**On Scientific, Parallel, and High-Performance Computing**

When one first learns to program a computer, one writes programs that operate according to a *serial* model of computation (although the underlying hardware and software we use does not strictly follow this computational model) i.e. we write a program that executes line by line, the one after the other, one computation at a time. In this workshop we will all work to break out of this paradigm and think in terms of a parallel model of computation.

## 0. Preface , Prerequisites

This series of notes

## 1. Methodology of Parallel Computing

Before we go into some technicalities about architecture, memory, and specific parallel programming frameworks we need to understand how to think in parallel in general.

## 1.1 Flynn's Taxonomy: How to Classify Parallelism

Now a computation consists of an *instruction* carried out on *data*. In the serial model, there is one thing , a processor that takes an instruction and acts on data. In the parallel model there is no longer one processor, and many computations are happening in parallel, at the same time. How to think through what a *parallel* computation is?

A simple answer is provided by Flynn's taxonomy. The basic serial computation takes one instruction and one single piece of data and does something. This is a single-instruction single-data computation (SISD).

There is another way to think through parallel computation. I call it the graph

## 1.2 Transforming Serial to Parallel

## 1.3 Challenges in Transforming Serial to Parallel

. Serial Hardware and Software

The history of the digital computer is much richer than the IBM, silicon-valley version of history spoon-fed to the public.

The digital computer as we know it grows intellectually out of an age of major mathematical and scientific revolution, out of the Hilbert program and out of advances in quantum and classical physics.

Alan Turing proposed in a famous paper in 1936 "ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM" , an abstract mathematical model - latter called the Turing Machine- which models computation and lays the theoretical foundation of the *stored-program* computer that was first developed (as far as I know) by von Neumann (

a major genius who established the rigorous mathematical foundations of quantum mechanics, who made major advances in mathematical logic, and who founded game-theory and helped found the discipline of computer science and engineering).

## 1.2 Von Neumann Architecture

The basic hardware model of the digital computer is the von Neumann architecture or stored-program architecture. It is a model that consists of three pieces i.e. main memory (in more modern parlance random access memory), central-processing unit (CPU) or processor, and interconnection between memory and CPU.

Main memory consists of a set of locations, which can store instructions and data. Every location has an address , which we use to access whats in the location, and instructions or data stored in the location.

The CPU consists of a control unit, and an arithmetic and logic unit (ALU). The control unit decides which instruction will be executed and the ALU executes the instructions. Data in the CPU and information about the state of the executing program are stored in special , very fast storage called registers. There is a special register termed the program counter and it stores the address of the next instruction to be executed.

Instructions and data are transferred between the CPU and the memory via the interconnect, traditionally a bus which consists of wires and some hardware controlling access to the wire.

Now the lifetime of a computation. When data or instructions are transferred from memory to CPU we say that the data or instructions are fetched and read from memory. When data is transferred from the CPU to memory the data or instructions are written to memory or stored. The separation of memory and CPU is called the von Neumann bottleneck. Why bottleneck? Imagine that the CPU is a factory, that builds stuff, say lamps, and the memory is a warehouse that stores the

resources needed to make stuff , the lamps, and also stores the lamps themselves. Then the interconnect is the highway that allows us to bring stuff from the factory to the warehouse. If the rate at which we produce stuff is much higher than the rate at which we can transport the stuff to and from the warehouse, then there will be a traffic jam and the amount of lamps we get back to the warehouse (in order to ship off to sell) will be limited by the time of transport to and from the warehouse.

To increase performance then we need a way to circumvent this bottleneck. But first we need to understand how more modern uniprocessor systems operate.

Note: this is a very barebones version of the von Neumann architecture. Additionally today we have many more components including a secondary-storage device, second memory, a disk, which is much slower to access to write to then RAM. We also have many different kinds of processing units including a floating-point unit an FPU.

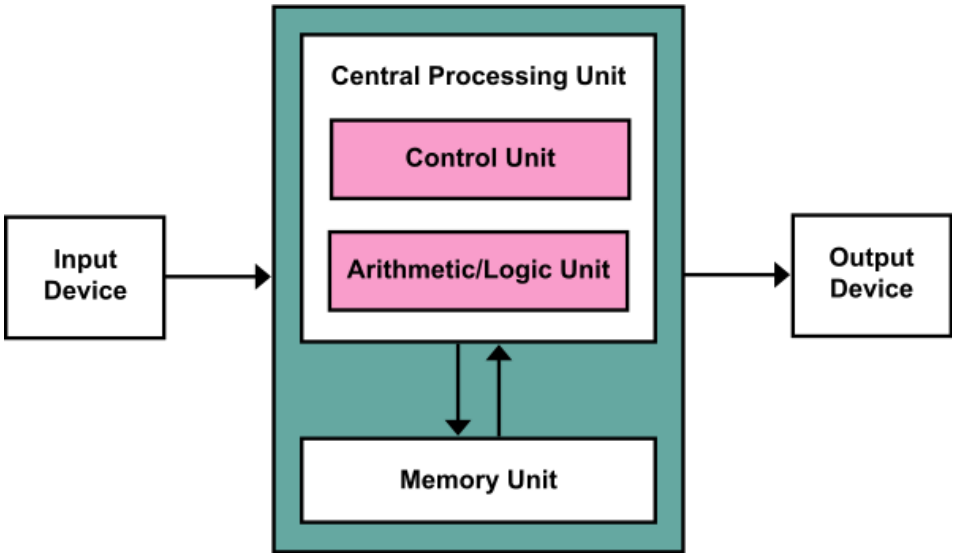



*Fig 1: figure of the Von Neumann Architecture*

## 1.3 Modifications to Von Neumann architecture

### 1.3.1 Processes, multi-tasking, and threads

An operating system or OS is a piece of software which manages hardware and software resources on a computer. It determines which programs run and when they can run, it manages and allocates memory to programs and access to peripheral devices such as harddisks, network interface cards, and keyboards and mice.

When a user runs a program , the OS creates a process- an instance of a program being executed. The process consists of:

1. The executable machine language program
2. A block of memory , which includes the code itself, a call stack that keeps track of active functions, and a heap, which is dynamically allocated memory during execution of a program.
3. Descriptors of resources such file descriptors
4. Security information
5. Information about state of process

Most modern systems are multitasking. The system provides support for the *apparent* simultaneous execution of multiple programs. Even on a single core a process runs for a small amount of time called a time slice. Then the OS can run a different program. A multitasking OS an change process many times.

Now in a multitasking OS, sometimes a process needs to wait on a resources - it will block, or stop. The OS can then run another process. However many programs can continue to do other useful work even though part of the program being executed is blocked. Let us say I am downloading a song from Spotify, I can still say look up a musician while it is downloading. Threading is a mechanism to accomplish just this, it provides a mechanism for programmers to divide their programs into independent tasks, so that when one thread is blocked another can run. Threads are usually contained within the same process, so they share the same executable, and share the same

memory and I/O devices. When a thread is started it forks off the process, and when it terminates it joins the process.

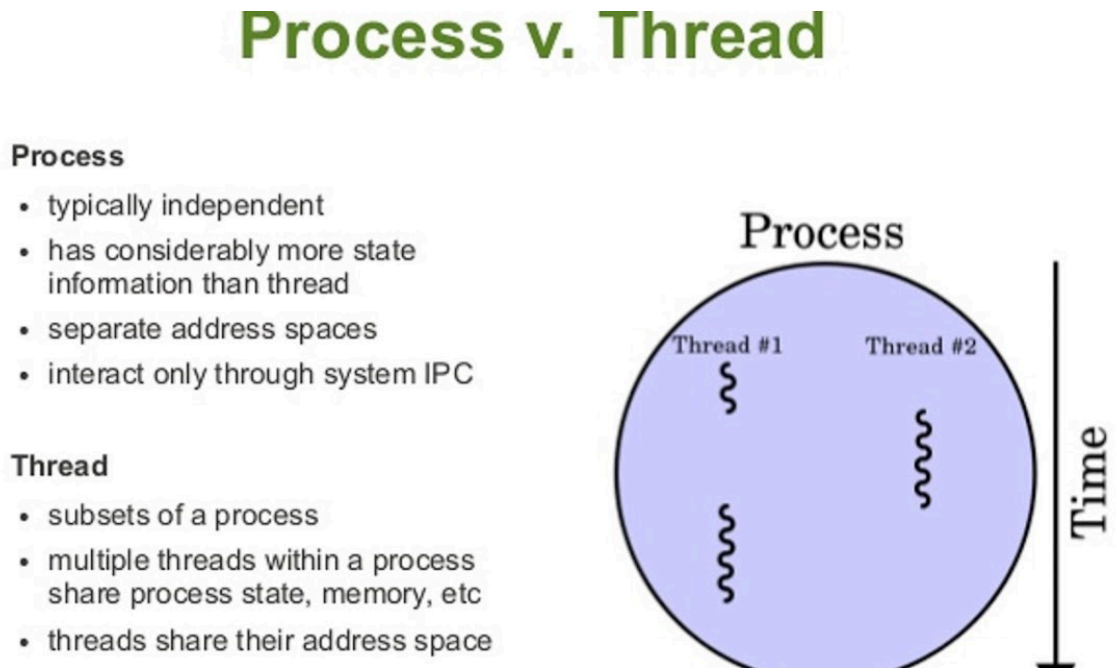



*Figure 2. Difference between Process and Thread , Process forking into multiple threads then joining after time slice*

*A socket is the physical socket where the physical CPU capsules are placed. A normal PC only have one socket.*

*Cores are the number of CPU-cores per CPU capsule. A modern standard CPU for a standard PC usually have two or four cores.*

*And some CPUs can run more than one parallel thread per CPU-core. Intel (the most common CPU manufacturer for standard PCs) have either one or two threads per core depending on CPU model.*

*If you multiply the number of socket, cores and threads, i.e. 2*6*2, then you get the number of "CPUs": 24. These aren't real CPUs, but the number of possible parallel threads of execution your system can do.*

### 1.3.2 Caching

Caching is a technique to address the bottleneck. Remember the factory example. How can we speed up the process of getting stuff back to our warehouse to ship out? Well, we could widen the road between the factory and the warehouse. We could also combine the factory and the warehouse together in one place. Caching in many ways is a combination of these solutions. We transport more instructions and data with better interconnections , and we can store instructions and data in special memory that is closer to registers than the CPU.

A cache is a collection of memory locations that can be accessed in less time than other memory locations, notably main memory.

Now how to decide what goes into these caches ? The principle of locality determines the answer to this question. The reasoning is that programs tend to use data and instructions that are physically close to recently used data and instructions. After executing an instruction, programs execute the next instruction, the programs tends to not jump around. Similarly after a program accesses one memory location, it tends to access one physically nearby. This reasoning is dubbed or codified in the *principle of locality*.

To exploit this principle of locality the system uses an effectively wider interconnect to send blocks of data and instructions called cache blocks. Generally, the cache is divided into levels , L1 is the smallest and fastest ,L2 .... LN which is the biggest and slowest. The OS system duplicates data and instructions to these caches which then is utilized during the execution of a program ( or not) . The CPU then works down the cache hierarchy, it checks if say some data is in L1, then L2 then on and on. When a cache is checked for information and the information is there that is called a cache hit. When it does not find the information that is called a miss.

Now just like with main memory, we can read and write for caches. Say we over-write some data in the L1 cache, we change the data that went from main memory to L1. Now there is an inconsistency between the data in L1 and main memory. How to deal with this? There are two basic approaches. In one, the write-through solution, the line is written to main after it is written to cache. In write-back, the data is marked

*dirty* , and when the cache line is replace by new data, the dirty line is written to memory.

Now when we program for performance, we have to make sure that we minimize cache misses, and maximize cache hits. We will come back to this latter.

### 1.3.3 Virtual Memory

Now in a multitasking OS many programs are running asynchronously. A program may access very large data sets , and all the instructions and data may not fit into main memory. How do we make sure that the many programs can all share main memory without each program corrupting the data and instructions of the other? Virtual memory is an answer to this problem.

Virtual memory operates off the principle that main memory can function as a cache for a much larger secondary storage. It keeps in main memory only those active parts of the many running programs hence leveraging spatial and temporal memory. The idle parts of the programs are kept in a block of secondary storage called **swap space.**

When the program wants to access a *physical* address there is a piece of hardware called the Memory Management Unit (MMU), which maps a **virtual address** to a physical address, it looks up the physical address associated with the virtual address based on a **lookup table**. Let us say a program P1 runs , and let us say occupies has virtual address 0-100, and this virtual address 0-100 maps to physical addresses 100-200. Now let us say program P2 runs with address virtual addresses from 0-100, and this maps to physical addresses 200-300 in physical memory. We see both have the “same” virtual address but they map to completely different spaces in physical memory.

Another Example.  Let us say a program P1 runs and has two parts A and B. Let us say P1A occupies has virtual address 0-50, and this virtual address 0-50 maps to physical addresses 100-150. Now let us say program P2 runs with address virtual addresses from 0-100, and this maps to physical addresses 150-250 in physical memory. Now P1B runs

and it maps 50-100 to 250-300. We see that contiguity in virtual memory need not be respected in physical memory. The OS will decide based on efficiency and performance, a highly complex problem we will not get into.

All of this means that the two programs P1 and P2 , in fact any number of programs, will not interfere with the way each program interacts with main memory. It only works on its assigned chunk of main memory, and it "knows" this assignment because it has information in virtual memory.

Now let us say we have 1 million programs running, each takes 1 space, then in order to access the stuff in main memory, the OS needs to construct a lookup table with 1 million entries mapping 1 space in virtual to one space in main memory. This will take up a lot of space and the lookup itself a lot of time. In order to deal with this virtual memory operates on chunks of data and instructions, blocks of main memory, called **pages** which are relatively large typically ranging today from 4 to 16 kilobytes i.e. main memory is chopped up into blocks called pages. Now paging allows the program to go to the virtual address, the MMU then looks up in a **page table** the physical address and redirects the program to a page. This is illustrated in figure 3.

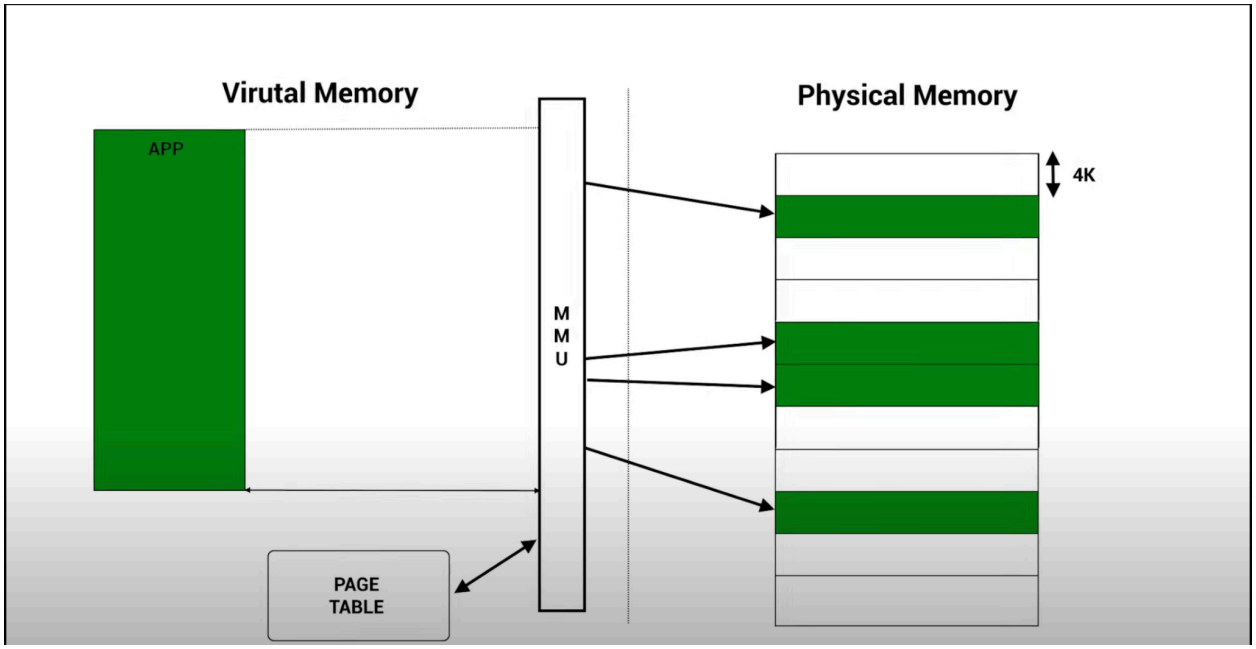


*Figure 3. The basic set-up of paging*

Now let us use an example. Look at figure 4. Say program P1 is assigned the 32 bits as shown in the top. Now the first 12 bits, the **offset-bits** , those on the right block in the portion labeled virtual address are copied directly from virtual address to physical address. The left 20 bits are acted on by the MMU, then looked up in the page table to find the physical address.

