Message Passing Interface (MPI)

- MPI, the *Message Passing Interface*, is a library, and a software standard developed by the MPI Forum to make use of the most attractive features of existing message passing systems for parallel programming.
- The public release of version 1.0 of MPI was made in June 1994 followed by version 1.1 in June 1995 and and shortly after with version 1.2 which mainly included corrections and specification clarifications.
- An MPI 1 process consists of a C or Fortran 77 program which communicates with other MPI processes by calling MPI routines. The MPI routines provide the programmer with a consistent interface across a wide variety of different platforms.
- Version 2.0, a major update of MPI, was released July 1997 adding among other features support for dynamic process creation, one-sided communication and bindings for Fortran 90 and C++. MPI 2.0 features are not covered here.
- Several commercial and free implementations of MPI 1.2 exist Most widely used free implementations of MPI 1.2 :
  - LAM-MPI : Developed at University of Notre Dame , http://www.lam-mpi.org/
  - MPI-CH: Developed at Argonne National Laboratory

    http://www-unix.mcs.anl.gov/mpi/mpich/

    - (MPI-CH 1.2.3 is the MPI implementation installed on the CE cluster).

Major Features of MPI 1.2

Standard includes 125 functions to provide:

- Point-to-point message passing
- Collective communication
- Support for process groups
- Support for communication contexts
- Support for application topologies
- Environmental inquiry routines
- Profiling interface

Compiling and Running MPI Programs

- **To compile MPI C programs use:**

```
mpicc  [linking flags] program_name.c  -o  program_name
```

Ex:
```
mpicc  hello.c  -o  hello
```

- **To run an MPI compiled program use:**

```
mpirun   -np <number of processes>  [mpirun_options]
-machinefile  < machinefile>  <program name and arguments>
```

The machinefile contains a list of the machines on which you want your MPI programs to run.

EX:
```
mpirun -np  4  -machinefile .rhosts   hello
```

starts four processes on the top four machines from machinefile .rhosts all running the program hello

MPI Static Process Creation & Process Rank

- MPI 1.2 does not support dynamic process creation, i.e. one cannot spawn a process from within a process as can be done with pvm:
  - All processes must be started together at the beginning of the computation. Ex: mpirun -np 4 -machinefile .rhosts hello
  - There is no equivalent to the pvm **pvm_spawn( )** call.
  - This restriction leads to the direct support of MPI for single program-multiple data (SPMD) model of computation where each process has the same executable code.
- **MPI process Rank:** A number between 0 to N-1 identifying the process where N is the total number of MPI processes involved in the computation.
  - The MPI function **MPI_Comm_rank** reports the rank of the calling process.
  - The MPI function **MPI_Comm_size** reports the total number of MPI processes.

MPI Process Initialization & Clean-Up

- The first MPI routine called in any MPI program *must* be the initialization routine **MPI_INIT**. Every MPI program must call this routine *once*, before any other MPI routines.
- An MPI program should call the MPI routine **MPI_FINALIZE** when all communications have completed. This routine cleans up all MPI data structures etc.
- MPI_FINALIZE does NOT cancel outstanding communications, so it is the responsibility of the programmer to make sure all communications have completed.
  - Once this routine is called, no other calls can be made to MPI routines, not even MPI_INIT, so a process cannot later re-enroll in MPI.

MPI Communicators, Handles

- MPI_INIT defines a default communicator called MPI_COMM_WORLD for each process that calls it.
- All MPI communication calls require a communicator argument and MPI processes can only communicate if they share a communicator.
- Every communicator contains a *group* which is a list of processes. The processes are ordered and numbered consecutively from zero, the number of each process being its *rank.* The rank identifies each process within the communicator.
- The group of MPI_COMM_WORLD is the set of all MPI processes.
- MPI maintains internal data structures related to communications etc. and these are referenced by the user through *handles*. Handles are returned to the user from some MPI calls and can be used in other MPI calls.

MPI Datatypes

- The data in a message to sent or received is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - Predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
  - A contiguous array of MPI datatypes
  - A strided block of datatypes
  - an indexed array of blocks of datatypes
  - An arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, such an array of (int, float) pairs, or a row of a matrix stored columnwise.

