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
  - -> Cooper
  - -> Agnes
  - -> Gregor
  - -> Jacques
  - -> Yohn J

- 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 https://github.com/cloudmesh/cloudmesh-mpi/issues/19

Message Passing Interface (MPI) is a message-passing standard that allows for efficient communication of data between the address spaces of multiple processes. Work on the creation of the standard began in 1992 as a collective effort undertaken by several organizations, institutions, vendors and users. Since the presentation of the first draft in November of 1993, the standard has undergone several revisions and updates leading to its current version: MPI 4.0 (June 2021).

MPI is a specification, meaning that there can be multiple implementations of the standard. Examples of popular implementations are MPICH and Open MPI, although many other free or commercial implementations exist.

Additionally, MPI is a language-independent interface. Although support for C and Fortran is included as part of the standard, multiple libraries providing bindings fot other languages are available, including those for Java, Julia, R, Ruby and Python.

Thanks to its standardized nature, the portability and scalability it provides, and to the many available implementations, MPI is a popular tool in the creation of high performance and parallel computing programs.

☐ TODO: Open, Ring https://github.com/cloudmesh/cloudmesh-mpi/issues/15
☐ TODO: Open, calculation of pi https://cvw.cac.cornell.edu/python/exercise https://github.com/cloudmesh/cloudmesh-mpi/projects/1?card_filter_query=monte

Maybe to complex:

☐ TODO: Open, k-means there may be others https://medium.com/@hyeamykim/parallel-k-means-from-scratch-2b297466fdcd

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 underneath the heading `MS-MPI Downloads` and download and install it. Select the two packages and click Next. When downloaded, click on them and complete the setups.

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

2. Open the system control panel and click on `Advanced system settings` (which can be searched for with the search box in the top-right, and then click `View advanced system settings`) and then click `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. The `Browse Directory...` button makes this easier and the `Variable name` can correspond to each directory, e.g. "MPI" and "MPI Bin" respectively

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
```

ideally while bash is in venv

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

```
$ xcode-select --install
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
$ brew install wget
$ brew install open-mpi
$ python3 -m venv ~/ENV3
$ source ~/ENV3/bin/activate
$ pip install mpi4py
```

3.3.5   Installing Homebrew on MacOS

1. Download Xcode from appstore

2. Open Terminal from Applications

3. Run:

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/uninstall)"
```

4. Enter password

5. Hit return

6. Follow instructions on screen

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   Ring computation using mpi4py

As an example of the use of mpi4py, we present an instance of communication between a group of processes organized in a ring.

[Processes organized in a ring perform a sum operation] (https://github.com/cloudmesh/cloudmesh-mpi/raw/main/doc/images/ring.png)

In the example, the user provides an integer that is transmitted from process with rank 0, to process with rank 1 and so on until the data returns to process 0. Each process increments the integer by 1 before transmitting it to the next one, so the final value received by process 0 after the ring is complete is the sum of the original integer plus the number of processes in the communicator group.

```python
#!/usr/bin/env python

#
# mpieec -n 4 python ring.py --count 1000
#
#   .....
#
from mpi4py import MPI
import click
from cloudmesh.common.StopWatch import StopWatch

@click.command()
@click.option('--count', default=1, help='Number of messages send.')
@click.option('--debug', default=False, help='Set debug.')
def ring(count=1, debug=Fasle):
    # Communicator
    comm = MPI.COMM_WORLD

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

    if rank == 0:
        print(f'Communicator group with {size} processes')
        # User provides data to transmit through the ring
        data = int(input('Enter an integer to transmit: '))
        # Data is modified
        data += 1
        # Data is sent to next process in the ring

    if rank == 0:
        Stopwatch.start(f"ring {size} {count}")

    for i in range(0, count):

        if rank == 0:
            comm.send(data, dest=rank + 1)
            # print(f'Process {0} transmitted value {data} to process {rank + 1}')
            # Data is received from last process in the ring
            data = comm.recv(data, source=size - 1)
            if debug:
                print(f'Final data received in process 0 after ring is completed: {data}')

        elif rank == size - 1:

            # Data is received from previous process in the ring
            data = comm.recv(source=rank - 1)
            # Data is modified
            data += 1
            # Data is sent to process 0, closing the ring
            comm.send(data, dest=0)
            #print(f'Process {rank} transmitted value {data} to process 0')

        elif 0 < rank < size -1:
            # Data is received from previous process in the ring
            data = comm.recv(source=rank - 1)
            # Data is modified
            data += 1
            # Data is sent to next process in the ring
```