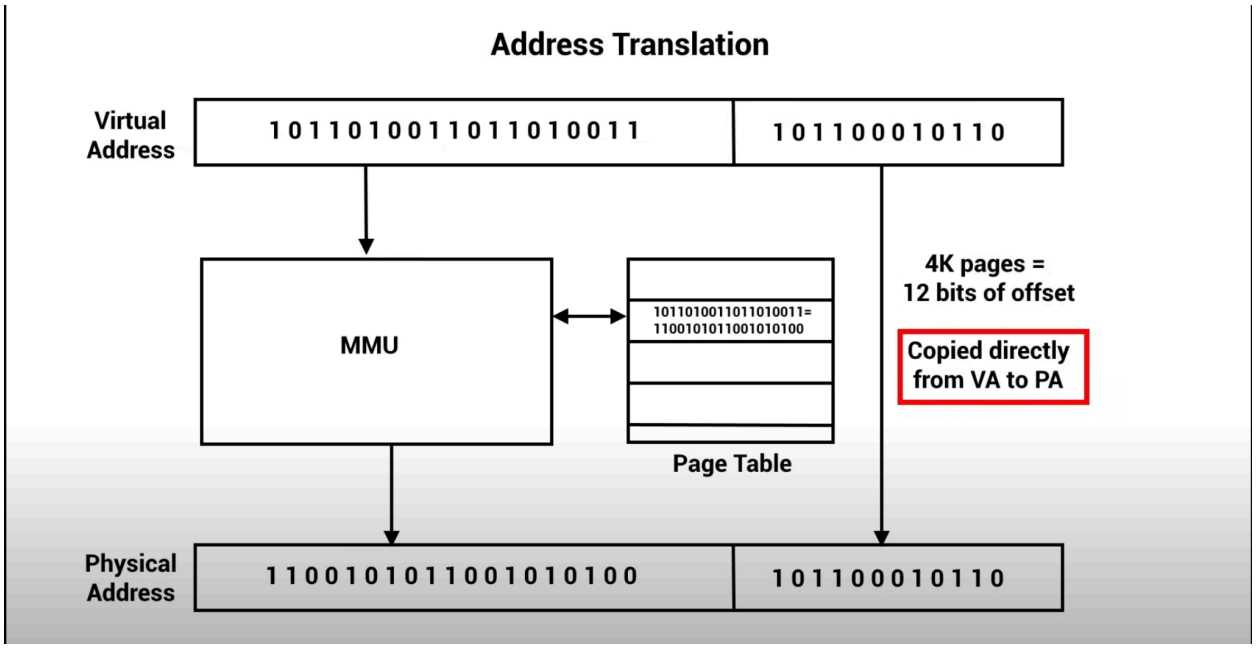



*Figure 4. How paging works using bits*

Now where are these tables actually contained? If it were contained in RAM it would take up too much space. There is another conundrum. In order for virtual addresses to map to physical addresses, the MMU needs to access physical ram, to find an entry in the table so it can then translate virtual address to physical address, and then access ram again! It looks like it accessed ram twice, just to accomplish an overall task that might have needed just one access to ram. How to escape this? Well processors typically have a special address translation cache, a **translation-lookaside buffer (TLB)**, a cache of recently looked up addresses i.e. it caches a small number of entries

from the page table into fast memory. We then expect that most of the memory reference will be to pages whose physical address is stored in the TLB, and the number of memory references that require access to the page table will be reduced. When we look for an address in the TLB but its there that is called a *hit* , when it is not a *miss*. Now if we attempt to access a page that not in memory and does not have a physical address, but only in say , secondary memory, then the failed access is called a **page fault**. We need to try to reduce misses, and we need to try to reduce page faults,

### 1.3.4 Instruction-level Parallelism

We can improve processor performance by having multiple processor components executing concurrently. The two main approaches are **pipelining** and **multiple issue.**

**Pipelining**

Shared vrs. Distributed Memory:

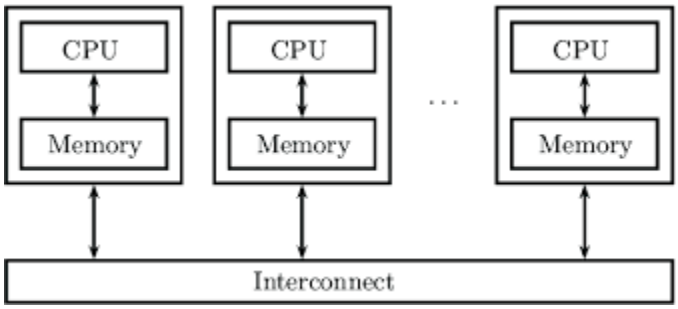


Figure 1.  A Schematic of a distributed-memory system

Message-Passing:

Message-passing is an example of a way to program a distributed-memory computational system. In such a system, data and instructions are communicated between each other. The data and instructions are **messages** to be communicated.

In the message-passing paradigm, a program running on one core-memory pair is called a process. Two processes communicate via functions. One process calls a send function , and the other receives a receive, and they talk to each other through those function calls. MPI is an implementation of that message-passing process and is not a *language* but a simple library, or here in python , a module.

## 2. Python, NumPy, and Pandas and Data-Science

### 2.1 Python

Programmers, data scientists, etc. love Python because it is *easy to use* and it is *expressive*, it allows you to say a lot, clearly, in a few lines and with minimal programming technique or artistry. It was developed primarily by Guido van Rossum at first in the late 1980's and implemented in 1989.

Python is an *interpreted*, *general purpose, dynamically-typed language that is fully object-oriented*. This is distinguishable from C which is a *general-purpose*,*statically-typed procedural language*.

An interpreted language is one that is not "directly" translated , or compiled, into machine code before it is executed. C for instance is a compiled language, a compiler translates whatever source code into machine code and then the machine-system executes it. Instead the interpreted language is translated into some intermediate language before it is translated to machine code and executed by the system. Python works as figure 2.1.1 shows:

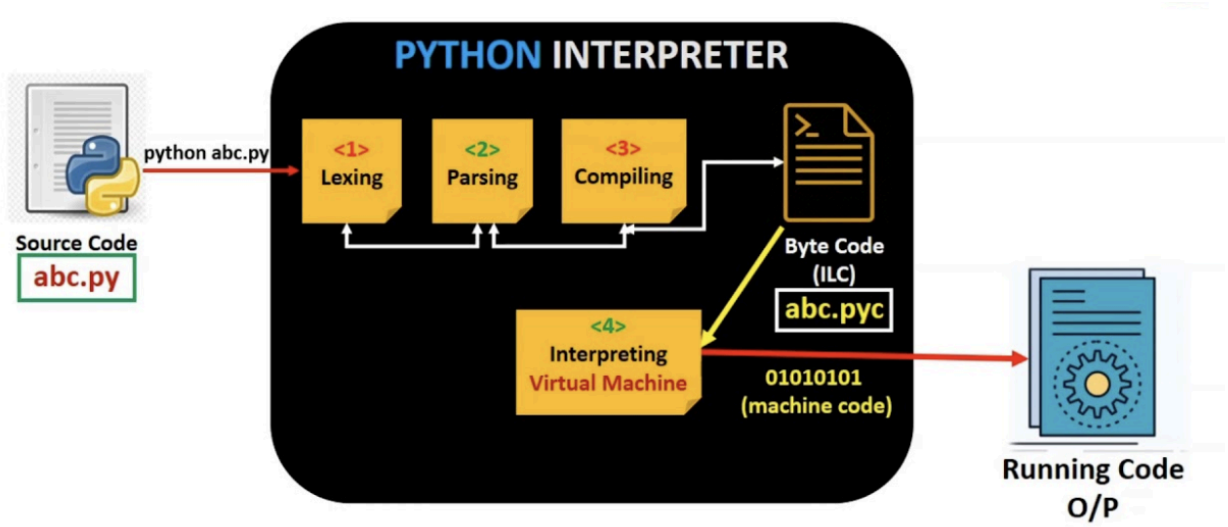



*Figure 2.1.1 The pipeline of interpretation, compilation, and execution of a python program. First the code is written, then submitted to the interpreter. The interpreter lexes, parses, then compiles the function into a "system-independent" byte-code that executes in a virtual machine which itself generates machine code. The code is then run on the machine.*

The source code is lexed , which means simply that it is broken up into tokens (words) that can be read by the interpreter. The code is then parsed, meaning those words are built up into phrases that are meaningful for the interpreter. The interpreter then converts this to a byte-code that is native to a python virtual machine that will then take that byte-code and turn it into machine-code to be executed by the system. The interpreter is called Cpython, and it is written in C.

A general purpose language is one which is *turing-complete*. In simplistic terms, if a language is turing-complete then *anything a human-being can compute, any function that is computable, can be written and computed in that language* (there is an ambiguity here where "language" stands both for a series of symbols with a syntax and semantics *and* a machine to execute that series of strings).

For a language to be a dynamically-typed language means that the user does not need to *explicitly* specify type in the *initialization* or

*evaluation* of a variable or piece of data. To initialize some data/a variable means to *define a type* and to evaluate means to *assign a value* to that variable

Finally, for a language to be fully object-oriented means that everything in the language is an object and every object is equipped with data and some attendant code. These objects are instances of classes which define a general formula for objects. In python everything is an object, and every object has its functions, called **methods** as well as data about it termed **attributes**. We access these methods via **object.method()** and attributes via **object.attribute**.

Let us explore the difference between python and C via example. Take the following iteration in C:

```c
#include <stdio.h>
int main(){
   int max = 100;
   int result = 0;
   for (int i = 0 ; i<max; i++){
       result += i ;
   }
   printf("result=%d", result);
   return 0;
}
```

Now in beautiful and succinct python:

```python
result = 0
for i in range(100):
   result += i
print(result)
```

Notice the difference. In the C program we *defined* the variables **i** and **result** i.e. we gave them a *type*. We then *initialized* the variables i.e. we gave them each a *value*. In python, the definition, the type of **result,** is inferred by the interpreter based on the initialization, the value given to it. Furthermore, **i** is dynamically created by the *iterator* **range** , which is a function that pumps out values when triggered.

A Python object is more than just a value. It also has a type, despite the *dynamically-typed* distinction. In fact, notice what the [Python Data Model documentation] says:

> Unset
>
> Every object has an identity, a type and a value. An object's identity never changes once it has been created; you may think of it as the object's address in memory. The 'is' operator compares the identity of two objects; the id() function returns an integer representing its identity.

Python is actually written in another lower-level language, C. Every Python object is therefore a *pointer* to a hidden C structure. A *pointer* is a reference to a location in memory. So when we say x=1000, in Python, we are pointing to a C structure, with a certain type for instance. Look at the Python 3.4 source code we find the integer (long) type defined looks like this :

```
struct _longobject{
   long ob_refcnt;
   PyTypeObject *ob_type;
   size_t ob_size;
   long ob_digit[1];
}
```

The integer contains four pieces of information

- *ob_refcnt* a reference count that helps Python handle memory allocation and deallocation.
- *PyTypeObject* a value encoding type of object
- *ob_size* object size of members
- *ob_digit* the actual integer value we expect to be represented.

That means when you store a Python , there is overhead, it takes time to store and operate on the object as opposed to operating directly on the C object. A C integer is just a label for a position in memory whose bytes encode an integer value. A Python integer is a pointer to a position in memory containing all Python object information including the bytes that contain the integer value.

## 2.2 SciPy Stack

The SciPy homepage defines SciPy as :

SciPy (pronounced “Sigh Pie”) is a Python-based ecosystem of open-source software for mathematics, science, and engineering:

*-https://www.scipy.org/*

Essentially, what is meant is that SciPy is a loose collection of Python packages and modules, a collection meant to facilitate computation demanded by scientific research. If you want to do data-science and machine-learning, almost everything you need is contained within this ecosystem, or collection.

In particular, some of the packages constituting the *stack* of SciPy are:

- NumPy: is the foundation of scientific computing in Python and is a shorthand for *Numerical Python*. It provides efficient numeric arrays, abnd wide support for numerical computation, including linear algebra, fourier transformations. The basic data structure and feature of the NumPy is the "N-dimensional array"-the *ndarray*. They are extremely efficient data structures for storage, queries, and modification.
- SciPy: the library is a collection of numerical algorithms, for domains such as signal processing, optimization, integration, and a wide class of statistics.
- Matplotlib: package which allows for plotting in 1-3 dimension. It is inspired by the Matlab language , and its own style/syntax of visualization.
- IPython: is an interactive interface that allows quick interaction with data.
- Jupyter: our famed environment for doing all things code. It also supports Ruby, Julia, Octave, bash, Perl, and R.
- pandas: is a package for columnar/tabular data. The basic data structure is the DataFrame which has rows and columns, like an excel spreadsheet. Extremely useful also for interacting with relational databases, time series, and machine-learning data.

Also allows, for data cleaning, munging, aggregation, and some basic plotting and statistics.
- scikit-learn: a unified package with all things machine-learning. We will end here!

We will be working with all of these extensively! How exciting, by the end of the course you will be budding scythonistas!

### 2.3 NumPy

Datasets come from many sources and have many formats and data-types. However, in order to do scientific computation, we need to compute on numbers, and a way to store and manipulate those numbers.

Despite the apparent heterogeneity, a great deal of data can be reduced to arrays of numbers. Images - digital ones- can be thought of arrays of numbers representing pixel brightness. Sound clips can be thought of as one-dimensional arrays of intensity versus time. Text can be converted to numerical representations.

NumPy is an efficient interface to store and operate on dense data buffers. They are similar to native-Python *lists* but *much more efficient* in storage, querying, and modification, especially as the data-scale increases. Because it is the core of the SciPy world, we will focus sometime on it.

#### 2.3.1 Basics of Numpy Ndarray and Memory Buffers

NumPy is built on top of an object, the numpy ndarray, which is a fancy-data structure written in C. That data-structure consists of two ingredients 1. The data buffer 2. Meta-data on the data buffer.

First let us explain some terminology about arrays. An array is a contiguous block of memory made up of homogeneous elements of fixed size of the same-type. The array also has *indices*, *stride* and a *blocklength*. The index is some symbol denoting order within the array, typically it is an *integer* beginning at 0 and going up to the number of total items n. The stride is the amount of space in bytes or bits or in terms of memory *between* elements. And the blocklength is the size in

bytes or bits or whatever memory unit in which each element is stored. Figure 2.3.1. show this structure for a 1-d array:

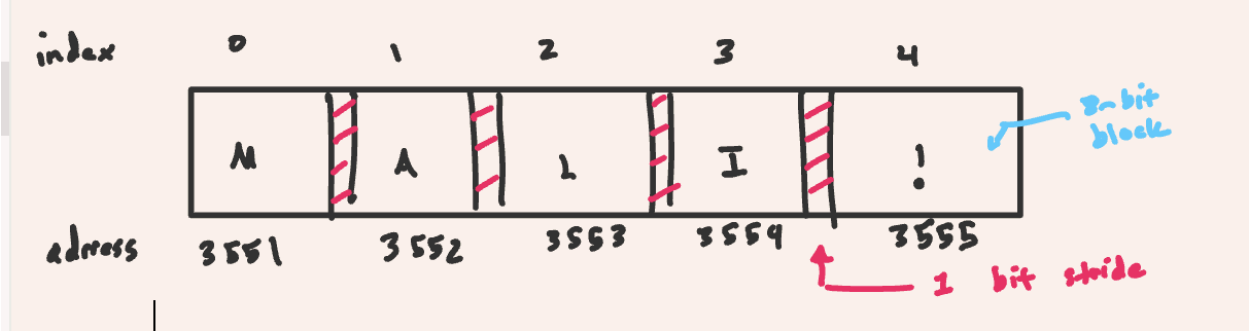



*Figure 2.3.1.1 Anatomy of an array.*

NumPy arrays have *homogeneous* data types, and they are implemented in C enabling vectorized operations. For instance, in Python list, to multiply 5 across the list, you do it element by element. An ndarray multiplies the 5 simultaneously so to speak across the vector.

They are also size efficient. Which makes sense, a nparray occupies only the amount of the data, plus a small overhead of metadata, such as attributes like *shape of the ndarray*. As we saw above Python instead has a lot of metadata loaded per object plus the actual data.

Now the data buffer is what we tend to think of as an array (like in C),a contiguous and fixed block of memory containing a set of fixed-sized elements of the same type. However, a NumPy array comes with additional information attached to this data buffer including but not limited to:

1. The size of the whole array in bytes
2. The dimensions and shape of the array
3. Information (via the **dtype** object) about the interpretation of the basic data element. The basic data element may be as simple as an int or a float, or it may be a compound object (e.g., struct-like), a fixed character field, or Python object pointers.
4. The separation between elements for each dimension (the stride). This does not have to be a multiple of the element size.

5. The byte order of the data (which may not be the native byte order). Byte order refers to how the memory is stored. In little-endian systems the least significant byte (LSB) is stored first. In big-endian systems the most significant byte is stored first. What this means is let us take for instance 124234, then the bytes for 4 are stored first in little, and bytes for 100000 are stored first in big.
6. Whether the buffer is read-only.
7. Whether the array is to be interpreted as C-order or Fortran-order. In C arrays are stored row-wise, that means it is rows that are stored contiguously. When we access say a matrix row we are accessing a contiguous block of memory. In Fortran, arrays are stored in column fashion. When we access say a whole column of a matrix this accesses a contiguous block of memory.

