R.V. COLLEGE OF
ENGINEERING BENGALURU –
560059
(Autonomous Institution Affiliated to VTU, Belagavi)

"MPI: Message Passing Interface USING PYTHON"

Parallel Architecture & Distributed Programming (16CS71)

REPORT
Submitted by

Prerana Shenoy S P 1RV17CS114

Under the Guidance of
Prof. Sandhya S

Department of Computer Science & Engineering

ABSTRACT

Parallel computing is now as much a part of everyone's life as personal computers, smart phones, and other technologies are. The Message Passing Interface Standard (MPI) is a message passing library standard based on the consensus of the MPI Forum, which has over 40 participating organizations, including vendors, researchers, software library developers, and users. The goal of the Message Passing Interface is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. As such, MPI is the first standardized, vendor independent, message passing library. The advantages of developing message passing software using MPI closely match the design goals of portability, efficiency, and flexibility. MPI is not an IEEE or ISO standard, but has in fact, become the "industry standard" for writing message passing programs on HPC platforms.

Python is a general purpose and high level programming language. Python is used for for developing desktop GUI applications, websites and web applications. Also, Python, as a high level programming language, allows you to focus on core functionality of the application by taking care of common programming tasks. Since Python is a general-purpose language, it can do a set of complex machine learning tasks and enable you to build prototypes quickly that allow you to test your product for machine learning purposes.

MPI for Python supports convenient, *pickle*-based communication of generic Python object as well as fast, near C-speed, direct array data communication of buffer-provider objects (e.g., NumPy arrays).

RV COLLEGE OF ENGINEERING®, BENGALURU - 560059
(Autonomous Institution Affiliated to VTU, Belagavi)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

Certified that the report titled " MPI: Message Passing Interface Using Python " has been carried out by Prerana Shenoy S P (1RV17CS114) **,** bonafide student of RV College of Engineering, Bengaluru, have submitted in partial fulfillment for the **Internal Assessment of Course: Parallel Architecture & Distributed Programming (16CS71)** during the year 2020-2021. It is certified that all corrections/suggestions indicated for the internal Assessment have been incorporated in the report.

Prof. Sandhya S
Faculty Incharge,
Department of CSE,
R.V.C.E , Bengaluru - 59.
59.

Dr. Ramakanth P
Head of Department,
Department of CSE,
R.V.C.E , Bengaluru -

RV COLLEGE OF ENGINEERING®, BENGALURU - 560059
(Autonomous Institution Affiliated to VTU)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DECLARATION

I, **Prerana Shenoy S P (1RV17CS114)** ,the student of Seventh Semester B.E., Computer Science and Engineering, R.V. College of Engineering, Bengaluru hereby declare that the report titled **" MPI Using Python"** has been carried out by me and submitted in partial fulfillment for the **Internal Assessment of Course: (16CS71) - Parallel Architecture & Distributed Programming** during the year 2020-2021. We do declare that the matter embodied in this report has not been submitted to any other university or institution for the award of any other degree or diploma.

Place: Bengaluru **Date: 1 January, 2021**

Acknowledgement

I take this opportunity to express my profound gratitude and deep regards to my guide Prof.Sandhya S for her exemplary guidance, monitoring and constant encouragement throughout the course of this project. The blessing, help and guidance given by her time to time shall carry me a long way in the journey of life on which I am about to embark.

I also take this opportunity to express a deep sense of gratitude to Dr.Ramakanth Kumar P, HoD, Department of Computer science and Engineering, R V College of Engineering ,for his valuable information and guidance, which helped me in completing this task through various stages.

I am obliged to my peers, for the valuable information provided by them in their respective fields. I am grateful for their cooperation during the period of my assignment.

Lastly, I thank almighty, my parents, brother, sisters and friends for their constant encouragement without which this assignment would not be possi

Table of Contents — Page no

Chapter 1

Introduction to MPI

MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran. The MPI standard defines both the syntax as well as the semantics of a core set of library routines that are very useful in writing message-passing programs.

