Parallel Computing

Collective Operations

Ackwonledgement: https://nyu-cds.github.io/python-mpi

Non-blocking communication

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

randNum = numpy.zeros(1)
diffNum = numpy.random.random_sample(1)

if rank == 1:
        randNum = numpy.random.random_sample(1)
        print("Process", rank, "drew the number", randNum[0])
        comm.Isend(randNum, dest=0)
        diffNum /= 3.14 # overlap communication
        print("diffNum=", diffNum[0])
        req = comm.Irecv(randNum, source=0)
        req.Wait()
        print("Process", rank, "received the number", randNum[0])

if rank == 0:
        print("Process", rank, "before receiving has the number", randNum[0])
        req = comm.Irecv(randNum, source=1)
        req.Wait()
        print("Process", rank, "received the number", randNum[0])
        randNum *= 2
        comm.Isend(randNum, dest=1)
```

Run this program using the command:

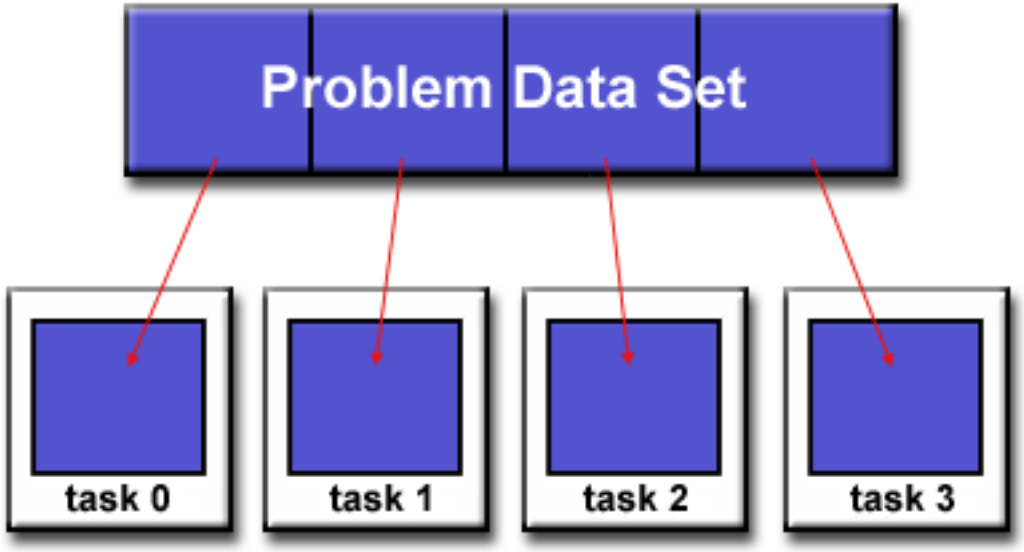
```
mpiexec -n 2 python mpi6.py
```

You should see output similar to the following:

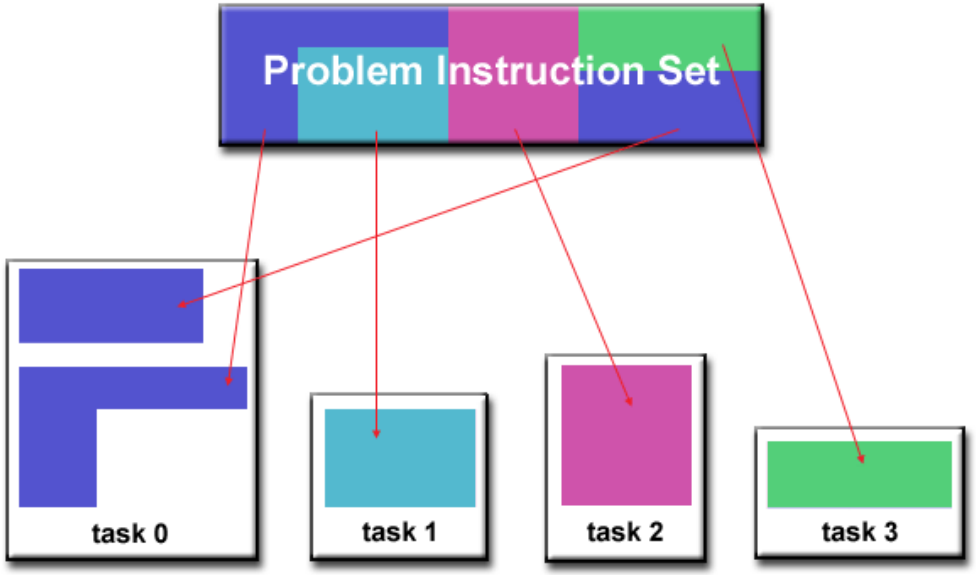
```
Process 0 before receiving has the number 0.0
Process 0 received the number 0.311574412567
Process 1 drew the number 0.311574412567
diffNum= 0.00213775044313
Process 1 received the number 1.93636680934
```

Problem Decomposition

Domain decomposition



Functional decomposition



Collective Operations

There are many situations in parallel programming when groups of processes need to exchange messages. Rather than explicitly sending and receiving such messages as we have been doing, the real power of MPI comes from group operations known as *collectives*.

