MPI Programming — Part 1

Objectives

- Basic structure of MPI code
- MPI communicators
- Sample programs

Introduction to MPI

The Message Passing Interface (MPI) is a library of subroutines (in Fortran) or function calls (in C) that can be used to implement a message-passing program.

MPI allows the coordination of a program running as multiple processes in a distributed-memory environment, yet it is flexible enough to also be used in a shared-memory environment.

MPI programs can be used and compiled on a wide variety of single platforms or (homogeneous or heterogeneous) clusters of computers over a network.

The MPI library is standardized, so working code containing MPI subroutines and function calls should work (without further changes!) on any machine on which the MPI library is installed.

A very brief history of MPI

MPI was developed over two years of discussions led by the MPI Forum, a group of roughly sixty people representing some forty organizations.

The MPI 1.0 standard was defined in Spring of 1994.

The MPI 2.0 standard was defined in Fall of 1997.

MPI 2.0 was such a complicated enhancement to the MPI standard that very few implementations exist!

Learning MPI can seem intimidating: MPI 1.1 has more than 125 different commands!

However, most programmers can accomplish what they want from their programs while sticking to a small subset of MPI commands (as few as 6).

In this course, we will stick to MPI 1.1.

Basic structure of MPI code

MPI programs have the following general structure:

- include the MPI header file
- declare variables
- initialize MPI environment
- $<$ compute, communicate, etc. $>$
- finalize MPI environment

Notes:

- The MPI environment can only be initialized once per program, and it cannot be initialized before the MPI header file is included.
- Calls to MPI routines are not recognized before the MPI environment is initialized or after it is finalized.

MPI header files

Header files are more commonly used in C than in older versions of Fortran, such as FORTRAN77.

In general, header files are usually source code for declarations of commonly used constructs.

Specifically, MPI header files contain the prototypes for MPI functions/subroutines, as well as definitions of macros, special constants, and datatypes used by MPI.

An `include` statement must appear in any source file that contains MPI function calls or constants.

In Fortran, we type

```
INCLUDE ‘mpif.h‘
```

In C, the equivalent is

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

MPI naming conventions

All MPI entities (subroutines, constants, types, etc.) begin with `MPI_` to highlight them and avoid conflicts.

In Fortran, they have the general form

```
MPI_XXXXX(parameter,...,IERR)
```

For example,

```
MPI_INIT(IERR)
```

The difference in C is basically the absence of `IERR`:

```
MPI_Xxxxx(parameter,...)
```

MPI constants are always capitalized, e.g.,

`MPI_COMM_WORLD`, `MPI_REAL`, etc.

In Fortran, MPI entities are always associated with the `INTEGER` data type.

MPI subroutines and return values

An MPI subroutine returns an error code that can be checked for the successful operation of the subroutine.

This error code is always returned as an `INTEGER` in the variable given by the last argument , e.g.,

```
  INTEGER IERR
  ...
  CALL MPI_INIT(IERR)
  ...
```

The error code returned is equal to the pre-defined integer constant `MPI_SUCCESS` if the routine ran successfully:

```
IF (IERR.EQ.MPI_SUCCESS) THEN
  do stuff given that routine ran correctly
END IF
```

If an error occurred, then `IERR` returns an implementation-dependent value indicating the specific error, which can then be handled appropriately.

MPI handles

MPI defines and maintains its own internal data structures related to communication, etc.

These data structures are accessed through handles.

Handles are returned by various MPI calls and may be used as arguments in other MPI calls.

In Fortran, handles are integers or arrays of integers; any arrays are indexed from 1.

For example,

- `MPI_SUCCESS` is an integer used to test error codes.
- `MPI_COMM_WORLD` is an integer representing a pre-defined communicator consisting of all processes.

Handles may be copied using the standard assignment operation.

MPI data types

MPI has its own reference data types corresponding to elementary data types in Fortran or C:

- Variables are normally declared as Fortran/C types.
- MPI type names are used as arguments to MPI routines when needed.
- Arbitrary data types may be built in MPI from the intrinsic Fortran/C data types.

MPI shifts the burden of details such as the floating-point representation to the implementation.

MPI allows for automatic translation between representations in heterogeneous environments.

In general, the MPI data type in a receive must match that specified in the send.

Basic MPI data types

MPI data type	Fortran data type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_PACKED	user-defined

MPI data type	C data type
MPI_INT	signed integer
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_CHAR	signed char
MPI_PACKED	user-defined

Communicators

A communicator is a handle representing a group of processes that can communicate with one another.

