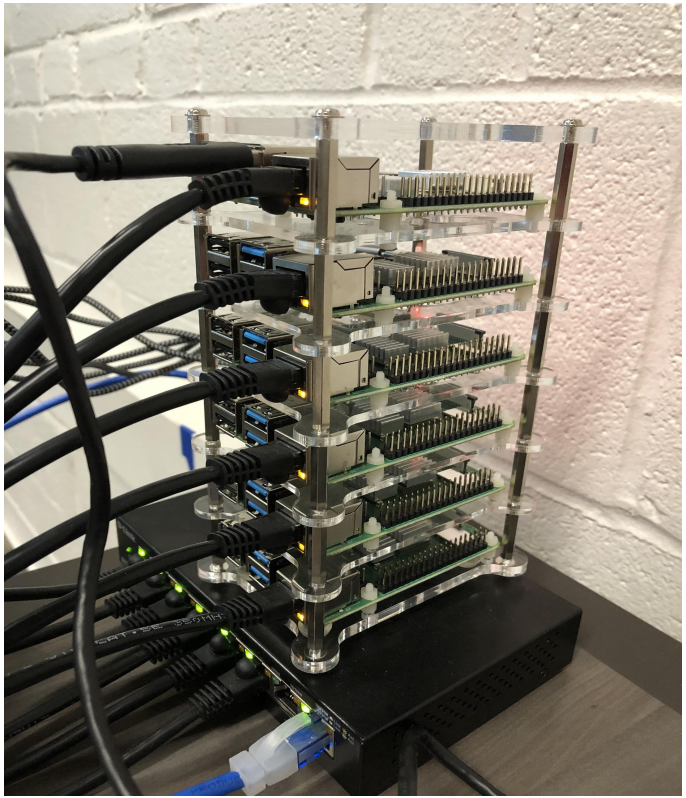
Distributed Computing with MPI4Py!

Advanced Topics 2021

Title slide by Rice Ewn

Building the Cluster



The cluster is a series of independent connected computers that can be viewed as a single system. Because there are many computers, a high number of parallelizable tasks can be distributed among them to optimize performance. They're all connected to a single network to allow for communication, and they can share a storage device and/or have local storage.

How is the cluster setup?

Nodes:

Physically a series of raspberry pis with one designated as the "head node" that communicates to the rest as "worker nodes."

To log on, there is an **account (ubuntu)** with **permissions (on every node)** using ssh **("secure shell" it's the connection to the server)** and public keys **("asymmetric encryption" instead of a password)**

MPI4Py

How does it work?

Program is sent to each individual node; ensured by the following line of code

"/usr/bin/parallel-rsync -h ~/workers -r /home/ubuntu/$1 /home/ubuntu"

mpi4py synchronizes the program across all nodes, not cluster0. It starts the command of the program to which the nodes carry out in parallel, denoted by "mpiexec"

Workflow

(push-run)> push-run student1 24 helloworld.py

echo -e "\nRunning job ~/$1/$2 with $3 processes\n"

/usr/bin/parallel-rsync -h ~/workers -r /home/ubuntu/$1 /home/ubuntu

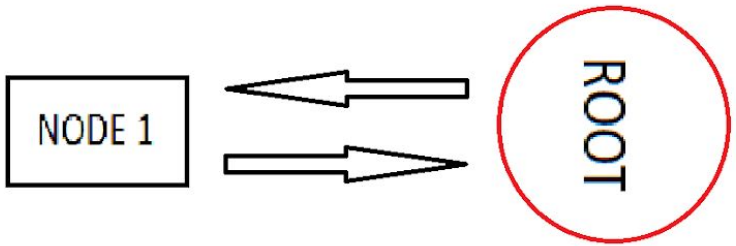
echo -e "\n\n\n"

/usr/bin/mpiexec --hostfile ~/mpi-hosts -n $3 **~/$1/$2 ← (file)**

Point to Point Communication

Point-to-Point Communication - What it is/How it works

- Direct communication from one node to another
- Can target a specific node and assign calculations directly to it



