Module 11: An introduction to the Message Passing Interface (MPI) using C

This is a short introduction to the Message Passing Interface (MPI) designed to convey the fundamental operation and use of the interface. This introduction is designed for readers with some background programming C, and should deliver enough information to allow readers to write and run their own (very simple) parallel C programs using MPI.

There exists a version of this tutorial for Fortran programers called Introduction the the Message Passing Interface (MPI) using Fortran.

What is MPI?

MPI is a library of routines that can be used to create parallel programs in C or Fortran77. Standard C and Fortran include no constructs supporting parallelism so vendors have developed a variety of extensions to allow users of those languages to build parallel applications. The result has been a spate of non-portable applications, and a need to retrain programmers for each platform upon which they work.
The MPI standard was developed to ameliorate these problems. It is a library that runs with standard C or Fortran programs, using commonly-available operating system services to create parallel processes and exchange information among these processes.

MPI is designed to allow users to create programs that can run efficiently on most parallel architectures. The design process included vendors (such as IBM, Intel, TMC, Cray, Convex, etc.), parallel library authors (involved in the development of PVM, Linda, etc.), and applications specialists. The final version for the draft standard became available in May of 1994.

MPI can also support distributed program execution on heterogenous hardware. That is, you may run a program that starts processes on multiple computer systems to work on the same problem. This is useful with a workstation farm.

Hello world

Here is the basic Hello world program in C using MPI:

```
   #include <stdio.h>
   #include <mpi.h>
   main(int argc, char **argv)
   {
      int ierr;

      ierr = MPI_Init(&argc, &argv);
      printf("Hello world\n");

      ierr = MPI_Finalize();
   }
```

If you compile hello.c with a command like

```
mpicc hello.c -o hello
```

you will create an executable file called hello, which you can execute by using the `mpirun` command as in the following session segment:

```
$ mpirun -np 4 hello
Hello world
Hello world
Hello world
Hello world
$
```

When the program starts, it consists of only one process, sometimes called the "parent", "root", or "master" process. When the routine MPI_Init executes within the root process, it causes the creation of 3 additional processes (to reach the number of processes (np) specified on the `mpirun` command line), sometimes called "child" processes.

Each of the processes then continues executing separate versions of the hello world program. The next statement in every program is the printf statement, and each process prints "Hello world" as directed. Since terminal output from every program will be directed to the same terminal, we see four lines saying "Hello world".

Identifying the separate processes

As written, we cannot tell which "Hello world" line was printed by which process. To identify a process we need some sort of process ID and a routine that lets a process find its own process ID. MPI assigns an integer to each process beginning with 0 for the parent process and incrementing each time a new process is created. A process ID is also called its "rank".
MPI also provides routines that let the process determine its process ID, as well as the number of processes that are have been created.

Here is an enhanced version of the Hello world program that identifies the process that writes each line of output:

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

   main(int argc, char **argv)
   {
      int ierr, num_procs, my_id;

      ierr = MPI_Init(&argc, &argv);

      /* find out MY process ID, and how many processes
were started. */

      ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
      ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

      printf("Hello world! I'm process %i out of %i
processes\n",
         my_id, num_procs);

      ierr = MPI_Finalize();
   }
```

When we run this program, each process identifies itself:

```
$ mpicc hello2.c -o hello2

$ mpirun -np 4 hello2
Hello world! I'm process 0 out of 4 processes.
Hello world! I'm process 2 out of 4 processes.
Hello world! I'm process 1 out of 4 processes.
Hello world! I'm process 3 out of 4 processes.
$
```

Note that the process numbers are not printed in ascending order. That is because the processes execute independently and execution order was not controlled in any way. The programs may print their results in different orders each time they are run.

(To find out which Origin processors and memories are used to run a program you can turn on the MPI_DSM_VERBOSE environment variable with "export MPI_DSM_VERBOSE=ON", or equivalent.)

To let each process perform a different task, you can use a program structure like:

```
   #include <mpi.h>

   main(int argc, char **argv)
   {
      int my_id, root_process, ierr, num_procs;
      MPI_Status status;

