AFRICAN CENTERS OF EXCELLENCE IN BIOINFORMATICS & DATA-INTENSIVE SCIENCE

ACE Workshop on HPC with Python
Day 2 MPI with Python

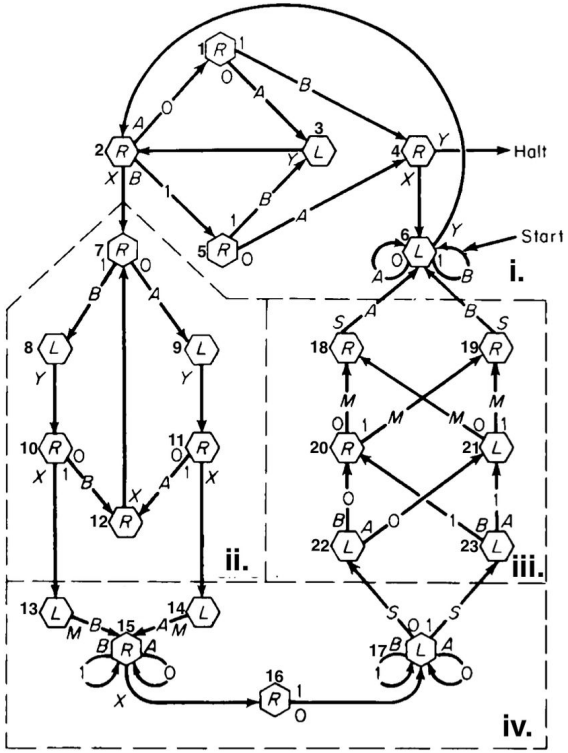
Welcome back!

- Agenda for today:
  - Thinking Parallel
  - MPI, mpi4py
    - point-to-point
    - collective
    - io operations
    - if we have time advanced topics
      - one-sided communication
      - communicators and topologies
      - custom data structures

AFRICAN CENTERS OF EXCELLENCE IN BIOINFORMATICS & DATA-INTENSIVE SCIENCE

Let us get philosophical…

Computer science is the study of *computation*, and of the agent of computation i.e. *the computer*. A computation consists of an *instruction* carried out on *data*. The instruction commands the computer to modify that data (not modifying data being a special or liminal case).



AFRICAN CENTERS OF EXCELLENCE IN BIOINFORMATICS & DATA-INTENSIVE SCIENCE

Serial vrs. Parallel Model

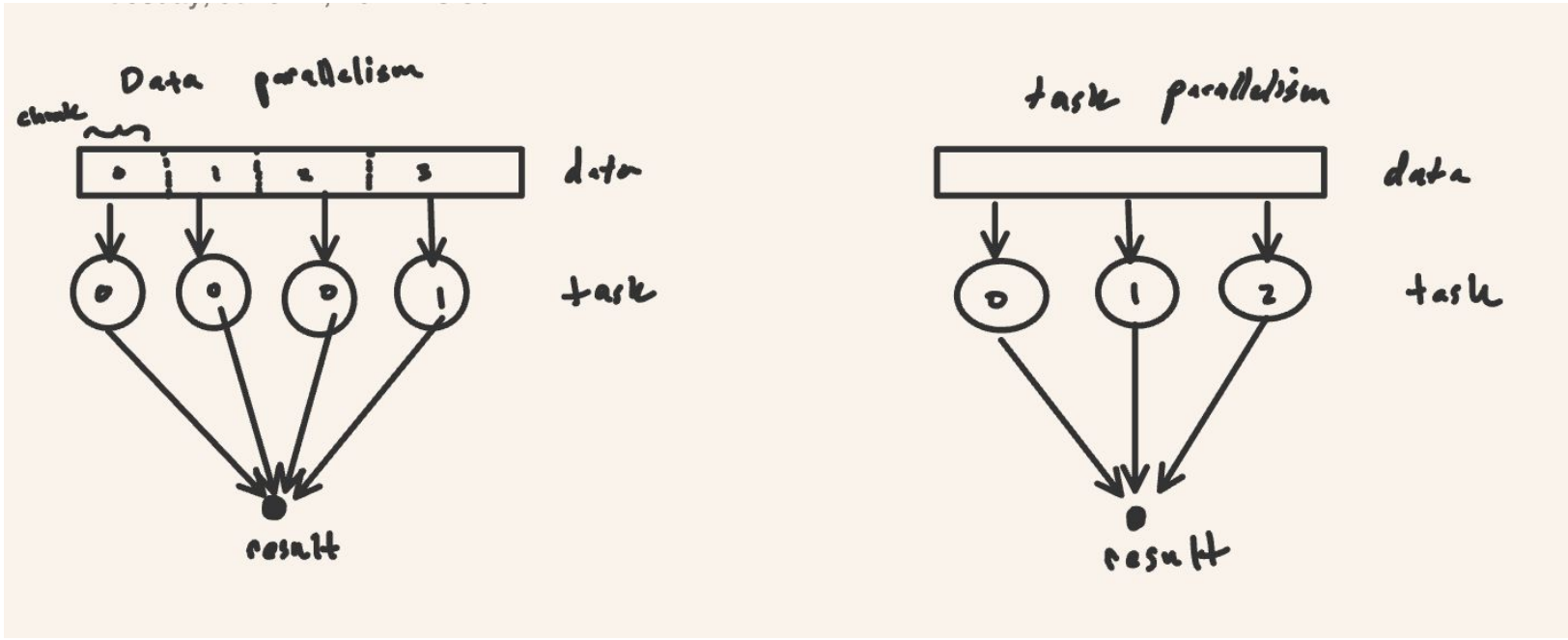
- Serial
  - Line-by-line
- Parallel
  - assume a parallel computer
  - many processors, cores, or nodes
  - execute concurrently

Task vrs. Data Parallelism

- Data parallelism divides the *data* between the various processors
- Task parallelism divides overall job into various sub-tasks
  - distributes the sub-tasks across the various processors.

Task vrs. Data Parallelism

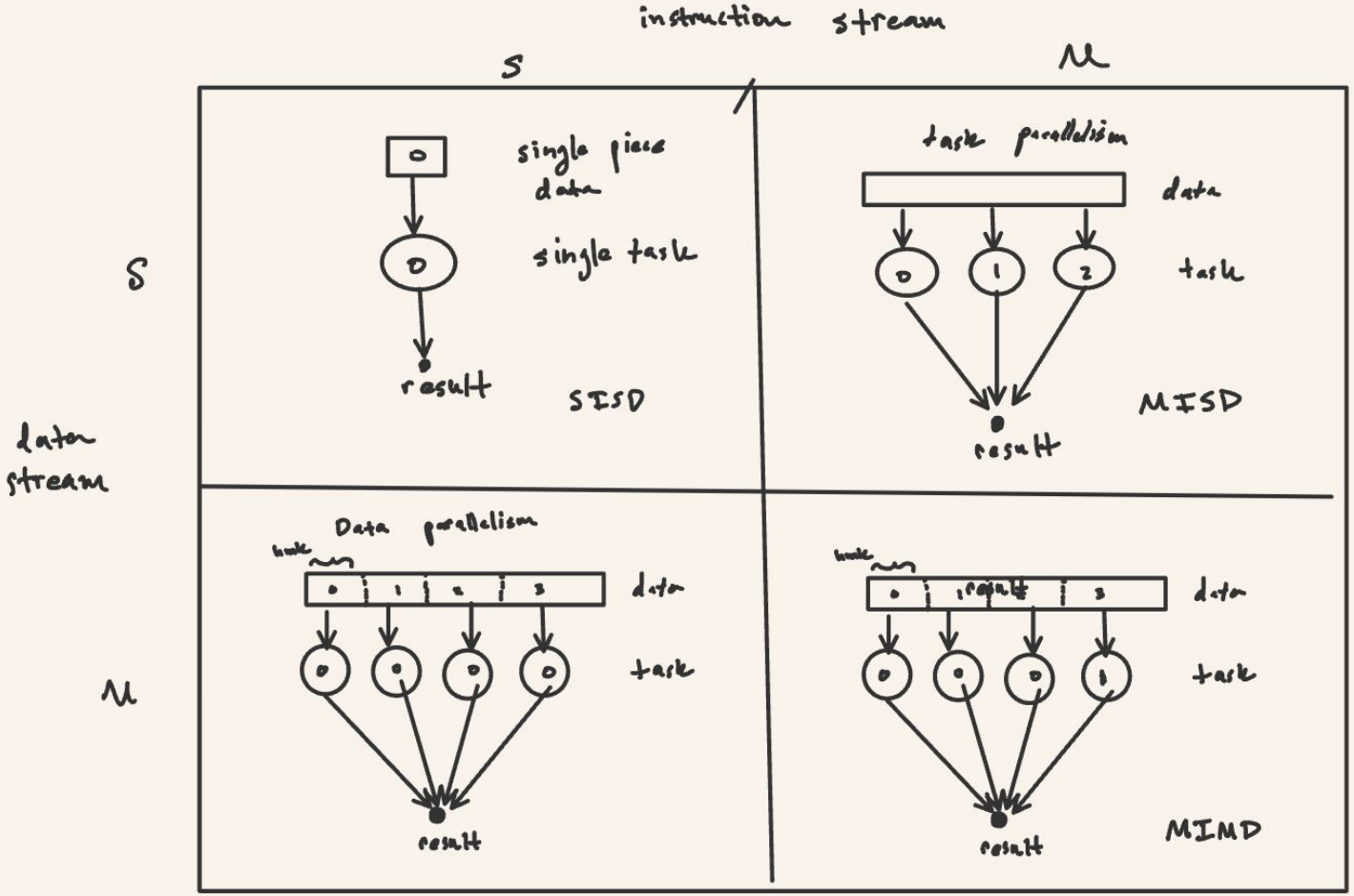
Figure 1.0.1: The block at top is the data. The circles are the processors or nodes. The data is fed into -> the processors. The result can look like anything in this depiction.



Flynn's Taxonomy

- single-instruction, single-data computation (SISD)
- single instruction and applies it to different pieces of data then it is single-instruction,multiple-data (SIMD)
- multiple-instructions and performs them on a single piece of data then we have multiple-instructions, single-data (MISD)
- multiple-instruction, multiple-data (MIMD).

