Towards Python MPI for Artificial Intelligence and Deep Learning Research

Gregor von Laszewski, Fidel Leal, Shannon Kerr, Erin Seliger, Cooper Young, Agness Lungu

Abstract

This is a pandoc test. For more information, please contact: laszewski@gmail.com

Contents

> **Note:** Do not edit this document, only edit the documents in the chapters directory

1   Preface

Add a preface

- Who we are and how this activity came about.

- Notation. We can copy what I have done in other books, but keep it simple e.g. we do not have to worry about epub

  - code
  - use of `verbatim` inline and in block
  - use of LaTex formulas
  - use of markdown
    * spaces before and after headline, itemize lists, e.g. online editors do that not correctly
    * hyperlinks
    * citation

1.1   Resources

- Github Actions: https://github.com/cloudmesh/cloudmesh-mpi/projects/1

  - -> Fidel
  - -> Erin
  - -> Shanon
  - -> Cooper
  - -> Agnes

- Repository: https://github.com/cloudmesh/cloudmesh-mpi

- Examples: https://github.com/cloudmesh/cloudmesh-mpi/tree/main/examples

- documents:

  - https://cloudmesh.github.io/cloudmesh-mpi/report-mpi.pdf
  - https://cloudmesh.github.io/cloudmesh-mpi/report-group.pdf

2   Overview

☐ TODO: Gregor, improves this section

After reflecting on the meeting I decided that in order to increase your python knowledge and to also lead you towards research we will be developing initially a tutorial that teaches students how to use MPI (Message Passing Interface). We do this with something that is called mpi4py. (later on we will use this to coordinate AI algorithms) We will develop multiple sections to this tutorial and each of you will each week work on a chapter so it can be redone by others. You can also work together and share each others ideas and thoughts openly as well as ask questions. We will do the following structured tasks (you will need to know what plagiarism is and when and how you need to cite):

3   Introduction to MPI

- What is MPI and why do you want to use it

- What are some example MPI functionalities and usage patterns (send receive, embarrassing parallel)

3.1 MPI

- ☐ TODO: Open, what is mpi
- ☐ TODO: Open, Ring
- ☐ TODO: Open kmeans
- ☐ TODO: Who?, calculation of pi
- ☐ TODO: Who?, find number count of 8 in randome numbers between 1-10

3.2 Installation

- ☐ TODO: Cooper, how to find te number of cores in linux and gitbash so we can use this t define the -n core parameter

Linux: nproc

osx: sysctl hw.physicalcpu hw.logicalcpu

- which one can we use?

windows: ??? we want command in gitbash that gives it

3.2.1 Installation of mpi4py on Windows

1. First you need to download msmpi from

   - https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi#ms-mpi-downloads

   Go to the download link and download and install it. Select the two packages and click Next. When downloaded clock on them to complete the setup

   - ☐ TODO: Cooper, this seems incomplete is this correct. I changed it as previous install instructions were also incomplete.

   ```
   msmpisetup.exe
   msmpisdk.msi
   ```

2. Open the system control panel and click on `Advanced system settings` and then `Environment Variables`

3. Under the user variables box click on `Path`

4. Click New in order to add `C:\Program Files (x86)\Microsoft SDKs\MPI` and `C:\Program Files\Microsoft MPI\Bin` to the Path

5. Close any open bash windows and then open a new one

6. Type the command

   ```
   $ which mpiexec
   ```

to verify if it works.

7. After you verified it is available, install mpi4py with

   ```
   $ bash $ pip install mpi4py $
   ```

8. The installation can be tested with `mpiexec -n 4 python -m   mpi4py.bench helloworld` (depending on the number of cores/nodes available to you, it may be necessary to reduce the number of copies that follow the -n option):

```
(ENV3) pi@red:~ $ mpiexec -n 4 python -m mpi4py.bench helloworld
Hello, World! I am process 0 of 4 on red.
Hello, World! I am process 1 of 4 on red.
Hello, World! I am process 2 of 4 on red.
Hello, World! I am process 3 of 4 on red.
```

3.2.2   Installing mpi4py in a Raspberry Pi

1. Activate our virtual environment:

```
$ python -m venv ~/ENV3
$ source ~/ENV3/bin/activate
```

2. Install Open MPI in your pi by entering the following command:

```
sudo apt-get install openmpi-bin
```

   After installation is complete you can check if it was successful by using

```
mpicc --showme:version
```

3. Enter

```
pip install mpi4py
```

   to download and install mpi4py.

