Lecture Notes for CSCI 379: Distributed Computing

Intro to MPI

Professor Talmage

January 27, 2023

---

1 What is MPI

MPI stands for *Message Passing Interface*, which is a specification for a message-passing communication system. There are then various library which implement this specification, such as `openmpi` and `mpich`. We'll be using `mpich`, since `openmpi` doesn't work well on our machines (for reasons that are over my head).

This is intended as a *extremely* basic crash course to get you as far as writing your first MPI program. There are many resources online to teach you more. I'll provide links to a few here, but you are welcome to search for more tutorials. Please send me links to any you find particularly helpful.

Note that I have not read through these, and some are not in python. (Most things you'll find are likely C/C++, and perhaps even FORTRAN–remember that MPI is an interface specification that can be implemented in different languages.)

- `https://mpi4py.readthedocs.io/en/stable/`
- `https://materials.jeremybejarano.com/MPIwithPython/index.html`
- `https://mpitutorial.com/tutorials/mpi-introduction/`
- `https://www.mpi-forum.org/docs/` (Fair warning, the specification is a 1100+ page document.)

2 Communication Examples

MPI provides a number of tools for communication between processes, and we'll look at some simple examples of how to use them.

First, MPI uses structures known as *communicators* to organize processes. While these can be used in interesting ways, for now we'll just rely on the world communicator:

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
```

In this example comm is the communicator for all processes, as specified by the `-n` parameter to `mpiexec`. Each process has a unique integer ID, starting at 0, and we have saved that to rank. For now, all our programs will start off with creating these variables so that we can interact with other processes.

Exercise: Write the above two lines into a file, add a print statement to print rank, and run your program with the command
mpiexec −n X python Y.py
where X is an integer (try with different values) and Y is the name of your python file. Try with different values until the output is not what you expect, then discuss why it is not.

2.1 Direct, Blocking

The simplest way to communicate with MPI is with blocking, point-to-point messaging. *Blocking* means that the process will not proceed past the messaging function call (send, receive, etc.) until it is "done" sending or receiving.[1] A simple example looks something like this:

Listing 1: Blocking Point-to-Point Example

```
if rank == 0:
    sendData = ''Hello world from process 0.''
    comm.send(sendData, dest=1)
else:
    recvData = comm.recv(source=0)
    print(f'p{rank} received a message that says ''{recvData}''')
```

A few things to note:

- First, note that we need different code for each process, since one should be sending and another receiving.
- We can specify certain source/destination restrictions, which can be good when a lot of processes are sending messages and you only want to listen for certain ones. You can also use tags, which are similar but allow processes to filter messages independently of ID (for example, recv with tag=3 will only receive messages sent with tag=3).

2.2 Direct, Non-Blocking

Non-blocking communication allows your code to continue with other tasks while the communication system works in the background. This can be very useful, though my understanding is that there are some risks with corrupting the memory associated with such messages, particularly if you are trying to send many messages very quickly. At the same time, if you need to check whether you have a message waiting, and continue doing other things if not, non-blocking communication is essential, as a blocking receive will halt your entire program until it receives the expected message.

Listing 2: Non-Blocking Point-to-Point (Bad) Example

```
if rank == 1:
    sendData = {'a':1, 'b':''frog''}
    comm.isend(sendData)
else:
    recvRequest = comm.irecv()
    print(f'p{rank} received message ''{recvRequest.test()}''')
```

[1] Note that "done" may be misleading. The library calls don't necessarily block until the message is received, but until the outbuffer is safe to use again.

- Non-blocking communication functions are named exactly as their blocking counterparts, except for a leading i: isend vs. send, irecv vs. recv, etc.
- Note that irecv does not actually return the data. Because it may not receive a messaged, it returns a *request* object, which we can then test to see if it has any data.
- Note that this code actually fails when we run it. There are two problems:
  1. There is an error message: isend requires two parameters. Recall that these are point-to-point messages. While we don't have to specify a source, allowing us to listen for a message from anyone, we must specify a destination, as we can only send this message to one place.
  2. The printed message may have been (False,None), which is not the {'a':1, 'b':'' frog''} we wanted. (If its not empty run a few more times and it will be.) This is because we tried to receive before the message arrived, and the code did not block when there was no message. It may work sometimes, and not other times, just depending on the timing of the message and the different processes.

Listing 3: Non-Blocking Point-to-Point (Fixed) Example

```
if rank == 1:
    sendData = {'a':1, 'b':''frog''}
    sendRequest = comm.isend(sendData, dest=0)
    sendRequest.wait()
else:
    recvRequest = comm.irecv()
    while True:
        recvData = recvRequest.test()
        if recvData[0]:
            break
    print(f'p{rank} received message ''{recvData[1]}''')
```

- There are two things we need to do to ensure the message actually arrives:
  1. The sender needs to wait(). If we do this immediately after the send, as in this example, we've turned the send into a blocking operation. In a more complex program, though, we can wait() later in the code. This ensures that process 0 does not terminate before successfully copying the message to the MPI communication system.
  2. We previously had the receiver only check for the message once. If this is done before it arrives, it will not find a message. We can fix this by looping, repeatedly calling test (Though, again, if we do nothing else, that turns this into a blocking receive. But this allows us to check for a message and continue other operations if it has not yet arrived).
- Note that the test() function returns a Request object, which is a (Bool,Optional) pair. That is, index 0 is a Boolean indicating whether a message arrived for this request and once that's true, index 1 is the data received, whatever type it may be.

2.3 Collective

Collective communication allows sending messages to and from more than one process at a time. The simplest version is broadcast, where one process wants to send to all others, but there are other operations in this category, such as gather, which is similar to convergecast, collecting ("gathering") values from many nodes to a single one, or scatter, which sends one element of a list to each process. We will only look at a simple example right now. I leave it to you to explore other operations and possiblities in the documentation.

Listing 4: Collective Communication Example

```
if rank == 3:
    commData = ''Everyone needs to know this!''
else:
    commData = None

print(f'Before␣communication,␣p{rank}␣has␣commData={commData}')

commData = comm.bcast(commData, root=3)

print(f'After␣communication,␣p{rank}␣has␣commData={commData}')
```

- Here, there are not different calls for sending and receiving. All processes indicate that they want to participate, and the root parameter indicates which node is sending.
- Note that commData appears as both a parameter and the return value. We could use different variables, but every process must define it to pass as a parameter.

3 Quirks and Cautions

- There are upper- and lower-case versions of many of the functions in the mpi4py library. Lowercase functions send/receive generic python objects. Uppercase send buffer-like objects, such as types from numpy. Be sure you're using the ones you intended.
- It may be wise to end larger programs with a waitall or similar call to ensure that there are not outstanding messages when you terminate. This can cause crashes, as MPI is unhappy terminating with outstanding messages.

4 Exercises

Exercise: Write code that implements a simple leader election algorithm by choosing the process with lowest clock value. (The python time module may be helpful, as may looking for other communication functions in `https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Comm.html`.)

Exercise: Use point-to-point messaging to write your own broadcast function. Be sure that it works for different values of $n \geq 1$.