Executing the code in the example by entering `mpiexec -n 2 python ring.py` in the terminal will produce the following result:

```
Communicator group with 4 processes
Enter an integer to transmit: 6
Process 0 transmitted value 7 to process 1
Process 1 transmitted value 8 to process 2
Process 2 transmitted value 9 to process 3
Process 3 transmitted value 10 to process 0
Final data received in process 0 after ring is completed: 10
As we can see, the integer provided to process 0 (6 in this case) was successively incremented by each
process in the communicator group to return a final value of 10 at the end of the ring.
```

3.6   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
```

3.7 MPI Functionality examples

There are some differences between mpi4py and the standard MPI implementations for C/Fortran worth noting.

In mpi4py, the standard MPI_INIT() and MPI_FINALIZE() commonly used to initialize and terminate the MPI environment are automatically handled after importing the mpi4py module. Although not generally advised, mpi4py still provides MPI.Init() and MPI.Finalize() for users interested in manually controlling these operations. Additionally, the automatic initialization and termination can be deativated. For more information on this topic, please check the following links:

- MPI.Init() and MPI.Finalize()
- Deactivating automatic initialization and termination on mpi4py

Another characteristic feature of mpi4py is the availablitly of uppercase and lowercase communication methods. Lowercase methods like `comm.send()` use Python's `pickle` module to transmit objects in a serialized manner. In contrast, the uppercase versions of methods like `comm.Send()` enable transmission of data contained in a contiguous memory buffer, as featured in the MPI standard. For additional information on the topic, check (https://mpi4py.readthedocs.io/en/stable/overview.html?highlight=pickle#communicating-python-objects-and-array-data).

3.8 MPI Point-to-Point Communication Examples

3.8.1 Sending/Receiving

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

[Sending and receiving data between two processes] (https://github.com/cloudmesh/cloudmesh-mpi/raw/main/doc/images/send_receive.png)

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 `recv()` method, with the following definition:

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

In this case, `buf` can specify the location for the received data to be stored. In more recent versions of MPI, 'buf' has been deprecated. In those cases, we can simply assign `comm.recv(source, tag, status)` as the value of our buffer variable in the receiving process. 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.

The following example illustrates the use of the uppercase versions of the methods `comm.Send()` and `comm.Recv()` to perform a buffered transmission of data between processes 0 and 1 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()

# Create empty buffer to receive data
buf = np.zeros(5, dtype=int)

# Process with rank 0 sends data to process with rank 1
if rank == 0:
    data = np.arange(1, 6)
    comm.Send([data, MPI.INT], dest=1)

# Process with rank 1 receives and stores data
if rank == 1:
    comm.Recv([buf, MPI.INT], source=0)

# Each process in the communicator group prints the content of its buffer
print(f'After Send/Receive, the value in process {rank} is {buf}')
```

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

```
After Send/Receive, the value in process 3 is [0 0 0 0 0]
After Send/Receive, the value in process 2 is [0 0 0 0 0]
After Send/Receive, the value in process 0 is [0 0 0 0 0]
After Send/Receive, the value in process 1 is [1 2 3 4 5]
```

Lastly, an example of the non-blocking send and receive methods `comm.isend()` and `comm.irecv()`. Non-blocking versions of these methods allow for the processes involved in transmission/reception of data to perform other operations in overlap with the communication. In contrast, the blocking versions of these methods previously exemplified do not allow data buffers involved in transmission or reception of data to be accessed until any ongoing communication involving the particular processes has been finalized.

```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:
    send = comm.isend(42, dest=1)
    send.wait()

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

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

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

```
After isend/ireceive, the value in process 2 is None
After isend/ireceive, the value in process 3 is None
After isend/ireceive, the value in process 0 is None
After isend/ireceive, the value in process 1 is 42
```

3.9 MPI Collective Communication Examples

3.9.1 Broadcast

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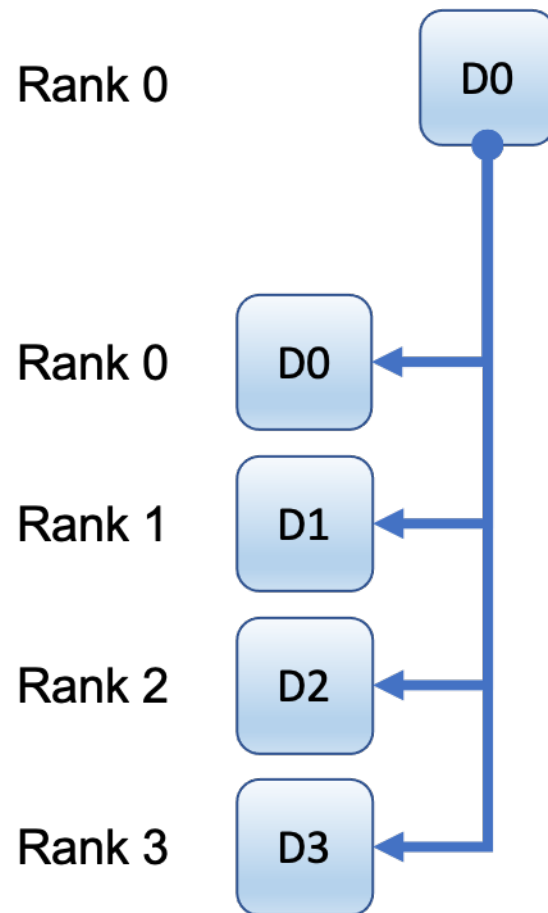
Figure 1: Broadcasting data from a root process to the rest of the processes in the communicator group

```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.