The communicator name is required as an argument to all point-to-point and collective operations.

- The communicator specified in the send and receive calls must agree for communication to take place.

- Processes can communicate only if they share a communicator.

There can be many communicators, and a process can be a member of a number of different communicators.

Within each communicator, processes are numbered consecutively (starting at 0).

This identifying number is known as the rank of the process in that communicator.

Communicators

The rank is also used to specify the source and destination in send and receive calls.

If a process belongs to more than one communicator, its rank in each can (and usually will) be different.

MPI automatically provides a basic communicator called `MPI_COMM_WORLD` that consists of all available processes.

Every process can communicate with every other process using the `MPI_COMM_WORLD` communicator.

Additional communicators consisting of subsets of the available processes can also be defined.

Getting communicator rank

A process can determine its rank in a communicator with a call to `MPI_COMM_RANK`.

In Fortran, this call looks like

```
    CALL MPI_COMM_RANK(COMM, RANK, IERR)
```

where the arguments are all of type `INTEGER`.

`COMM` contains the name of the communicator, e.g., `MPI_COMM_WORLD`.

The rank of the process within the communicator `COMM` is returned in `RANK`.

The analogous syntax in C looks like

```
int MPI_Comm_rank(
      MPI_Comm comm    /* in  */
      int* my_rank_p  /* out */);
```

Getting communicator size

A process can also determine the *size* (number of processes) of any communicator to which it belongs with a call to `MPI_COMM_SIZE`.

In Fortran, this call looks like

```
    CALL MPI_COMM_SIZE(COMM, SIZE, IERR)
```

where the arguments are all of type `INTEGER`.

Again, `COMM` contains the name of the communicator.

The number of processes associated with the communicator `COMM` is returned in `SIZE`.

The analogous syntax in C looks like

```
int MPI_Comm_size(
      MPI_Comm comm    /* in  */
      int* comm_sz_p  /* out */);
```

Finalizing MPI

The last call to an MPI routine in any MPI program should be to `MPI_FINALIZE`.

`MPI_FINALIZE` cleans up all MPI data structures, cancels incomplete operations, etc.

`MPI_FINALIZE` must be called by all processes!

If any processes do not call `MPI_FINALIZE`, the program will hang.

Once `MPI_FINALIZE` has been called, no other MPI routines (including `MPI_INIT`!) may be called.

In Fortran, the call looks like

```
CALL MPI_FINALIZE(IERR)
```

In C, the analogous syntax is

```
int MPI_Finalize(void);
```

`hello, world` — ***serial Fortran***

Here is a simple serial Fortran90 program called `helloWorld.f90` to print the message "hello, world":

```
PROGRAM helloWorld

  PRINT *, "hello, world"

END PROGRAM helloWorld
```

Compiled and run with the commands

```
     gfortran helloWorld.f90 -o helloWorld
     ./helloWorld
```

produces the unsurprising output

```
     hello, world
```

`hello, world` ***with MPI***

`hello, world` is the classic first program of anyone using a new computer language.

Here is a simple MPI version using Fortran90.

```
PROGRAM helloWorldMPI

  INCLUDE 'mpif.h'
  INTEGER IERR
  ! Initialize MPI environment
  CALL MPI_INIT(IERR)

  PRINT *, "hello, world!"

  ! Finalize MPI environment
  CALL MPI_FINALIZE(IERR)
END PROGRAM helloWorldMPI
```

`hello, world` *with MPI*

To compile and link this program to an executable called `helloWorldMPI`, we could do something like

```
>> mpif90 -c helloWorldMPI.f90
>> mpif90 -o helloWorldMPI helloWorldMPPI.o
```

or all at once using

```
>> mpif90 helloWorldMPI.f90 -o helloWorldMPI
```

Without a queuing system, we could use

```
>> mpirun -np 4 ./helloWorldMPI
```

Thus, when run on four processes, the output of this program is

```
hello, world
hello, world
hello, world
hello, world
```

`hello, world`, ***the sequel***

We now modify the `hello, world` program so that each process prints its rank as well as the total number of processes in the communicator `MPI_COMM_WORLD`.

```
PROGRAM helloWorld2MPI

  INCLUDE 'mpif.h'
  ! Declare variables
  INTEGER RANK, SIZE, IERR

  ! Initialize MPI environment
  CALL MPI_INIT(IERR)
  ! Not checking for errors :)

  ! Get the rank
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, IERR)
  ! Get the size
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, SIZE, IERR)
  ! Display the result
  PRINT *, "Processor", RANK, "of ", SIZE, &
       "says, 'hello, world'"

  ! Finalize MPI environment
  CALL MPI_FINALIZE(IERR)
END PROGRAM helloWorld2MPI
```