MPI Indispensable Functions

- MPI_Init - Initialize MPI
- MPI_Finalize - Terminate 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_Bcast -   Broadcasts data
- MPI_Reduce - Combines values into a single value

MPI_Init , MPI_Finalize

MPI_Init

- The call to MPI_Init is required in every MPI program and must be the first MPI call. It establishes the MPI execution environment.

  int MPI_Init(int *argc, char ***argv)

  Input:

  argc - Pointer to the number of arguments

  argv - Pointer to the argument vector

MPI_Finalize

- This routine terminates the MPI execution environment; all processes must call this routine before exiting.

  int MPI_Finalize(**void)**

MPI_Comm_size

- This routine determines the size (i.e., number of processes) of the group associated with the communicator given as an argument.

  int MPI_Comm_size(MPI_Comm comm, int *size)

  Input:

      comm - communicator (handle)

  Ouput:

      size - number of processes in the group of comm

EX: MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

MPI_Comm_rank

- The routine determines the rank (i.e., which process number am I?) of the calling process in the communicator.

  int MPI_Comm_rank(MPI_Comm comm, int *rank)

  Input:

  comm - communicator (handle)

  Output:

  rank - rank of the calling process in the group of comm (integer)

EX: MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

Simple MPI C Program Hello.c

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Multiple Program Multiple Data MPMD in MPI

- Static process creation in MPI where each process has the same executable code leads to the direct support of MPI for single program-multiple data (SPMD).
- In general MPMD and master-slave models are supported by utilizing the rank of each process to execute different portions of the code.
- EX:

```
main (int argc, char **argv)
{
     MPI_Init(&argc, &argc);                              /* initialize MPI */
         .  .  .
     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);              /* find process rank
    */
     if (myrank==0)
    master();
     else
    slave();
         .  .  .
     MPI_Finalize();
}
```

where **master( )** and **slave( )** are procedures in the main program to be executed by the master process and slave process respectively.

Variables Within Individual MPI Processes

- Given the SPMD model, any global declarations of variables will be duplicated in each process. Variables and data that is not to be duplicated will need to be declared locally, that is, declared within code only executed by process, for example as in:

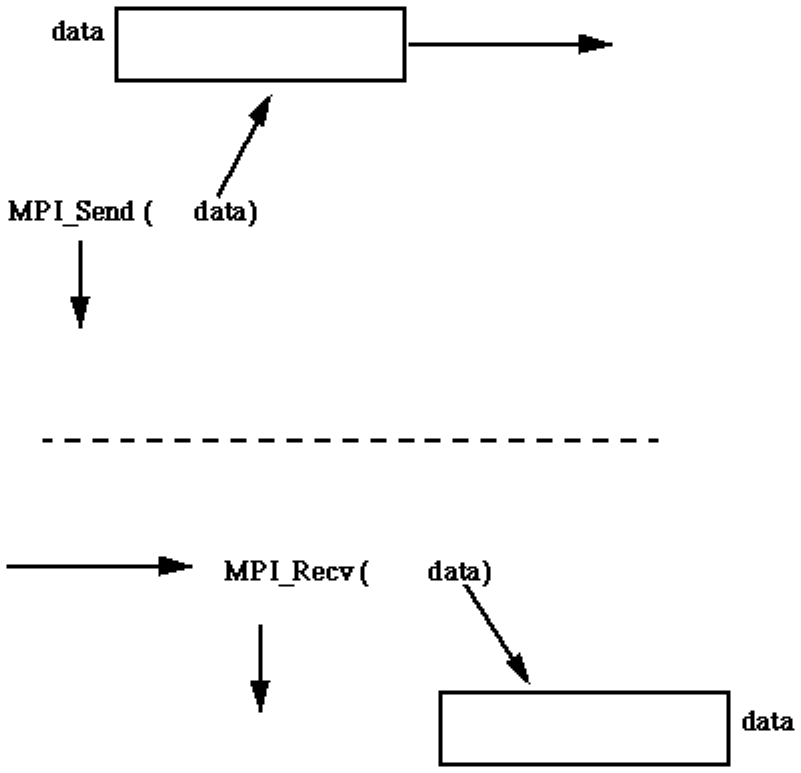
```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find process rank */
if(myrank==0) {
    int x,y, data[100];
   .  .
} else {
    int x,y;
   .   .
}
```

- Here **x** and **y** in process 0 are different local variables to **x** and **y** in process 1. The array, **data[100]**, might be for example some data which is to be sent from the master to the slaves.

