Towards Python MPI for Artificial Intelligence and Deep Learning Research

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

Abstract

Today state of science and tool kits python has become the predominantly programming language. However, previously existing parallel programming paradigms such as message passing in interface (MPI) have proven to be a useful asset when it comes to enhancing complex data flows while executing them on multiple computers including supercomputers. However many students do not have the time to learn C programming to utilize such advanced cyberinfrastructure. Hence, it is advantageous to access MPI from Python. We will be showcasing to you how you can easily deploy and use MPI via a tool called mpi4pi. We will also show you how to use mpi4pi in support of AI workflows. This is done by using TensorFlow with it, as well as the execution of parallel execution of TensorFlow. For more information, please contact: laszewski@gmail.com

Contents

1 Preface

1.1 Project Link Collection

- Github Actions:
  - -> All
  - -> Fidel
  - -> Erin
  - -> Shanon
  - -> Cooper
  - -> Agnes
  - -> Gregor
- 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

To check out the repository use

```
$ git clone git@github.com:cloudmesh/cloudmesh-mpi.git
```

or

```
$ git clone https://github.com/cloudmesh/cloudmesh-mpi.git
```

1.2 Document Notation

To keep things uniform we use the following document notations.

1. Empty lines are to be placed before and after a context change such as a headline, paragraph, list, image inclusion.
2. All code is written in code blocks using the > and three back quotes. A rendered example looks as follows:

   ```
   this is an example
   ```

3. Using code inclusion in a line is not allowed. Single quote inclusion must be used for filenames, and other names as they are refereed to in code blocks.

4. Do showcase command inclusion we use a block but preceed every command with a `$` or other prefix indicating the computer on which the command is executed.

   ```
   $ ls
   ```

5. Formulas are written as LaTeX formulas
6. We use regular markdown
7. Inclusions are handled by pandoc-include
8. bibliography is managed via footnotes

2 Overview

Today state of science and tool kits python has become the predominantly programming language. However, previously existing parallel programming paradigms such as message passing in interface (MPI) have proven to be a useful asset when it comes to enhancing complex data flows while executing them on multiple computers including supercomputers. However many students do not have the time to learn C programming to utilize such advanced cyberinfrastructure. Hence, it is advantageous to access MPI from Python. We will be showcasing to you how you can easily deploy and use MPI via a tool called `mpi4pi`. We will also show you how to use `mpi4pi` in support of AI workflows. This is done by using TensorFlow with it, as well as the execution of parallel execution of TensorFlow.

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 random numbers between 1-10

3.2 Prerequisite

For the examples listed in this document, it is important to know the number of cores in your computer. This can be found out through the command line or a python program.

In Python, you can do it with

```
import multiprocessing
multiprocessing.cpu_count()
```

or as a command line

```
$ python -c "import multiprocessing;  print(multiprocessing.cpu_count())"
```

Alternatively, you can use the following

Linux:

```
$ nproc
```

macOS:

```
$ sysctl hw.physicalcpu hw.logicalcpu
```

Windows:

```
$ wmic CPU Get DeviceID,NumberOfCores,NumberOfLogicalProcessors
```

3.3 Installation

3.3.1 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 click on them to complete the setup

   ```
   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

   ```
   $ 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.
   ```

3.3.2 Installing mpi4pi on Ubuntu

The instalation of mpi4py on ubuntu is relatively easy. Please follow these steps. We recommend that you create a python `venv` so you do not by accident interfere with your system python. As usual you can activate it in your `.bashrc file while adding the source line there. Lastly, make sure you check it out and adjust the`-n‘ parameters to the number of cores of your machine.

```
bash $ sudo apt install python3.9 python3.9-dev $ python3 -m venev ~/ENV3 $ source
`/ENV3/bin/activate` (ENV3) $ sudo apt-get install -y mpich-doc mpich  (ENV3) $ pip
install mpi4py -U (ENV3) $ mpiexec -n 4 python -m mpi4py.bench helloworld >
```

3.3.3 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.
   ```

3.3.4 Installing mpi4py in MacOS

A similar process can be followed to install mpi4py in macOS. In this case, we can use Homebrew to get Open MPI floowed by installing mpi4py in your venv

```
$ brew install open-mpi
$ python3 -m venv ~/ENV3
$ source ~/ENV3/bin/activate
$ pip install mpi4py
```

3.4 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 a different order.