The following code shows how to access these attributes , or meta-data on an example array. Simultaneously notice how we are creating an array out of a list of lists :

```
"""
Depict some attribute info about ndarray for 2 dimensional
example.

Date: 05/21/2024
Author: Djamil Lakhdar-Hamina

"""

import numpy as np


def main():
   x = np.array([[1, 2, 3], [1, 2, 3], [1, 2, 3]], dtype=">i4", order="c")
   x.flags.writeable = False
   print(f"1. number of bytes: {x.nbytes}")
   print(f"2. number of dimensions: {x.ndim}")
   print(f"2.1 shape of array: {x.shape}")
   print(f"3. datatype and itemsize: {x.dtype, x.dtype.itemsize}")
   print(f"4. separation for dimensions: {x.strides}")
   print(
```

```
      f"5. byte order ('=' means native, < means little-endian, > means big-endian)\
: {x.dtype.byteorder}"
   )
   print(f"6-7.: {x.flags}")


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

Which gives output:

```
1. number of bytes: 36
2. number of dimensions: 2
2.1 shape of array: (3, 3)
3. datatype and itemsize: (dtype('>i4'), 4)
4. seperation for dimensions: (12, 4)
5. byte order ('=' means native, < means little-endian, > means big-endian): >
6-7.:   C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : True
  WRITEBACKIFCOPY : False
```

First we created a numpy ndarray , we specified a number of arguments, we set for instance the datatype via **dtype** (here a big-endian integer of 4 bytes) and the contiguity order via **order** (we made it c or row order via **c**). We even make this ndarray only readable and not writable. Then we asked for its total size in bytes which is 36. It is like a matrix, a list of lists, and therefore has 2 dimensions i.e. a dimension is the number of numbers we need to specify where a single item is in the ndarray). We then get the shape, which is the actual number in each dimension. We query the datatype contained in this array and the size of each item. We then ask for the stride or separation between each element. The **strides** attribute returns a tuple (n0,n1…n) each number refers to the stride for each dimension. Here the strides (12,4) means that we need 12 bytes to get across a row to the next row, and only 3 within a row to get to the next item in the row. This makes sense, there are 3 items in the row of size 4 bytes, each item is 4 bytes. Finally we query the byte-order and we query a number of other information.

Numpy Ndarrays can also be of any *dimensionality*. In the simplest terms the shape or dimensionality of an array is the number of numbers or indices we need to access an item. Geometrically it can be thought of as analogous to *space*. A 1-d array will be referred mathematically to as a *vector* and a 2-d array will be referred to as a *matrix*. Anything above 2 will be simply called an n-d array. A 3-d day array can be thought of as a vector of matrices, a 4-d as a matrix of matrices. The two senses of *dimension* are depicted in figure 2.3.2:

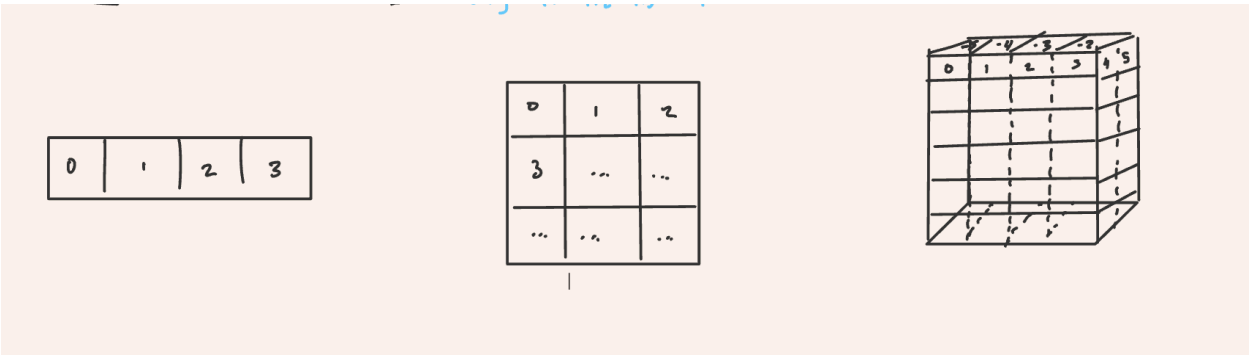


*Figure 2.3.2. Example of 1,2,3 dimensional ndarrays. These can be **any** dimension, hence ndarray (n-dimensional). The dimension will only be restricted by memory conditions of the system.*

Why use this numpy array? Efficiency ! Let's check the difference in time, first let's create an array, then create a list copy of the values in the array:

```
import numpy as np
from timeit import timeit
```

We run a multiplication by 5 on one-hundred thousand elements. Wow incredible difference! We utilize the **timeit** module with its **timeit** function.

```
import numpy as np
from timeit import timeit

val1 = timeit("[5*i for i in array_list]",
              setup="import numpy as np; array = np.arange(1e5); \
              array_list=array.tolist()", number=1000)
val2 = timeit("5*array",
              setup="import numpy as np; array = np.arange(1e5)", number=1000)

print(f"element-by-element took {val1} seconds")
print(f"vectorized operation took {val2} seconds")
```

The timeit object ran our code 1000 times. I had to utilize the **setup** command to import numpy and define variables. That command simply runs whatever commands *before* the code itself is executed. Running this script gives :

```
element-by-element took 2.7545725419186056 seconds
vectorized operation took 0.02678541699424386 seconds
```

This is almost 100x in speedup! Absolutely awesome, ad we acheived that speedup by writing *even simpler code* than in pure python.

### 2.3.2 Types in NumPy

Numpy encapsulates types within a **dtype** class. A data type object (an instance of **numpy.dtype** class) describes how the bytes in the fixed-size memory buffer corresponding to an array item should be treated. It describes the following features of the data:

1. Type of data (integer, float, Python object, etc.)
2. Size of data (how many bytes is in *e.g.* the integer)
3. Byte order of the data (little-endian or big-endian)
4. If the data type is structured data type, an aggregate of other data types, (*e.g.*, describing an array item consisting of an integer and a float),
    1. what are the names of the "fields" of the structure, by which they can be accessed,
    2. what is the data-type of each field, and
    3. which part of the memory block each field takes.
5. If the data type is a sub-array, what is its shape and data type.

We can for instance create our "own" data type and access a number of information about this type via the attributes and methods of this data type object.

```
>>> dt= np.dtype("<f8")
>>> print(dt)
float64
>>> print(dt.byteorder)
=
>>> print(dt.itemsize)
8
```

We can create "structured" arrays very easily. Say we want to create a structured array for a 3-d grid, where each value is a point in the grid and that point also contains extra info. That info is itself an array , a sub-array. The following code does this:

```
>>> point_type = np.dtype([('x', np.float64),
          ('y', np.float64),
          ('z', np.float64),
          ('info', np.float64, (3,))])
>>> points = np.array([(1.0, 2.0, 3.0, (1.0, 2.0, 3.0)),
                 (4.0, 5.0, 6.0, (7.0, 8.0, 9.0))], dtype=point_type)
>>> print(points[1])
(4., 5., 6., [7., 8., 9.])
>>> print(points['x'])
[1. 4.]
```

The most common datatypes are listed below.

| Data Type | Description |
|---|---|
| bool_ | Boolean (True or False) stored as byte |
| int_ | Default integer type (same as C long) |
| intc | Identical C int |
| intp | Integer used for indexing |
| int8 | Byte -128 to 127 |
| int16 | Integer (-32768, 32767) |
| int32 | Integer (-2147483648, 2147483647) |
| int64 | Integer (-9223372036854775808, 9223372036854775807) |
| uint8 | unsigned integer (0 to 255) |
| uint16 | Unsigned integer (0 to 65535) |

| | |
|---|---|
| uint32 | Unsigned integer (0 to 4294967295) |
| uint64 | Unsigned integer (0 to 18446744073709551615) |
| float_ | Shorthand for float64 |
| float16 | Half-precision float: sign bit, 5 bits exponent, 10 bits mantissa |
| float32 | Single-precision float: sign bit, 8 bits exponent, 23 bits mantissa |
| float64 | Double-precision float: sign bit, 11 bits exponent, 52 bits mantissa |
| complex_ | Introduction to Course |
| complex64 | Introduction to Course |
| complex128 | Introduction to Course |

There is also a special **np.NaN** object representing precisely missing or “untyped” data. When we mentioned that numpy arrays are homogeneous this is not quite true. Numpy arrays are homogeneous up to missing data. An array can have elements of a type *and* elements of **np.NaN.**

### 2.3.3 Creating Arrays

We can create ndarrays either explicitly or “from scratch”.

#### Creating Arrays from Lists

The absolute simplest form :

```
>>> import numpy as np
```

```
>>> np.array([1,2,3,4,5])
array([1, 2, 3, 4, 5])
```

Specify the data type:

```
>>> np.array([1, 2, 3, 4, 5], dtype='float')
array([1., 2., 3., 4., 5.])
>>> np.array([1, 2, 3, 4, 5], dtype='f64')
array([1., 2., 3., 4., 5.])
>>> np.array([1, 2, 3, 4, 5], dtype=int)
array([1, 2, 3, 4, 5])
>>> np.array([1, 2, 3, 4, 5], dtype='i8')
array([1, 2, 3, 4, 5])
```

Initialize with nested-lists:

```
>>> np.array([[1, 2, 3, 4, 5], [1, 2, 3, 4, 5]], dtype='<f8')
array([[1., 2., 3., 4., 5.],
       [1., 2., 3., 4., 5.]])
```

Which is matrix-like.

Creating Arrays from Scratch

```
## array of zeros created from scatch
>>> np.zeros(5,dtype="int")
array([0, 0, 0, 0, 0])
```

```
## matrix of ones created from scatch
>>> np.ones((3,2), dtype='int')
array([[1, 1],
       [1, 1],
       [1, 1]])
```

```
## array of value specified created from scatch
>>> np.full((2,2),42, dtype='int')
array([[42, 42],
       [42, 42]])
```

```
# array of linear sequence
>>> begin = 10
```

```
>>> end = 20
>>> step = 2
>>> np.arange(begin, end, step, dtype='int')
array([10, 12, 14, 16, 18])
```

```
## 5 values spaced between 1
>>> np.linspace(0,1, 5)
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

```
## array of random values based on n dimensions
>>> np.random.random((3,4))
array([[0.19251461, 0.51908365, 0.28737893, 0.71698662],
       [0.49852127, 0.62150998, 0.67259703, 0.63946668],
       [0.34070119, 0.1366799 , 0.52846061, 0.88499201]])
```

```
## array of random values in an interval 0-10 and (,,,) dimensions
>>> np.random.randint(0,10,(3,3))
array([[3, 6, 1],
       [9, 1, 2],
       [7, 2, 5]])
```

We can even initiate an array of characters but only in the form of a buffer of characters or of string bytes.

```
>>> message = b"ace is the best!"
>>> np.frombuffer(message,dtype='|S1',count=5)
array([b'a', b'c', b'e', b' ', b'i'], dtype='|S1')
```

Notice that we specified type "**|S1**" which means a string-byte and we initialized the string with b which means **byte.**

### Indexing

We can index a one-dimensional array as follows:

```
>>> x = np.random.randint(0,10,(3))
>>> x
array([2, 0, 9])
>>> x[1]
0
>>> x[-2]
0
```

We can index a multidimensional array as follows:

```
>>> x = np.random.randint(0,10,(3,3,2))
array([[[9, 7],
        [9, 3],
        [4, 5]],

       [[4, 4],
        [0, 6],
        [2, 7]],

       [[0, 6],
        [6, 7],
        [3, 2]]])
>>> x[0,0,0]
9
>>>  x[2,0,1]
6
```

We created 3 , 3x2 matrices. The first dimension is to pick out a matrix. The other two are to pick out elements within the given matrix.

We can modify values using indexes.

```
>>>  x[2,0,1] = 10
>>>  x[2,0,1]
10
```

### Slicing

We were selecting or modifying individual elements of an array using indexing. But we can also select and modify subarrays of an array using a common *slice* notation found in many languages. It follows the following syntax with : and [] :

```
x[start:stop:step]
```

Where start is as you may imagine where the slice begins, stops where it ends, and step by how much you need to increment to "select" an

element. The defaults are start=0, end=size of array, and step is 1. Let us look at how to slice in one and multiple dimensions.

In one-dimension we have.

```
>>> x = np.random.randint(0,10,(5,))
>>> x
array([5, 6, 3, 6, 1, 0, 7, 6, 0, 6])
>>> # from the 4th to the 9th element not including 9
>>> x[3:8]
array([6, 1, 0, 7, 6])
>>> # every 2 elements
>>> x[::2]
array([5, 3, 1, 7, 0])
>>> # nifty way to reverse the array
>>> x[::-1]
```

In multiple dimensions we have. In general, we really only deal with single or two-dimensional arrays, because those are what we think of in scientific computing as vectors and matrices, a fundamental mathematical object and data structure within science. In fact, matrix computation, and n-dimensional computation, becomes so important that many times, scientists develop their own n-dimensional notation, syntax, and mathematics. Think of the Einstein notation or the bra-ket notation in physics. NumPy even has for instance the *einsum* or Einstein summation of arrays.

The syntax of slicing is x[slice1,slice2,slice3...slicen] up to however many dimensions the numpy array has. Each slice can be treated as it was above with the same syntax.

```
>>> x = np.random.randint(0, 5, (4, 4, 3))
>>> x
array([[[2, 1, 4],
        [4, 2, 0],
        [0, 0, 4],
        [1, 0, 1]],

       [[2, 0, 1],
        [4, 4, 1],
```

```
        [4, 3, 1],
        [2, 1, 1]],

       [[3, 0, 0],
        [3, 1, 2],
        [4, 1, 3],
        [4, 3, 4]],

       [[2, 3, 1],
        [2, 1, 3],
        [2, 0, 2],
        [1, 4, 4]]])
>>> # pick out second matrix , 3rd row, first column
>>> x[1,2,0]
>>> # after the second matrix, pick out after 3nd row , reverse order of columns
>>> x[1:, 2:, ::-1]
```

Now when we slice an array we are actually creating a *view* of the array and not a *copy*. This means that when we modify the view we modify the original array. If you want to create a new copy then utilize the copy method for the array.

```
>>> x = np.linspace(0,1,5)
>>> x
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
>>> x_view = x[1:]
>>> x_view[:] = 1.0
>>> x
array([0.  , 1,0, 1.0 , 1.0, 1.  ])
>>> x_copy = x.copy()
>>> x_copy[1:] = 2.0
>>> x
array([0.  , 1,0, 1.0 , 1.0, 1.  ])
```

Notice that the original x was not modified by the modification of the copy.

## Reshaping

Reshaping means taking a np array with a certain number of dimensions and changing them. This can be abstract so lets see this with a series of concrete example.

Let us transform an array to matrix. Let us say we have have nine elements so from this let's create a 3x3 matrix.

```
>>> x = np.linspace(0, 1, 9)
array([0.   , 0.125, 0.25 , 0.375, 0.5  , 0.625, 0.75 , 0.875, 1.   ])
>>> x.reshape((3,3))
array([[0.   , 0.125, 0.25 ],
       [0.375, 0.5  , 0.625],
       [0.75 , 0.875, 1.   ]])
```

Let us transform a matrix to a 3-d array. Let us say we have a 9x9 matrix. Let us split this up into a 9x3x3 ndarray. We could have made a 3x9x3 or even a 3x3x9.

```
>>> x = np.linspace(0, 1, 81)
>>> x = x.reshape((9,9))
>>> x = x.reshape((3,3,3))
array([[[0.07196549, 0.43934395, 0.73158514],
        [0.90040658, 0.40903829, 0.64692336],
        [0.55109731, 0.64045129, 0.92195318]],

       [[0.6422511 , 0.40403126, 0.05380738],
        [0.46551075, 0.10308538, 0.9035816 ],
        [0.35511235, 0.47387929, 0.85130791]],

       [[0.75403115, 0.0091519 , 0.51000428],
        [0.08957171, 0.39863597, 0.39516172],
        [0.89907956, 0.79184014, 0.31525406]],

       [[0.15244042, 0.25210631, 0.98242253],
        [0.9766994 , 0.38894752, 0.77701415],
        [0.71089708, 0.23884599, 0.86088028]],

       [[0.0801215 , 0.45139511, 0.51849045],
        [0.26563942, 0.52444046, 0.11321709],
        [0.01959629, 0.09577526, 0.1074409 ]],
```

```
       [[0.33396691, 0.06679736, 0.87091217],
        [0.1110298 , 0.43952828, 0.6852537 ],
        [0.35911367, 0.35466731, 0.16248045]],

       [[0.21631513, 0.40738741, 0.15727467],
        [0.36694705, 0.40792479, 0.27992191],
        [0.47079416, 0.28931423, 0.17697809]],

       [[0.04266383, 0.74002514, 0.38688891],
        [0.18705322, 0.97946776, 0.6288542 ],
        [0.96202895, 0.62418823, 0.25668882]],

       [[0.53145507, 0.42013421, 0.42067023],
        [0.02882082, 0.60415347, 0.36447763],
        [0.92944354, 0.94302686, 0.34113327]]])
```

Notice that very often dimensions are written as *tuples* (dimension 0, dimension 1, dimension 2). Dimensions 1 is like a row and dimension 2 like a column. The higher dimensions take matrices, or matrices of matrices … etc. as elements. I hope you see the pattern to reshaping but I can state it more formally.

**Rule: If you have an array with n elements and wish to reshape it into an array with dimensions (n0,n1,n2….n) then n%(n0*n1*n2…*n)= 0  where % is modulus operator.**

We can also reshape 1-d arrays. We can make them  what are called row vectors or column vectors!

```
>>> x = np.linspace(0, 1, 9)
array([0.   , 0.125, 0.25 , 0.375, 0.5  , 0.625, 0.75 , 0.875, 1.   ])
```

### Joining, Splitting

We can actually work to join or split arrays to form new arrays. You can join or concatenate two arrays with **np.concatenate**, **np.vstack**, and **np.hstack**.

**np.concatenate** joins two arrays along a certain axis. What does this mean? It is a bit abstract so let us proceed by example.