Blocking Send/Receive Routines

- Blocking routines (send or receive) return when they are locally complete.
- In the context of a blocking send routine (**MPI_Send( )**) the local conditions are that the location being used to hold the data being sent can be altered, and blocking send will send the message and return then.
- This does not mean that the message has been received, just that the process is free to move on without adversely affecting the message being sent.
- A blocking receive routine (**MPI_Recv( )**) will also return when it is locally complete, which in this case means that the message has been received into the destination location and the destination location can be read.

Blocking Send/Receive Routines



MPI_Send (Basic or Blocking Send)

- This routine performs a basic (blocking) send; this routine may block and returns when the send is locally complete

  int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

  Input:
  buf - initial address of send buffer (choice)
  count - number of elements in send buffer (nonnegative integer)
  datatype - datatype of each send buffer element (handle)
  dest - rank of destination (integer)
  tag - message tag (integer)
  comm - communicator (handle)

- **MPI_Recv** (Basic or Blocking Receive)
- This routine performs a basic blocking receive.

  int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Output:

buf - initial address of receive buffer

status - status object, provides information about message received; status is a structure of type MPI_Status, the element status.MPI_SOURCE is the source of the message received, and the element status.MPI_TAG is the tag value.

Input:

count - maximum number of elements in receive buffer (integer)

datatype - datatype of each receive buffer element (handle)

source - rank of source (integer)

tag - message tag (integer)

comm - communicator (handle)

MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message.
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **MPI_ANY_TAG** as the tag in a receive.
- Similarly, messages can be screened at the receiving end by specifying a specific source, or not screened by specifying **MPI_ANY_SOURCE** as the source in a receive.

Blocking Send/Receive Code Example

To send an integer x from process 0 to process 1:

```
   .
   .
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find process
   rank */
if (myrank==0) {
    int x;
    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);
   .
   .
```

Nonblocking Send/Receive Routines

- Nonblocking routines return immediately whether or not they are locally (or globally) complete.
- The nonblocking send, **MPI_Isend()**, where **I** refers to the word "immediate", will return even before the source location is safe to be altered.
- The nonblocking receive, **MPI_Irecv()**, will return even if there is no message to accept.
- The nonblocking send, **MPI_Isend()**, corresponds to the pvm routine **pvm_send()** and the nonblocking receive, **MPI_Irecv()**, corresponds to the pvm routine **pvm_nrecv()**. The formats are:

  MPI_Isend(buf, count, datatype, dest, tag, comm, request)

  MPI_Irecv(buf, count, datatype, source, tag, comm, request)
- The additional parameter, **request**, is used when it is necessary to know that the operation has actually completed.
- Completion can be detected by separate routines, **MPI_Wait( )** and **MPI_Test( )**. **MPI_Wait( )** returns if the operation has actually completed and will wait until completion occurs. **MPI_Test( )** returns immediately with a flag set indicating whether the operation has completed at this time. These routines need the identity of the operation, in this context the nonblocking message passing routines which is obtained from request.
- Nonblocking routines provide the facility to overlap communication and computation which is essential when communication delays are high.

