Parallel Computing

Message Passing

Ackwonledgement: https://nyu-cds.github.io/python-mpi

MPI Installation

On your laptop

These instructions are based on installing compilers and MPI via the **conda package manager**, as it provides a convenient way to install binary packages in an isolated software environment.

- The instructions focus on installation on **MacOS** and **Linux** computers, as well as **Windows computers using the Windows Subsystem for Linux (WSL)**.
- Instructions for installing WSL on Windows can be found here
- Installing compilers and MPI natively on Windows is also possible through Cygwin and the Microsoft Distribution of MPICH, but we recommend that you instead use WSL which is available for Windows 10 and later versions.

https://pdc-support.github.io/introduction-to-mpi/setup.html

MPI Installation

Installing conda

Begin by installing Miniconda:

1. Download the 64-bit installer from here for your operating system
   - for MacOS and Linux, choose the bash installer
   - on Windows, open a Linux-WSL terminal and type: `wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh`. If wget is not a recognised command, type `sudo apt-get install wget` and provide the password you chose when installing WSL.
2. In a terminal, run the installer with `bash Miniconda3-latest-<operating-system>-x86_64.sh` (replace with correct name of installer)
3. Agree to the terms of conditions, specify the installation directory (the default is usually fine), and answer "yes" to the questions "Do you wish the installer to initialize Miniconda3 by running conda init?"

You now have miniconda and conda installed. Make sure that it works by typing `which conda` and see that it points to where you installed miniconda (you may have to open a new terminal first).

We recommend that you create an isolated conda environment (this is good practice in software development):

Bash

```
conda create --name mpi-intro python=3.7
conda activate mpi-intro
```

MPI Installation

If you want to use Python for the exercises, you will need to install mpi4py. mpi4py can be installed either using pip or conda, but with pip you will need to install MPI yourself first (e.g. OpenMPI or MPICH), while conda will install its own MPI libraries. If you don't already have MPI installed on your laptop, it will be easiest to use conda:

Bash

```
(mpi-intro) $ conda install -c conda-forge mpi4py
```

Please also verify the installation. The following command should not give an error message:

Bash

```
(mpi-intro) $ python -c "from mpi4py import MPI"
```

and the following command should give a version number:

Bash

```
(mpi-intro) $ mpirun --version
```

Point-to-point communication

The most used method of programming distributed-memory MIMD systems is message passing, or some variant of message passing. MPI is the most widely used standard.

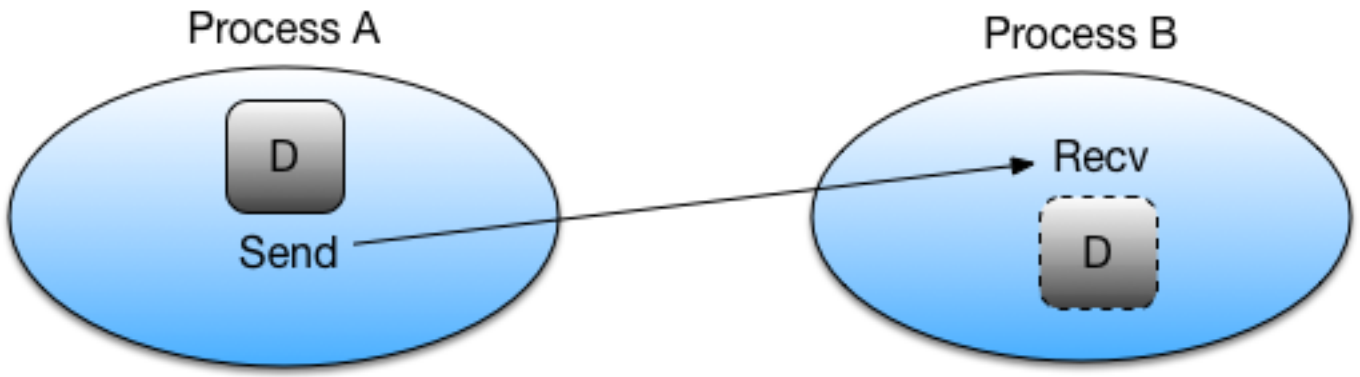
In basic message passing, the processes coordinate their activities by explicitly sending and receiving messages. Explicit sending and receiving messages is known as point-to-point communication.

