Dynamic Receiving with MPI Probe (and MPI Status)

Author: Wes Kendall

In the previous lesson, I discussed how to use MPI_Send and MPI_Recv to perform standard point-to-point communication. I only covered how to send messages in which the length of the message was known beforehand. Although it is possible to send the length of the message as a separate send / recv operation, MPI natively supports dynamic messages with just a few additional function calls. I will be going over how to use these functions in this lesson.

> **Note** - All of the code for this site is on GitHub. This tutorial's code is under tutorials/dynamic-receiving-with-mpi-probe-and-mpi-status/code.

The MPI_Status structure

As covered in the previous lesson, the `MPI_Recv` operation takes the address of an `MPI_Status` structure as an argument (which can be ignored with `MPI_STATUS_IGNORE`). If we pass an `MPI_Status` structure to the `MPI_Recv` function, it will be populated with additional information about the receive operation after it completes. The three primary pieces of information include:

1. **The rank of the sender**. The rank of the sender is stored in the `MPI_SOURCE` element of the structure. That is, if we declare an `MPI_Status stat` variable, the rank can be accessed with `stat.MPI_SOURCE`.
2. **The tag of the message**. The tag of the message can be accessed by the `MPI_TAG` element of the structure (similar to `MPI_SOURCE`).

3. **The length of the message**. The length of the message does not have a predefined element in the status structure. Instead, we have to find out the length of the message with `MPI_Get_count`.

```
MPI_Get_count(
    MPI_Status* status,
    MPI_Datatype datatype,
    int* count)
```

In `MPI_Get_count`, the user passes the `MPI_Status` structure, the `datatype` of the message, and `count` is returned. The `count` variable is the total number of `datatype` elements that were received.

Why would any of this information be necessary? It turns out that `MPI_Recv` can take `MPI_ANY_SOURCE` for the rank of the sender and `MPI_ANY_TAG` for the tag of the message. For this case, the `MPI_Status` structure is the only way to find out the actual sender and tag of the message. Furthermore, `MPI_Recv` is not guaranteed to receive the entire amount of elements passed as the argument to the function call. Instead, it receives the amount of elements that were sent to it (and returns an error if more elements were sent than the desired receive amount). The MPI_Get_count function is used to determine the actual receive amount.

An example of querying the MPI_Status structure

The program that queries the `MPI_Status` structure is in check_status.c. The program sends a random amount of numbers to a receiver, and the receiver then finds out how many numbers were sent. The main part of the code looks like this.

```
const int MAX_NUMBERS = 100;
int numbers[MAX_NUMBERS];
int number_amount;
if (world_rank == 0) {
    // Pick a random amount of integers to send to process one
    srand(time(NULL));
    number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;

    // Send the amount of integers to process one
    MPI_Send(numbers, number_amount, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("0 sent %d numbers to 1\n", number_amount);
} else if (world_rank == 1) {
    MPI_Status status;
    // Receive at most MAX_NUMBERS from process zero
    MPI_Recv(numbers, MAX_NUMBERS, MPI_INT, 0, 0, MPI_COMM_WORLD,
             &status);

    // After receiving the message, check the status to determine
    // how many numbers were actually received
    MPI_Get_count(&status, MPI_INT, &number_amount);

    // Print off the amount of numbers, and also print additional
    // information in the status object
    printf("1 received %d numbers from 0. Message source = %d, "
           "tag = %d\n",
           number_amount, status.MPI_SOURCE, status.MPI_TAG);
}
```

As we can see, process zero randomly sends up to `MAX_NUMBERS` integers to process one. Process one then calls `MPI_Recv` for a total of `MAX_NUMBERS` integers. Although process one is passing `MAX_NUMBERS` as the argument to `MPI_Recv`, process one will receive **at most** this amount of numbers. In the code, process one calls `MPI_Get_count` with `MPI_INT` as the datatype to find out how many integers were actually received. Along with printing off the size of the received message, process one also prints off the source and tag of the message by accessing the `MPI_SOURCE` and `MPI_TAG` elements of the status structure.

As a clarification, the return value from `MPI_Get_count` is relative to the datatype which is passed. If the user were to use `MPI_CHAR` as the datatype, the returned amount would be four times as large (assuming an integer is four bytes and a char is one byte). If you run the check_status program from the *tutorials* directory of the repo, the output should look similar to this.

```
>>> cd tutorials
>>> ./run.py check_status
mpirun -n 2 ./check_status
0 sent 92 numbers to 1
1 received 92 numbers from 0. Message source = 0, tag = 0
```

As expected, process zero sends a random amount of integers to process one, which prints off information about the received message.

Using MPI_Probe to find out the message size

Now that you understand how the `MPI_Status` object works, we can now use it to our advantage a little bit more. Instead of posting a receive and simply providing a really large buffer to handle all possible sizes of messages (as we did in the last example), you can use `MPI_Probe` to query the message size before actually receiving it. The function prototype looks like this.

```
MPI_Probe(
    int source,
    int tag,
    MPI_Comm comm,
    MPI_Status* status)
```

`MPI_Probe` looks quite similar to `MPI_Recv`. In fact, you can think of `MPI_Probe` as an `MPI_Recv` that does everything but receive the message. Similar to `MPI_Recv`, `MPI_Probe` will block for a message with a matching tag and sender. When the message is available, it will fill the status structure with information. The user can then use `MPI_Recv` to receive the actual message.

The lesson code has an example of this in probe.c. Here's what the main source code looks like.

```
int number_amount;
if (world_rank == 0) {
    const int MAX_NUMBERS = 100;
    int numbers[MAX_NUMBERS];
    // Pick a random amount of integers to send to process one
    srand(time(NULL));
    number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;

    // Send the random amount of integers to process one
    MPI_Send(numbers, number_amount, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("0 sent %d numbers to 1\n", number_amount);
} else if (world_rank == 1) {
    MPI_Status status;
    // Probe for an incoming message from process zero
    MPI_Probe(0, 0, MPI_COMM_WORLD, &status);

    // When probe returns, the status object has the size and other
    // attributes of the incoming message. Get the message size
    MPI_Get_count(&status, MPI_INT, &number_amount);

    // Allocate a buffer to hold the incoming numbers
    int* number_buf = (int*)malloc(sizeof(int) * number_amount);

    // Now receive the message with the allocated buffer
    MPI_Recv(number_buf, number_amount, MPI_INT, 0, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("1 dynamically received %d numbers from 0.\n",
           number_amount);
    free(number_buf);
}
```

Similar to the last example, process zero picks a random amount of numbers to send to process one. What is different in this example is that process one now calls `MPI_Probe` to find out how many elements process zero is trying to send (using `MPI_Get_count`). Process one then allocates a buffer of the proper size and receives the numbers. Running the code will look similar to this.

```
>>> ./run.py probe
```

```
mpirun -n 2 ./probe
0 sent 93 numbers to 1
1 dynamically received 93 numbers from 0
```

Although this example is trivial, `MPI_Probe` forms the basis of many dynamic MPI applications. For example, master/slave programs will often make heavy use of `MPI_Probe` when exchanging variable-sized worker messages. As an exercise, make a wrapper around `MPI_Recv` that uses `MPI_Probe` for any dynamic applications you might write. It makes the code look much nicer :-)

Up next

Do you feel comfortable using the standard blocking point-to-point communication routines? If so, then you already have the ability to write endless amounts of parallel applications! Let's look at a more advanced example of using the routines you have learned. Check out the application example using MPI_Send, MPI_Recv, and MPI_Probe.

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.