MPI with Python

Gregor von Laszewski laszewski@gmail.com, Fidel Leal, Jacques Fleischer, Cooper Young

Abstract

Today python has become the predominantly programming language to coordinate scientific applications especially machine and deep learning applications. However, previously existing parallel programming paradigms such as **Message Passing 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. The framework is well known in the C-language community. However many practitioners do not have the time to learn C to utilize such advanced cyberinfrastructure. Hence, it is advantageous to access MPI from Python. We showcase how you can easily deploy and use MPI from Python via a tool called `mpi4pi`. For more information, please contact: laszewski@gmail.com

Contents

1 Preface

1.1 Document Management in GitHub

Note: The source document is managest at https://cloudmesh.github.io/cloudmesh-mpi/doc/chapters
To make changes or corrections please use a pull request

The repository, documentation, and examples are available at:

- 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 backquotes. A rendered example looks as follows:

   ```
   this is an example
   ```

3. Single quote inclusion must be used for filenames and other names as they are referred to in code blocks.
4. To showcase command inclusion, we use a block but precede every command with a `$` or other prefix indicating the computer on which the command is executed.

   ```
   $ ls
   ```

5. bibliography is managed via footnotes

2 Introduction

(Same as abstract): Today Python has become the predominantly programming language to coordinate scientific applications, especially machine and deep learning applications. However, previously existing parallel programming paradigms such as **Message Passing 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. The framework is well known in the C-language community. However, many practitioners do not have the time to learn C to utilize such advanced cyberinfrastructure. Hence, it is advantageous to access MPI from Python. We showcase how you can easily deploy and use MPI from Python via a tool called `mpi4pi`.

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

Multiple implementations following the standard exist, including the two most popular MPICH [1] and OpenMPI [2]. However, other free or commercial implementations exist [3].

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

[1] Refernce missing

[2] Reference missing

[3] Refernces missing

Thanks to its user-focused abstractions, its standardization, portability, and scalability, and availability MPI is a popular tool in the creation of high-performance and parallel computing programs.

3 Installation

Next, we discuss how to install mpi4p on various systems. We will focus on installing it on a single computer using multiple cores.

3.1 Getting the CPU Count

For the examples listed in this document, knowing the number of cores on your computer is important. 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())"
```

However, you can also use the command line tools that we have included in our documentation.

3.2 Windows 10 EDU or PRO

Note: We have not tested this on Windows home.

1. We assume you have installed GitBash on your computer. The installation is easy, but be careful to watch the various options at install time. Make sure it is added to the Path variable.

   For details see: https://git-scm.com/downloads

2. We also assume you have installed Python3.9 according to either the installation at python.org or conda. We do recommend the installation from python.org.

   https://www.python.org/downloads/

   You will need to install a python virtual env to avoid conflict by accident with your system installed version of Python.

   For details on how to do this, please visit our extensive documentation at [???]

3. Microsoft has its own implementation of MPI which we recommend at this time. 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
   ```

4. 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...`

5. Under the user variables box, click on `Path`
6. 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
7. Close any open bash windows and then open a new one
8. Type the command

   ```
   $ which mpiexec
   ```

   to verify if it works.
9. After you verified it is available, install mpi4py with

   ```
   $ pip install mpi4py
   ```

   ideally, while bash is in venv
10. Next, find out how many processes you can run on your machine and remember that number. You can do this with

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

    Alternatively, you can use a python program as discussed in the section “Getting the CPU Count”

3.3 macOS

1. Find out how many processes you can run on your machine and remember that number. You can do this with

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

2. First, install python 3 from https://www.python.org/downloads/
3. Next, install homebrew and install the open-mpi version of MPI as well as mpi4py:

   ```
   $ 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
   ```

3.4 Ubuntu

These instructions apply to 20.04 and 21.04. Please use 20.04 in case you like to use GPUs.

1. First, find out how many processes you can run on your machine and remember that number. You can do this with

```
$ nproc
```

2. The installation 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. In our example, we have chosen the number 4, you may have to change that value

```
$ 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
```

3.5 Raspberry Pi

1. Install Open MPI in your pi by entering the following command assuming a PI4, PI3B+ PI3, PI2:

```
$ python -m venv ~/ENV3
$ source ~/ENV3/bin/activate
$ sudo apt-get install openmpi-bin
$ mpicc --showme:version
$ pip install mpi4py
```

If you have other Raspberry Pi's you may need to update the core count according to the hardware specification.

3.6 Testing the Installation

On all systems, the installation is very easy. Just change in our example the number 4 to the number of cores in your system.

```
(ENV3) $ mpiexec -n 4 python -m mpi4py.bench helloworld
```

You will see an output similar to

```
Hello, World! I am process 0 of 4 on myhost.
Hello, World! I am process 1 of 4 on myhost.
Hello, World! I am process 2 of 4 on myhost.
Hello, World! I am process 3 of 4 on myhost.
```

where `myhost` is the name of your computer.

Note: *the messages can be in a different order.*

4 Hosts, Machinefile, Rankfile

4.1 Running MPI on a Single Computer

In case you like to try out MPI and just use it on a single computer with multiple cors, you can skip this section for now and revisit it, once you scale up and use multiple computers.

4.2 Running MPI on Multiple Computers

MPI is designed for running programs on multiple computers. One of these computers serves as manager and communicates to its workers. To define on which computer is running what, we need to have a a configuration file that lists a number of hosts to participate in our set of machines, the MPI cluster.

