15-440
Distributed Systems
Recitation 9

Slides By: Hend Gedawy

& Previous TAs

Announcements

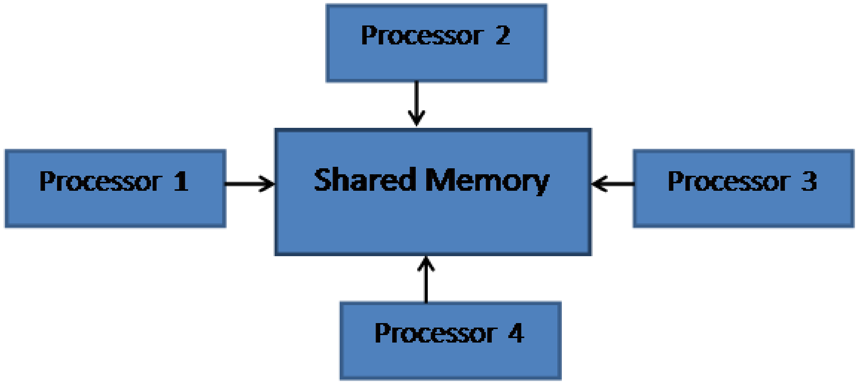
- **P3** Out (Due Nov. 16)

Outline

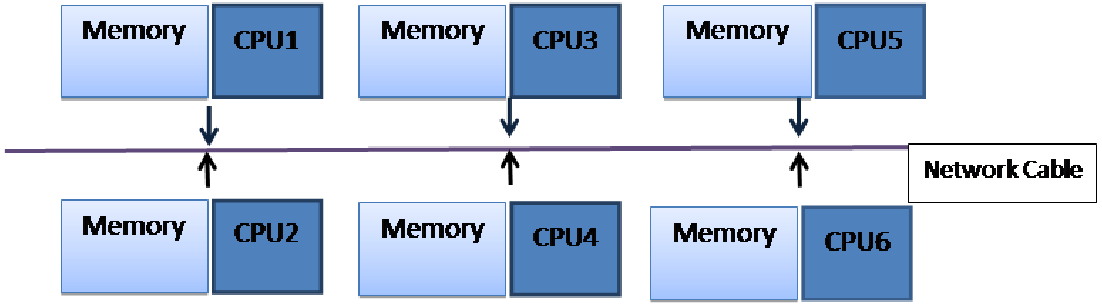
- **Parallel Programming Models**
- MPI Skeleton & Concepts
- Mpi4py Initialization & Insights
- Mpi4py Point-to-Point Communication
- Mpi4py Collective Communication
- Setting up & Running MPI on your Cluster

Parallel Programming Models

- Shared Memory Model



- Message Passing Model



Parallel Programming Models

Shared Memory	Message Passing

Parallel Programming Models

Shared Memory	**Message Passing**
Communicating processes usually reside on the same machine	Typically used in a distributed environment where communicating processes reside on remote machines connected through a network.
Faster communication strategy.	Relatively slower communication strategy
More difficult to synchronize	Easier to synchronize
Example: OpenMP	Example: MPI

Outline

- Parallel Programming Models
- **MPI Skeleton & Concepts**
- Mpi4py Initialization & Insights
- Mpi4py Point-to-Point Communication
- Mpi4py Collective Communication
- Setting up & Running MPI on your Cluster

What is MPI?