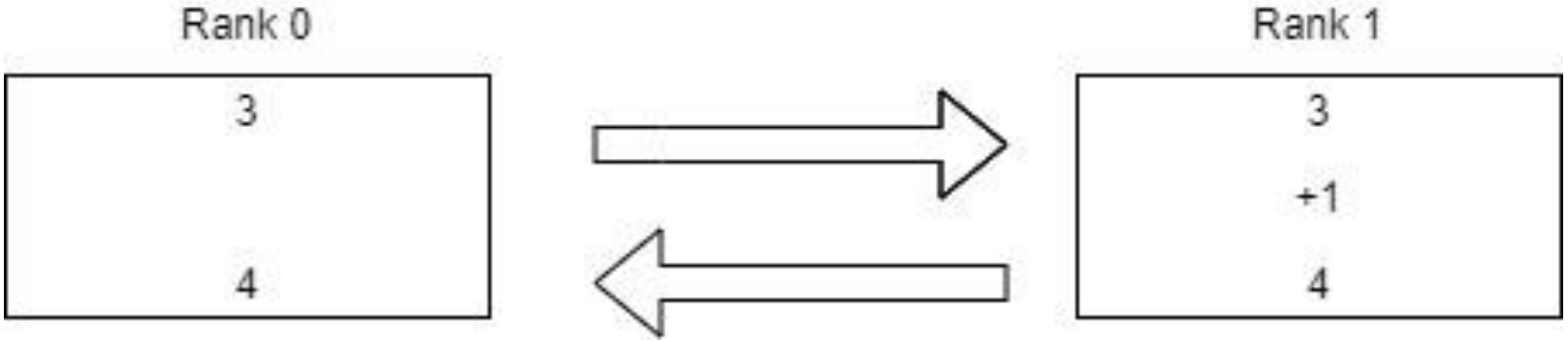
Point-to-Point Communication - Commands/Technicalities

send() and recv() vs isend() and irecv()

- Key Parameters:
  - object to be sent
  - dest/source int, which clarifies where the data goes
  - optional tag int marks specific uses of commands
- The i- is short for “instance”
- The default commands directly connect to each other
  - They will wait to finish before proceeding, which affects whether the script works
- The instance commands create a Request that only transfers data when it is resolved with a complementary command
  - This creates some room to modify behavior with Request commands
- Capitalized methods (Send() instead of send()) are for buffer objects, e.g. numpy arrays

Basic Point-to-point

1. 3 is sent from root to rank 1
2. Rank 1 receives 3 and ads one to it
3. Rank 1 sends back 1+3 (4) to the root
4. The root receives this data and prints it out