The MPI library contains over 125 routines, but the number of key concepts is much smaller. In fact, it is possible to write fully-functional message-passing programs by using only the six routines

- **MPI_Init** Initializes MPI.
- **MPI_Finalize** Terminates MPI.
- **MPI_Comm_size** Determines the number of processes.
- **MPI_Comm_rank** Determines the label of the calling process.
- **MPI_Send** Sends a message.
- **MPI_Recv** Receives a message.

Chapter 2

MPI Using Python

MPI for Python supports convenient, *pickle*-based communication of generic Python object as well as fast, near C-speed, direct array data communication of buffer-provider objects (e.g., NumPy arrays).

What is picke-based communication?
The **pickle** module implements an algorithm for turning an arbitrary Python object into a series of bytes. This process is also called "*serializing*" the object. The byte stream representing the object can then be transmitted or stored, and later reconstructed to create a new object with the same characteristics.

Mpi4py

- Full-featured Python bindings for MPI.
- API based on the standard MPI-2 C++ bindings.
- Almost all MPI calls are supported.
- Operations are primarily methods on communicator objects
- Popular on Linux clusters and in the SciPy community
- Optimized communication of NumPy arrays

Implementation

script.py
```
from mpi4py import MPI
comm = MPI.COMM_WORLD
print("%d of %d" % (comm.Get_rank(), comm.Get_size()))
```
```
$ mpirun -n 4 python script.py
```
MPI Init is called when mpi4py is imported
MPI Finalize is called when the script exits

Communicator

- Objects that provide the appropriate scope for all communication operations
- Intra-communicators for operations within a group of processes

- Inter-communicators for operations between two groups of processes
- MPI.COMM WORLD is most commonly used communicator

Point to Point Communication

- "**send**" and "**recv**" are the most basic communication operations. These are blocking operations.
- They're also a bit tricky since they can cause your program to hang.
- *comm.send(obj, dest, tag=0)*
- *comm.recv(source=MPI.ANY SOURCE, tag=MPI.ANY TAG, status=None)*
- "**tag**" can be used as a filter
- "**dest**" must be a rank in communicator
- "**source**" can be a rank or MPI.ANY SOURCE (wild card)
- "**status**" used to retrieve information about recv'd message

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
if rank == 0:
    msg = 'Hello, world'
    comm.send(msg, dest=1)
elif rank == 1:
    s = comm.recv()
    print "Rank %d: %s" % (rank, s)
```

Output:
Rank 1: Hello, world

Deadlock

```
s = range(1000000)
src = rank - 1 if rank != 0 else size - 1
dst = rank + 1 if rank != size - 1 else 0
comm.send(s, dest=dst) # This will probably hang
m = comm.recv(source=src)
```

Solution to Deadlock -

Odd-even processes

```
if rank % 2 == 0:
     comm.send(s, dest=dst)
     m = comm.recv(source=src)
Else:
     m = comm.recv(source=src)
     comm.send(s, dest=dst)
```

Nonblocking (isend/irecv + test/wait)

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    data = {'a': 7, 'b': 3.14}
    req = comm.isend(data, dest=1, tag=11)
    req.wait()
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    data = req.wait()
```

Collective Operations

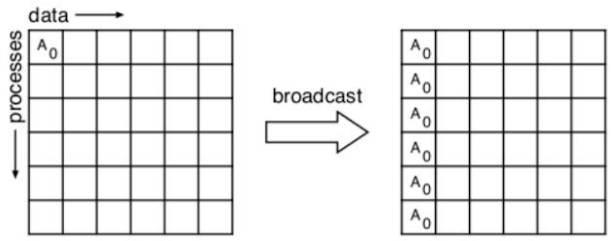
- High level operations
- Support 1-to-many, many-to-1, many-to-many operations
- Must be executed by all processes in specified communicator at the same time
- Convenient and efficient
- Tags not needed
- “root” argument used for 1-to-many and many-to-1 operations

comm.barrier()

- Synchronization operation
- Every process in communicator group must execute before any can leave

comm.bcast(obj, root=0)

Broadcasts (sends) a message from the process with rank "root" to all other processes in the group

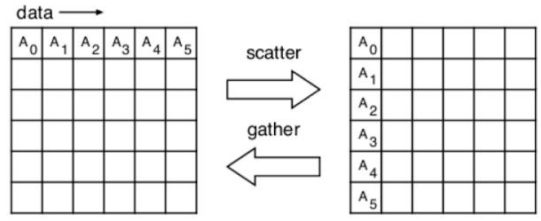


comm.scatter(sendobj, root=0)

Distributes distinct messages from a single source task to each task in the group.

