1

Parallel Programming with MPI

Lab Objective: *In the world of parallel computing, MPI is the most widespread and standardized message passing library. As such, it is used in the majority of parallel computing programs. In this lab, we explore and practice the basic principles and commands of MPI to further recognize when and how parallelization can occur.*

MPI: the Message Passing Interface

At its most basic, the Message Passing Interface (MPI) provides functions for sending and receiving messages between different processes. MPI was developed to provide a standard framework for parallel computing in any language. It specifies a library of functions — the syntax and semantics of message passing routines — that can be called from programming languages such as Fortran and C.

MPI can be thought of as "the assembly language of parallel computing," because of this generality.[1] MPI is important because it was the first portable and universally available standard for programming parallel systems and continues to be the de facto standard today.

For more information on how MPI works and how to get it installed on your machine, see the additional material for this lab.

> **NOTE**
>
> Most modern personal computers now have multicore processors. Programs that are designed for these multicore processors are "parallel" programs and are typically written using OpenMP or POSIX threads. MPI, on the other hand, is designed for any general architecture.

Why MPI for Python?

In general, programming in parallel is more difficult than programming in serial because it requires managing multiple processors and their interactions. Python, however, is an excellent language for simplifying algorithm design because it allows for problem solving without too much detail. Unfortunately, Python is not designed for high performance computing and is a notably slower

---

[1] *Parallel Programming with MPI*, by Peter S. Pacheco, pg. 7.

scripted language. It is best practice to prototype in Python and then to write production code in
fast compiled languages such as C or Fortran.

In this lab, we will explore the Python library `mpi4py` which retains most of the functionality
of C implementations of MPI and is a good learning tool. If you do not have the MPI library and
mpi4py installed on your machine, please refer to the Additional Material at the end of this lab.
There are three main differences to keep in mind between mpi4py and MPI in C:

- Python is array-based while C is not.

- mpi4py is object oriented but MPI in C is not.

- mpi4py supports two methods of communication to implement each of the basic MPI com-
  mands. They are the upper and lower case commands (e.g. `Bcast(...)` and `bcast(...)`). The
  uppercase implementations use traditional MPI datatypes while the lower case use Python's
  pickling method. Pickling offers extra convenience to using mpi4py, but the traditional method
  is faster. In these labs, we will only use the uppercase functions.

Using MPI

We will start with a Hello World program.

```python
#hello.py
from mpi4py import MPI

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()

print("Hello world! I'm process number {}.".format(RANK))
```

hello.py

Save this program as `hello.py` and execute it from the command line as follows:

```
$ mpiexec -n 5 python hello.py
```

The program should output something like this:

```
Hello world! I'm process number 3.
Hello world! I'm process number 2.
Hello world! I'm process number 0.
Hello world! I'm process number 4.
Hello world! I'm process number 1.
```

Notice that when you try this on your own, the lines will not necessarily print in order. This is
because there will be five separate processes running autonomously, and we cannot know beforehand
which one will execute its `print()` statement first.

> **ACHTUNG!**
>
> It is usually bad practice to perform I/O (e.g., call `print()`) from any process besides the root

> process (rank 0), though it can be a useful tool for debugging.

How does this program work? First, the `mpiexec` program is launched. This is the program which starts MPI, a wrapper around whatever program you to pass into it. The `-n 5` option specifies the desired number of processes. In our case, 5 processes are run, with each one being an instance of the program "python". To each of the 5 instances of python, we pass the argument `hello.py` which is the name of our program's text file, located in the current directory. Each of the five instances of python then opens the `hello.py` file and runs the same program. The difference in each process's execution environment is that the processes are given different ranks in the communicator. Because of this, each process prints a different number when it executes.

