Introduction to MPI for Python with MPI4Py

Pascal Paschos, PhD
Sr. HPC Specialist
pascal.paschos@northwestern.edu

Alper Kinaci, PhD
Sr. Computational Specialist
akinaci@northwestern.edu

Research Computing Services

Northwestern
INFORMATION TECHNOLOGY

Message Passing Interface (MPI)

- A standard library interface for coding on distributed memory systems
- Only message passing library that can be considered a standard
- It is supported on virtually all HPC platforms
- C, C++, Fortran, Python, Java and R bindings
- Different implementations exist: OpenMPI, MPICH, Intel MPI etc.
- The programmer is responsible for correctly identifying parallelism and implementing parallel algorithms

Advantages of Message Passing

- *Universality*: The model works on separate processors connected by any network
- *Performance/Scalability:*
  - As core counts increase, management of memory becomes a key performance factor
  - Each message passing process directly uses only its local data, avoiding the complexities and possible conflicts of shared data model

Layout in Fortran & C

FORTRAN

```fortran
PROGRAM mpilayout
USE MPI ! f90 designation
! include "mpif.h" ! f77 designation
integer ierr


CALL MPI_INIT(ierr)


CALL MPI_FINALIZE(ierr)


END PROGRAM mpilayout
```

C

```c
#include <mpi.h>

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


MPI_Init(&argc, &argv);


MPI_Finalize();


}
```

NON-MPI

MPI CALL

NON-MPI

Layout of an Python Code with mpi4y

```
#!/usr/bin/env python
"""
Simple MPI layout
"""

from mpi4py import MPI
```

MPI REGION

- MPI module in mpi4py package contains the routines used for parallelization

- MPI_Init is called when mpi4py is imported
- MPI_Finalize is called when script exits

Running mpi4py on Quest

- Load python module

> module load python

- Run the MPI code

> mpirun -np 2 python <script.py>

Hello World

```
#!/usr/bin/env python
"""
Hello World, serial
"""

import sys

sys.stdout.write("Hello, World!"+"\n")
```

MPI parallelized

```
#!/usr/bin/env python
"""
Hello World, parallel
"""

from mpi4py import MPI
import sys

sys.stdout.write("Hello, World!"+"\n")
```

A Simple Python MPI Code

```
#!/usr/bin/env python
"""
Hello World, parallel
"""

from mpi4py import MPI
import sys

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

sys.stdout.write(
    "Hello, World! I am process %d of %d on %s.\n"
    % (rank, size, name))
```

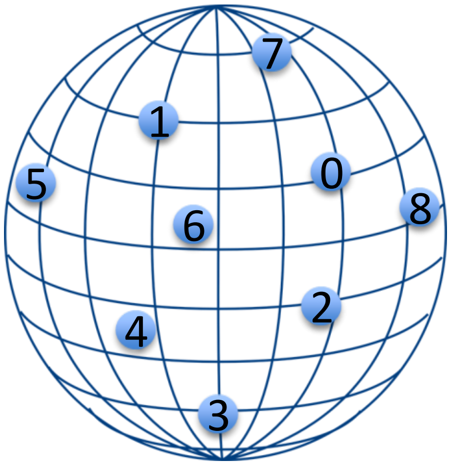
Minimal or no message communication: Embarrassingly parallel

MPI Environment

Python	comm=MPI.COMM_WORLD

- Communicator: a structure in which we identify a group of processes

- COMM_WORLD: constant which includes the whole activated communicator processes in an instance. It is a standard communicator object of MPI module in mpi4y



MPI.COMM_WORLD

MPI Environment

Python	MPI.COMM_WORLD.Get_size()

- Size: the number of processes in a communicator

Python	MPI.COMM_WORLD.Get_rank()

- Rank: Within the communicator, each processes is assigned to an integer from 0 to size-1

MPI Communication

- As understood from the name, MPI processes communicate by passing messages

Point-to-Point Communication

MPI Communication

Message
(data+envelope)

- A message contains the data and its envelope

DATA	ENVELOPE
Buffer, initial address	Communicator
Count	Source
Data type	Destination
	Tag

MPI Communication

Point-to-Point Communication

Message is passed from one process to another

- Blocking
(Comm.send, Comm.receive)
- Non-blocking
(Comm.isend, Comm.irecv)