      /* Create child processes, each of which has its own
variables.
       * From this point on, every process executes a
separate copy
       * of this program.  Each process has a different
process ID,
       * ranging from 0 to num_procs minus 1, and COPIES of
all
       * variables defined in the program. No variables are
shared.
       **/

      ierr = MPI_Init(&argc, &argv);

      /* find out MY process ID, and how many processes
were started. */

      ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
      ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

      if( my_id == 0 ) {

         /* do some work as process 0 */
      }
      else if( my_id == 1 ) {

         /* do some work as process 1 */
      }
      else if( my_id == 2 ) {

         /* do some work as process 2 */
      }
      else {

         /* do this work in any remaining processes */
      }
      /* Stop this process */

      ierr = MPI_Finalize();
   }
```

Basic MPI communication routines

It is important to realize that separate processes share no memory variables. They appear to be using the same variables, but they are really using COPIES of any variable defined in the program.
As a result, these programs cannot communicate with each other by exchanging information in memory variables. Instead they may use any of a large number of MPI communication routines. The two basic routines are:

- MPI_Send, to send a message to another process, and
- MPI_Recv, to receive a message from another process.

The syntax of MPI_Send is:

```
int MPI_Send(void *data_to_send, int send_count,
MPI_Datatype send_type,
      int destination_ID, int tag, MPI_Comm comm);
```

- `data_to_send`: variable of a C type that corresponds to the `send_type` supplied below
- `send_count`: number of data elements to be sent (nonnegative int)
- `send_type`: datatype of the data to be sent (one of the MPI datatype handles)
- `destination_ID`: process ID of destination (int)
- `tag`: message tag (int)
- `comm`: communicator (handle)

Once a program calls MPI_Send, it blocks until the data transfer has taken place and the `data_to_send` variable can be safely reused. As a result, these routines provide a simple synchronization service along with data exchange.
The syntax of MPI_Recv is:

```
int MPI_Recv(void *received_data, int receive_count,
MPI_Datatype receive_type,
      int sender_ID, int tag, MPI_Comm comm, MPI_Status
*status);
```

- received_data: variable of a C type that corresponds to the receive_type supplied below
- receive_count: number of data elements expected (int)
- receive_type: datatype of the data to be received (one of the MPI datatype handles)
- sender_ID: process ID of the sending process (int)
- tag: message tag (int)
- comm: communicator (handle)
- status: status struct (MPI_Status)

The receive_count, sender_ID, and tag values may be specified so as to allow messages of unknown length, from several sources (MPI_ANY_SOURCE), or with various tag values (MPI_ANY_TAG).

The amount of information actually received can then be retrieved from the status variable, as with:

```
count MPI_Get_count(&status, MPI_FLOAT,
&true_received_count);
received_source = status.MPI_SOURCE;
received_tag = status.MPI_TAG;
```

MPI_Recv blocks until the data transfer is complete and the received_data variable is available for use.

The basic datatypes recognized by MPI are:

MPI datatype handle	**C datatype**
MPI_INT	int
MPI_SHORT	short
MPI_LONG	long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_CHAR	char
MPI_BYTE	unsigned char
MPI_PACKED	

There also exist other types like: MPI_UNSIGNED, MPI_UNSIGNED_LONG, and MPI_LONG_DOUBLE.

A common pattern of process interaction

A common pattern of interaction among parallel processes is for one, the master, to allocate work to a set of slave processes and collect results from the slaves to synthesize a final result.

The master process will execute program statements like:

```
   /* distribute portions of array1 to slaves. */

   for(an_id = 1; an_id < num_procs; an_id++) {

      start_row = an_id*num_rows_per_process;

      ierr = MPI_Send( &num_rows_to_send, 1, MPI_INT,
            an_id, send_data_tag, MPI_COMM_WORLD);

      ierr = MPI_Send( &array1[start_row],
num_rows_per_process,
            MPI_FLOAT, an_id, send_data_tag,
MPI_COMM_WORLD);
   }