Collective communications allow the sending of data between multiple processes of a group simultaneously.

Collective Operations

Commonly used collective communication operations are the following:

- Synchronization
  - Processes wait until all members of the group have reached the synchronization point

- Global communication functions
  - Broadcast data from one member to all members of a group
  - Gather data from all members to one member of a group
  - Scatter data from one member to all members of a group

- Collective computation (reductions)
  - One member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.

- Collective Input/Output
  - Each member of the group reads or writes a section of a file.

Collective communication

One of the things to remember about collective communication is that it implies a *synchronization point* among processes. This means that all processes must reach a point in their code before they can all begin executing again.

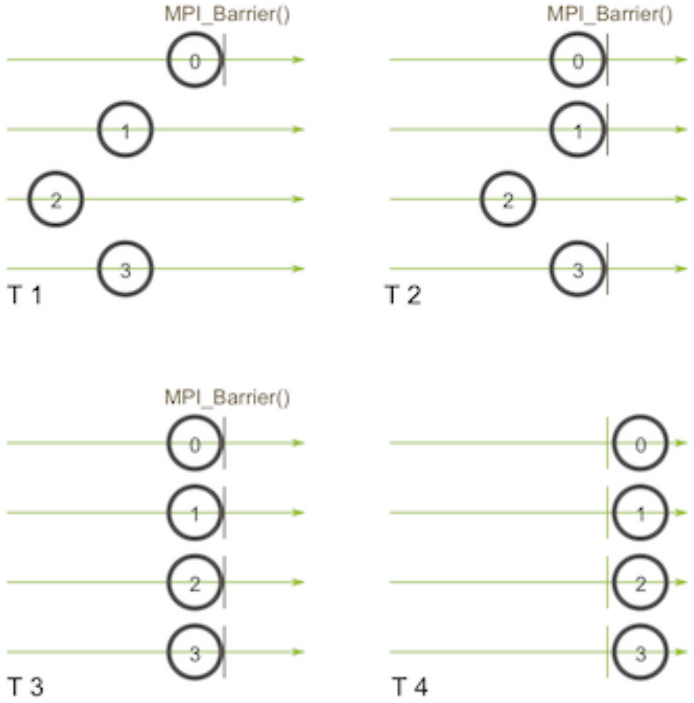
Collective communication routines must involve all processes within the scope of a communicator.

- Unexpected behavior, including program failure, can occur if even one task in the communicator doesn't participate.
- It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.

Collective communication, synchonization points

MPI has a special function that is dedicated to synchronizing processes: Comm.Barrier()

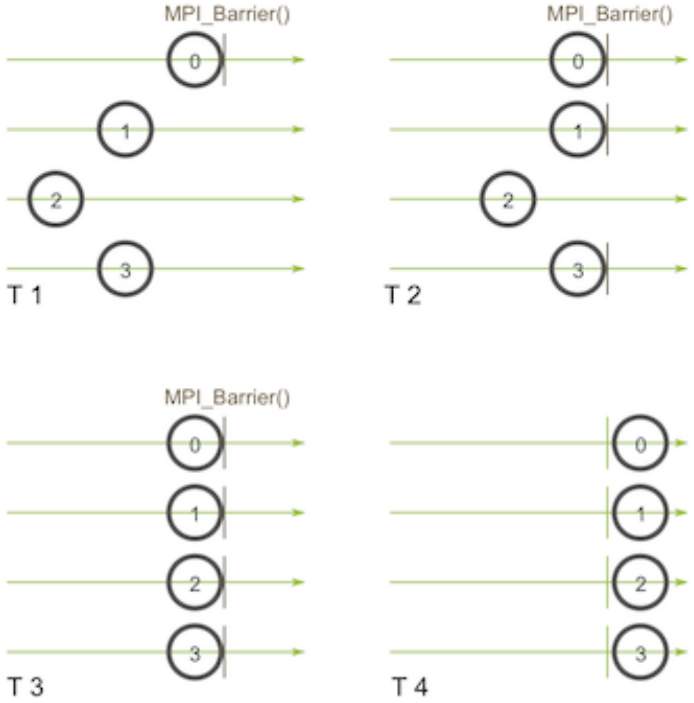