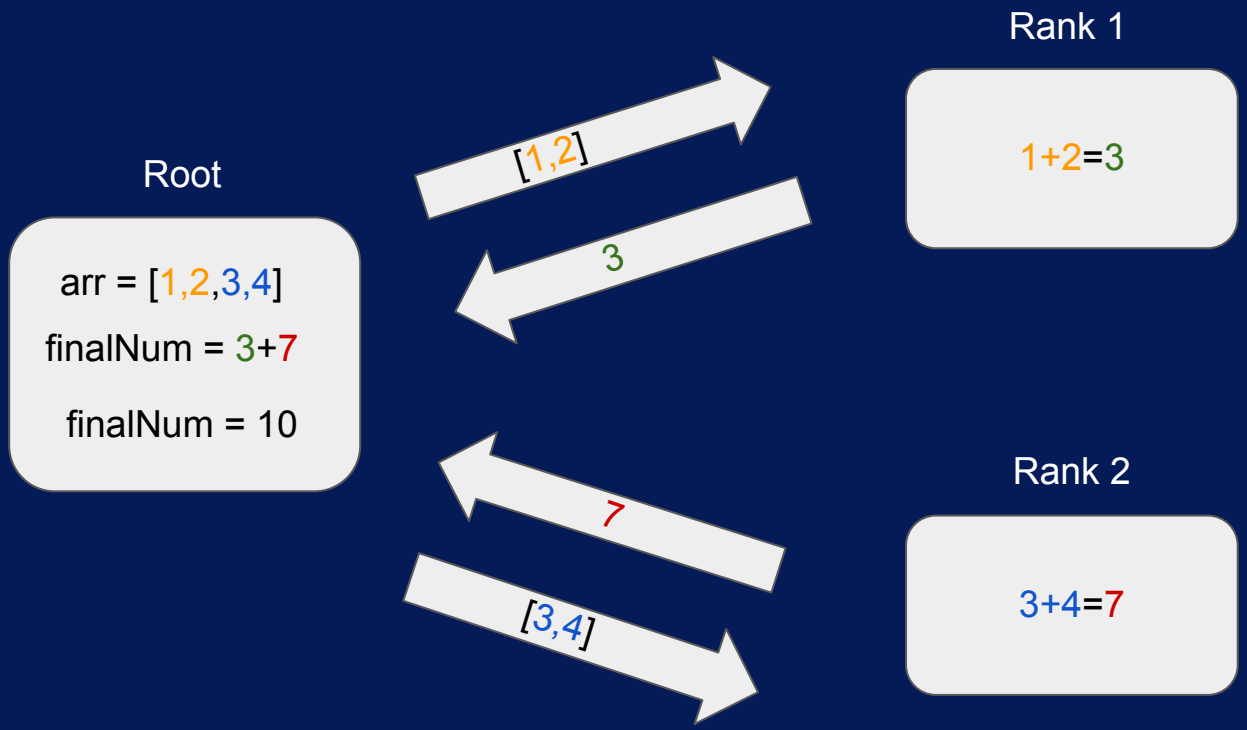


Basic Point-to-point Script

```
 1 #!/usr/bin/env python3
 2
 3 from mpi4py import MPI
 4
 5 comm = MPI.COMM_WORLD
 6
 7 rank = comm.rank
 8 size = comm.size
 9 name = MPI.Get_processor_name()
10
11 num = 3
12 root = 0

13 # Code for the root
14 if rank == root:
15     commNode = 1
16     comm.send(num, dest=commNode)
17     print ('From rank ', rank, 'we sent:', num)
18     recvd = comm.recv(source=commNode)
19     print ('From rank', commNode, ', root received:', recvd)

20 # Code for rank 1
21 elif rank == 1:
22     anum = comm.recv(source=root)
23     print ('From root, rank ', rank ,'received:', anum)
```

```
Output:
From rank 0 we sent: 3
From root, rank 1 received: 3
From rank 1, root received: 4
```

Complex Point-to-point Diagram



Complex Sample Script (Part 1)

```
# Code must run with at least 3 processors, more will have no effect
total = 0
nums = [1, 2, 3, 4]

# This is the code that will run on the root node
if rank == 0:
     data = nums
# Sends the data
     comm.send(nums, dest=1)
     comm.send(nums, dest=2)
     print ('From rank', rank, 'we sent', data)
# This while loop waits for the data to be sent back, and checks that there is an answer
before proceeding
     while total == 0:
# This line receives and totals the data from the individual nodes
          total = comm.recv(source=1) + comm.recv(source=2)
     print (total)
```

Complex Sample Script (Part 2)

```
# This code runs on node 1, it totals the data, and sends it back to the root
elif rank == 1:
    data = comm.recv(source=0)
    final = nums[0] + nums[1]
    comm.send(final, dest=0)
    print (final)


# This code runs on node 2, it totals the data, and sends it back to the root
elif rank == 2:
    data = comm.recv(source=0)
    final = nums[2] + nums[3]
    comm.send(final, dest=0)
    print (final)
```

```
Output:
From rank 0 we sent: [1,2,3,4]
3
7
10
```

Point-to-Point Communications - Use Cases

- Easily designate tasks to specific processes
- Collective methods are automatic and require specific parameters to work