Running this code on 6 processes could produce something like:

```
spiteri@robson:~/test> mpirun -np 6 ./helloWorld2MPI
 Process 0 of  6 says, 'hello, world'
 Process 2 of  6 says, 'hello, world'
 Process 1 of  6 says, 'hello, world'
 Process 3 of  6 says, 'hello, world'
 Process 4 of  6 says, 'hello, world'
 Process 5 of  6 says, 'hello, world'
```

`hello, world`, *the sequel*

Let's take a look at a variant of this program in C:

```
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void)
   char       greeting[MAX_STRING];  /* String storing message */
   int        comm_sz;               /* Number of processes    */
   int        my_rank;               /* My process rank        */

   /* Start up MPI */
   MPI_Init(NULL, NULL);

   /* Get the number of processes */
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   /* Get my rank among all the processes */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
  if (my_rank != 0)
     /* Create message */
     sprintf(greeting, "Greetings from process %d of %d!",
           my_rank, comm_sz);
     /* Send message to process 0 */
     MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
           MPI_COMM_WORLD);
   else
     /* Print my message */
     printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
     for (int q = 1; q < comm_sz; q++)
        /* Receive message from process q */
        MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
           0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        /* Print message from process q */
        printf("%s\n", greeting);



  /* Shut down MPI */
  MPI_Finalize();

  return 0;
 /* main */
```

`hello, world`, *the sequel*

Recall we compile this code via

```
mpicc -g -Wall -o mpi_hello mpi_hello.c
```

The code is run with (basically) an identical call as for the Fortran program:

```
mpirun -n <number of processes> ./mpi_hello
```

So the output from the command

```
mpirun -n 4 ./mpi_hello
```

will be

```
Greetings from process 0 of 4!
Greetings from process 1 of 4!
Greetings from process 2 of 4!
Greetings from process 3 of 4!
```

`hello, world`, *the sequel*

The main difference between the Fortran and C programs is that the latter *sends* the greeting as a message for process 0 to *receive* (and print).

The syntax of `MPI_Send` is

```
int MPI_Send(
     void*          msg_buf_p      /* in */,
     int            msg_size       /* in */,
     MPI_Datatype   msg_type       /* in */,
     int            dest           /* in */,
     int            tag            /* in */,
     MPI_Comm       communicator   /* in */);
```

The first three arguments determine the contents of the message; the last three determine the destination.

`hello, world`, *the sequel*

`msg_buf_p` is a pointer to the block of memory containing the contents of the message, in this case the string `greeting`.

`msg_size` is the number of characters in the string plus 1 for the `'\0'` string termination character in C.

`msg_type` is of type `MPI_CHAR`; so together we see that the message contains `strlen(greeting)+1 chars`.

Note: the size of the string `greeting` is not necessarily the same as that specified by `msg_size` and `msg_type`.

`dest` gives the destination process.

`tag` is a non-negative `int` that can be used to distinguish messages that might otherwise be identical; e.g., when sending a number of floats, a tag can indicate whether a given float should be printed or used in a computation.

Finally, `MPI_Comm` is the communicator; `dest` is defined relative to it.

hello, world, *the sequel*

MPI_Recv has a similar (complementary) syntax to MPI_Send, but with a few important differences.

```
int MPI_Recv(
     void*          msg_buf_p      /* out */,
     int            buf_size       /* in  */,
     MPI_Datatype   buf_type       /* in  */,
     int            source         /* in  */,
     int            tag            /* in  */,
     MPI_Comm       communicator   /* in  */
     MPI_Status     status_p       /* out */);
```

Again, the first three arguments specify the memory available (buffer) for receiving the message.

The next three identify the message.

Both tag and communicator must match those sent by the sending process.

We discuss status_p shortly.

It is not often used, so in the example the MPI constant MPI_STATUS_IGNORE was passed.

`hello, world`, *the sequel*

In order for a message to be received by `MPI_Recv`, it must be *matched* to a send.

Necessary but not sufficient conditions are that the tags and communicators match and the source and destination processes must be consistent; e.g., process A is expecting a message from process B.

The only other caveat is that the sending and receiving buffers must be compatible.

We will assume this means they are of the same type and that the receiving buffer size is at least as large as the sending buffer size.

`hello, world`, *the sequel*

One needs to beware that receiving processes are generally receiving messages from many sending processes, and it is generally impossible to predict the order in which messages are sent (or received).

