Tutorial

Introduction to MPI on python using mpi4py

SciPyLA 2016

https://github.com/jeudy/tutorial_mpi4py

Tutorial requisites

- Python 2.7+
- Numpy
  - pip install numpy
- An implementation of MPI (openmpi or mpich)
  - sudo apt-get install openmpi-bin openmpi-common libopenmpi-dev
- Mpi4py
  - pip install mpi4py

http://swatandnurv.blogspot.com/2013/07/install-mpi4py-on-ubuntu.html

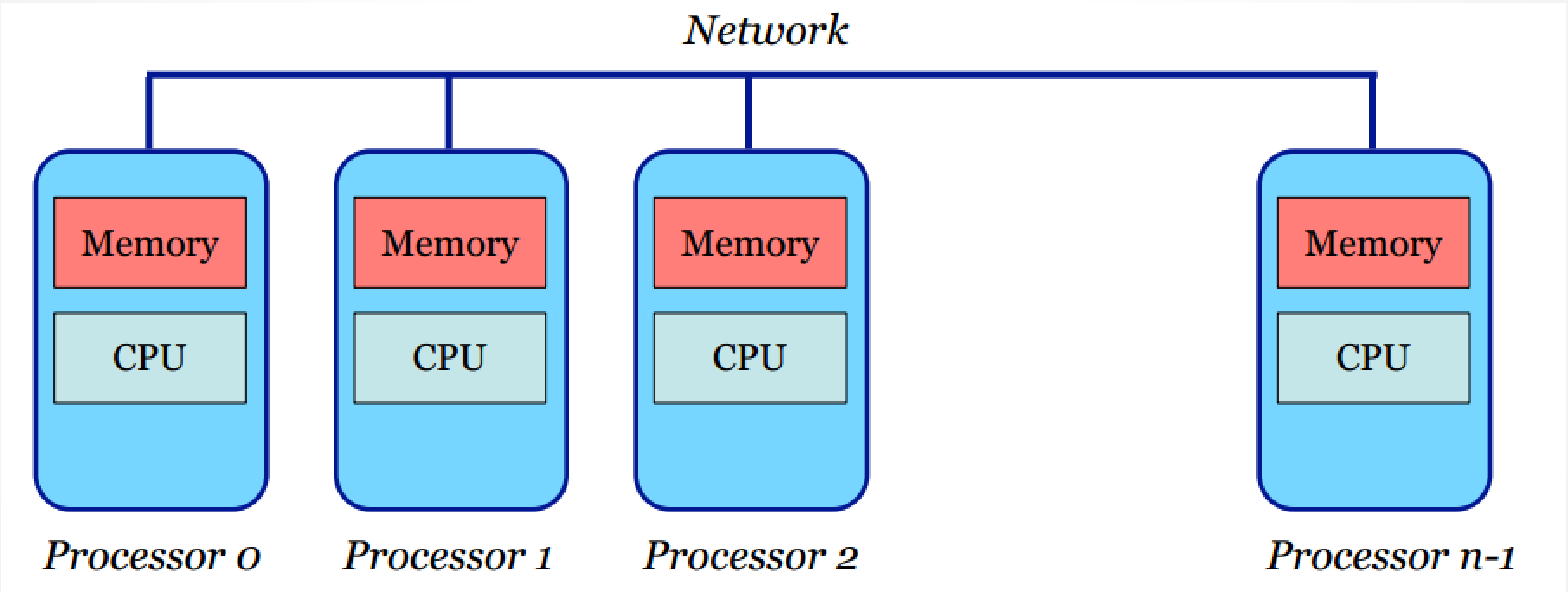
Parallel programming

- Distribute a program's work among several execution threads or processors.
- Today's computers come with multiple processors/cores.
- A typical program executes its instructions sequentially using a single processor
- Each processor has its own private memory space.

Parallel programming

- **OpenMP**: Takes advantage of the multi core architecture capable of executing several threads in parallel: **Shared memory**.
- **MPI**: several computers connected using the network forming a distributed system (**distributed memory**), potentially having thousands of processors available for processing.

