MPI 01: An Introduction to MPI with Open MPI

Draft 1 -- Patrick Garrity -- St. Olaf College

Until recently, parallelism in software could not be achieved on most single machines. Before multi-core existed, much of high performance computing (HPC) relied on message passing, which allows multiple computers to collaborate on a single problem. Even now message passing is a large part of HPC, and has been given even more power by the presence of multiple cores.

1. Message Passing

One programming technique to achieve parallelism is called message passing. In message passing, different processes communicate with one another through messages. These messages are tagged pieces of data sent across a network. Since this technology is network-based, it allows processes on many different machines to communicate with one-another, and in turn for multiple computers to run a coordinated effort to execute a program. In addition, on a multi-core machine a message passing program can send messages between multiple processes running on a single machine.

There are a large number of libraries for message passing, and different institutions may be using upwards of ten of these at a time. This code is called the message passing interface, or MPI. There are a number of proprietary implementations, but there are also open source implementations that are community-maintained. One of these, Open MPI, is the result of the convergence of multiple MPI projects. Open MPI implements the MPI-2.1 (current) standard and is actively maintained, making it an attractive option for doing message passing. Fortunately, different implementations of MPI are code-compatible if they follow the standard, so any MPI-2.1 implementation should work for this lesson.

2. Example

This lesson will explain how to use MPI through two examples that demonstrate the basic concepts of both the library and the message passing. The first example shows how to initialize and finalize MPI, while getting some information about the environment.

```
// File: mpi01.cpp
#include <iostream>
#include <mpi.h>

using std::cout;
using std::endl;

int main(int argc, char ** argv)
{
    int rank = 0;
    int size = 0;
    int length = 0;
    char name[MPI_MAX_PROCESSOR_NAME];

    MPI::Init(argc, argv);

    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    MPI::Get_processor_name(name, length);

    cout << "Process " << rank << " of " << size <<
        " on processor: " << name << endl;

    MPI::Finalize();
    return 0;
}
```

Code Breakdown

This example demonstrates how to initialize MPI, and get some basic information about its environment. All MPI applications require initialization and finalization, and termination of a process without finalizing it will cause an error. This program gets the number of processes created, and reports from each process its number and name of its overarching processor. This behavior will become more clear with a set of sample output. Important lines of code have been highlighted.

`MPI::Init(argc, argv);`

This line initializes MPI, and acts as an entry point to MPI code. This function supports command-line arguments as well. After a call to `MPI::Init()` has been made, the program is running identical code on multiple processes until `MPI::Finalize()` is called.

`MPI::Finalize();`

This is a required call after `MPI::Init()` has been called. The `MPI::Finalize()` function ends MPI execution and performs internal cleanup. Every process in an MPI application must finalize in order to successfully terminate.

`rank = MPI::COMM_WORLD.Get_rank();`

The `Get_rank()` function returns the rank, or process number, of the current process within the `COMM_WORLD`. This is a *communicator* (group) containing all processes available to the program.

`size = MPI::COMM_WORLD.Get_size();`

The `Get_size()` function returns the size of the `COMM_WORLD`, or the total number of processes available to the program.

`char name[MPI_MAX_PROCESSOR_NAME];`
`MPI::Get_processor_name(name, length);`

The `Get_processor_name()` function returns the name of the processor that is executing the current process. The name refers to the name of the *machine* running the process, and not that of the CPU. The `MPI_MAX_PROCESSOR_NAME` constant defines the maximum length of such a name.

How to Compile and Link the Example
$ mpiCC -c mpi01.cpp -o mpi01.o
$ mpiCC -o mpi01 mpi01.o

Example Output
```
$ mpirun -np 4 ./mpi01
Process 1 of 4 on processor: pulse
Process 2 of 4 on processor: pulse
Process 3 of 4 on processor: pulse
Process 0 of 4 on processor: pulse
```

This example was run on a single dual-core machine with a *hostname* of `pulse`. Using the `mpirun` command, it was specified that 4 processes were to be used by this program. Since the host machine only had 2 cores, only up to two of the processes were able to run concurrently.

3. Open MPI Command Line: mpiCC

Using Open MPI requires knowledge of some command line programs provided with Open MPI. The first of these is `mpiCC`, which invokes the C++ compiler along with the correct arguments required to compile an MPI program. This is the recommended way to compile and link Open MPI source code, because it ensures that the correct dependencies will be accounted for. Since the `mpiCC` program simply calls the underlying compiler, additional arguments can be added. Note that `mpiCC` is synonymous with the command `mpicxx`.