To allow for flexibility in handling received messages, MPI provides a special constant `MPI_ANY_SOURCE` as a wildcard that can be used in `MPI_Recv`.

Similarly, the wildcard constant `MPI_ANY_TAG` can be used in `MPI_Recv` to handle messages with any tag.

Two final notes:

1. Only receiving processes can use wildcards; sending processes must specify a destination process rank and a non-negative tag.

2. There is no wildcard that can be used for communicators; both sending and receiving processes must specify communicators.

`hello, world`, ***the sequel***

It is possible for a receiving process to successfully receive a message while not knowing the sender, the tag, and the size of the message.

All this information is of course known and can be accessed via the `status_p` argument.

`status_p` has type `MPI_Status*`, which is a `struct` having at least the members `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`.

We can determine the sender and tag of a received message from `MPI_Recv` by accessing

```
status_p.MPI_SOURCE
status_p.MPI_TAG
```

The amount of data received is not directly accessible but can be retrieved with a call to `MPI_Get_count`.

```
MPI_Get_count(&status_p, recv_type, &count)
```

Of course, all of these issues are not necessary and potentially should be avoided in the name of efficiency.

`hello, world`, ***the sequel***

What happens exactly when a message is sent from one process to another depends on the particular system, but the general procedure is the following.

The sending process assembles the message, i.e., the "address" information plus the data themselves.

The sending process then either *buffers* message and returns or *blocks*; i.e., it does not return until it can begin transmission.

(Other functions are available if it is important that we know when a message is actually *received*, etc.)

Specific details depend on the implementation, but typically messages that are below a certain threshold in size are automatically buffered by `MPI_Send`.

`hello, world`, ***the sequel***

In contrast, `MPI_Recv` always blocks until a matching message has been received.

So when a call from `MPI_Recv` returns, we know (barring errors!) that the message has been safely stored in the receive buffer.

(There is a non-blocking version of this function that returns having only checked whether a matching message is available, whether one is or not.)

MPI also requires that messages be *non-overtaking*, i.e., the order in which messages sent by one processor to another must be reflected in the order in which they can be received.

However, the order in which messages are received (let alone sent!) by other processes cannot be controlled.

As usual, it is important for programmers to maintain the mindset that the processes run autonomously.

`hello, world`, ***the sequel***

A potential pitfall of blocking communication is that if a process tries to receive a message for which there is no matching send, it will block forever or *deadlock*.

In other words, the program will hang.

It is therefore critical when programming to ensure every receive has a matching send.

This includes ensuring that tags match and that the source and destination are never the same process so the program does not hang but also so that messages are not inadvertantly received!

Similarly, unmatched calls to `MPI_Send` will typically hang the sending process; “best” case, the message is buffered so the process can continue but the message will be lost.

Sample Program: Trapezoidal Rule

Historically in mathematics, quadrature refers to the act of trying to find a square with the same area of a given circle.

In mathematical computing, quadrature refers to the numerical approximation of definite integrals.

Let $f(x)$ be a real-valued function of a real variable, defined on a finite interval $a \leq x \leq b$.

We seek to compute the value of the definite integral

$$\int_a^b f(x)\,dx.$$

Note: This is a real number.

In line with its historical meaning, “quadrature” might conjure up the vision of plotting the function on graph paper and counting the little squares that lie underneath the curve.

Sample Program: Trapezoidal Rule

Let $\Delta x = b - a$ be the length of the integration interval.

The trapezoidal rule T approximates the integral by the area of a trapezoid with base Δx and sides equal to the values of $f(x)$ at the two end points.

$$T = \Delta x \left(\frac{f(a) + f(b)}{2} \right).$$

Put differently, the trapezoidal rule forms a *linear interpolant* between $(a, f(a))$ and $(b, f(b))$ and integrates the interpolant exactly to define the rule.

Naturally the error in using the trapezoidal rule depends on Δx: in general, as Δx increases, so does the error.

Sample Program: Trapezoidal Rule

This leads to the *composite* trapezoidal rule:

Assume the domain $[a, b]$ is partitioned into n (equal) subintervals so that

$$\Delta x = \frac{b-a}{n}.$$

Let $x_i = a + i\Delta x$, $i = 0, 1, \ldots, n$.