4. The installation can be tested with `mpiexec -n 4 python -m   mpi4py.bench helloworld` (depending on the number of cores/nodes available to you, it may be necessary to reduce the number of copies that follow the -n option) In a PI4, the previous test returned:

```
(ENV3) pi@red:~ $ mpiexec -n 4 python -m mpi4py.bench helloworld
Hello, World! I am process 0 of 4 on red.
Hello, World! I am process 1 of 4 on red.
Hello, World! I am process 2 of 4 on red.
Hello, World! I am process 3 of 4 on red.
```

3.2.3   Installing mpi4py in MacOS

☐ TODO: Agnes, incomplete

A similar process can be followed to install mpi4py in MacOS. In this case, we can use Homebrew to get Open MPI by entering:

`$ brew install open-mpi`

Once Open MPI is working, steps 3 and 4 from the PI4 installation can be followed in order to download and install mpi4py.

3.3   Hello World

To test if it works a build in test program is available.

To run it on on a single host with n cores (lest assume you have 2 cores), you can use:

```
mpiexec -n 4 python -m mpi4py.bench helloworld
Hello, World! I am process 0 of 5 on localhost.
Hello, World! I am process 1 of 5 on localhost.
Hello, World! I am process 2 of 5 on localhost.
Hello, World! I am process 3 of 5 on localhost.
```

Note that the messages can be in different order.

To run it on mulitple hosts with each having n cores please create a hostfile as follows:

☐ TODO: Open, how to run it on multiple hosts on the PI

3.4 Machine file, hostfile, rankfile

Run sudo apt-get install -y python-mpi4py on all nodes.

Test the installation: mpiexec -n 5 python -m mpi4py helloworld

THIS CAN BE DONE BEST WITH CLOUDMESH

FIRTS TEST BY HAND

☐ TODO: Open, VERIFY

```
mpirun.openmpi \
  -np 2 \
  -machinefile /home/pi/mpi_testing/machinefile \
  python helloworld.py
```

The machinefile contains the ipaddresses

```
pi@192. ....
yout add the ip addresses
```

☐ TODO: Open, learn about and evaluate and test if we can do

```
mpirun -r my_rankfile --report-bindings ...

Where the rankfile contains:
rank 0=compute17 slot=1:0
rank 1=compute17 slot=1:1
rank 2=compute18 slot=1:0
rank 3=compute18 slot=1:1
```

3.5 MPI Functionality examples

3.6 MPI Collective Communication functionality examples

3.6.1 Broadcast `comm.bcast()`

☐ TODO: Cooper, explenation is missing

In this example, we broadcast a two-entry Python dictionary from a root process to the rest of the processes in our communicator group.
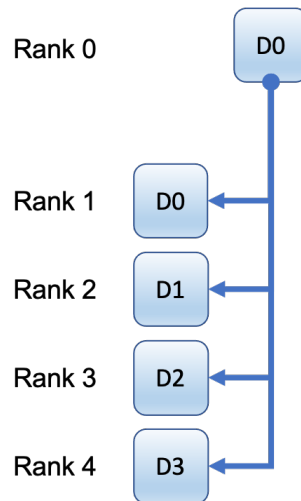
Figure 1: Example to broadcast data to different processors from the one with rank 0

```python
from mpi4py import MPI

# Set up the MPI Communicator
comm = MPI.COMM_WORLD

# Get the rank of the current process in the communicator group
rank = comm.Get_rank()

if rank == 0:   # Process with rank 0 gets the data to be broadcast
    data = {'size' : [1,3,8],
            'name' : ['disk1', 'disk2', 'disk3']}
else:  # Other processes' data is empty
    data = None

# Print data in each process
print("before broadcast, data on rank %d is "%comm.rank, data)

# Data from process with rank 0 is broadcast to other processes in our
# communicator group
data = comm.bcast(data, root=0)

# Print data in each process after broadcast
print("after broadcast, data on rank %d is "%comm.rank, data)
```

