An Introduction to MPI with Python

Prof. Amey Karkare

Dr. Anando Gopal Chatterjee

Acknowledgements / Disclaimer

- Reused slides by
  - Preeti Malakar
  - Stephen Weston
- Rather than having an independent tutorial, we shall compare Python-MPI support with the C version

Anaconda 3

If you install Anaconda3 on your local system, mpi4py must be installed using

```
$ conda install mpi4py
```

and “mpiexec” of anaconda3 must be used. To verify that write

```
$ which mpiexec
```

You must get the path of anaconda installation in its output.

To run a code in 8 cores type

```
$ mpiexec -n 8 python eg1.py
```

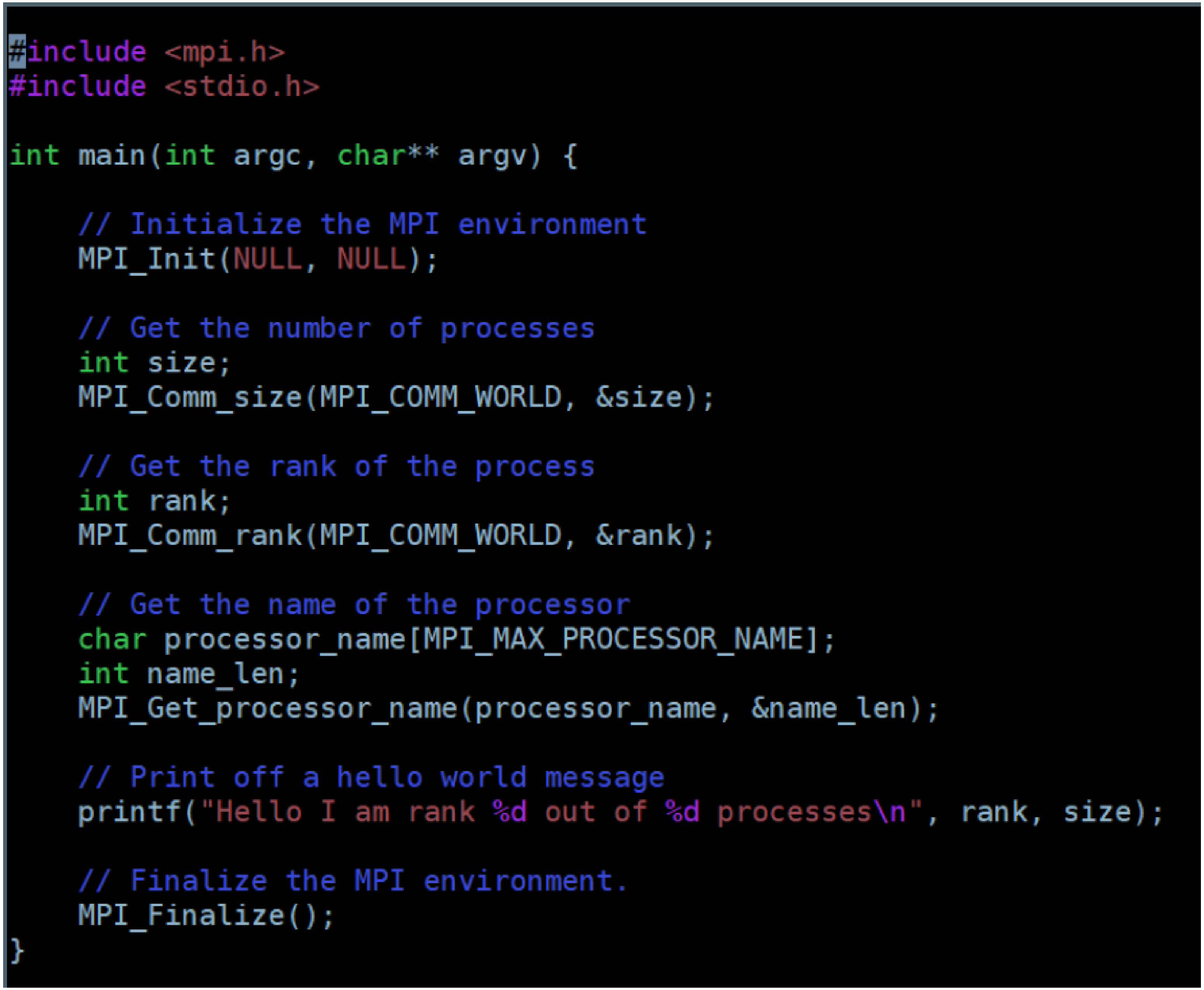
Python-MPI Sources

- Many Python modules support parallel computing.
- See http://wiki.python.org/moin/ParallelProcessing
- Some active ones:
  - mpi4py
  - multiprocessing
  - jug
  - Celery
  - dispy
  - Parallel Python