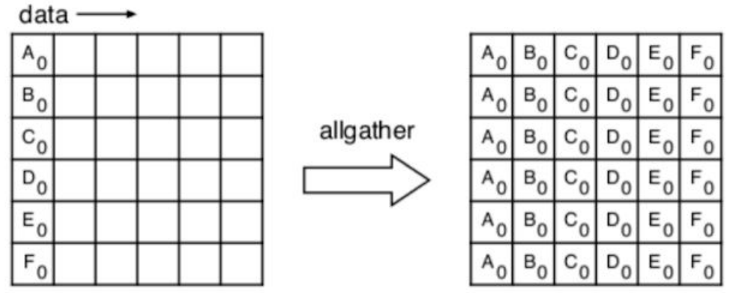
comm.gather(sendobj, root=0)

Gathers distinct messages from each task in the group to a single destination task.



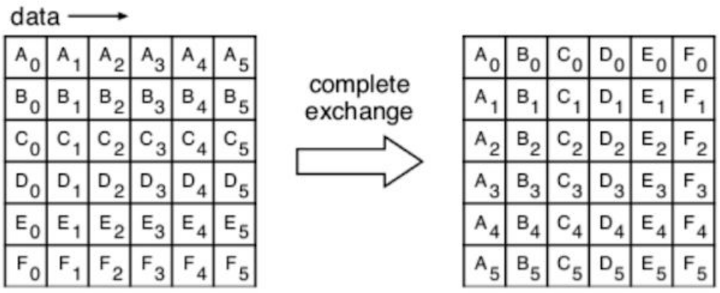
comm.allgather(sendobj)

Concatenation of data to all tasks in a group. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group.



comm.alltoall(sendobj)

Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index.



comm.reduce(sendobj, op=MPI.SUM, root=0)

Applies a reduction operation on all tasks in the group and places the result in one task.

comm.allreduce(sendobj, op=MPI.SUM)

Applies a reduction operation and places the result in all tasks in the group.

Op = MPI.MAX, MPI.MIN, MPI.SUM, MPI.PROD, MPI.LAND, MPI.LOR, MPI.BAND, MPI.BOR, MPI.MAXLOC, MPI.MINLOC

Sending Pickleable Python Objects

```
import numpy as np
from mpi4py import MPI

def rbind(comm, x):
    return np.vstack(comm.allgather(x))

comm = MPI.COMM_WORLD
x = np.arange(4, dtype=np.int) * comm.Get_rank()
a = rbind(comm, x)
```

Parallel map

The map function can be parallelized

```
x = range(20)
r = map(sqrt, x)
```

The trick is to split “x” into chunks, compute on your chunk, and then combine everybody’s results:

```
m = int(math.ceil(float(len(x)) / size))
x_chunk = x[rank*m:(rank+1)*m]
r_chunk = map(sqrt, x_chunk)
r = comm.allreduce(r_chunk)
```

Chapter 3

Requirement Specification

3.1 Hardware Requirements

- Processor of 2.2G Hz or higher speed
- 20MB Hard Disk Space
- 1GB RAM
- Keyboard

3.2 Software Requirements

- Ubuntu LTS/Virtual box
- mpi4py library
- Python interpreter
- C++ compiler

Chapter 4

Algorithm and Code

4.1 Algorithm - Kmeans Clustering

repeat nstart times
    Randomly select K points from the data set as initial centroids
    Do
        Form K clusters by assigning each point to closer centroid
        Recompute the centroid of each cluster
    until centroids do not change
    Compute the quality of the clustering
    if this is the best set of centroids found so far
        Save this set of centroids
    end
End