After running `mpiexec -n 4 python bcast.py` we get the following:

```
before broadcast, data on rank 0 is
   {'size': [1, 3, 8], 'name': ['disk1', 'disk2', 'disk3']}
before broadcast, data on rank 1 is  None
before broadcast, data on rank 2 is  None
before broadcast, data on rank 3 is  None
after broadcast, data on rank 0 is
   {'size': [1, 3, 8], 'name': ['disk1', 'disk2', 'disk3']}
after broadcast, data on rank 1 is
   {'size': [1, 3, 8], 'name': ['disk1', 'disk2', 'disk3']}
after broadcast, data on rank 2 is
   {'size': [1, 3, 8], 'name': ['disk1', 'disk2', 'disk3']}
after broadcast, data on rank 3 is
   {'size': [1, 3, 8], 'name': ['disk1', 'disk2', 'disk3']}
```

As we can see, the process with rank 1, received the data broadcast from rank 0.

3.6.1.1   Scatter `comm.scatter()`

☐ TODO: Cooper, explenation is missing

In this example, with scatter the members of a list among the processes in the communicator group.
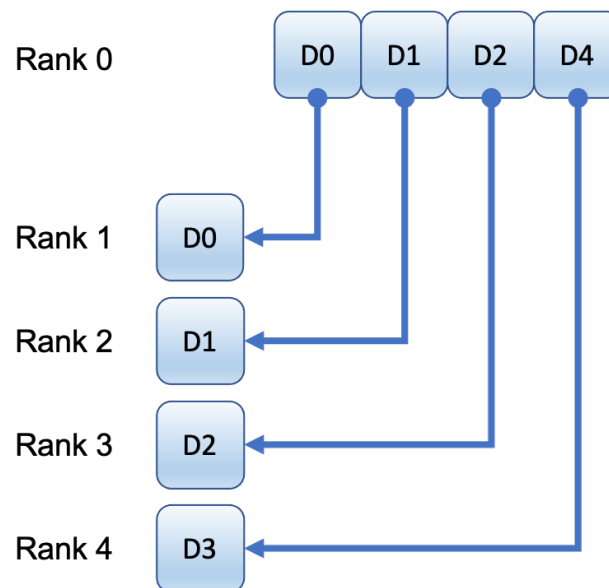
☐ TODO: All, add images

Figure 2: Example to scatter data to different processors from the one with Rank 0

```python
from mpi4py import MPI

# Communicator
comm = MPI.COMM_WORLD

# Number of processes in the communicator group
size = comm.Get_size()

# Get the rank of the current process in the communicator group
rank = comm.Get_rank()

# Process with rank 0 gets a list with the data to be scattered
if rank == 0:
    data = [(i+1)**2 for i in range(size)]
else:
    data = None

# Print data in each process
print("before scattering, data on rank %d is "%comm.rank, data)

# Scattering occurs
data = comm.scatter(data, root=0)

# Print data in each process after scattering
print("data for rank %d is "%comm.rank, data)
```

Executing `mpiexec -n 4 python scatter.py` yields:

```
before scattering, data on rank 2 is  None
before scattering, data on rank 3 is  None
before scattering, data on rank 0 is  [1, 4, 9, 16]
before scattering, data on rank 1 is  None
data for rank 2 is  9
data for rank 1 is  4
data for rank 3 is  16
data for rank 0 is  1
```

The members of the list from process 0 have been successfully scattered among the rest of the processes in the communicator group.

3.6.1.2   Gather `comm.gather()`

☐ TODO: Cooper, explenation is missing

In this example, data from each process in the communicator group is gathered in the process with rank 0.
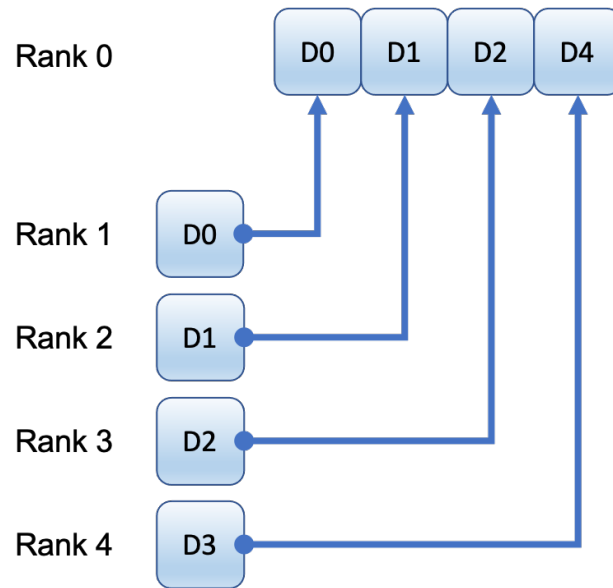
Figure 3: Example to gather data to different processors from the one with rank 0

```python
from mpi4py import MPI

# Communicator
comm = MPI.COMM_WORLD

# Number of processes in the communicator group
size = comm.Get_size()

# Get the rank of the current process in the communicator group
rank = comm.Get_rank()

# Each process gets different data, depending on its rank number
data = (rank+1)**2

# Print data in each process
print("before gathering, data on rank %d is "%comm.rank, data)

# Gathering occurs
data = comm.gather(data, root=0)

# Process 0 prints out the gathered data, rest of the processes
# print their data as well
if rank == 0:
    print("after gathering, process 0's data is ", data)
else:
    print("after gathering, data in rank %d is "%comm.rank, data)
```