The configuration file specifying this is called a machinefile or rankfile. We will explain the differences to them in this section.

4.2.1 Prerequisit

Naturally, the requisite to use a cluster is that you

1. have MPI and mpi4py installed on each of the computers, and
2. have access via ssh on each of these computers

If you use a Raspberry PI cluster, we recommend using our cloudmesh-pi-burn program [TODOREF]. This will conveniently create you a Raspberry PI cluster with login features established. You still need to install mpi4py, however on each node.

If you use another set of resources, you will often see the recommendation to use passwordless ssh key between the nodes. This we only recommend if you are an expert and have placed the cluster behind a firewall. If you experiment instead with your own cluster, we recommend that you use password-protected SSH keys on your manager node and populate them with ssh-copy-id to the worker computers. To not always have to type in your password to the different machines, we recommend you use `ssh-agent`, and `ssh-add`.

4.2.2 Using Hosts

In the case of multiple computers, you can simply specify the hosts as a parameter to your MPI program that you run on your manager node

```
(ENV3) $ mpiexec -n 4 -host re0,red1,red2,red3 python -m mpi4py.bench helloworld
```

To specify how many processes you like to run on each of them, you can use the option `-ppn` followed by the number.

```
(ENV3) $ mpiexec -n 4 -pn 2 -host re0,red1,red2,red3 python -m mpi4py.bench helloworld
```

As today we usually have multiple cores on a processor, you could be using that core count as the parameter.

4.2.3 Machinefile

To simplify the parameter passing to MPI you can use machine files instead. This allows you also to define different numbers of processes for different hosts. Thus it is more flexible. In fact, we recommend that you use a machine file in most cases as you then also have a record of how you configured your cluster.

The machine file is a simple text file that lists all the different computers participating in your cluster. As MPI was originally designed at a time when there was only one core on a computer, the simplest machine file just lists the different computers. When starting a program with the machine file as option, only one core of the computer is utilized.

The machinefile can be explicitly passed along as a parameter while placing it in the manager machine

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

An example of a simple machinefile contains the IP addresses. The username can be proceeded by the IP address.

```
pi@192.168.0.10:1
pi@192.168.0.11:2
pi@192.168.0.12:2
pi@192.168.0.13:2
pi@192.168.0.14:2
```

In many cases, your machine name may be available within your network and known to all hosts in the cluster. In that case, it is more convenient. To sue the machine names.

```
pi@red0:1
pi@red1:2
pi@red2:2
pi@red3:2
pi@red4:2
```

Please make sure to change the IP addresses or name of your hosts according to your network.

4.2.4 Rankfiles for Multiple Cores

In contrast to the host parameter, you can fine-tune the placement of processes to computers with a `rankfile`. This may be important if your hardware has, for example specific computers for data storage or GPUs.

If you like to add multiple cores from a machine, you can also use a `rankfile`

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

Where the rankfile contains:
rank 0=pi@192.168.0.10 slot=1:0
rank 1=pi@192.168.0.10 slot=1:1
rank 2=pi@192.168.0.11 slot=1:0
rank 3=pi@192.168.0.10 slot=1:1
```

In this configuration, we only use 2 cores from two different PIs.

5 MPI Functionality

In this section, we will discuss several useful MPI communication features.

5.1 Differences to the C Implementation of MPI

Before we start with a detailed introduction, we like to make those that have experience with non Python versions of MPI aware of some differences.

5.1.1 Initialization

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 deactivated. For more information on this topic, please check the original mpi4py documentation:

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

5.1.2 Capitalization for Pickle vs. Memory Messages

Another characteristic feature of mpi4py is the availability 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, the manual section Communicating Python Objects and Array Data.

5.2 MPI Functionality

5.2.1 Communicator

All MPI processes need to be addressable and are grouped in a `communicator`. The default communicator is called `world` and assigns a rank to each process within the communicator.

Thus all MPI programs we will discuss here start with

```
comm = MPI.COMM_WORLD
```

In the MPI program, the function

```
rank = comm.Get_rank()
```

returns the rank. This is useful to be able to write conditional programs that depend on the rank. Rank `0` is the rank of the manager process.

5.2.2 Point-to-Point Communication

5.2.2.1 Send and Recieve Python Objects The `send()` and `recv()` methods provide for functionality to transmit data between two specific processes in the communicator group. It can be applied to any Python data object that can be pickled. The advantage is that the object is preserved, however it comes with the disadvantage that pickling the data takes more time than a direct memory copy.

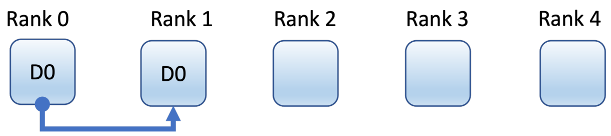

Figure 1: Sending and receiving data between two processes

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.

On 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.

```
#!/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 see, the transmission only occurred between processes 0 and 1, and no other process was affected.

5.2.2.2 Send and Recive Python Memory Objects The following example illustrates the use of the uppercase versions of the methods `comm.Send()` and `comm.Recv()` to perform a transmission of data between processes from memory to memory. In our example we will agian be sending a message between processors of rank 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]
```

5.2.2.3 Non-blocking send and Recieve MPI can also use non-blocking communications. This allows the program to send the message without waiting for the completion of the submission. This is useful for many parallel programs so we can overlap communication and computation while both take place simultaneously. The same can be done with receive, but if a message is not available and you do need the message, you may have to probe or even use a blocked receive. To wait for a message to be sent or received, we can also use the wait method , effectively converting the non-blocking message to a blocking one.

Next, we showcase 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 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.

