tar xzvf mpich2-xyz.tar.gz
```

- Assuming that you have Sun Java installed in /usr/local/java, I recommend the following steps to build MPICH2.

```
mkdir /usr/local/src/mpich2
./configure --prefix=/usr/local/mpich2 --enable-mpe --enable-cxx  \
--enable-romio --with-java-home=/usr/local/java/jre 2>&1 | tee config
make 2>&1 | tee make.log
make install 2>&1 | tee install.log
```

- For more details on installation, please read the instructions in the README file in the MPICH2 source.
- Add /usr/local/mpich2/bin to your PATH.
- Add /usr/local/mpich2/share/man in the /etc/man.config file to enable viewing man pages in KDE Konqueror.

MPI "Blocking" Send Call

- MPI "Blocking" send returns when "locally complete" – when the location used to hold the message can be safely altered without affecting the message being sent.
- ```
  int MPI_Send(void *buf, int count,
               MPI_Datatype datatype,
               int destination, int tag,
               MPI_Comm communicator);
  ```
- MPI_ANY_TAG is a wildcard value for tag.
  ```

MPI Blocking Recv Call

- MPI blocking recv returns when a message is received.
- ```
  int MPI_Recv(void *buf, int count,
               MPI_Datatype datatype,
               int source, int tag,
               MPI_Comm communicator, MPI_Status *status);
  ```
- `MPI_ANY_SOURCE` and `MPI_ANY_TAG` are wildcard values for destination and tag respectively.
  ```

Example Code Using Send and Recv

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank); /* find rank */
if (myrank == 0) {
    int x = 12345;
    MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
}
```

Let's Play Ping Pong

```
const int PROCESS_0 = 0;
const int PROCESS_1 = 1;
MPI_Comm_rank(MPI_COMM_WORLD,&myrank); /* find rank */
for (;;) {
    if (myrank == 0) {
        int x = 12345;
        MPI_Send(&x, 1, MPI_INT, PROCESS_1, msgtag, MPI_COMM_WORLD);
        MPI_Recv(&x, 1, MPI_INT, PROCESS_1, msgtag, MPI_COMM_WORLD, status)
    } else if (myrank == 1) {
        int x;
        MPI_Recv(&x, 1, MPI_INT, PROCESS_0, msgtag,MPI_COMM_WORLD, status);
        MPI_Send(&x, 1, MPI_INT, PROCESS_0, msgtag, MPI_COMM_WORLD);
    }
}
```

Example: Parallel Sum in MPI

For source code, see the folder MPI/parallel_sum. Four variations (assuming p processes):

- `spmd_sum_0.c`: This version sends the n numbers to all processes. Each process then adds its share and sends partial sum back to process 0, which then adds the p partial sums to get the total sum. *This program gives incorrect answers*.

Example: Parallel Sum in MPI

For source code, see the folder MPI/parallel_sum. Four variations (assuming p processes):

- `spmd_sum_0.c`: This version sends the n numbers to all processes. Each process then adds its share and sends partial sum back to process 0, which then adds the p partial sums to get the total sum. *This program gives incorrect answers*.
- `spmd_sum_1.c`: Corrected version of `spmd_sum_0.c`.

Example: Parallel Sum in MPI

For source code, see the folder MPI/parallel_sum. Four variations (assuming p processes):

- `spmd_sum_0.c`: This version sends the n numbers to all processes. Each process then adds its share and sends partial sum back to process 0, which then adds the p partial sums to get the total sum. *This program gives incorrect answers*.
- `spmd_sum_1.c`: Corrected version of `spmd_sum_0.c`.
- `spmd_sum_2.c`: Process 0 sends n/p elements (for p processes) to each process. Each process then adds its share and sends partial sum back to process 0, which then adds the p partial sums to get the total sum.

Example: Parallel Sum in MPI

For source code, see the folder MPI/parallel_sum. Four variations (assuming p processes):

- `spmd_sum_0.c`: This version sends the n numbers to all processes. Each process then adds its share and sends partial sum back to process 0, which then adds the p partial sums to get the total sum. *This program gives incorrect answers*.
- `spmd_sum_1.c`: Corrected version of `spmd_sum_0.c`.
- `spmd_sum_2.c`: Process 0 sends n/p elements (for p processes) to each process. Each process then adds its share and sends partial sum back to process 0, which then adds the p partial sums to get the total sum.
- `spmd_sum_3.c`: Assumes that each process had n/p elements to begin with. Each process then adds its share and sends partial sum back to process 0, which then adds the p partial sums to get the total sum.