The mpi4py module

- Python interface to MPI
- Based on MPI-2 C++ bindings
- Almost all MPI calls supported
- Popular on Linux clusters and in the SciPy community
- Operations are primarily methods on communicator objects
- Supports communication of pickleable Python objects
- Optimized communication of NumPy arrays
- API docs: http://pythonhosted.org/mpi4py/apiref/index.html

A Minimal MPI Program (C)



```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {

    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Get the rank of the process
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello I am rank %d out of %d processes\n", rank, size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

A Minimal MPI Program (Python)

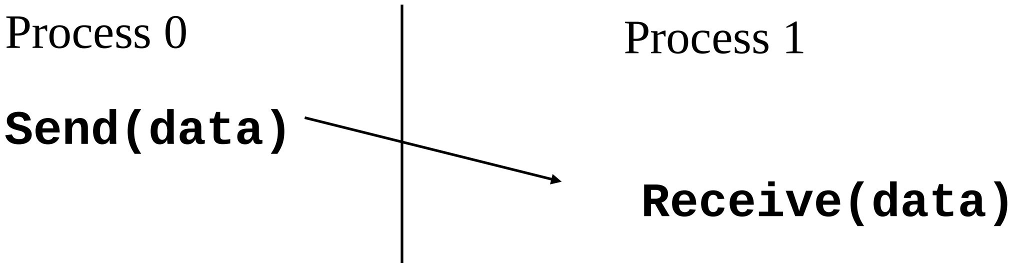
```
from mpi4py import MPI
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
name = MPI.Get_processor_name()
print(“Hello I am rank %d of %d” %

      (rank, size))
```

Notes on C and Python

- C and Python bindings correspond closely
- In C, mpi.h must be #included
- In Python, MPI must be imported from mpi4py module
- In C,
  - MPI_Init and MPI_Finalize are called explicitly
- In Pyhton,
  - MPI Init is called when mpi4py is imported
  - MPI Finalize is called when the script exits

MPI Basic Send/Receive

- We need to fill in the details in



Process 0

`Send(data)`

Process 1

`Receive(data)`

- Things that need specifying:
  - How will “data” be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

send and recv

- “send” and “recv” are the most basic communication operations.
- comm.send(obj, dest, tag=0)
- comm.recv(source=MPI.ANY SOURCE, tag=MPI.ANY TAG, status=None)
- These are blocking operations
  - can cause your program to hang.

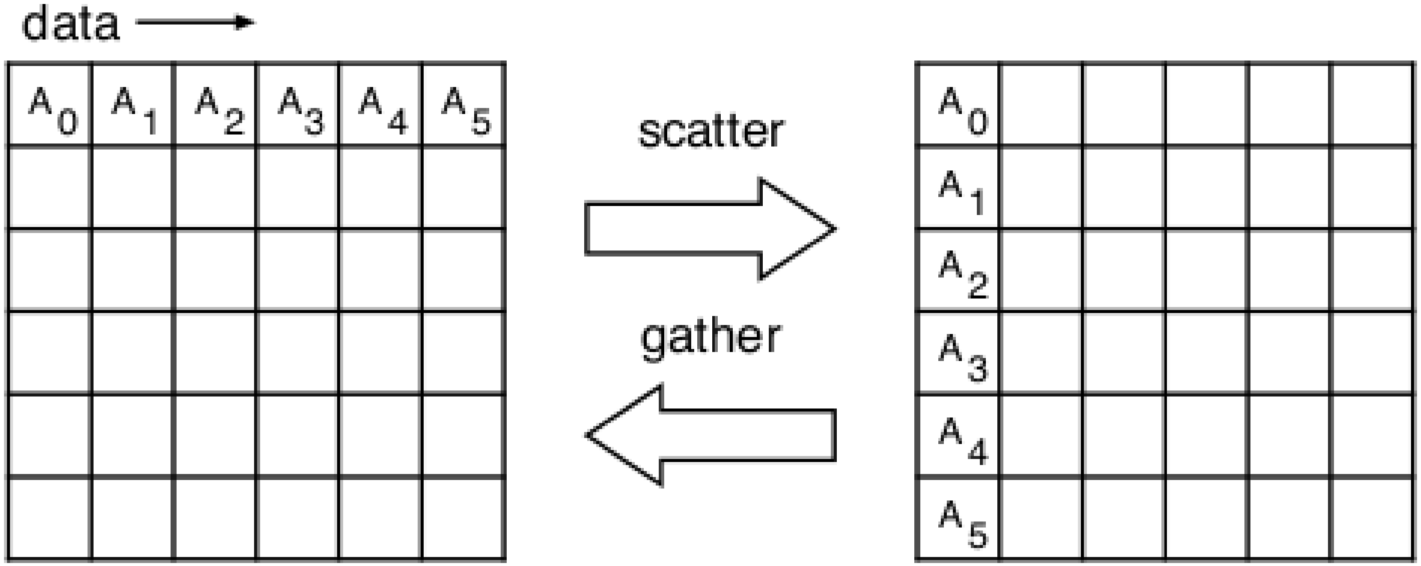
Example

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

if rank == 0:
    msg = 'Hello, there'
    comm.send(msg, dest=1)
elif rank == 1:
    s = comm.recv()
    print "rank %d: %s" % (rank, s)
```