The name of the function is quite descriptive - the function forms a barrier, and no processes in the communicator can pass the barrier until all of them call the function. Here's an illustration. Imagine the horizontal axis represents execution of the program and the circles represent different processes:



Collective communication, synchonization points

MPI has a special function that is dedicated to synchronizing processes: Comm.Barrier()

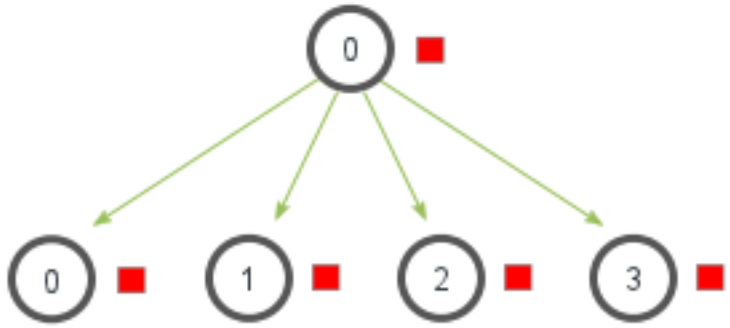
Process zero first calls Barrier at the first-time snapshot (T1). While process zero is hung up at the barrier, process one and three eventually make it (T2). When process two finally makes it to the barrier (T3), all of the processes then begin execution again (T4).



Broadcasting

A broadcast is one of the standard collective communication techniques. During a broadcast, one process sends the same data to all processes in a communicator.

One of the main uses of broadcasting is to send out user input to a parallel program or send out configuration parameters to all processes.

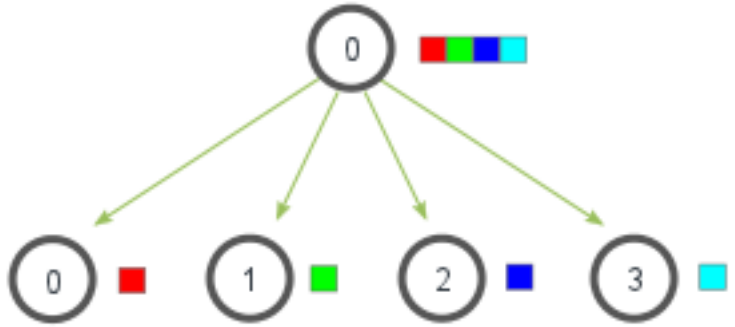


Scatter

Scatter is a collective operation that is very similar to broadcast. Scatter involves a designated root process sending data to all processes in a communicator.

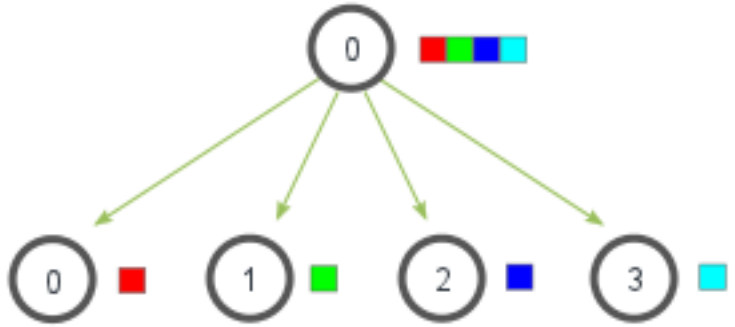
The primary difference between broadcast and scatter is:

- Broadcast sends the same piece of data to all processes while scatter sends chunks of an array to different processes.



Scatter

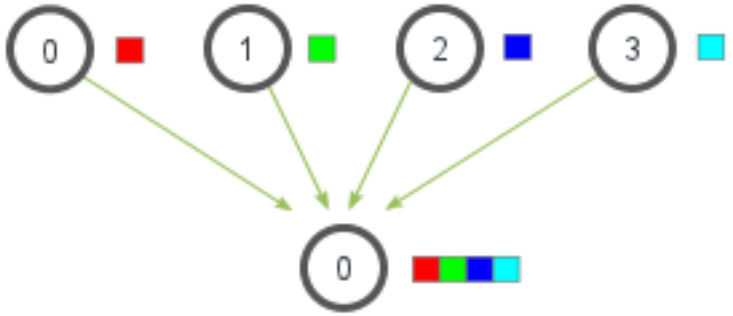
The scatter takes an array of elements and distributes the elements in the order of process rank. The first element (in red) goes to process zero, the second element (in green) goes to process one, and so on.