First , let us take two 1-d arrays and join them to make a bigger 1-d array.

```
>>> x = np.array([1,2,3,4,5])
>>> y = np.array([6,7,8,9,10])
>>> np.concatenate([x,y])
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> np.concatenate((x,y))
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> z=np.array([1,2,3]); np.concatenate((x,y,z))
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 1 , 2 ,3])
```

Easy enough. Now let us take a matrix and concatenate it two different ways. We could add *rows* or we could add *columns*. So we want to join along either the 1rst row axis, which is indexed by 0, or the 2nd column axis, indexed by 1.

```
>>> x = np.array([[1,2,3],[1,2,3]])
>>> y = np.array([[4,5,6],[7,8,9]])
# create a new matrix by adding rows of y to x
>>> np.concatenate([x,y],axis=0)
array([[1, 2, 3],
       [1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
# create a new matrix by adding colunmns of y to x
>>> np.concatenate([x,y],axis=1)
array([[1, 2, 3, 4, 5, 6],
       [1, 2, 3, 7, 8, 9]])
```

We can acheive the same with **np.vstack**, and **np.hstack** which respectively line up the matrices vertically or horizontally. It is a better option when working with arrays of unequal dimensions. It also makes more sense to me , it is more explicit.

```
# create a new matrix by adding rows of y to x
```

```
>>> np.vstack([x,y])
array([[1, 2, 3],
       [1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
# create a new matrix by adding columns of y to x
>>> np.hstack([x,y])
array([[1, 2, 3, 4, 5, 6],
       [1, 2, 3, 7, 8, 9]])
```

It even works with unequal sized arrays. However, the number of elements in the dimension joined along *must agree*.

```
>>> x = np.array([[1, 2, 3], [1, 2, 3]])
>>> y = np.array([1,3,4])
>>> np.vstack((x,y))
array([[1, 2, 3],
       [1, 2, 3],
       [1, 3, 4]])
```

The opposite of combining is splitting arrays. For these we can use the parallel **np.split** , **np.vsplit**, and **np.hsplit**. With splitting you put the array name, and a list of split points signifing the index of the dimension you want to split along . The split points determine how many arrays you want returned. N points lead to N+1 arrays.

```
>>> x =  np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.split(x,[2,7])
[array([0, 1]), array([2, 3, 4, 5, 6]), array([7, 8, 9])]
```

It returned a list of three arrays, split at the third index and split at the 8th index. Now how about a matrix?

```
>>> x = np.arange(25).reshape((5,5))
>>> x
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

```
>>> np.split(x,[2,3],axis=0)
[array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]]), array([[10, 11, 12, 13, 14]]), array([[15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])]

>>> np.split(x,[2,3],axis=1)
[array([[ 0,  1],
       [ 5,  6],
       [10, 11],
       [15, 16],
       [20, 21]]), array([[ 2],
       [ 7],
       [12],
       [17],
       [22]]), array([[ 3,  4],
       [ 8,  9],
       [13, 14],
       [18, 19],
       [23, 24]])]
```

The first means split along the third row, then split along the 4th. So we have a first array of 2 rows and 5 columns, an arary of 1 row and 5 columns, and an array of 2 rows and 5 columns. The second command does the same but with columns. Now, try the same with **np.hsplit** and **np.vsplit** as an exercise.

### Ufuncs

Universal functions or , **ufuncs** for short, are extremely fast vector operations. In many ways NumPY is just a high-level interface to perform *vectorized operations* , and ufuncs are ways to do these operations , bounded operations on each element of an array. These are extremely fast methods for iteration.

A vectorized operation is simply an operation that acts on each element of an array *at the same time* rather than element by element. Obviously this *concurrency* implies *speed-up*.

There are two main kinds, unary and binary ufuncs, the first which oprates on one operand and the other on two. A list of the most common ufuncs :

| Operator | Equivalent Ufunct | Description |
|---|---|---|
| + | np.add | Addition |
| - | np.negative | Negative, Negation |
| - | np.subtract | Subtraction |
| * | np.multiply | Multiplication |
| / | np.divide | Division |
| // | np.floor_divide | Division, truncate e.g. 1//2 = 0 |
| ** | np.power | Power |
| % | np.mod | Modulus or remainder, e.g. 10 % 5 is 5. If you divide once, left with |

We can create our *own binary ufuncs*, custom ufuncs via the **vectorize** command. Let us say we have a function **mag**, it computes the magnitude of two elements which is to say **mag(a,b)** means sqrt(**(a**2+b**2)**) e.g. mag(1,2)=sqrt(1^2+2^2)=sqrt(5). To make sure this function is vectorized we write:

```
>>> docstring = """
Calculate the magntiude between two elements element by element in array

  This function two arrays, finds the magnitude of 1rst element and 1rst
  element of second array , 2nd element and 2nd element of the second array
  etc.
```

```
   Parameters:
   x (np.ndarray[(int | float)]): The first array
   y (np.ndarray[(int | float)]): The second array

   Returns:
   np.ndarray: the array with each magnitude calculated


"""
>>> dtype=np.float32
>>> def mag(x: (int | float), y: (int | float)) -> (int | float):
      return (x**2+y**2)**(1/2)
>>> mag_vect = np.vectorize(mag, doc=docstring, otype=dtype)
>>> x = y = np.arange(10)
>>> mag_vect(x, 1)
array([1.        , 1.41421356, 2.23606798, 3.16227766, 4.12310563,
       5.09901951, 6.08276253, 7.07106781, 8.06225775, 9.05538514])
>>> mag_vect(x, y)
array([ 1.        ,  1.73205081,  3.        ,  4.35889894,  5.74456265,
        7.14142843,  8.54400375,  9.94987437, 11.35781669, 12.76714533])
```

We utilized the **vectorize** class. An instance of it is a vectorized python function, our very own ufunc, and this class has signature:

```
class vectorize(
   pyfunc: (...) -> Any,
   otypes: str | Iterable[DTypeLike] | None = ...,
   doc: str | None = ...,
   excluded: Iterable[int | str] | None = ...,
   cache: bool = ...,
   signature: str | None = ...
)
```

**Pyfunc** is of course the function we vectorize, **otypes** is the output type. The **doc** argument is the docstring. We access it via:

```
>>> mag_vect.__doc__
'"\nCalculate the magntiude between two elements element by element in array \n\n
This function two arrays, finds the magnitude of 1rst element and 1rst\n    element of
```

```
second array , 2nd element and 2nd element of the second array \n      etc. \n\n
Parameters:\n         x  (np.ndarray[(int  |  float)]):  The  first  array\n         y
(np.ndarray[(int  |  float)]):  The  second  array\n\n     Returns:\n     np.ndarray: the
array with each magnitude calculated \n\n\n'
```

**Signature** is a string that specifies what the inputs and the outputs to our vectorized function is. It takes the form of '(dim), (dim) -> (dim)' where '()' is a special dimension meaning more or less "any". Let us define a signature of a vector of size m and another of size m give a new vector of size m.

```
>>> mag_vect = np.vectorize(mag, doc=docstring,cache=True,signature='(m),(m) -> (m)')
>>> x = y = np.arange(10)
>>> mag_vect(x, 1)
ValueError: 0-dimensional argument does not have enough dimensions for all core
dimensions ('m',)
>>> mag_vect = np.vectorize(mag, doc=docstring,cache=True,signature='(m),(1) -> (m)')
>>> mag_vect(x, 1)
array([1.        , 1.41421356, 2.23606798, 3.16227766, 4.12310563,
       5.09901951, 6.08276253, 7.07106781, 8.06225775, 9.05538514])
>>> mag_vect = np.vectorize(mag, doc=docstring,cache=True,signature='(),() -> ()')
>>> mag_vect(x, 1)
array([1.        , 1.41421356, 2.23606798, 3.16227766, 4.12310563,
       5.09901951, 6.08276253, 7.07106781, 8.06225775, 9.05538514])
>>> mag_vect(x, y)
array([ 1.        ,  1.73205081,  3.        ,  4.35889894,  5.74456265,
        7.14142843,  8.54400375,  9.94987437, 11.35781669, 12.76714533])
```

**Excluded** is simply which arguments will not be vectorized. Let us take another function:

```
>>> def polyval(p, x):
    _p = list(p)
    res = 0
    while _p:
        co = _p.pop(0)
        res += co**x
    return res
```

```
>>> vpolyval = np.vectorize(polyval, excluded=['p'])
>>> vpolyval(p=[1, 2, 3], x=[0, 1])
array([3, 6])
```

We defined a **polyval** function. This non-vectorized function takes a list **p,** representing coefficients a,b,c..n of a polynomial, and x, representing a power such that **polyval(p,x)=a^x+b^x+c^x+...n^x**. For instance:

```
>>> polyval([1,2,3],1)

6
```

So we still want each element in our position **p** to be treated as an integer and not as a vector. Therefore we exclude it as a vector argument. Now We can do the same *positionally via:*

```
>>> vpolyval.excluded.add(0)
```

**Cache** is an option to cache the first function to determine which output type if **otypes** is not specified. Finally we can use the decorator syntax:

```
>>> @np.vectorize(doc=docstring,cache=True,signature='(m),(m) -> (m)')
def mag(x: (int | float), y: (int | float)) -> (int | float):
   return (x**2+y**2)**(1/2)
```

We need to mention an important fact here i.e. many ufuncs **overload** native python operators. To overload means to overwrite or redefine native operator behavior based on a new type or class. We can add two numpy arrays via **np.add** or we could just use **+.** Furthermore, when we use say the **np.absolute** function we can actually just write **abs** and the interpreter infers based on the type that **np.absolute** is actually being called. Some examples of ufunc use:

```
>>> x = np.arange(10)
# unary ufunc negation
```

```
>>> -x
array([ 0, -1, -2, -3, -4, -5, -6, -7, -8, -9])
# binary ufunct addition
>>> np.add(x,x)
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
>>> x+x
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
# overload example
>>> np.absolute(-x)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> abs(-x)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

### Reductions

In computer science there are operations called reductions. A reduction takes some array [a,b,c,...n] and chains together all elements with an operator let us call it * to produce a single value x i.e. x= a*b*c…*n. We can call **reduce** on a ufunc to produce that single value for instance:

```
>>> x = np.arange(1, 101, dtype="float32")
>>> np.add.reduce(x)
5050.0
>>> np.multiply.reduce(x)
<stdin>:1: RuntimeWarning: overflow encountered in reduce
inf
```

Wait what happened? Well the number produced by the reducing multiplication is too large to be stored as a 32-bit float. We can of course change the type to make sure this does not happen to , depending on your system , a 128 or 64 bit number.

```
>>> x = np.float64(x)
>>> np.multiply.reduce(x)
9.33262154439441e+157
```

We can also store intermediate results of a reduction via **accumulate**.

```
>>> x = np.arange(1, 10, dtype="float64")
>>> np.multiply.accumulate(x)
```

```
array([1.0000e+00, 2.0000e+00, 6.0000e+00, 2.4000e+01, 1.2000e+02,
       7.2000e+02, 5.0400e+03, 4.0320e+04, 3.6288e+05])
```

### Statistical Aggregations

We can perform a number of statistical and other operations via some common functions listed here :

| Function | NaN-safe Ufunct | Description |
|---|---|---|
| np.sum | np.nansum | Compute the sum |
| np.prod | np.nanprod | Compute the product of elements |
| np.mean | np.nanmean | Compute the mean of elements |
| np.std | np.nanstd | Compute the standard deviation |
| np.var | np.nanvar | Compute the variance |
| np.min | np.nanmin | Find the minimum value |
| np.max | np.nanmax | Find the maximum |
| np.argmax | np.nanargmax | Find the index of the maximum value |
| np.argmin | np.nanargmin | Find the index of the minimum |
| np.median | np.nanmedian | Compute the median of the elements |

| | | |
|---|---|---|
| np.percentile | np.nanpercentile | Compute the percentile rank of each elemen |

They are rather obvious to use but an interesting feature to mention is that these numpy functions can perform the operation solely on a dimension e.g. :

```
>>> x = np.arange(25).reshape((5,5))
>>> x
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
>>> np.mean(x)
12.0
>>> x.mean()
>>> 12.0
## by row
>>> x.mean(axis=0)
array([10., 11., 12., 13., 14.])
>>> x.mean(axis=1)
array([ 2.,  7., 12., 17., 22.])
```

### Broadcasting

Broadcasting refers to a technique of operating conjointly on arrays of different dimensions, more formally it is a set of rules for applying binary ufuncs on arrays of differing dimensions. It allows us to apply vectorized operations to vectors of different sizes, retaining the efficiency of the numpy array and its ufuncs. For arrays of the same size binary operations just operate on an element-by-element basis. This is what we came to expect.

We actually used broadcasting when we added or multiplied a scalar to an array. We can do this for a one or multidimensional array.

```
>>> a=np.arange(3)
>>> M=np.ones((3,3))
>>> a+5
[5 6 7]
>>> M+a
[[1. 2. 3.]
 [1. 2. 3.]
 [1. 2. 3.]]
```

Conceptually, we should think of the scalar shape=(1,) as being expanded to dim=(1,3) [5,5,5] and then added element by element to **a** . In the case of **M**, a is then expanded from (1x3) to a (3x3) array where each row added is simply [0,1,2] to each row of M. Another way of seeing it is that the original (1x3) was expanded where three of the same array are stacked until you get a (3x3) array.

Now there are more mysterious cases, such as the following:

```
>>> a=np.arange(3)
>>> a
array([0, 1, 2])
>>> b=np.arange(3)[:,np.newaxis]
>>> b
array([[0],
       [1],
       [2]])

>>> a+b
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

What just happened? The above example can be depicted in figure 2.3.3.3:

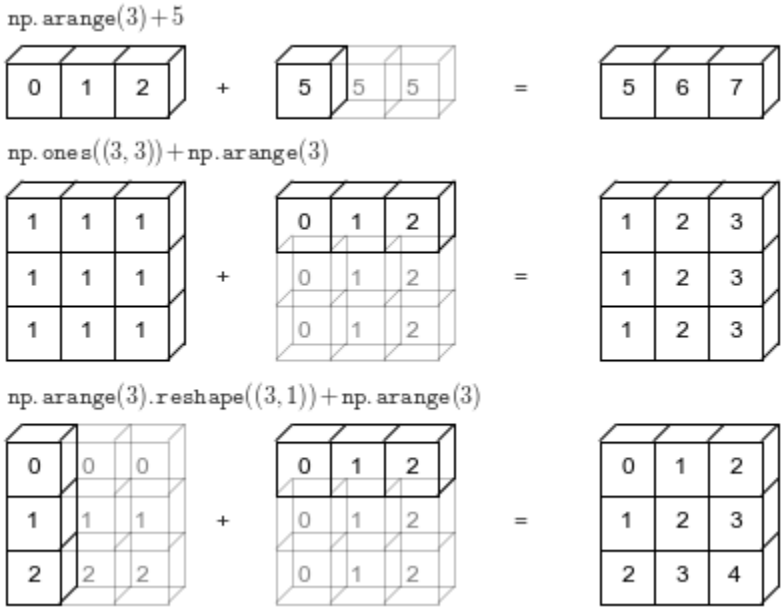



*Figure 2.3.3.3*

Rules of Broadcasting

Now let us see how we can approach such cases, they are in fact *extremely* important.

Here are the rules:

1. If the two arrays differ in their dimensions, the shape of the one with the fewer is padded with ones on its leading left side.
2. If the shape of the two arrays does not agree in any dimension, the array with shape equal to one in that dimension is stretched to match that other shape.
3. If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

Let us look at some cases to illustrate broadcasting.

Example 1 : One-Operand Broadcast

Let's add a two-dimensional array to a one dimensional array. We run the following program from command line to output print statements in an informative way:

```
import numpy as np
M=np.ones((2,3))
a=np.arange(3)
print("M.shape =", M.shape)
print("a.shape =",a.shape)
## By rule 1 , if two arrays differ in number of dimensions ,
## the shape of the one with fewer dimensions is padded with ones on its leading left
side.
print("M.shape ->",M.shape)
print("a.shape->","(1,3)")
```

Which gives output:

```
M.shape = (2, 3)
a.shape = (3,)
M.shape -> (2, 3)
a.shape -> (1,3)
```

But the first dimension still does not match the first dimension. By rule 2.

```
print("M.shape =", M.shape)
print("a.shape ->","(2,3)")
```

Giving output:

```
M.shape -> (2, 3)
a.shape -> (2,3)
```

Now they agree! Figure 2.3.3.4 depicts the process:

*Figure 2.3.3.4: Broadcasting of a matrix and a vector for addition.*

### Example 2: Two-Operand Broadcast

Let's take a look at an example where both arrays need to be broadcast.

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
print("a.shape =", (3, 1))
print("b.shape =", (3,))
```

Gives:

```
a.shape = (3, 1)
b.shape = (3,)
```

But rule II says we have to pad b with a dimension on the *left*. Which means we have.

```
a.shape = (3, 1)
b.shape = (1,3)
```

Now rule II says we have to match the one in b to the shape in a. So we get:

```
a.shape = (3, 1)
b.shape = (3,3)
```

Now rule II says we have a one in a , so we have to match its dimension to the dimension in b:

```
a.shape = (3, 3)
b.shape = (3,3)
```

Now we can add the two.

```
print(a+b)
```

Gives:

```
array([[0, 1, 2],

      [1, 2, 3],

      [2, 3, 4]])
```

*Figure 2.3.3.5: Depicting the broadcast-algorithm at work.*

### Fancy Indexing

In this section we will explore another way of indexing arrays *fancy indexing*. The basic concept is simple: pass an array of indices to select multiple elements. With fancy indexing, the resultant array's shape reflects the index array as opposed to the array being indexed.

```
>>> rand = np.random.RandomState(1917)
>>> x = rand.randint(10, size=10)
>>> x
array([7, 8, 9, 6, 5, 6, 2, 5, 4, 0])
>>> indexes = [1,4,5]
>>> x[indexes]
array([8, 5, 6])
```

We can do so in many dimensions:

```
>>> X=np.arange(12).reshape(3,4)
>>> index_row=np.array([0,1,2])
>>> index_col=np.array([1,2,1])
>>> X
array([[ 0,  1,  2,  3],
```

```
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> X[index_row,index_col]
array([1, 6, 9])
```

What happens is that you align each element in the row with the column. So you are indexing X[0,1], X[1,2], and X[2,1]. You can apply broadcasting rules to the index arrays. Figure 2.3.3.6 represents how this works:

Figure 2.3.3.7: How multidimensional masks work!