Executing `mpiexec -n 4 python gather.py` yields:

```
before gathering, data on rank 2 is  9
before gathering, data on rank 3 is  16
before gathering, data on rank 0 is  1
before gathering, data on rank 1 is  4
after gathering, data in rank 2 is  None
after gathering, data in rank 1 is  None
after gathering, data in rank 3 is  None
after gathering, process 0's data is  [1, 4, 9, 16]
```

The data from processes with rank `1` to `size - 1` have been successfully gathered in process 0.

3.6.1.3   Broadcasting buffer-like objects `comm.Bcast()`

☐ TODO: is there any difference should it be moved to bcast section?

In this example, we broadcast a NumPy array from process 0 to the rest of the processes in the communicator group.

```python
from mpi4py import MPI
import numpy as np

# Communicator
comm = MPI.COMM_WORLD

# Get the rank of the current process in the communicator group
rank = comm.Get_rank()

# Rank 0 gets a NumPy array containing values from 0 to 9
if rank == 0:
    data = np.arange(0,10,1, dtype='i')

# Rest of the processes get an empty buffer
else:
    data = np.zeros(10, dtype='i')

# Print data in each process
print("before broadcasting, data for rank %d is: "%comm.rank, data)

# Broadcast occurs
comm.Bcast(data, root=0)

# Print data in each process after broadcast
print("after broadcasting, data for rank %d is: "%comm.rank, data)
```

Executing `mpiexec -n 4 python npbcast.py` yields:

```
before broadcasting, data for rank 1 is:  [0 0 0 0 0 0 0 0 0 0]
before broadcasting, data for rank 2 is:  [0 0 0 0 0 0 0 0 0 0]
before broadcasting, data for rank 3 is:  [0 0 0 0 0 0 0 0 0 0]
before broadcasting, data for rank 0 is:  [0 1 2 3 4 5 6 7 8 9]
after broadcasting, data for rank 0 is:   [0 1 2 3 4 5 6 7 8 9]
after broadcasting, data for rank 2 is:   [0 1 2 3 4 5 6 7 8 9]
after broadcasting, data for rank 3 is:   [0 1 2 3 4 5 6 7 8 9]
after broadcasting, data for rank 1 is:   [0 1 2 3 4 5 6 7 8 9]
```

As we can see, the values in the array at process with rank 0 have been broadcast to the rest of the processes in the communicator group.

3.6.1.4   Scattering buffer-like objects `comm.Scatter()`

☐ TODO: is there any difference should it be moved to scatter section?

In this example, we scatter a NumPy array among the processes in the communicator group.

```python
from mpi4py import MPI
import numpy as np

# Communicator
comm = MPI.COMM_WORLD

# Number of processes in the communicator group
size = comm.Get_size()

# Get the rank of the current process in the communicator group
rank = comm.Get_rank()

# Data to be sent
sendbuf = None

# Process with rank 0 populates sendbuf with a 2-D array,
# based on the number of processes in our communicator group
if rank == 0:
    sendbuf = np.zeros([size, 10], dtype='i')
    sendbuf.T[:,:] = range(size)

    # Print the content of sendbuf before scattering
    print('sendbuf in 0: ', sendbuf)

# Each process getd a buffer (initially containing just zeros)
# to store scattered data.
recvbuf = np.zeros(10, dtype='i')

# Print the content of recvbuf in each process before scattering
print('recvbuf in  %d: '%rank, recvbuf)

# Scattering occurs
comm.Scatter(sendbuf, recvbuf, root=0)

# Print the content of sendbuf in each process after scattering
print('Buffer in process %d contains: '%rank, recvbuf)
```