MPI's send and receive calls operate in the following manner:

- First, process A decides a message needs to be sent to process B.
- Process A then packs up all of its necessary data into a buffer for process B.
- Process A indicates that the data should be sent to process B by calling the Send function.
- Before process B can receive the data, it needs to acknowledge that it wants to receive it. Process B does this by calling the Recv function.

Point-to-point communication

Every time a process sends a message, there must be a process that also indicates it wants to receive the message. i.e., calls to Send and Recv are always paired.



How does a process know where to send a message?

When an MPI program is first started

- The number of processes is fixed
- Each of the processes is assigned a unique integer starting from 0

Communicator

MPI processes are arranged in logical collections

MPI.COMM_WORLD contains all the processes in the MPI program

In mpi4py, communicators are represented by the Comm class.

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
print('size=%d, rank=%d' % (size, rank))
```

How does a process know where to send a message?

A process to learn about other processes, through

- Get_size(), the size of the communicator, that returns the total number of processes contained in the communicator
- Get_rank(), that returns the rank of the calling process within the communicator. Note that Get_rank() will return a different value for every process in the MPI program

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
print('size=%d, rank=%d' % (size, rank))
```

mpiexec

- It is possible to start an MPI with more or less processes than the program is expecting, so it is always a good idea to design the code so that it will accept any number of processes.
- Alternatively, check how many processes have been started and only run if the number is what you expect.

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
print('size=%d, rank=%d' % (size, rank))
```

```
mpiexec -n 4 python mpi1.py
```

```
size=4, rank=1
size=4, rank=0
size=4, rank=2
size=4, rank=3
```

What do you notice about the order that the program prints the values in? Hint: try running the program a few times and see what happens.

One MPI program, multiple MPI processes

When an MPI program is run, each process consists of the same code. However, as we've seen, there is one, and only one, difference: *each process is assigned a different rank value*. This allows code for each process to be embedded within one program file.

In the following code, all processes start with the same two numbers `a` and `b`. However, although there is only one file, each process performs a different computation on the numbers. Process 0 prints the sum of the numbers, process 1 prints the result of multiplying the numbers, and process 2 prints the maximum value.

```
from mpi4py import MPI
rank = MPI.COMM_WORLD.Get_rank()

a = 6.0
b = 3.0
if rank == 0:
        print(a + b)
if rank == 1:
        print(a * b)
if rank == 2:
        print(max(a,b))
```

Run this program using the command:

```
mpiexec -n 3 python mpi2.py
```

You should now see output similar to:

```
9.0
18.0
6.0
```

One MPI program, multiple MPI processes

Modify the above program to add the statement `print("end of MPI")` as the last line. What happens when you run the program now? What happens if you increase the number of processes to 4? Can you explain what you are seeing?

```
from mpi4py import MPI
rank = MPI.COMM_WORLD.Get_rank()

a = 6.0
b = 3.0
if rank == 0:
        print(a + b)
if rank == 1:
        print(a * b)
if rank == 2:
        print(max(a,b))
```

Run this program using the command:

```
mpiexec -n 3 python mpi2.py
```

Output?

Point-to-point communication

The simplest message passing involves two processes:
a sender and a receiver.

```
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

randNum = numpy.zeros(1)

if rank == 1:
        randNum = numpy.random.random_sample(1)
        print("Process", rank, "drew the number", randNum[0])
        comm.Send(randNum, dest=0)

if rank == 0:
        print("Process", rank, "before receiving has the number", randNum[0])
        comm.Recv(randNum, source=1)
        print("Process", rank, "received the number", randNum[0])
```

Point-to-point communication

The simplest message passing involves two processes:
a sender and a receiver.

```
import numpy
```

What is NumPy?

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

Point-to-point communication

The simplest message passing involves two processes:
a sender and a receiver.

```
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

randNum = numpy.zeros(1)

if rank == 1:
        randNum = numpy.random.random_sample(1)
        print("Process", rank, "drew the number", randNum[0])
        comm.Send(randNum, dest=0)

if rank == 0:
        print("Process", rank, "before receiving has the number", randNum[0])
        comm.Recv(randNum, source=1)
        print("Process", rank, "received the number", randNum[0])
```

Point-to-point communication

The simplest message passing involves two processes:
a sender and a receiver.