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

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

3.5 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
```

3.6 MPI Functionality examples

3.7 MPI Point-to-Point Communication Examples

3.7.1 Sending/Receiving `comm.send()` 'comm.receive()

The `send()` and `receive()` methods provide for functionality to transmit data between two specific processes in the communicator group.

MISSING IMAGE Sending and receiving data between two processes

Here is the definition for the `send()` method:

```
comm.send(buf, dest, tag)
```

`Buf` represents the data to be transmitted, `dest` and `tag` are integer values that specify the rank of the destination process, and a tag to identify the message being passed, respectively. `Tag` is particularly useful for cases when a process sends multiple kinds of messages to another process.

In the other end is the `send()` method, with the following definition:

```
comm.send(buf, source, tag, status)
```

In this case, `buf` can specify the location for the received data to be stored. Additionally, `source` and `tag` can specify the desired source and tag of the data to be received. They can also be set to `MPI.ANY_SOURCE` and `MPI.ANY_TAG`, or be left unspecified.

In the following example, an integer is transmitted from process 0 to process 1.

```python
#!/usr/bin/env python
from mpi4py import MPI

# Communicator
comm = MPI.COMM_WORLD

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

# Variable to receive the data
data = None

# Process with rank 0 sends data to process with rank 1
if rank == 0:
    comm.send(42, dest=1)

# Process with rank 1 receives and stores data
if rank == 1:
    data = comm.recv(source=0)

# Each process in the communicator group prints its data
print(f'After send/receive, the value in process {rank} is {data}')
```

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

```
After send/receive, the value in process 2 is None
After send/receive, the value in process 3 is None
After send/receive, the value in process 0 is None
After send/receive, the value in process 1 is 42
```

As we can appreciate, transmission only occurred between processes 0 and 1, and no other process was affected.

3.8 MPI Collective Communication Examples

3.8.1 Broadcast `comm.bcast()`

The `bcast()` method and it is buffered version `Bcast()` broadcast a message from a specified "root" process to all other processes in the communicator group.

In terms of syntax, `bcast()` takes the object to be broadcast and the parameter `root`, which establishes the rank number of the process broadcasting the data. If no root parameter is specified, `bcast` will default to broadcasting from the process with rank 0.

In this example, we broadcast a two-entry Python dictionary from a root process to the rest of the processes in the communicator group.

The following code snippet shows the creation of the dictionary in process with rank 0. Notice how the variable `data` remains empty in all the other processes.

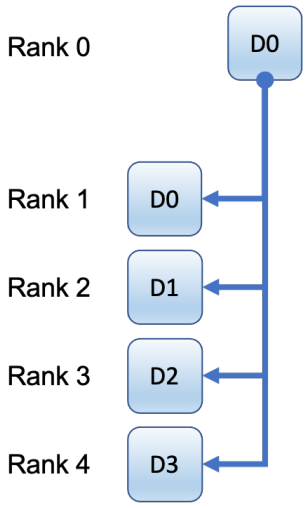

Figure 1: Broadcasting data from a root process to the rest of the processes in the communicator group

```python
#!/usr/bin/env 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(f'before broadcast, data on rank {rank} is: {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(f'after broadcast, data on rank {rank} is: {data}')
```

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

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

As we can see, all other processes received the data broadcast from the root process.

3.8.1.1 Scatter `comm.scatter()`

☐ TODO: Fidel, explenation is missing

In this example, with `scatter` the members of a list among the processes in the communicator group.

☐ TODO: All, add images

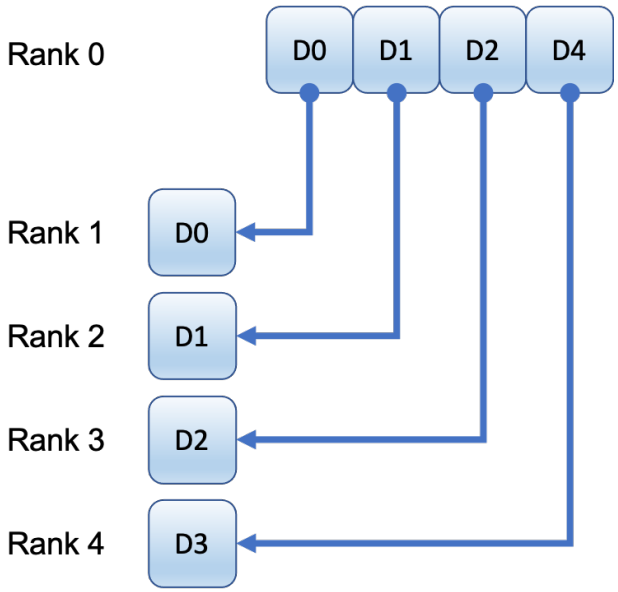

Figure 2: Example to scatter data to different processors from the one with Rank 0