Flynn's Taxonomy



AFRICAN CENTERS OF EXCELLENCE IN BIOINFORMATICS & DATA-INTENSIVE SCIENCE

Shared vrs. Distributed Memory

- How does memory work ?
- Shared vrs. Distributed Memory
  - Shared
    - processes share common storage space
    - can modify each others variables
  - Distributed
    - private storage space
    - indirectly modify each others data via e.g. messages
    - we will work with distributed system
- There are even Hybrid Systems
  - becoming more common

Shared vrs. Distributed Memory

Introduction to MPI

- In order to write distributed-programming code we will utilize message-passing interface (MPI)
- MPI is an application programming interface (API)
  - basic idea is coordination and synchronization via “messages”

Serial -> Parallel

Ian Foster in *Designing and Building Parallel Programs* provides a sort of informal algorithm to do just this.

1. Partition : divide computation to be performed and data operated on by the computation into small tasks
2. Communication: determine what communication needs to be carried out among the tasks
3. Aggregation: combine the tasks and communications identified in the first step into larger tasks. Combine those tasks that are actually “path-dependent” e.g. if task A must run before task B they must be aggregated into a task C.
4. Mapping: assign the composite tasks to processes/threads. Done so communication is minimized.

Challenges to Writing Parallel Code

A data-dependency , originally a compiler theory term, refers to the dependency between statement and its predecessor i.e. the statement executed before. There are 3 principle ones:

- true dependency , read-after-write
- anti-dependency, write-after-read
- output dependency , write-after-write

True Dependency (RAW)

This occurs when an instruction depends on result of a previous execution. For instance:

```
A = 3
B = A
C = B
print(C)
```

Instruction 3 truly depends on instruction 2. C is modified by the instruction before. If I change the instruction before it changes C. I can't say put instruction 2 on one process and instruction 3 on another.

Anti-Dependency (WAR)

Occurs when an instruction requires a value that is later updated for instance:

```
B = 3
A = B + 1
B = 7
print(A)
```

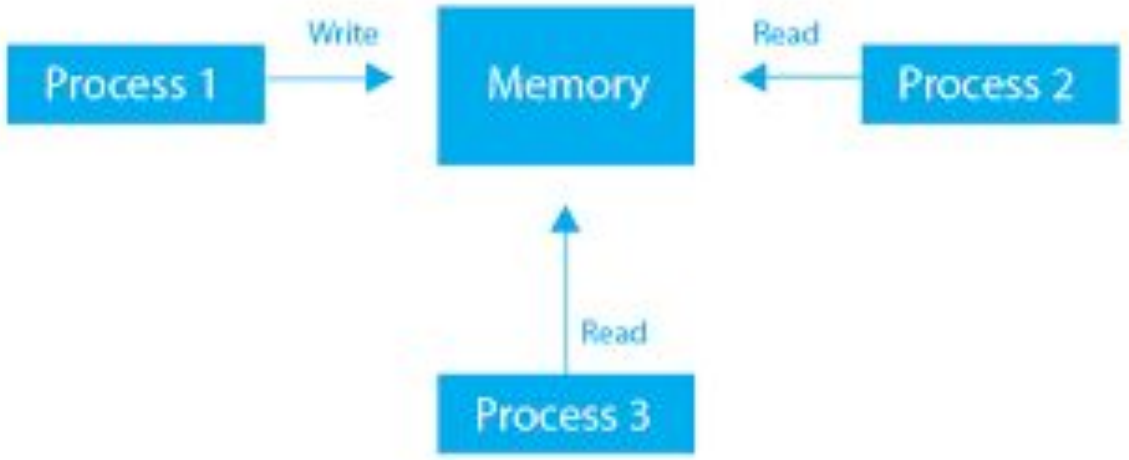
Instruction 2 anti-depends on instruction 3. Instruction 2 modifies ‘3’ but then latter the variable B is changed. If I change the order the last print statement changes. If I divide the instructions across tasks then the program will not work.

Load-Balancing

- Partition work across processes as equitably as possible
- Otherwise causes a bottleneck
  - program only as fast as the heaviest load
- Load-balancing is equitable distribution of work across nodes

Synchronization

- We have to make sure that the processes are coordinated correctly, that they are synchronized in the overall parallel program
  - otherwise we can get invalid results



Example:

Communication Overhead

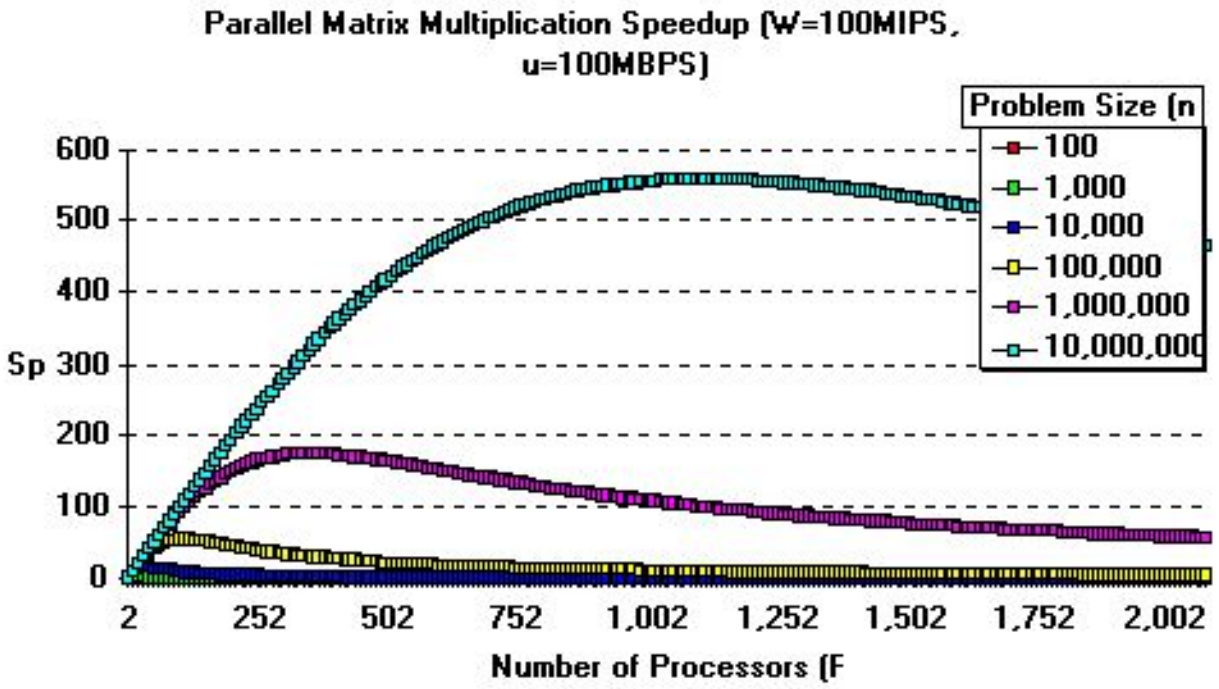
- Hypothesis: if I have n processors , I can speed up my program n-times!
- No…. **Communication ain't free!**
- Two considerations for effectiveness of parallelism:
  - The parallelizability (percent of program we can parallelize in line number) will impose a bottleneck
  - Communication Overhead

Parallelizability

- Amount of program that can be made parallel
- We take line number as an estimate of time
- Number of lines that can be made parallel/total_lines = parallelizability

Communication Overhead

Increasing processor number will increase communication time, we will see generally a graph like this:



AFRICAN CENTERS OF EXCELLENCE IN BIOINFORMATICS & DATA-INTENSIVE SCIENCE

Speedup-Equation

Express the relationship between parallelizabiklity

$$speedup = 1/((1-p) + p/N)$$

Exercise 0 : We need more tools

- Install mpi
- pip install mpi4py

Break time: 10 minutes

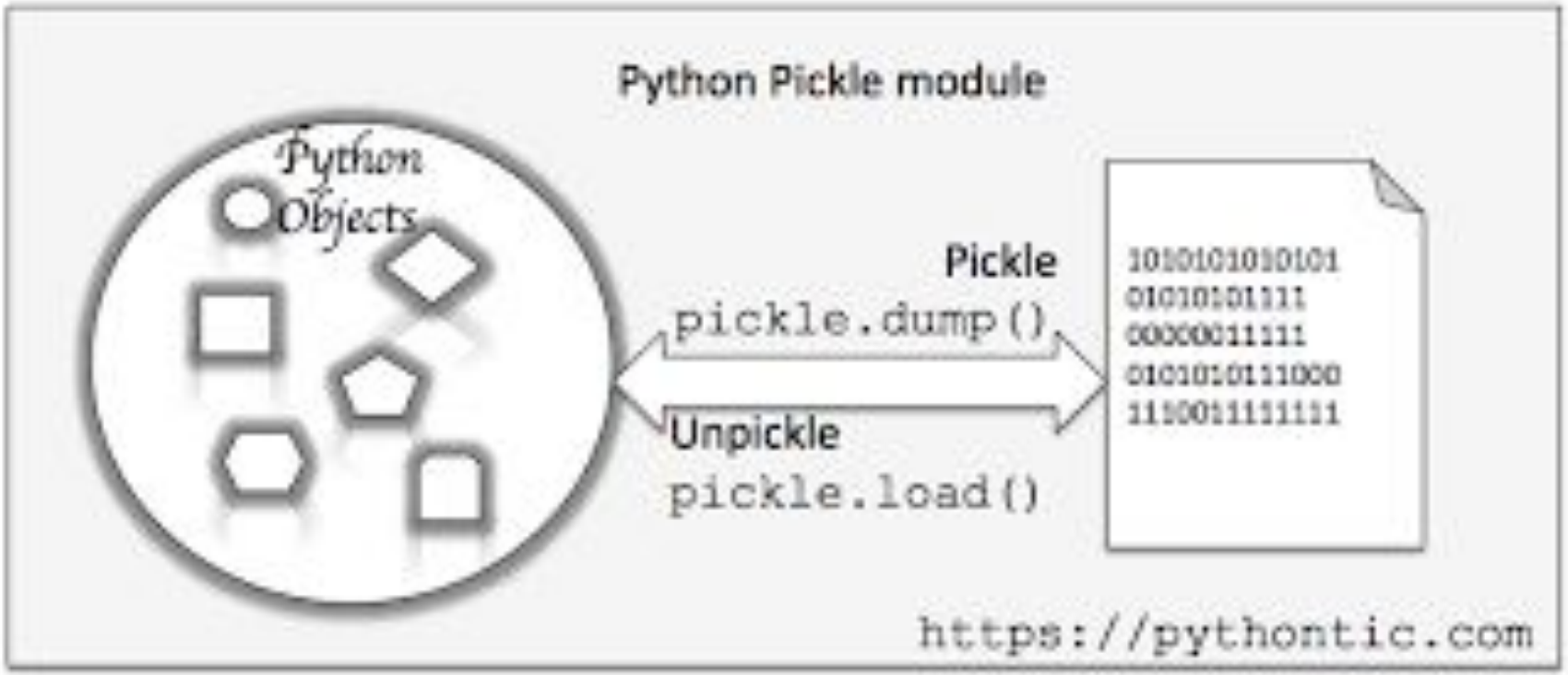
MPI and mpi4py

- message-passing interface MPI is an API
  - treat parallel computation as communication (of messages) between nodes
- Not a language
  - in C it is a library, suite of code
  - in python it is a module
  - effects how compilation works as well
- controller-worker paradigm