Beowulf Cluster



MPI

- Message Passing Interface
- Designed in 1992 in a supercomputing conference to decide a standard.
- Its a model where a program passes messages between processes to execute a task.
- Example: a central process assigns work to "slave" processes, transmitting the data to use.

MPI

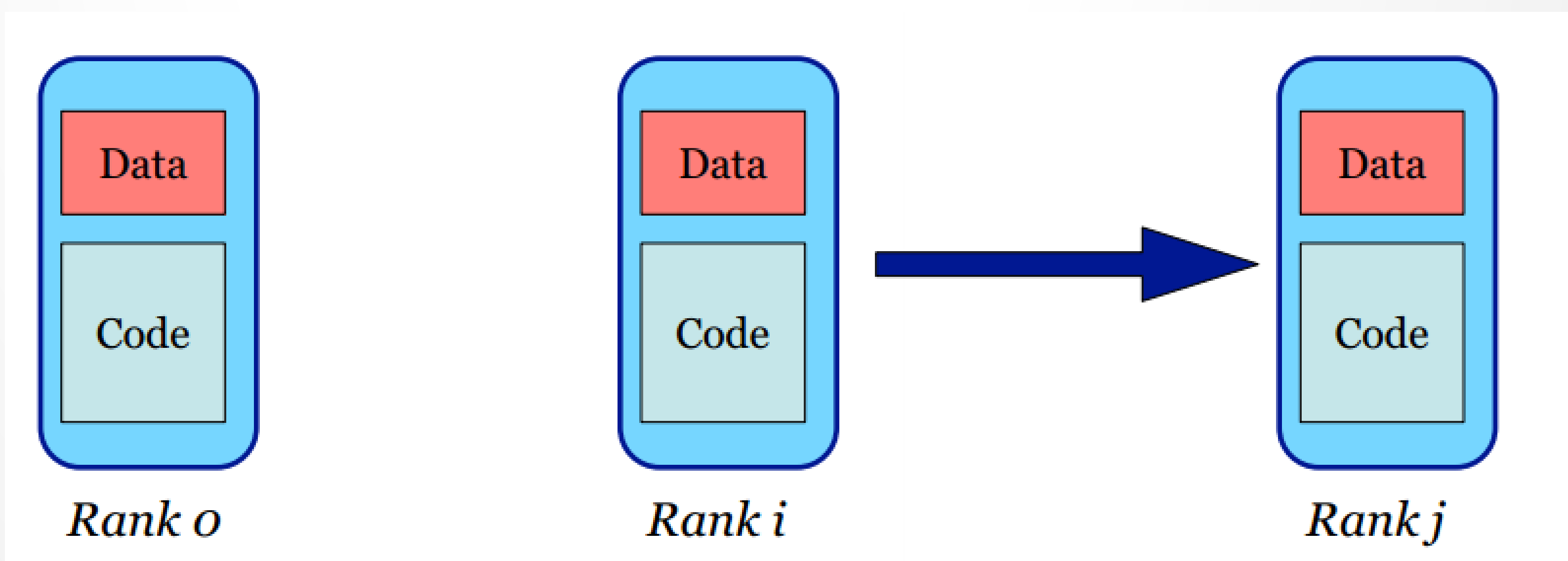
- Its a **library** that has special functions to allow communication and synchronization between processes.
- Implemented in many programming languages. Has wrappers for C and Fortran compilers (mpicc and mpif90).

MPI

- Open-source implementations:
  - Open Mpi: http://www.open-mpi.org/
  - Mpich: http://www.mpich.org/
  - **Mpi4py**: uses some of the previous implementations, from python, without any additional compilers (http://mpi4py.readthedocs.io/).
- MPI uses a special launcher to run programs: mpirun:
  - **mpirun** -np 10 program.exe

MPI Concepts

- **Communicator**: group of processes that can communicate to each other.
- **Rank**: numeric identifier for each process within a communicator. Values usually go from 0 to N-1