   /* and, then collect the results from the slave
processes,
    * here in a variable called array2, and do something
with them. */

   for(an_id = 1 an_id < num_procs; an_id++) {

      ierr = MPI_Recv( &array2, num_rows_returned,
MPI_FLOAT,
            MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);

      /* do something with array2 here */

   }
   /* and then print out some final result using the
    * information collected from the slaves. */
```

In this fragment, the master program sends a contiguous portion of array1 to each slave using MPI_Send and then receives a response from each slave via MPI_Recv. In practice, the master does not have to send an array; it could send a scalar or some other MPI data type, and it could construct array1 from any components to which it has access.

Here the returned information is put in array2, which will be written over every time a different message is received. Therefore, it will probably be copied to some other variable within the receiving loop.

Note the use of the MPI constant MPI_ANY_SOURCE to allow this MPI_Recv call to receive messages from any process. In some cases, a program would need to determine exactly which process sent a message received using MPI_ANY_SOURCE. status.MPI_SOURCE will hold that information, immediately following the call to MPI_Recv.

The slave program to work with this master would resemble:

```
   /* Receive an array segment, here called array2 */.

   ierr = MPI_Recv( &num_rows_to_receive, 1 , MPI_INT,
         root_process, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);

   ierr = MPI_Recv( &array2, num_rows_to_receive,
MPI_FLOAT,
         root_process, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);

   /* Do something with array2 here, placing the result in
array3,
    * and send array3 to the root process. */

   ierr = MPI_Send( &array3, num_rows_to_return, MPI_FLOAT,
         root_process, return_data_tag, MPI_COMM_WORLD);
```

There could be many slave programs running at the same time. Each one would receive data in array2 from the master via MPI_Recv and work on its own copy of that data. Each slave would construct its own copy of array3, which it would then send to the master using MPI_Send.

A non-parallel program that sums the values in an array

The following program calculates the sum of the elements of a array. It will be followed by a parallel version of the same program using MPI calls.

```
   #include <stdio.h>
   #define max_rows 10000000

   int array[max_rows];

   main(int argc, char **argv)
   {
      int i, num_rows;
      long int sum;

      printf("please enter the number of numbers to sum:
");
      scanf("%i", &num_rows);

      if(num_rows > max_rows) {
         printf("Too many numbers.\n");
         exit(1);
      }

      /* initialize an array */

      for(i = 0; i < num_rows; i++) {
         array[i] = i;
      }

      /* compute sum */

      sum = 0;
      for(i = 0; i < num_rows; i++) {
         sum += array[i];
      }

      printf("The grand total is: %i\n", sum);
   }
```

Design for a parallel program to sum an array

The code below shows a common Fortran structure for including both master and slave segments in the parallel version of the example program just presented. It is composed of a short set-up section followed by a single `if...else` loop where the master process executes the statments between the brackets after the `if` statement, and the slave processes execute the statements between the brackets after the `else` statement.

```
   /* This program sums all rows in an array using MPI
parallelism.
    * The root process acts as a master and sends a portion
of the
    * array to each child process.  Master and child
processes then
    * all calculate a partial sum of the portion of the
array assigned
    * to them, and the child processes send their partial
sums to
    * the master, who calculates a grand total.
    **/

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

   int my_id, root_process, ierr, num_procs, an_id;
   MPI_Status status;

   root_process = 0;

   /* Now replicate this process to create parallel
processes.

   ierr = MPI_Init(&argc, &argv);

   /* find out MY process ID, and how many processes were
started */

   ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
   ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

   if(my_id == root_process) {
      /* I must be the root process, so I will query the
user
       * to determine how many numbers to sum.

       * initialize an array,

       * distribute a portion of the array to each child
process,

       * and calculate the sum of the values in the segment
assigned
       * to the root process,

       * and, finally, I collect the partial sums from
slave processes,
       * print them, and add them to the grand sum, and
print it */
   }

   else {

      /* I must be slave process, so I must receive my
array segment,

       * calculate the sum of my portion of the array,

       * and, finally, send my portion of the sum to the
root process. */

   }

   /* Stop this process */

   ierr = MPI_Finalize();
}
```

The complete parallel program to sum a array

Here is the expanded parallel version of the same program using MPI calls.