Picking and Marshalling

- How does MPI work , send messages in mpi4py?pi4py makes use of python **picking** and **marshaling** to communicate pythonic-data across processes.
  - The **pickle** module serializes and deserializes data using ASCII or binary formats, it takes code and produces a byte stream in ASCII or binary format.
  - The **marshal** module produces a bytestream specific to python and independent of system.
  - Good to know because whatever cannot be easily pickled will impose serious communication overhead

Pickling and Marshalling



Communicators

- A communicator is an object that links a group of *processes* and allows them to coordinate and communicate.
  - It is made up of a *group* and a *context*
- The communicator that includes all processes is called **COMM_WORLD**, and by default is instantiated by

```
from mpi4py import MPI


comm = MPI.COMM_WORLD
```

More Stuff on Communicators

- Communicator sets up ids for each process and it organizes the processors in an **ordered topology**.
- The processes compose a **group**, a set of processes.
- We identify each process within group by **rank**.
- If there are n processes , then they get labeled 0,1,2…n-1 and the size of the group is n.
- You determine the rank of a process , and the size of the group via:

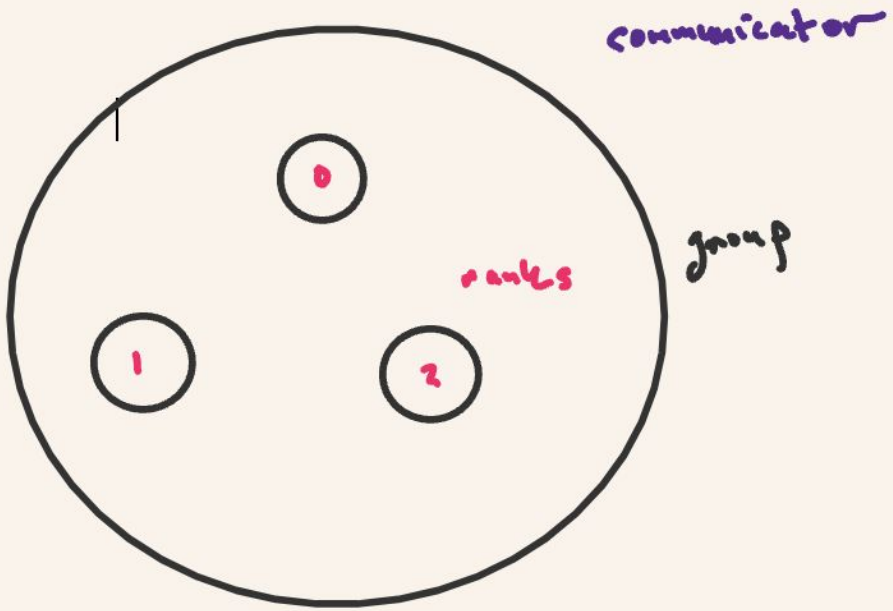
More on Communicators

```
"""
An MPI hello world program

Date: 05/13/2024
Author : Djamil Lakhdar-Hamina

"""
from mpi4py import MPI



def main():

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

First Program : Hello World

```
"""
An MPI hello world program

Date: 05/13/2024
Author : Djamil Lakhdar-Hamina

"""
from mpi4py import MPI


def main():

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

    print(f"Hello world! I am process {rank} and the size of my group size is {size}")


if __name__ == "__main__":
    main()
```

More on Communicators



What is happening ?

- Program gets distributed to each processor, and then it runs in each processor.
- result is printed to standard output. First, we create a communicator object.
- Within python this COMM_WORLD is an object, a class, and it has its *methods* and its *attributes*. We use these methods and access these attributes to communicate between processes and to perform computations.
- We accessed the rank attribute and the size attribute via, respectively, **Get_rank()** and **Get_size()** and then we printed that information for each rank, for each process.

Point-to-Point Communication

- **Point-to-point communication** is very simply communication between two processes, to the exclusion of others. It is a binary-relation , and this form of communication will be counterposed to **collective communication**.
  - Initially it is implemented by **send(),recv(), and sendrecv()**

Point-to-Point Communication

- Now how can we communicate *between* processes, we use the *send* and *receive* functions, which are methods of communicators.
- Let us say that we designate process 0 as a “head” process, and that the rest of the processes send out some data to the head, this is how we would accomplish this:

Point-to-Point : Send

- Send function signature:

```
(method) def send(
    obj: Any,
    dest: int,
    tag: int = 0
) -> None
```

Point-to-Point : MPI.ANY_SOURCE

```
if rank != 0:
    data = randint(LOWER_LIMIT, UPPER_LIMIT)
    comm.send(data, dest=0)
else:
    for r in range(1, size):
        data = comm.recv(source=MPI.ANY_SOURCE)
        print(f"my name is rank 0 and I received the number {data}")
```

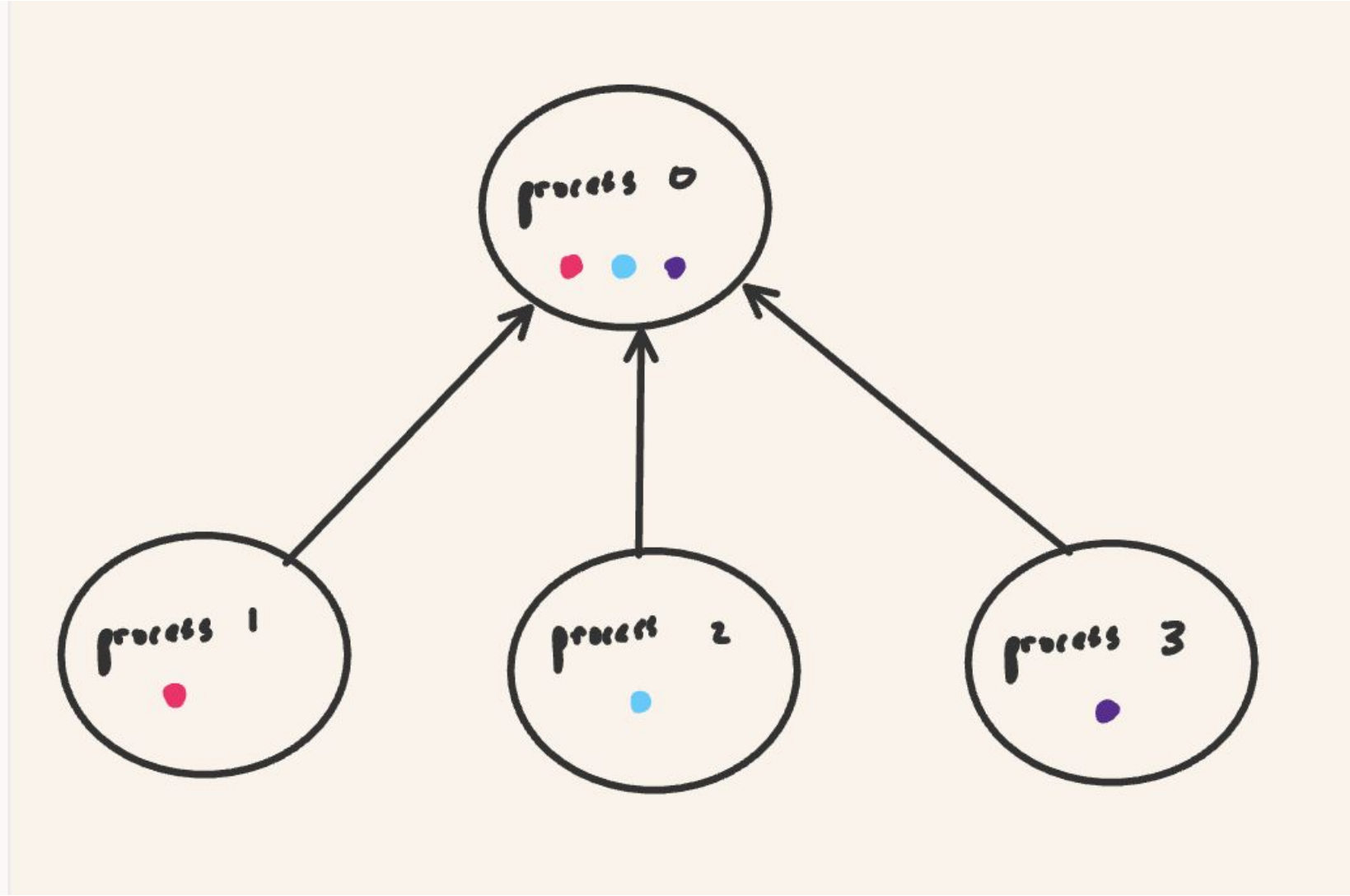
MPI.Status()

- The **MPI.Status()** object , passed to a receive function , tells us information about the point-to-point communication
- That object has a number of attributes, such as byte_count, and error status.
- Set information via methods or access information via attributes, above we show how to access information about source,tag, and byte count of the message.

MPI.Status()

