Parallel Computing

Non-blocking Communication

Problem Decomposition

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

Blocking communication

The sender or receiver is not able to perform any other actions until the corresponding message has been sent or received (to be accurate, until the buffer is safe to use.)

Disadvantages:

- potential computational time is simply wasted while waiting for the call to complete
- blocking communication can easily lead to deadlock

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

randNum = numpy.zeros(1)

if rank == 1:
        randNum = numpy.random.random_sample(1)
        print("Process", rank, "drew the number", randNum[0])
        comm.Send(randNum, dest=0)

if rank == 0:
        print("Process", rank, "before receiving has the number", randNum[0])
        comm.Recv(randNum, source=1)
        print("Process", rank, "received the number", randNum[0])
```

Non-blocking communication

An alternate approach is to allow the program to continue execution while the messages is being sent or received.

In MPI, non-blocking communication is achieved using the Isend and Irecv methods.
The Isend and Irecv methods initiate a send and receive operation respectively, and then return immediately.

These methods return an instance of the Request class, which uniquely identifies the started operation. The completion can then be managed using the Test, Wait, and Cancel methods of the Request class. The management of Request objects and associated memory buffers involved in communication requires careful coordination. Users must ensure that objects exposing their memory buffers are not accessed at the Python level while they are involved in nonblocking message-passing operations.

Non-blocking communication

The following example performs the same simple blocking send and receive as demonstrated previously, however this time it is done with the non-blocking versions of the send and receive methods. The calls to Wait() immediately following the non-blocking methods will block the process until the corresponding send and receives have completed.

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

randNum = numpy.zeros(1)

if rank == 1:
        randNum = numpy.random.random_sample(1)
        print("Process", rank, "drew the number", randNum[0])
        req = comm.Isend(randNum, dest=0)
        req.Wait()

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])
```

Run this program using the command:

```
mpiexec -n 2 python mpi5.py
```

You should see output similar to the following:

```
Process 0 before receiving has the number 0.0
Process 0 received the number 0.97400925874
Process 1 drew the number 0.97400925874
```

Challenge

What happens if you comment out *both* lines containing the comm.Wait calls? Can you explain what you are seeing?

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

randNum = numpy.zeros(1)

if rank == 1:
        randNum = numpy.random.random_sample(1)
        print("Process", rank, "drew the number", randNum[0])
        req = comm.Isend(randNum, dest=0)
        req.Wait()

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])
```

Non-blocking communication

Now let's create a truly non-blocking version of the send and receive program. Note there is no need to wait after process 1 sends the message, nor after process 0 sends the reply. However, it is necessary for process 1 to wait for the reply so that it knows the message has been fully received before trying to print it out. Similarly, process 0 must wait for the full message before trying to compute randNum * 2.

```
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)
        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 0.570093200547
Process 1 drew the number 0.570093200547
Process 1 received the number 0.623148825134
```

Overlapping communication and computation

Now let's modify this program so that process 1 overlaps a computation with sending the message and receiving the reply. The computation should be dividing diffNum by 3.14 and printing the result.