Collective Communication

scatter

GATHER

Scatter and Gather

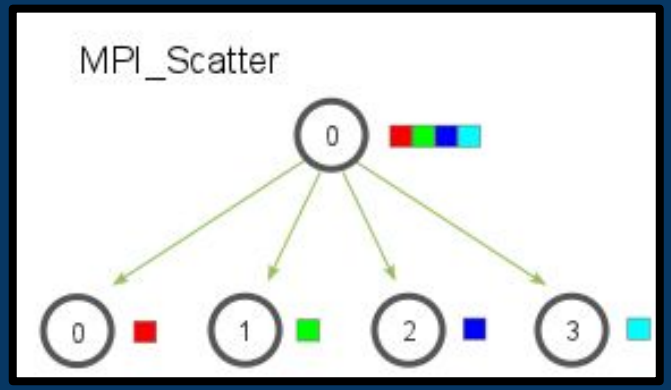
Scatter

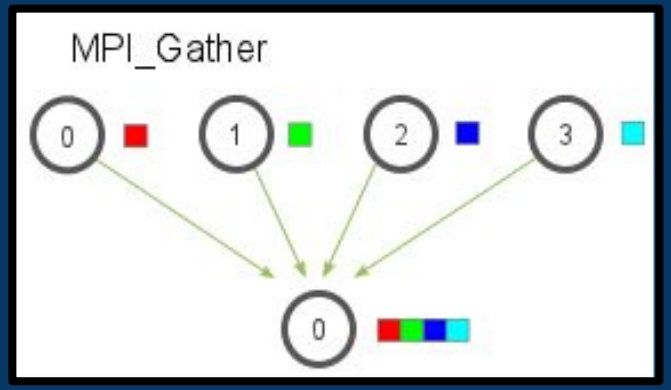
Gather

SCATTER/GATHER INTRO

Group functions meant to facilitate the collection and distribution of data

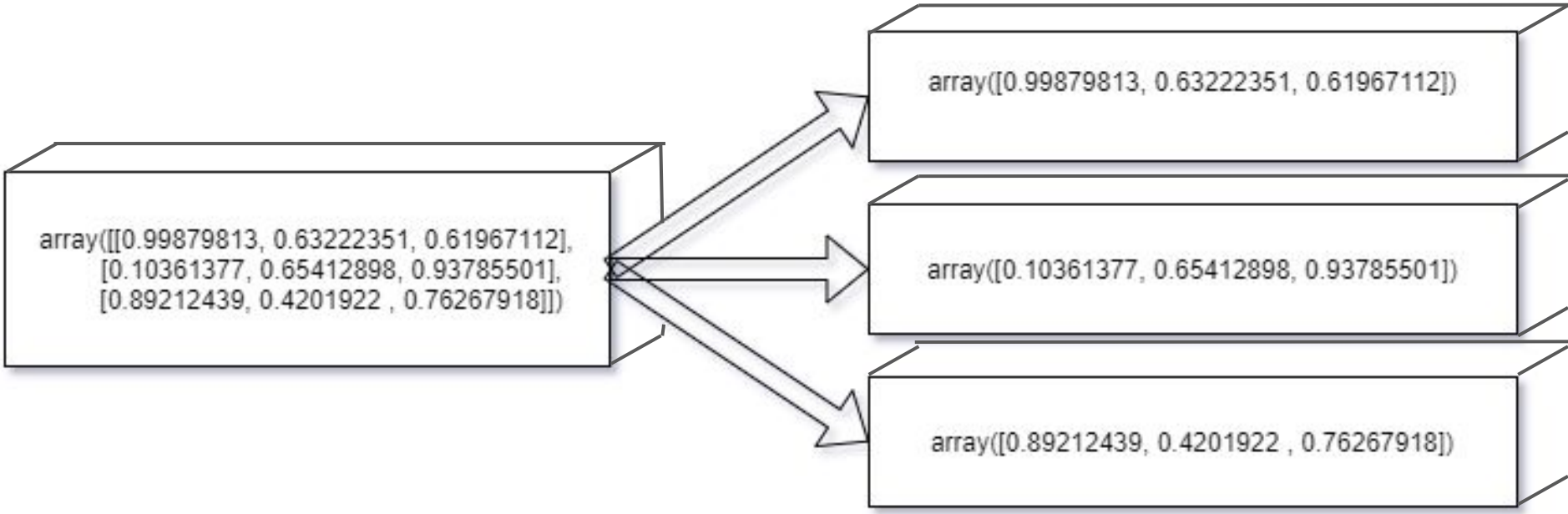
- Scatter: Divides a big array into a number of smaller parts equal to the number of processes and sends each process (including the source) a piece of the array in rank order.
- Gather: The opposite of scatter; receives data stored in small arrays from all the processes (including the source or root) and concatenates it in the receive array in rank order.





