



hendrik.nolte@gwdg.de

Hendrik Nolte

Parallel Python

MPI for Python

Learning Objectives

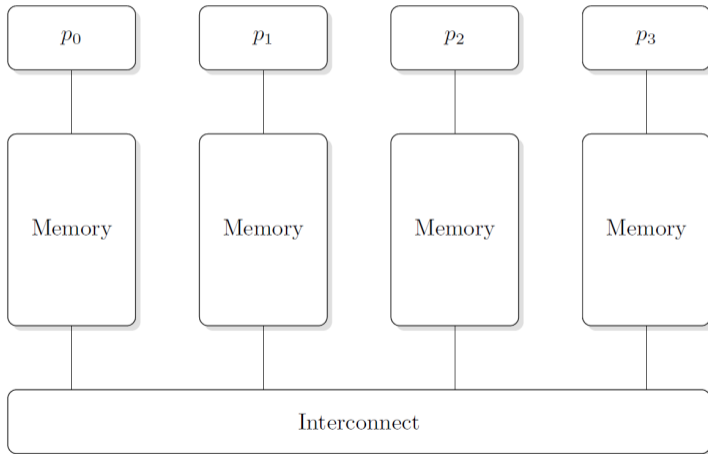
- Understand Python MPI libraries and communication
- Run Python code in parallel on multiple machines
- Improve Numpy code via MPI parallelization

Table of contents

- 1 Overview
- 2 Sending and Receiving
- 3 Communication
- 4 Process Management

Repetition - MPI

MPI is used for distributed memory systems



Execution of an MPI Program

- Launching an MPI-parallelized Pythonscript (e.g., with `mpirun`, `mpiexec` ...) will start n Python interpreters
- All processes contain the same code, thus they are independent and identical processes

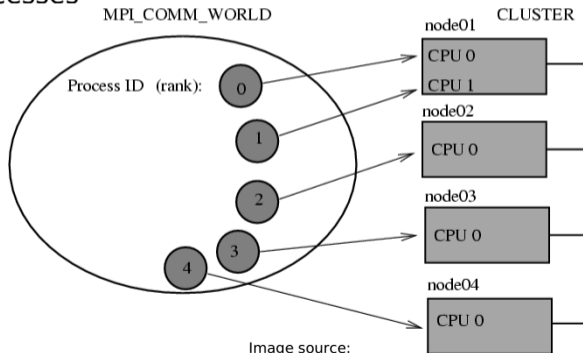


Image source:

https://lsi.vc.ehu.eus/pablogn/docencia/manuales/linuxcourse.rutgers.edu/lessons/HPC_1/sec_4.php.html

Ranks and Communicators

- Rank: Unique id given to one process to distinguish between them
- Communicator: Group of processes
 - ▶ Communication takes always place in a certain communicator
 - ▶ Rank of a process can be different in different communicators

minimal-mpi.py

```
from mpi4py import MPI
```

```
comm = MPI.COMM_WORLD
```

```
size = comm.get_size()
```

```
rank = comm.get_rank()
```

```
If rank == 0:
```

```
    # do stuff that only process 0 should do
```

Sending and Receiving Data - Example

send-recv.py

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    data = {'a': 1, 'b': 2, 'c': 'test string'}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
    print(data)
```

run send-recv

```
$ mpirun -n 2 python3 send-recv.py
{'a': 1, 'b': 2, 'c': 'test string'}
```

Sending and Receiving Data - Summary

- Arbitrary Python objects can be sent and received without the manual need for serialization from the user
 - ▶ MPI functions pickle under the hood
- `send(data, dest, tag)`
 - ▶ `data`: Data, i.e. a Python object to send
 - ▶ `dest`: Rank of the destination process
 - ▶ `tag`: Arbitrary id for this message
- `recv(source, tag)`
 - ▶ `source`: Rank of the sending process
 - ▶ `tag`: ID of the message, must match the tag in the send function
 - ▶ The **return value** is the send data
- There are also the non-blocking functions `isend` and `irecv`

Sending and Receiving Data - Summary

- Objects need to be serialized to a byte stream when sending
- Byte stream needs to be deserialized on the receiving process
 - ▶ Additional overhead for communication
- Specifically in scientific computing it is necessary to be able to efficiently exchange large amounts of data
- For this contiguous NumPy arrays can be communicated with a largely reduced overhead
- Use `Send(data, dest, tag)` and `Recv(data, source, tag)`
 - ▶ Notice the **capitalized Send** and **Recv**
- Data array has to exist beforehand on the receiving process

Sending and Receiving NumPy Arrays

numpy-send-recv.py

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    data = numpy.arange(100, dtype=numpy.float)
    comm.Send(data, dest=1, tag=11)
elif rank == 1:
    data = numpy.empty(100, dtype=numpy.float)
    comm.Recv(data, source=0, tag=11)
```

Summary

- send/recv for all general Python objects, slow
- Send/Recv for continuous arrays, fast

