

LOUISIANA STATE UNIVERSITY - SHREVEPORT

INDEPENDENT STUDY

Independent Study in Cluster Computing and Virtualization

Authors:

Jay JAIN
Blaise LEONARDS
Joshua SCHOOLS

Supervisor:

Dr. Alfred MCKINNEY

*A document submitted in fulfillment of the requirements
for CSC 495 (Independent Study)*

in the

Department of Computer Science

November 28, 2017

"For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers. "

Gene Amdahl

LOUISIANA STATE UNIVERSITY - SHREVEPORT

Abstract

Dr. Alfred McKinney
Department of Computer Science

Independent Study

Independent Study in Cluster Computing and Virtualization

by
Jay JAIN
Blaise LEONARDS
Joshua SCHOOLS

Understanding the fundamentals of parallel computing in the age of big data is integral to a comprehensive computer science education. Over the course of the Fall 2017 semester, major mathematical and programming concepts were identified and implemented on a self-built Raspberry Pi cluster. The cluster consisted of a master node (Raspberry Pi 3) and four slave nodes (Raspberry Pi Zeros). Using the Python message passing interface (`mpi4py`), the value of π was calculated using the Monte Carlo method at different iterations. The timing and accuracy of the calculated values of π were compared using different parallel computing methods. In addition, the Docker virtualization interface was explored in order to understand the interrelation between parallel and cloud computing.

Acknowledgements

The authors would like to thank Dr. McKinney for giving us the opportunity to participate in this independent study and his enduring kindness. We would also like to thank Mr. Phillip C.S.R. Kilgore for his words of encouragement and guidance through the project. Lastly, we would like to thank Dr. Cvek and Dr. Trutschl for allowing us to house our Raspberry Pi cluster in their laboratory.

Contents

Abstract	ii
Acknowledgements	iii
1 Background	1
1.1 Introduction	1
1.2 Goals and Objectives	1
1.3 Raspberry Pi	1
1.3.1 ARM Architecture	2
1.3.2 Raspberry Pi 3	2
1.3.3 Raspberry Pi Zero W	2
1.4 Network Architecture	2
1.5 Message Passing Interface (MPI)	3
1.5.1 mpi4py	3
1.6 Docker	3
2 Programs and Results	6
2.1 Calculating Pi	6
2.1.1 Collective Communication Operations	6
2.1.2 Dynamic Process Management	6
2.1.3 Remote Memory Access	7
2.2 Results	8
3 Conclusion	10
3.1 Conclusion	10
A Code Examples	11
A.1 MPI Operations	11
A.1.1 Broadcast Example	11
A.1.2 Scatter Example	11
A.1.3 Gather Example	12
A.1.4 Reduction Example	12

List of Abbreviations

ARM	Advanced RISC Machine
ALU	Arithmetic Logic Unit
CCO	Collective Communication Operation
DPM	Dynamic Process Management
MPI	Message Passing Interface
RMA	Remote Memory Access
RISC	Reduced Instruction Set Computer
RPi	Raspberry Pi
SoC	System on a Chip

1 Background

1.1 Introduction

The idea for this independent study came from multiple conversations by the authors regarding the lack of a parallel or cluster computing course at the university. Subsequently, we have realized the importance of large scale processing in the age of big data. Understanding and analyzing big sets of data is a necessity in an age where almost all electronic devices (large and small) are connected to a network and feeding in trillions of data points per second.

We decided to utilize a set of Raspberry Pi computers as a small-scale example of a cluster computing platform. Initially, we thought that utilizing commodity hardware such as servers for our cluster would be the way to go. Unfortunately, we do not have access to such hardware, but after further research, we concluded that the Raspberry Pi would be sufficient to demonstrate concepts in parallel and cluster computing.