```
#!/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
```

5.3 Collective Communication

5.3.1 Broadcast

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

5.3.1.1 Broadcast of a Python Object 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.

Thus, the two lines are functionally equivalent.

```
data = comm.bcast(data, root=0)
data = comm.bcast(data)
```

In our following 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.

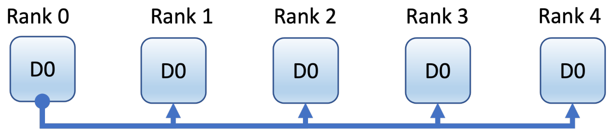

Figure 2: Broadcasting data from a root process to the rest of the processes in the communicator group

```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.

5.3.1.2 Broadcast of a Memory Object

In our following 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.

5.3.2 Scatter

While bradcast send all data to all processes, scatter send chunks of data to each process.

In our next example, we will `scatter` the members of a list among the processes in the communicator group. We illustrate the concept in the next figure, where we indicate the data that is scattered to the rnaked processes with D_i

5.3.2.1 Scatter Python Objects

The example program executing the sactter is showcased next

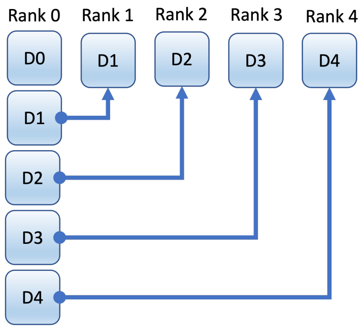

Figure 3: Example to scatter data to different processors from the one with rank 0

```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.

5.3.2.2 Scatter from Python Memory 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.

5.3.3 Gather

The gather function is the inverse function to scatter. Data from each process is gathered in consecutive order based on the rank of the processor.

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

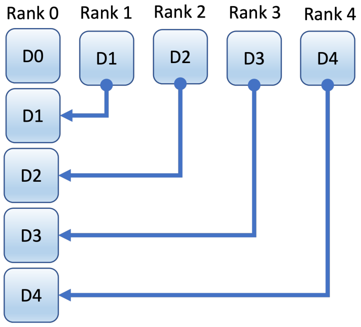

Figure 4: Example to gather data to different processors from the one with rank 0

```
#!/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.

5.3.3.2 Gather from Python Memory 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.

5.3.4 Allgather Memory Objects

This method is a many-to-many communication operation, where data from all processors is gathered in a continuous memory object on each of the processors. This is functionally equivalent to

1. A gather on rank 0
2. A Scatter from rank 0

However, this operation naturally has a performance bottleneck while all communication goes through rank0. Instead, we can use parallel communication between all of the processes at once to improve the performance. The optimization is implicit, and the user does not need to worry about it.

We demonstrate its use in the following 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).

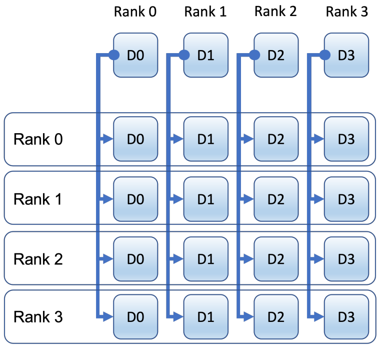

Figure 5: Example to gather the data from each process into ALL of the processes in the group

```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 similar to

```
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.

We have not provided an example for the Python object version as it is essentially the same and can easily be developed as an exercise.

5.4 Process Management

5.4.1 Dynamic Process Management with spawn

So far, we have focussed on MPI used on a number of hosts that are automatically creating the process when mpirun is used. However, MPI also offers the ability to sawn a process in a communicator group. This can be achieved by using a spawn communicator and command.

Using

```
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.

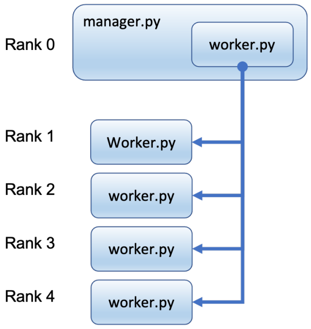

Figure 6: Example to spawn a program and start it on the different processors from the one with rank 0

```
#!/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`.

5.4.2 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.

```
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 so on...'))

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()
```