```python
#!/usr/bin/env 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 before scattering
print(f'before scattering, data on rank {rank} is: {data}')

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

# Print data in each process after scattering
print(f'after scattering, data on rank {rank} is: {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.

3.8.1.2 Gather `comm.gather()`

☐ TODO: Fidel, explenation is missing

In this example, data from each process in the communicator group is gathered in the process with rank 0.

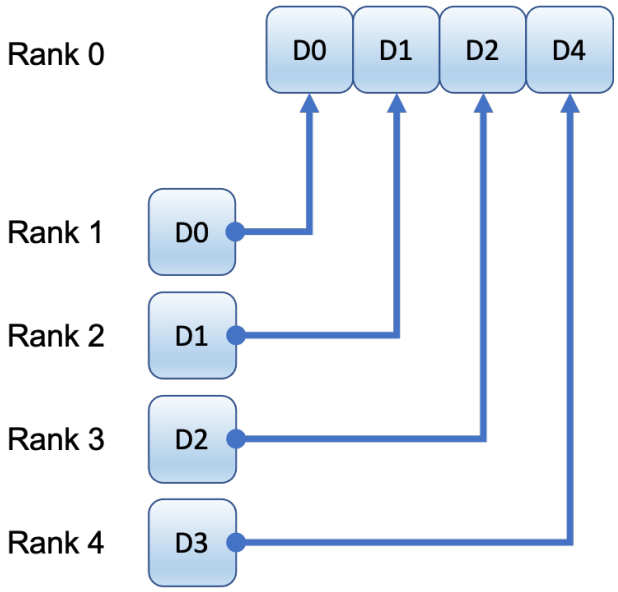

Figure 3: Example to gather data to different processors from the one with rank 0

```python
#!/usr/bin/env 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(f'before gathering, data on rank {rank} is: {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(f'after gathering, process 0\'s data is: {data}')
else:
    print(f'after gathering, data in rank {rank} is: {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.

3.8.1.3 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
#!/usr/bin/env python
import numpy as np
from mpi4py import MPI

# 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 before broadcast
print(f'before broadcasting, data for rank {rank} is: {data}')

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

# Print data in each process after broadcast
print(f'after broadcasting, data for rank {rank} is: {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 the process with rank 0 have been broadcast to the rest of the processes in the communicator group.

3.8.1.4 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
#!/usr/bin/env python
import numpy as np
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()

# 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(f'sendbuf in 0: {sendbuf}')

# Each process gets 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(f'recvbuf in {rank}: {recvbuf}')

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

# Print the content of sendbuf in each process after scattering
print(f'Buffer in process {rank} contains: {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.

3.8.1.5 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
#!/usr/bin/env python
import numpy as np
from mpi4py import MPI

# 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(f'Buffer in process {rank} before gathering: {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(f'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(f'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.

3.8.1.6 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).

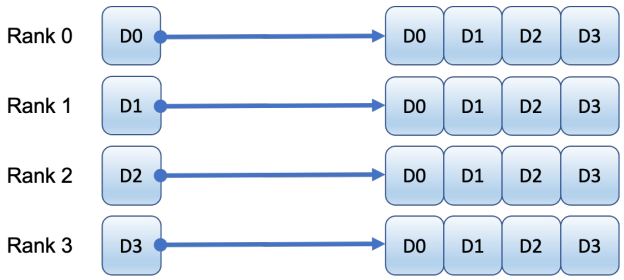

Figure 4: Example to gather the data from each process into ALL of the processes in the group

```python
#!/usr/bin/env python
import numpy as np
from mpi4py import MPI

# 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(f'Process {rank} table before Allgather: {table}\n')

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

# Print table in each process after gathering
print(f'Process {rank} table after Allgather: {table}\n')
```

Executing

```
$ mpiexec -n 4 python allgather_buffer.py`
```

results in the output

```
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.