1.2 Goals and Objectives

The ultimate goal of the course is to gain experience in a highly-relevant and in-demand software. Utilizing Python for the MPI, we will be able to gain a better understanding of the internal logic that goes into programming parallel operations. Although the project is limited to the Raspberry Pi, the mpi4py interface can be scaled to supercomputers and larger data warehouses using the same programming paradigm. More specifically, utilizing the Docker engine will give us a better understanding of the interaction between hardware and software in terms of virtualization and cluster computing. For instance, Docker can be utilized to organize and execute instances of Hadoop which will force us to consider the trade-offs when allocating resources to solve big-data problems. Throughout the process of implementation, we aim to keep clear and concise documentation that can make the results of our project reproducible. In addition, we would like to analyze the efficiency of Docker as a tool that can be used to increase efficiency in data-processing and web-application management for businesses and organizations.

1.3 Raspberry Pi

The Raspberry Pi is a low-cost, single-board computer that is capable of running light-weight GNU/Linux operating systems such as Raspbian. The Raspberry Pi utilizes the system on a chip (SoC) architecture in order to integrate components of the computer such as the microprocessor, graphics processing unit, and WiFi device. The Raspberry Pi is fairly easy to configure as there is a large, open-source community willing to help with setup issues. Additionally, the Raspberry Pi Foundation has put in a lot of effort to provide detailed and up-to-date documentation.

1.3.1 ARM Architecture

The Raspberry Pi central processing unit utilizes the ARM Architecture developed by the ARM Holdings. The ARM architecture paradigm is extremely relevant as there have been over 100 billion devices produced (as of 2017) that utilize the ARM instruction set architecture. Most handheld devices including iPads and gaming consoles utilize ARM cores.

ARM, also known as Advanced RISC (Reduced Instruction Set Computing) Machine requires less transistors than x86 processors (usually found in personal computers) which make it an attractive candidate for embedded systems seeking to lower costs and energy consumption on devices. The ARM architecture supported only a core-bit width of 32, but the newest version of ARM, ARMv8 now supports both 32 and 64 bits as of 2011. Note, the Raspberry Pi utilizes ARMv7. ARMv7 adheres to the load/store architecture which separates instructions into memory access and ALU (Arithmetic Logic Unit) operations. Memory access is simply the process of transferring data from the memory to registers. ALU operations consist of the actual operations on the loaded memory.

1.3.2 Raspberry Pi 3

The Raspberry Pi 3 Model B consists of a quad-core 64-bit CPU with one gigabyte of RAM in the system on a chip configuration. Additionally, the RPi 3 is equipped with wireless LAN and Bluetooth capabilities. It is important to note that while the WiFi interface is speedy enough for basic internet usage, it does cause considerable bottlenecks when utilizing a parallel processing interface. In fact, the 802.11n WiFi speeds were clocked around 45 Mbits/second, while the USB Gigabit LAN clocked around 320 Mbits/second. Configuring USB Gigabit LAN with a Raspberry Pi cluster does take a considerable amount of time to configure as MAC addresses and USB devices have to be specifically assigned in order to ensure a functional RPi cluster.

1.3.3 Raspberry Pi Zero W

The Raspberry Pi Zero W is a lighter-weight WiFi-enabled version of the RPi 3. It has a 1GHz, single-core CPU with 512 MB of RAM. We utilized four RPi Zero W computers as our slave nodes. These nodes were utilized in order to carry out parallel processing while the RPi 3 was utilized as the master node which sent out jobs using the message passing interface (MPI).

1.4 Network Architecture

In the early stages of the experiment the cluster was independently connected to the university's Wi-Fi network eduroam. Due to wireless standards and data transmission speeds on the wireless network were hard capped at 54 megabits per second. This limitation along with interference created a less than optimal environment for our experiment. Our solution for this was to implement a wired connection between the nodes. The Raspberry Pi Zero doesn't have a standard Ethernet port like the standard Raspberry Pi 2/3. With this limitation in mind, the only remaining option was to implement IP over USB. The Linux kernel within Raspbian has built in USB gadget support. This support lets allows users to turn the data USB port on a Pi Zero into a variety of different USB devices, and for this experiment a virtual network interface. The four slave nodes were connected via USB data cables to a

USB hub. The USB hub is then connected to the Master RPI 3. The final result is that our cluster was able to transmit data up to 480 megabits per second compared to 54 megabits per second wirelessly, almost 9 times faster. This extremely low latency connection completely eliminated the network bottleneck that was encountered in the beginning. Utilizing USB hubs, this could expand to 127 nodes assuming each node is independently powered not through the USB hub.