MPI and Python combine to make succinct source code. In the above program, the line `from mpi4py import MPI` loads the MPI module from the mpi4py package. The line `COMM = MPI.COMM_WORLD` accesses a static communicator object, which represents a group of processes which can communicate with each other via MPI commands. The next line, `RANK = COMM.Get_rank()`, accesses the processes *rank* number. A rank is the process's unique ID within a communicator, and they are essential to learning about other processes. When the program `mpiexec` is first executed, it creates a global communicator and stores it in the variable `MPI.COMM_WORLD`. One of the main purposes of this communicator is to give each of the five processes a unique identifier, or rank. When each process calls `COMM.Get_rank()`, the communicator returns the rank of that process. `RANK` points to a local variable, which is unique for every calling process because each process has its own separate copy of local variables. This gives us a way to distinguish different processes while writing all of the source code for the five processes in a single file.

Here is the syntax for `Get_size()` and `Get_rank()`, where `Comm` is a communicator object:

Comm.Get_size() Returns the number of processes in the communicator. It will return the same number to every process. Parameters:

Return value - the number of processes in the communicator

Return type - integer

Example:

```python
#Get_size_example.py
from mpi4py import MPI
SIZE = MPI.COMM_WORLD.Get_size()
print("The number of processes is {}.".format(SIZE))
```

Get_size_example.py

Comm.Get_rank() Determines the rank of the calling process in the communicator. Parameters:

Return value - rank of the calling process in the communicator

Return type - integer

Example:

```python
#Get_rank_example.py
from mpi4py import MPI
RANK = MPI.COMM_WORLD.Get_rank()
print("My rank is {}.".format(RANK))
```

Get_rank_example.py

The Communicator

A communicator is a logical unit that defines which processes are allowed to send and receive messages. In most of our programs we will only deal with the `MPI.COMM_WORLD` communicator, which contains all of the running processes. In more advanced MPI programs, you can create custom communicators to group only a small subset of the processes together. This allows processes to be part of multiple communicators at any given time. By organizing processes this way, MPI can physically rearrange which processes are assigned to which CPUs and optimize your program for speed. Note that within two different communicators, the same process will most likely have a different rank.

Note that one of the main differences between `mpi4py` and MPI in C or Fortran, besides being array-based, is that `mpi4py` is largely object oriented. Because of this, there are some minor changes between the `mpi4py` implementation of MPI and the official MPI specification.

For instance, the MPI Communicator in `mpi4py` is a Python class and MPI functions like `Get_size()` or `Get_rank()` are instance methods of the communicator class. Throughout these MPI labs, you will see functions like `Get_rank()` presented as `Comm.Get_rank()` where it is implied that `Comm` is a communicator object.

Separate Codes in One File

When an MPI program is run, each process receives the same code. However, each process is assigned a different rank, allowing us to specify separate behaviors for each process. In the following code, the three processes perform different operations on the same pair of numbers.

```python
#separateCode.py
from mpi4py import MPI
RANK = MPI.COMM_WORLD.Get_rank()

a = 2
b = 3
if RANK == 0:
    print a + b
elif RANK == 1:
    print a*b
elif RANK == 2:
    print max(a, b)
```

separateCode.py

Problem 1. Write a program in which processes with an even rank print "Hello" and process with an odd rank print "Goodbye." Print the process number along with the "Hello" or "Goodbye" (for example, "Goodbye from process 3").

Message Passing between Processes

Let us begin by demonstrating a program designed for two processes. One will draw a random number and then send it to the other. We will do this using the routines `Comm.Send()` and `Comm.Recv()`.

```python
#passValue.py
import numpy as np
from mpi4py import MPI

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()

if RANK == 1:  # This process chooses and sends a random value
    num_buffer = np.random.rand(1)
    print("Process 1: Sending: {} to process 0.".format(num_buffer))
    COMM.Send(num_buffer, dest=0)
    print("Process 1: Message sent.")
if RANK == 0:  # This process recieves a value from process 1
    num_buffer = np.zeros(1)
    print("Process 0: Waiting for the message... current num_buffer={}.".format
        (num_buffer))
    COMM.Recv(num_buffer, source=1)
    print("Process 0: Message recieved! num_buffer={}.".format(num_buffer))
```

passValue.py

To illustrate simple message passing, we have one process choose a random number and then pass it to the other. Inside the receiving process, we have it print out the value of the variable `num_buffer` before it calls `Recv()` to prove that it really is receiving the variable through the message passing interface.