Source Code for Parallel Sum in MPI

```
/* lab/MPI/parallel_sum/spmd_sum_mpi_0.c: */
/* appropriate header files */
#include <mpi.h>
const int PARTIAL_SUM = 1;

int add(int me, int n, int *data, int nproc)
{
    int i;

    int low = me *(n/nproc);
    int high = low +(n/nproc);
    int sum = 0;
    for(i=low; i<high; i++)
        sum += data[i];

    return(sum);
}
```

Example 2: Parallel Sum in MPI (spmd_sum_0.c) (contd)

```
int main(int argc, char **argv)
{
    int i;
    int total = 0;                /* total sum of data elements */
    int *partial_sums;
    int *data;
    int me, n, nproc;

    if (argc != 2) {
      fprintf(stderr,"Usage: %s  <number of elements>\n",argv[0]);
      exit(1);
    }
    n = atoi(argv[1]);      // number of data elements
    data = (int *) malloc(sizeof(int)*n);     // data array

    /* Start up MPI */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);

    if (me == 0) {
        /* Generate numbers to be summed up */
        for (i=0; i<n; i++)
            data[i] = 1;
    }
```

Example 2: Parallel Sum in MPI (contd)

```
    /* Broadcast initial data to other processes */
    if( MPI_Bcast( data, n, MPI_INT, 0,MPI_COMM_WORLD)!= MPI_SUCCESS)
             fprintf(stderr, "Oops! An error occured in MPI_Bcast()\n");

    int result = add(me, n, data, nproc);
    if (me == 0) {
            partial_sums = (int *)malloc(sizeof(int)*nproc);
        /* Process 0 gets its partial sum from local variable result */
        partial_sums[0] = result;
    } else {
        /* Other processes send partial sum to the process 0 */
        MPI_Send(&result, 1, MPI_INT, 0, PARTIAL_SUM, MPI_COMM_WORLD );
    }
    if (me == 0) {
        printf(" I got %d from %d\n", partial_sums[me], me);
        /* Wait for results from other processes */
        for (i=0; i<nproc-1; i++) {
            MPI_Recv(&partial_sums[i+1], 1, MPI_INT, i+1, PARTIAL_SUM,
                                            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("I got %d from %d\n", partial_sums[i+1], i+1);
        }
        /* Compute the global sum */
        for (i=0; i<nproc; i++)
            total += partial_sums[i];
        printf("The total is %d\n", total);
    }
    MPI_Finalize();
    exit(0);
}
```

MPI Functions Introduced in the Parallel Sum Example

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);`
- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);`
- `int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm);`
- `MPI_ANY_SOURCE` and `MPI_ANY_TAG` are wildcard values.
- `double MPI_Wtime()`. Time in seconds since some arbitrary point in the past.
- `double MPI_Wtick()`. Time in seconds between successive ticks of the clock.

Visualization and Logging

- Link with the libraries `-llmpe -lmpe` to enable logging and the MPE environment. Then run the program as usual and a log file will be produced.
- The log file can be visualized using the `jumpshot` program that comes bundled with MPICH2.

Token Ring Example

```c
/* See complete example in lab/MPI/token-ring/token_ring_mpi.c */
void pass_the_token(int me, int nproc)
{
  int token, src, dest;
  int count  = 1;
  const int TOKEN = 4;
  int msgtag = TOKEN;
  MPI_Status status;

  /* Determine neighbors in the ring */
  if (me == 0) src = nproc -1;
  else src = me - 1;

  if (me == nproc - 1) dest = 0;
  else dest = me + 1;

  if( me == 0 ) {
      token = dest;
      MPI_Send(&token,count, MPI_INT, dest, msgtag, MPI_COMM_WORLD);
      MPI_Recv(&token, count, MPI_INT, src, msgtag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("token ring done\n");
  } else {
      printf("received token ring on %d from %d \n",me, src);
      MPI_Recv(&token, count, MPI_INT, src, msgtag, MPI_COMM_WORLD, &status);
      MPI_Send(&token, count, MPI_INT,dest,msgtag,MPI_COMM_WORLD);
    }
}
```

Collective Operations, Communicators and Groups

- `int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);`
- `int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);`
- `int MPI_Comm_group(MPI_Comm comm, MPI_Group *group);`
- `int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group newgroup);`
- `int MPI_Group_free(MPI_Group *group);`
- `int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm);`
- `int MPI_Comm_free(MPI_Comm *comm);`

Calculating π using Monte Carlo Simulation

- The following code example calculates π to a specified precision using a Monte Carlo simulation.
- This example illustrates the use of MPI groups and communicators. We designate one process to be the random number generator. All other processes form a separate group and communicator to do the simulation.
- The example also illustrates the use of the `MPI_AllReduce` function.