SCATTER SCRIPT

```
#!/usr/bin/env python3

from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD # Some basic setup stuff
rank = comm.Get_rank()
root = 0
processor = MPI.Get_processor_name()

data = np.random.rand(comm.size, comm.size)  # Create a <size>x<size> (24x24) grid of random numbers between 0 and 1
send_buff = [] # Define the array that will actually be scattered

if comm.rank == root: # Only on the root processor…
        send_buff = data # Put the random numbers into the array that will be scattered
        print(data) # Print out the random numbers

scat = comm.scatter(send_buff, root) # Scatter the random numbers from the root
print("I am {} : {}. I got this array : {}".format(rank, processor, scat)) # Let every node print out what it received
```


array([[[0.99879813, 0.63222351, 0.61967112],
[0.10361377, 0.65412898, 0.93785501],
[0.89212439, 0.4201922 , 0.76267918]])
array([0.99879813, 0.63222351, 0.61967112])
array([0.10361377, 0.65412898, 0.93785501])
array([0.89212439, 0.4201922 , 0.76267918])

BROKEN SCATTER SCRIPT

```
#!/usr/bin/env python3

from mpi4py import MPI

comm = MPI.COMM_WORLD

#array created with only 3 elements
arr = [1,2,3]

# if numProcessors != numElements: the scatter command will not work
num = comm.scatter(arr)
print(num)
```

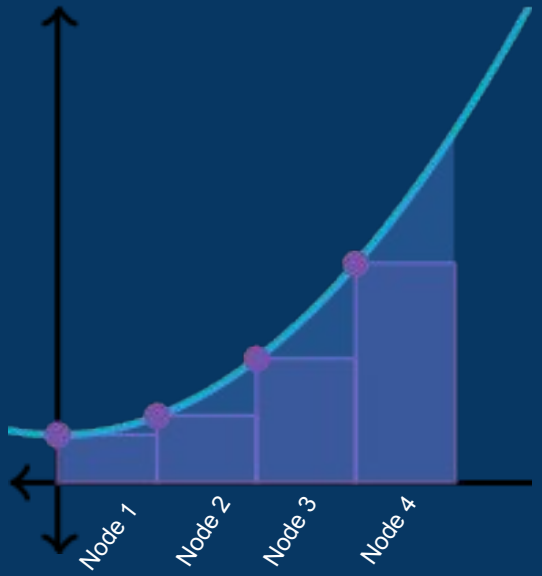
GATHER SCRIPT

```
#!/usr/bin/env python3

from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
root = 0
processor = MPI.Get_processor_name()
sendbuff = []

if comm.rank == 0:
        m = np.array(range(comm.size *comm.size), dtype=float)
        m.shape = (comm.size, comm.size)
        print(m)
        sendbuff = m

v = comm.scatter(sendbuff, root)
print("I am rank {} on {} and I received array {}".format(rank,processor,v ))

v=v*v
recvbuff = comm.gather(v, root)

if comm.rank == 0:
        print("I am rank {} on {} and gathered \n{}".format(rank, processor, np.array(recvbuff)))
```

Riemann Sums (feat. ScatGat)

In the following code example scatter/gather will be used to estimate the value of pi using Riemann sums:

1. Each subinterval is declared in the root
2. Subintervals are scattered to the different processors
3. The processors calculate the area of the interval
4. This data is gathered into the root where it is added together
5. The data is compared to pi to evaluate the error



COMBINED SCATTER/GATHER SCRIPT

```
#! /usr/bin/env python3

from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.size
rank = comm.Get_rank()
processor = MPI.Get_processor_name()
root = 0

def trapz(f,a,b,N=50):
        x = np.linspace(a,b,N+1)
        y = f(x)
        y_right = y[1:]
        y_left = y[:-1]
        dx = (b-a)/N
        return (dx/2) * np.sum(y_right + y_left)
a = 0
b = 1

n = 10**7
f = lambda x :np.sqrt(1- x**2)
s = np.linspace(a, b, size+1)
send_buff = []

if(rank == root):
        for i in range(len(s)-1):
                send_buff.append([s[i], s[i+1]])
        print(send_buff)

scat = comm.scatter(send_buff, root)
temp = trapz(f, scat[0], scat[1], n)
print("I am rank {} on {} and I received {}. The integral is
{}".format(rank, processor, scat, temp))

recvbuff = comm.gather(temp, root)

if(rank == root):
        print("I am rank {} on {}, and received {} and the sum *
4 is {} which {} away from pi".format(rank, processor,
recvbuff, sum(recvbuff)*4, np.pi-sum(recvbuff)*4))
```

On the interval from a to b, creates an array of size+1 elements, and they're evenly spaced.