1.5 Message Passing Interface (MPI)

MPI is a standardized and portable parallel computing library usually used on supercomputers. The standards for MPI were first developed as parallel programs were being written in C, C++, and FORTRAN. Today, many other languages such as Python, R, and Java use wrapper classes in order to implement message passing interfaces written in C++ and FORTRAN.

1.5.1 mpi4py

mpi4py is a message passing interface library for the Python programming language. We decided to use the Python programming language as it offers the implementation of high-level data structures with a dynamic typing and binding paradigm. Additionally, mpi4py has become one of the most utilized parallel computing libraries, thus there exists much online documentation and assistance. For instance, the development of Python libraries such as NumPy make it easy to implement sometimes complex data structures such as arrays and dataframes in a user-friendly manner. Figure 1.1 shows some of the most basic parallel operations that mpi4py supports. The broadcast operation (A.1.1) sends out the same operation to all slave nodes. The scatter operation (A.1.2) sends out different operations to the worker nodes. The gather operation (A.1.3) takes the output from the operation and sends it back to the master node. The reduction operation (A.1.4) takes the output from the slave nodes and performs some type of operation on that data. Please see code examples in Appendix A for a better understanding of MPI operations.

1.6 Docker

Docker is an abstract, containerization system utilized for the automation and deployment of GNU/Linux operating systems. Docker allows many single-images (known as containers) to run within one instance of Linux. This reduces the overhead of having to start and maintain many virtual machines. In the context of Docker, an image is a lightweight, executable package that has everything that is needed to run a piece of software, including code, libraries, environment variables, and configuration files. A container is the actual instance of an image when the image is executed on the operating system. The container runs independent of the host environment and can only access host files and networks if specified to do so. It is also important to note that containers run on the host machine's kernel because the host kernel will produce better performance metrics than a virtual machine would. Figure 1.3 illustrates the aforementioned concepts.

Although Docker saves resources on installing and maintaining virtual machines, Docker has an even more attractive advantage for firms involved in database management, big data processing, and the organization of commodity hardware. Docker allows users to pull images of applications and operating systems directly from a