```
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

randNum = numpy.zeros(1)

if rank == 1:
        randNum = numpy.random.random_sample(1)
        print("Process", rank, "drew the number", randNum[0])
        comm.Send(randNum, dest=0)

if rank == 0:
        print("Process", rank, "before receiving has the number", randNum[0])
        comm.Recv(randNum, source=1)
        print("Process", rank, "received the number", randNum[0])
```

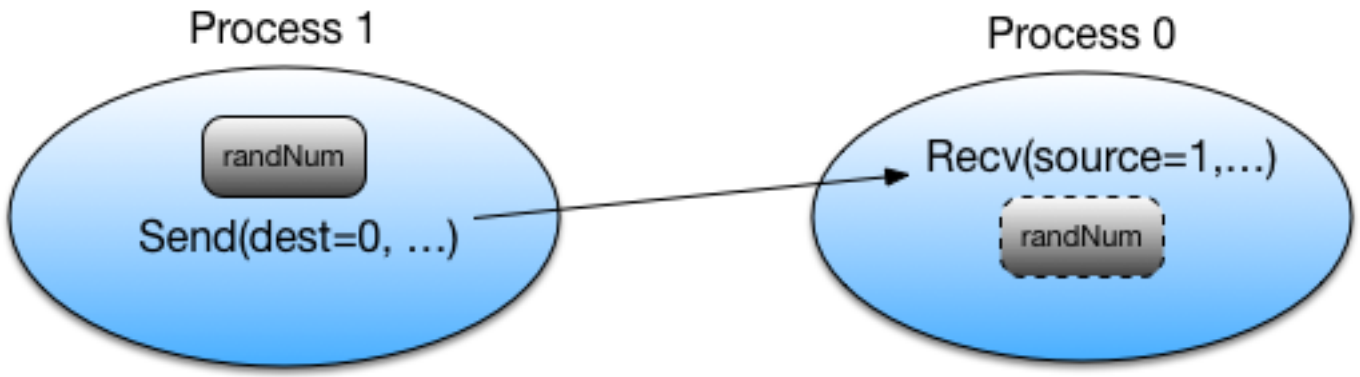
Run this program using the command:

```
mpiexec -n 2 python mpi3.py
```

```
Process 0 before receiving has the number 0.0
Process 0 received the number 0.815583406506
Process 1 drew the number 0.815583406506
```

Blocking functions

The Send and Recv functions are referred to as blocking functions (we will look at non-blocking functions later). If a process calls Recv it will simply wait until a message from the corresponding Send is received before proceeding. Similarly the Send will wait until the message has been reveived by the corresponding Recv.



Deadlock

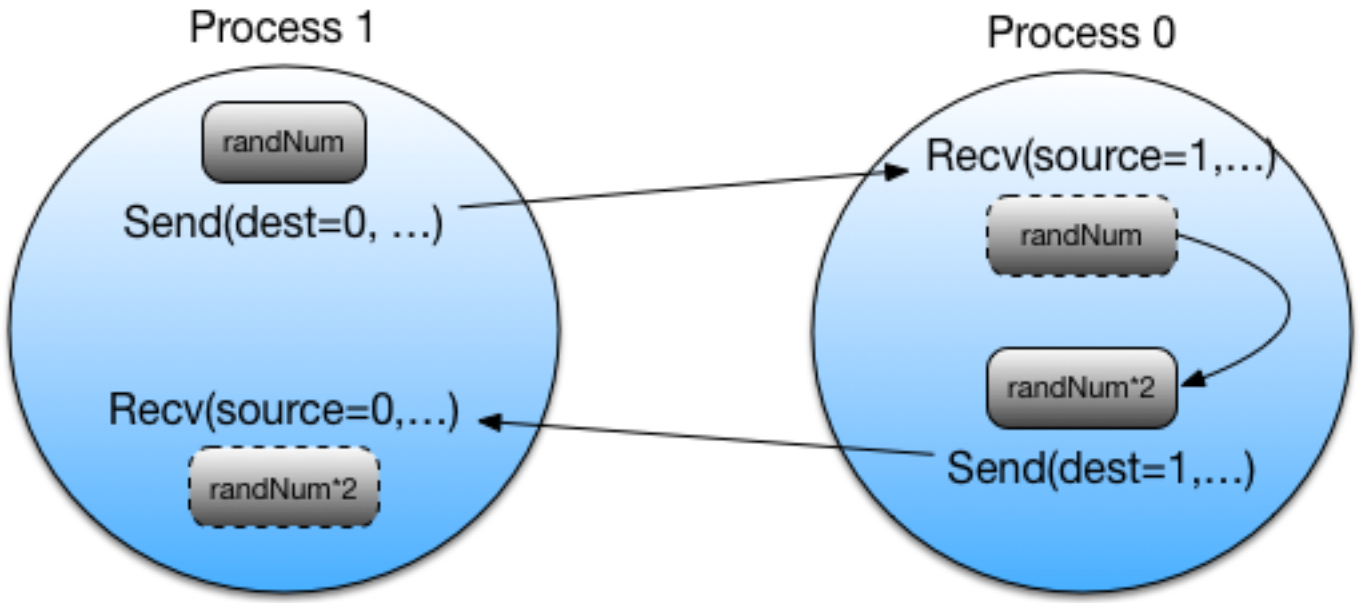
Because `Send` and `Recv` are blocking functions, a very common situation that can occur is called deadlock. This happens when one process is waiting for a message that is never sent. We can see a simple example of this by commenting out the `comm.Send` and running the program below.