Gather

Gather is the inverse of scatter. The gather operation takes elements from many processes and gathers them to one single process.

This routine is highly useful to many parallel algorithms, such as parallel sorting and searching.



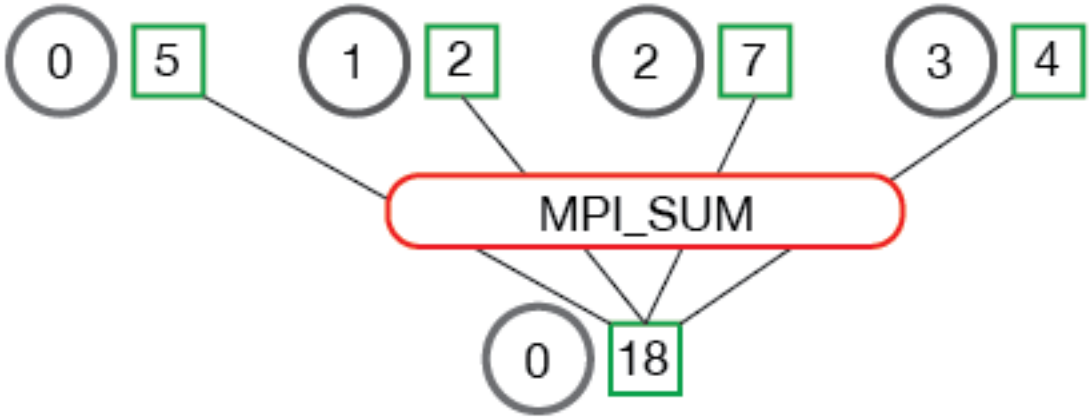
Reduction

Reduce is a classic concept from functional programming. Data reduction involves reducing a set of numbers into a smaller set of numbers via a function. For example, let's say we have a list of numbers [1, 2, 3, 4, 5]. Reducing this list of numbers with the sum function would produce sum([1, 2, 3, 4, 5]) = 15. Similarly, the multiplication reduction would yield multiply([1, 2, 3, 4, 5]) = 120.

Tt can be very cumbersome to apply reduction functions across a set of distributed numbers. Along with that, it is difficult to efficiently program non-commutative reductions, i.e., reductions that must occur in a set order. Luckily, there is a handy function called Comm.Reduce() that will handle almost all the common reductions that a programmer needs to do in a parallel application.

Reduction

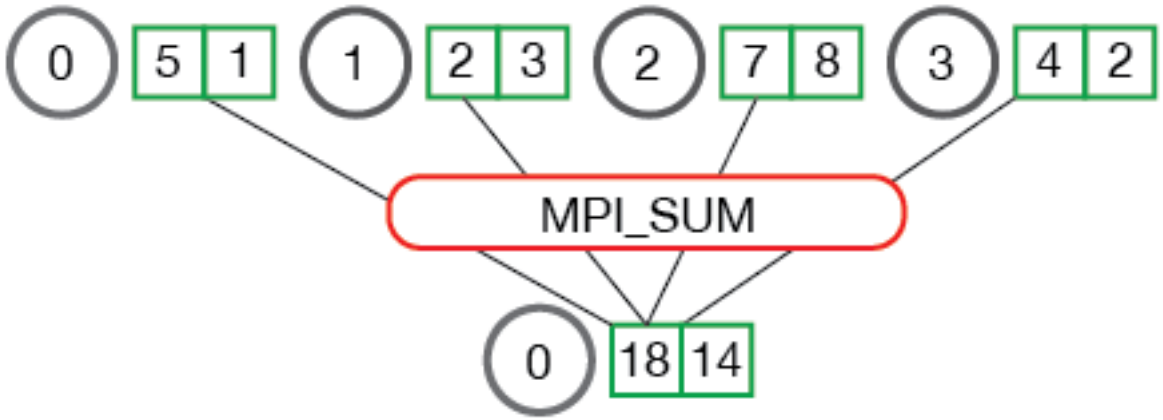
The Comm.Reduce method takes an array of input elements and returns an array of output elements to the root process. The output elements contain the reduced result. MPI contains a set of common reduction operations that can be used.



Reduction

It is also useful to see what happens when processes contain multiple elements.

The processes from the below illustration each have two elements. The resulting summation happens on a per-element basis. In other words, instead of summing all of the elements from all the arrays into one element, the i^{th} element from each array are summed into the i^{th} element in result array of process 0.