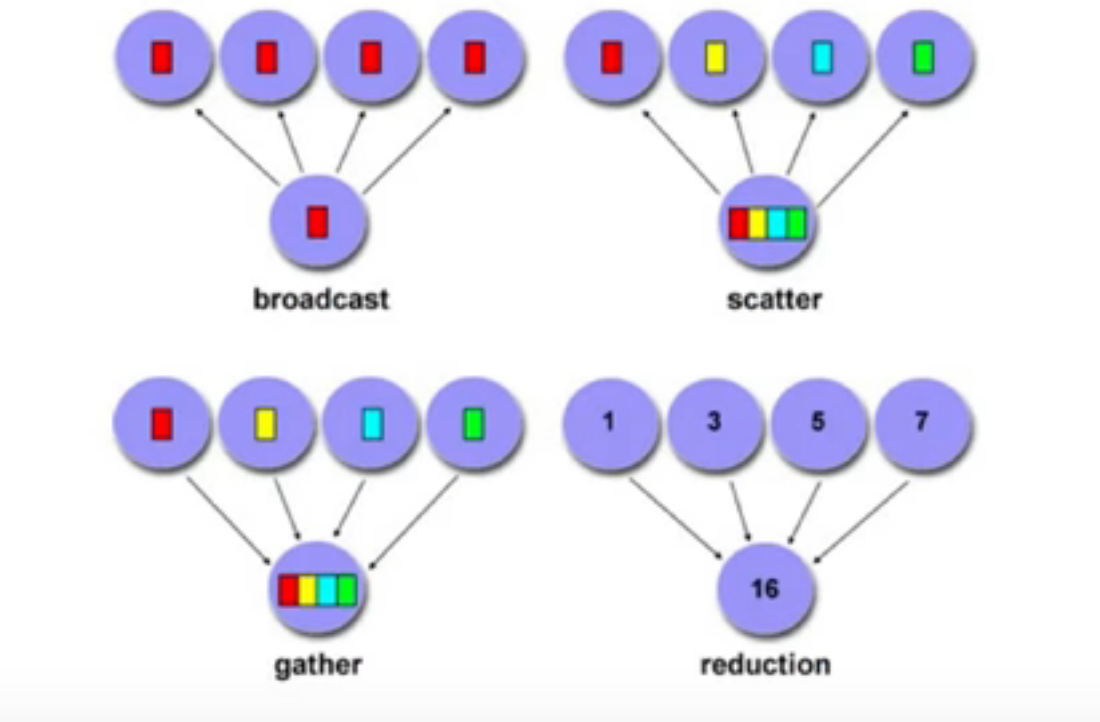


FIGURE 1.1: Four main supported MPI operations in mpi4py.

Object/Method	Functionality	Python Syntax
Communicator	Contains number of processes and rank information	<code>comm = MPI.COMM_WORLD</code>
rank	Rank of this process	<code>comm.Get_rank()</code>
broadcast	Broadcast data	<code>comm.bcast(data, root=0)</code>
gather	Gathers data from nodes	<code>comm.gather(data, root=0)</code>
size	Obtains number of available processors	<code>Size = comm.Get_size()</code>
scatter	Scatters information to nodes.	<code>comm.scatter(data, root=0)</code>

FIGURE 1.2: Description of MPI operations with respective Python syntax

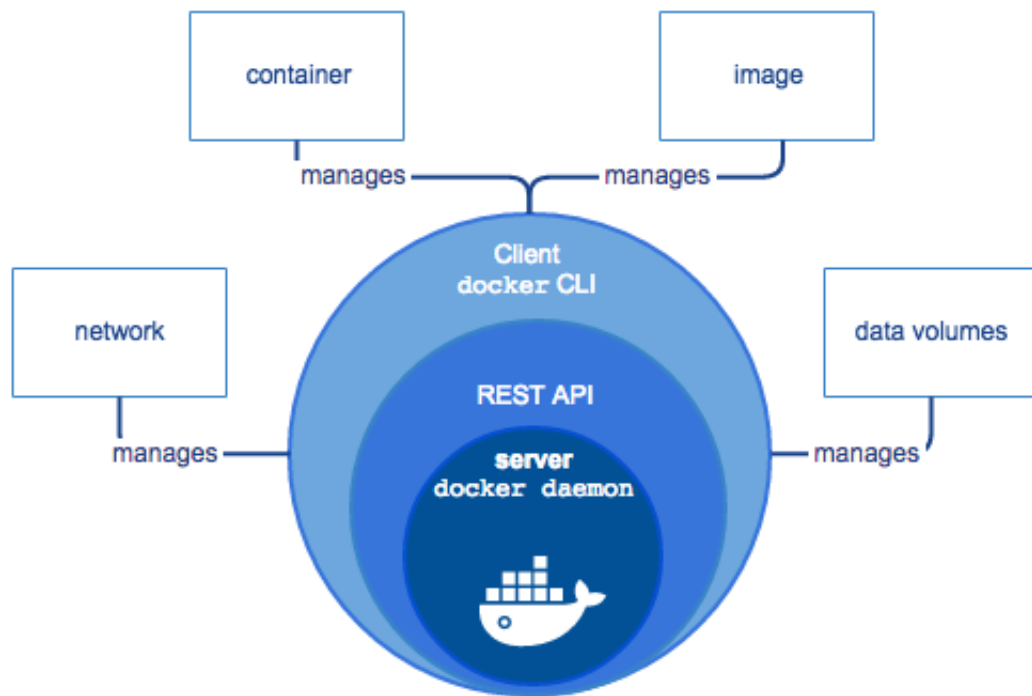


FIGURE 1.3: Illustration of Docker architecture.

centralized repository instead of having to reinstall software or applications on individual computers. Not only does it save times, it allows companies to scale up computing power and utilize computing resources to a fuller extent without having to install and configure environment variables, configuration files, or libraries each time. Docker also allows system administrators to automate processes regarding when to start and stop Linux containers and how to access networks. Additionally, Docker containers can communicate with each other thus allowing for a scalable cluster-implementation of containers.

