MPI Send and Receive

Author: Wes Kendall

Sending and receiving are the two foundational concepts of MPI. Almost every single function in MPI can be implemented with basic send and receive calls. In this lesson, I will discuss how to use MPI's blocking sending and receiving functions, and I will also overview other basic concepts associated with transmitting data using MPI.

> **Note -** All of the code for this site is on GitHub. This tutorial's code is under tutorials/mpi-send-and-receive/code.

Overview of sending and receiving with MPI

MPI's send and receive calls operate in the following manner. First, process *A* decides a message needs to be sent to process *B*. Process A then packs up all of its necessary data into a buffer for process B. These buffers are often referred to as *envelopes* since the data is being packed into a single message before transmission (similar to how letters are packed into envelopes before transmission to the post office). After the data is packed into a buffer, the communication device (which is often a network) is responsible for routing the message to the proper location. The location of the message is defined by the process's rank.

Even though the message is routed to B, process B still has to acknowledge that it wants to receive A's data. Once it does this, the data has been transmitted. Process A is acknowledged that the data has been transmitted and may go back to work.

Sometimes there are cases when A might have to send many different types of messages to B. Instead of B having to go through extra measures to differentiate all these messages, MPI allows senders and receivers to also specify message IDs with the message (known as *tags*). When process B only requests a message with a certain tag number, messages with different tags will be buffered by the network until B is ready for them.

With these concepts in mind, let's look at the prototypes for the MPI sending and receiving functions.

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

```
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status)
```

Although this might seem like a mouthful when reading all of the arguments, they become easier to remember since almost every MPI call uses similar syntax. The first argument is the data buffer. The second and third arguments describe the count and type of elements that reside in the buffer. `MPI_Send` sends the exact count of elements, and `MPI_Recv` will receive **at most** the count of elements (more on this in the next lesson). The fourth and fifth arguments specify the rank of the sending/receiving process and the tag of the message. The sixth argument specifies the communicator and the last argument (for `MPI_Recv` only) provides information about the received message.

Elementary MPI datatypes

The `MPI_Send` and `MPI_Recv` functions utilize MPI Datatypes as a means to specify the structure of a message at a higher level. For example, if the process wishes to send one integer to another, it would use a count of one and a datatype of `MPI_INT`. The other elementary MPI datatypes are listed below with their equivalent C datatypes.

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	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_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

For now, we will only make use of these datatypes in the following MPI tutorials in the beginner category. Once we have covered enough basics, you will learn how to create your own MPI datatypes for characterizing more complex types of messages.

MPI send / recv program

As stated in the beginning, the code for this is available on GitHub, and this tutorial's code is under tutorials/mpi-send-and-receive/code.

The first example in the tutorial code is in send_recv.c. Some of the major parts of the program are shown below.

```
// Find out rank, size
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int number;
if (world_rank == 0) {
    number = -1;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n",
           number);
}
```

`MPI_Comm_rank` and `MPI_Comm_size` are first used to determine the world size along with the rank of the process. Then process zero initializes a number to the value of negative one and sends this value to process one. As you can see in the `else if` statement, process one is calling `MPI_Recv` to receive the number. It also prints off the received value. Since we are sending and receiving exactly one integer, each process requests that one `MPI_INT` be sent/received. Each process also uses a tag number of zero to identify the message. The processes could have also used the predefined constant `MPI_ANY_TAG` for the tag number since only one type of message was being transmitted.

You can run the example code by checking it out on GitHub and using the `run.py` script.

```
>>> git clone https://github.com/wesleykendall/mpitutorial
>>> cd mpitutorial/tutorials
>>> ./run.py send_recv
mpirun -n 2 ./send_recv
Process 1 received number -1 from process 0
```

As expected, process one receives negative one from process zero.