```
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
```

Other operations

The Request.Wait method blocks the calling process until a corresponding communication has completed. This is not always desirable, so MPI provides a means of testing for completion without waiting. This is done using the Request.Test() method, which will return True when the message operation has completed.

When using non-blocking communication, it is sometimes necessary to cancel a pending communication. This is achieved by calling the Request.Cancel() method.

Problem Decomposition

One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks so the can work on on the problem simultaneously. This is known as *decomposition* or *partitioning*. There are two main ways to decompose an algorithm:

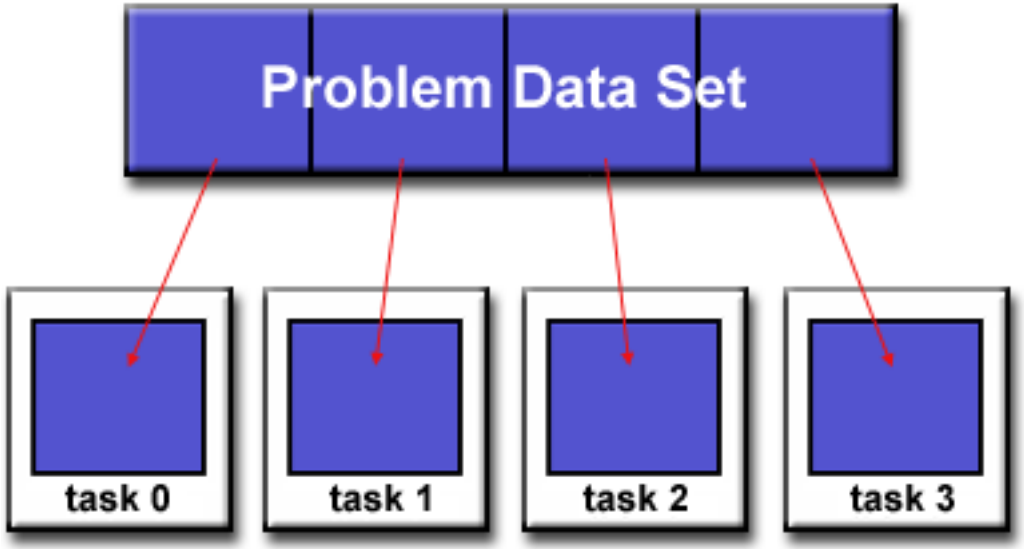
- *domain decomposition*
- *functional decomposition*

Domain decomposition

In this type of partitioning, the *data* associated with a problem is decomposed. Each parallel task then works on a portion of the data.

Key characteristics of domain decomposition are:

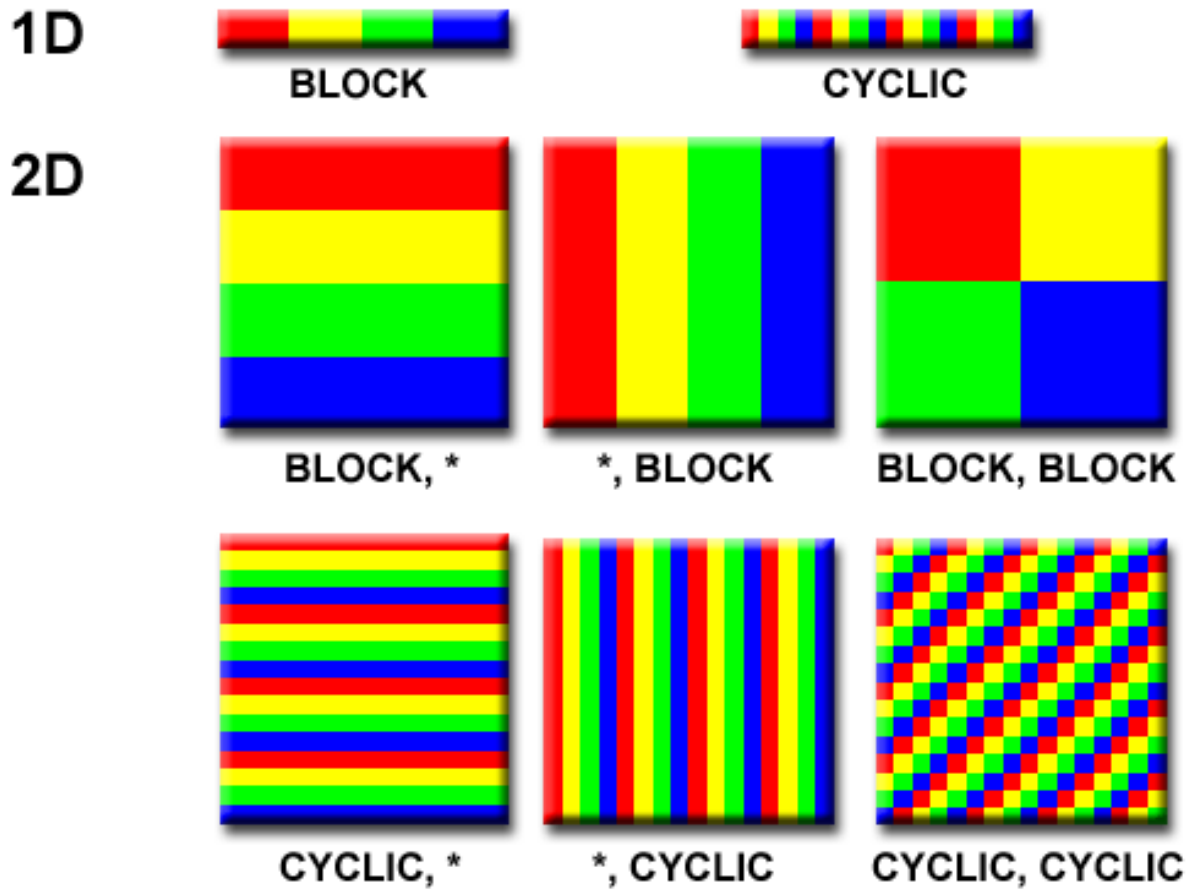