```
   /* This program sums all rows in an array using MPI
parallelism.
    * The root process acts as a master and sends a portion
of the
    * array to each child process.  Master and child
processes then
    * all calculate a partial sum of the portion of the
array assigned
    * to them, and the child processes send their partial
sums to
    * the master, who calculates a grand total.
    **/

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

   #define max_rows 100000
   #define send_data_tag 2001
   #define return_data_tag 2002

   int array[max_rows];
   int array2[max_rows];

   main(int argc, char **argv)
   {
      long int sum, partial_sum;
      MPI_Status status;
      int my_id, root_process, ierr, i, num_rows,
num_procs,
         an_id, num_rows_to_receive, avg_rows_per_process,
         sender, num_rows_received, start_row, end_row,
num_rows_to_send;

      /* Now replicte this process to create parallel
processes.
       * From this point on, every process executes a
seperate copy
```

```
       * of this program */

      ierr = MPI_Init(&argc, &argv);

      root_process = 0;

      /* find out MY process ID, and how many processes
were started. */

      ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
      ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

      if(my_id == root_process) {

         /* I must be the root process, so I will query the
user
          * to determine how many numbers to sum. */

         printf("please enter the number of numbers to sum:
");
         scanf("%i", &num_rows);

         if(num_rows > max_rows) {
            printf("Too many numbers.\n");
            exit(1);
         }

         avg_rows_per_process = num_rows / num_procs;

         /* initialize an array */

         for(i = 0; i < num_rows; i++) {
            array[i] = i + 1;
         }

         /* distribute a portion of the bector to each
child process */

         for(an_id = 1; an_id < num_procs; an_id++) {
            start_row = an_id*avg_rows_per_process + 1;
            end_row   = (an_id + 1)*avg_rows_per_process;

            if((num_rows - end_row) < avg_rows_per_process)
               end_row = num_rows - 1;

            num_rows_to_send = end_row - start_row + 1;

            ierr = MPI_Send( &num_rows_to_send, 1 ,
MPI_INT,
                  an_id, send_data_tag, MPI_COMM_WORLD);

            ierr = MPI_Send( &array[start_row],
num_rows_to_send, MPI_INT,
                  an_id, send_data_tag, MPI_COMM_WORLD);
         }

         /* and calculate the sum of the values in the
segment assigned
          * to the root process */

         sum = 0;
         for(i = 0; i < avg_rows_per_process + 1; i++) {
            sum += array[i];
         }

         printf("sum %i calculated by root process\n",
sum);

         /* and, finally, I collet the partial sums from
the slave processes,
          * print them, and add them to the grand sum, and
print it */

         for(an_id = 1; an_id < num_procs; an_id++) {

            ierr = MPI_Recv( &partial_sum, 1, MPI_LONG,
MPI_ANY_SOURCE,
                  return_data_tag, MPI_COMM_WORLD,
&status);
```

```
            sender = status.MPI_SOURCE;

            printf("Partial sum %i returned from process
%i\n", partial_sum, sender);

            sum += partial_sum;
         }

         printf("The grand total is: %i\n", sum);
      }

      else {

         /* I must be a slave process, so I must receive my
array segment,
          * storing it in a "local" array, array1. */

         ierr = MPI_Recv( &num_rows_to_receive, 1, MPI_INT,
               root_process, send_data_tag, MPI_COMM_WORLD,
&status);

         ierr = MPI_Recv( &array2, num_rows_to_receive,
MPI_INT,
               root_process, send_data_tag, MPI_COMM_WORLD,
&status);

         num_rows_received = num_rows_to_receive;

         /* Calculate the sum of my portion of the array */

         partial_sum = 0;
         for(i = 0; i < num_rows_received; i++) {
            partial_sum += array2[i];
         }