4. Open MPI Command Line: mpirun

The other key command line program provided by Open MPI is `mpirun`. This program allows developers to specify the number of processes, which computers to run on, and even which cores to run on, to an MPI application. At least a basic knowledge of this program is required for any developer using Open MPI. In addition to this lesson, the man page for `mpirun` is an excellent resource on its usage.

mpirun Basic Syntax
```
$ mpirun [ options ] <program> [ <args> ]
```

Specifying the Number of Processes
```
$ mpirun -np N <program> [ <args> ]
```
The above command runs program with **N** processes.

Specifying Which Machines to Use
```
$ mpirun -H host01,host02,... <program> [ <args> ]
$ mpirun --hostfile HOSTFILE <program> [ <args> ]
```
Hosts can be provided through either a list, or a file with one host entry per line. For more information on hostfiles, please see the `mpirun` man page. The first command runs program on

machines host01, host02, and so on. The second command runs program on the machines specified in the local file HOSTFILE.

Again, reading the manpage for mpirun will provide extensive information on more advanced usage, including specifying a number of processes per machine, binding to specific cores, and associating specific ranks with specific machines. This type of control can be extremely useful for controlling and debugging Open MPI applications.

5. Another Example

The following example demonstrates how to use MPI to allow processes to communicate with one another. The initial example did not explicitly use this communication, but this program will send data from one process to another based on rank.

```
// File: mpi02.cpp
#include <iostream>
#include <mpi.h>

using std::cout;
using std::endl;

int main(int argc, char ** argv)
{
    int rank = 0;
    int size = 0;
    int dest_proc = 0;
    int source_proc = 0;
    int tag = 1;
    MPI::Status status;

    char output_message[6] = { 'H', 'e', 'l', 'l', 'o', '\0' };
    char input_message[6];

    MPI::Init(argc, argv);

    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();

    if (size <= 1)
    {
        MPI::COMM_WORLD.Abort(1);
    }

    if (rank == 0)
    {
        for (int i = 1; i < size; ++i)
        {
            dest_proc = i;
            source_proc = i;

            MPI::COMM_WORLD.Send(output_message, 6, MPI_CHAR, dest_proc,
                tag);
            MPI::COMM_WORLD.Recv(input_message, 6, MPI_CHAR, source_proc,
                tag, status);

            cout << rank << " got " << status.Get_count(MPI_CHAR)
                << " chars from " << status.Get_source() << ": "
                << input_message << endl;
        }
    }
    else
    {
        dest_proc = 0;
        source_proc = 0;

        MPI::COMM_WORLD.Recv(input_message, 6, MPI_CHAR, source_proc,
            tag, status);
        MPI::COMM_WORLD.Send(output_message, 6, MPI_CHAR, dest_proc,
            tag);

        cout << rank << " got " << status.Get_count(MPI_CHAR) << " chars "
            << "from " << status.Get_source() << ": "
            << input_message << endl;
    }

    MPI::Finalize();
    return 0;
}
```

Code Breakdown

This example essentialy uses the head node (rank 0) to *ping* the other processes in the world. The head node sends a message ("Hello") to each of the other processes. Once they receive this message, they send back an identical message to the head node. The head node goes through this process one node at a time - this program *could* be re-written such that it did not. Upon receiving a message, each process reports that it received the message by printing to standard output the number of characters received, and where they came from. Lines of MPI code new to this example have been highlighted in the source listing.

`MPI::Status status;`

This creates a default `Status` object that can be used to obtain information about communication in MPI. In this case, it is used to collect information in the call to `Recv()`. Further down the source, where processes are reporting their messages, the `Status` object is used to get the number of characters received [`status.Get_count(MPI_CHAR)`] and the number of the process which sent those characters [`status.Get_source()`].

`MPI_CHAR`
This is a constant of type `MPI::Datatype` used by MPI to identify data of type char. This is necessary because MPI does not know what type the data is after sending it to another process. If the sending process attaches information on what type it is (hence `MPI_CHAR`), then the receiving process can cast the received data to the appropriate type. In addition to `MPI_CHAR` there are other constant type identifiers, such as `MPI_INT`.

`MPI::COMM_WORLD.Abort(1);`
The `Abort()` function forces MPI to immediately abandon execution on all processes with an error. The integer (1) is an error status code. The choice of the number 1 in this example was arbitrary. In this case, `Abort()` is called when there is only one process. Since the loop that sends and receives messages would never execute with only one process (since `size==1`) this `Abort()` is not strictly necessary, but it was included for demonstration purposes.