In our next example, we broadcast a NumPy array from process 0 to the rest of the processes in the communicator group using the uppercase `comm.Bcast()` method.

```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.

3.9.1.1   Scatter

☐ TODO: Fidel, explanation is missing

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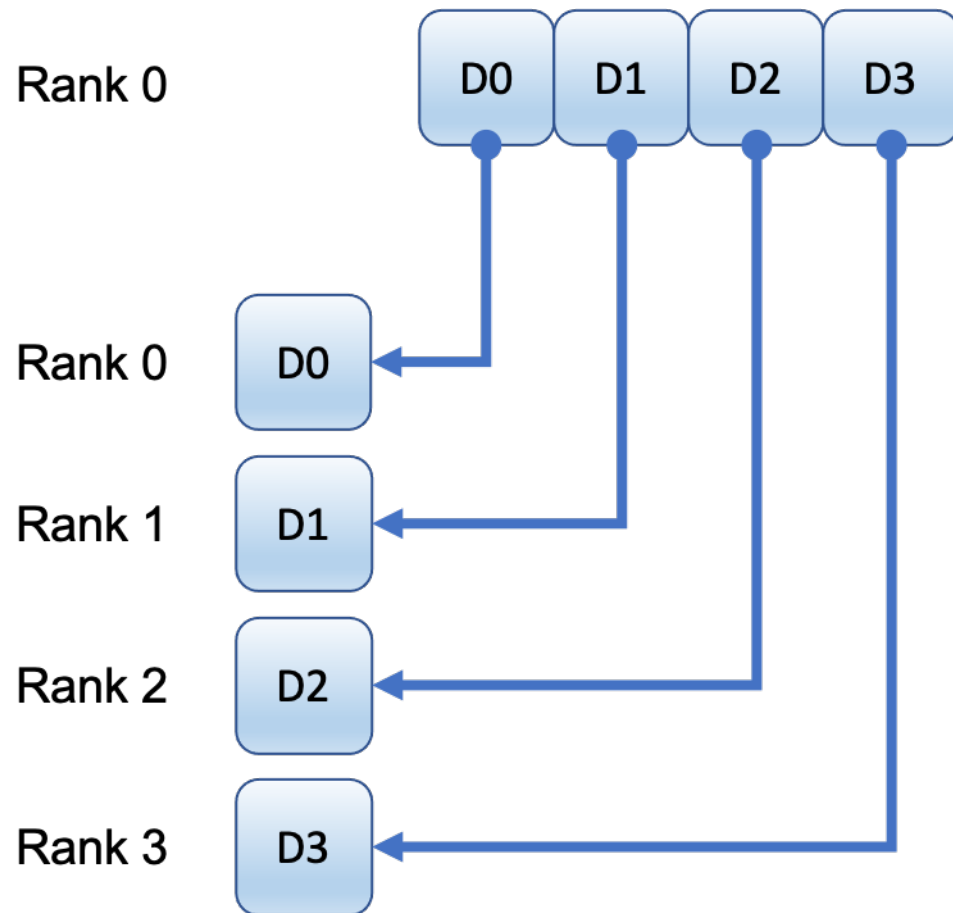
☐ TODO: All, add images

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.

In the following example, we scatter a NumPy array among the processes in the communicator group by using the uppercase version of the method `comm.Scatter()`.

```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.