Monte Carlo Calculation of π

```
/* compute pi using Monte Carlo method */
/* appropriate header files */
#include <mpi.h>
#include <mpe.h>
#define CHUNKSIZE      1000
#define INT_MAX RAND_MAX
/* message tags */
#define REQUEST  1
#define REPLY    2
int main( int argc, char *argv[] )
{
    int iter, in, out, i, max, ranks[1], done;
    double x, y, Pi, error, epsilon;
    int numprocs, myid, server, totalin, totalout, workerid;
    int rands[CHUNKSIZE], request;
    MPI_Comm world, workers;
    MPI_Group world_group, worker_group;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    world  = MPI_COMM_WORLD;
    MPI_Comm_size(world,&numprocs);
    MPI_Comm_rank(world,&myid);
    server = numprocs-1;    /* last proc is server */
    if (myid == 0)
        sscanf( argv[1], "%lf", &epsilon );
```

Monte Carlo Calculation of π (contd)

```
MPI_Bcast( &epsilon, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD );
MPI_Comm_group( world, &world_group );
ranks[0] = server;
MPI_Group_excl( world_group, 1, ranks, &worker_group );
MPI_Comm_create( world, worker_group, &workers );
MPI_Group_free(&worker_group);
if (myid == server) {   /* I am the random number server */
    do {
        MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE, REQUEST,
         world, &status);
        if (request) {
        for (i = 0; i < CHUNKSIZE; ) {
            rands[i] = random();
            if (rands[i] <= INT_MAX) i++;
        }
        MPI_Send(rands, CHUNKSIZE, MPI_INT,
                     status.MPI_SOURCE, REPLY, world);
        }
    }
    while( request>0 );
```

Monte Carlo Calculation of π (contd)

```
    } else {            /* I am a worker process */
        request = 1;
        done = in = out = 0;
        max  = INT_MAX;         /* max int, for normalization */
        MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
        MPI_Comm_rank( workers, &workerid );
        iter = 0;
...
```

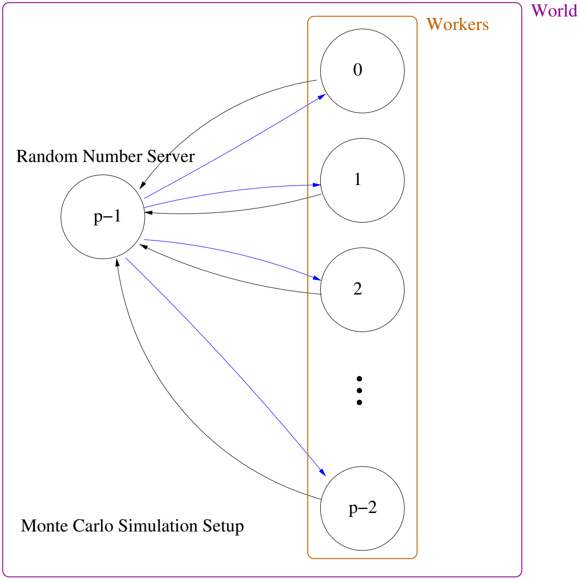
Monte Carlo Calculation of π (contd)

```
    while (!done) {
        iter++;
        request = 1;
        MPI_Recv(rands, CHUNKSIZE, MPI_INT, server, REPLY,
            world, &status);
        for (i=0; i<CHUNKSIZE-1; ) {
            x = (((double) rands[i++])/max) * 2 - 1;
            y = (((double) rands[i++])/max) * 2 - 1;
            if (x*x + y*y < 1.0)
                in++;
            else
                out++;
            }
            MPI_Allreduce(&in, &totalin, 1, MPI_INT, MPI_SUM, workers);
            MPI_Allreduce(&out, &totalout, 1, MPI_INT, MPI_SUM, workers);
            Pi = (4.0*totalin)/(totalin + totalout);
            error = fabs( Pi-3.141592653589793238462643);
            done = (error < epsilon || (totalin+totalout) > 1000000);
            request = (done) ? 0 : 1;
            if (myid == 0) {
                printf( "\rpi = %23.20f", Pi );
                MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
            } else {
                if (request)
                    MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
            }
    }
    MPI_Comm_free(&workers);
}
```