Executing `mpiexec -n 4 python npscatter.py` yields:

```
recvbuf in  1:  [0 0 0 0 0 0 0 0 0 0]
recvbuf in  2:  [0 0 0 0 0 0 0 0 0 0]
recvbuf in  3:  [0 0 0 0 0 0 0 0 0 0]
sendbuf in 0:  [[0 0 0 0 0 0 0 0 0 0]
                [1 1 1 1 1 1 1 1 1 1]
                [2 2 2 2 2 2 2 2 2 2]
                [3 3 3 3 3 3 3 3 3 3]]
recvbuf in  0:  [0 0 0 0 0 0 0 0 0 0]
Buffer in process 2 contains:  [2 2 2 2 2 2 2 2 2 2]
Buffer in process 0 contains:  [0 0 0 0 0 0 0 0 0 0]
Buffer in process 3 contains:  [3 3 3 3 3 3 3 3 3 3]
Buffer in process 1 contains:  [1 1 1 1 1 1 1 1 1 1]
```

As we can see, the values in the 2-D array at process with rank 0, have been scattered among all our processes

in the communicator group, based on their rank value.

3.6.1.5   Gathering buffer-like objects `comm.Gather()`

☐ TODO: is there any difference should it be moved to gather section?

In this example, we gather a NumPy array from the processes in the communicator group into a 2-D array in process with rank 0.

```python
from mpi4py import MPI
import numpy as np

# Communicator group
comm = MPI.COMM_WORLD

# Number of processes in the communicator group
size = comm.Get_size()

# Get the rank of the current process in the communicator group
rank = comm.Get_rank()

# Each process gets an array with data based on its rank.
sendbuf = np.zeros(10, dtype='i') + rank

# Print the data in sendbuf before gathering
print('Buffer in process %d before gathering: '%rank, sendbuf)

# Variable to store gathered data
recvbuf = None

# Process with rank 0 initializes recvbuf to a 2-D array conatining
# only zeros. The size of the array is determined by the number of
# processes in the communicator group
if rank == 0:
    recvbuf = np.zeros([size,10], dtype='i')

    # Print recvbuf
    print('recvbuf in process 0 before gathering: ', recvbuf)

# Gathering occurs
comm.Gather(sendbuf, recvbuf, root=0)

# Print recvbuf in process with rank 0 after gathering
if rank == 0:
        print('recvbuf in process 0 after gathering: \n', recvbuf)
```

Executing `mpiexec -n 4 python npgather.py` yields:

```
Buffer in process 2 before gathering:  [2 2 2 2 2 2 2 2 2 2]
Buffer in process 3 before gathering:  [3 3 3 3 3 3 3 3 3 3]
Buffer in process 0 before gathering:  [0 0 0 0 0 0 0 0 0 0]
Buffer in process 1 before gathering:  [1 1 1 1 1 1 1 1 1 1]
recvbuf in process 0 before gathering:
 [[0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 0 0 0 0 0]
  [0 0 0 0 0 0 0 0 0 0]]
recvbuf in process 0 after gathering:
 [[0 0 0 0 0 0 0 0 0 0]
  [1 1 1 1 1 1 1 1 1 1]
  [2 2 2 2 2 2 2 2 2 2]
  [3 3 3 3 3 3 3 3 3 3]]
```
The values contained in the buffers from the different processes in the group have been gathered in
the 2-D array in process with rank 0.

3.6.1.6   Gathering buffer-like objects in all processes `comm.Allgather()`   In this example, each
process in the communicator group computes and stores values in a NumPy array (row). For each process,
these values correspond to the multiples of the process' rank and the integers in the range of the communicator
group's size. After values have been computed in each process, the different arrays are gathered into a 2D
array (table) and distributed to ALL the members of the communicator group (as opposed to a single member,
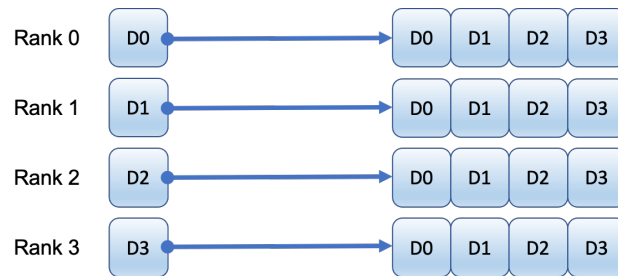which is the case when `comm.Gather()` is used instead).