Then the composite trapezoidal rule is

$$\begin{aligned} T_n &= \frac{\Delta x}{2} \sum_{i=1}^{n} (f(x_{i-1}) + f(x_i)) \\ &= \Delta x \left[\frac{f(a) + f(b)}{2} + \sum_{i=1}^{n-1} f(x_i) \right]. \end{aligned}$$

Sample Program: Trapezoidal Rule

Serial code for this method might look like

```
/* input a, b, n */
dx = (b-a)/n;
approx = (f(a)+f(b))/2;
for (i = 1; i <= n-1; i++){
  x_i = a + i*dx;
  approx += f(x_i);
}
approx = dx*approx;
```

Sample Program: Trapezoidal Rule

Following Foster's methodology, we

1. partition the problem (find the area of a single trapezoid, add them up)

2. identify communication (information about single trapezoid scattered, single areas gathered)

3. aggregate tasks (there are probably more trapezoids than cores, so split $[a, b]$ into `comm_sz` subintervals)

4. map tasks to cores (subintervals to cores; results back to process 0)

Sample Program: Trapezoidal Rule

```
/* File:     mpi_trap1.c
 * Purpose:  Use MPI to implement a parallel version of the trapezoidal
 *           rule.  In this version the endpoints of the interval and
 *           the number of trapezoids are hardwired.
 *
 * Input:    None.
 * Output:   Estimate of the integral from a to b of f(x)
 *           using the trapezoidal rule and n trapezoids.
 *
 * Compile:  mpicc -g -Wall -o mpi_trap1 mpi_trap1.c
 * Run:      mpiexec -n <number of processes> ./mpi_trap1
 *
 * Algorithm:
 *    1.  Each process calculates "its" interval of
 *        integration.
 *    2.  Each process estimates the integral of f(x)
 *        over its interval using the trapezoidal rule.
 *    3a. Each process != 0 sends its integral to 0.
 *    3b. Process 0 sums the calculations received from
 *        the individual processes and prints the result.
 *
 * Note:  f(x), a, b, and n are all hardwired.
 *
 * IPP:   Section 3.2.2 (pp. 96 and ff.)
 */
#include <stdio.h>

/* We'll be using MPI routines, definitions, etc. */
#include <mpi.h>

/* Calculate local integral  */
double Trap(double left_endpt, double right_endpt, int trap_count,
   double base_len);
```

```
/* Function we're integrating */
double f(double x);

int main(void) {
   int my_rank, comm_sz, n = 1024, local_n;
   double a = 0.0, b = 3.0, dx, local_a, local_b;
   double local_int, total_int;
   int source;

   /* Let the system do what it needs to start up MPI */
   MPI_Init(NULL, NULL);

   /* Get my process rank */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   /* Find out how many processes are being used */
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   dx = (b-a)/n;         /* dx is the same for all processes */
   local_n = n/comm_sz; /* So is the number of trapezoids   */

   /* Length of each process' interval of
    * integration = local_n*dx.  So my interval
    * starts at: */
   local_a = a + my_rank*local_n*dx;
   local_b = local_a + local_n*dx;
   local_int = Trap(local_a, local_b, local_n, dx);

   /* Add up the integrals calculated by each process */
   if (my_rank != 0) {
      MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
            MPI_COMM_WORLD);
   } else {
      total_int = local_int;
      for (source = 1; source < comm_sz; source++) {
         MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         total_int += local_int;
      }
   }

   /* Print the result */
   if (my_rank == 0) {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %.15e\n",
          a, b, total_int);
   }

   /* Shut down MPI */
   MPI_Finalize();

   return 0;
} /*  main  */


/*------------------------------------------------------------------
 * Function:     Trap
 * Purpose:      Serial function for estimating a definite integral
 *               using the trapezoidal rule
 * Input args:   left_endpt
 *               right_endpt
 *               trap_count
 *               base_len
 * Return val:   Trapezoidal rule estimate of integral from
 *               left_endpt to right_endpt using trap_count
 *               trapezoids
 */
double Trap(
      double left_endpt  /* in */,
      double right_endpt /* in */,
      int    trap_count  /* in */,
      double base_len    /* in */) {
   double estimate, x;
   int i;
   estimate = (f(left_endpt) + f(right_endpt))/2.0;
   for (i = 1; i <= trap_count-1; i++) {
      x = left_endpt + i*base_len;
      estimate += f(x);
   }
   estimate = estimate*base_len;

   return estimate;
} /*  Trap  */


/*------------------------------------------------------------------
 * Function:    f
 * Purpose:     Compute value of function to be integrated
 * Input args:  x
 */
double f(double x) {
   return x*x;
} /* f */
```

Summary

- MPI header file, initialize / finalize MPI environment
- MPI entities (naming, error codes, handles, data types)
- `hello, world`, and `hello, world`, the sequel
- Basic trapezoidal rule