3.8.1.7 send receive

☐ TODO, Fidel send recieve

3.8.1.8 Dynamic Process Management with `spawn`

Using >`python > MPI.Comm_Self.Spawn >`

will create a child process that can communicate with the parent. In the spawn code example, the manager broadcasts an array to the worker.

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

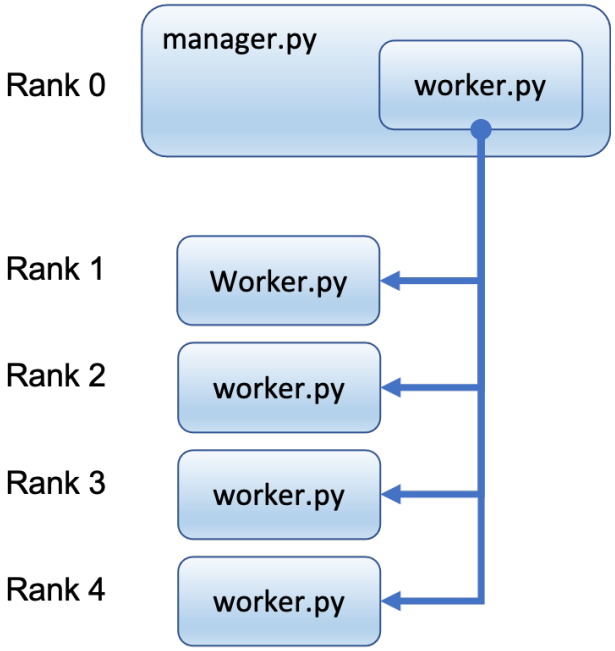

Figure 5: Example to spawn a program and start it on the different processors from the one with rank 0

```
#!/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()
```

```
#!/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 directory 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`.

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

☐ TODO: Cooper, spawn, pull

3.8.1.9.1 Futures

☐ TODO: Open, futures

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

3.8.1.10 Examples for other collective communication methods

☐ TODO: Agnes, introduction

3.9 MPI-IO

☐ TODO: Agnes, MPI-IO

3.9.1 Collective I/O with NumPy arrays

☐ TODO: Agnes - IO and Numpy

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

☐ TODO: Agnes, noncontigious IO

3.10 Monte Carlo calculation of Pi

☐ TODO: Open, improve

☐ TODO: Open WHAT IS THE PROBLEM GOAL

We start with the Mathematical formulation of the Monte Carlo calculation 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.

```
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: Open, Drawing

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

3.10.1 Program

☐ TODO: Open, PI Montecarlo

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

☐ TODO: Open, 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

3.10.2 Counting Numbers

The following program generates arrays of random numbers each 20 (n) in length with the highest number possible being 10 (max_number). It then uses a function called count() to count the number of 8's in each

data set. The number of 8's in each list is stored count_data. Count_data is then summed and printed out as the total number of 8's.

```
# Run with
#
# mpiexec -n 4 python count.py
#

#
# To change the values set them on your terminal with
#
# export N=20
# export MAX=10
# export FIND=8

# 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 os
import random

from mpi4py import MPI

# Getting the input values or set them to a default

n = os.environ.get("N") or 20
max_number = os.environ.get("MAX") or 10
find = os.environ.get("FIND") or 8

# 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 = []
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)
    total = sum(count_data)
    print(f"Total number of {find}'s:", total)
```

Executing `mpiexec -n 4 python count.py` gives us:

```
1 1 [7, 5, 2, 1, 5, 5, 5, 4, 5, 2, 6, 5, 2, 1, 8, 7, 10, 9, 5, 6]
3 3 [9, 2, 9, 8, 2, 7, 7, 2, 10, 1, 2, 5, 3, 5, 10, 8, 10, 10, 8, 10]
2 3 [1, 3, 8, 5, 7, 8, 4, 2, 8, 5, 10, 7, 10, 1, 6, 5, 9, 6, 6, 7]
0 3 [6, 9, 10, 2, 4, 8, 8, 9, 4, 1, 6, 8, 6, 9, 7, 5, 5, 6, 3, 4]
0 [3, 1, 3, 3]
Total number of 8's: 10
```

3.11 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 assess 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.

3.12 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
- 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