Collective Communication

Message passes to all processes in a communicator

Point-to-Point Communication

- Blocking versus non-blocking:
  - Blocking routines do not return until it is safe to use the routine's buffer (i.e. variables)
    - safe: the buffer has been copied to system or receiver buffer
  - Non-blocking routines do not wait for communication to complete
  - Non-blocking routines allow the overlap of computation and communication in order to gain performance

Point-to-Point Communication

- Usual communication methods are included in Comm class of MPI module

- Comm.send(buf,dest,tag)

Data Envelope

Variable	Definition
buf	Address of send buffer
dest	Rank of destination
tag	Message tag

Point-to-Point Communication

- Comm.recv(buf,source,tag,status)

Data Envelope

Variable	Definition
buf	Address of receive buffer
source	Rank of source
tag	Message tag
status	Status object (sender rank, tag, length)

Point-to-Point Communication

- Deadlock: Message passing cannot be completed.

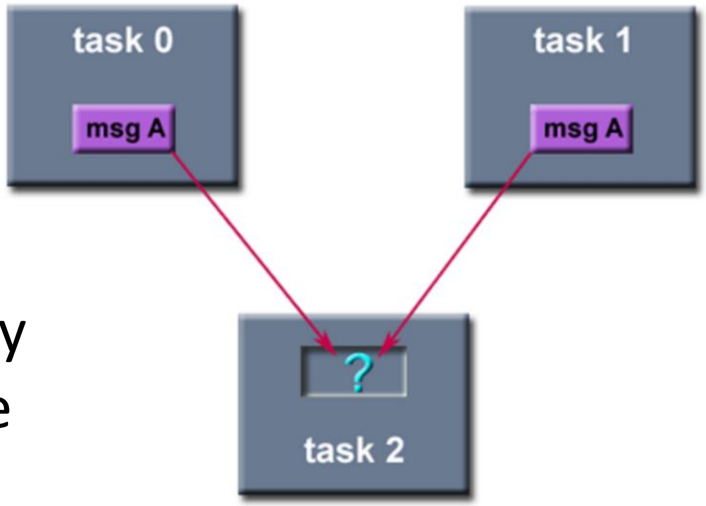
```
if rank == 0:
   comm.send(..., dest=1, …)
   comm.recv(..., source=1, …)
elif rank == 1 :
   comm.send(..., dest=0, …)
   comm.recv(..., source=0, …)
```

```
if rank == 0:
   comm.send(..., dest=1, …)
   comm.recv(..., source=1, …)
elif rank == 1:
   comm.recv(..., dest=0, …)
   comm.send(..., source=0, …)
```

Point-to-Point Communication

- Order and Fairness
  - MPI keeps the order of send (or receive) requests from the same task
  - MPI does not guarantee fairness

If 2 different tasks send messages that match another tasks receive, only one send will be complete



Communicating Objects & Arrays

- mpi4py provides two sets of functions for communication:
  - All lower case methods (send, recv etc.) are for communicating generic python data objects. Can be slow.
  - Upper-case initial letter case (Send, Recv etc.) are for communicating array data (such as NumPy arrays) which occupies continuous memory blocks. Fast communication.