2 Programs and Results

2.1 Calculating Pi

To calculate the value of π we decided to utilize the Monte Carlo method. The algorithm is as follows given that a circle and a square have a ratio of areas that is equal to $\pi / 4$:

1. Draw a circle inscribed within a square. (Figure 2.1)
2. Randomly pick points within the area of the square.
3. Find the ratio of the number of points falling within the circle to the total of number of points.
4. The ratio of those points should be $\pi / 4$.
5. To obtain the value of π , multiply by 4.

The more random points (or "darts") thrown into the given area, the more accurate the value of pi will be. This is because as the numerator (number of points within the circle) and denominator (number of total points) gets larger, there will numbers with more decimal places. The following subsections will discuss the implementation of the above algorithm in three different MPI programming paradigms.

2.1.1 Collective Communication Operations

One program that we used to calculate π utilized the collective communication operations (CCO) method. The collective operation heavily relies on the communicator object which holds necessary information for assigning nodes their respective rank or job number. All processes (or in this case nodes) call the communicator object with the same arguments which ensures synchronization. Note, this is the most primitive form of parallel processing for our purposes as the problem is divided up and the slave nodes perform the same operations. Subsequently, this is not the most efficient way to program a parallel job as not all job management should be hinged on the actual operation that needs to be completed.

2.1.2 Dynamic Process Management

With the advent of the MPI-2 standard, the implementation of dynamic processes have become much more common place with dynamic process management (DPM). DPM allows parent processes to create new child processes during runtime using spawn functionality. In addition, it can connect previously existing processes with each other with intercommunicator objects. The spawn function can create communicator objects with the original arguments or it can take new arguments.

Dynamic Process Management (DPM) is used in the master (or parent) process to spawn new worker (or child) processes running a Python interpreter. The master process uses a separate thread to communicate back and forth with the workers. The worker processes serve the execution of tasks in the main (and only) thread until they are signaled for completion. DPM is highly compatible with our cluster

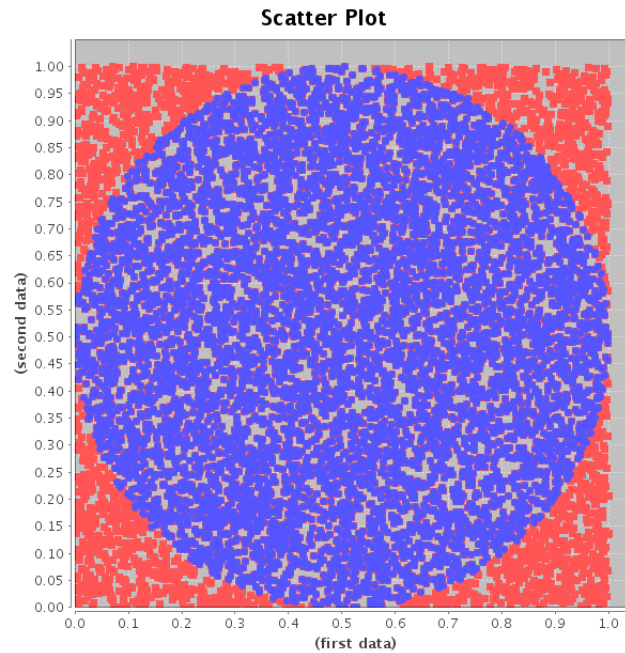


FIGURE 2.1: Depiction of Monte Carlo method to calculate π .

configuration, because the process model provides a mechanism to create new processes and establish communication between them and the existing MPI application. It also provides mechanisms to establish communication between two existing MPI applications, even when one did not start the other.