Takes the above list and creates a list of tuples. The tuples are the intervals which will be scattered

Using the function f declared above on the scattered interval

Scatter the intervals to each of the cores to compute the trapz function on that given interval.

[[0.0, 0.16666666666666666], [0.16666666666666666, 0.3333333333333333], [0.3333333333333333, 0.5], [0.5, 0.6666666666666666], [0.6666666666666666, 0.8333333333333333], [0.8333333333333333, 1.0]]

I rank 1 on processor cluster0. I received the interval [0.16666666666666666, 0.3333333333333333], and broke it down into [0.16666667 0.19444444 0.22222222 0.25 0.27777778 0.30555556
 0.33333333] subintervals : [[0.16666666666666666, 0.19444444444444442], [0.19444444444444442, 0.2222222222222222], [0.2222222222222222, 0.25], [0.25, 0.2777777777777778], [0.2777777777777778, 0.3055555555555556], [0.3055555555555556, 0.3333333333333333]]

I rank 2 on processor cluster0. I received the interval [0.3333333333333333, 0.5], and broke it down into [0.33333333 0.36111111 0.38888889 0.41666667 0.44444444 0.47222222
 0.5   ] subintervals : [[0.3333333333333333, 0.3611111111111111], [0.3611111111111111, 0.3888888888888889], [0.3888888888888889, 0.41666666666666663], [0.41666666666666663, 0.4444444444444444], [0.4444444444444444, 0.4722222222222222], [0.4722222222222222, 0.5]]
I rank 3 on processor cluster0. I received the interval [0.5, 0.6666666666666666], and broke it down into [0.5   0.52777778 0.55555556 0.58333333 0.61111111 0.63888889
 0.66666667] subintervals : [[0.5, 0.5277777777777778], [0.5277777777777778, 0.5555555555555556], [0.5555555555555556, 0.5833333333333333], [0.5833333333333333, 0.6111111111111111], [0.6111111111111111, 0.6388888888888888], [0.6388888888888888, 0.6666666666666666]]

I rank 0 on processor cluster0. I received the interval [0.0, 0.16666666666666666], and broke it down into [0.   0.02777778 0.05555556 0.08333333 0.11111111 0.13888889
 0.16666667] subintervals : [[0.0, 0.027777777777777776], [0.027777777777777776, 0.05555555555555555], [0.05555555555555555, 0.08333333333333333], [0.08333333333333333, 0.1111111111111111], [0.1111111111111111, 0.1388888888888889], [0.1388888888888889, 0.16666666666666666]]

I rank 4 on processor cluster1. I received the interval [0.6666666666666666, 0.8333333333333333], and broke it down into [0.66666667 0.69444444 0.72222222 0.75 0.77777778 0.80555556
 0.83333333] subintervals : [[0.6666666666666666, 0.6944444444444444], [0.6944444444444444, 0.7222222222222222], [0.7222222222222222, 0.75], [0.75, 0.7777777777777777], [0.7777777777777777, 0.8055555555555555], [0.8055555555555555, 0.8333333333333333]]

I rank 5 on processor cluster1. I received the interval [0.8333333333333333, 1.0], and broke it down into [0.83333333 0.86111111 0.88888889 0.91666667 0.94444444 0.97222222
 1.   ] subintervals : [[0.8333333333333333, 0.861111111111111], [0.861111111111111, 0.8888888888888888], [0.8888888888888888, 0.9166666666666666], [0.9166666666666666, 0.9444444444444444], [0.9444444444444444, 0.9722222222222222], [0.9722222222222222, 1.0]]

I am rank 0 on cluster0, and received [0.16589181437511716, 0.16116148061562116, 0.1512524437545209, 0.13501008686820073, 0.10956073205231581, 0.06252160573162974] and the sum * 4 is 3.1415926535896226 which 1.7053025658242404e-13 away from pi

Broadcast and Reduce