```
status = MPI.Status()

if rank != 0:
   data = randint(LOWER_LIMIT, UPPER_LIMIT)
   comm.send(data, dest=0 , tag=1)
else:
   for r in range(1, size):
       data = comm.recv(source=r, status=status)
        print(f"my name is rank 0 and I received the number {data} from rank
{r}")
           print(f"The status object : \n source: {status.source} \n tag:
{status.tag} \n byte_count : {status.count}")
```

Depiction of Program



Blocking and Non-Blocking

- send-receive in this form are examples of **blocking-communication** (to be distinguished later from **non-blocking** communication).
- That means once a process sends a message, no more code is executed within that process, instead the process produces a lock, and the process will not resume computation *until* another process receives the message (at which point it unlocks)

Sendrecv

We can couple the send and receive commands in a nifty command **sendrecv** function whose function signature is:

```
(method) def sendrecv(
    sendobj: Any,
    dest: int,
    sendtag: int = 0,
    recvbuf: Buffer | None = None,
    source: int = ANY_SOURCE,
    recvtag: int = ANY_TAG,
    status: Status | None = None
) -> Any
```

Program with sendrecv

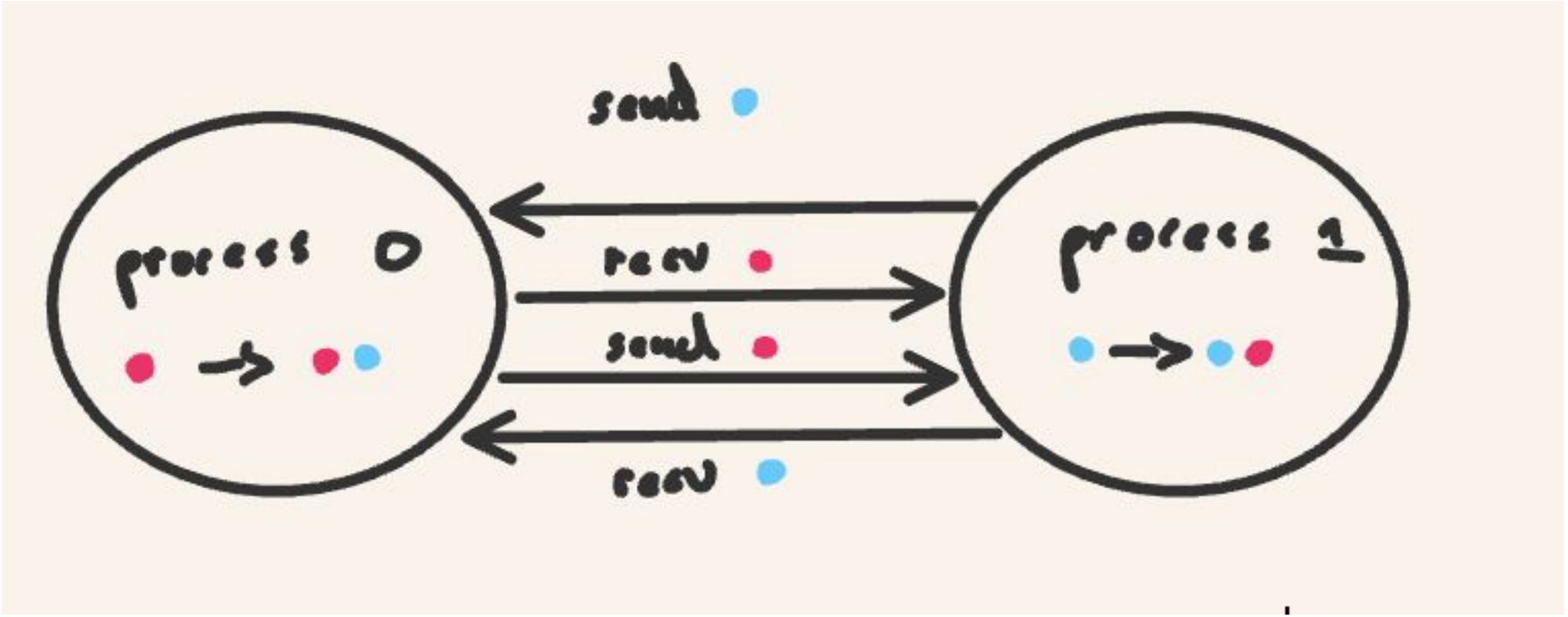
AFRICAN
CENTERS
OF EXCELLENCE
IN BIOINFORMATICS &
DATA-INTENSIVE SCIENCE

Excercise 1: Ring-Program

Write a ring-program. This program passes a token around processors in order and writes out output like:

```
token is at 0 and is 0
token is at 1 and is -1
token is at 2 and is -1
token is at 3 and is -1
token is at 4 and is -1
token is back at 0 and is -1!
```

Example Program:



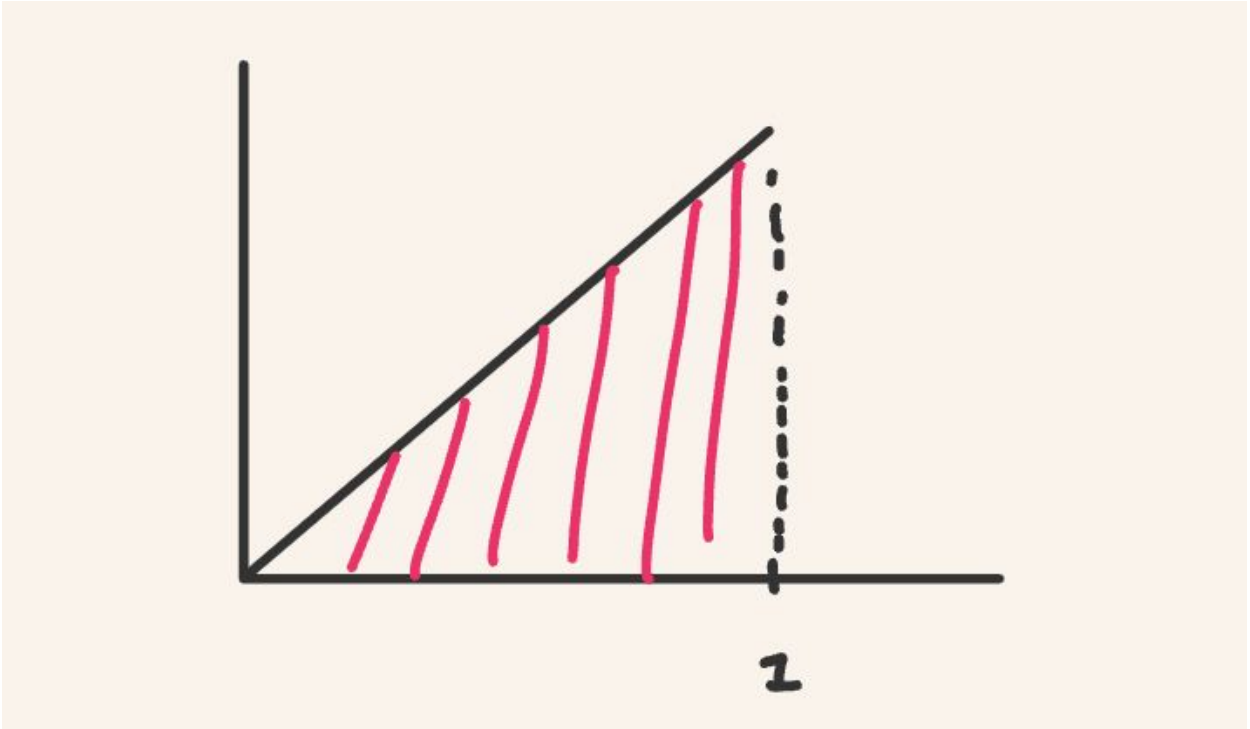
Application: Integrals and Riemann Sums

- The calculus of infinitesimals, or just calculus, begins and is built on two mathematical operations derivative and the integral
- Now for our purposes the *integral I of a function f(x) between points (real numbers) a and b is simply the area or volume under the curve between a and b*. The mathematical statement above is expressed notationally as:

$$\int_a^b f(x)\ dx$$

Integrals and Riemann Sums : Example

$$\int_0^1 x \, dx$$

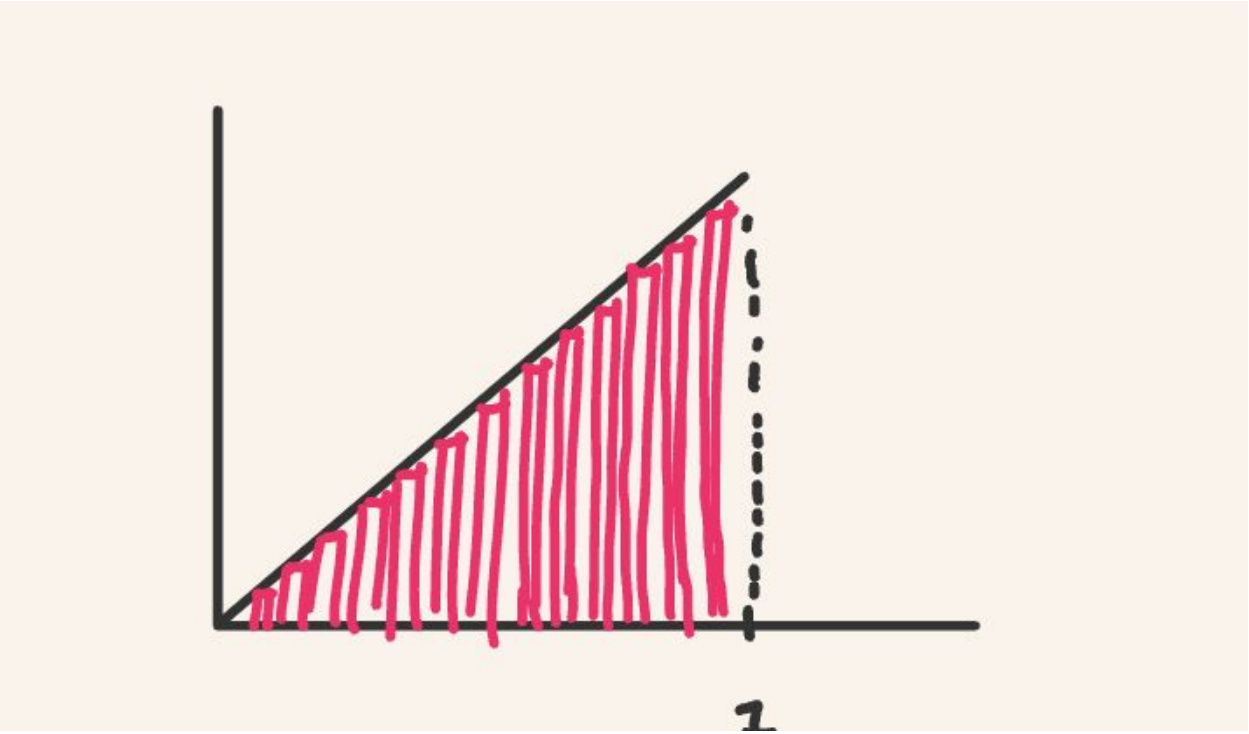


Riemann Sum

- Now there are symbolic rules to calculate these integrals
- But we can also *approximate them* with just summation and multiplication

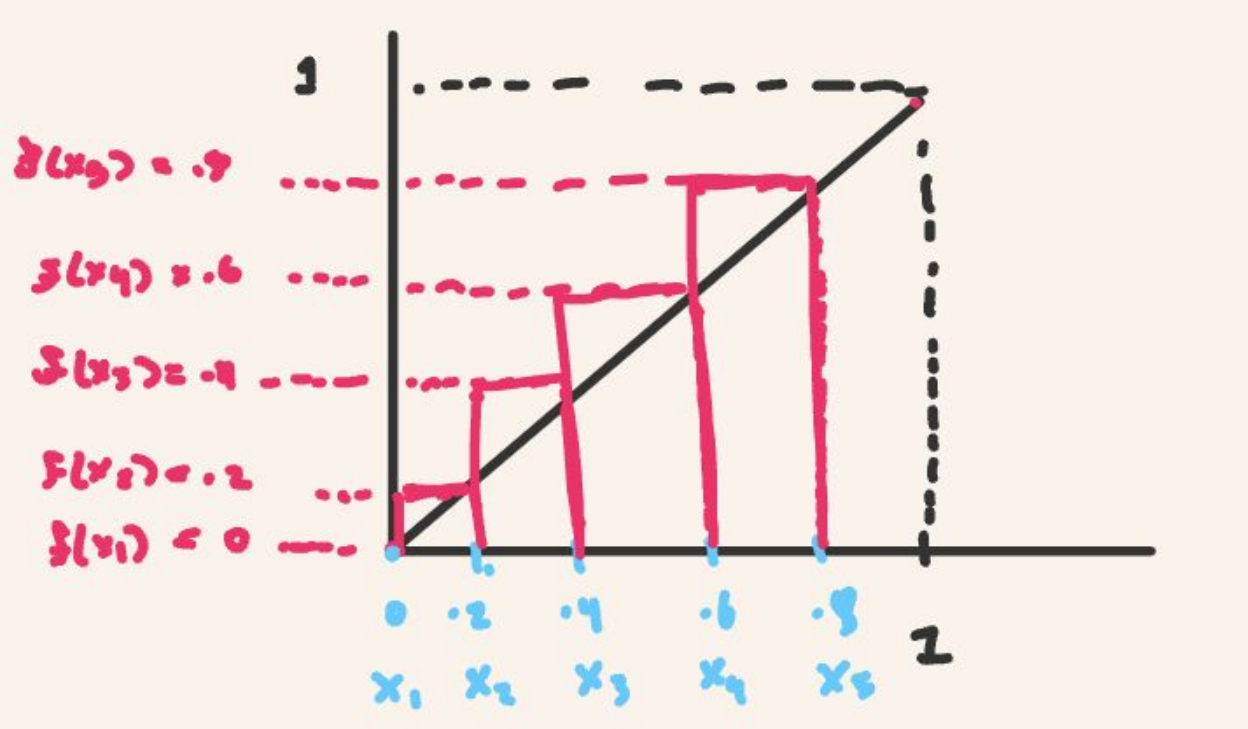
Riemann Sum

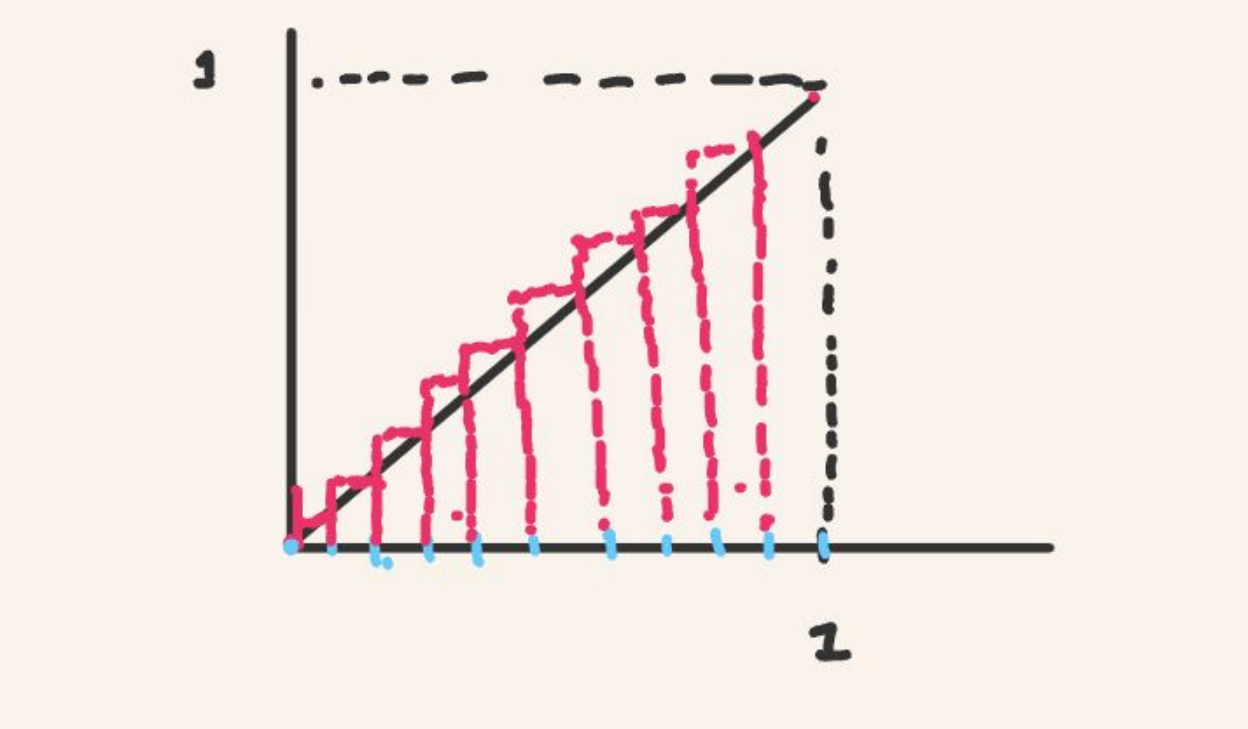
- Let us choose there to be n rectangles.
- Now each rectangle is a **partition** and we label it n1,n2...n.
- Each x is placed in intervals of x=(b-a)/n and we label those x's x1,x2...xn.
- The height of the rectangle is then f(xi) for an arbitrary point i. Then the area of an arbitrary rectangle is f(xi)* x. We then sum up all these rectangles and express this notationally as :



Riemann Sum

$$\int_{a}^{b} f(x)\ dx \approx \sum_{i=1}^{n} f(x_i) * \Delta x$$


1
0
.2
.4
.6
.8
1
x_1
x_2
x_3
x_4
x_5

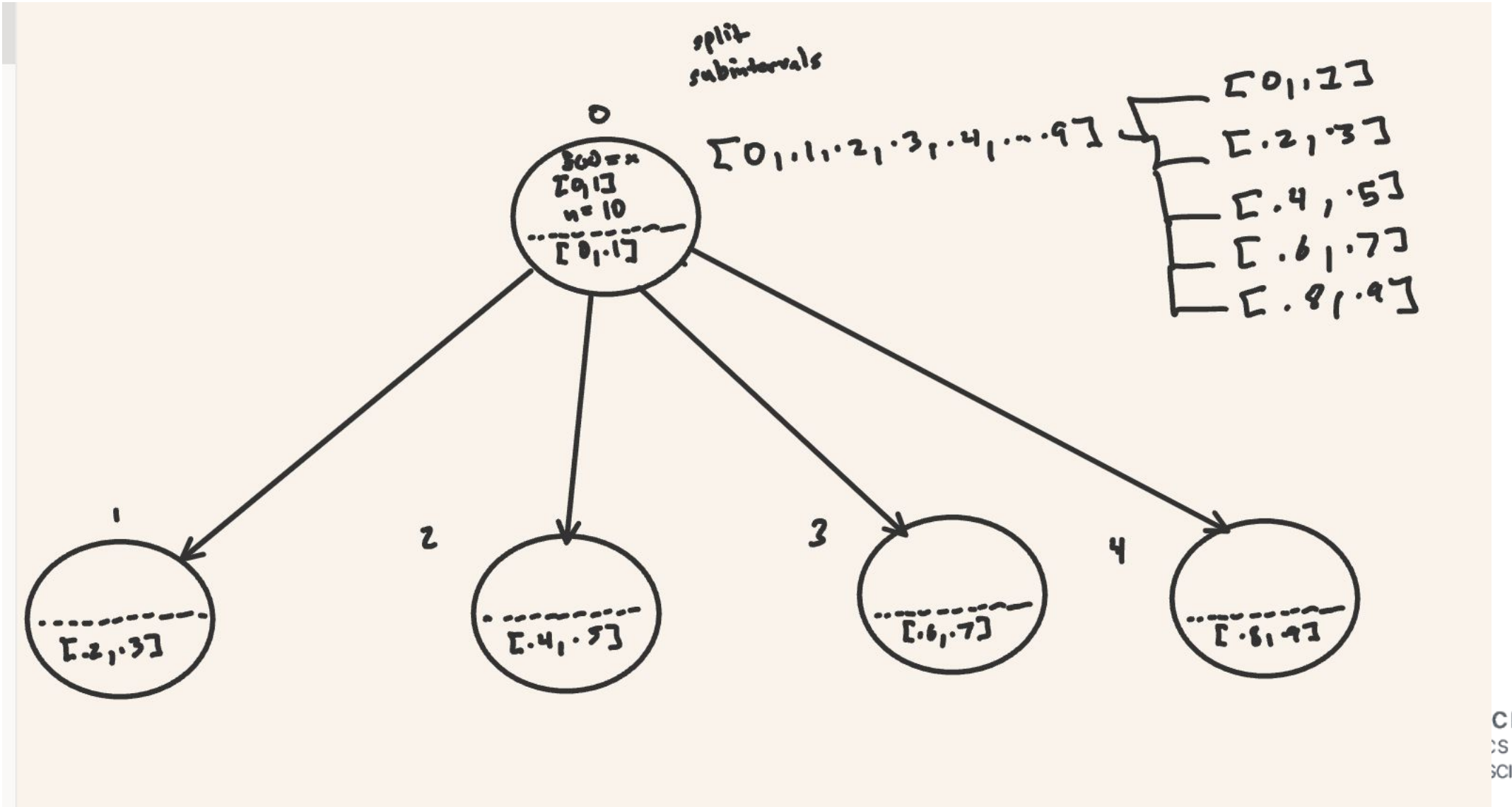

1
1

Serial Version of Program