Other operations

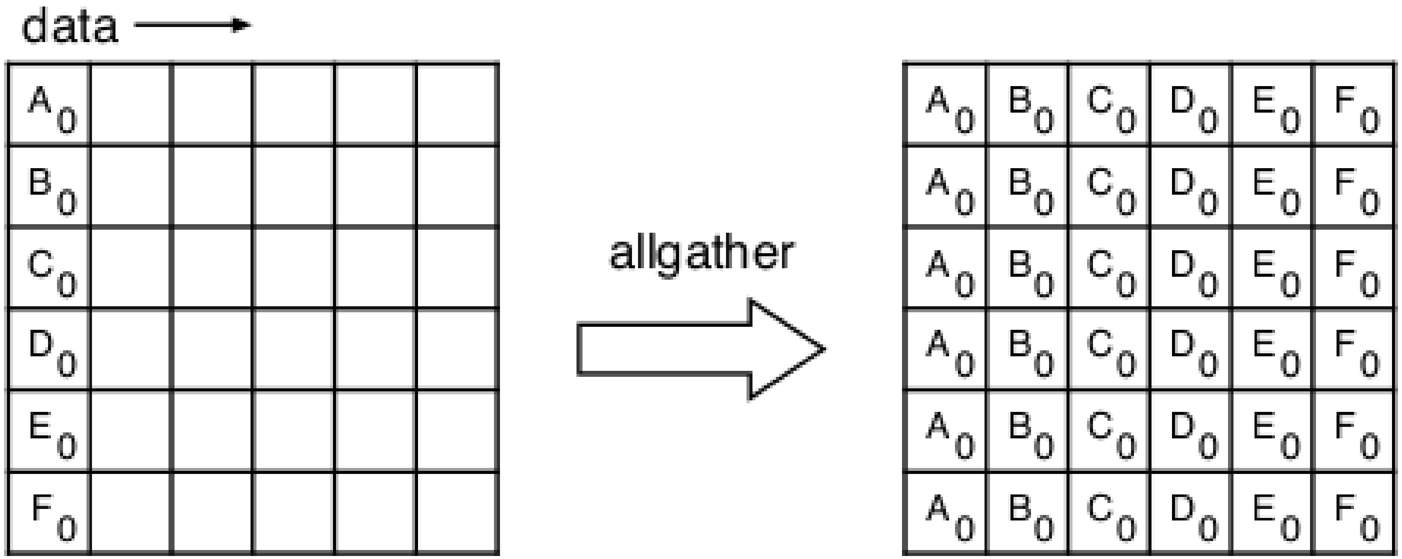
Collectives: Scatter and Gather

comm.scatter(sendobj, root=0) - where sendobj is iterable
comm.gather(sendobj, root=0)