### Comparisons, Masks, Booleans

There are also a number of **ufuncs** which implement comparison element-wise. The result is an array of equal length to the original which is composed of True or False based on comparison made.

| Operator | Equivalent Ufunct |
|---|---|
| == | np.equal |
| != | np.not_equal |
| < | np.less |
| <= | np.less_equal |
| > | np.greater |
| >= | np.greater_equa |

Let us see a few of these at play and show their immense utility especially with regards to indexing. Here is a very simple example of their use:

```
>>> MEANING_OF_LIFE = 42
>>> rng=np.random.RandomState(MEANING_OF_LIFE)
>>> x=rng.randint(10,size=(3,4))
>>> x
array([[6, 3, 7, 4],
       [6, 9, 2, 6],
       [7, 4, 3, 7]])

>>> x==6
array([[ True, False, False, False],
       [ True, False, False,  True],
       [False, False, False, False]])
>>> x >=6
array([[ True, False,  True, False],
       [ True,  True, False,  True],
       [ True, False, False,  True]])
```

You could also do things with booleans, like counting how many fit some condition.

```
>>> np.sum(x<6)
5

>>> np.sum(x<6, axes=0)

array([0, 2, 2, 1])
```

You can do this by axis.

We can also check if *any* element or *all* elements follow some condition:

```
>>> np.any(x==6)
True
>>> np.all(x==6)
False
>>> np.any(x==6, axis=1)
True
```

The coolest thing though is that we can *index* with boolean expressions. It is such a common pattern to use boolean **ufuncs** to subset arrays, that this is an operation known as *masking*. We can combine conditions using bitwise operators. For instance:

```
>>> x_subarray=x[x<6]
>>> x
array([[6, 3, 7, 4],
       [6, 9, 2, 6],
       [7, 4, 3, 7]])
>>> #between 6 inclusive and 8 exclusive
>>> x[(x >= 6) & (x<8)]
array([7, 6, 6])
```



How do we write performative code in numpy? Well we could always get a bigger machine. What if we cannot? We can always write better algorithms that scale up. What if we cannot? There is something we can do that is really quite easy.

There is also a module named **numba** and it is simply awesome for producing efficient scientific code. We will go through some of the very, very basics here.

According to Numba's documentation:

" Numba is a compiler for Python array and numerical functions that gives you the power to speed up your applications with high performance functions written directly in Python.

Numba generates optimized machine code from pure Python code using the LLVM compiler infrastructure. With a few simple annotations, array-oriented and math-heavy Python code can be just-in-time optimized to performance similar as C, C++ and Fortran, without having to switch languages or Python interpreters.

Numba's main features are:

- on-the-fly code generation (at import time or runtime, at the user's preference)
- native code generation for the CPU (default) and GPU hardware
- integration with the Python scientific software stack (thanks to Numpy)

Refresher on decorators

Python is also a *functional language*, which means that "functions are first-class citizens". This means that functions can be passed to other functions directly by name (and not by pointer like in C). They can also *modify* the behavior of other functions and that is *precisely what a decorator is*. A decorator in python is basically syntactic sugar to modify the behavior of a function. Let us give an example. Let us say we want a function that prints the results of another function. We can write a function:

```
def print_result(func):
    """
    Print the result of the given function.

    Parameters:
    func (Callable): The function to execute and print its result.
    """
    result = func
    print(result)
```

But there is a niftier way of writing this, where we can invoke **func** with its arguments and have a decorator modify its behavior to print the result of the invocation.

```
def decorator_print_result(func):
    """
    Decorator that prints the result of the decorated function.

    Parameters:
    func (Callable): The function to wrap.

    Returns:
    Callable: The wrapped function that prints its result.
```

```
    """
    @wraps(func)
    def printer(*args, **kwargs):
        result = func(*args, **kwargs)
        print(result)
        return result
    return printer
```

This function precisely modifies the behavior of our original function , whatever it is. So for instance let us say we have a function that sorts a list in ascending order (do not worry about its implementation) , but here it is:

```
@decorator_print_result
def bubble_sort(x: list, in_place=False, out=None) -> list:
    """
    Sort a list using the bubble sort algorithm.

    Parameters:
    x (list): The list to sort.
    in_place (bool, optional): If True, sort the list in place. Default is False.
      out (list, optional): If provided, the sorted list will be stored in this
parameter.

    Returns:
    list: The sorted list.
    """
    n = len(x)
    for i in range(n):
        for j in range(i, n):
            if x[j - 1] > x[j]:
                swap = x[j]
                x[j] = x[j - 1]
                x[j-1] = swap
    return x
```

The following now has the desired modified behavior. The reason we use a decorator itself in this **@wraps** is because without that it transfers *meta-data* about the function from the modified to the modifying function. For instance, when we ask the modified function its name via **__name__** we will get the name of the decorator and not the original. We can run all this via to see the output:

```
"""
This program demonstrates the use of a decorator to print the result of a function.
It includes a bubble sort implementation that is decorated to print its result.

The main steps are:
1. Define a simple print function to print the result of a given function.
2. Define a decorator that wraps a function to print its result when called.
3. Implement the bubble sort algorithm and decorate it to print its sorted list.
4. Execute the bubble sort function on an example list.

Usage:
------
Simply run the script to see the output of the bubble sort function.

"""
from functools import wraps


def print_result(func):
    """
    Print the result of the given function.

    Parameters:
    func (Callable): The function to execute and print its result.
    """
    result = func
    print(result)


def decorator_print_result(func):
    """
    Decorator that prints the result of the decorated function.

    Parameters:
    func (Callable): The function to wrap.

    Returns:
    Callable: The wrapped function that prints its result.
    """
    @wraps(func)
    def printer(*args, **kwargs):
```

```
        result = func(*args, **kwargs)
        print(result)
        return result
    return printer


@decorator_print_result
def bubble_sort(x: list, in_place=False, out=None) -> list:
    """
    Sort a list using the bubble sort algorithm.

    Parameters:
    x (list): The list to sort.
    in_place (bool, optional): If True, sort the list in place. Default is False.
      out (list, optional): If provided, the sorted list will be stored in this
parameter.

    Returns:
    list: The sorted list.
    """
    n = len(x)
    for i in range(n):
        for j in range(i, n):
            if x[j - 1] > x[j]:
                swap = x[j]
                x[j] = x[j - 1]
                x[j-1] = swap
    return x


if __name__ == "__main__":
    example = [3, 3, 6, 7]
    bubble_sort(example)
    print_result(bubble_sort(example))
    bubble_sort(example)
```

Which gives output :

```
[3, 3, 6, 7]
[3, 3, 6, 7]
[3, 3, 6, 7]
```

```
[3, 3, 6, 7]
```

## Basics of Jit

The central feature of the Numba module is the **@jit** decorator. Jit means "just-in-time" and it is a method of compilation that saves the compilation to the last second as the program is run i.e compilation occurs at *run-time*. This decorator takes python and compiles it into efficient machine-code. Different invocation modes trigger different behaviors and compilation features.

There are two basic ways for the code to be compiled i.e. lazy versus eager evaluation. Lazy evaluation is just what it sounds like i.e. an expression or program is not evaluated or compiled until *run-time*. Eager evaluation means that the expression or program is compiled before *run-time*.

In Numba, lazy evaluation of a function means that the types are inferred at run-time, and based on that information optimized code is generated. For instance:

```
>>> from numba import jit
>>> @jit
>>> def f(x,y):
   return x-y

>>> f(1,2)
-1
```

The function **f** is in fact not compiled until the function itself is executed.

By contrast we can specify the function signature and this implies that we are eagerly evaluating the function:

```
>>> from numba import jit, float32
>>> @jit(float32(float32,float32))
def f(x:float,y:float):
   return x*y
```

```
>>> f(1.0,2.0)
2.0
```

The function will be compiled right at initialization of the function **def f**. The signature is:

```
(float32(float32,float32))
```

It has the form **output_type(input_type1, input_type2, …input_type_n).** An array is simply some type followed by [:]. So for instance a jit-compiled function of three arrays would be:

```
>>> @jit(float32[:](float32[:],float32[:],float32[:]))
def f(x,y,z):
   return x*y
>>> x = y = z = np.arange(5, dtype="float32")
>>> x
array([0., 1., 2., 3., 4.], dtype=float32)
>>> f(x, y, z)
array([ 0.,  3.,  6.,  9., 12.], dtype=float32)
```

There are some nifty options for this jit-decorator.

**nopython**

We can for instance specify whether or not the decorator “accesses” the Cpython interpreter via the **nopython** option. This is the default and it produces faster code in general, *but* it means that your function has to be very clear about what types are used i.e. the type must be inferabble with no ambiguity (have a unique signature) e.g.:

```
@jit(float32[:](float32[:],float32[:],float32[:]), nopython=True)
def f(x,y,z):
   return x*y
```

When you put **nopython=False** you utilize **object mode** and it generates code that does make use of Cpython and so of python objects. It will not increase performance *unless* you use a feature called loop-jitting.

**nogil**

We can also make use of the **nogil** option. The **GIL** is the global interpreter lock and it is a lock or mutex that allows the interpreter to only function with a single-thread. This protects python objects and prevents the interpreter from interacting with other processes and threads ensuring thread-safety and preventing race-conditions. If we want to run other python processes concurrently then we can disable the **GIL**:

```
@jit(float32[:](float32[:],float32[:],float32[:]), nopython=True, nogil=True)
def f(x,y,z):
   return x*y
```

This will not work with **object** mode.

**nocache**

Let us say we have to run the program compiling this function multiple times. Then it would be silly to *recompile* the function each time we execute the function. The **nocache** option allows us to save information on compilation to not have to recompile the function by storing necessary information in a *cache file*.

```
@jit(float32[:](float32[:],float32[:],float32[:]),      nopython=True,      nogil=True,
cache=True)
def f(x,y,z):
   return x*y
```

**parallel**

We can parallelize our numba-generated code via the **parallel** option. We will be going into *parallelism* in much, much greater detail later on but for now think of it, very vaguely, as running the *same job many times at the same time*. We can generate this parallelized code as such:

```
@jit(float32[:](float32[:],float32[:],float32[:]),nopython=True,nogil=True,
cache=True, parallel=True)
def f(x,y,z):
   return x*y
```

There are limitations of course, based on whether numba can convert serial into parallel code. Let us go through *some* of them as quoted here:

1. All numba array operations that are supported include common arithmetic functions between Numpy arrays, and between arrays and scalars, as well as Numpy ufuncs. They are often called element-wise or point-wise array operations:
    - unary operators: + - ~
    - binary operators: + - * / /? % | >> ^ << & ** //
    - comparison operators: == != < <= > >=
    - Numpy ufuncs that are supported in nopython mode.
    - User defined **DUFunc** through **vectorize()**.
2. Numpy reduction functions sum, prod, min, max, argmin, and argmax. Also, array math functions mean, var, and std.
3. Numpy array creation functions zeros, ones, arange, linspace, and several random functions (rand, randn, ranf, random_sample, sample, random, standard_normal, chisquare, weibull, power, geometric, exponential, poisson, rayleigh, normal, uniform, beta, binomial, f, gamma, lognormal, laplace, randint, triangular).
4. Numpy dot function between a matrix and a vector, or two vectors. In all other cases, Numba's default implementation is used.
5. Multi-dimensional arrays are also supported for the above operations when operands have matching dimension and size. The full semantics of Numpy broadcast between arrays with mixed dimensionality or size is not supported, nor is the reduction across a selected dimension.
6. Array assignment in which the target is an array selection using a slice or a boolean array, and the value being assigned is either a scalar or another selection where the slice range or bitarray are inferred to be compatible.

Now the **parallel** will fail for instance in the following case of a mutation of python object because there is no obvious way to ensure thread-safety:

```
@nb.jit(parallel=True)
def invalid():
   tmp = []
   for i in range(100):
       tmp.append(i)
```

Numba and Ufuncs

If you remember before we spoke about the **vectorize** function or decorator. The next decorators I want to *briefly* show you are called **vectorize and guvectorize**. The truth is the **vectorize** I showed you before was little more than a wrapper for a for-loop and so it does not improve performance. These versions however will!

**vectorize**

So **vectorize** like before runs either in lazy or eager mode. When you submit a function with a signature the function generates a numpy **ufunc**. It compiles the function right there without execution. When you evaluate it lazily it generates a **dynamic ufunc** known as **DUfunc** which generates a new function everytime a new type of input is utilized or executed through the function. An example taking the distance between two points:

```
"""
Exhibit the use of Numba and NumPy to create and use a universal function (ufunc).

This program defines a ufunc to calculate the magnitude,
optimized with Numba for performance. It then applies this ufunc to arrays of points.

Functions:
   magnitude(x: float, y: float) -> float: Computes the magnitude
   between points
 main() -> None: Initializes arrays of points and prints their magnitude.
"""
import numba as nb
import numpy as np


@nb.vectorize([nb.float32(nb.float32, nb.float32)])
```

```
def magnitude(x: float, y: float) -> float:
   return (x**2 + y**2) ** (1 / 2)


def main() -> None:
   points_x, points_y = (
       np.array([1.0, 2.0, 3.0, 4.0], dtype=np.float32),
       np.array([5.0, 6.0, 7.0, 8.0], dtype=np.float32),
   )

   print(f"magnitudes : {magnitude(points_x, points_y)}")


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

Which gives:

```
magnitudes : [5.0990195 6.3245554 7.615773  8.944272 ]
```

Notice that the signature must be put in a *list* , in fact in any *iterable*. If you pass multiple signatures you must pass more specific to less specific, lower byte precision before higher.

```
@nb.vectorize([nb.int32(nb.int32, nb.int32),
               nb.int32(nb.int64, nb.int64),
               nb.float32(nb.float32, nb.float32),
               nb.float64(nb.float64, nb.float64)])
def magnitude(x: float, y: float) -> float:
  return (x**2 + y**2) ** (1 / 2)
```

Why use this decorator? Well because we get to leverage the features , attributes, and methods of a **ufunc** including reductions and aggregations e.g. , **reduce** and **mean**. For a full list check out here. We can check the dimension number, or the order (c or fortran ordered). We can actually leverage **cpu** or **parallel** targets , the first recommended for small and the second medium-sized data and computational intensity.

**Guvectorize**

Now imagine we want to pass not two vectors into our universal function but two vectors, *themselves* having vectors. To do so we use the **guvectorize** , a decorator that produces a **gufunc** function that generalizes the **ufunc**. It allows us to build ufuncs that take arbitrary dimensional arrays and produces (perhaps different) arbitrary n-dimensional arrays.

Let us say for instance we want to know the distance between 2 points in 3-d space. We pass one array of 3-d points, and a second array of 3-d points. Each point is an array itself where x is the first element , z the last.  We have:

```
"""
This program computes the Euclidean distances between pairs of 3D points
using the Numba library to optimize the calculations. It defines a
generalized universal function (gufunc) to perform the distance calculations
efficiently on arrays of points.

The main steps are:
1. Importing necessary libraries (`numpy` for array handling and `numba` for
  performance optimization).
2. Defining the `compute_distance` function using `numba`'s `guvectorize`
  decorator to compute the Euclidean distance between pairs of 3D points.
3. Using example arrays of 3D points to demonstrate the computation of distances.

Usage:
------
Simply run the script to see the output of the distances between the example
points.

Example:
--------
Distances between points: [10.39230485  2.82842712 10.39230485]

Dependencies:
-------------
- numpy
- numba
```

```
To install the dependencies, use:
$ pip install numpy numba
"""
import numpy as np
from numba import guvectorize

# Define the generalized universal function using guvectorize
@guvectorize(['void(float64[:],      float64[:],      float64[:])'],      '(n),(n)->()',
nopython=True)
def compute_distance(point1, point2, result):
    """
    Compute the Euclidean distance between two points in 3D space.

    Parameters:
    point1 : array_like
        First point in 3D space, an array of shape (3,)
    point2 : array_like
        Second point in 3D space, an array of shape (3,)
    result : array_like
        Output array to store the computed distance
    """
    diff = 0.0
    for i in range(point1.shape[0]):
        diff += (point1[i] - point2[i]) ** 2
    result[0] = np.sqrt(diff)


if __name__ == "__main__":
    # Example arrays of 3D points
    points1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
    points2 = np.array([[9, 8, 7], [6, 5, 4], [3, 2, 1]])

    # Compute the distances
    distances = np.empty(points1.shape[0])
    compute_distance(points1, points2, distances)

    print("Distances between points:", distances)
```

Which gives :

```
    Distances between points: [10.77032961  2.82842712 10.77032961]
```

## Advanced Application: Learning Performance Through Monte Carlo

What if I told you that you can compute the value of pi utilizing a dart board and a dart? It turns out that via a monte carlo calculation we can simulate just this process. How does it work ?

Take a square and make it 2x2 units of area. Make a circle whose radius is 1 unit of length. Now throw n number of darts randomly , close your eyes and throw. Count how many landed in the circle and not the square (ok your blind accuracy has gotta be pretty good because the darts cannot exceed the square). Now multiply the area x hits/n. As n-> infinity you will get closer and closer to pi. Now let us write this in beautiful jitted code to show increases in performance. Then I want to take the lessons away and briefly state easy ways to improve performance.

The following program performs the monte carlo calculation while also testing various options:

```
"""
This program estimates the value of π (pi) using the Monte Carlo method with
different levels of optimization and precision. It uses the Numba library to
optimize the calculations and includes a decorator to measure the execution time
of each function.