- Message Passing Interface
- Defines a set of API declarations on message passing (such as send, receive, broadcast, etc.), and what behavior should be expected from the implementations.
- The *de-facto* method of writing message-passing applications
- Applications can be written in C, Python and calls to MPI can be added where required

MPI Program Skeleton

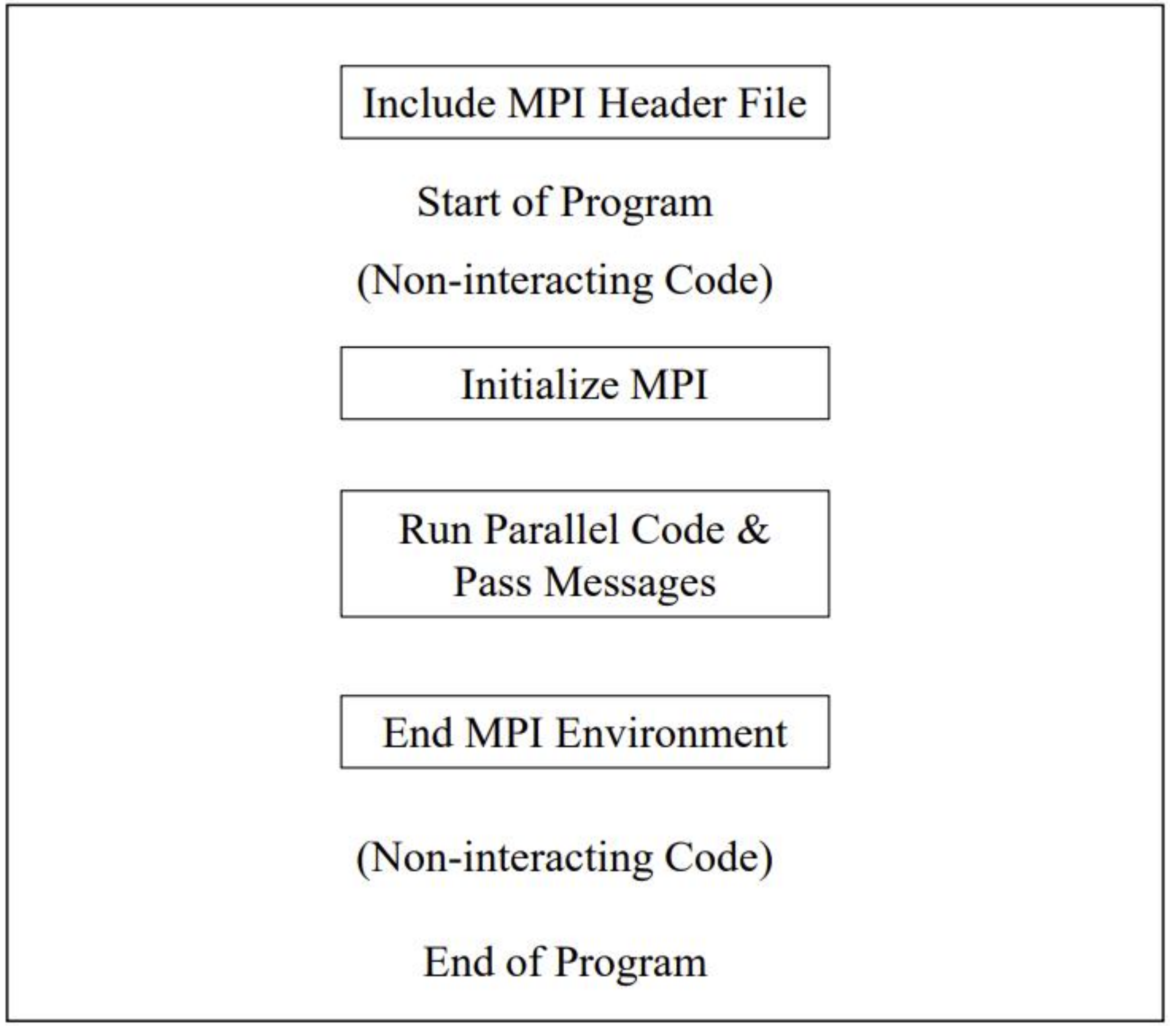
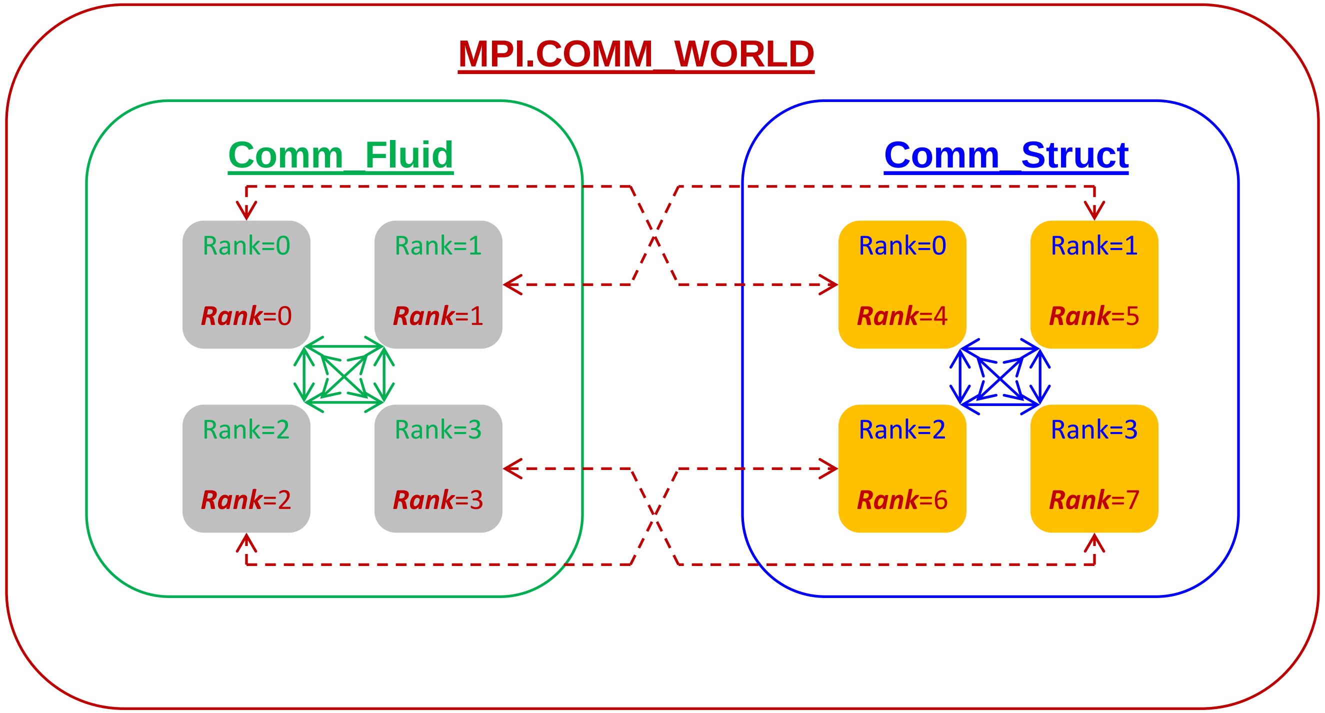