Although DPM offers much more flexibility and efficiency by handling bottlenecks, it requires much more code in order to implement. The code has to determine what jobs will be done by respective child process and how to handle multi-stage spawning. Multi-stage spawning occurs when two child process have to communicate with each other. The programmer has to determine if the child process will communicate with each other directly or if the parent node will act as the intercommunicator. See Figure 2.2 for a visual representation of the parent job in blue with respective child jobs in green.

2.1.3 Remote Memory Access

Remote memory access (RMA) is another feature implemented in MPI-2. RMA is highly convenient for parallel programming as it does not require the programmer to have send and receive calls for both sides of communication. For instance, all previous programs required an explicit call to send the data and an explicit call to receive the data. The RMA method allows one method call to send the data and at the same time it will automatically initiate a call to receive the data when necessary. This allows processes to act independently of each other when sending and receiving data. Traditionally, when a process sends a job or piece of data to another process or job, it has to wait until that information has been received. With RMA, once the information has been sent, the process is free to move onto its next task. For RMA to work, the process must specify a contiguous region of memory called a window that is open to all other processes. The other processes that are trying to access this space of memory must be aware of this window.

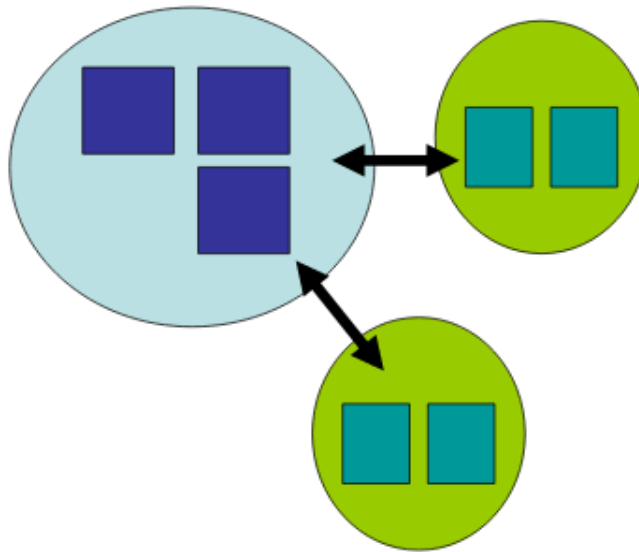


FIGURE 2.2: An instance of multi-stage spawning.

Remote Memory Access (RMA) supplements the traditional two-sided, send/receive based MPI communication model with a one-sided, put/get based interface. One-sided communication that can take advantage of the capabilities of highly specialized network hardware. Additionally, this extension lowers latency and software overhead in applications written using a shared-memory-like paradigm, which is especially useful in massively parallel computer clusters (in our case our RPi cluster). The MPI specification revolves around the use of objects called windows; they intuitively specify regions of a process's memory that have been made available for remote read and write operations. The published memory blocks can be accessed through three functions for put (remote send), get (remote write), and accumulate (remote update or reduction) data items. It is useful in calculating π because in a reduce operation when all processes are sending a value to the original process via RMA it obtains the value of π much faster.

2.2 Results

During the testing process, we decided to specify four nodes for the parallel processing. The RPi 3 was used as the master node where we executed programs and the RPi Zeros were the slave nodes which executed the specified job. It is important to note that all nodes of the Raspberry Pi require access to the program that is being executed. To solve this problem, we mounted a SSH filesystem (SSHFS) which means that when a file is updated in a directory, that change is made across all systems that have the directory mounted.

Figure 2.3 depicts the time it took to calculate the value of π using the three different parallel processing methods.

From the results, we can see that the dynamic process management (DPM) method had the highest efficiency which may be attributed to its extremely efficient scheduling system. The collective communications and remote memory methods had very

Walltime in seconds for three parallel processing methods			
	CCO	DPM	RMA
Trial 1	3.031211138	2.3352607489	3.5124759674
Trial 2	3.035000086	2.4041705132	3.5124800205
Trial 3	3.037312984	2.4411482215	3.5124299526

FIGURE 2.3: Walltime of three different execution methods.