Monte Carlo Calculation of π (contd)

```
    if (myid == 0) {
        printf( "\npoints: %d\nin: %d, out: %d, <ret> to exit\n",
           totalin+totalout, totalin, totalout );
        getchar();
    }
    MPI_Finalize();
    exit(0);
}
```

Processes in Monte Carlo Simulation



A better way of splitting Communicators

```
int MPI_Comm_split(MPI_Comm oldcomm, int color, int key, MPI_Comm newcomm);
```

Using the above function we can split the communicator in the Monte Carlo example as follows:

```
color = (myid == server);
MPI_Comm_split(world, color, 0, &workers);
```

MPE Graphics

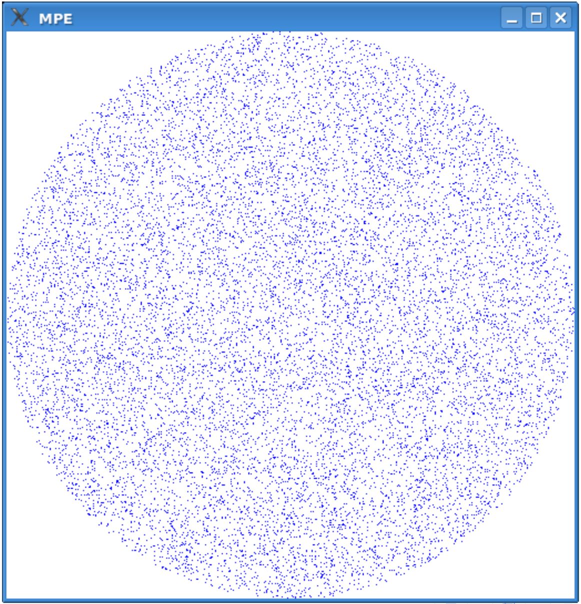
- Simple graphics capability is built in the MPE environment that comes with MPI.
- Allows users to draw points, lines, circles, rectangles as well manipulate colors.
- Need to link in `-lmpe` and `-lX11` libraries.

Monte Carlo Simulation with Graphics

The following shows the new code that was added to the previous example. Also we need to add -lX11 linker option to link in the X Windows graphics library.

```
#include <mpe_graphics.h>

MPE_XGraph graph;

MPE_Open_graphics(&graph, MPI_COMM_WORLD, (char *)0, -1, -1,
                    WINDOW_SIZE, WINDOW_SIZE, MPE_GRAPH_INDEPENDENT);

MPE_Draw_point(graph, (int) (WINDOW_SIZE/2 + x * WINDOW_SIZE/2),
       (int) (WINDOW_SIZE/2 + y * WINDOW_SIZE/2), MPE_BLUE);


MPE_Update(graph);

MPE_Close_graphics(&graph);
```

Screenshot of Monte Carlo in Action



References

- *Using MPI: Portable Parallel Programming with the Message-Passing Interface (2nd edition)* by William Group, Ewing Lusk and Anthony Skjellum. The MIT press.
- *Using MPI-2: Advanced Features of the Message-Passing Interface* by William Group, Ewing Lusk and Rajeev Thakur.
- *MPI-The Complete Reference: Volume 1, The MPI Core* by Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra.
- *MPI-The Complete Reference: Volume 2, The MPI Extensions* by William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewibg Lusk, Bill Nitzberg, William Saphir and Marc Snir. The MIT press.

MPI Basic Calls Summary

```
int MPI_Init(int *argc, char ***argv)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Finalize()
int MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag,
             MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype type, int src, int tag,
             MPI_Comm comm, MPI_Status *status)
int MPI_Bcast(void *buf, int count, MPI_Datatype type, int root, MPI_Comm comm)
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype type,
             MPI_Op op, int root, MPI_Comm comm)
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype type,
             MPI_Op op, MPI_Comm comm)
double MPI_Wtime()
double MPI_Wtick()
```