Figure 4: Example to gather the data from each process into ALL of the processes in the group

```python
from mpi4py import MPI
import numpy as np



# Communicator group
comm = MPI.COMM_WORLD

# Number of processes in the communicator group
size = comm.Get_size()

# Get the rank of the current process in the communicator group
rank = comm.Get_rank()

# Initialize array and table
row   = np.zeros(size)
table = np.zeros((size, size))

# Each process computes the local values and fills its array
for i in range(size):
    j = i * rank
    row[i] = j

# Print array in each process
print("Process %d table before Allgather: "%rank, table, "\n")

# Gathering occurs
comm.Allgather([row,  MPI.INT], [table, MPI.INT])

# Print table in each process after gathering
print("Process %d table after Allgather: "%rank, table, "\n")
```

Executing `mpiexec -n 4 python allgather_buffer.py` yields:

```
Process 1 table before Allgather:  [[0. 0.]
 [0. 0.]]

Process 0 table before Allgather:  [[0. 0.]
 [0. 0.]]

Process 1 table after Allgather:  [[0. 0.]
 [0. 1.]]

Process 0 table after Allgather:  [[0. 0.]
 [0. 1.]]
```

As we see, after `comm.Allgather()` is called, every process gets a copy of the full multiplication table.

3.6.1.7   send receive

☐ TODO, Fidel send recieve

3.6.1.8   Dynamic Process Management with `spawn`

☐ TODO: Cooper, explenation is missing

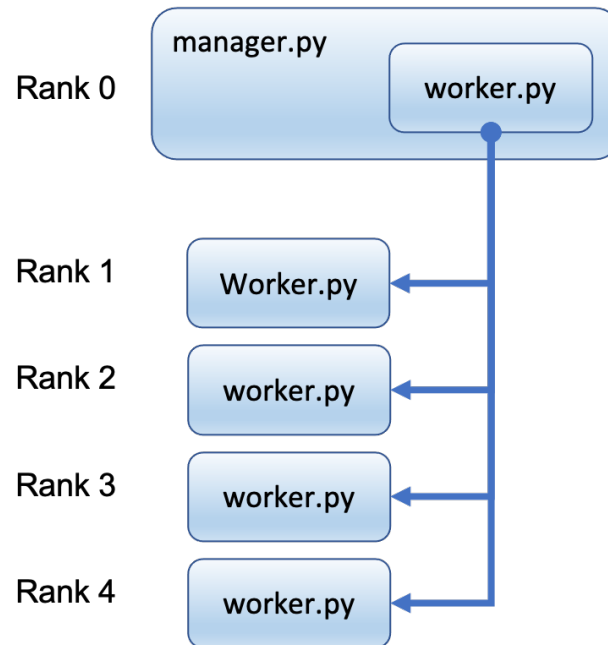In this example, we have two python programs, the first one being the manager and the second being the worker.

Figure 5: Example to spawn a program and start it on the different processors from the one with rank 0

```python
#!/usr/bin/env python
from mpi4py import MPI
import numpy
import sys
import time
print("Hello")
comm = MPI.COMM_SELF.Spawn(sys.executable,
                           args=['worker.py'],
                           maxprocs=5)
rank = comm.Get_rank()
print(f"b and rank: {rank}")

N = numpy.array(100, 'i')
comm.Bcast([N, MPI.INT], root=MPI.ROOT)
#print(f"ROOT: {MPI.ROOT}")
print('c')
PI = numpy.array(0.0, 'd')

print('d')
comm.Reduce(None, [PI, MPI.DOUBLE],
            op=MPI.SUM, root=MPI.ROOT)
print(PI)

comm.Disconnect()
```

```python
#!/usr/bin/env python
from mpi4py import MPI
import numpy
import time
import sys
comm = MPI.Comm.Get_parent()
size = comm.Get_size()
rank = comm.Get_rank()

N = numpy.array(0, dtype='i')
comm.Bcast([N, MPI.INT], root=0)
print(f"N: {N} rank: {rank}")

h = 1.0 / N
s = 0.0
for i in range(rank, N, size):
    x = h * (i + 0.5)
    s += 4.0 / (1.0 + x**2)
PI = numpy.array(s * h, dtype='d')
comm.Reduce([PI, MPI.DOUBLE], None,
            op=MPI.SUM, root=0)

#time.sleep(60)
comm.Disconnect()
#MPI.Finalize()
#sys.exit()
#MPI.Unpublish_name()
#MPI.Close_port()
```