3.9.1.2 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.
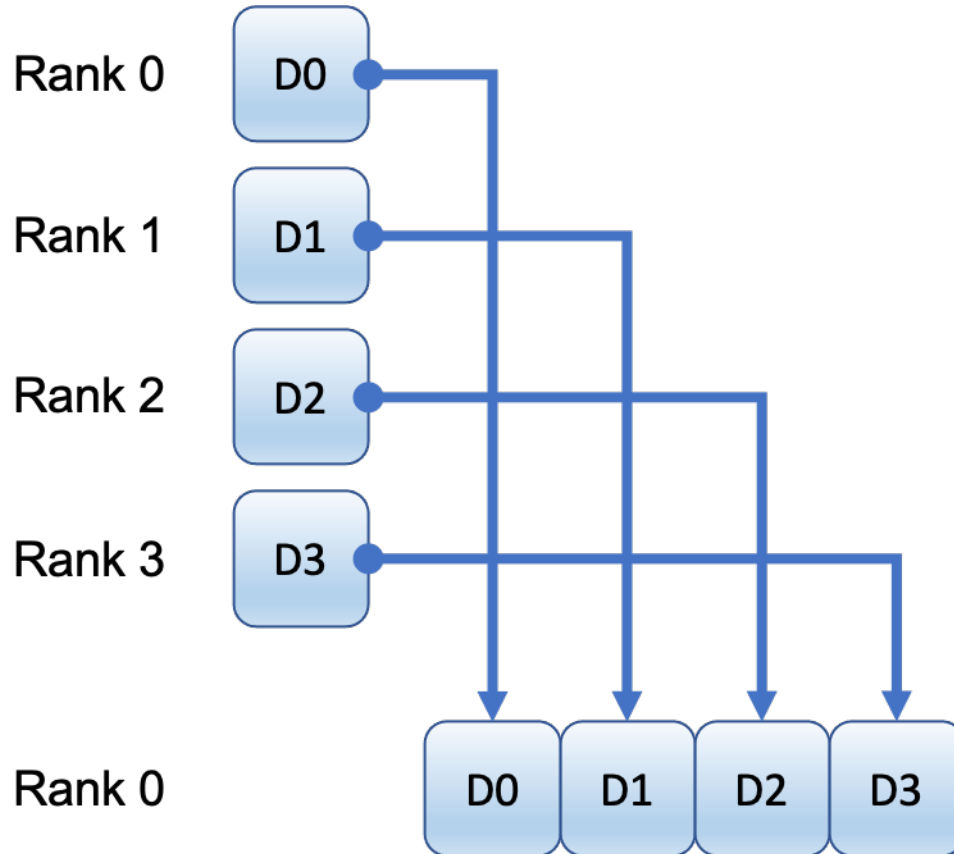