Photo credits:
https://princetonuniversity.github.io/PUbootcamp/sessions/parallel-programming/Intro_PP_bootcamp_2018.pdf

MPI Concepts

- *Communicator*
  - Defines which *collection of processes* may communicate with each other to solve a certain problem
  - In this collection, each process is assigned a unique *rank*, and they explicitly communicate with one another by their ranks.
  - When an MPI application starts, it automatically creates a communicator comprising all processes and names it MPI.COMM_WORLD
    - This is the biggest communicator your program has
    - Sub communicators can be created to tackle sub problems
- *Rank*
  - Within a communicator, every process has its own unique ID referred to as *rank*
  - *Root or master machine will have rank 0*
    - *It usually splits/distributes the work and reduces or gathers partial results*
  - Ranks are used by the programmer to specify the source and destination of messages

MPI Concepts – Local and Global Ranks



✓Ranks within MPI.COMM_WORLD are printed in red
✓Ranks within Comm_Fluid are printed in green
✓Ranks within Comm_Struct are printed in blue

Carnegie Mellon University Qatar
جامعة كارنيجي ميلون في قطر

Outline

- Parallel Programming Models
- MPI Skeleton & Concepts
- **Mpi4py Initialization & Insights**
- Mpi4py Point-to-Point Communication
- Mpi4py Collective Communication
- Setting up & Running MPI on your Cluster

Mpi4Py - Initialization

- MPI for Python (Mpi4py) library provides Python bindings for the Message Passing Interface (MPI) standard.
- Importing the library
  - from mpi4py import MPI
  - Will take care of initialization of MPI library (Unlike in C will have to do it explicitly)
- MPI_Finalize() is called when all python processes exit
- Initializing the main parallel workflow variables
  - **comm** = MPI.COMM_WOLD
  - **myrank**= comm.Get_rank()
  - **nproc**= comm.Get_size()

Mpi4py – Types of Communicated Objects

- Any kind of **generic** python **objects**
  - e.g. dictionaries, lists, …
  - Use lower case methods: **s**end, **r**ecv, **b**cast,….
  - Introduces Overhead: a binary representation of the message is created to send and restored after received
- Python **buffer-like objects** allocated in contagious memory
  - e.g. NumPy arrays, …
  - Use upper case analogues, **S**end, **R**ecv, **B**cast,…

Mpi4Py – Hello World

```
from mpi4py import MPI

if (__name__ == '__main__'):
    comm = MPI.COMM_WORLD
    myrank = comm.Get_rank()
    nproc = comm.Get_size()

    print("Hello, World ! from process {0} of {1} \n"
          .format(myrank, nproc))
```

To Run: mpiexec –np 4 python3 helloWorld.py
mpirun –np 4 python3 helloWorld.py

Outline

- Parallel Programming Models
- MPI Skeleton & Concepts
- Mpi4py Initialization & Insights
- **Mpi4py Point-to-Point Communication**
- Mpi4py Collective Communication
- Setting up & Running MPI on your Cluster

MPI Point-Point Send and Recv

Blocking Communication:

- Sending:
  - Generic Objects: comm.**send**(sendobj, dest=1, tag=0)
  - Numpy Buffer: comm.**Send**([sendarray, count, datatype], dest=1, tag=0)
- Receiving:
  - Generic Objects: **recvobj** = comm.**recv**(src=0, tag=0)
  - Numpy Buffer: comm.**Recv**([recvarray, count, datatype], src=0, tag=0)

Non-Blocking Communication:

- Sending:
  - Generic Objects: **reqs** = comm.**isend**(object, dest=1, tag=0)
  - Numpy Buffer: **reqs** = comm.**Isend**([sendarray, count, datatype], dest=1, tag=0)
  - reqs.**wait**()
- Receiving:
  - Generic Objects: **reqr** = comm.**irecv**(src=0, tag=0)
  - NumpyBuffer: **reqr** = comm.**Irecv**([recvarray, count, datatype], src=0, tag=0)
  - **data =** reqr.**wait**()
- MPI.Request.**Waitall**([reqs, reqr])

Parameters:

- sendarray/recvarray is the data buffer
- count and datatype of elements that reside in the buffer
- dest /src specify the rank of the sending/receiving process
- tag of the message (optional)
- reqs/reqr are request objects

Why do we need a tag?

Point to Point Communication Example-Generic Object

```
from mpi4py import MPI

if (__name__ == '__main__'):
    comm = MPI.COMM_WORLD
    myrank = comm.Get_rank()
    nproc = comm.Get_size()

    if (myrank == 0):
        a = {"Day": "Monday", "Age": 20, "z": [90, 3, 1]}
        for i in range(1, nproc):
            comm.send(a, dest=i, tag=7)
    else:
        a_recv = comm.recv(source=0, tag=7)
        print("I'm process {0} and received: {1}\n"
              .format(myrank, a_recv))
```