`MPI::COMM_WORLD.Send(output_message, 6, MPI_CHAR, dest_proc, tag);`
This line sends a message to the specified member of the `COMM_WORLD`. Its arguments are interpreted as follows:

- <u>Argument 1</u> (`output_message`): This is some block of data that will be send to the destination process. It is actually send as a `void *` (void pointer), which means that it is sent as typeless data. In this example, `output_message` is an array of characters containing the data "`Hello`" (null-terminated).
- <u>Argument 2</u> (`6`): This is an integer representing the number of data elements being sent. Since an array of data is being sent in this example, the size of the array is passed. It is also possible to send only part of an array: for instance, if this example had used the number `3` instead of `6`, the data "`Hel`" would have been sent. If an array is not sent, then the number `1` is given since there is only one data element.
- <u>Argument 3</u> (`MPI_CHAR`): This argument is an object of type `MPI::Datatype` that is used to identify the type of data being sent. In this example, `MPI_CHAR` is used (as described above). This signifies that the data which is sent as an array of type `void` should be converted to an array of type `char` upon retrieval.
- <u>Argument 4</u> (`dest_proc`): This argument specifies which process (by rank) the message should be sent to.
- <u>Argument 5</u> (`tag`): The final argument is a tag that is attached to the message. A tag is an integer that can be used to differentiate between types of messages. For example, a program may have tags (`1`, `2`, and `3`) where `1` represents an error code, `2` represents a number to add by, and `3` represents a number to multiply by (the data in this case would be an integer). This program only uses a single tag since it has no need for differentiation between message types.

`MPI::COMM_WORLD.Recv(input_message, 6, MPI_CHAR, source_proc, tag, status);`
The `Recv()` function is used to receive data that has been sent by another process in the `COMM_WORLD` (using the `Send()` function). Its arguments can be interpreted similarly to those of Send():

- <u>Argument 1</u> (input_message): A buffer in which to store received data. Data is initially retrieved as a void pointer, so MPI must be told what type to convert it to.

- Argument 2 (6): The size of the buffer in number of data elements.
- Argument 3 (MPI_CHAR): The type of data in the buffer.
- Argument 4 (source_proc): The process (by rank) from which to receive the message. In the case of Recv() the constant MPI_ANY_SOURCE may be used to signify that the function is willing to receive data from any source.
- Argument 5 (tag): The type of tag to receive. In the case of Recv() the constant MPI_ANY_TAG may be used to signify that the function is willing to receive data of any tag type.
- Argument 6 (status): A MPI::Status object that is used to store additional information about the received message.

One final topic to discuss concerning the source code is the concept of *blocking vs. non-blocking*. In this example, *blocking* sends and receives are used. What this means, is that each call to `Send()` will not complete until it knows its message has been received. Similarly, each call to `Recv()` will not complete until it has received some message. This behavior is called *blocking* because it blocks the program from continuing until some condition has been met. Once blocked, a program will wait for input (or some result) possibly indefinitely. Non-blocking calls that send and receive messages in MPI do exist, but they are beyond the scope of this lesson.

How to Compile and Link the Example
```
$ mpiCC -c mpi02.cpp -o mpi02.o
$ mpiCC -o mpi02 mpi02.o
```

Example Output
```
$ mpirun -np 2 ./mpi02
0 got 6 characters from 1: Hello
1 got 6 characters from 0: Hello

$ ./mpi02
--------------------------------------------------------------------------
MPI_ABORT was invoked on rank 0 in communicator MPI_COMM_WORLD
with errorcode 1.

NOTE: invoking MPI_ABORT causes Open MPI to kill all MPI processes.
You may or may not see output from other processes, depending on
exactly when Open MPI kills them.
--------------------------------------------------------------------------

$ mpirun -np 4 ./mpi02
1 got 6 characters from 0: Hello
0 got 6 characters from 1: Hello
0 got 6 characters from 2: Hello
2 got 6 characters from 0: Hello
0 got 6 characters from 3: Hello
3 got 6 characters from 0: Hello
```

Three examples were given demonstrating the behavior when 1, 2, and 4 processes are used. It can be seen that when running the example with a single process, Abort() is called as expected and the example crashes. When running with 2 and 4 processes the output occurs (in order) as expected.

6. Conclusion

A surprising amount of work can be accomplished using little more than sends and receives in MPI. However, the library is extensive and offers great depth to message passing, including process management. Even so its moderate learning curve combined with applicability make it a powerful tool even for relatively inexperienced users. Questions that should be answered by this extended introduction are:

- What is MPI? What is Open MPI?
- What are the two essential parts of an MPI application?
- How are the Open MPI command line tools used?
- What is one way to pass messages between processes?