To run teh program use:

```
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 will output a png image of a Julia set upon successful execution.

You can modify your number of processors accordingly matching your hardware infrastructure.

For example, entering the number 3 will produce a 1920x1440 photo because 640x480 times 3 is 1920x1440. Then, the program should output a visualization of a Julia data set as a png image.

6 Simple MPI Example Programs

In this section, we will showcase to you some simple MPI example programs.

6.1 GPU Programming with MPI

In case you have access to computers with GPUS you can naturally utilize them accordingly from Python with the appropriate GPU drivers.

In case not all have a GPU, you can use rankfiles to control the access and introduce through conditional programming based on rank access to the GPUs.

7 Examples

7.1 MPI Ring Example

The MPI Ring example program is one of the classical programs every MPI programmer has seen. Here a message is sent from the manager to the workers while the processors are arranged in a ring, and the last worker sends the message back to the manager. Instead of just doing this once, our program does it multiple times and adds every time a communication is done 1 do the integer send around. Figure 1 showcases the process graph of this application.

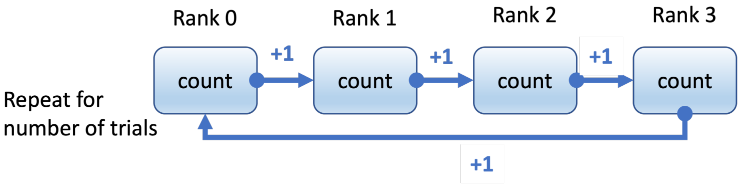

Figure 7: Processes organized in a ring perform a sum operation

In the example, the user provides an integer that is transmitted from the 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
# USSAGE: 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):
    comm = MPI.COMM_WORLD   # Communicator
    rank = comm.Get_rank()  # Get the rank of the current process
    size = comm.Get_size()  # Get the size of the communicator group
    if rank == 0:
        print(f'Communicator group with {size} processes')
        data = int(input('Enter an integer to transmit: '))  # Input the data
        data += 1                                            # Data is modified
    if rank == 0:  # ONly processor 0 uses the stopwatch
        Stopwatch.start(f"ring {size} {count}")
    for i in range(0, count):
        if rank == 0:
            comm.send(data, dest=rank + 1)  # send data to neighbor
            data = comm.recv(data, source=size - 1)
            if debug:
                print(f'Final data received in process 0: {data}')
        elif rank == size - 1:
            data = comm.recv(source=rank - 1)  # recieve data from neighbor
            data += 1                          # Data is modified
            comm.send(data, dest=0)            # Sent to process 0, closing the ring
        elif 0 < rank < size -1:
            data = comm.recv(source=rank - 1)  # recieve data from neighbor
            data += 1                          # Data is modified
            comm.send(data, dest=rank + 1)     # send to neighbor
    if rank == 0:
        print(f'Final data received in process 0: {data}')
        assert data == count * size          # verify
    if rank == 0:
        Stopwatch.stop(f"ring {size} {count}")   #print the time
        Stopwatch.benchmark()

if __name__ == '__main__':
    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.

7.2 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.

The program allows you to set the program parameters. Note that the program has on purpose a bug in it as it does not communicate the values m, max_number, or find with a broadcast from rank 0 to all workers. Your task is to modify and complete this program.

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

# To change the values set them on your terminal on the
# machine running rank 0 with

# export N=20
# export MAX=10
# export FIND=8

# Assignment:
# Add to this code the bradcast of the 3 parameters to all workers

import os
import random
from mpi4py import MPI

# Get 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


comm = MPI.COMM_WORLD   # Communicator
size = comm.Get_size()  # Number of processes
rank = comm.Get_rank()  # Rank of this process

# 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(rank, count, data)  # Print data from each process
count_data = comm.gather(count, root=0) # Gather the data

# Process 0 prints out the gathered data, rest of the processes
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
```

7.3 Monte Carlo Calculation of Pi

A very nice example to showcase the potential for doing lots of parallel calculations is to calculate the number pi. This is quite easily achieved while using a monte Carlo method.

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.

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

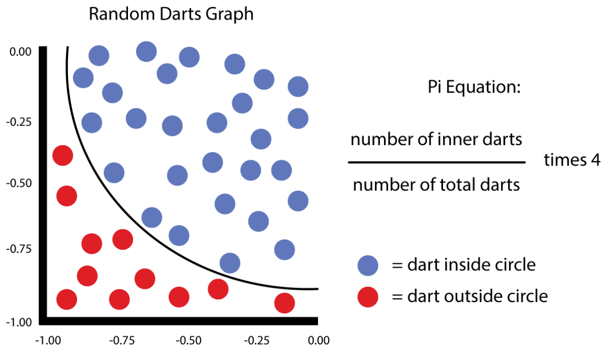

Figure 8: montecarlographic

The following montecarlo.py program generates an estimation of pi using the methodology and equation shown above. Increasing the total number of iterations will increase the accuracy.

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

start = time.time()

inside = 0                          # Number of darts that land inside.
trials = 100000                     # Number of Trials.

for i in range(0, trials):          # Iterate for the number of darts.
    x2 = r.random()**2              # Generate random x, y in [0, 1]
    y2 = r.random()**2

    if m.sqrt(x2 + y2) < 1.0:       # Increment if inside unit circle.
        inside += 1

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

print(pi)                            # Value of pi found
print(end - start)                   # Execution time
```

Instead of running this on one processor, we can run the calculation on many. Implicitly this increases the accuracy while running more trials at the same time as we run them all in parallel. Overhead does exist by starting the MPI program and gathering the result. However, if the trial number is large enough, it is negligible.

The following program shows the MPI implementation:

```
# Originaly from https://cvw.cac.cornell.edu/python/exercise
# Modified by us
"""
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
    produced by all workers, and compute the mean and
    standard deviation across the independent runs
    """

    if rank == 0:
        data = [N for i in range(size)]
    else:
        data = None
    data = comm.scatter(data, root=0)
    #
    pi_est = estimate_pi(N)
    #
    pi_estimates = comm.gather(pi_est, root=0)
    if rank == 0:
        return pi_estimates


def estimate_pi_statistics(comm, Ndarts, Nruns_per_worker):
    results = []
    for i in range(Nruns_per_worker):
        result = estimate_pi_in_parallel(comm, Ndarts)
        if rank == 0:
            results.append(result)
    if rank == 0:
        pi_est_mean = np.mean(results)
        pi_est_std  = np.std(results)
        return pi_est_mean, pi_est_std

if __name__ == '__main__':
    """
    for N from 4**5 to 4**14 (integer powers of 4),
    compute mean and standard deviation of estimates of pi
    by throwing N darts multiple times (Nruns_total times,
    distributed across workers)
    """
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    if rank == 0:
        print("MPI size = {}".format(size))
        sys.stdout.flush()
    Nruns_total = 64
    Nruns_per_worker = Nruns_total // size
    #
    estimates = []
    for log4N in range(5,15):
        N = int(4**log4N)
        result = estimate_pi_statistics(comm, N, Nruns_per_worker)
        if rank == 0:
            pi_est_mean, pi_est_std = result
            estimates.append((N, pi_est_mean, pi_est_std))
            print(N, pi_est_mean, pi_est_std)
            sys.stdout.flush()
    if rank == 0:
        estimates = np.array(estimates)
        plt.figure()
        plt.errorbar(np.log2(estimates[:,0]), estimates[:,1], yerr=estimates[:,2])
        plt.ylabel('estimate of pi')
        plt.xlabel('log2(number of darts N)')
        plt.savefig('pi_vs_log2_N.png')
        plt.figure()
        plt.ylabel('log2(standard deviation)')
        plt.xlabel('log2(number of darts N)')
        plt.plot(np.log2(estimates[:,0]), np.log2(estimates[:,2]))
        plt.savefig('log2_std_vs_log2_N.png')
    MPI.Finalize()

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

To run this program using git bash, change directory to the folder containing this program and issue the command:

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

The number after `-n` can be changed to however many cores one has on their processor.

Note: Please be advised that we use Cloudmesh.StopWatch which is a convenient program to measure time and display the details for the computer. However, it is not threadsafe and, at this time, only measures times in the second range. If your calculations for other programs are faster or the trial number is too slow, you can use other benchmarking methods.

Furthermore, the numba version of the program can be run instead, which is slightly faster.

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

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

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)

@jit(nopython=True)
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()
```

	No Jit (1280x960)	Jit Enabled (1280x960)	No Jit (1920x1440)	Jit Enabled (1920x1440)
1 Core	44.898 s	45.800 s	68.489 s	68.610 s
2 Cores	45.578 s	45.714 s	67.718 s	69.326 s
3 Cores	43.026 s	44.521 s	68.785 s	69.487 s
4 Cores	44.746 s	44.552 s	73.606 s	70.257 s
5 Cores	43.223 s	42.825 s	68.226 s	68.443 s
6 Cores	45.134 s	44.402 s	68.437 s	67.637 s

- These benchmark times were generated using a Ryzen 5 3600 CPU with 16 GB RAM on a Windows 10 computer.

The improvement in shorter runtime with jit is not apparent likely because the computations required to run this program are not very complex; the same reason applies to why the increase in cores does not improve runtime.

However, this example showcases how to run examples with a parameter to explore the behavior on multiple cores. Naturally, you can use and explore other parameters once added to the program.

7.3.1 Assignments

1. Use numba to speed up the code. Create a tutporial including instalation instructions.
2. Display results with matplotlib as created by the picture
3. Modify cloudmesh.Stopwatch so we can use it for smaller time measurments

7.4 Other MPI Example Programs

You will find lots of example programs on the internet when you search for it. Please let us know about such examples and we will add the here. You can also contribute to our repository and add example programs that we then include in this document. In return you will become a co-author or get acknowledged.

- A program to calculate PI is provided at
  - https://cvw.cac.cornell.edu/python/exercise
  - https://github.com/cloudmesh/cloudmesh-mpi/projects/1?card_filter_query=monte
- A program to calculate k-means is provided at
  - https://medium.com/@hyeamykim/parallel-k-means-from-scratch-2b297466fdcd

8 Parameter Management

Although this next topic is not directly related to MPI and mpi4py, it is very useful in general. Often we ask ourselves the question, how do we pass parameters to a program, includding MPI. There are multiple ways to achieve this. With environment variables, command-line arguments, and configuration files. We will explain each of these methods and provide simple example.

8.1 Using the Shell Variables to Pass Parameters

`os.environ` in Python allows us to easily access environment variables that are defined in a shell. It returns a dictionary having user's environmental variable as key and their values as value.

To demonstrate its use, we have written a `count.py` program that uses `os.environ` to optionally pass parameters to an MPI program.

This example is included in a previous Section `Counting Numbers` and we like you to look it over.

If the user changed the value of N, MAX, or FIND in the terminal using, for example, `export FIND="5"` (shown below) os.environ.get("FIND") would set the find variable equal to 5.

```
$ export FIND="5"
$ mpiexec -n 4 python count.py
1 0 [9, 6, 8, 3, 4, 8, 5, 6, 6, 3, 5, 6, 10, 5, 5, 1, 1, 2, 1, 3]
3 0 [3, 7, 2, 8, 4, 6, 5, 7, 4, 4, 7, 6, 1, 7, 10, 2, 1, 9, 2, 8]
2 0 [10, 8, 10, 8, 7, 2, 2, 7, 4, 3, 3, 7, 10, 8, 1, 5, 1, 4, 6, 5]
0 0 [5, 8, 9, 1, 2, 7, 1, 5, 5, 6, 3, 6, 10, 9, 7, 10, 5, 3, 6, 5]
0 [0, 0, 0, 0]
Total number of 5's: 0
```

However, if the user does not define any environment variables find will default to 8.