Mpi4py

- Created by Lisandro Dalcin.
- Uses an object oriented approach, keeping the standard semantics of MPI.
- Can communicate any native python object, or user defined objects using the **pickle** module for serialization.
- To run a parallel python program:

```
mpirun python -np 10 myprogram.py
```

Initialization

- In general, MPI programs need to initialize and release the environment through function calls: Init and Finalize.
- In mpi4py case, the Init call is not critical. The environment is automatically initialized when the library is imported.

```
import mpi4py.MPI as MPI

if not MPI.Is_initialized():
    MPI.Init()

MPI.Finalize()
```

MPI environment

- From the program, you can access the Communicator information, and each process can get its **rank**.
  - COMM_WORLD represents the Communicator
  - .Get_size()
  - .Get_rank()

MPI's Hello World

```python
"""
Simplest mpi program
To execute this run: mpirun -np 4 python hello.py
"""

import mpi4py.MPI as MPI

if not MPI.Is_initialized():
    MPI.Init()

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

print "I am process {id}. Total size: {size}\n".format(id=myid, size=size)

MPI.Finalize()
```

hello.py

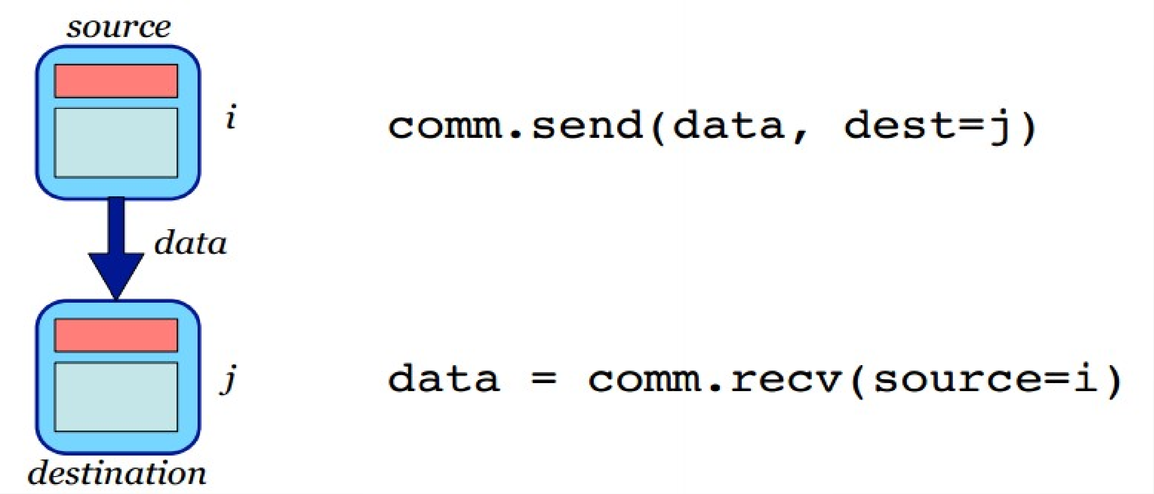
MPI's Hello World

- mpirun -np 10 python hello.py

```
I am process 5. Total size: 10
I am process 0. Total size: 10

I am process 2. Total size: 10

I am process 3. Total size: 10

I am process 6. Total size: 10

I am process 7. Total size: 10


I am process 8. Total size: 10
I am process 4. Total size: 10


I am process 1. Total size: 10
I am process 9. Total size: 10
```

Point to Point communication

- A process can send messages to another process directly, using its rank as id.
- Two operations are defined: **send** and **receive**.
- May be synchronous or asynchronous.



Point to Point communication

- **send** y **recv** transmit native or user created python objects using pickle. Synchronous (blocking).
- isend and irecv are asynchronous. They return *Request* objects with methods test() and wait()
- **Send** y **Recv** transmit buffers (i.e: numpy arrays)
- Same pattern on most functions (lowercase for *python objects*, uppercase for *buffers*).