Collectives: All Gather

comm.allgather(sendobj) - where sendobj is iterable

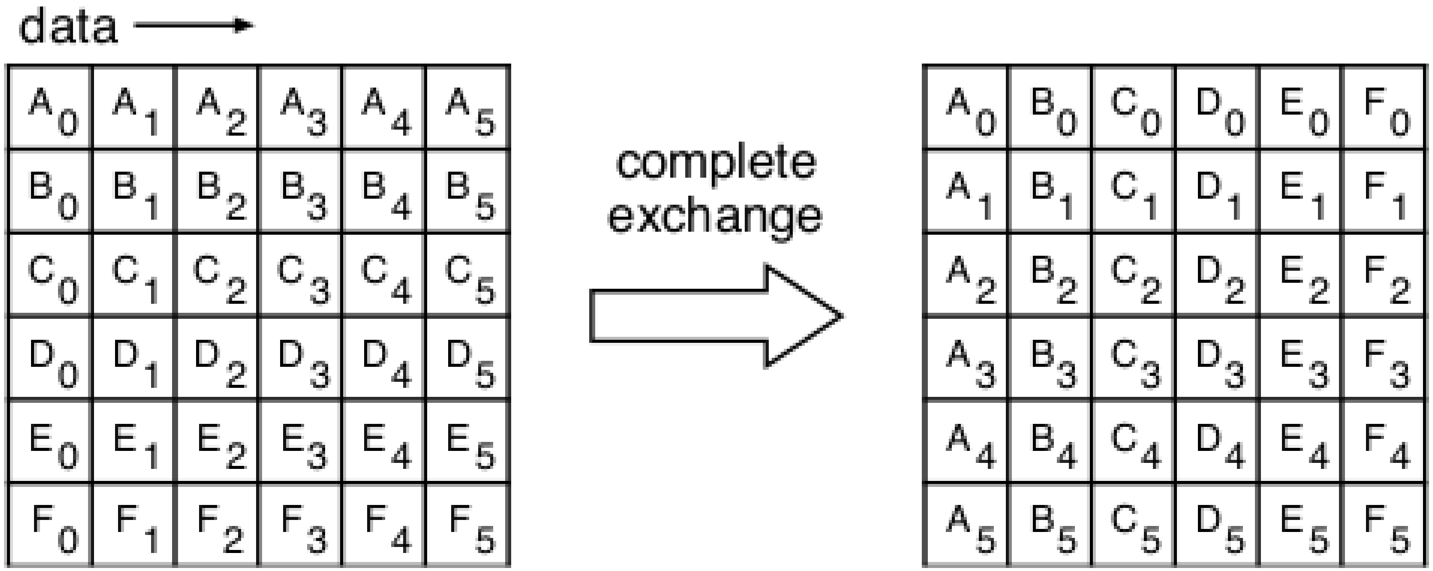


A_0					
B_0					
C_0					
D_0					
E_0					
F_0					

A_0	B_0	C_0	D_0	E_0	F_0
A_0	B_0	C_0	D_0	E_0	F_0
A_0	B_0	C_0	D_0	E_0	F_0
A_0	B_0	C_0	D_0	E_0	F_0
A_0	B_0	C_0	D_0	E_0	F_0
A_0	B_0	C_0	D_0	E_0	F_0