The main steps are:
1. Importing necessary libraries (`numpy` for array handling, `numba` for
  performance optimization, `functools` for wrapping functions, and `time` for
  measuring execution time).
2. Defining a `timeit` decorator to measure and return the execution time of
  functions.
3. Implementing several versions of the Monte Carlo π estimation function:
  - `monte_carlo_pi`: A plain NumPy implementation.
  - `jitted_monte_carlo_pi`: A Numba-jitted implementation.
  - `less_precision_jitted_monte_carlo_pi`: A Numba-jitted implementation with
    less precision.
  - `fast_math_jitted_monte_carlo_pi`: A Numba-jitted implementation with
    `fastmath` enabled.
  - `parallel_jitted_monte_carlo_pi`: A Numba-jitted implementation with
    `fastmath` and parallel computation enabled.
```

```
4. Using the `main` function to execute and compare the performance and
  precision of these implementations.

Usage:
------
Simply run the script to see the output of the estimated values of π and the
execution times for each implementation.

"""
import time
from functools import wraps

import numba as nb
import numpy as np


def timeit(func):
    @wraps(func)
    def timeit_wrapper(*args, **kwargs):
        start_time = time.perf_counter()
        result = func(*args, **kwargs)
        end_time = time.perf_counter()
        total_time = end_time - start_time
        return result, total_time

    return timeit_wrapper


@timeit
def monte_carlo_pi(samples: (int | float), area) -> int | float:
    hit = 0
    samples_ = np.copy(samples)
    while samples_ > 0:
        samples_ = samples_ - 1
        x, y = np.random.rand(), np.random.rand()
        if (x**2 + y**2) ** (1 / 2) <= 1:
            hit += 1
    return area * hit / samples


@timeit
@nb.jit(nb.float64(nb.int64, nb.float64))
```

```
def jitted_monte_carlo_pi(samples: nb.int32, area: nb.float32) -> nb.float64:
    hit = 0.0
    for _ in nb.prange(samples):
        x, y = np.random.rand(), np.random.rand()
        if (x**2 + y**2) ** (1 / 2) <= 1.0:
            hit += 1.0

    return area * hit / samples


@timeit
@nb.jit(nb.float32(nb.int32, nb.float32))
def less_precision_jitted_monte_carlo_pi(
    samples: nb.int32, area: nb.float32
) -> nb.float32:
    hit = 0.0
    for _ in nb.prange(samples):
        x, y = np.random.rand(), np.random.rand()
        if (x**2 + y**2) ** (1 / 2) <= 1.0:
            hit += 1.0

    return area * hit / samples


@timeit
@nb.jit(nb.float32(nb.int32, nb.float32), fastmath=True)
def fast_math_jitted_monte_carlo_pi(samples: nb.int32, area: nb.float32) ->
nb.float32:
    hit = 0.0
    for _ in nb.prange(samples):
        x, y = np.random.rand(), np.random.rand()
        if (x**2 + y**2) ** (1 / 2) <= 1.0:
            hit += 1.0

    return area * hit / samples


@timeit
@nb.jit(nb.float32(nb.int32, nb.float32), fastmath=True, parallel=True)
def parallel_jitted_monte_carlo_pi(samples: nb.int32, area: nb.float32) -> nb.float32:
    hit = 0.0
    for _ in nb.prange(samples):
```

```
        x, y = np.random.rand(), np.random.rand()
        if (x**2 + y**2) ** (1 / 2) <= 1.0:
            hit += 1.0

    return area * hit / samples


def main() -> str:
    samples = 1e7
    area = 4.0
    num_execs = 25
    functions = [
        monte_carlo_pi,
        jitted_monte_carlo_pi,
        less_precision_jitted_monte_carlo_pi,
        fast_math_jitted_monte_carlo_pi,
        parallel_jitted_monte_carlo_pi,
    ]

    for func in functions:
        times, pis = [], []
        for _ in range(num_execs):
            pi, time = func(samples=samples, area=area)
            pis.append(pi)
            times.append(time)
        ave_pi, ave_time = np.mean(np.array([pis, times]), axis=1)
        print(f"{func.__name__}\npi ≈ {ave_pi}\ntime:{ave_time}\n")


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

Which gives output:

```
monte_carlo_pi
pi ≈ 3.141537248
time:5.809030539849773

jitted_monte_carlo_pi
pi ≈ 3.1417039680000003
```

```
time:0.04790139175951481

less_precision_jitted_monte_carlo_pi
pi ≈ 3.1415740394592286
time:0.04782529009506106

fast_math_jitted_monte_carlo_pi
pi ≈ 3.1415791797637937
time:0.04795218328945339

parallel_jitted_monte_carlo_pi
pi ≈ 3.141505012512207
time:0.012324998462572694
```

With this program, we define a number of functions making use of more and more options and features hoping to optimize our code. We run each function 25 times and then store the times and the pi calculation result taking the average of each as the result to print.

Now the first function **monte_carlo_pi** is just a serial , no numba monte carlo calculation. The next makes use of numba in eager-mode, specifying a signature. We see a speedup of over 100x ! The next function is **less_precision_jitted_monte_carlo**. We utilize less precise floats, **float32**, "only" 32 bits. In general, we should not use data types that require more precision than is needed. More precise types are more expensive to store, to perform manipulations on, etc. We do not see too much of a performance difference here. The next function is **fast_math_jitted_monte_carlo** which is an option at the compiler level that may make our calculations less precise, degrades precision (therefore use it *only if precision is not the concern* ). The next makes use of LLVM-flags for fast math operations, we can restrict which mathematical operations actually get "fastmathed". Check out what these flags mean here. We do not see much improvement. Finally, we have **parallel_jitted_monte_carlo,** which makes use of **parallel** or automatic parallelization. The speedup is immense ! We have sped up our program by almost 500x!

So some lessons in general (though this example maybe did not show these):

1. Use **nopython** mode
2. Make use of numpy ufuncs
3. Use *only* amount of precision needed
4. Use **fastmath** compiler optimization
5. Parallelize when possible

## 3. Message-Passing Interface with Python

What is a message?

How does this apply to parallel computation

Very often

On Python and MPI: mpi4py

Pickling

Marshaling

### 3.1 Communicators:

A communicator is an object that links a group of *processes* and allows them to coordinate and communicate.

The communicator that includes all processes is called COMM_WORLD, and by default is instantiated by:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
```

It sets up ids for each processor and it organizes the processors in an **ordered topology**. The processes compose a **group**, a set of processes. We identify each process within the 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:

```
"""
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()
```

To run this stupid program, we use :

```
Unset
mpiexec -n 4 hello_world.py
```

Which gives output:

```
Hello world! I am process 0 and the size of my group size is 3
Hello world! I am process 2 and the size of my group size is 3
Hello world! I am process 1 and the size of my group size is 3
```

What is happening in this program ? Globally, the program gets distributed to each processor, and then it runs in each processor. The 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.

Mpiexec is a command from the mpi module, it is a complex command which one can investigate here. It is also a synonym for mpirun. All we did was compile the program, and we did so by indicating the number of processes via -n ${n} and the application to run, here ./bin/python3.12 (the python in our virtual environment). We utilized the -m flag to indicate the module mpi4py and then we imputed the filename. For more on mpiexec and mpirun consult section 4.3.1.

## 3.2 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**.

### 3.2.1 Send and Receive

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:

```
"""
```

```
Have each worker node produce a random integer, then send it to the root process to be
printed by root.

Date: 05/13/2024

"""

from mpi4py import MPI
from random import randint


def main():

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

   LOWER_LIMIT = 1
   UPPER_LIMIT = 10

   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=r)
           print(f"my name is rank 0 and I received the number {data} from rank {r}")


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

*Program 3.2.1 A program to send messages from a root (process 0) to other processes which print message*

When we run:

```
Unset
mpiexec -n 4 ./bin/python3.12 -m mpi4py
../examples/communication/point-to-point/pure_python/send_receive.py
```

We get:

```
my name is rank 0 and I received the number 3 from rank 1
my name is rank 0 and I received the number 10 from rank 2
my name is rank 0 and I received the number 8 from rank 3
```

Of course this program will not give the same output, since we make use of the python module's randint() function.

In this program we created our communicator object, grouping together 4 processes and assigning them each a rank. We then accessed the rank of each process, and the size of the group. We set a lower and upper limit to the random integer to be drawn since randint has function signature randint(a : int, b:int) -> int , meaning that the function takes two inputs , a and b both integers, and draws a number between [a,b] where a is less than b. Then we specify whether a process has a rank 0 or not. Implicitly within this program we are utilizing a **controller-worker paradigm** (archaically known as master-slave) which is to say a process, with rank say 0, is treated as a head node which coordinates the data and computation throughout all the processes. Here process 0 is a controller node, the others are workers.

The other nodes produce random integers and then send that data to the controller node. These worker nodes utilize the **send** function , which is a *method* of a communicator, here COMM_WORLD. The function signature is :

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

Obj is any python object, or data, to be sent. It can be integers, dictionaries, lists, even functions! The dest argument means destination, where the message is sent. Finally a **tag** is yet another piece of information, an attribute, which *identifies the message just as rank identifies processes*.

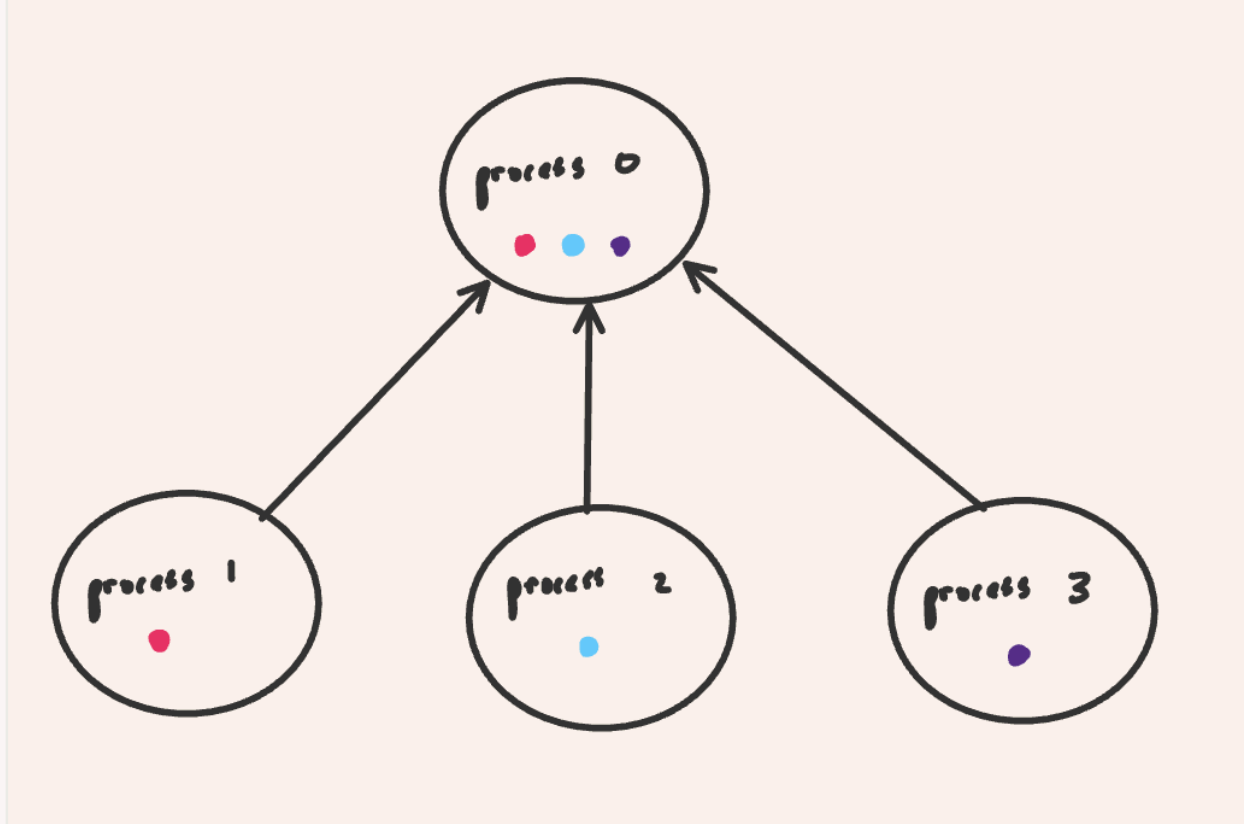



*Fig 3.2.1.1 Send-receive paradigm : A one-to-one relation between processors, above program depicted. Head-worker is process 0, each circle color is an individual piece of data. Notice the pieces of data all differ, and they are sent from the worker nodes to the head node.*

There is a cool modification to the program above. Say we are a processor and want to receive data from any other processor that is sending a message, data, instructions. Then we do not have to specify the source explicitly we can simply write:

```
from mpi4py import MPI
from random import randint
```

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

LOWER_LIMIT = 1
UPPER_LIMIT = 10

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}")
```

Making use of the **MPI.ANY_SOURCE** object, which obviously indicates to the program that the message can come from *any source*. Now There is a final object I would like to introduce here:

```
from mpi4py import MPI
from random import randint

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

LOWER_LIMIT = 1
UPPER_LIMIT = 10

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}")
```

The **MPI.Status()** object , which once passed to a receive function , tells us information about the point-to-point communication between processes. That object has a number of attributes, such as byte_count, and error status. We can 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. The output of:

```
Unset
mpiexec -n 3 ./bin/python3.12 -m mpi4py
../examples/communication/point-to-point/pure_python/status.py
```

Is:

```
my name is rank 0 and I received the number 8 from rank 1
The status object :
 source: 1
 tag: 1
 byte_count : 5
my name is rank 0 and I received the number 6 from rank 2
The status object :
 source: 2
 tag: 1
 byte_count : 5
```

There are a number of other methods and attributes for the Status object which can be found here.

Now it is important to note that 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).

### 3.2.2 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
```

Sendobj is obviously the object to be sent. Dest is the destination process. Sendtag , the way to identify the sent message. Recvbuf is a memory buffer (we will explain what this later in the Numpy Objects section). Source is the source process of the message, recvtag is a tag to identify the received message. Status is a status object, but this argument is purely optional. The function will return *any* python object, including nothing, known as None in python.