Nonblocking Send Code Example

- To send an integer x from process 0 to process 1 and allow process 0 to continue immediately:

```
   .
   .
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find process
   rank */
if (myrank==0) {
    int x;
    MPI_ISend(x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD,
   req1);
    compute();
    MPI_Wait(req1, status);
} else if (myrank==1) {
    int x;
    MPI_Recv(x, 0, MPI_INT, 1, msgtag, MPI_COMM_WORLD,
   status);
   .
```

MPI_Bcast (Data Broadcast)

- This routine broadcasts data from the process with rank "root" to all other processes of the group.

int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Input/Output:

buffer - starting address of buffer (choice)
count - number of entries in buffer (integer)
datatype - data type of buffer (handle)
root - rank of broadcast root (integer)
comm - communicator (handle)

MPI_Reduce

- This routine combines values on all processes into a single value using the operation defined by the parameter op.

  int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

  Input:
  sendbuf - address of send buffer (choice)
  count - number of elements in send buffer (integer)
  datatype - data type of elements of send buffer (handle)
  op - reduce operation (handle) (user can create using MPI_Op_create or use predefined operations MPI_MAX, MPI_MIN, MPI_PROD, MPI_SUM, MPI_LAND, MPI_LOR, MPI_LXOR, MPI_BAND, MPI_BOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC in place of MPI_Op op.
  root - rank of root process (integer)
  comm - communicator (handle)

  Output:
  recvbuf - address of receive buffer (choice, significant only at root)

MPI Collective Communications Functions

The principal collective operations are:

MPI_Barrier( )	/* Blocks process until all processes have called it */
• MPI_Bcast( )	/* Broadcast from root to all other processes */
• MPI_Gather( )	/* Gather values for group of processes */
• MPI_Scatter( )	/* Scatters a buffer in parts to group of processes */
• MPI_Alltoall( )	/* Sends data from all processes to all processes */
• MPI_Reduce( )	/* Combine values on all processes to single value */
• MPI_Reduce_scatter( )	/* Combine values and scatter results */
• MPI_Scan( )	/* Compute prefix reductions of data on processes */

Partitioned Sum MPI Program Example: sum.c (part 1 of 3)

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

#define MAXSIZE 1000

int add(int *A, int low, int high)
{
        int res, i;

        for(i=low; i<high; i++)
                res += A[i];

        return(res);
}

void main(argc,argv)
int argc;
char *argv[];
{
   int done = 0, n, myid, numprocs;
   int data[MAXSIZE], i, low, high, myres, res;
   char fn[255];
   char *fp;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

Partitioned Sum Program Example: sum.c (part 2 of 3)

```
n = 0;
 while (!done)
 {
    if (myid == 0)
    {
       if (n==0) n=100; else n=0;

       strcpy(fn,getenv("HOME"));
       strcat(fn,"/MPI/rand_data.txt");

       /* Open Input File and Initialize Data */
       if ((fp = fopen(fn,"r")) == NULL)
       {
              printf("Can't open the input file: %s\n\n", fn);
              exit(1);
       }

       for(i=0; i<MAXSIZE; i++)
              fscanf(fp,"%d", &data[i]);
   }

    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD);
```

Partitioned Sum MPI Program Example: sum.c (part 3 of 3)

```
if (n == 0)
          done = 1;
       else
       {
       low = myid * 100;
       high = low + 100;

       myres = add(data, low, high);
       printf("I got %d from %d\n", myres, myid);
       MPI_Reduce(&myres, &res, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
 if (myid == 0)
printf("The sum is %d.\n", res);
}
}
  MPI_Finalize();
  }
```

MPI Program Example To Compute Pi

Part 1 of 2

The program computes Pi by computing the area under the curve $f(x) = 4/(1+x^2)$ between 0 and 1.

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.141592653589793238462643;
   double mypi, pi, h, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   while (!done)  {
     if (myid == 0) {
       printf("Enter the number of intervals: (0 quits) ");
       scanf("%d",&n);
     }
     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
     if (n == 0) break;
```

MPI Program Example To Compute Pi

Part 2 of 2

```
    h   = 1.0 / (double) n;
     sum = 0.0;
     for (i = myid + 1; i <= n; i += numprocs) {
       x = h * ((double)i - 0.5);
       sum += 4.0 / (1.0 + x*x);
     }
     mypi = h * sum;
     MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                 MPI_COMM_WORLD);
     if (myid == 0)
       printf("pi is approximately %.16f, Error is %.16f\n",
               pi, fabs(pi - PI25DT));
   }
   MPI_Finalize();
  return 0;
}
```