Point to Point Communication Example– Buffer Type Objects

```
from mpi4py import MPI
import numpy as np

if (__name__ == '__main__'):
    comm = MPI.COMM_WORLD
    myrank = comm.Get_rank()
    nproc = comm.Get_size()

    if (myrank == 0):
        a = np.arange(10, dtype='i')
        for i in range(1, nproc):
            comm.Send([a, 10, MPI.INT], dest=i, tag=7)
    else:
        my_a = np.zeros(10, dtype='i')
        comm.Recv([my_a, 10, MPI.INT], source=0, tag=7)
        print("I'm process {0} and received: {1}\n"
              .format(myrank, my_a))
```

Point to Point Communication – Sum of the first N integers

```
from mpi4py import MPI
import numpy as np

if (__name__ == '__main__'):
    comm = MPI.COMM_WORLD
    myrank = comm.Get_rank()
    nproc = comm.Get_size()
    N = 1000
    startval = int(N * myrank / nproc + 1)
    endval = int(N * (myrank+1) / nproc)
    partial_sum = np.array(0, dtype='i')

    for i in range(startval, endval+1):
        partial_sum += i
    if (myrank != 0):
        comm.Send([partial_sum, 1, MPI.INT], dest=0, tag=7)
    else:
        tmp_sum = np.array(0, dtype='i')
        for i in range(1, nproc):
            comm.Recv([tmp_sum, 1, MPI.INT], source=i, tag=7)
            partial_sum += tmp_sum
        print("The sum is {0}\n".format(partial_sum))
```

- Make each processor add up an interval of values from 0 to N
- Assign an interval to each processor based on its rank
- All processors will do a partial sum
- All except root, will send the result
- Root will add up the sums from all the processors

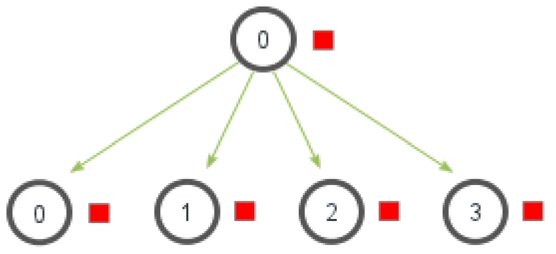
Outline

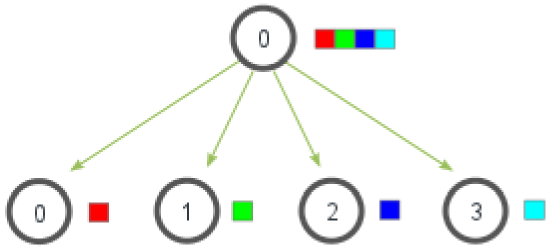
- Parallel Programming Models
- MPI Skeleton & Concepts
- Mpi4py Initialization & Insights
- Mpi4py Point-to-Point Communication
- **Mpi4py Collective Communication**
- Setting up & Running MPI on your Cluster

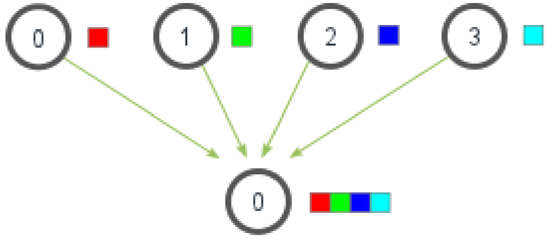
Collective Communication

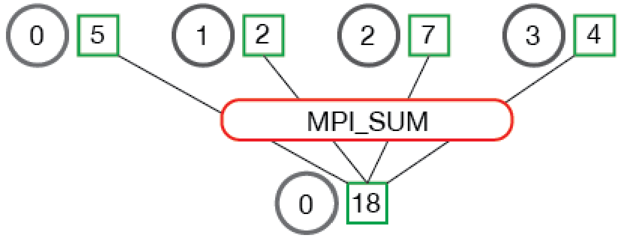
- Collective communication allows you to exchange data among a *group of processes*
- It must involve all processes in the scope of a communicator
- Hence, it is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operation