Point-to-Point Communication

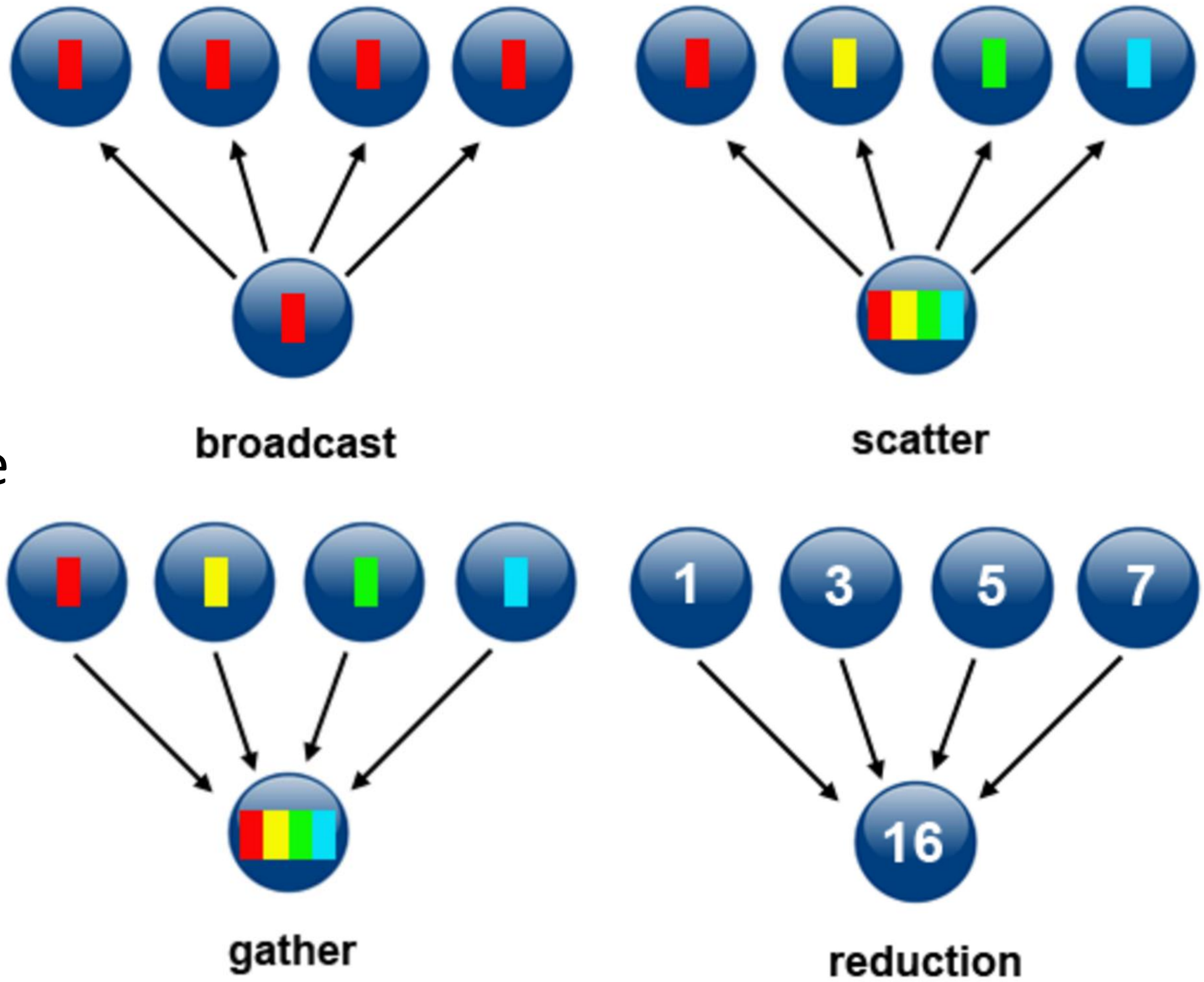
```
import numpy as np
from mpi4py import MPI

comm = MPI.COMM_WORLD
message = np.arange(1)
comm.Barrier()
status = MPI.Status()

size = comm.size
rank = comm.rank

if rank == 0:
        message[0] = 48151623
        print 'process ', rank, ' sends ' , message
        comm.Send(message, dest=1)
        print ' '
elif rank == 1:
        comm.Recv(message, source=0,status=status)
        print ' '
        print 'process ', rank, ' receives ', message
```

A blocking communication between 2 processes for array data

Collective Communication

<u>Types of Collective Communications</u>

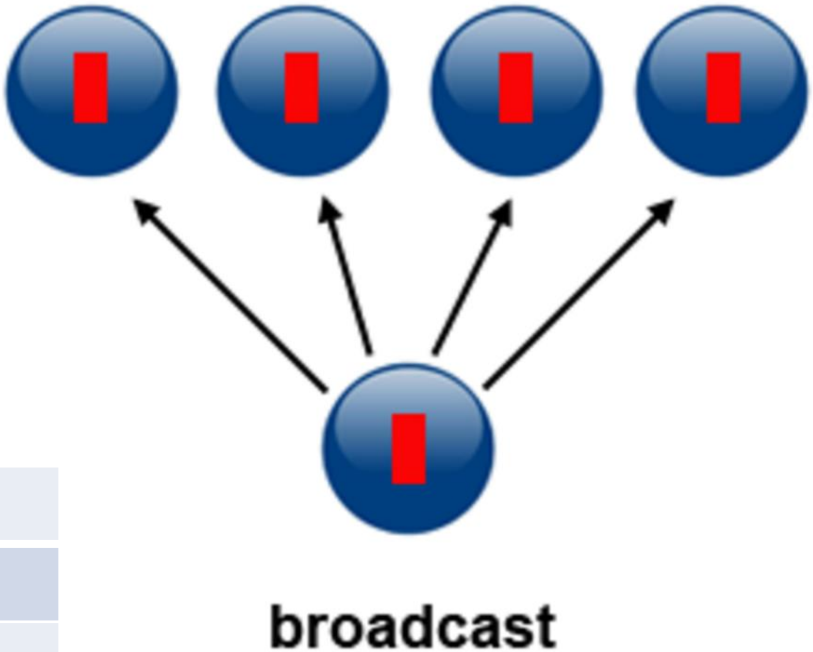
- Synchronization
- Data Movement
- Collective Compute