An example of its usage is below:

```
"""
Have each node produce a random integer, then the two nodes exchange their random
integers.

Date: 05/13/2024

"""
```

```
from mpi4py import MPI
from random import randint


def main():

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

    assert size == 2
    LOWER_LIMIT = 1
    UPPER_LIMIT = 10

    if rank == 0:
        local_int = randint(LOWER_LIMIT, UPPER_LIMIT)
        new_local_int = comm.sendrecv(sendobj=local_int, dest=1, source=MPI.ANY_SOURCE)
        print(f"my name is rank 0 and I received the number {new_local_int} from rank
{rank}")
    else:
        local_int = randint(LOWER_LIMIT, UPPER_LIMIT)
        new_local_int = comm.sendrecv(sendobj=local_int, dest=0, source=MPI.ANY_SOURCE)
        print(f"my name is rank 0 and I received the number {new_local_int} from rank
{rank}")


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

When we run the program via:

```
Unset
mpiexec -n 2 ./bin/python3.12 -m mpi4py
../examples/communication/point-to-point/pure_python/sendrecv.py
```

We get:

```
my name is rank 0 and I received the number 10 from rank 1
my name is rank 0 and I received the number 9 from rank 0
```

This is depicted below in figure 3.2.2.1.

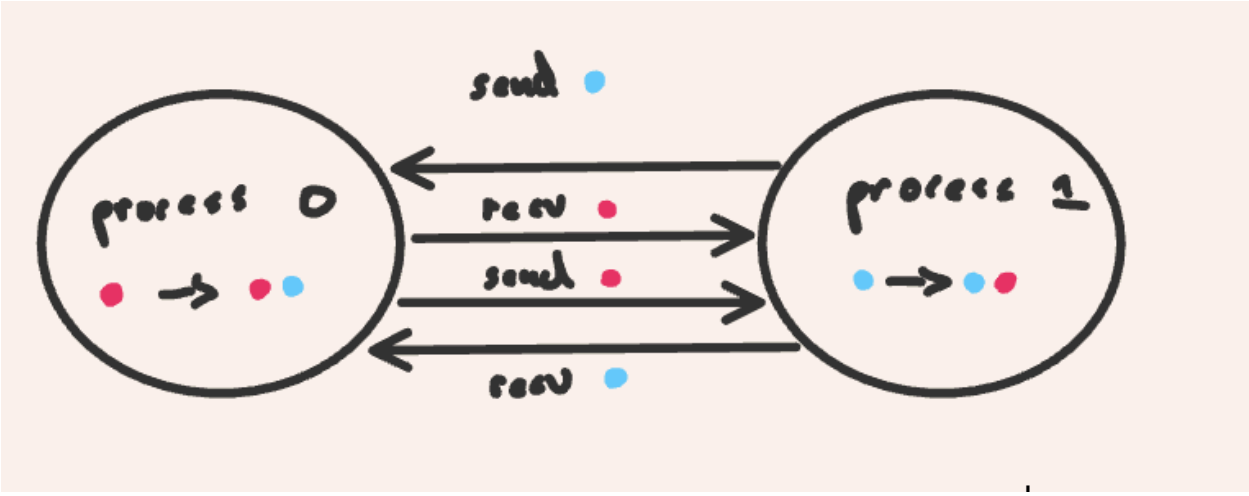



*Figure 3.2.2.1 Sendrecv paradigm. This works like a trade between the two processors. However ,the local data is preserved as well.*

### 3.2.3 Advanced Application : Integrals and Riemann Sums

The above programs are pedagogical and extremely rudimentary. Let us see how we would actually utilize the send and receive paradigm for a real world scientific application.

The calculus of infinitesimals, or just calculus, begins and is built on two mathematical operations i.e. the 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* (this is extremely basic and unrigorous but bear with me). The mathematical statement above is expressed notationally as:

$$\int_{a}^{b} f(x)\, dx$$

So for instance let us take the function x between 0 and 1 which notationally is :

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

Now we can see graphically in figure 3.2.3.1 that the area under the curve is ½. The area is ½ because we have a triangle with base 1, and height 1, and the formula for the area of a triangle is ½* base *height.

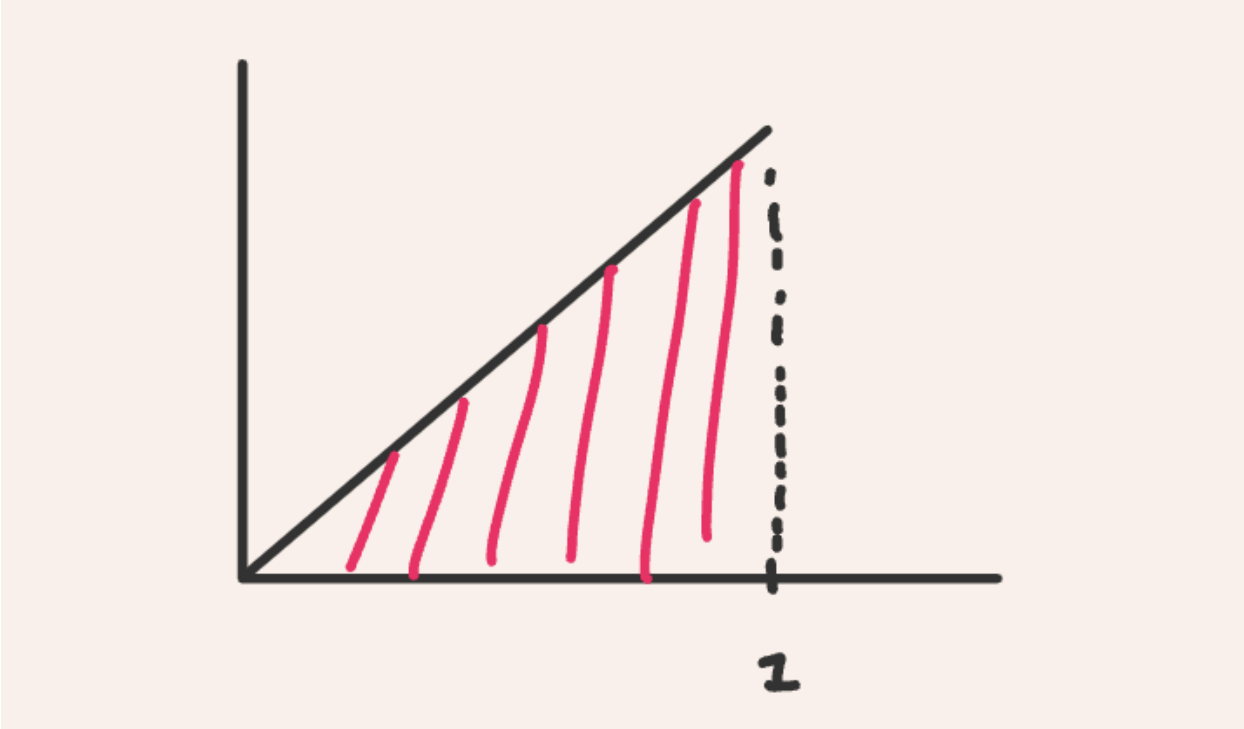



*Figure 3.2.3.1 : Area under the curve is ½ .*

Now there are symbolic rules for computing these areas under the triangle , and there are indeed ways of computing these integrals utilizing symbolic computing. But there is a way of approximating these integrals and in fact in some sense defining these integrals via a simple method that makes use only of addition and multiplication!

This method is called the riemann sum (there are nuances between left, middle and right riemann sums but we will ignore those here).

We can see the area under the curve as made of little smaller areas. Let us say we approximate the area under the curve as a bunch of rectangles. Then we would have something as depicted in figure 3.2.3.1.

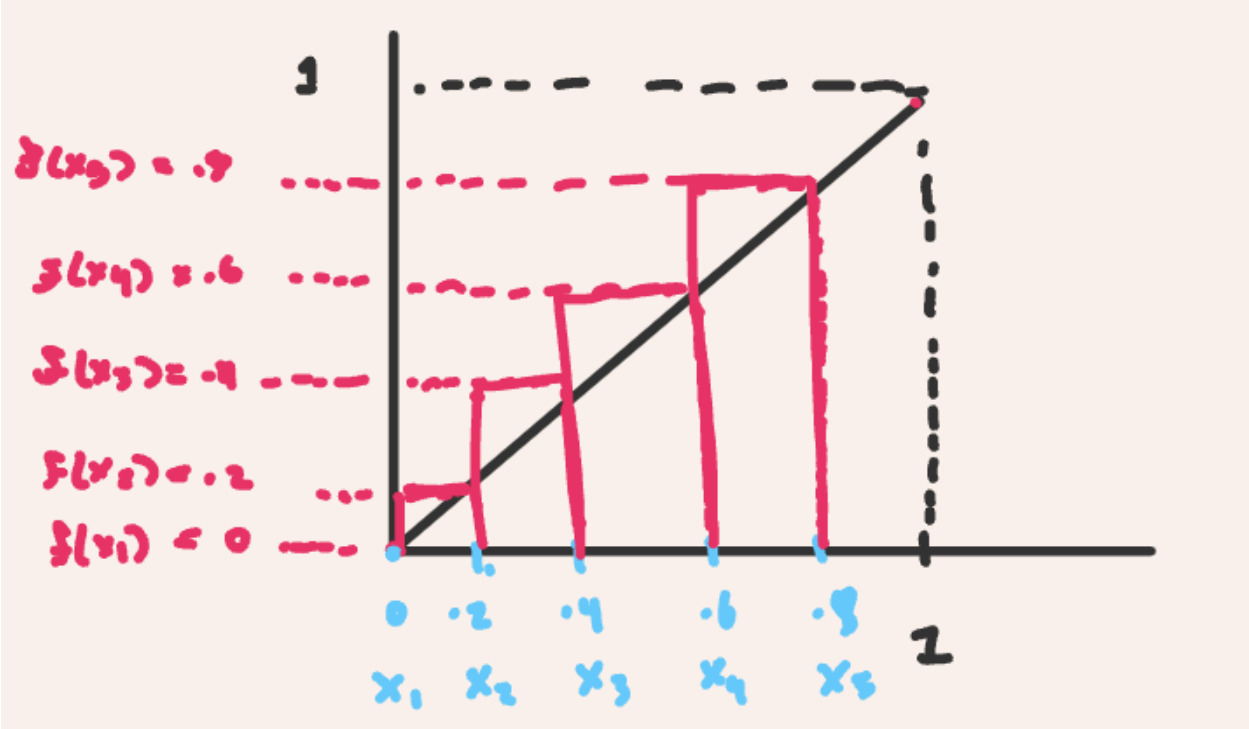
*Figure 3.2.3.2: Splitting up the area into rectangles.*

Now if I made the base of each rectangle smaller and smaller, and I added more and more rectangles it turns out that in the limit that the base of each rectangle becomes infintesimal that the sum of these rectangles *is* the integral defined above. Let us say we want to simply approximate the integral though. Let us choose there to be n rectangles. Now each rectangle is a **partition** and we label it $n_1, n_2 \ldots n$. Each x is placed in intervals of $\Delta x =$(b-a)/n and we label those x's $x_1, x_2 \ldots x_n$. The height of the

rectangle is then $f(x_i)$ for an arbitrary point i. Then the area of an arbitrary rectangle is $f(x_i) * \Delta x$. We then sum up all these rectangles and express this notationally as

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

We depict the partition process below in figure 3.2.3.3 for f(x)=x, [a,b]=[0,1], and n=5. Of course 5 partitions will give us a terrible approximation, but as we increase the number of partitions we get closer and closer to the actual answer at the expense of having to do more and more computation i.e. we have accuracy versus cost.



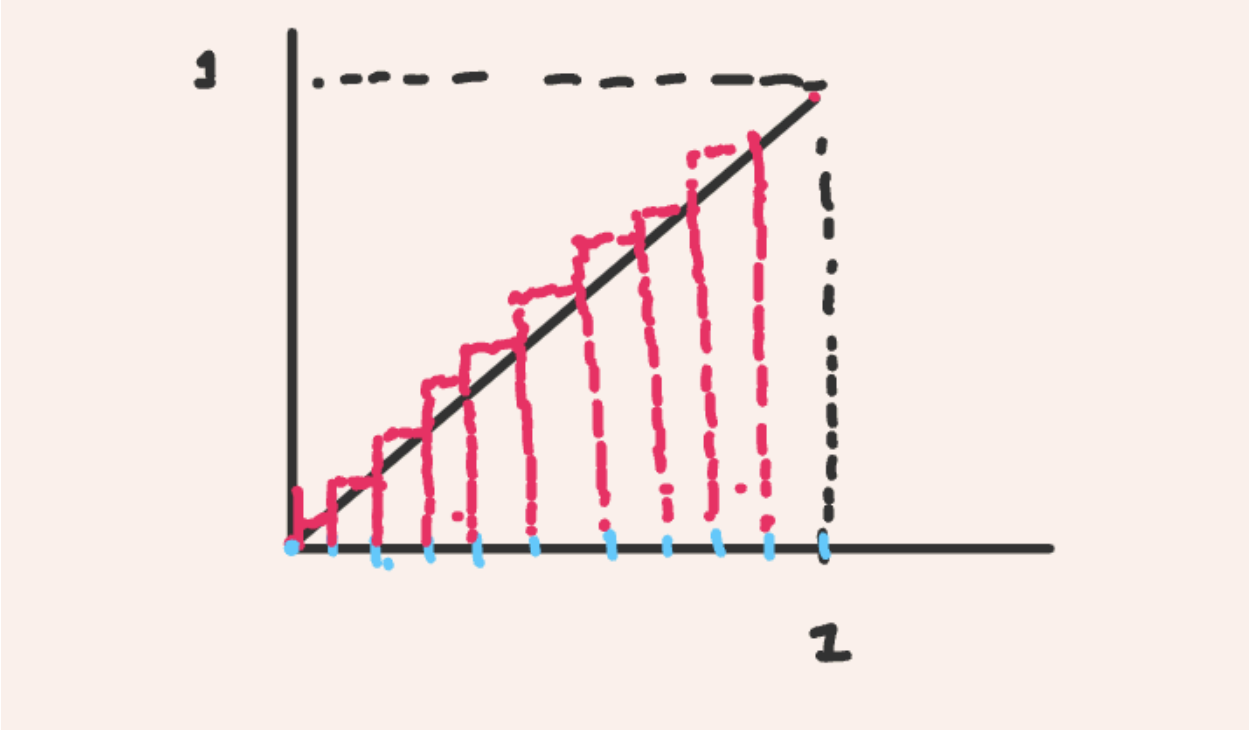



*Figure 3.2.3.3: The left-sided Riemann Sum for f(x)=x, [a,b]=[0,1] , and n=5. The sum = .4 which is an underestimate. The next figure Depicts the approximation getting closer, notice with n=10 that the overall area gets closer and closer to the triangle we want.*

This is simple enough to implement serially. I also added some pythonic magic for those interested:

```
"""
Perform a left-sided riemann sum given user-inputed function, interval, and a
partition number.

Date: 05/14/2024
Author: Djamil Lakhdar-Hamina
Last Modified : 05/15/2024

"""

from typing import Callable
```

```
def is_single_line(expression:  str) -> bool:
    """
    Takes a string expression and checks if the string is one-line by counting
    /n (newline) characters.

    Parameters:
    - expression: the function body in string form

    Returns:
    A boolean indicating if function body is single-lined (true)

    """
    # Count the number of newline characters
    newline_count = expression.count('\n')
    # Return True if there is only one newline character, False otherwise
    return newline_count == 0


def create_function_from_user_input() -> Callable:
    """
    Dynamically create a single-lined function from user input.

    Parameters:

    Returns:
    A single-lined function of form f(x): return expression
    """
    locals_dict = {}
    # Get a string from the user
    function_string = input("Enter a single-lined function body: ")
    # Check that string is well-formed
    assert is_single_line(function_string)
    # Execute code while making sure that scope of f is not just in exec bloc
    exec(f"def f(x): return {function_string}", {}, locals_dict)
    return locals_dict['f']

def riemann_sum_left(f: Callable[[float, int], float],
                     interval: tuple, n: int) -> float or int:
    """
    Takes function f, an interval (a,b) defined as a tuple, and a partition number n
    and approximates the integral of f on [a,b]
```

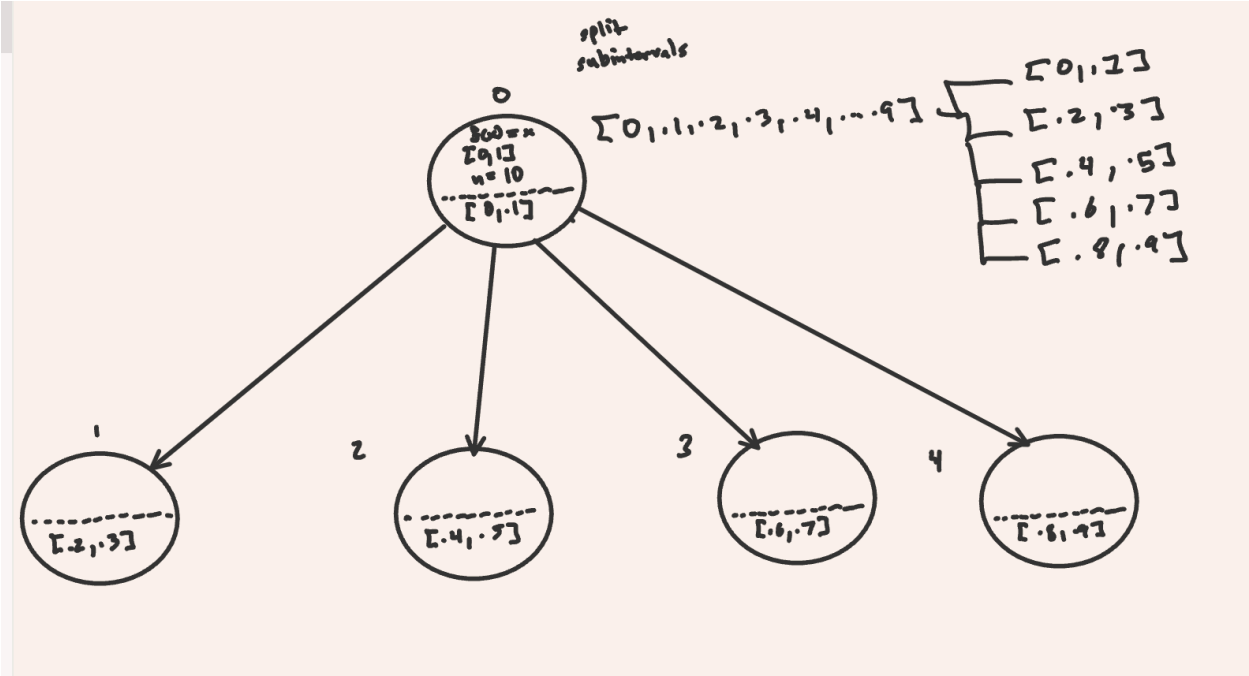
```
    Parameters:
    - f: function
    - interval: interval (a,b) where b>a
    - n : partition number, how many rectangles to approximate integral

    Returns:
    integral, the result of integration process, "area under the curve".

    Example:
    """

    assert interval[1] > interval[0]

    delta_x = (interval[1]-interval[0])/n
    x = [interval[0] + i*delta_x for i in range(0, n)]
    sum = 0
    for i in range(0, n):
        sum += f(x[i])*delta_x
    return sum


def main():

    f = create_function_from_user_input()

    assert callable(f)

    a, b, n = input('''Enter start of interval,\
end of interval, and partition number: ''').split()

    # type convert into appropriate format
    a , b = tuple(map(float, [a, b]))
    n = int(n)

    assert a < b

    result = riemann_sum_left(f, (a, b), n)
    print(f"The integral of {f} on [{a},{b}] ghiven n={n}:\n result: {result}\n")


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

The program is simple enough. The user inputs a single-line that defines a function. The program makes use of the **.split()** method, which will allow you to enter three inputs at once, each input separated by a white-space. The user then inputs the interval [a,b] and then the number of partitions. That is what **create_function_from_user_input()** does. After some type conversions and error checking, the program then calculates the Riemann sum. The heart of the program is defined by a function **riemann_sum_left**() with signature :

```
(function) def riemann_sum_left(
    f: (float, int) -> float,
    interval: tuple,
    n: int
) -> float
```

First we, take the function ,it can be a function of floats or of integers, and it returns floats. Here it is $3x^2$. The interval we define as a tuple (a,b). Finally, the partition is just an integer n. The function returns a float. The function itself proceeds by checking that b>a, then it defines a delta_x,which above we wrote as $\Delta x$. It then produces a list of evenly spaced points to be evaluated to give the height of each rectangle. Finally, we go through each point, each rectangle, compute the area of the rectangle and sum up the area of all the rectangles. Run the program yourself, you have your very own integral approximator!