To execute the example please go to the examples directoy and run the manager program

```
$ cd examples/spawn
$ mpiexec -n 4 python manager.py
```

This will result in:

```
N: 100 rank: 4
N: 100 rank: 1
N: 100 rank: 3
N: 100 rank: 2
Hello
b and rank: 0
c
d
3.1416009869231245
N: 100 rank: 0
N: 100 rank: 1
N: 100 rank: 4
N: 100 rank: 3
N: 100 rank: 2
Hello
b and rank: 0
c
d
3.1416009869231245
N: 100 rank: 0
```

This output depends on which child process is received first. The output can vary.

> WARNING: When running this program it may not terminate. To terminate use for now CTRL-C.

3.6.1.9   task processing (spawn, pull, . . .)

☐ TODO: Cooper, spawn, pull

3.6.1.9.1   Futures

☐ TODO: Open, futures

https://mpi4py.readthedocs.io/en/stable/mpi4py.futures.html

3.6.1.10   Examples for other collective communication methods

☐ TODO: Agnes, introduction

3.7   MPI-IO

☐ TODO: Agnes, MPI-IO

3.7.1   Collective I/O with NumPy arrays

☐ TODO: Agnes - IO and Numpy

3.7.2   Non-contiguous Collective I/O with NumPy arrays and datatypes

☐ TODO: Agnes, noncontigious IO

3.8   Monte Carlo calculation of Pi

☐ TODO: Shannon, improve

☐ TODO: Shannon WHAT IS THE PROBLEM GOAL

We start with the Mathematical formulation of the Monte Carlo calulation of pi. For each quadrant of the unit square, the area is pi. Therefore, the ratio of the area outside of the circle is pi over four. With this in mind, we can use the Monte Carlo Method for the calculation of pi.

```python
import random as r
import math as m
import time

start = time.time()
# Number of darts that land inside.
inside = 0
# Total number of darts to throw.
total = 100000

# Iterate for the number of darts.
for i in range(0, total):
  # Generate random x, y in [0, 1].
    x2 = r.random()**2
    y2 = r.random()**2
    # Increment if inside unit circle.
    if m.sqrt(x2 + y2) < 1.0:
        inside += 1

# inside / total = pi / 4
pi = (float(inside) / total) * 4
end = time.time()

# It works!
print(pi)

#Total time it takes to execute, this changes based off total
print(end - start)
```

☐ TODO: SHannon, Drawing

☐ TODO: Open, HOW AND WHY DO WE NEED MULTIPLE COMPUTERS

3.8.1   Program

☐ TODO: Shannon, PI montecarlo

☐ TODO: Shannon, Example program to run Montecarlo on multiple hosts

☐ TODO: Shannon, Benchmarking of the code

Use for benchmarking * cloudmesh.common (not thread safe, but still can be used, research how to use it in multiple threads) * other strategies to benchmark, you research (only if really needed * Use numba to speed up the code * describe how to install * showcase basic usage on our monte carlo function * display results with matplotlib

3.8.2   Counting Numbers

☐ TODO: Erin, Explain the program

```python
#TODO
#how do you generate a random number
#how do you generate a list of random numbers
#how do you find the number 8 in a list
#how do you gather the number 8
import random
from mpi4py import MPI

# Communicator
comm = MPI.COMM_WORLD

N = 20
max_number = 10
find = 8

# Number of processes in the communicator group
size = comm.Get_size()

# Get the rank of the current process in the communicator group
rank = comm.Get_rank()

# Each process gets different data, depending on its rank number
data = []
for i in range(N):
    r = random.randint(1,max_number)
    data.append(r)
count = data.count(find)

# Print data in each process
print(rank,count,data)

# Gathering occurs
count_data = comm.gather(count, root=0)

# Process 0 prints out the gathered data, rest of the processes
# print their data as well
if rank == 0:
    print(rank, count_data)
    b = sum(count_data)
    print(b)
    print("probability", (4*N)*(1/max_number))
```