Synchronous send / recv sample

- send_recv_object.py
- send_recv_buffer.py

Asynchronous send / recv sample

- send_recv_async.py
- send_recv_async_wait.py

Example: Ping Pong

- A message is exchanged between 2 processes, 10 times, carrying a counter.
- The process with rank 1 increments the counter when it receives the message.

mpirun -np 2 python pingpong.py

Collective communication

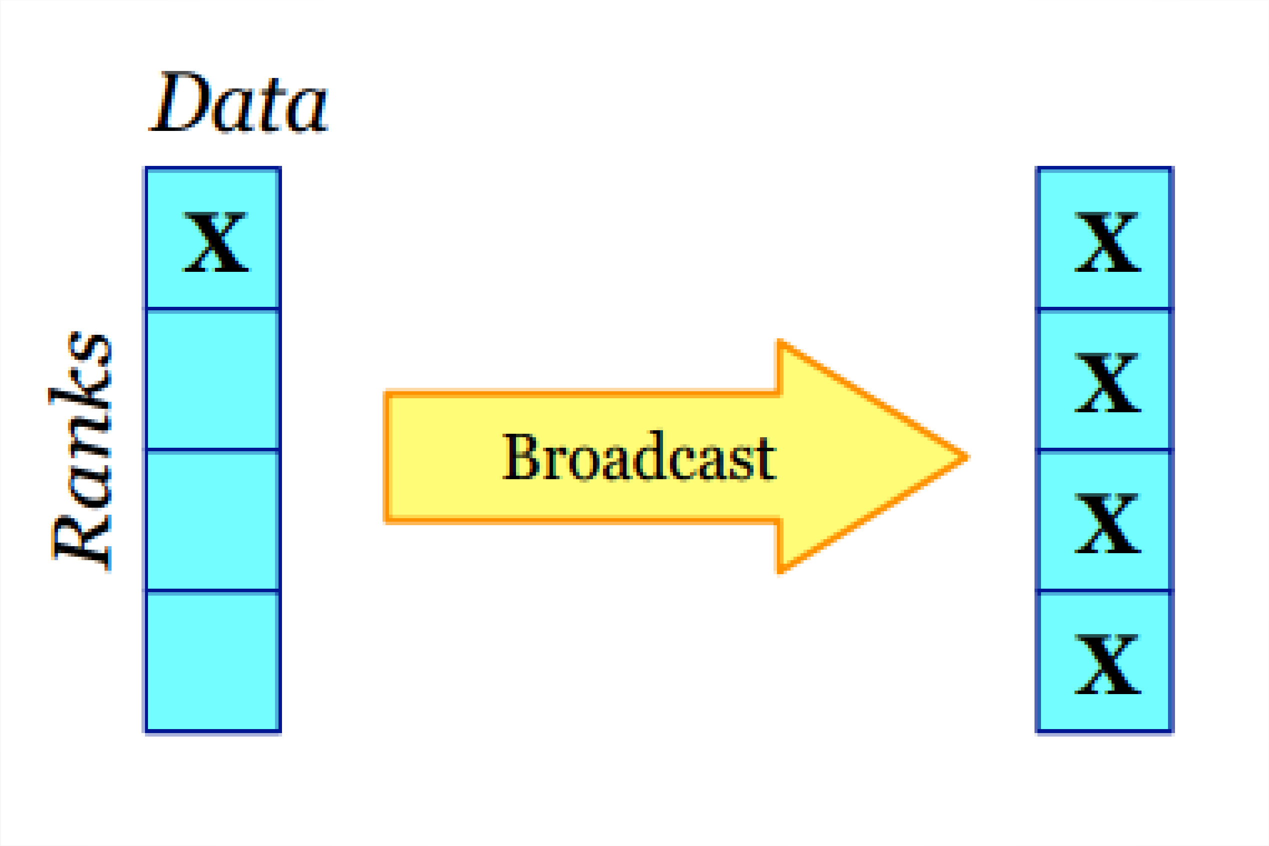
- Sometimes it is convenient to share data with all the processes in the Communicator, coordinated by a root process.
- There are several collective communication operations:
  - Broadcast
  - Reduce
  - Scatter
  - Gather

SciPyLA 2016

Broadcast

- One to Many.
- A root process transmit data to all other processes.
- In the function call, the rank of the root node is specified.
- The root node transmit the data, all others receive it.