Collective operations

📌 `Comm.Barrier()`

Synchronization operation. Creates a barrier synchronization in a group. Each task, when reaching the Barrier() call, blocks until all tasks in the group reach a `Barrier()` call. Then all tasks are free to proceed.

📌 `Comm.Bcast(buf, root=0)`

Data movement operation. Broadcasts (sends) a message from the process with rank “root” to all other processes in the group.

📌 `Comm.Scatter(sendbuf, recvbuf, root=0)`

Data movement operation. Distributes distinct messages from a single source task to each task in the group.

📌 `Comm.Gather(sendbuf, recvbuf, root=0)`

Data movement operation. Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of `Scatter()`.

Collective operations

📌 `Comm.Reduce(sendbuf, recvbuf, op=MPI.SUM, root=0)`

Reduces values on all processes to a single value by applying the operation op. Operations include:

- `MPI.MAX` - Returns the maximum element.
- `MPI.MIN` - Returns the minimum element.
- `MPI.SUM` - Sums the elements.
- `MPI.PROD` - Multiplies all elements.
- `MPI.LAND` - Performs a logical and across the elements.
- `MPI.LOR` - Performs a logical or across the elements.
- `MPI.BAND` - Performs a bitwise and across the bits of the elements.
- `MPI.BOR` - Performs a bitwise or across the bits of the elements.
- `MPI.MAXLOC` - Returns the maximum value and the rank of the process that owns it.
- `MPI.MINLOC` - Returns the minimum value and the rank of the process that owns it.

Collective operations: Bcast