         /* and finally, send my partial sum to hte root
process */

         ierr = MPI_Send( &partial_sum, 1, MPI_LONG,
root_process,
               return_data_tag, MPI_COMM_WORLD);
      }
      ierr = MPI_Finalize();
   }
```

The following table shows the values of several variables during the execution of sumarray_mpi. The information comes from a two-processor parallel run, and the values of program variables are shown in **both** processor memory spaces. Note that there is only one process active prior to the call to MPI_Init.

Value histories of selected variables
within the master and slave processes
during a 2-process execution of program sumarray_mpi

Program location	**Before MPI_Init**	**After MPI_Init**		**Before MPI_Send to slave**		**After MPI_Recv by slave**		**After MPI_Recv by master**	
Variable Name	**Proc 0**	**Proc 0**	**Proc 1**	**Proc 0**	**Proc 1**	**Proc 0**	**Proc 1**	**Proc 0**	**Proc 1**
root_process	0	0	0	0	0	0	0	0	0
my_id	.	0	1	0	1	0	1	0	1
num_procs	.	2	2	2	2	2	2	2	2
num_rows	.	.	.	6	.	6	.	6	.
avg_rows_ per_process	.	.	.	3	.	3	.	3	.
num_rows_ received	.	.	.	.	.	.	3	.	3
array[0]	.	.	.	1.0	.	1.0	.	1.0	.
array[1]	.	.	.	2.0	.	2.0	.	2.0	.

array[2]	.	.	.	3.0	.	3.0	.	3.0	.
array[3]	.	.	.	4.0	.	4.0	.	4.0	.
array[4]	.	.	.	5.0	.	5.0	.	5.0	.
array[5]	.	.	.	6.0	.	6.0	.	6.0	.

Logging and tracing MPI activity

It is possible to use `mpirun` to record MPI activity, by using the options `-mpilog` and `-mpitrace`. For more information about this facility see `man mpirun`.

Collective operations

MPI_Send and MPI_Recv are "point-to-point" communications functions. That is, they involve one sender and one receiver. MPI includes a large number of subroutines for performing "collective" operations. Collective operation are performed by MPI routines that are called by each member of a group of processes that want some operation to be performed for them as a group. A collective function may specify one-to-many, many-to-one, or many-to-many message transmission.

MPI supports three classes of collective operations:

- synchronization,
- data movement, and
- collective computation

These classes are not mutually exclusive, of course, since blocking data movement functions also serve to synchronize process activity, and some MPI routines perform both data movement and computation.

Synchronization

The MPI_Barrier function can be used to synchronize a group of processes. To synchronize a group of processes, each one must call MPI_Barrier when it has reached a point where it can go no further until it knows that all its cohorts have reached the same point. Once a process has called MPI_Barrier, it will be blocked until all processes in the group have also called MPI_Barrier.

Collective data movement

There are several routines for performing collective data distribution tasks:
MPI_Bcast
Broadcast data to other processes
MPI_Gather, MPI_Gatherv
Gather data from participating processes into a single structure
MPI_Scatter, MPI_Scatter
Break a structure into portions and distribute those portions to other processes
MPI_Allgather, MPI_Allgatherv
Gather data from different processes into a single structure that is then sent to all participants (Gather-to-all)
MPI_Alltoall, MPI_Alltoallv
Gather data and then scatter it to all participants (All-to-all scatter/gather)
The routines with "V" suffixes move variable-sized blocks of data.

The subroutine MPI_Bcast sends a message from one process to all processes in a communicator.

```
int MPI_Bcast(void *data_to_be_sent, int send_count,
MPI_Datatype send_type,
      int broadcasting_process_ID, MPI_Comm comm);
```

When processes are ready to share information with other processes as part of a broadcast, **ALL of them must execute a call to MPI_BCAST.** There is no separate MPI call to receive a broadcast.

MPI_Bcast could have been used in the program sumarray_mpi presented earlier, in place of the MPI_Send loop that distributed data to each process. Doing so would have resulted in excessive data movement, of course. A better solution would be MPI_Scatter or MPI_Scatterv.

The subroutines MPI_Scatter and MPI_Scatterv take an input array, break the input data into separate portions and send a portion to each one of the processes in a communicating group.