Figure 3: Example to gather data to different processors from the one with rank 0

```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.

The example showcases the use of the uppercase method `comm.Gather()`. NumPy arrays from the processes in the communicator group are gathered 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.

3.9.1.3   Gathering buffer-like objects in all processes   In this example, each process in the communi-
cator 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
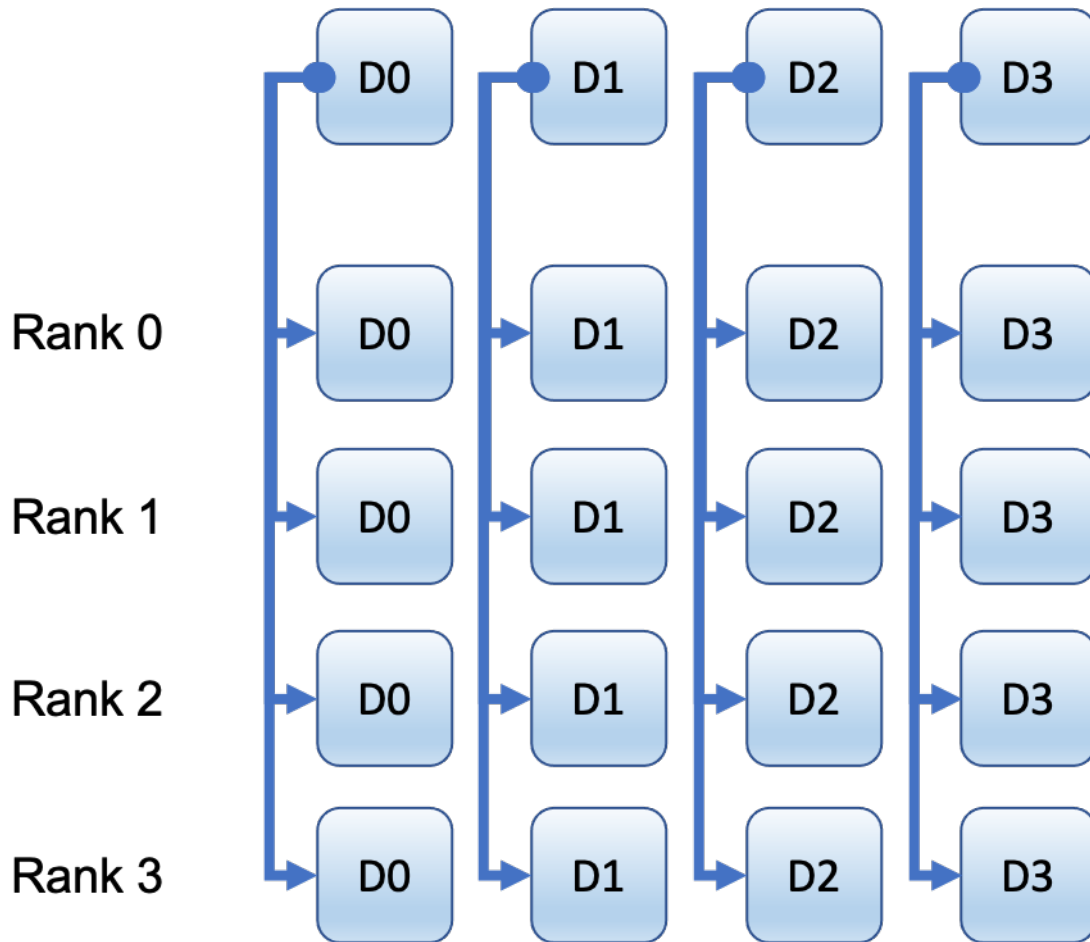case when `comm.Gather()` is used instead).

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.

3.9.1.4   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.
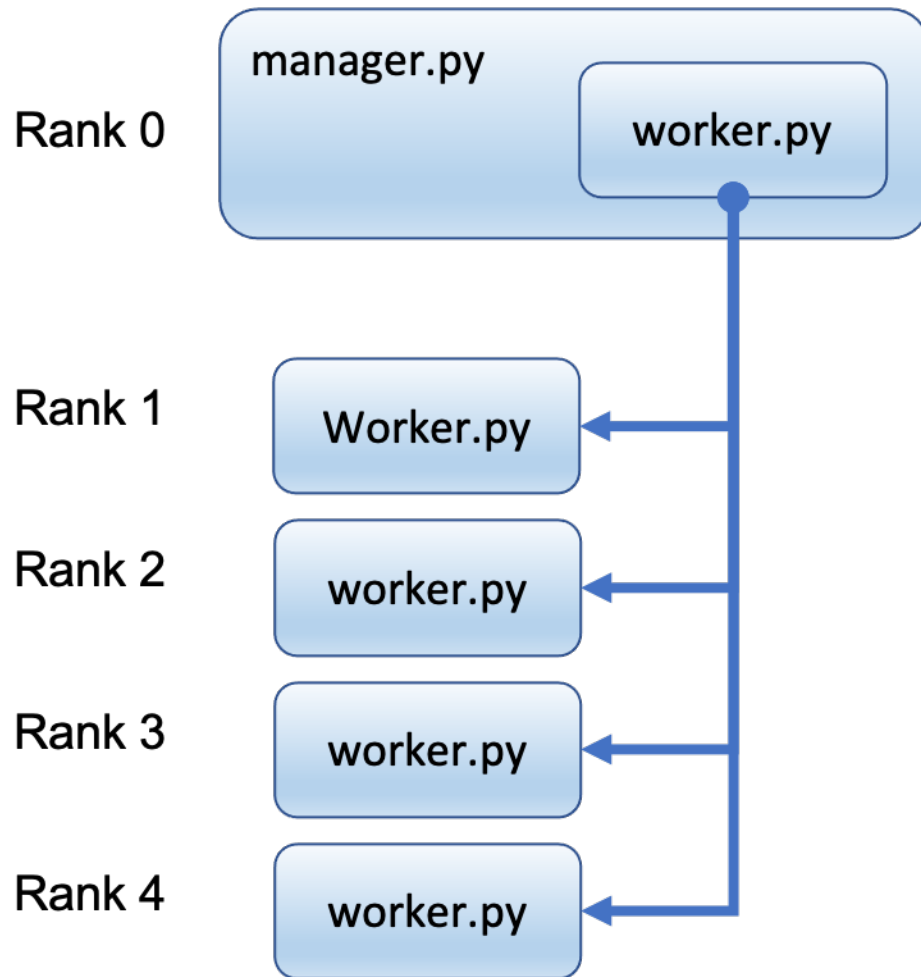

Figure 5: Example to spawn a program and start it on the different processors from the one with rank 0

```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 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`.

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

☐ TODO: Cooper, spawn, pull

3.9.1.6   Examples for other collective communication methods

☐ TODO: Agness, introduction

3.10   Futures

Futures is an mpi4py module that runs processes in parallel for intercommunication between such processes. The following Python program creates a visualization of a Julia set by utilizing this Futures modules, specifically via MPIPoolExecutor.

```python
from mpi4py.futures import MPIPoolExecutor
import matplotlib.pyplot as plt
import numpy as np
from cloudmesh.common.StopWatch import StopWatch

multiplier = int(input('Enter 1 for 640x480 pixels of Julia visualization image, 2 for 1280x960, and

StopWatch.start("Overall time")
x0, x1, w = -2.0, +2.0, 640*multiplier
y0, y1, h = -1.5, +1.5, 480*multiplier
dx = (x1 - x0) / w
dy = (y1 - y0) / h

c = complex(0, 0.65)

def julia(x, y):
    z = complex(x, y)
    n = 255
    while abs(z) < 3 and n > 1:
        z = z**2 + c
        n -= 1
    return n

def julia_line(k):
    line = bytearray(w)
    y = y1 - k * dy
    for j in range(w):
        x = x0 + j * dx
        line[j] = julia(x, y)
    return line

if __name__ == '__main__':

    with MPIPoolExecutor() as executor:
        image = executor.map(julia_line, range(h))
        image = np.array([list(l) for l in image])
        plt.imsave("julia.png", image)

StopWatch.stop("Overall time")
StopWatch.benchmark()
```