```
$ mpiexec -n 4 python count.py
1 0 [5, 5, 2, 6, 6, 3, 5, 3, 3, 2, 3, 9, 7, 1, 3, 7, 1, 7, 1, 3]
3 1 [7, 1, 5, 1, 2, 2, 10, 7, 2, 1, 2, 6, 4, 6, 10, 10, 5, 8, 10, 10]
2 0 [5, 1, 4, 4, 9, 9, 5, 1, 1, 3, 9, 3, 5, 2, 5, 7, 9, 7, 10, 5]
0 1 [6, 6, 5, 6, 4, 10, 3, 5, 5, 2, 5, 2, 7, 6, 7, 8, 5, 7, 6, 4]
0 [1, 0, 0, 1]
Total number of 8's: 2
```

Assignment:

1. One thing we did not do is use the broadcast method to properly communicate the 3 environment variables. We like you to improve the code and submit to us.

Setting the parameter can either be done vi the export shell command such as

```
$ export N=8
```

or while passing the parameter in the same line as a command such as demonstrated next

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

This can be generalized while using a file with many different parameters and commands. For example, placing this in a file called `run.sh`

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

It allows us to execute the programs sequentially in the file with

```
$ sh run.sh
```

Let us assume we use the python program

```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()
```

This Python program does not set a variable N on its own. It refers to os.environ which should have previously set N as shown in the beginning of this document's git bash log. The program does the same procedures as the previous program once N is set and passed from os.environ.

We are using, in our case also the cloudmesh.StopWatch to allow us easily to fgrep for the results we may be interested in to conduct benchmarks. Here is an example workflow to achieve this

```
# 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"
```

8.1.1 Using click to pass parameters

Click is a convenient mechanism to define parameters that can be passed via options to python programs. To showcase its use please inspect the following program

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

@click.command()
@click.option('--n', default=1, help='Number of processors.')
def work(n):
    n=int(n)
    StopWatch.start(f"processors {n}")
    sleep(0.1*n)
    print(n)
    StopWatch.stop(f"processors {n}")
    StopWatch.benchmark()

if __name__ == '__main__':
    work()
```

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
```

8.2 Resources

Here are a couple of links that may be useful. We have not yet looked over them but include them.

- 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

8.2.1 Assignment

1. Review the resources and provide a short summary that we add to this document above the appropriate link

9 Appendix

9.1 Git Bash on Windows

Git bash is a implementation of the bash shell fro windows that also includes Git.

Git is an open-source software which helps to manage repository version control, particularly with GitHub repos.

To verify whether you have Git in the first place, you can press `Win + R` on your desktop, type `cmd`, and press `Enter`. Then type `git clone` and press `Enter`. If you do not have Git installed, it will say `'git' is not recognized as an internal or external command...`

As long as Git does not change the structure of their website and hyperlinks, you should be able to download Git from here and skip to Step #2: https://git-scm.com/downloads

1. Open a web browser and search `git`. Look for the result that is from `git-scm.com` and click Downloads.

2. Click `Download for Windows`. The download will commence. Open the file once it is finished downloading.
3. The UAC Prompt will appear. Click `Yes` because Git is a safe program. It will show you Git's license: a GNU General Public License. After understanding the terms, click `Next`. 1. The GNU General Public License means that the program is open-source (free of charge).
4. Click `Next` to confirm that `C:\Program Files\Git` is the directory where you want Git to be installed.
5. Click `Next` unless you would like an icon for Git on the desktop (in which case you can check the box and then click `Next`).
6. Click `Next` to accept the text editor, click `Next` again to Let Git decide the default branch name, click `Next` again to run Git from the command line and 3rd party software, click `Next` again to use the OpenSSL library, click `Next` again to checkout Windows-style, click `Next` again to use MinTTY, click `Next` again to use the default git pull, click `Next` again to use the Git Credential Manager Core, click `Next` again to enable file system caching, and then click `Install` because the experimental features are not necessary.
7. Wait for the green progress bar to finish. Congratulations— you have installed Git and Git Bash. You can now run it as an administrator by pressing the Windows key, typing `git bash`, right clicking `Git Bash`, and clicking `Run as administrator`. Click `Yes` in the UAC prompt that appears.

9.2 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.

9.2.1 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 an administrative user:

Figure: Screenshot of opening gitbash in admin shell

```
$ 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

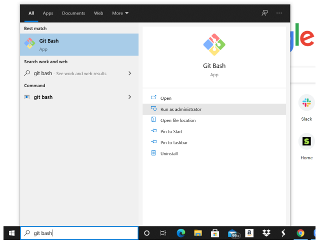

Figure 9: administrativegitbash

```
$ 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.

9.3 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`.

9.4 Benchmarks

This sectiion is in more detail published at this link. If the link does not work use this Link.

9.4.1 Introduction

We explain how we can manage long-running benchmarks. There are many useful tools to conducting benchmarks such as `timeit`, `cprofile`, `line_profiler`, and `memry_profiler` to name only a few. However, we present here an extremely easy way to obtain runtimes while dealing with the fact that they could run multiple hours or even days and could cause your system to crash. Hence if we wor to run it in a single program it will lead to a loss of information and many hours of unneeded replication.

We use and demonstrate how we achieve this with a simple StopWatch, creation of shell scripts and even the integration of Jupyter notebooks.

9.4.2 Prerequisites

As usual, we recommend that you use a virtual env. dependent on where your python 3 is installed, please adapt accordingly (python, or python3). Also, test out which version of python you have. On Windows, we assume you have gitbash installed and use it.