```python
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = numpy.arange(15).reshape(3, 5)
else:
    data = numpy.empty(15, dtype='int').reshape(3, 5)

comm.Bcast(data, root=0)
print('Rank: ', rank, ', data: ', data)
```

Collective operations: Scatter

```python
from mpi4py import MPI
import numpy as np

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

numDataPerRank = 10
data = None
if rank == 0:
    data = np.linspace(1,size*numDataPerRank,numDataPerRank*size)

Recvbuf = np.empty(numDataPerRank, dtype='d')
comm.Scatter(data, Recvbuf, root=0)
print('Rank: ', rank, ', Recvbuf received: ', Recvbuf)
```

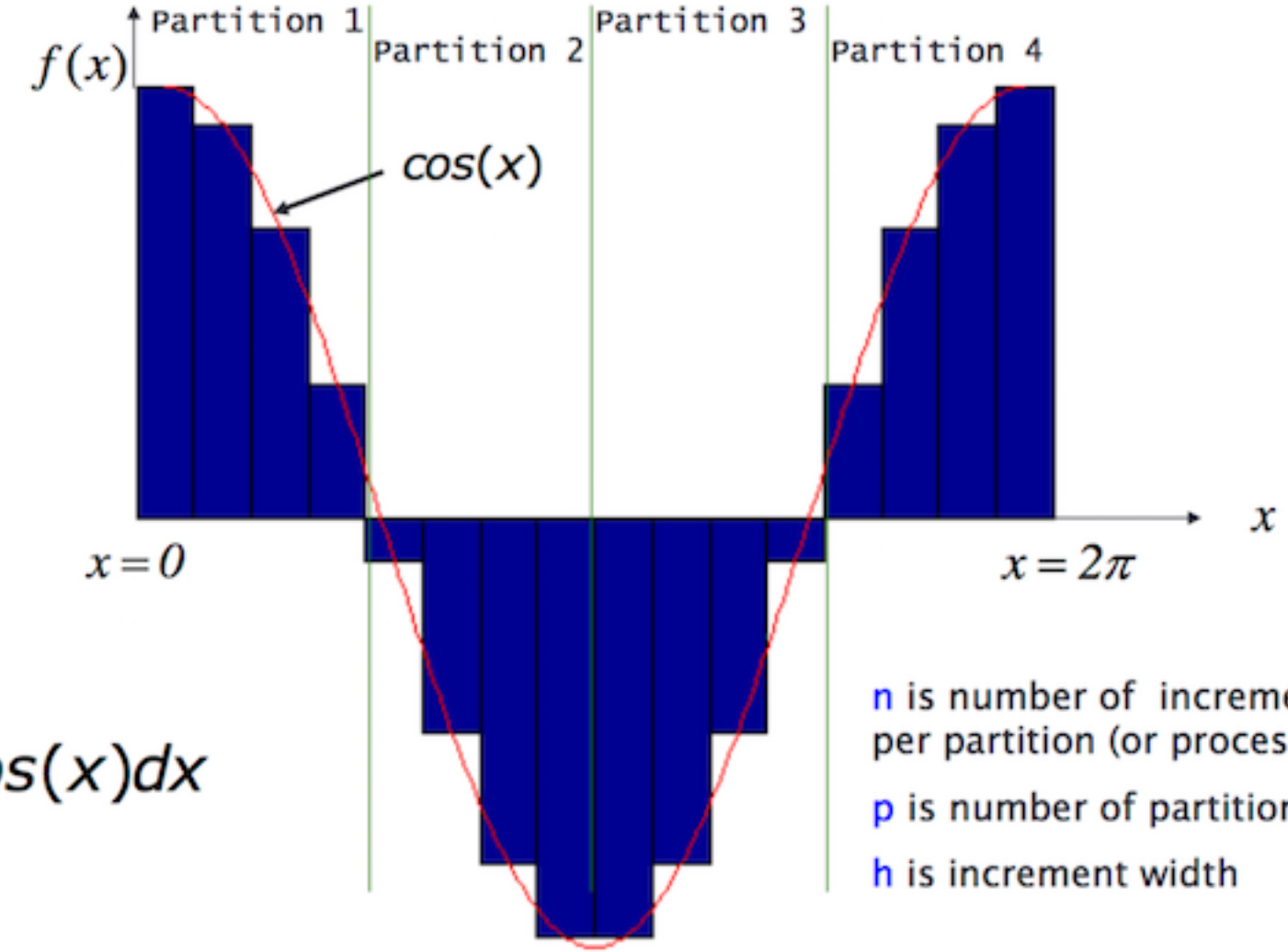
Collective operations: Gather

```python
from mpi4py import MPI
import numpy as np

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

numDataPerRank = 10
sendbuf = np.linspace(rank*numDataPerRank+1, (rank+1)*numDataPerRank, numDataPerRank)
print('Rank: ', rank, ', sendbuf: ', sendbuf)

recvbuf = None
if rank == 0:
    recvbuf = np.empty(numDataPerRank*size, dtype='d')

comm.Gather(sendbuf, recvbuf, root=0)

if rank == 0:
    print('Rank: ', rank, ', recvbuf received: ', recvbuf)
```

Collective operations: Reduce

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

value = np.array(rank, 'd')

print(' Rank: ', rank, ' value = ', value)

# initialize the np arrays that will store the results:
value_sum = np.array(0.0, 'd')
value_max = np.array(0.0, 'd')

# perform the reductions:
comm.Reduce(value, value_sum, op=MPI.SUM, root=0)
comm.Reduce(value, value_max, op=MPI.MAX, root=0)

if rank == 0:
    print(' Rank 0: value_sum = ', value_sum)
    print(' Rank 0: value_max = ', value_max)
```