Patterns of Collective Communication





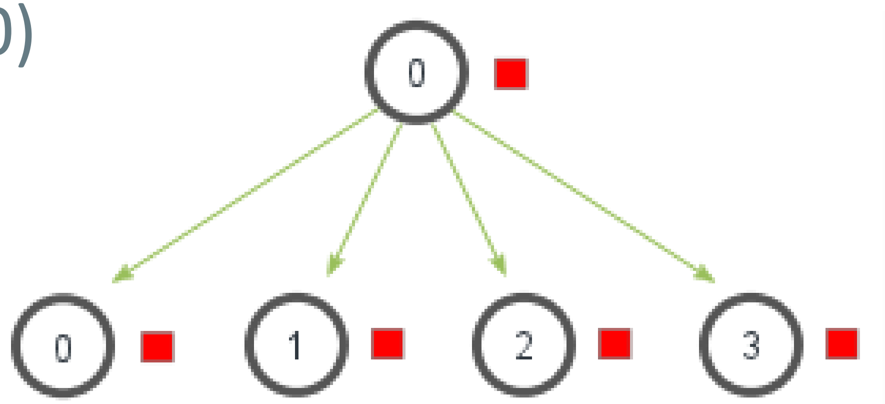




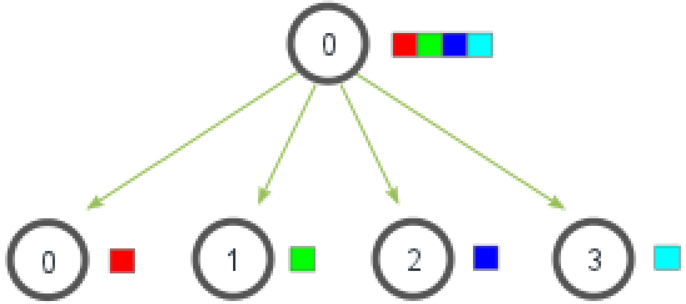
جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Patterns of Collective Communication - Broadcast

- Broadcasts a message from the process with rank *root to all other processes* of the group
- Generic Objects:
  - **recvobj** = comm.**bcast**(sendobj, root=0)
- Numpy Buffer:
  - comm.**Bcast**(buf, root=0)
  - buf = [**recvbuf**, count, datatype]