```
int MPI_Scatter(void *send_data, int send_count,
MPI_Datatype send_type,
      void *receive_data, int receive_count, MPI_Datatype
receive_type,
      int sending_process_ID, MPI_Comm comm);
or

int MPI_Scatterv(void *send_data, int *send_count_array,
int *send_start_array,
      MPI_Datatype send_type, void *receive_data, int
receive_count,
      MPI_Datatype receive_type, int sender_process_ID,
MPI_Comm comm);
```

- `data_to_send`: variable of a C type that corresponds to the MPI `send_type` supplied below
- `send_count`: number of data elements to send (int)
- `send_count_array`: array with an entry for each participating process containing the number of data elements to send to that process (int)
- `send_start_array`: array with an entry for each participating process containing the displacement relative to the start of `data_to_send` for each data segment to send (int)
- `send_type`: datatype of elements to send (one of the MPI datatype handles)

- `receive_data`: variable of a C type that corresponds to the MPI `receive_type` supplied below
- `receive_count`: number of data elements to receive (int)
- `receive_type`: datatype of elements to receive (one of the MPI datatype handles)
- `sender_ID`: process ID of the sender (int)
- `receive_tag`: receive tag (int)
- `comm`: communicator (handle)
- `status`: status object (MPI_Status)

The routine MPI_Scatterv could have been used in the program sumarray_mpi presented earlier, in place of the MPI_Send loop that distributed data to each process.

MPI_Bcast, MPI_Scatter, and other collective routines build a communication tree among the participating processes to minimize message traffic. If there are N processes involved, there would normally be N-1 transmissions during a broadcast operation, but if a tree is built so that the broadcasting process sends the broadcast to 2 processes, and they each send it on to 2 other processes, the total number of messages transferred is only O(ln N).

Collective computation routines

Collective computation is similar to collective data movement with the additional feature that data may be modified as it is moved. The following routines can be used for collective computation.

MPI_Reduce

Perform a reduction operation. That is, apply some operation to some operand in every participating process. For example, add an integer residing in every process together and put the result in a process specified in the MPI_Reduce argument list.

MPI_Allreduce

Perform a reduction leaving the result in all participating processes
MPI_Reduce_scatter
Perform a reduction and then scatter the result
MPI_Scan
Perform a reduction leaving partial results (computed up to the point of a process's involvement in the reduction tree traversal) in each participating process. (parallel prefix)
The subroutine MPI_Reduce combines data from all processes in a communicator using one of several reduction operations to produce a single result that appears in a specified target process.

```
int MPI_Reduce(void *data_to_be_sent, void
*result_to_be_received_by_target,
      int send_count, MPI_Datatype send_type, MPI_Op
operation,
      int target_process_ID, MPI_Comm comm);
```

When processes are ready to share information with other processes as part of a data reduction, all of the participating processes execute a call to MPI_Reduce, which uses local data to calculate each process's portion of the reduction operation and communicates the local result to other processes as necessary. Only the target_process_ID receives the final result.

MPI_Reduce could have been used in the program sumarray_mpi presented earlier, in place of the MPI_Recv loop that collected partial sums from each process.

Collective computation built-in operations

Many of the MPI collective computation routines take both built-in and user-defined combination functions. The built-in functions are:

Operation handle	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_LAND	Logical AND
MPI_LOR	Logical OR
MPI_LXOR	Logical Exclusive OR
MPI_BAND	Bitwise AND
MPI_BOR	Bitwise OR

MPI_BXOR	Bitwise Exclusive OR
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

A collective operation example

The following program integrates the function sin(X) over the range 0 to 2 pi. It will be followed by a parallel version of the same program that uses the MPI library.