Numerical integration



$$\int_0^{2\pi} cos(x)dx$$

n is number of increments per partition (or processor)

p is number of partitions

h is increment width

Parallel point-to-point version

```
import numpy
from math import acos, cos
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

def integral(a_i, h, n):
    integ = 0.0
    for j in range(n):
        a_ij = a_i + (j + 0.5) * h
        integ += cos(a_ij) * h
    return integ

pi = 3.14159265359
n = 500
a = 0.0
b = pi / 2.0
h = (b - a) / (n * size)
a_i = a + rank * n * h

# All processes initialize my_int with their partition calculation
my_int = numpy.full(1, integral(a_i, h, n))

print("Process ", rank, " has the partial integral ", my_int[0])

if rank == 0:
    # Process 0 receives all the partitions and computes the sum
    integral_sum = my_int[0]
    for p in range(1, size):
        comm.Recv(my_int, source=p)
        integral_sum += my_int[0]

    print("The integral = ", integral_sum)
else:
    # All other processes just send their partition values to process 0
    comm.Send(my_int, dest=0)
```

This program can be run with the following command:

```
mpiexec -n 4 python midpoint_par.py
```

When run, output similar to the following will be generated:

```
Process  0  has the partial integral  0.382683442201
The integral =  1.0000000257
Process  1  has the partial integral  0.32442335716
Process  2  has the partial integral  0.216772756896
Process  3  has the partial integral  0.0761204694451
```

Parallel collective version

```
import numpy
from math import acos, cos
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

def integral(a_i, h, n):
    integ = 0.0
    for j in range(n):
        a_ij = a_i + (j + 0.5) * h
        integ += cos(a_ij) * h
    return integ

pi = 3.14159265359
a = 0.0
b = pi / 2.0
dest = 0
my_int = numpy.zeros(1)
integral_sum = numpy.zeros(1)

# Initialize value of n only if this is rank 0
if rank == 0:
    n = numpy.full(1, 500, dtype=int) # default value
else:
    n = numpy.zeros(1, dtype=int)

# Broadcast n to all processes
print("Process ", rank, " before n = ", n[0])
comm.Bcast(n, root=0)
print("Process ", rank, " after n = ", n[0])

# Compute partition
h = (b - a) / (n * size) # calculate h *after* we receive n
a_i = a + rank * h * n
my_int[0] = integral(a_i, h, n[0])

# Send partition back to root process, computing sum across all partitions
print("Process ", rank, " has the partial integral ", my_int[0])
comm.Reduce(my_int, integral_sum, MPI.SUM, dest)

# Only print the result in process 0
if rank == 0:
    print('The Integral Sum =', integral_sum[0])
```

This program is run with the command:

```
mpiexec -n 4 python midpoint_coll.py
```

The following is an example of the output generated:

```
Process  0  before n =  500
Process  3  before n =  0
Process  1  before n =  0
Process  2  before n =  0
Process  0  after n =  500
Process  2  after n =  500
Process  1  after n =  500
Process  3  after n =  500
Process  0  has the partial integral  0.382683442201
Process  1  has the partial integral  0.32442335716
Process  2  has the partial integral  0.216772756896
Process  3  has the partial integral  0.0761204694451
The Integral Sum = 1.0000000257
```

Communication of buffer-like objects

Method names starting with an **upper-case** letter (of the Comm class),
like Send(), Recv(), Bcast(), Scatter(), Gather(), etc.

In general, buffer arguments to these calls must be explicitly specified by using a 2/3-list/tuple like [data, MPI.DOUBLE], or [data, count, MPI.DOUBLE] (the former one uses the byte-size of data and the extent of the MPI datatype to define count).

Communication of generic Python objects

For **all-lowercase** methods (of the Comm class), like send(), recv(), bcast(), etc, an object to be sent is passed as a parameter to the communication call, and the received object is simply the return value.

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

a = [10]
b = 10000
m = "hello"
s = bytearray(m.encode("ASCII"))
if rank == 1:
        print("Buffer Test: Process", rank, "begin the value ", s)
        s[1] = 100
        comm.Send([s, 5, MPI.CHAR], dest=0)
        print("Buffer Test: Process", rank, "sent the value ", s)
if rank == 0:
        print("Buffer Test: Process", rank, "before receiving has the value ", s, flush=True)
        comm.Recv([s, 5, MPI.CHAR], source=1)
        print("Buffer Test: Process", rank, "received the value", s.decode("ASCII"))


if rank == 1:
        print("Object Test: Process", rank, "drew the value ", b)
        m = m.capitalize()
        b += 10
        comm.send(b, dest=0)
        print("Object Test: Process", rank, "sent the value ", b)
if rank == 0:
        print("Object Test: Process", rank, "before receiving has the value ", b, flush=True)
        b = comm.recv(source=1)
        print("Object Test: Process", rank, "received the value ", b)
```