Collectives: All to All

comm.alltoall(sendobj) - where sendobj is iterable

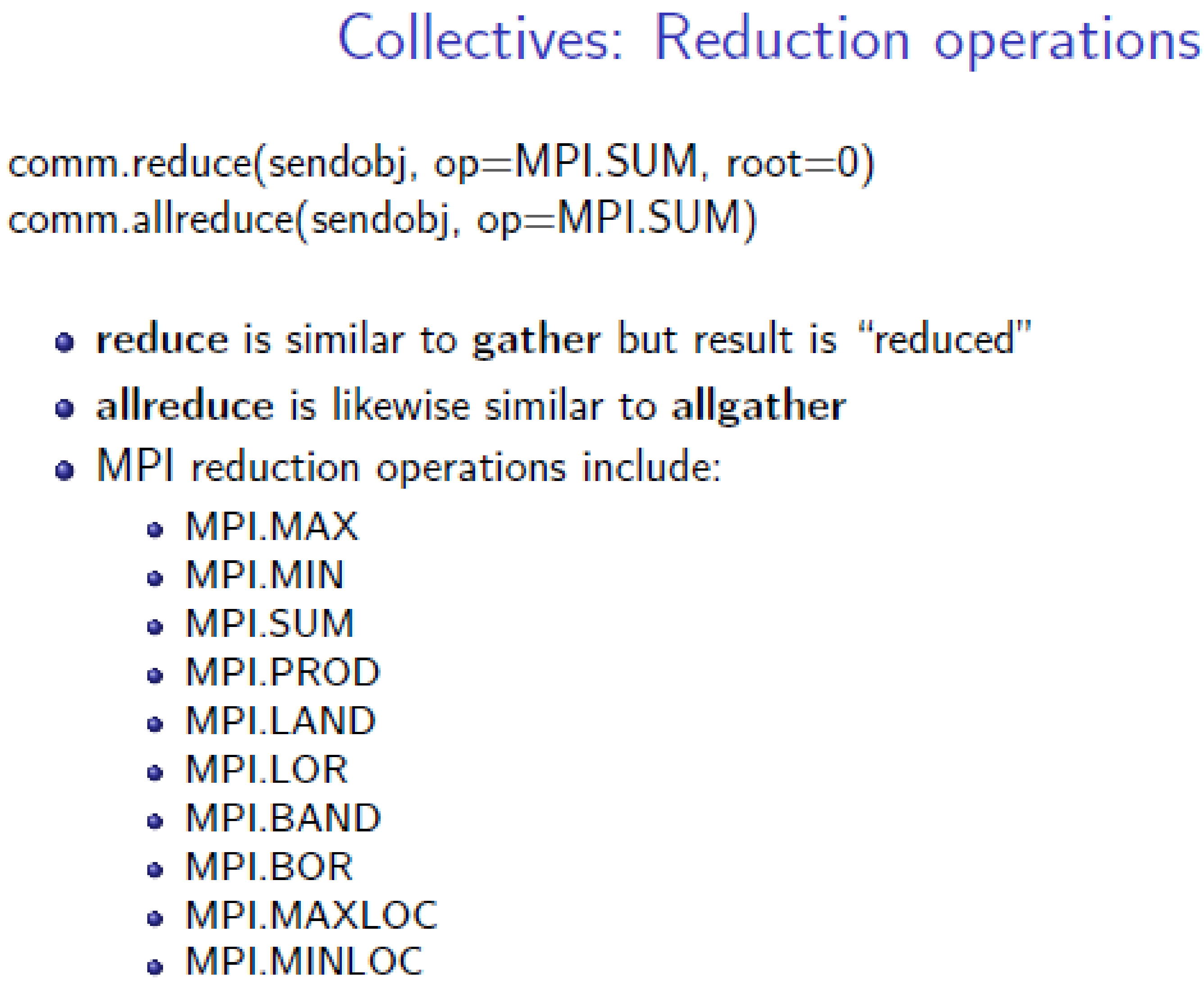


data →

A_0	A_1	A_2	A_3	A_4	A_5
B_0	B_1	B_2	B_3	B_4	B_5
C_0	C_1	C_2	C_3	C_4	C_5
D_0	D_1	D_2	D_3	D_4	D_5
E_0	E_1	E_2	E_3	E_4	E_5
F_0	F_1	F_2	F_3	F_4	F_5

complete exchange

A_0	B_0	C_0	D_0	E_0	F_0
A_1	B_1	C_1	D_1	E_1	F_1
A_2	B_2	C_2	D_2	E_2	F_2
A_3	B_3	C_3	D_3	E_3	F_3
A_4	B_4	C_4	D_4	E_4	F_4
A_5	B_5	C_5	D_5	E_5	F_5



Collectives: Reduction operations

```
comm.reduce(sendobj, op=MPI.SUM, root=0)
comm.allreduce(sendobj, op=MPI.SUM)
```

- **reduce** is similar to **gather** but result is "reduced"
- **allreduce** is likewise similar to **allgather**
- MPI reduction operations include:
  - MPI.MAX
  - MPI.MIN
  - MPI.SUM
  - MPI.PROD
  - MPI.LAND
  - MPI.LOR
  - MPI.BAND
  - MPI.BOR
  - MPI.MAXLOC
  - MPI.MINLOC

Sending Python Objects

- Generic Python objects can be sent between processes using the “lowercase” communication methods if they can be pickled.
- Buffer-provider objects can be sent between processes using the “uppercase” communication methods which can be significantly faster.

Tutorial

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

- Lots of Examples