```
$ python3 -- version   # observe that you have the right version
$ python3 -m venv ~/ENV
$ source ~/ENV3/bin/activate
# or for Windows gitbash
# source ~/ENV3/Scripts/activate
```

9.4.3 System Parameters

It is essential that we benchmark programs to show their effect on the time consumed to obtain the results. Various factors play a role. This includes the number of physical computers involved, the number of processors on each computer, the number of cores on each computer, and the number of threads for each core. We can summarise these parameters as a vector such as

```
S(N, p, c, t)
```

Where

- `S` = is a placeholder for the system
- `N` = Number of computers or nodes
- `p` = Number of processors per node
- `c` = Number of cores per processor
- `t` = Number of threads per processor

In some cases, it may be more convenient to specify the total values as

```
S^T(N, N*p, N*p*c, N*p*c*t)
```

and

- `T` = indicates total

In the case of heterogeneous systems, we define multiple such vectors to form a list of vectors.

For the rest of the section, we assume the system is homogeneous.

9.4.3.1 System Information

Cloudmesh provides an easy command that can be used to obtain information to derive these values while using the command. However, it only works if the number of processors on the same node is 1.

```
pip install cloudmesh-cmd5
cms help    # call it after the install as it sets some defaults
cms sysinfo
```

The output will be looking something like

```
+------------------+----------------------------------------------+
| Attribute        | Value                                        |
+------------------+----------------------------------------------+
| cpu              | Intel(R) Core(TM) i7-7920HQ CPU @ 3.10GHz    |
| cpu_cores        | 4                                            |
| cpu_count        | 8                                            |
| cpu_threads      | 8                                            |
| frequency        | scpufreq(current=3100, min=3100, max=3100)   |
| mem.active       | 5.7 GiB                                      |
| mem.available    | 5.8 GiB                                      |
| mem.free         | 96.7 MiB                                     |
| mem.inactive     | 5.6 GiB                                      |
| mem.percent      | 63.7 %                                       |
| mem.total        | 16.0 GiB                                     |
| mem.used         | 8.2 GiB                                      |
| mem.wired        | 2.4 GiB                                      |
| platform.version | 10.16                                        |
| python           | 3.9.5 (v3.9.5:0a7dcbdb13, ...)               |
| python.pip       | 21.1.2                                       |
| python.version   | 3.9.5                                        |
| sys.platform     | darwin                                       |
| uname.machine    | x86_64                                       |
| uname.node       | mycomputer                                   |
| uname.processor  | i386                                         |
| uname.release    | 20.5.0                                       |
| uname.system     | Darwin                                       |
| uname.version    | Darwin Kernel Version 20.5.0: ....           |
| user             | gregor                                       |
+------------------+----------------------------------------------+
```

To obtain the vectors you can say

```
cms sysinfo -v
cms sysinfo -t
```

where `-v` specifies the vector and `-t` the totals. Knowing these values will help you structure your benchmarks.

9.4.3.2 Parameters

A benchmark is typically run while iterating over a number of parameters and measuring some system parameters that are relevant for the benchmark, such as the runtime of the program or application.

Let us assume our application is called `f` and its parameters are `x` and `y`

To create benchmarks over x and y we can generate them in various ways.

9.4.3.3 Python only solution

For all programs, we will store the output of the benchmarks in a directory called `benchmark`. Please create it.

```
$ mkdir benchmark
```

you may be able to run your benchmark simply as a loop this is especially the case for smaller benchmarks.

```
import pickle
from cloudmesh.common.StopWatch import StopWatch

def f(x,y, print_benchmark=False, checkpoint=True):
    # run your application with values x and y
    print (f"Calculate f({x},{y})")
    StopWatch.start(f"f{x},{y}")
    result = x*y
    StopWatch.stop(f"f{x},{y}")
    if print_benchmark:
        StopWatch.benchmark()
    if checkpoint:
        pickle.dump(result, open(f"benchmark/f-{x}-{y}.pkl", "wb" ))
    return result

x_min = 0
x_max = 2
d_x = 1
y_min = 0
y_max = 1
d_y = 1
for x in range(x_min, x_max, dx):
    for y in range(y_min, y_max, dy):
        # run the function with parameters
        result = f(x ,y, print_benchmark=True)
```

9.4.3.4 Script solution In some cases, the functions themselves may be large and in case the benchmark causes a crash of the python program executing it we would have to start over. In such cases, it is better to develop scripts that take parameters so we can execute the program through shell scripts and exclude those that fail.

For this, we rewrite the python program via command-line arguments that we pass along.

```
# stored in file f.py
import click

@click.command()
@click.option('--x', default=20, help='The x value')
@click.option('--x', default=40, help='The y value')
@click.option('--print_benchmark', default=True, help='prints the benchmark result')
@click.option('--checkpoint', default=True, help='Creates a checkpoint')
f(x,y, print_benchmark=False, checkpoint=True):
    ... see previous program
    return result

if __name__ == '__main__':
    f()
```

Now we can run this program with

```
$ python f.py --x 10 --y 5
```

To generate now the different runs from the loop we can do it either via Makefiles or write a program creating commands where we produce a script listing each invocation. Let us call this program `sweep-generator.py`.