More explanations

3.9   GPU Programming with MPI

Only possibly for someone with GPU (contact me if you do) Once we are finished with MPI we will use and look at python dask and other frameworks as well as rest services to interface with the mpi programs. This way we will be able to expose the cluster to anyone and they do not even know they use a cluster while exposing this as a single function . . . (edited)

The github repo is used by all of you to have write access and contribute to the research effort easily and in parallel. You will get out of this as much as you put in. Thus it is important to set several dedicated hours aside (ideally each week) and contribute your work to others.

It is difficult to asses how long the previous task takes as we just get started and we need to learn first how we work together as a team. If I were to do this alone it may take a week, but as you are less experienced it would likely take longer. However to decrease the time needed we can split up work and each of you will work on a dedicated topic (but you can still work in smaller teams if you desire). We will start assigning tasks in github once this is all set up.

3.10   Resources MPI

- https://research.computing.yale.edu/sites/default/files/files/mpi4py.pdf
- https://www.nesi.org.nz/sites/default/files/mpi-in-python.pdf
- https://www.kth.se/blogs/pdc/2019/08/parallel-programming-in-python-mpi4py-part-1/
- http://education.molssi.org/parallel-programming/03-distributed-examples-mpi4py/index.html
- http://www.ceci-hpc.be/assets/training/mpi4py.pdf
- https://www.csc.fi/documents/200270/224366/mpi4py.pdf/825c582a-9d6d-4d18-a4ad-6cb6c43fefd8

4   Deep Lerning on the PI

☐ TODO : Open, Install and use tensorflow
☐ TODO : Open, Install and use horovod

4.1   Tensorflow

- tensorflow 2.3 https://itnext.io/installing-tensorflow-2-3-0-for-raspberry-pi3-4-debian-buster-11447cb31fc4
- Tensrflow: https://magpi.raspberrypi.org/articles/tensorflow-ai-raspberry-pi This seems for older tensorflow we want 2.5.0
- Tensorflow 1.9 https://blog.tensorflow.org/2018/08/tensorflow-19-officially-supports-raspberry-pi.html
- Tensorflow 2.1.0 https://qengineering.eu/install-tensorflow-2.1.0-on-raspberry-pi-4.html
- Tensorflow https://www.instructables.com/Google-Tensorflow-on-Rapsberry-Pi/
- Horovod with mpi4py, see original horovod documentation
- Horovod goo, see if that works

4.2   Tensorflow Lite

build on ubunto and rasperry os are slightly different

- https://www.hackster.io/news/benchmarking-tensorflow-lite-on-the-new-raspberry-pi-4-model-b-3fd859d05b98

4.3   Horovod mpi4pi

- https://github.com/horovod/horovod#mpi4py

4.4   Applications

4.4.1   Object tetection

4.4.2   Time series analysis

5   Appendix

5.1   Hardware of current students

- Fidel Leal,
  - Equipment

- ∗ MacBook Pro 2015, 16GB RAM i7, SSD 512GB
- ∗ PC, 64-bit, 8GB RAM, i5, SSD <240GB, speed>
  - · Windows 10 Education
- ∗ Editor: Pycharm, vim
- Shannon Kerr,
  - – Equipment
    - ∗ Dell Inspiron, i5, 8GB, 1.6GHz 5482,
    - ∗ HDD 1TB 5.4K 6GB/s
    - ∗ Windows 64bit Home
    - ∗ Editor: Vim
- Erin Seliger
  - – Equipment
    - ∗ Windows hp 2020, 16GB RAM, i7, 64-bit operating system
    - ∗ Windows Pro 64bit
    - ∗ Editor: Vim
- Cooper Young
  - – Equipment
    - ∗ Dell Inspiron 7000, i7 2 Ghz, 16GB RAM, Intel Optane 512GB SSD
    - ∗ Windows 10 Education 64bit
    - ∗ Vim, Pycharm, Pico
- Agness Lungu
  - – Equipment
    - ∗ Lenovo V570, 16GB RAM, intel(R) core(TM) i5-2430M, 64-bit operating system,
    - ∗ Windows 10 Education
    - ∗ editor: Vim, Pycharm