AFRICAN
CENTERS
OF EXCELLENCE
IN BIOINFORMATICS &
DATA-INTENSIVE SCIENCE

How Can we Parallelize ?

- The simplest answer is by deploying a single-instruction, multiple-data framework
- Perform a *data-partition for a task across multiple processes*.
- None of the various computations of the areas depend on one another


split
subintervals
0
[0,1]
[0,.1,.2,.3,.4,...,.9]
[.2,.3]
[.4,.5]
[.6,.7]
[.8,.9]
1
2
3
4

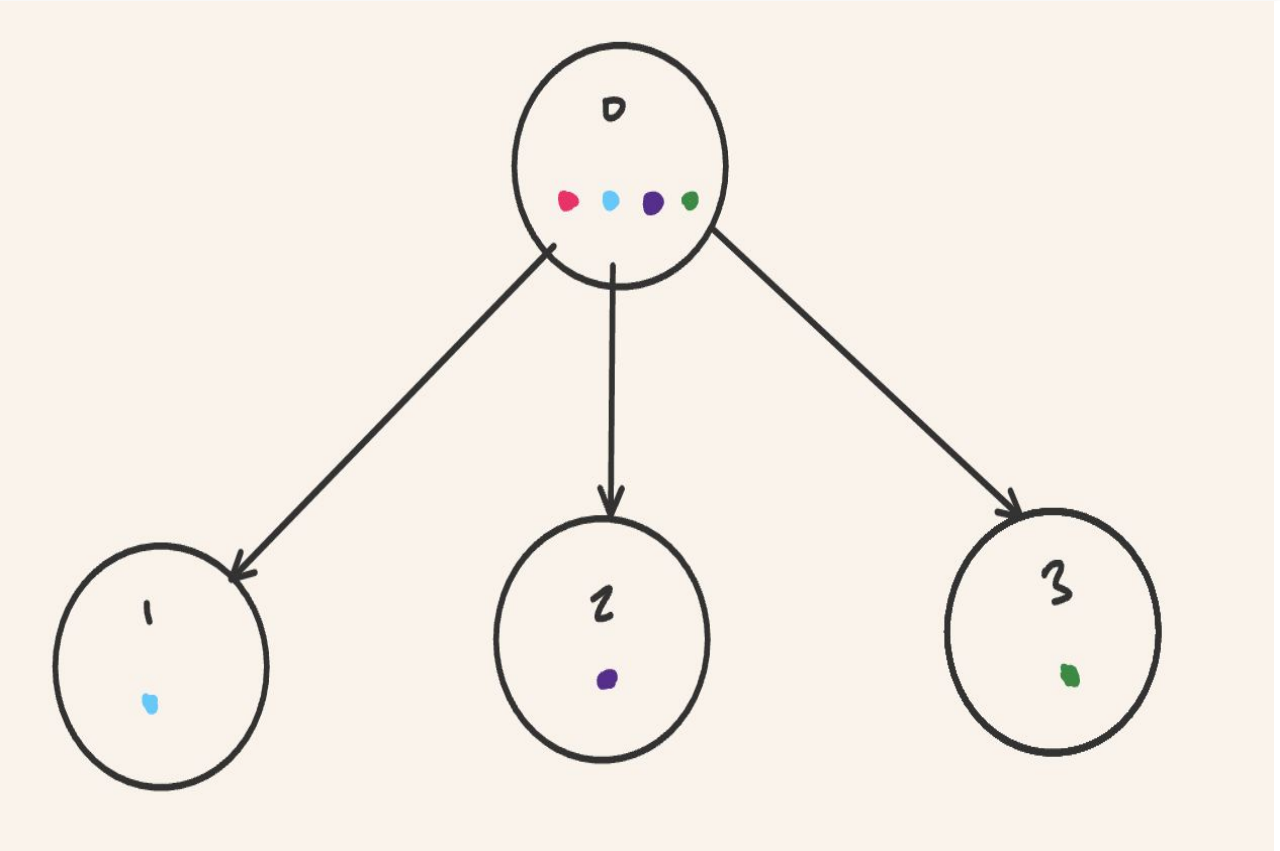
Break time : 10 minutes

Collective Communication

- Now we will develop the concept of **collective communication**
  - communication which involves *all* processes,
- Whereas send and receive binary (send a message from one process to another process),
  - functions which involve all the processes in the communicator.

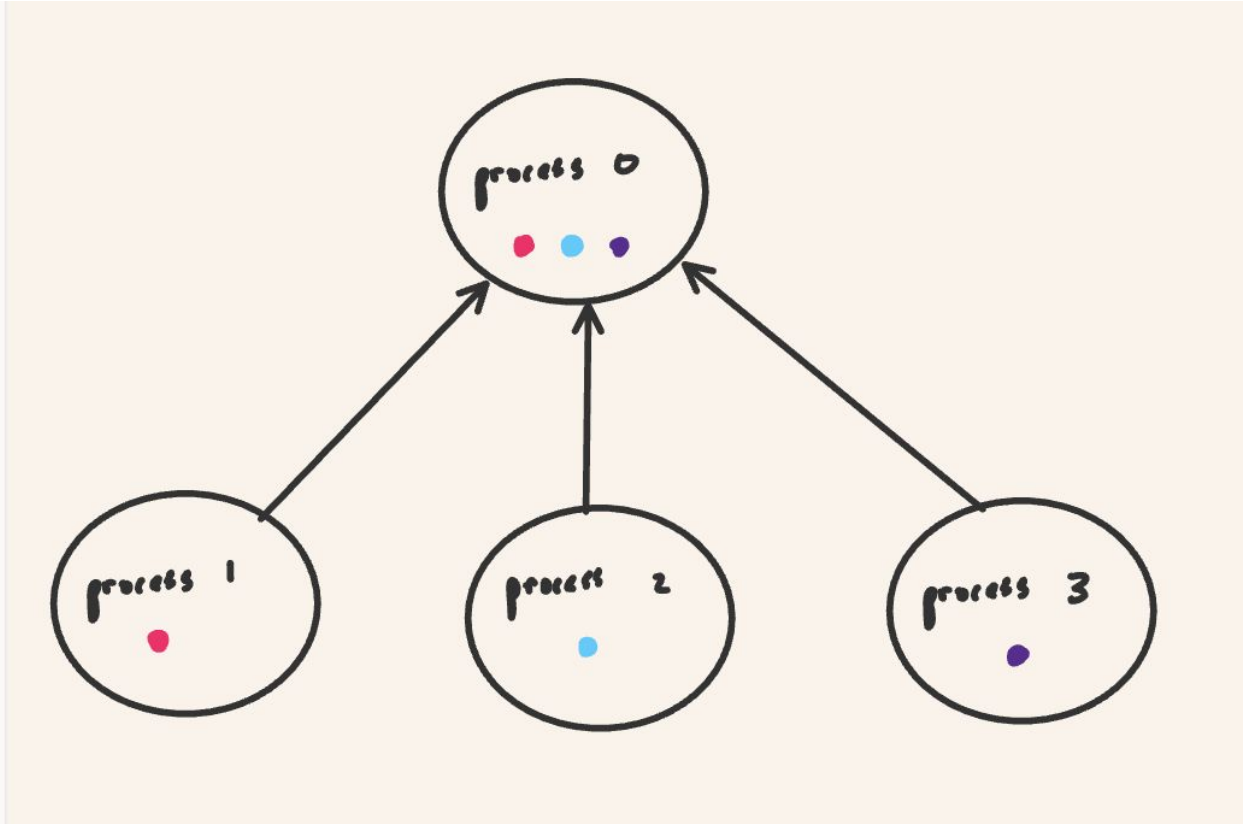
Scatter-gather

Say we wanna break up some data , and send different parts of the data i.e. a process called **sharding** the data.



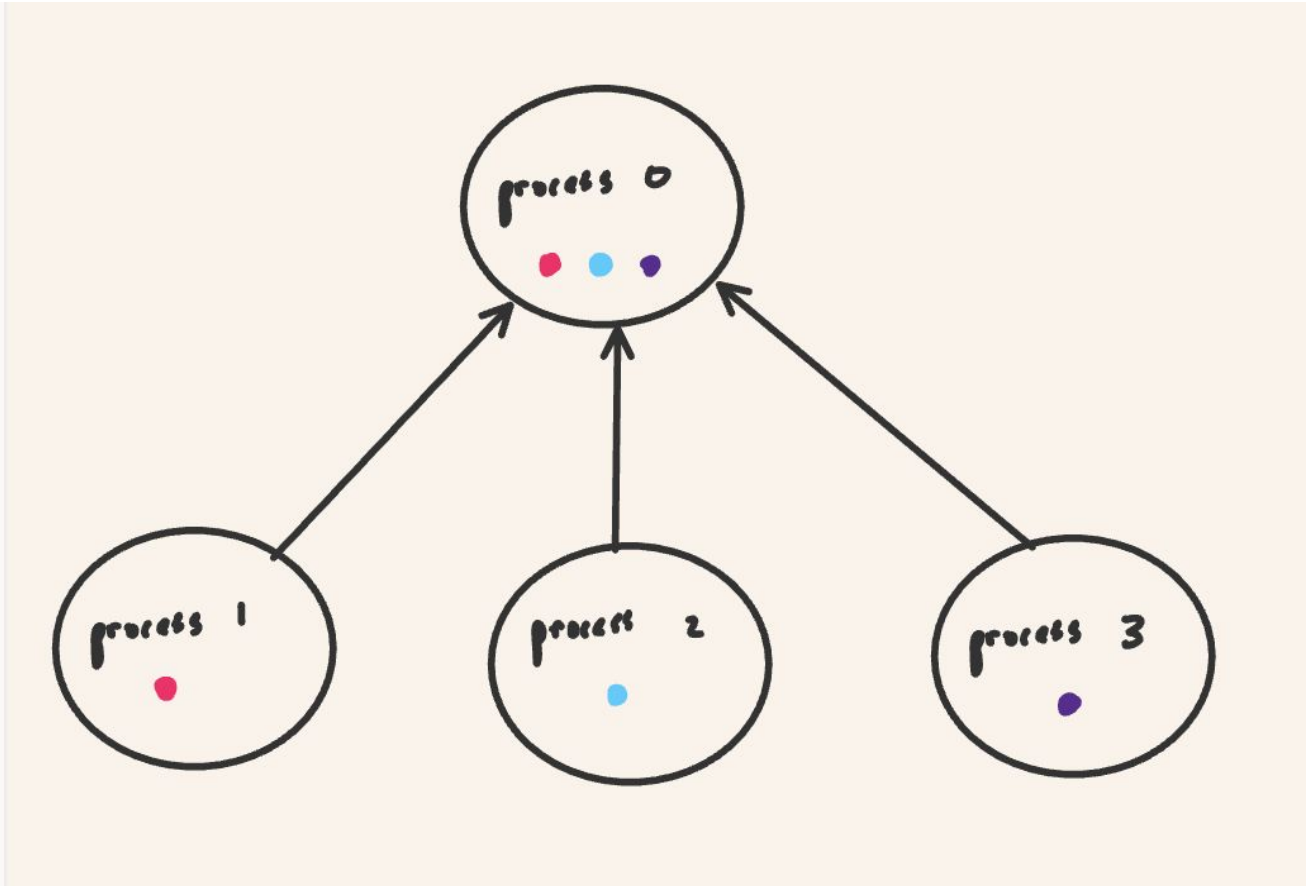
Scatter-gather

Before we had to write a loop, and use the send function from rank 0 to the other processes then each process had to receive the data i the form of a message from rank 0 (like in our Riemann sum program). This is a rather verbose way of doing things. Why not have *a single* command that sends out the relevant piece of data to each process? This command is **scatter.**



Scatter-gather

Now let us say we want to bring back and bundle that data into some object e.g. say a list. The command that achieves this is **gather**.



Signatures of Scatter-Gather