The program must be run through the terminal with the command:

```
mpiexec -n 1 python julia-futures.py
```

The number after -n can be changed to however many cores are in the computer's processor. For example, a dual-core processor can use -n 2 so that more worker processes work to execute the same program.

The program should output a png image of a Julia set upon successful execution.

3.11   MPI-IO

☐ TODO: Agness, MPI-IO

3.11.1   Collective I/O with NumPy arrays

How to use Numpy with MPI 1. Download Numpy with `pip install numpy` in a terminal 2. `import numpy as np` to use numpy in the program 3. Advantages of numpy over lists - Numpy stores memory contiguously - Uses a smaller number of bytes - Can multiply arrays by index - It's faster - Can store different data types including images - Contains random number generators

Numpy syntax 1. To define an array type: `np.nameofarray([1,2,3])` 2. To get the dimension of the array: `nameofarray.ndim` 3. To get the shape of the array (the number of rows and columns): `nameofarray.shape` 4. To get the type of the array: `nameofarray.dtype` 5. To get the number of bytes: `nameofarray.itemsize` 6. To get the number of elements in the array: `nameofarray.size` 7. To get the total size: `nameofarray.size * nameofarray.itemsize`

☐ TODO: Agness - IO and Numpy

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

☐ TODO: Agness, noncontigious IO

3.12   Monte Carlo Calculation of Pi

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.

Random Darts Graph
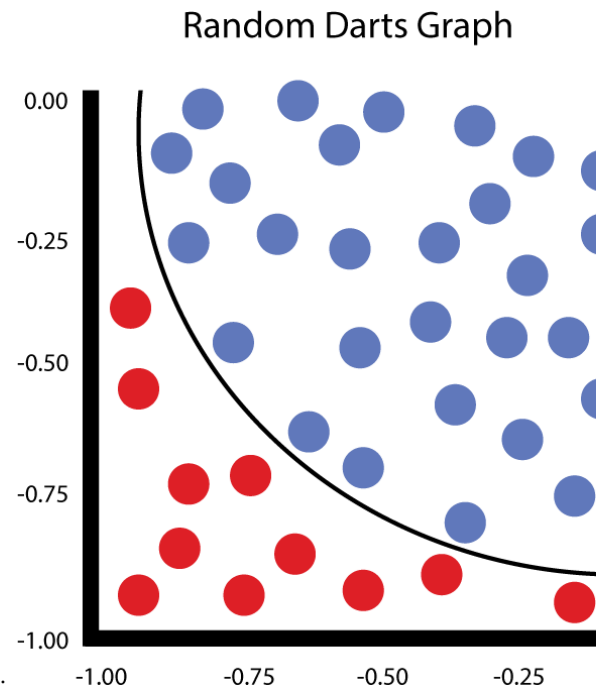
The following is a visualization of the program's methodology to calculate pi:

The following montecarlo.py program generates a very rough estimation of pi using the methodology and equation shown above.

```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)
```

However, if a more exact approximation of pi is needed, then the following program can be run instead, using multiple cores of the processor using mpiexec:

```python
# Credit to Cornell University for this script, retrieved from https://cvw.cac.cornell.edu/python/ex

from __future__ import print_function, division
"""
An estimate of the numerical value of pi via Monte Carlo integration.
Computation is distributed across processors via MPI.
"""

import numpy as np
from mpi4py import MPI
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import sys
from cloudmesh.common.StopWatch import StopWatch

StopWatch.start("Overall time")
def throw_darts(n):
    """
    returns an array of n uniformly random (x,y) pairs lying within the
    square that circumscribes the unit circle centered at the origin,
    i.e., the square with corners at (-1,-1), (-1,1), (1,1), (1,-1)
    """
    darts = 2*np.random.random((n,2)) - 1
    return darts

def in_unit_circle(p):
    """
    returns a boolean array, whose elements are True if the corresponding
    point in the array p is within the unit circle centered at the origin,
    and False otherwise -- hint: use np.linalg.norm to find the length of a vector
    """
    return np.linalg.norm(p,axis=-1)<=1.0

def estimate_pi(n, block=100000):
    """
    returns an estimate of pi by drawing n random numbers in the square
    [[-1,1], [-1,1]] and calculating what fraction land within the unit circle;
    in this version, draw random numbers in blocks of the specified size,
    and keep a running total of the number of points within the unit circle;
    by throwing darts in blocks, we are spared from having to allocate
    very large arrays (and perhaps running out of memory), but still can get
    good performance by processing large arrays of random numbers
    """
    total_number = 0
    i = 0
    while i < n:
        if n-i < block:
            block = n-i
        darts = throw_darts(block)
        number_in_circle = np.sum(in_unit_circle(darts))
        total_number += number_in_circle
        i += block
    return (4.*total_number)/n

def estimate_pi_in_parallel(comm, N):
    """
    on each of the available processes,
    calculate an estimate of pi by drawing N random numbers;
    the manager process will assemble all of the estimates
```