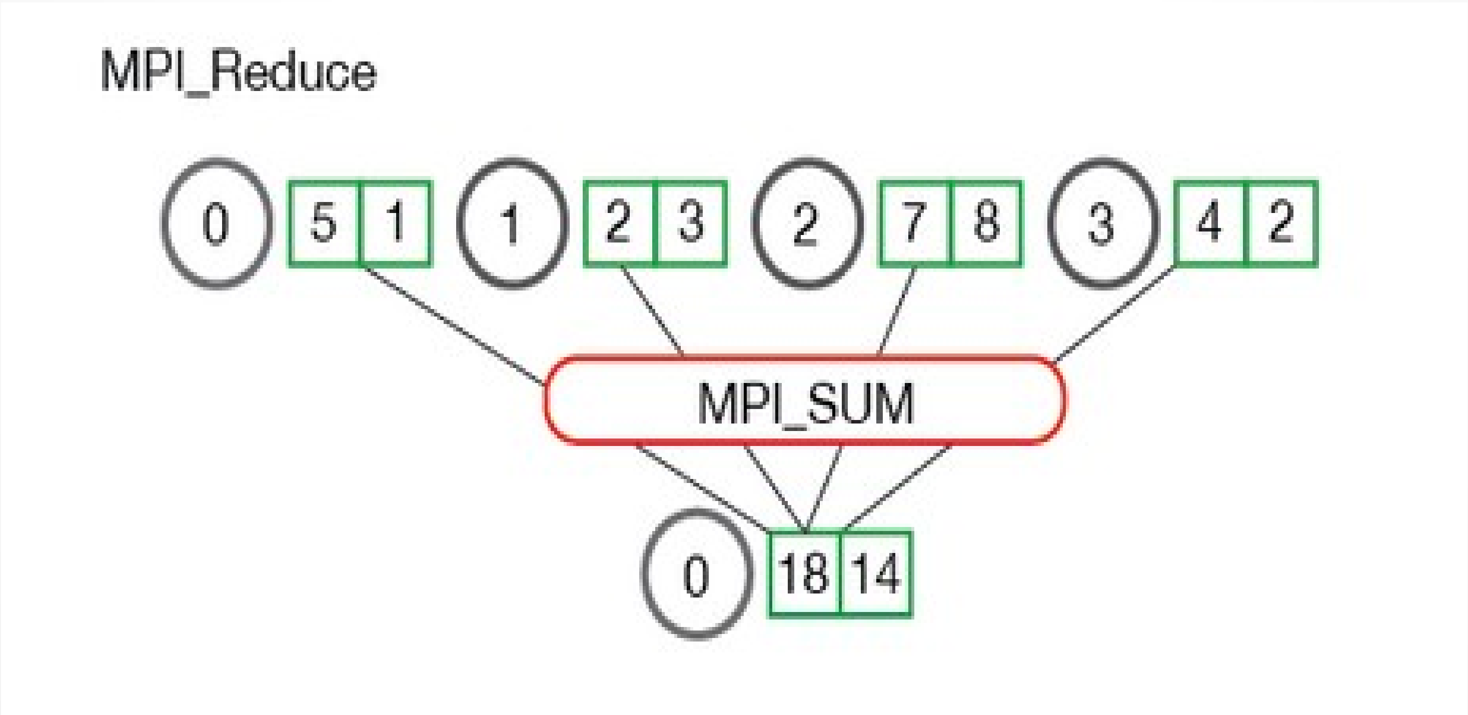
Broadcast



broadcast_object.py – broadcast_buffer.py

Reduction

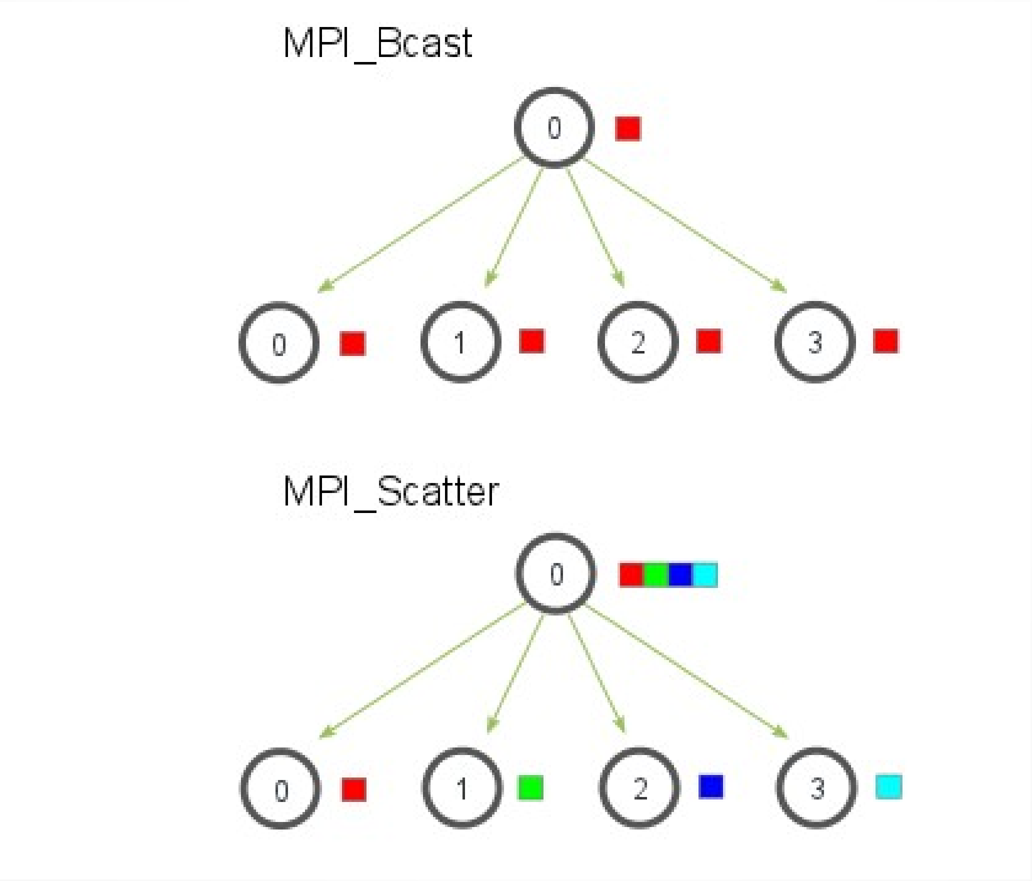
- Many to One.
- A mathematical operation is applied on the data in the root node indicated in the function call.