MPI ping pong program

The next example is a ping pong program. In this example, processes use `MPI_Send` and `MPI_Recv` to continually bounce messages off of each other until they decide to stop. Take a look at ping_pong.c. The major portions of the code look like this.

```
int ping_pong_count = 0;
int partner_rank = (world_rank + 1) % 2;
while (ping_pong_count < PING_PONG_LIMIT) {
    if (world_rank == ping_pong_count % 2) {
        // Increment the ping pong count before you send it
        ping_pong_count++;
        MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0,
                 MPI_COMM_WORLD);
        printf("%d sent and incremented ping_pong_count "
               "%d to %d\n", world_rank, ping_pong_count,
               partner_rank);
    } else {
        MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("%d received ping_pong_count %d from %d\n",
               world_rank, ping_pong_count, partner_rank);
    }
}
```

This example is meant to be executed with only two processes. The processes first determine their partner with some simple arithmetic. A `ping_pong_count` is initiated to zero and it is incremented at each ping pong step by the sending process. As the `ping_pong_count` is incremented, the processes take turns being the sender and receiver. Finally, after the limit is reached (ten in my code), the processes stop sending and receiving. The output of the example code will look something like this.

```
>>> ./run.py ping_pong
```

```
0 sent and incremented ping_pong_count 1 to 1
0 received ping_pong_count 2 from 1
0 sent and incremented ping_pong_count 3 to 1
0 received ping_pong_count 4 from 1
0 sent and incremented ping_pong_count 5 to 1
0 received ping_pong_count 6 from 1
0 sent and incremented ping_pong_count 7 to 1
0 received ping_pong_count 8 from 1
0 sent and incremented ping_pong_count 9 to 1
0 received ping_pong_count 10 from 1
1 sent and incremented ping_pong_count 1 to 0
1 received ping_pong_count 2 from 0
1 sent and incremented ping_pong_count 3 to 0
1 received ping_pong_count 4 from 0
1 sent and incremented ping_pong_count 5 to 0
1 received ping_pong_count 6 from 0
1 sent and incremented ping_pong_count 7 to 0
1 received ping_pong_count 8 from 0
1 sent and incremented ping_pong_count 9 to 0
1 received ping_pong_count 10 from 0
```

The output of the programs on other machines will likely be different because of process scheduling. However, as you can see, process zero and one are both taking turns sending and receiving the ping pong counter to each other.

Ring Program

I have included one more example of `MPI_Send` and `MPI_Recv` using more than two processes. In this example, a value is passed around by all processes in a ring-like fashion. Take a look at ring.c. The major portion of the code looks like this.

```
int token;
if (world_rank != 0) {
    MPI_Recv(&token, 1, MPI_INT, world_rank - 1, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received token %d from process %d\n",
           world_rank, token, world_rank - 1);
} else {
    // Set the token's value if you are process 0
    token = -1;
}
MPI_Send(&token, 1, MPI_INT, (world_rank + 1) % world_size,
         0, MPI_COMM_WORLD);

// Now process 0 can receive from the last process.
if (world_rank == 0) {
    MPI_Recv(&token, 1, MPI_INT, world_size - 1, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received token %d from process %d\n",
           world_rank, token, world_size - 1);
}
```

The ring program initializes a value from process zero, and the value is passed around every single process. The program terminates when process zero receives the value from the last process. As you can see from the program, extra care is taken to assure that it doesn't deadlock. In other words, process zero makes sure that it has completed its first send before it tries to receive the value from the last process. All of the other processes simply call `MPI_Recv` (receiving from their neighboring lower process) and then `MPI_Send` (sending the value to their neighboring higher process) to pass the value along the ring. `MPI_Send` and `MPI_Recv` will block until the message has been transmitted. Because of this, the printfs should occur by the order in which the value is passed. Using five processes, the output should look like this.

```
>>> ./run.py ring
Process 1 received token -1 from process 0
Process 2 received token -1 from process 1
Process 3 received token -1 from process 2
Process 4 received token -1 from process 3
Process 0 received token -1 from process 4
```

As we can see, process zero first sends a value of negative one to process one. This value is passed around the ring until it gets back to process zero.

Up next

Now that you have a basic understanding of `MPI_Send` and `MPI_Recv`, it is now time to go a little bit deeper into these functions. In the next lesson, I cover how to probe and dynamically receive messages. Feel free to also examine the MPI tutorials for a complete reference of all of the MPI lessons.

Having trouble? Confused? Feel free to leave a comment below and perhaps I or another reader can be of help.

This site is hosted entirely on GitHub. This site is no longer being actively contributed to by the original author (Wes Kendall), but it was placed on GitHub in the hopes that others would write high-quality MPI tutorials. Click here for more information about how you can contribute.