33

To run this program using git bash, change directory to the folder containing this program and issue the command: `$ mpiexec -n 2 python parallel_pi.py` The number after `-n` can be changed to however many cores one has on their processor.

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

3.12.1   Program

☐ 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

3.12.2   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.

```python
# 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)
```

35

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
```

3.13   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.

3.14   Python Ecosystem

It is possible to pass parameters from Git Bash into a Python environment using os.environ and a shell file.

```python
from cloudmesh.common.StopWatch import StopWatch
from time import sleep
import os

n=int(os.environ["N"])
StopWatch.start(f"processors {n}")
sleep(0.1*n)
print(n)
StopWatch.stop(f"processors {n}")
StopWatch.benchmark()
```

Ensure that this code is saved in a particular directory. Then create a shell file named run.sh with the following contents:

```
$ N=1; python environment-parameter.py
$ N=2; python environment-parameter.py
```

Save the following py file in the same directory as well: > `python > !include ../examples/parameters/click-parameter.p` >

You must cd (change directory) into the directory with all of these files in Git Bash. Input the following commands into Git Bash:

```
# This command creates an environment variable called N
$ export N=10
# This command prints the environment variable called N
```

```
$ echo $N
# This command launches a Python environment
$ python -i
>>> import os
>>> os.environ["N"]
>>> exit()
$ python environment-parameter.py
$ sh run.sh
$ sh run.sh | fgrep "csv,processors"
$ python click-parameter.py
# You can manually set the variable in git bash in the same line as you open the .py file
$ python click-parameter.py --n=3
```

3.15 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
- Cooper Young
  - Equipment
    * Dell Inspiron 7000, i7 2 Ghz, 16GB RAM, Intel Optane 512GB SSD
    * Windows 10 Education 64bit
    * Editor: Vim, Pycharm, Pico
- Jacques Fleischer
  - Equipment
    * Homebuilt computer, Ryzen 5 3600 3.6 GHz, NVIDIA GTX 1660, 16 GB 3200 MHz RAM, WD 1 TB M.2 SSD, 64-bit operating system
    * Windows 10 Home (21H1)
    * Editor: PyCharm, VSCode

6 Make on Windows

Makefiles provide a good feature to organize workflows while assembling programs or documents to create an integrated document. Within `makefiles` you can define targets that you can call and are then executed. Preconditions can be used to execute rules conditionally. This mechanism can easily be used to define complex workflows that require a multitude of interdependent actions to be performed. Makefiles are executed by the program `make` that is available on all platforms.