- Supported operations: Max, Min, Sum, Prod



reduction.py

Scatter

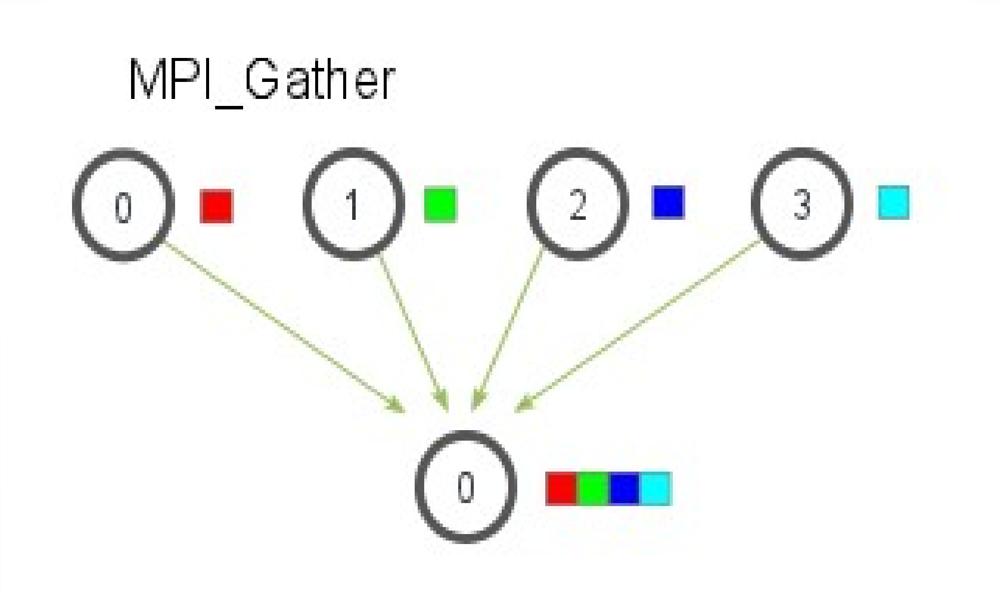
- Similar to Broadcast, but instead of sending the same data to all the processes, it distributes them.