```
x_min = 0
x_max = 2
d_x = 1
y_min = 0
y_max = 1
d_y = 1
for x in range(x_min, x_max, dx):
    for y in range(y_min, y_max, dy):
        print (f"cms banner f({x}, {y}; "
               f"python f.py --x {x} --y {y}")
```

The result will be

```
cms banner f(0,0); python f.py --x 0 --y 0
...
```

and so on. The banner will print a nice banner before you execute the real function so it is easier to monitor when execution

To create a shell script, simply redirect it into a file such as

```
$ python sweep-generator.py > sweep.sh
```

Now you can execute it with

```
$ sh sweep.sh | tee result.log
```

The `tee` command will redirect the output to the file result, while still reporting its progress on the terminal. In case you want to run it without monitoring or tee is not supported properly you just run it as

```
$ sh sweep.sh >> result.log
```

In case you need to monitor the progress for the latter you can use

```
$ tail -f result.log
```

The advantage of this approach is that you can in case of a failure identify which benchmarks succeeded and exclude them from your next run of `sweep.sh` so you do not have to redo them. This may be useful if you identify that you ran out of resources for a parameterized run and it crashed.

9.4.3.5 Integrating timers

The beauty about cloudmesh is that it has built-in timers and if properly used we can use them even across different invocations of the function f.

we simply have to `fgrep` to the log file to extract the information in the `csv` lines with

```
fgrep "#csv" result.log
```

This can then be further post-processed.

Cloudmesh also includes a `cloudmesh.Shell.cm_grep`, `cloudmesh.common.readfile`, and other useful functions to make the processing of shell scripts and their output easier.

9.4.3.6 Integration of Jupyter Notebooks Jupyter notebooks provide a simple mechanism to prototype. However, how do we now integrate them into a benchmarking suite? Certainly, we can just create the loop in the notebooks conducting the parameter sweep, but in case of a crash, this becomes highly unscalable.

So what we have to do is augment a notebook so that we can

1. pass along the parameters,
2. execute it from the command line.

For this, we use `papermill` that allows us to just do these two tasks. INstall it with

```
pip install papermill
```

Then when you open up jupyter-lablab and import our code. Create a new cell. In this cell you place all parameters for your run that you like to modify such as

```
x = 0
y = 0
```

This cell can be augmented with a tag called "parameters". To do this open the "cog" and enter in the tag name "parameters". Make sure you save the tag and the notebook. Now we can use `papermill` to run our notebook with parameters such as

```
$ mkdir benchmark
$ papermill sweep.ipynb benchmark/sweep-0-0.ipynb --x 0 --y 0 | tee benchmark/result-0-0.log
...
```

Naturally, we can auto-generate this as follows

```
x_min = 0
x_max = 2
d_x = 1
y_min = 0
y_max = 1
d_y = 1
for x in range(x_min, x_max, dx):
    for y in range(y_min, y_max, dy):
        print (f"cms banner f({x}, {y}; "
               f"papermill sweep.ipynb benchmark/sweep-{x}-{y}.ipynb"
               f"    --x {x} --y {y}"
               f"    | tee benchmark/result-{x}-{y}.log")
```

This will produce a series of commands that we can also redirect into a shell script and then execute

9.4.4 Combining the logs

As we have the logs all in the benchmark directory, we can even combine them and select the `csv` lines with

```
$ cat benchmark/*.log | fgrep "#csv"
```

Now you can apply further processing such as importing it into pandas or any other spreadsheet-like tools you like to use for the analysis.

9.4.4.1 Feedback This is a draft and if you see any issue, do a pull request and improve or send e-mail to laszewski@gmail.com with improvement suggestions.

10 Assignments

1. Develop a section explaining what MPI-IO is
2. Develop a section to explain Collective I/O with NumPy arrays.
3. Add a section on how to use Numpy with MPI, including the installation of NumPy. This is not to have a tutorial about numpy, but how to use numpy within mpi4py. Subtasks include
4. Download Numpy with `pip install numpy` in a terminal
5. `import numpy as np` to use NumPy in the program
6. Explain the advantages of NumPy over pickled lists
    - Numpy stores memory contiguously
    - Uses a smaller number of bytes
    - Can multiply arrays by index
    - It is faster
    - Can store different data types, including images
    - Contains random number generators
7. Add a specific, very small tutorial on using some basic numpy features as they may be useful for MPI application development. This may include the following and be added to the appendix
    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`

Please, note that we have a very comprehensive tutorial on NumPy and there is no point to repeat that, we may just point to it and improve that tutorial where needed instead.

5. The images in https://mpitutorial.com/tutorials/ seem much better when it comes to for example scatter. Your task is to create better images for all examples.
6. Convert the parallel rank program from https://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/ to mpi4py. Write a tutorial for it.
7. Develop tutorials that showcase multiple communicators and groups. See https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/
8. Complete the count example while adding a broadcast to it to communicate the parameters. Provide a modified tutorial.
9. Test out the machinefile, host, and rankfile section. Improve if needed.

11 Acknowledgements

We like to thank Erin Seliger and Agness Lungua for their effort on our very early draft of this paper.

12 References