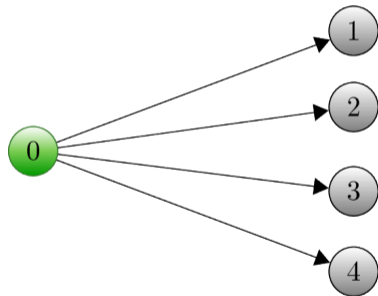
Collective Communication - Broadcast

broadcast.py

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'key1' : [7, 2.72, 2+3j],
            'key2' : ('abc', 'xyz')}
else:
    data = None
data = comm.bcast(data, root=0)
```



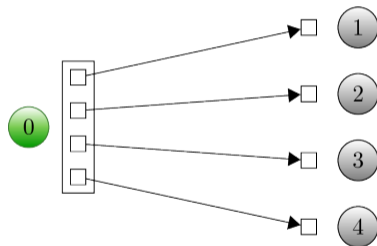
Collective Communication - Scatter

scatter.py

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
    data = [(i+1)**2 for i in range(size)]
else:
    data = None
data = comm.scatter(data, root=0)
assert data == (rank+1)**2
```



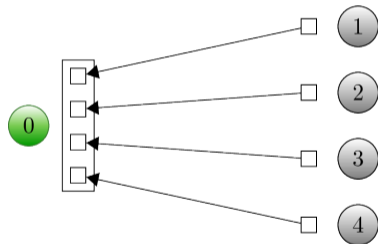
Collective Communication - Gather

gather.py

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

data = (rank+1)**2
data = comm.gather(data, root=0)
if rank == 0:
    for i in range(size):
        assert data[i] == (i+1)**2
else:
    assert data is None
```



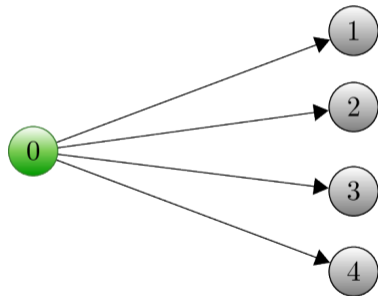
Collective Communication - Broadcasting a Numpy Array

broadcast-numpy.py

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = np.arange(100, dtype='i')
else:
    data = np.empty(100, dtype='i')
comm.Bcast(data, root=0)
for i in range(100):
    assert data[i] == i
```



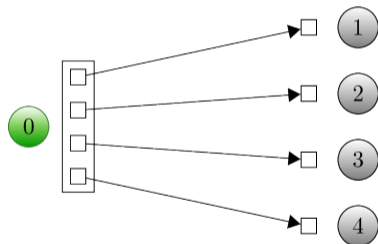
Collective Communication - Scatter a Numpy Array

scatter-numpy.py

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

sendbuf = None
if rank == 0:
    sendbuf = np.empty([size, 100], dtype='i')
    sendbuf.T[:, :] = range(size)
recvbuf = np.empty(100, dtype='i')
comm.Scatter(sendbuf, recvbuf, root=0)
assert np.allclose(recvbuf, rank)
```

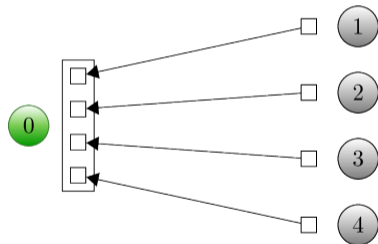


Collective Communication - Gather a Numpy Array

gather.py

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
sendbuf = np.zeros(100, dtype='i') + rank
recvbuf = None
if rank == 0:
    recvbuf = np.empty([size, 100], dtype='i')
comm.Gather(sendbuf, recvbuf, root=0)
if rank == 0:
    for i in range(size):
        assert np.allclose(recvbuf[i,:], i)
```



Exceptions and Deadlocks

- Upon import, mpi4py is being automatically initialized
- Avoid a deadlock by using `-m mpi4py`

```
deadlock.py
```

```
from mpi4py import MPI
assert MPI.COMM_WORLD.Get_size() > 1
rank = MPI.COMM_WORLD.Get_rank()
if rank == 0:
    1/0
    MPI.COMM_WORLD.send(None, dest=1, tag=42)
elif rank == 1:
    MPI.COMM_WORLD.recv(source=0, tag=42)
```

```
$ mpirun -n 10 python3 deadlock.py
Traceback (most recent call last):
  ZeroDivisionError: division by zero
```

```
$ mpirun -n 10 python3 -m mpi4py deadlock.py
Traceback (most recent call last):
```

Dynamic Process Management I

- Since MPI-2 provides a process models, which allows creating new processes and establishing communication between them and the existing MPI application
- Useful for sequential applications built on top of parallel modules or in a client/server model

communicator.py

```
from mpi4py import MPI
import numpy
import sys

comm = MPI.COMM_SELF.Spawn(sys.executable,
args=['cpi.py'],
maxprocs=5)

N = numpy.array(100, 'i')
comm.Bcast([N, MPI.INT], root=MPI.ROOT)
PI = numpy.array(0.0, 'd')
comm.Reduce(None, [PI, MPI.DOUBLE],
op=MPI.SUM, root=MPI.ROOT)
print(PI)
comm.Disconnect()
```

Dynamic Process Management II

advanced-communicator.py

```
from mpi4py import MPI
import numpy
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)
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)
comm.Disconnect()
```

Further Reading

Single Most important Source:

<https://mpi4py.readthedocs.io/en/stable/index.html>