Here is the syntax for `Send()` and `Recv()`, where `Comm` is a communicator object:

Comm.Send(buf, dest=0, tag=0) Performs a basic send from one process to another. Parameters:

> **buf (array-like)** : data to send
>
> **dest (integer)** : rank of destination
>
> **tag (integer)** : message tag

The `buf` object is not as simple as it appears. It must contain a pointer to a Numpy array. It cannot, for example, simply pass a string. The string would have to be packaged inside an array first.

Comm.Recv(buf, source=0, tag=0, Status status=None) Basic point-to-point receive of data. Parameters:

> **buf (array-like)** : initial address of receive buffer (choose receipt location)
>
> **source (integer)** : rank of source
>
> **tag (integer)** : message tag
>
> **status (Status)** : status of object

Example:

```
1  #Send_example.py
2  from mpi4py import MPI
   import numpy as np
4
   RANK = MPI.COMM_WORLD.Get_rank()
6
   a = np.zeros(1, dtype=int)   # This must be an array.
8  if RANK == 0:
       a[0] = 10110100
10     MPI.COMM_WORLD.Send(a, dest=1)
   elif RANK == 1:
12     MPI.COMM_WORLD.Recv(a, source=0)
       print(a[0])
```

Send_example.py

Problem 2. Write a script that runs on two processes and passes an n by 1 vector of random values from one process to the other. Write it so that the user passes the value of n in as a command-line argument. The following code demonstrates how to access command-line arguments.

```
from sys import argv

# Pass in the first command line argument as n.
n = int(argv[1])
```

NOTE

`Send()` and `Recv()` are referred to as *blocking* functions. That is, if a process calls `Recv()`, it will sit idle until it has received a message from a corresponding `Send()` before it will proceed. (However, in Python the process that calls `Comm.Send` will *not* necessarily block until the message is received, though in C, `MPI_Send` does block) There are corresponding *non-blocking* functions `Isend()` and `Irecv()` (The I stands for immediate). In essence, `Irecv()` will return immediately. If a process calls `Irecv()` and doesn't find a message ready to be picked up, it will indicate to the system that it is expecting a message, proceed beyond the `Irecv()` to do other useful work, and then check back later to see if the message has arrived. This can be used to dramatically improve performance.

Problem 3. Write a script in which the process with rank i sends a random value to the process with rank $i + 1$ in the global communicator. The process with the highest rank will

send its random value to the root process. Notice that we are communicating in a ring. For communication, only use `Send()` and `Recv()`. The program should work for any number of processes. Does the order in which `Send()` and `Recv()` are called matter?

NOTE

When calling `Comm.Recv`, you can allow the calling process to accept a message from any process that happened to be sending to the receiving process. This is done by setting source to a predefined MPI constant, `source=ANY_SOURCE` (note that you would first need to import this with from `mpi4py.MPI` `import` `ANY_SOURCE` or use the syntax `source=MPI.ANY_SOURCE`).

Application: Monte Carlo Integration

Monte Carlo integration uses random sampling to approximate volumes (whereas most numerical integration methods employ some sort of regular grid). It is a useful technique, especially when working with higher-dimensional integrals. It is also well-suited to parallelization because it involves a large number of independent operations. In fact, Monte Carlo algorithms can be made "embarassingly parallel" — the processes don't need to communicate with one another during execution, simply reporting results to the root process upon completion.