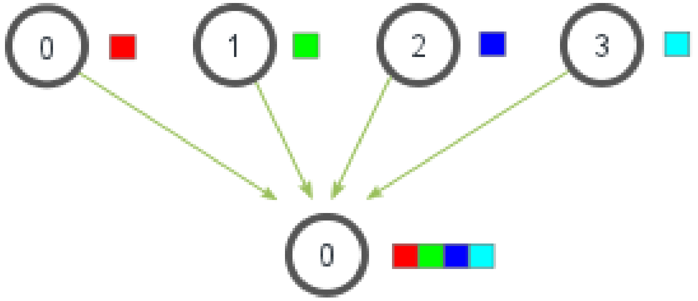


Patterns of Collective Communication - Scatter



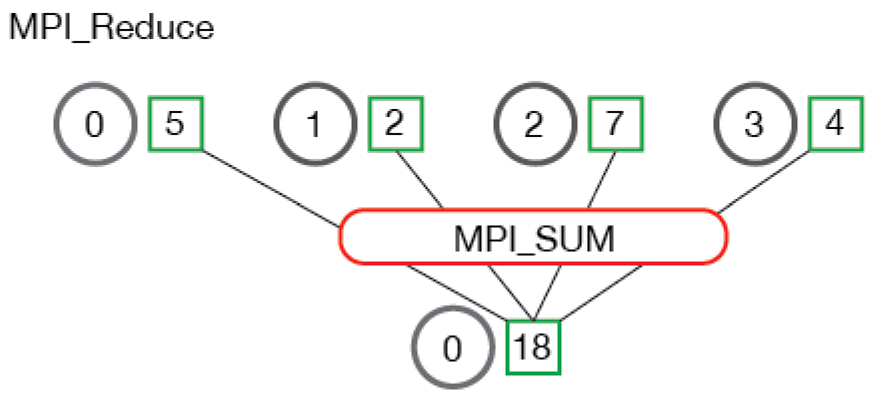
- Distributes elements of sendbuf to all processes in comm
- Generic Objects:
  - **recvobj** = comm.**scatter**(sendobj, root=0)
  - *sendObj: a single value or a list/tuple of size comm.size()*
  - *recvobj: a single value*
- Numpy Buffer:
  - comm.**Scatter**(sendbuf, **recvbuf**, root=0)
- Although the root process (sender) contains the entire data array, *Scatter* will copy the appropriate element into the recvbuf of the process.
- sendcount and recvcount are counts <u>per process</u>

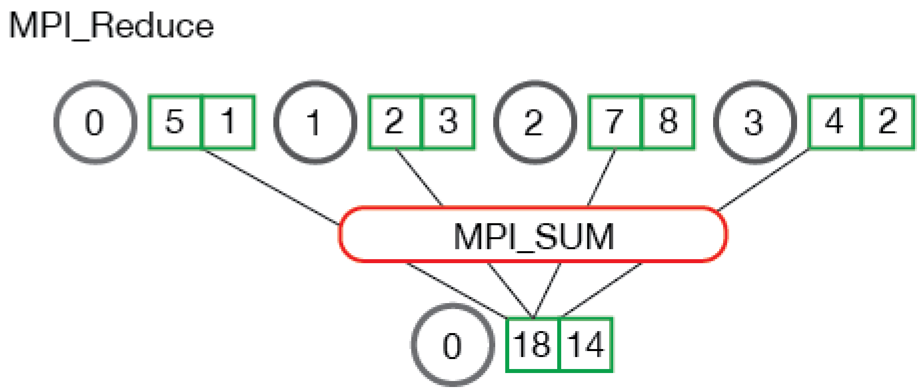
Patterns of Collective Communication - Gather



- Inverse of MPI_Scatter
- Generic Object:
  - **recvobj** = comm.**gather**(sendobj, root=0) #
  - recvObj: a list of size comm.size()
  - *sendObj: a single value or a list/tuple of size comm.size()*
- Numpy Buffer:
  - comm.**Gather**(sendbuf, **recvbuf**, root=0)
- Only the root process needs to have a valid receive buffer.
  - All other calling processes can pass NULL for recv_data

Patterns of Collective Communication - Reduce





Patterns of Collective Communication - Reduce

- Reduces values on all processes within a group.
- Generic Object:
  - **reducedobj** = comm.**reduce**(sendobj, op=MPI.OPERATION, root=0)
- Numpy Buffer:
  - comm.**Reduce**(sendbuf, **reducedbuf**, op=MPI.OPERATION, root=0)
- The sendbuf parameter is an array of elements of type datatype that each process wants to reduce.
- The reducedbuf is only relevant on the process with a rank of root.
- The reducedbuf array contains the reduced result.
- The op parameter is the operation that you wish to apply to your data.
- MPI contains a set of common reduction operations that can be used (SUM, MAX, MIN, ..)