4.2 Code

Sequential.py

```
import numpy as np
from scipy.cluster.vq import kmeans, whiten
import time

t1=time.time()
obs = whiten(np.genfromtxt('kmeans-master/data.csv',
dtype=float, delimiter=','))
K = 10
nstart = 10000
np.random.seed(0) # for testing purposes
centroids, distortion = kmeans(obs, K, nstart) #mean
(non-squared) Euclidean distance
print('Best distortion for %d tries: %f' % (nstart,
distortion))
t2=time.time()-t1
print("Time: %lf"%(t2))
```

Parallel.py

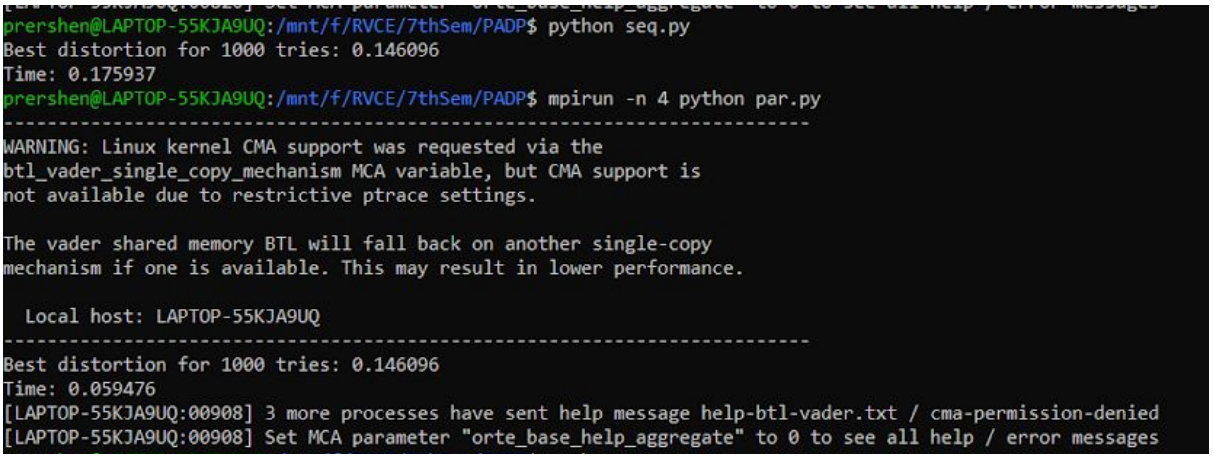
```
import numpy as np
from scipy.cluster.vq import kmeans, whiten
from operator import itemgetter
from math import ceil
from mpi4py import MPI
import time

comm = MPI.COMM_WORLD
rank = comm.Get_rank(); size = comm.Get_size()
np.random.seed(seed=rank)

t1=time.time()
obs = whiten(np.genfromtxt('kmeans-master/data.csv',
dtype=float, delimiter=','))
K = 10; nstart = 10000
n = int(ceil(float(nstart) / size))
centroids, distortion = kmeans(obs, K, n)
results = comm.gather((centroids, distortion), root=0)
if rank == 0:
    results.sort(key=itemgetter(1))
    result = results[0]
    print('Best distortion for %d tries: %f' %
(nstart, result[1]))
    t2=time.time()-t1
    print("Time: %lf"%(t2))
```

Chapter 5

Results and Snapshots



Execution Time

Iterations	Serial	Parallel
100	0.035912	0.014602
1000	0.175937	0.059476
10000	1.198641	0.416754

Chapter 6

Conclusion

The programs were executed in Ubuntu LTS 18 on a Windows system with 4 processes. We can see from the table that the parallel program takes very less time when compared to the serial program. Hence, mpi4py can be used for parallel computing in Python effectively.

Chapter 7

References

1. Introduction to Parallel Computing, Anantha Grama, Anshul Gupta,George Karypis, Vipin Kumar, 2nd Edition, 2013, Pearson Education, ISBN 13: 9788131708071 .
2. MPI: The Complete Reference, Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra, 1995, MIT Press, ISBN 0-262-69184-1.
3. https://computing.llnl.gov/tutorials/mpi/#Abstract