- Data divided into pieces of same size and mapped to different processors
- Processor works only on data assigned to it
- Communicates with other processors when necessary



Domain decomposition

There are many ways to partition the data.

Examples of domain decomposition
- Matrix operations
- Finite difference calculations
- Numerical integration

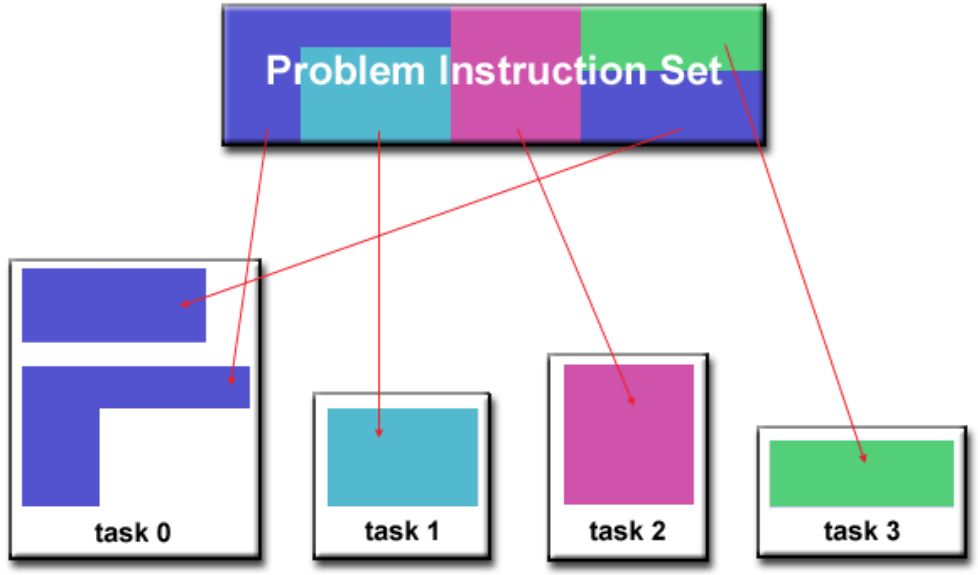


Functional decomposition

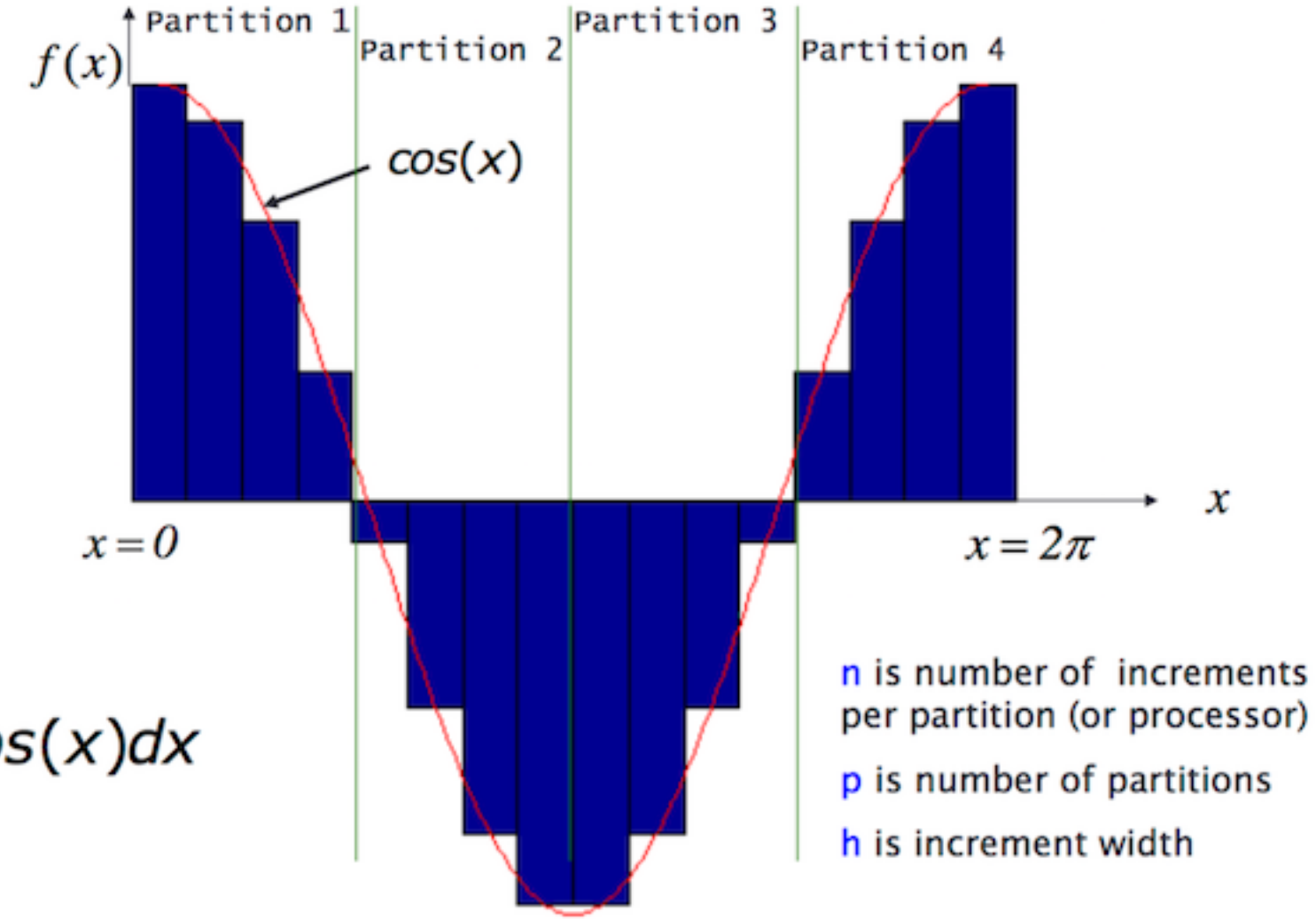
For function decomposition, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.

Key characteristics of functional decomposition are:

- Used when pieces of data require different processing times
- Performance limited by the slowest process
- Program decomposed into several small tasks
- Tasks assigned to processors as they become available
- Implemented in a master/slave paradigm



Domain decomposition example



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

n is number of increments per partition (or processor)

p is number of partitions

h is increment width

Domain decomposition example

$$\int_a^b cos(x)dx = \sum_{i=0}^{p-1} \sum_{j=0}^{n-1} \int_{a_i+j*h}^{a_i+(j+1)*h} cos(x)dx$$

$$\approx \sum_{i=0}^{p-1} \left[\sum_{j=0}^{n-1} cos(a_{ij}) * h \right] \text{ where } \begin{cases} h = \dfrac{b-a}{pn} \\ a_i = a + i * n * h \\ a_{ij} = a_i + (j + 0.5) * h \end{cases}$$

p is the number of partitions

n is the number of increments per partition

Serial version

```
from math import acos, cos

# Compute the inner sum
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
p = 4
n = 500
a = 0.0
b = pi / 2.0
h = (b - a) / (n * p)

integral_sum = 0.0

# Compute the outer sum
for i in range(p):
    a_i = a + i * n * h
    integral_sum += integral(a_i, h, n)

print("The integral = ", integral_sum)
```

When run, this code generates the following output:

```
The integral =  1.0000000257
```

Parallel point-to-point version

Since the problem has already been decomposed into separate partitions, it is easy to implement a parallel version of the algorithm. In this case, each of the partitions can be computed by a separate process. Once each process has computed its partition, it sends the result back to a root process (in this case process 0) which sums the values and prints the result.

```
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
```