MPI Program Execution

1. MPI programs

- Consider the "hello, world" program.

```
#include <stdio.h>
int main(void)
{
        printf("hello, worldnn");
        return 0;
}
```

- Let's write a program similar to "hello, world" that makes some use of MPI. Instead of having each process simply print a message, we'll designate one process to do the output, and the other processes will send it messages, which it will print.

- In parallel programming, it's common for the processes to be identified by nonnegative integer ranks. So if there are p processes, the processes will have ranks 0, 1, 2, : : : , p-1. For our parallel "hello, world," let's make process 0 the designated process, and the other processes will send it messages.

```
1  #include <stdio.h>
2  #include <string.h>  /* For strlen             */
3  #include <mpi.h>     /* For MPI functions, etc */
4
5  const int MAX_STRING = 100;
6
7  int main(void) {
8     char       greeting[MAX_STRING];
9     int        comm_sz;  /* Number of processes */
10    int        my_rank;  /* My process rank     */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17       sprintf(greeting, "Greetings from process %d of %d!",
18             my_rank, comm_sz);
19       MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       printf("Greetings from process %d of %d!\n", my_rank,
             comm_sz);
23       for (int q = 1; q < comm_sz; q++) {
24          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          printf("%s\n", greeting);
27       }
28    }
29
30    MPI_Finalize();
31    return 0;
32 }  /* main */
```

- We're using a text editor to write the program source, and the command line to compile and run. Many systems use a command called mpicc for compilation:

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

- Typically, ***mpicc*** is a script that's a wrapper for the C compiler. A wrapper script is a script whose main purpose is to run some program. In this case, the program is the C compiler. However, the wrapper simplifies the running of the compiler by telling it where to find the necessary header files and which libraries to link with the object file.

- Many systems also support program startup with mpiexec:

  $ mpiexec -n <number of processes> ./mpi_hello

  So to run the program with one process, we'd type

  $ mpiexec -n 1 ./mpi hello

  and to run the program with four processes, we'd type

  $ mpiexec -n 4 ./mpi hello

  With one process the program's output would be

  Greetings from process 0 of 1!

  and with four processes the program's output would 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!

- The first thing to observe is that this is a C program. For example, it includes the standard C header files ***stdio.h*** and ***string.h***. It also has a ***main*** function just like any other C program.

- Line 3 includes the ***mpi.h*** header file. This contains prototypes of MPI functions, macro definitions, type definitions, and so on; it contains all the definitions and declarations needed for compiling an MPI program.

- The identifiers defined by MPI start with the string MPI . The first letter following the underscore is capitalized for function names and MPI-defined types.

2. MPI_Init and MPI_Finalize:

- In Line 12 the call to MPI Init tells the MPI system to do all of the necessary setup. For example, it might allocate storage for message buffers, and it might decide which process gets which rank. As a rule of thumb, no other MPI functions should be called before the program calls ***MPI_Init***. Its syntax is:

```
int MPI_Init
(
int*          argc p      /* in/out */,
char***       argv p      /* in/out */

);
```

- The arguments, argc p and argv p, are pointers to the arguments to main, argc, and argv. However, when our program doesn't use these arguments, we can just pass NULL for both.

- In Line 30 the call to MPI_Finalize tells the MPI system that we're done using MPI, and that any resources allocated for MPI can be freed. The syntax is quite simple:

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

In general, no MPI functions should be called after the call to MPI Finalize. Thus, a typical MPI program has the following basic outline:

```
. . .
#include <mpi.h>
. . .
int main(int argc, char_ argv[])
{
. . .
/* No MPI calls before this*/
MPI_Init(&argc, &argv);
. . .
MPI_Finalize();
/* No MPI calls after this*_/
        . . .
return 0;
}
```

3. Communicators, MPI Comm_size and MPI Comm rank

- In MPI a communicator is a collection of processes that can send messages to each other. One of the purposes of MPI_Init is to define a communicator that consists of all of the processes started by the user when she started the program. This communicator is called MPI_COMM_WORLD. The function calls in Lines 13 and 14 are getting information about MPI_COMM_WORLD. Their syntax is:

```
int MPI_Comm_size(
MPI_Comm comm          /* in */,
Int * comm_sz_ p       /* out*/);


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

- For both functions, the first argument is a communicator and has the special type defined by MPI for communicators, MPI_Comm. MPI_Comm _size returns in its second argument the number of processes in the communicator, and

MPI_Comm_ rank returns in its second argument the calling process' rank in the communicator.

4. SPMD programs

- Notice that we compiled a single program—we didn't compile a different program for each process—and we did this in spite of the fact that process 0 is doing something different from the other processes: it's receiving a series of messages and printing them, while each of the other processes is creating and sending a message.

  - This is quite common in parallel programming. In fact, most MPI programs are written in this way. That is, a single program is written so that different processes carry out different actions, and this is achieved by simply having the processes branch on the basis of their process rank. Recall that this approach to parallel programming is called single program, multiple data, or SPMD. The if□else statement in Lines 16 through 28 makes our program SPMD.

5. Communication

- In Lines 17 and 18, each process, other than process 0, creates a message it will send to process 0. (The function sprintf is very similar to printf, except that instead of writing to stdout, it writes to a string.)

- Lines 19–20 actually send the message to process 0. Process 0, on the other hand, simply prints its message using printf, and then uses a for loop to receive and print the messages sent by processes 1, 2,… ,comm_sz-1. Lines 24–25 receive the message sent by process q, for q = 1, 2, : : : ,comm_sz -1.

6. MPI_Send

- MPI Send, whose syntax 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, ***msg_buf_p***, ***msg_size***, and ***msg_type***, determine the contents of the message. The remaining arguments, dest, tag, and communicator, determine the destination of the message.

- The first argument, msg_buf_p, is a pointer to the block of memory containing the contents of the message. In our program, this is just the string containing the message, greeting.

- The second and third arguments, msg_size and msg_type, determine the amount of data to be sent. In our program, the msg_size argument is the number of characters in the message plus one character for the '\0' character that terminates C strings. The msg type argument is MPI CHAR. These two arguments together tell the system that the message contains strlen(greeting)+1 chars.
- Since C types (int, char, and so on.) can't be passed as arguments to functions, MPI defines a special type, MPI_Datatype.

Datatypes

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

- The fourth argument, ***dest***, specifies the rank of the process that should receive the message.

- The fifth argument, ***tag***, is a nonnegative int. It can be used to distinguish messages that are otherwise identical. For example, suppose process 1 is sending floats to process 0. Some of the floats should be printed, while others should be used in a computation. Then the first four arguments to MPI Send provide no information regarding which floats should be printed and which should be used in a computation. So process 1 can use, say, a tag of 0 for the messages that should be printed and a tag of 1 for the messages that should be used in a computation.

- The final argument to MPI Send is a communicator. All MPI functions that involve communication have a communicator argument.

7) MPI Recv

- The first six arguments to MPI Recv correspond to the first six arguments of MPI Send:

```
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*/);
```

- Thus, the first three arguments specify the memory available for receiving the message: ***msg_buf_p*** points to the block of memory
- ***buf_size*** determines the number of objects that can be stored in the block
- ***buf type*** indicates the type of the objects.
- The next three arguments identify the message.
- The ***source*** argument specifies the process from which the message should be received.
- The ***tag*** argument should match the tag argument of the message being sent.
- The communicator argument must match the communicator used by the sending process.

8) Message matching

- Suppose process q calls MPI_Send with
  MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send _tag, send_comm);

- Also suppose that process r calls MPI_Recv with

  MPI_Recv (recv_buf_p, recv_buf_sz, recv_type, src, recv_ tag,recv_ comm, &status);

- Then the message sent by q with the above call to MPI_Send can be received by r with the call to MPI_Recv if
  recv_comm = send_comm,
  recv_tag = send_tag,
  dest = r,
  src = q.

- These conditions aren't quite enough for the message to be ***successfully*** received, however. The parameters specified by the first three pairs of arguments, send_buf_p/recv_buf_p, send_buf_sz/recv_buf_sz, and send_ type/recv_type, must specify compatible buffers.

- Most of the time, the following rule will suffice:

  If recv_type = send_type and recv_buf_sz>= send_ buf_ sz, then the message sent by q can be successfully received by r.

9) The status_p argument

- The MPI type MPI_Status is a struct with at least the three members MPI_SOURCE, MPI_TAG, and MPI_ERROR. Suppose our program contains the definition

  MPI_Status status;

- Then, after a call to MPI_Recv in which &status is passed as the last argument, we can determine the sender and tag by examining the two members:

  status.MPI_SOURCE
  status.MPI_TAG

- The amount of data that's been received isn't stored in a field that's directly accessible to the application program. However, it can be retrieved with a call to MPI_Get_count.
- For example, suppose that in our call to MPI_Recv, the type of the receive buffer is recv_type and, once again, we passed in &status. Then the call

  MPI_Get_count(&status, recv type, &count）

  will return the number of elements received in the count argument.
- In general, the syntax of MPI Get count is

```
int MPI_Get_count(
      MPI_Status*      status_p        /*in*/,
      MPI_Datatype     type            /*in*/,
      Int*             count_ p        /*out*/);
```

- Note that the count isn't directly accessible as a member of the MPI_ Status variable simply because it depends on the type of the received data, and, consequently, determining it would probably require a calculation (e.g. (number of bytes received)=(bytes per object)). If this information isn't needed, we shouldn't waste a calculation determining it.

10) Semantics of MPI Send and MPI Recv

- What exactly happens when we send a message from one process to another? Many of the details depend on the particular system, but we can make a few generalizations.

- The sending process will assemble the message. For example, it will add the "envelope" information to the actual data being transmitted—

the destination process rank, the sending process rank, the tag, the communicator, and some information on the size of the message.

- Once the message has been assembled, there are essentially two possibilities: the sending process can buffer the message or it can block.

- If it buffers the message, the MPI system will place the message (data and envelope) into its own internal storage, and the call to MPI_Send will return.

- Alternatively, if the system blocks, it will wait until it can begin transmitting
- the message, and the call to MPI_Send may not return immediately. Thus, if we use MPI_Send, when the function returns, we don't actually know whether the message has been transmitted.

- We only know that the storage we used for the message, the send buffer, is available for reuse by our program. If we need to know that the message has been transmitted, or if we need for our call to MPI Send to return immediately—regardless of whether the message has been sent—MPI provides alternative functions for sending.

- The exact behavior of MPI_Send is determined by the MPI implementation. However,typical implementations have a default "cutoff" message size. If the size of a message is less than the cutoff, it will be buffered.
- If the size of the message is greater than the cutoff, MPI Send will block. Unlike MPI_Send, MPI_Recv always blocks until a matching message has been received. Thus, when a call to MPI_Recv returns, we know that there is a message stored in the receive buffer (unless there's been an error).