scatter_objects.py – scatter_buffer.py

Gather

- Opposite to scatter. Each node sends a piece of data, and all pieces are gathered in the root process.



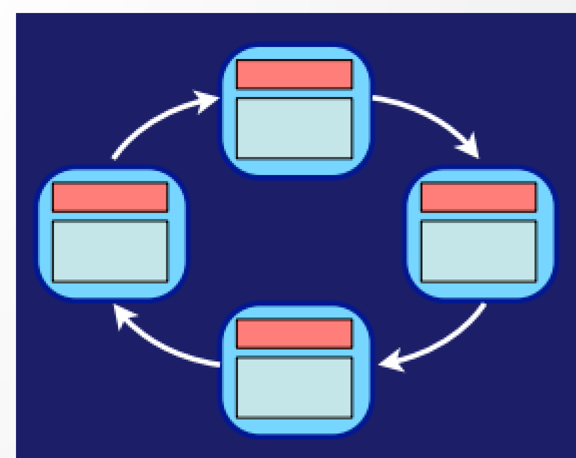
gather_object.py – gather_buffer.py

Exercises!

Exercise: Ring

- We'll have a ring of processes. Each process will send a message with its rank to the next. Each process will print the message it receives.
- mpirun -np 5 python ring.py

```
I am process 3 an received message from [2]
I am process 4 an received message from [3]
I am process 0 an received message from [4]
I am process 1 an received message from [0]
I am process 2 an received message from [1]
```



Exercise: Distributed sum

- In the root process, create an array of 10 random integers between 1 and 100 (numpy.random.random_integers)
- Distribute the array among all processes as evenly as possible.
- Each node will sum the values it got (partial sum).
- Transmit the partial sum from each node to the root using Reduce.
- Sum all partial sums on the root node

Exercise: Distributed sum

```
Data in root is:  [62 44 68 60  1 70 28 53 60 29]
I am process 0 and my range goes from 0 to 2. My data: [62 44]. \ Partial sum: 106
I am process 1 and my range goes from 2 to 4. My data: [68 60]. \ Partial sum: 128
I am process 2 and my range goes from 4 to 6. My data: [ 1 70]. \ Partial sum: 71
I am process 3 and my range goes from 6 to 10. My data: [28 53 60 29]. \ Partial sum: 170
Reduced sum is: 475. From original data: 475
```

Exercise: Parallel pi

- You are given a pi_serial.py script that calculates the number pi using the Wallis product:

$$\prod_{n=1}^{\infty}\left(\frac{2n}{2n-1}\cdot\frac{2n}{2n+1}\right)=\frac{2}{1}\cdot\frac{2}{3}\cdot\frac{4}{3}\cdot\frac{4}{5}\cdot\frac{6}{5}\cdot\frac{6}{7}\cdot\frac{8}{7}\cdot\frac{8}{9}\cdots=\frac{\pi}{2}$$

Exercise: Parallel pi

- Make a parallel version, **pi_mpi.py** that uses MPI to distribute the processing load among different processes.
- Observe the processor load while running each script (the serial and the parallel version).
- Run some benchmarks to estimate the performance enhancement.

References

- http://mpitutorial.com/beginner-mpi-tutorial/
- http://coco.sam.pitt.edu/~emeneses/teaching/mpi
- http://mpi4py.readthedocs.io/