Other Patterns of Collective Communication

*1. **Broadcast***

*2. **Scatter***

*3. **Gather***

*4. **Reduce***

*5. **Allgather:** Similar to Gather, but all processes receive result (not just the Root)*

*6. **Alltoall:** Sends data from all processes to all processes*

*7. **Allreduce:** Similar to Reduce, but the result appear in receive buffers of all processes (not just the root)*

*9. **Reducescatter:** Reduce followed by Scatter*

……

Collective communication – Scatter Generic Object Example

```
from mpi4py import MPI

if (__name__ == '__main__'):
    comm = MPI.COMM_WORLD
    myrank = comm.Get_rank()
    nproc = comm.Get_size()
    assert nproc == 3     #this basic example works only in 3 proc
    if myrank == 0:
        #object to scatter MUST be tuple or list of size comm.Get_size
        fulldata = [ 23, "AB", ["z", 22]]
        print("I'm {0} fulldata is: {1}".format(myrank,fulldata))
    else:
        fulldata = None     #all the procs must have a value for fulldata

    mydata = comm.scatter(fulldata, root=0)
    print("After Scatter, I'm {0} and mydata is: {1}".format(myrank,mydata))
```

Collective communication – Scatter Buffer-like Object Example

```
from mpi4py import MPI
import numpy as np

if (__name__ == '__main__'):
    comm = MPI.COMM_WORLD
    myrank = comm.Get_rank()
    nproc = comm.Get_size()
    assert nproc == 3
    if myrank == 0:
        fulldata = np.arange(9, dtype='i')
        print("I'm {0} fulldata is: {1}".format(myrank,fulldata))
    else:
        fulldata = None

    count = 3
    mydata = np.zeros(count, dtype='i')
    comm.Scatter([fulldata, count, MPI.INT],[mydata, count, MPI.INT],root=0)
    print("After Scatter, I'm {0} and mydata is: {1}".format(myrank,mydata))
```

Collective communication – Sum of the first N Integers Example

```
from mpi4py import MPI
import numpy as np

if (__name__ == '__main__'):
    comm = MPI.COMM_WORLD
    myrank = comm.Get_rank()
    nproc = comm.Get_size()
    N = 1000
    startval = int(N * myrank / nproc + 1)
    endval = int(N * (myrank+1) / nproc)
    partial_sum = np.array(0, dtype='i')
    for i in range(startval, endval+1):
        partial_sum += i

    tot_sum = np.array(0, dtype='i')
    comm.Reduce([partial_sum, 1, MPI.INT],
                [tot_sum, 1, MPI.INT], op=MPI.SUM, root=0)

    if (myrank == 0):
        print("The sum is {0}\n".format(tot_sum))
```

Outline

- Parallel Programming Models
- MPI Skeleton & Concepts
- Mpi4py Initialization & Insights
- Mpi4py Point-to-Point Communication
- Mpi4py Collective Communication
- Setting up & Running MPI on your Cluster

Setting up you cluster

- ssh to head node
  - 15440-<andrewID>-n01.qatar.cmu.edu
- ssh to all 3 other worker nodes (using machine names)
  - Make sure to accept keys the first time
  - Try to ssh again to make sure it is not asking for keys permission
- Create your machine file in the head node
  - This should have list of all machine names
  - Place it in the same folder as your code
- On all nodes, install the library by running:
  - pip install mpi4py

15440-<andrewID>-n01.qatar.cmu.edu
15440-<andrewID>-n02.qatar.cmu.edu
15440-<andrewID>-n03.qatar.cmu.edu
15440-<andrewID>-n04.qatar.cmu.edu

Running Mpi4py program on your cluster

- You write and run your code in the head node (n01)

- Run the command

MPI Parameters | **Your Program file and parameters**

```
mpirun -n 4 -machinefile machinesFile python3 collective_sumIntegers.py
```

  - -n: the number of machines that you will run the code on (4) for Project 3
  - -machinefile: the file that has the hostnames for the machines in your cluster

Credit

- https://indico.cism.ucl.ac.be/event/101/attachments/105/241/mpi4py2021.pdf
- http://ceciliajarne.web.unq.edu.ar/wp-content/uploads/sites/43/2019/06/talk_04.pdf
- https://cloudmesh.github.io/cloudmesh-mpi/report-mpi.pdf
- https://materials.jeremybejarano.com/MPIwithPython/overview.html