```
(method) def scatter(
    sendobj: Sequence[Any],
    root: int = 0
) -> Any
```

```
(method) def gather(
    sendobj: Any,
    root: int = 0
) -> (list[Any] | None)
```

Basic Program:

```
from mpi4py import MPI


def main():

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

    foo = [1, 2, 3, 4, 5]

    local_foo = comm.scatter(foo, root=0)
    print(f"Sent {local_foo} to {rank}")
    new_foo = comm.gather(local_foo, root=0)
    if rank == 0 :
        print(f"Received {new_foo}")


if __name__ == "__main__":
    main()
```

Application: Vector-Vector Addition

Now that we have a very basic understanding, let us show how scatter-gather works via a relatively simple example, the addition of two vectors, then we will rework our riemann sum.

Application: What Is a Vector?

A vector mathematical object can be represented as a list of numbers where each number is an **element** of the vector
The order of the elements matters, two vectors with the same elements but different orders are *different vectors*.
When we add those vectors, they have to have the same **dimension**, or number of elements. We then add those elements **element-wise** e.g. [1,2,3] + [1,2,3] = [2,4,6].
How would we parallelize this vector addition?

Application: Vector-Vector Addition

Strategy:

1. Break vectors into sub-vectors
2. Scatter vectors
3. Add sub-vectors
4. Gather vectors

Broadcast

It sends *a* piece of data to *all* the processes within the communicator.

Using Broadcast in Riemann Sum

```
# Distribute the info and sub-interval across processes.
# Receive the info and compile function
if rank == 0:
    local_interval = batched_x[0]
    local_sum = riemann_sum_left(f, local_interval, delta_x)
    for i in range(1, size):
        info = (batched_x[i], function_string, delta_x)
        comm.send(info, dest=i)
else:
    local_interval, function_string, delta_x = comm.recv(source=0)
    f = create_function_on_process(function_string)
```

```
function_string, delta_x = comm.bcast((function_string, delta_x), root=0)
```

Exercise 2:

Objective: Distribute parts of a string to multiple processes, have each process compute the length of its assigned part, and gather the lengths in the root process.

Steps:

1. Initialize MPI.
2. Create a long string in the root process.
3. Scatter the parts of the string to all processes.
4. Each process computes the length of its assigned part.
5. Gather the lengths from all processes in the root process.
6. In the root process, compute the total length and print it.

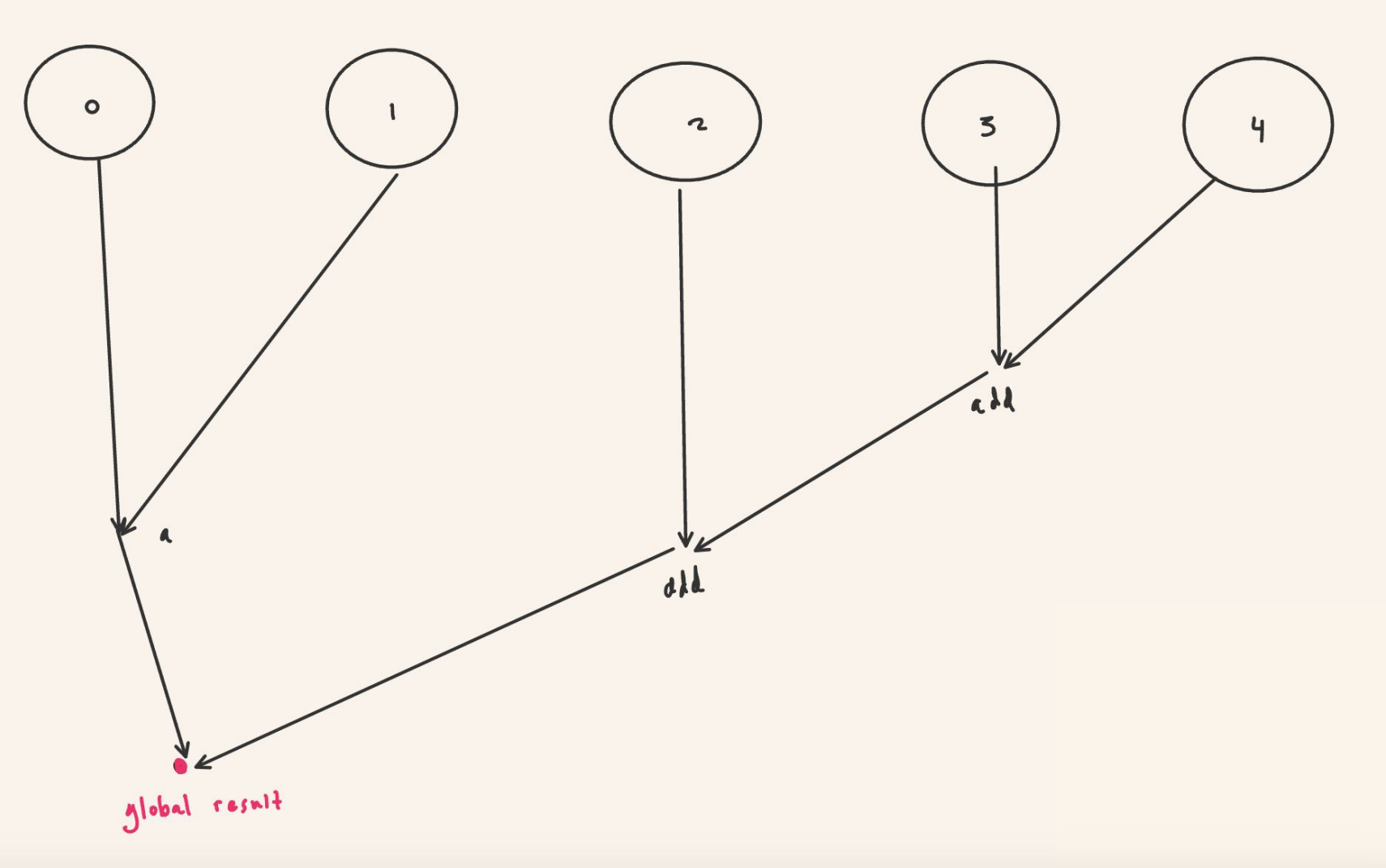
Reduce: Why Reduce?

Let us say that we have some data scattered across various processes and we wish to somehow take them all and perform an operation between them. Before we could send the data from all the various processes, and then sum them on process 0. The problem is that that is inefficient because *only one process is doing the operation*. How can we improve the performance of such a send paradigm by having multiple processes perform the operation?

Reduce: Tree-like Communication

Say we have a bunch of numbers scattered across processes and we wish to add those numbers and get one global sum. We could resort to a **tree-like** pattern of communication to improve on performance.

Reduce: Tree-like Communication



Reduce

- Would be very difficult to solve this problem
- We leave details to programmers of MPI and utilize **reduce()** with signature:

```
(method) def reduce(
    sendobj: Any,
    op: Op | ((Any, Any) -> Any) = SUM,
    root: int = 0
) -> (Any | None)
```

Basic Example:

```
from mpi4py import MPI


def main():

