Lab 1

Parallel Programming with MPI

Lab Objective: *Use MPI to implement basic parallel programs.*

Why Parallel Computing?

When a single processor takes too long to perform a computationally intensive task, there are two simple solutions. The first is simply to build a faster processor. Unfortunately, physics gets in the way. In particular, the problem of heat dissipation has kept processor speeds from increasing as quickly in recent years as they did in the past. The second solution is to have mutiple processors work together on the same task. This is the main idea behind parallel computing.

Today, high computing performance is achieved using many processors. These processors communicate with each other and coordinate their tasks with a message passing system. Essentially, a 'supercomputer' is made up of many normal computers, each with its own memory. These normal computers are all running the same program, but each takes a different execution path through the code as a result of the interactions that message passing makes possible.

Taking advantage of parallel processors is challenging; one cannot simply take a traditional program and expect it to run faster on a supercomputer because such programs consists of a single process — a set of instructions to be executed sequentially. A parellel program must be written which consists of many processes which can be executed simultaneously.

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. It does not specify a particular programming language. Rather, MPI specifies a library of functions — the syntax and semantics of message passing routines — that can be called from other programming languages such as Python 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, universally available standard for programming parallel systems and continues to be the *de facto* standard today.

> **Note**
>
> Most modern personal computers now have multicore processors. Programs that are designed for these multicore processors are "parallel" programs in a sense, typically written using OpenMP or POSIX threads. MPI, on the other hand, is designed with a different kind of architecture in mind.

Why MPI for Python?

In general, programming in parallel is more difficult than programming in serial. Python is an excellent language for algorithm design because it allows one to solve problems without getting bogged down in details. However, Python is not designed specifically for high performance computing, so in practice it is good to prototype in Python and then to write production code in fast compiled languages such as C or Fortran.

We use the Python library `mpi4py` because it retains most of the functionality of C implementations of MPI, making it a good learning tool. There are three main differences to keep in mind between mpi4py and MPI in C:

- Python is array-based. C and Fortran are not.
- mpi4py is object oriented. MPI in C is not.
- mpi4py supports two methods of communication to implement each of the basic MPI commands. 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

As tradition has it, we will start with a Hello World program.

```
1 #hello.py
2 from mpi4py import MPI
3
4 COMM = MPI.COMM_WORLD
5 RANK = COMM.Get_rank()
6
7 print "Hello world! I'm process number {}.".format(RANK)
```

hello.py

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

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

```
$ mpirun -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, though it can oftentimes be a useful tool for debugging.

How does this program work? First, the `mpirun` 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 wonderfully 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()`, is where things get interesting. A rank is the process's id within a communicator, and they are essential to learning about other processes. When the program `mpirun` 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.

In more advanced MPI programs, you can create your own communicators, which means that any process can be part of more than one communicator at any given time. In this case, the process will likely have a different rank within each communicator.

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:

```
#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:

```
#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. 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.

```
#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 the the 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`.

```
#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 recieving process, we have it print out the value of the variable `num_buffer` *before* it calls `Recv` to prove that it really is recieving 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:

```
#Send_example.py
from mpi4py import MPI
import numpy as np

RANK = MPI.COMM_WORLD.Get_rank()

a = np.zeros(1, dtype=int)  # This must be an array
if RANK == 0:
    a[0] = 10110100
    MPI.COMM_WORLD.Send(a, dest=1)
elif RANK == 1:
    MPI.COMM_WORLD.Recv(a, source=0)
    print a[0]
```

Send_example.py

Problem 2. Write a script `passVector.py` that 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.

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 proceeed. (However, the process that calls `Comm.Send` will *not* necessarily block until the message is recieved- it depends on the implementation) 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 `passCircular.py` 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. (Hint: Remember that `Send` and `Recv` are blocking functions. 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 happend 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 plotting random points inside a square. The probability of a given point landing inside the inscibed circle should be $\pi/4$.

```python
import random

n = 100000
s = 0

for i in range(n):
    x = random.uniform(-1.0,1.0)
    y = random.uniform(-1.0,1.0)

    if x**2 + y**2 <= 1:
        s += 1

print 4.0*s/n
```

pi.py

Problem 4. Write a script that finds the volume of an m-sphere using n processes to test p random points. All of these variables should be passed from the command line. The n processes should pass their individual results up to the root process, which then calculates an overall average.

Note

Good parallel code should pass as little data as possible between processes. Sending large or frequent messages can bog down performance, 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.