```
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

randNum = numpy.zeros(1)

if rank == 1:
        randNum = numpy.random.random_sample(1)
        print("Process", rank, "drew the number", randNum[0])
        comm.Send(randNum, dest=0)

if rank == 0:
        print("Process", rank, "before receiving has the number", randNum[0])
        comm.Recv(randNum, source=1)
        print("Process", rank, "received the number", randNum[0])
```

More Send and Recv

Previously, we saw how to send a message from one process to another. Now we're going to try sending a message to a process and receiving a message back again.

Let's modify the previous code so that when the process 0 receives the number, it multiplies it by two and sends it back to process 1. Process 1 should then print out the new value.



More Send and Recv

```
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

randNum = numpy.zeros(1)

if rank == 1:
        randNum = numpy.random.random_sample(1)
        print("Process", rank, "drew the number", randNum[0])
        comm.Send(randNum, dest=0)
        comm.Recv(randNum, source=0)
        print("Process", rank, "received the number", randNum[0])

if rank == 0:
        print("Process", rank, "before receiving has the number", randNum[0])
        comm.Recv(randNum, source=1)
        print("Process", rank, "received the number", randNum[0])
        randNum *= 2
        comm.Send(randNum, dest=1)
```

Run this program using the command:

```
mpiexec -n 2 python mpi4.py
```

Here is the output you should see:

```
Process 0 before receiving has the number 0.0
Process 0 received the number 0.405456788104
Process 1 drew the number 0.405456788104
Process 1 received the number 0.810913576208
```

More Send and Recv

The receiving process does not always need to specify the source when issuing a `Recv`. Instead, the process can accept any message that is being sent by another process. This is done by setting the source to `MPI.ANY_SOURCE`.

```
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

randNum = numpy.zeros(1)

if rank == 1:
        randNum = numpy.random.random_sample(1)
        print("Process", rank, "drew the number", randNum[0])
        comm.Send(randNum, dest=0)
        comm.Recv(randNum, source=MPI.ANY_SOURCE)
        print("Process", rank, "received the number", randNum[0])

if rank == 0:
        print("Process", rank, "before receiving has the number", randNum[0])
        comm.Recv(randNum, source=MPI.ANY_SOURCE)
        print("Process", rank, "received the number", randNum[0])
        randNum *= 2
        comm.Send(randNum, dest=1)
```

A Final word on the Send and Recv methods

📌 `Comm.Send(buf, dest=0, tag=0)`

Performs a basic send. This send is a point-to-point communication. It sends information from exactly one process to exactly one other process.

Parameters:

- `Comm` (MPI comm) – communicator we wish to query
- `buf` (choice) – data to send
- `dest` (integer) – rank of destination
- `tag` (integer) – message tag

📌 `Comm.Recv(buf, source=0, tag=0, status=None)`

Performs a point-to-point receive of data.

Parameters:

- `Comm` (MPI comm) – communicator we wish to query
- `buf` (choice) – initial address of receive buffer (choose receipt location)
- `source` (integer) – rank of source
- `tag` (integer) – message tag
- `status` (Status) - status of object

A Final word on the Send and Recv methods

📌 `Comm.Send(buf, dest=0, tag=0)`

Sometimes there are cases when a process might have to send many different types of messages to another process. Instead of having to go through extra measures to differentiate all these messages, MPI allows senders and receivers to also specify message IDs (known as *tags*) with the message. The receiving process can then request a message with a certain tag number and messages with different tags will be buffered until the process requests them.

- `tag` (integer) – message tag

📌 `Comm.Recv(buf, source=0, tag=0, status=None)`

- `tag` (integer) – message tag