On Linux, it is likely to be pre-installed, while on macOS you can install it with Xcode. On Windows, you have to install it explicitly. We recommend that you install `gitbash` first. After you install `gitbash`, you can install `make` from an administrative `gitbash` terminal window. To start one, go to the search field next to the Windows icon on the bottom left and type in gitbash without a `RETURN`. You will then see a selection window that includes `Run as administrator. Click on it. As you run it as administrator, it will allow you to install`make'. The following instructions will provide you with a guide to install make under windows.

6.1 Installation

Please visit

- https://sourceforge.net/projects/ezwinports/files/

and download the file

- ['make-4.3-without-guile-w32-bin.zip'](#)

After the download, you have to extract and unzip the file as follows in a gitbash that you started as administrative user:
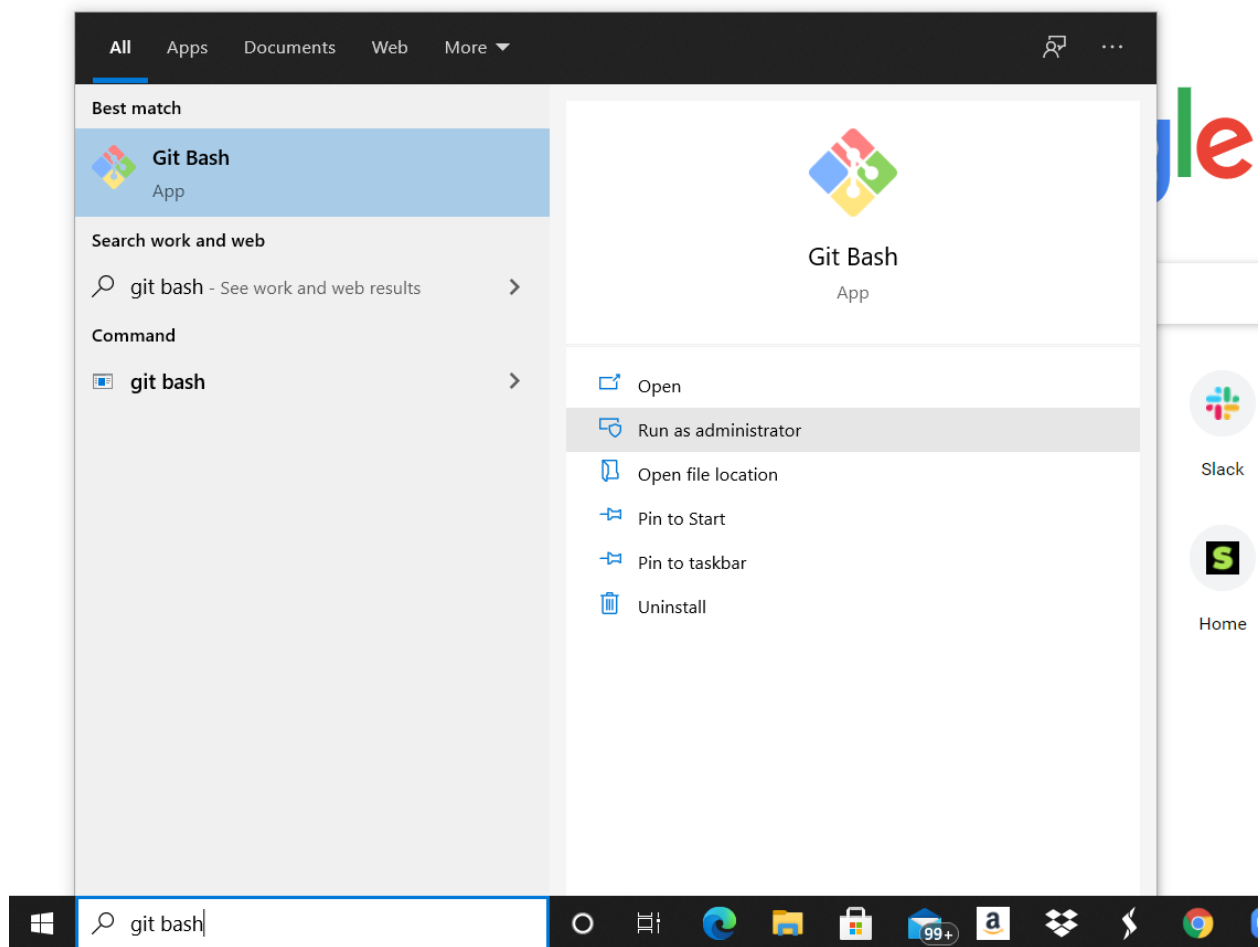
figure: screenshot of opening gitbash in admin shell

Figure 6: administrativegitbash

```
$ cp make-4.3-without-guile-w32-bin.zip /usr
$ cd /usr
$ unzip make-4.3-without-guile-w32-bin.zip
```

Now start a new terminal (a regular non-administrative one) and type the command

```
$ which make
```

It will provide you the location if the installation was successful

```
/usr/bin/make
```

to make sure it is properly installed and in the correct directory.

6.2   Installing WSL on Windows 10

WSL is a layer that allows the running of Linux executables on a Windows machine.

To install WSL2 your computer must have Hyper-V support enabled. This does not work on Windows Home, and you need to upgrade to Windows Pro, Edu, or some other Windows 10 version that supports it. Windows Edu is typically free for educational institutions. The Hyper-V must be enabled from your Bios, and you need to change your settings if it is not enabled.

More information about WSL is provided at

- https://docs.microsoft.com/en-us/windows/wsl/install-win10

To install WSL2, you can follow these steps while using Powershell as an administrative user and run

```
ps$ dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
ps$ dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
ps$ wsl --set-default-version 2
```

Next, Download the Ubuntu 20.04 LTS image from the Microsoft store

- https://www.microsoft.com/en-us/p/ubuntu/9nblggh4msv6?activetab=pivot:overviewtab

Run Ubuntu and create a username and passphrase.

Make sure not just to give an empty passphrase but chose a secure one.

Next run in Powershell

```
ps$ wsl.exe --set-version Ubuntu20.04 2
```

Now you can use the Ubuntu distro freely. The WSL2 application will be in your shortcut menu in `Start`.