Collective Communication

Comm.Bcast (buf, root)

Broadcasts a message from one process to members in a communicator

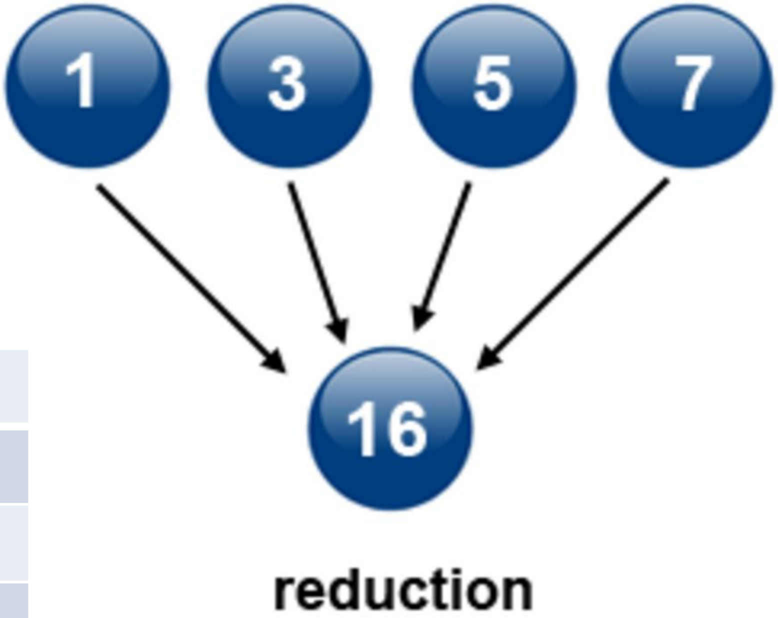


Variable		Definition
buf		Address of input buffer
root		Process id of root process

Collective Communication

Comm.Reduce (sendbuf,recbuf,op,root)

Performs a reduction operation to the vector of elements in the sendbuf of the group members and places the result in recvbuf on root



Variable		**Definition**
sendbuf		Address of send buffer
recvbuf		Address of receive buffer
op		Operation (+,-,x,max,min, …)
root		Process id of root process

Collective Communication

Comm.barrier Comm.Barrier	All processes within a communicator will be blocked until all processes within the communicator have entered the call.
Comm.bcast Comm.Bcast	Broadcasts a message from one process to members in a communicator.
Comm.reduce Comm.Reduce	Performs a reduction operation to the vector of elements in the sendbuf of the group members and places the result in recvbuf on root.
Comm.gather Comm.Gather	Collects data from the sendbuf of all processes in comm and place them consecutively to the recvbuf on root based on their process rank.
Comm.scatter Comm.Scatter	Distribute data in sendbuf on root to recvbuf on all processes in comm.
Comm.allreduce Comm.Allreduce	Same as MPI_REDUCE, except the result is placed in recvbuf on all members in a communicator.
Comm.allgather Comm.Allgather	Same as GATHER/GATHERV, except now data are placed in recvbuf on all processes in comm.
Comm.alltoall Comm.Alltoall	The j-th block of the sendbuf at process i is send to process j and placed in the i-th block of the recvbuf of process j.

References

- https://computing.llnl.gov/tutorials/mpi/
- http://sc.tamu.edu/shortcourses/SC-MPI/mpi_shortcourse_v4.pdf
- http://mpi4py.scipy.org/docs/usrman/tutorial.html
- https://mpi4py.scipy.org/docs/apiref/mpi4py.MPI-module.html
- https://wiki.python.org/moin/ParallelProcessing