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

    LOWER_LIMIT = 1
    UPPER_LIMIT = 10

    randomn_int = randint(LOWER_LIMIT, UPPER_LIMIT)
    print(f"rank {rank}: {randomn_int}")
    global_sum = comm.reduce(randomn_int, op=MPI.SUM, root=0)

    if rank == 0:
        print("the global sum is:", global_sum)


if __name__ == "__main__":
    main()
```

Custom Reduce

We can actually define our own instances of the **Op** class , we can define our own user-defined operations!

Op is a class defined in MPI. It has a method **Create()** whose signature is:

```
(method) def Create(
    function: (Buffer, Buffer, Datatype) -> Buffer,
    commute: bool = False
) -> Op
```

Exercise 3: **Broadcast and Reduce**

Objective: Calculate the sum of an array where each process receives the same array using broadcast, and then computes the sum of its elements. Finally, reduce the sums from all processes to get the total sum.

Example : Magnitude

$$M = \sqrt{a^2 + b^2 + c^2 .... n^2}$$

So for our example we have:

$$M = \sqrt{2^2 + 2^2 + 2^2 + 2^2} = \sqrt{4 \times 2^2} = 4$$

Back To Riemann Sum

We can continue to improve our riemann sum program. When we perform the summation on process 0 we had to send the data on each process to 0 then add on 0. As stated above this is not the most efficient means of performing the global summation and we can also make our script much more succinct by using:

```
local_interval = comm.scatter(batched_x)
local_sum = riemann_sum_left(f, local_interval, delta_x)
global_sum = comm.reduce(local_sum, op=MPI.SUM)
```

Advanced: Matrix-Multiplication

What is a matrix?

A matrix is a mathematical object , a block of numbers, or a list of vectors e.g.:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Matrix-Multiplication

Now how would we multiply two of these objects? Let us go through a specific example and then generalize. Take the matrix above and multiply by itself:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Matrix-Multiplication

- First we must know the dimensions.
- There are three rows and three columns , 3x3.
- If there were m rows and n columns the dimensions would be mxn.
- Now to multiply two matrices the dimensions must satisfy certain conditions
  - if matrix A has dimensions mxr and B sxn then A*B = C dim mxn if and only if r=s.
- Now very importantly two vectors can be multiplied, and this is called the inner product. Take vector A=[1,2,3] and B=[4,5,6] then A*B= [1*4,2*5,3*6] = [4,10,18].

Matrix-Multiplication

- Above we have two 3x3 matrices so our new matrix is 3x3. Take row vector 1 multiply by column vector 1 then add all the elements up. This gives the entry [1,1]
- Now take row vector 1 and multiply by column vector 2, add all the elements up. This gives the entry [1,2].
- See the pattern ? For every row vector i and column vector j, we multiply the ith row by the jth column , add the elements

Some Ways to Parallelize

The Serial and Parallel Programs

Numpy Objects

- Distinction between methods relating to general python objects , which are written lowercase, such as **send**
- Those relating to memory or data buffers , such as ndarrays , which are written in uppercase such as **Send**

Numpy Objects for Users

The main difference from the point of view of the user is simply that in the case of the memory-buffer methods , the object sent or received or operated on is specified using a 2-3 value list or tuple:

```
[data, MPI.DOUBLE] or [data, count, MPI.DOUBLE]
```

In the former case, the first piece specifies the object itself and the second the data type. In the latter case, the number of items is explicitly stated in the **count** argument.

Sending and Receiving Buffers

Now generally there is a buffer that stores data to be sent, a sending buffer, and a buffer that stores data to be received.

Exercise 4:

Objective: Distribute parts of a byte-string to multiple processes, have each process reverse the string, then send back the the reversed strings to be gathered back up.

Steps:

1. Initialize MPI.
2. Create a long byte-string in the root process.
3. Turn this long byte-string into a numpy array.
4. Scatter the parts of the string to all processes.
5. Each process computes the length of its assigned part.
6. Gather the lengths from all processes in the root process.
7. In the root process, compute the total length and print it.

Input-Output (IO) in MPI

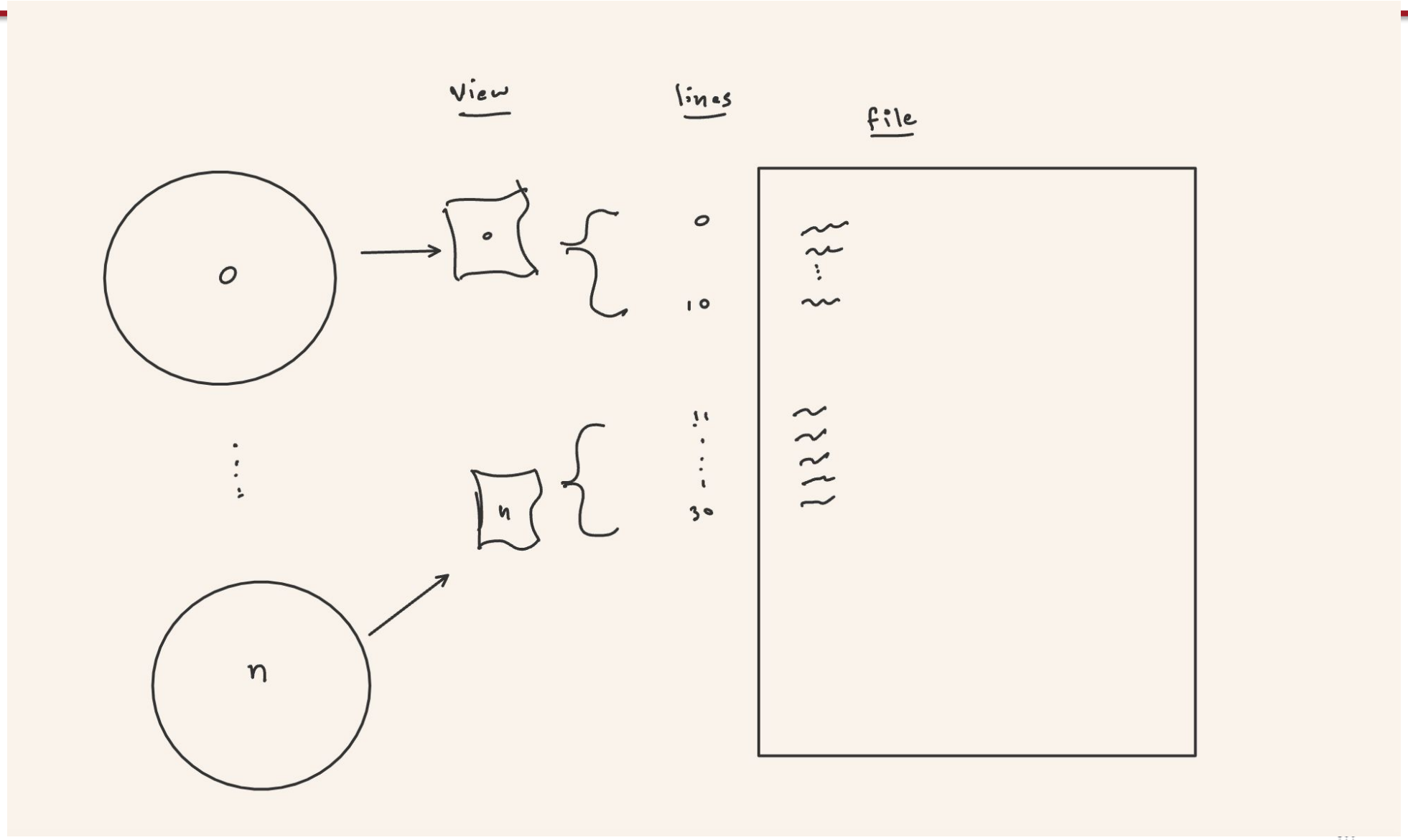
Here we will delve into the limited, but powerful, capabilities of IO in mpi4py.

In general, modern systems deploy what is called the POSIX standard for file systems. However, POSIX doesn’t provide a model for efficient parallel IO. With the release of the MPI-2 standard, MPI sets up an efficient and optimized interface for parallel IO.

File Class

"In *mpi4py*, all MPI IO operations are performed through instances of the **File** class. File handles are obtained by calling the **File.Open** method at all processes within a communicator and providing a file name and the intended access mode. After use, they must be closed by calling the **File.Close** method. Files even can be deleted by calling method **File.Delete**. After creation, files are typically associated with a per-process *view*. The view defines the current set of data visible and accessible from an open file as an ordered set of elementary data types. This data layout can be set and queried with the **File.Set_view** and **File.Get_view** methods respectively."

Process, File, View



Point-to-Point

There is within IO for MPI, a distinction between point-to-point and collective io-operations. The point-to-point pattern of communication necessitates the use of **Read** and **Seek** to write from multiple processes to a file.

IO Modes

1. MPI_MODE_RDONLY
   - Open the file for read-only access.
2. MPI_MODE_RDWR
   - Open the file for both reading and writing.
3. MPI_MODE_WRONLY
   - Open the file for write-only access.
4. MPI_MODE_CREATE
   - Create the file if it does not exist. This mode is often used in conjunction with MPI_MODE_WRONLY or MPI_MODE_RDWR.
5. MPI_MODE_EXCL
   - Error if the file already exists when MPI_MODE_CREATE is also specified. This mode ensures that a new file is created and an existing file is not overwritten.
6. MPI_MODE_DELETE_ON_CLOSE
   - Delete the file when it is closed.
7. MPI_MODE_UNIQUE_OPEN
   - Ensure that the file will not be concurrently opened elsewhere.
8. MPI_MODE_SEQUENTIAL
   - File will only be accessed sequentially.
9. MPI_MODE_APPEND
   - Sets the file pointer to the end of the file before every write operation. This mode ensures that new data is appended to the end of the file.

Collective

- The main difference from the point of the user is that we do not need to specify the **Seek** operation
- We utilize collective functions like:
  - **Write_at_all** , **Read_at_all**
  - **Write_at**, **Read_at**
  - **Write_ordered**, **Read_ordered**

Collective Non-Contiguous

We mentioned that a file and a view in MPI-IO is in fact a series of ordered elementary data types. The following code utilizes this “lower-level” fact to write and read data from a numpy buffer to a file in parallel and in a non-contiguous manner.

Conclusion

We learned basics of mpi and we combined it with numpy.

Tomorrow we will work on mpi, numpy, and slurm and you will be able to use the supercomputer successfully.

Thank you and good job!!!