```
   /* program to integrate sin(x) between 0 and pi by
computing
    * the area of a number of rectangles chosen so as to
approximate
    * the shape under the curve of the function.
    *
    * 1) ask the user to choose the number of intervals,
    * 2) compute the interval width (rect_width),
    * 3) for each interval:
    *
    * a) find the middle of the interval (x_middle),
    * b) compute the height of the rectangle,
sin(x_middle),
    * c) find the area of the rectangle as the product of
    *    the interval width and its height sin(x_middle),
and
    * d) increment a running total.
    **/

   #include <stdio.h>
   #include <math.h>

   #define PI 3.1415926535

   main(int argc, char **argv)
   {
      int i, num_intervals;
      double rect_width, area, sum, x_middle;
      printf("Please enter the number of intervals to 
interpolate: ");
      scanf("%i", &num_intervals);

      rect_width = PI / num_intervals;

      sum = 0;
      for(i = 1; i < num_intervals + 1; i++) {

         /* find the middle of the interval on the X-axis. 
*/

         x_middle = (i - 0.5) * rect_width;
         area = sin(x_middle) * rect_width;
         sum = sum + area;
      }

      printf("The total area is: %f\n", (float)sum);
   }
```

The next program is an MPI version of the program above. It uses MPI_Bcast to send information to each participating process and MPI_Reduce to get a grand total of the areas computed by each participating process.

```
   /* This program integrates sin(x) between 0 and pi by 
computing
    * the area of a number of rectangles chosen so as to 
approximate
    * the shape under the curve of the function using MPI.
    *
    * The root process acts as a master to a group of child 
process
    * that act as slaves.  The master prompts for the 
number of
    * interpolations and broadcasts that value to each 
slave.
    *
    * There are num_procs processes all together, and a 
process
    * computes the area defined by every num_procs-th 
interval,
    * collects a partial sum of those areas, and sends its
partial
    * sum to the root process.
    **/

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

   #define PI 3.1415926535

   main(int argc, char **argv)
   {
      int my_id, root_process, num_procs, ierr,
num_intervals, i;
      double rect_width, area, sum, x_middle, partial_sum;
      MPI_Status status;

      /* Let process 0 be the root process. */

      root_process = 0;

      /* Now replicate this process to create parallel
processes. */

      ierr = MPI_Init(&argc, &argv);

      /* Find out MY process ID, and how many processes
were started. */

      ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
      ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

      if(my_id == root_process) {

         /* I must be the root process, so I will query the
user
            to determine how many interpolation intervals
to use. */
```

```
         printf("Please enter the number of intervals to
interpolate: ");
         scanf("%i", &num_intervals);
      }

      /* Then...no matter which process I am:
       *
       * I engage in a broadcast so that the number of
intervals is
       * sent from the root process to the other processes,
and ...
       **/
      ierr = MPI_Bcast(&num_intervals, 1, MPI_INT,
root_process,
            MPI_COMM_WORLD);

      /* calculate the width of a rectangle, and */

      rect_width = PI / num_intervals;

      /* then calculate the sum of the areas of the
rectangles for
       * which I am responsible.  Start with the (my_id
+1)th
       * interval and process every num_procs-th interval
thereafter.
       **/
      partial_sum = 0;
      for(i = my_id + 1; i <num_intervals + 1; i +=
num_procs) {

         /* Find the middle of the interval on the X-axis.
*/
         x_middle = (i - 0.5) * rect_width;
         area =  sin(x_middle) * rect_width;
         partial_sum = partial_sum + area;
      }
      printf("proc %i computes: %f\n", my_id,
(float)partial_sum);
```

```
      /* and finally, engage in a reduction in which all 
partial sums
       * are combined, and the grand sum appears in 
variable "sum" in
       * the root process,
       **/
      ierr = MPI_Reduce(&partial_sum, &sum, 1, MPI_DOUBLE,
            MPI_SUM, root_process, MPI_COMM_WORLD);

      /* and, if I am the root process, print the result. 
*/

      if(my_id == root_process) {
         printf("The integral is %f\n", (float)sum);

         /* (yes, we could have summed just the heights, 
and
          * postponed the multiplication by rect_width til 
now.) */
      }

      /* Close down this processes. */

      ierr = MPI_Finalize();
   }
```

Simultaneous send and receive

The subroutine MPI_Sendrecv exchanges messages with another process. A send-receive operation is useful for avoiding some kinds of unsafe interaction patterns and for implementing remote procedure calls.

A message sent by a send-receive operation can be received by MPI_Recv and a send-receive operation can receive a message sent by an MPI_Send.