Error		
CCO	DPM	RMA
0.005207865	8.33333E-06	0.005207865

FIGURE 2.4: The error between the calculated and known values of π

similar execution times. In addition, the DPM method had the lowest error when compared to the actual value of π . Even more interesting is that the CCO and RMA methods had the same error values which may be attributed to the similar algorithmic implementations of the Monte Carlo Method.

3 Conclusion

3.1 Conclusion

After testing the Monte Carlo method to calculate the value of π , the dynamic process management (DPM) method of programming calculated the value of π the fastest with the least amount of deviation from the known value of π . Although DPM was superior to the other methods, it requires more lines of code and a higher-level of programming ability.

Throughout the course of the semester, the authors have gained knowledge in parallel computing applications such as Python's `mpi4py` library and various mathematical methods utilized to improve throughput and efficiency in programming. In addition, the authors have gained experience in setting up cluster computing environments using GNU/Linux software and configuring networks in order to optimize interconnections between cluster computing nodes.

A Code Examples

A.1 MPI Operations

The color of links can be changed to your liking using:

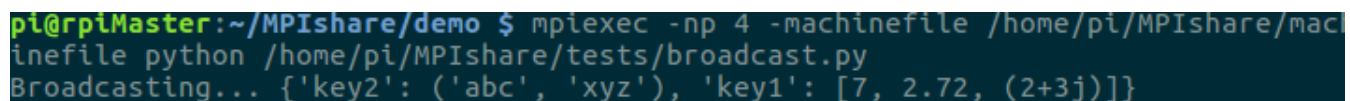
A.1.1 Broadcast Example

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'key1' : [7, 2.72, 2+3j], 'key2' : ( 'abc', 'xyz')}
    print 'Broadcasting...',data
else:
    data = None

data = comm.bcast(data, root=0)
print 'Broadcasting...',data
```

Output:



```
pi@rpiMaster:~/MPIshare/demo $ mpiexec -np 4 -machinefile /home/pi/MPIshare/machinefile python /home/pi/MPIshare/tests/broadcast.py
Broadcasting... {'key2': ('abc', 'xyz'), 'key1': [7, 2.72, (2+3j)]}
```

A.1.2 Scatter Example

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
    data = [(i+1)**2 for i in range(size)]
else:
    data = None

data = comm.scatter(data, root=0)
assert data == (rank+1)**2
```

Output:

```
pi@rpiMaster:~/MPIshare/tests $ mpiexec -np 4 -machinefile /home/pi/MPIshare/machinefile python /home/pi/MPIshare/tests/scatter.py
Scattering... [1, 4, 9, 16]
```

A.1.3 Gather Example

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

data = (rank+1)**2
data = comm.gather(data, root=0)

if rank == 0:
    for i in range(size):
        assert data[i] == (i+1)**2
        print data[i], 'from node', i
    print 'The final array:', data
else:
    assert data is None
```

Output:

```
pi@rpiMaster:~/MPIshare/demo $ mpiexec -np 4 -machinefile /home/pi/MPIshare/machinefile python /home/pi/MPIshare/tests/gather.py
1 from node 0
4 from node 1
9 from node 2
16 from node 3
The final array: [1, 4, 9, 16]
```

A.1.4 Reduction Example

```
from mpi4py import MPI
import numpy
import sys
comm = MPI.COMM_SELF.Spawn(sys.executable, args=['cpi.py'], maxprocs=5)
N = numpy.array(100, 'i')
comm.Bcast([N, MPI.INT], root=MPI.ROOT)
PI = numpy.array(0.0, 'd')
comm.Reduce(None, [PI, MPI.DOUBLE], op=MPI.SUM, root=MPI.ROOT)
print (PI)
comm.Disconnect()
```

Output:

```
pi@mpiMaster:~/MPIshare/tests $ mpiexec -np 4 -machinefile /home/pi/MPIshare/machinefile python /home/pi/MPIshare/tests/reduce.py
3.14160098692
3.14160098692
3.14160098692
3.14160098692
```