In a simple example, the following code calculates the value of π by sampling random points inside the square $[-1, 1] \times [-1, 1]$. Since the volume of the unit circle is π and the volume of the square is 4, the probability of a given point landing inside the unit circle is $\pi/4$, so the proportion of samples that fall within the unit circle should also be $\pi/4$. The program samples $N = 2000$ points, determines which samples are within the unit circle (say M are), and estimates $\pi \approx 4M/N$.

```python
# pi.py
import numpy as np
from scipy import linalg as la


# Get 2000 random points in the 2-D domain [-1,1]x[-1,1].
points = np.random.uniform(-1, 1, (2,2000))

# Determine how many points are within the unit circle.
lengths = la.norm(points, axis=0)
num_within = np.count_nonzero(lengths < 1)

# Estimate the circle's area.
print(4 * (num_within / 2000))
```

pi.py

```
$ python pi.py
3.166
```

Problem 4. The n-dimensional *open unit ball* is the set $U_n = \{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x}\|_2 < 1\}$. Write a script that accepts integers n and N on the command line. Estimate the volume of U_n by drawing N points over the n-dimensional domain $[-1,1] \times [-1,1] \times \cdots \times [-1,1]$ on each available process except the root process (for a total of $(r-1)N$ draws, where r is the number of processes). Have the root process print the volume estimate.
(Hint: the volume of $[-1,1] \times [-1,1] \times \cdots \times [-1,1]$ is 2^n.)

When $n = 2$, this is the same experiment outlined above so your function should return an approximation of π. The volume of the U_3 is $\frac{4}{3}\pi \approx 4.18879$, and the volume of U_4 is $\frac{\pi^2}{2} \approx 4.9348$. Try increasing the number of sample points N or processes r to see if your estimates improve.

NOTE

Good parallel code should pass as little data as possible between processes. Sending large or frequent messages requires a level of synchronization and causes some processes to pause as they wait to receive or send messages, negating the advantages of parallelism. It is also important to divide work evenly between simultaneous processes, as a program can only be as fast as its slowest process. This is called load balancing, and can be difficult in more complex algorithms.

Additional Material

Installation of MPI

MPI is a library of functions that interface with your computer's hardware to provide optimal parallel computing performance. In order to use mpi4py, we need to have an MPI Library on installed on the computer as well as the mpi4py package. When you invoke mpi4py in your python code, mpi4py takes what you have written in python and applies it using an MPI Library, so only installing mpi4py is not enough to use MPI.

Installing MPI Library

1. For Linux/Mac: We recommend using OpenMPI for your MPI Library installation, though it is not the only library available.

   - Download the binary files from `https://www-lb.open-mpi.org/software/ompi/v4.0/`.
   - Extract the files from their compressed form and navigate into the new folder titled "openmpi-X.X.X".
   - Configure the files so that they will install correctly on your machine.
   - Compile OpenMPI and install it.

   The following is a bash script written for Linux that will install OpenMPI version 4.0.2. It will take about 15 minutes to complete.

   ```
   #!/bin/bash
   # download openMPI
   wget https://download.open-mpi.org/release/open-mpi/v4.0/openmpi↩
       -4.0.2.tar.gz
   ```

```
# extract the files
tar -zxf openmpi-4.0.2.tar.gz
cd openmpi-4.0.2
# configure the files
./configure --prefix=/usr/local/openmpi
# compile openMPI
make all
# install openMPI
sudo make install
```

Finally, you must add OpenMPI to your PATH variable. This is so your computer knows where to look when it wants to execute a certain MPI command. Here is a link that describes how to edit the PATH variable `https://gist.github.com/nex3/c395b2f8fd4b02068be37c961301caa7`.

On linux you will open a file called .bashrc, on Mac the file is called .bash_profile, both are in the home directory. Add the following line, save the file, and restart your terminal.

```
export PATH=/usr/local/openmpi/bin:$PATH
```

2. For Windows: There is only one free MPI library available for Windows at `https://msdn.microsoft.com/en-us/library/bb524831(v=vs.85).aspx`. Download the appropriate .exe or .msi file to install on your machine.

Installing mpi4py

1. For All Systems: The easiest installation is using `conda install mpi4py`. You may also run `pip install mpi4py`