```
MPI_Sendrecv(&data_to_send, send_count, send_type, 
destination_ID, send_tag,
      &received_data, receive_count, receive_type, 
sender_ID, receive_tag,
      comm, &status)
```

- `data_to_send`: variable of a C type that corresponds to the MPI `send_type` supplied below
- `send_count`: number of data elements to send (int)
- `send_type`: datatype of elements to send (one of the MPI datatype handles)
- `destination_ID`: process ID of the destination (int)
- `send_tag`: send tag (int)
- `received_data`: variable of a C type that corresponds to the MPI `receive_type` supplied below
- `receive_count`: number of data elements to receive (int)
- `receive_type`: datatype of elements to receive (one of the MPI datatype handles)
- `sender_ID`: process ID of the sender (int)
- `receive_tag`: receive tag (int)
- `comm`: communicator (handle)
- `status`: status object (MPI_Status)

MPI tags

MPI_Send and MPI_Recv, as well as other MPI routines, allow the user to specify a tag value with each transmission. These tag values may be used to specify the message type, or "context," in a situation where a program may receive messages of several types during the same program. The receiver simply checks the tag value to decide what kind of message it has received.

MPI communicators

Every MPI communication operation involves a "communicator." Communicators identify the group of processes involved in a communication operation and/or the context in which it occurs. The source and destination processes specified in point-to-point routines like MPI_Send and MPI_Recv must be members of the specified communicator and the two calls must reference the same communicator.
Collective operations include just those processes identified by the communicator specified in the calls.

The communicator MPI_COMM_WORLD is defined by default for all MPI runs, and includes all processes defined by MPI_Init during that run. Additional communicators can be defined that include all or part of those processes. For example, suppose a group of processes needs to engage in two different reductions involving disjoint sets of processes. A communicator can be defined for each subset of MPI_COMM_WORLD and specified in the two reduction calls to manage message transmission.

MPI_Comm_split can be used to create a new communicator composed of a subset of another communicator. MPI_Comm_dup can be used to create a new communicator composed of all of the members of another communicator. This may be useful for managing interactions within a set of processes in place of message tags.

More information

This short introduction omits many MPI topics and routines, and covers most topics only lightly. In particular, it omits discussions of topologies, unsafe communication and non-blocking communication.
For additional information concerning these and other topics please consult:

- the major MPI Web site, where you will find versions of the standards: http://www.mcs.anl.gov/mpi
- the books:
  - Gropp, Lusk, and Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press, 1994
  - Foster, Ian, *Designing and Building Parallel Programs*, available in both hardcopy (Addison-Wesley Publishing Co., 1994) and on-line versions,
  - Pacheco, Peter, A User's Guide to MPI, which gives a tutorial introduction extended to cover derived types, communicators and topologies, or
- the newsgroup comp.parallel.mpi

Exercises

Here are some exercises for continuing your investigation of MPI:

- Convert the hello world program to print its messages in rank order.
- Convert the example program sumarray_mpi to use MPI_Scatter and/or MPI_Reduce.
- Write a program to find all positive primes up to some maximum value, using MPI_Recv to receive requests for integers to test. The master will loop from 2 to the maximum value on
  1. issue MPI_Recv and wait for a message from any slave (MPI_ANY_SOURCE),
  2. if the message is zero, the process is just starting,
     if the message is negative, it is a non-prime,
     if the message is positive, it is a prime.
  3. use MPI_Send to send a number to test.
- and each slave will send a request for a number to the master, receive an integer to test, test it, and return that integer if it is prime, but its negative value if it is not prime.
- Write a program to send a token from processor to processor in a loop.

Credit: http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml

Document prepared by:

Daniel Thomasset and
Michael Grobe
Academic Computing Services
The University of Kansas

with assistance and overheads provided by
The National Computational Science Alliance (NCSA) at
The University of Illinois