But how can we speed-up the Riemann sum process? The simplest answer is by deploying a single-instruction, multiple-data framework where we perform a *data-partition for a task across multiple processes*. None of the various computations of the areas depend on one another. We can calculate the areas independently assigning a subset of the intervals and areas to different processes. We depict this using the point-to-point send and receive functions below in figure 3.2.3.4:




The program is parallelized below in a point-to-point fashion:

```
"""
Perform a left-sided riemann sum given user-inputed function, interval, and a
partition number
in a parallel fashion making use of send and receive.

Date: 05/14/2024
Author: Djamil Lakhdar-Hamina
Last Modified : 05/15/2024

"""

from mpi4py import MPI
from typing import Callable, Iterable, Type, List
from itertools import islice


def batched(iterable: Iterable, iterable_type : Type, n: int or float) ->
List[Iterable]:
    """
    Takes an iterable and breaks it up into iterable_type iterators of n size .

    Parameters:
```

```
    - iterable: the iterator to break up
    - iterable_type : the type of iterator to be broken up into
    - n : the number of elements in the iterator

    Returns:
    A list of iterators with number of elements n

    """
    # batched('ABCDEFG', 3) → ABC DEF G
    if n < 1:
        raise ValueError('n must be at least one')
    it = iter(iterable)
    while batch := iterable_type(islice(it, n)):
        yield batch


def is_single_line(expression:  str) -> bool:
    """
Perform  a  left-sided  riemann  sum  given  user-inputed  function,  interval,  and  a
partition number
in a parallel fashion making use of send and receive.

Date: 05/14/2024
Author: Djamil Lakhdar-Hamina
Last Modified : 05/15/2024

"""

from mpi4py import MPI
from typing import Callable, Iterable, Type, List
from itertools import islice


def  batched(iterable:  Iterable,  iterable_type  :  Type,  n:  int  or  float)  ->
List[Iterable]:
    """
    Takes an iterable and breaks it up into iterable_type iterators of n size .

    Parameters:
    - iterable: the iterator to break up
    - iterable_type : the type of iterator to be broken up into
    - n : the number of elements in the iterator
```

```
    Returns:
    A list of iterators with number of elements n

    """
    # batched('ABCDEFG', 3) → ABC DEF G
    if n < 1:
        raise ValueError('n must be at least one')
    it = iter(iterable)
    while batch := iterable_type(islice(it, n)):
        yield batch


def is_single_line(expression:  str) -> bool:
    """
    Takes a string expression and checks if the string is one-line by counting
    /n (newline) characters.

    Parameters:
    - expression: the function body in string form

    Returns:
    A boolean indicating if function body is single-lined (true)

    """
    # Count the number of newline characters
    newline_count = expression.count('\n')
    # Return True if there is only one newline character, False otherwise
    return newline_count == 0


def create_function_string_from_user_input() -> str:
    """
    Dynamically create a single-lined function string from user input.

    Parameters:

    Returns:
    A single-lined function of form f(x): return expression
    """
    # Get a string from the user
    print("Enter a single-lined function body: ")
```

```
    function_string = input()
    # Check that string is well-formed
    assert is_single_line(function_string)
    # Execute code while making sure that scope of f is not just in exec bloc
    return function_string


def create_function_on_process(function_string: str) -> Callable :
    """
    Compiles a function from a function string on a process.

    Parameters:
    - function_string: string or body of function

    Returns:
    The function with body given

    """
    locals_dict = {}
    exec(f"def f(x): return {function_string}", {}, locals_dict)
    return locals_dict['f']


def riemann_sum_left(f: Callable[[float, int], float],
                     x: List[float or int],
                     delta_x: float or int) -> float or int:
    """
    Takes function f, an interval (a,b) defined as a tuple, and a partition number n
    and approximates the integral of f on [a,b]

    Parameters:
    - f: function
    - interval: interval (a,b) where b>a
    - n : partition number, how many rectangles to approximate integral

    Returns:
    integral, the result of integration process, "area under the curve".

    Example:
    """
    sum = 0
    for x_i in x:
```

```
        sum += f(x_i)*delta_x
    return sum


def main():

    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    global_sum = 0

    # make sure local copies on each processor
    f = function_string = a = b = n = delta_x = local_interval = None

    if rank == 0 :

        # Define function on root process
        function_string = create_function_string_from_user_input()
        f = create_function_on_process(function_string)
        # Check that it was "compiled"
        assert callable(f)

        # User input for integral
        print('''Enter start of interval,\
end of interval, and partition number: ''')
        a, b, n = input().split()

        # Type convert into appropriate format
        a , b = tuple(map(float, [a, b]))
        n = int(n)

        # Check that user inputs fulfill conditions
        try :
            if n % size != 0:
                raise ValueError("Number of partitions \
isn't evenly split between processes")
            if a >= b:
                raise ValueError("The lower limit of the interval\
greater or equal to upper")
        except ValueError as e:
            print(e)
```

```
        # Evenly partition the interval into subintervals
        # of size partition space, one per process
        delta_x = (b-a)/n
        x = [a + i*delta_x for i in range(0, n)]
        partition_size = n // size
        batched_x = list(batched(x, list, partition_size))

    comm.Barrier()

    # 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, a , b, n , delta_x)
            comm.send(info, dest=i)
    else:
        local_interval, function_string, a , b, n , delta_x = comm.recv(source=0)
        f = create_function_on_process(function_string)

    # Perform the local summation then send back
    if rank != 0:
        local_sum = riemann_sum_left(f, local_interval, delta_x)
        comm.send(local_sum, dest=0)
    else:
        global_sum = local_sum
        for process in range(1, size):
            local_sum = comm.recv(source=process)
            global_sum += local_sum
        print(f"sum: {global_sum}")

    return global_sum


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

This program looks like a doozy and it is indeed doing a lot. However, go to the **main()** section and ignore the functions before. Follow the code as we describe what happens.

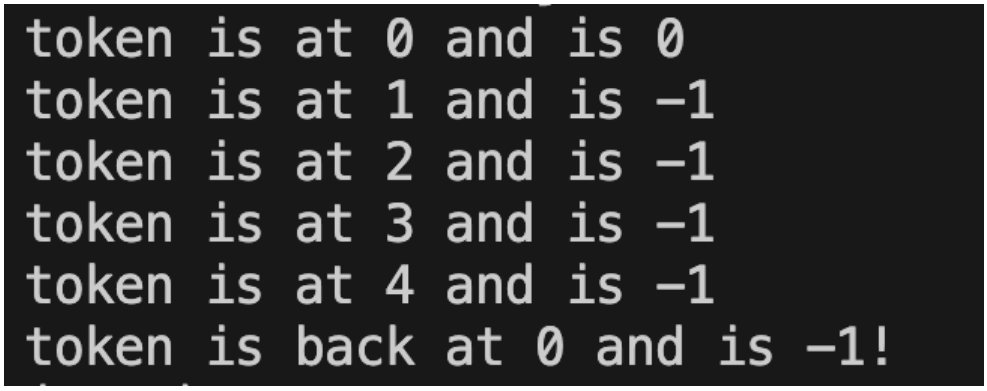
First, what is occurring is that process 0 is serving as a collector of user input. We first create a function string out of a function body as dictated by the user via **create_function_string_from_user()**. We then create a function *for that process* via **create_function_on_process()** and then test if that object was "compiled" as a function via an assertion (callable checks if the object is indeed a function). Then we ask for the user to input the beginning and end of interval and the number of partitions. We make use of **.split()** method which breaks up the input into three variables based on the use of a white-space separation by the user. We do some error checking to see if the number of partitions can be evenly divided between processes and to check if the beginning of the interval is smaller than the end. Finally, we actually build the list of subintervals , where each subinterval i corresponds to a process i.

Ignore the **comm.Barrier()** we will come back to this later.

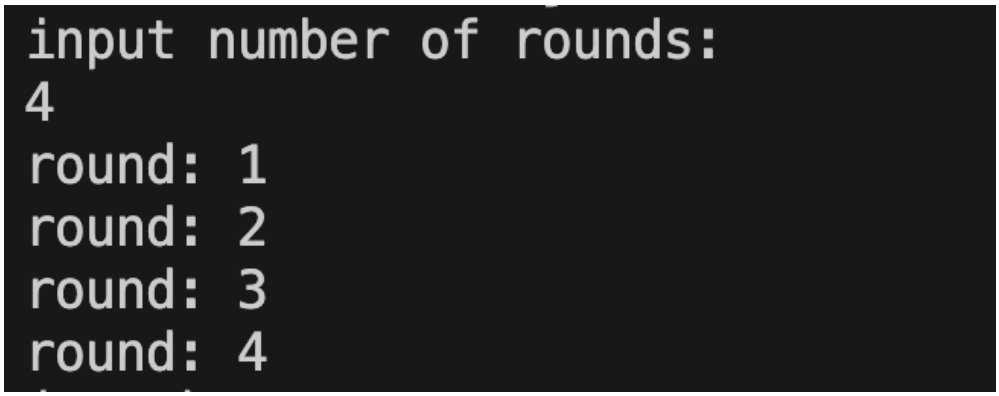
Next, the process 0 computes its Riemann sum for the subinterval 0. Then it *sends* the subinterval and a number of other pieces of information including the function_string input above, the subinterval, and the delta_x (the spacing between points). The other processes *receive* the info, the function is then compiled. Finally each process computes its local sum. They send all their sums back to the process 0 and process 0 then adds them up. Phew!

### 3.2.4 Exercises

1. * Run the parallel riemann_sum_parallel_naive.py program yourself do it for different inputs, take notes on results.
2. ** Program a ring. This program passes a token around processors in order.




3. Program a ping pong program, this process has a token sent between two processes n number of times. Output is like :




4. *** Implement the midpoint and trapezoidal rules for Riemann sums.
5. *** Write an improved Riemann sum program. Think about different strategies for replacing serial parts with concurrent parts e.g. the process 0 generates all the different subintervals, can we distribute this task across processes?
6. **** Generalize the Riemann sum program to 2, 3 and n dimensions!
7. **** There is a distinct method to calculate integrals called a monte carlo integration. Learn about monte carlo integration. Implement monte carlo integration.

## 3.3 Collective Communication

Now we will develop the concept of **collective communication** , which means communication which involves *all* processes, communication performed by collective functions. Whereas send and receive are binary (send a message from one process to another process), there are

functions which involve all the processes in the communicator. We will now go through some of these functions.

### 3.3.1 Scatter-gather

Say we wanna break up some data , and send different parts of the data i.e. a process called **sharding** the data. This is depicted in figure 3.3.1.1.

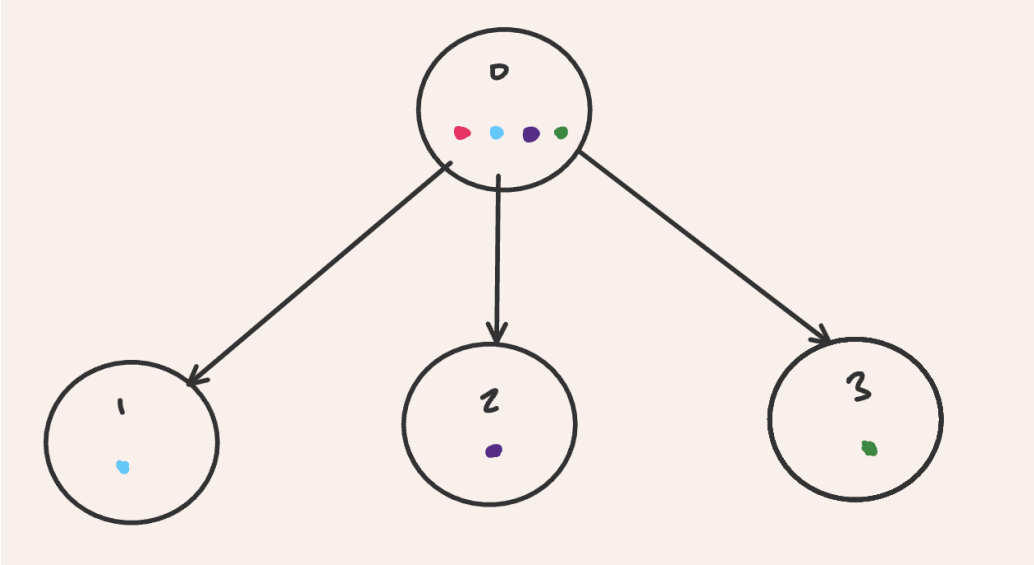



*Figure* 3.3.1.1 : *The dots represent data that is scattered across all processes. Notice that the red dot stays in the 0 rank process.*

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.**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**.

*Figure 3.3.1.2: Gather is depicted.*

How would we accomplish this **scatter-gather** action? Let us reflect a bit on **scatter** and **gather** in python. Scatter and gather are methods of a communicator, and they have the signature:

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

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

The scatter method takes a sequence of any objects, a sequence being for instance a list, or a tuple and “any '' meaning quite literally any python object e.g. a function,a list, an integer. It also takes a root integer, this being the rank of the process we scatter the data *from*. It then returns any sort of python object. Gather is similar except there is a subtle return. The gather function returns all of the objects sent from the various processes and puts them in a list! This stupid program shows this:

```
"""
Scatters and gathers a list

Date: 05/16/2024
Author: Djamil Lakhdar-Hamina

"""

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()
```

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. A vector is a mathematical object which for *our purposes* can be represented as a list of numbers where each number is an **element** of the vector (this is beyond simplistic but this is not a linear algebra course where a vector is a mathematical object in vector space such that the object satisfies certain conditions). 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?

Well we shard the vectors , add those shards together and then reassemble the overall result of the vector addition. We scatter the shards and then gather the result. Below is the code:

```
"""
Shards two vectors into two disjoint set of pieces, scatters the shards, adds the
shards
then reassembles the shards into an overall vector.

Date: 05/16/2024
Author: Djamil Lakhdar-Hamina

"""

from mpi4py import MPI
from random import randint
```

```python
from itertools import islice, chain
from typing import Iterable, Type, List, Generator


def part_gen(iterable: Iterable, iterator_type: Type,  n: int) -> Generator:
    """
    Takes an iterable and breaks it up into iterable_type iterators of n size .

    Parameters:
    - iterable: the iterator to break up
    - iterable_type : the type of iterator to be broken up into
    - n : the number of elements in the iterator

    Returns:
    A generator of iterators with number of elements n

    """
    # batched('ABCDEFG', 3) → ABC DEF G
    if n < 1:
        raise ValueError('n must be at least one')
    it = iter(iterable)
    while batch := iterator_type(islice(it, n)):
        yield batch


def partition(iterable: Iterable,
              outer_iterator_type: Type,
              inner_iterator_type: Type,
              n: int) -> Iterable[Iterable]:
    """
    Takes an iterable and breaks it up into iterable_type iterators of n size .

    Parameters:
    - iterable: the iterator to break up
    - outer_iterator_type: the iterator to wrap inner pieces in
    - iterable_type : the type of iterator to be broken up into
    - n : the number of elements in the iterator

    Returns:
    A actual iterator of iterators with number of elements n

    Example:
```

```
    >>> partition([1,2,3,4,5,6],tuple, list,2)
        ([1,2],[3,4],[5,6])

    """
    return outer_iterator_type(part_gen(iterable, inner_iterator_type, n))


def flatten(outer_iterable: Iterable, iterable: Iterable[Iterable]) -> Iterable:
    """
    Flatten an iterable into a one-dimensional outer_itreable.

    Parameters:
    - outer_iterable: the flattened iterator type
    - iterable: the iterator of iterators to be flattened

    Returns:
    An actual one dimensioal iterator.

    Example:

    >>> flatten(list, [[1,2,3],[1,2]])
    [1,2,3,1,2]

    """
    return outer_iterable(chain.from_iterable(iterable))


def rand_vector(lower_limit: int,
                upper_limit: int , n) -> list[int]:

    """"
    Produces a random vector whose elements are between
    lower limit and upper limit of length n.
     Parameters:
    - lower_limit: lower limit of element
    - upper_limit: upper limit of element

    Returns:
    list of integers
    """
    return [randint(lower_limit, upper_limit) for _ in range(0, n)]
```

```
def produce_zero_vector(n: int) -> list[int]:
    """"
    Produces a vector of zeros of length n
     Parameters:
    - n: length of vector

    Returns:
    list of zeros
    """
    return [0 for _ in range(0, n)]


def vector_sum(x: list, y: list) -> list:
    """
    Add two lists (vectors) element wise.

    Parameters:
    - x : first vector of size n
    - y : second vector of size (n) or else program fails

    Returns:
    a vector or list representing x+y element-wise
    """

    try :
        if len(x) != len(y):
            raise ValueError("the length of x and y do not match")
        if type(x[0]) is not type(y[0]):
            raise ValueError("the type of x and y do not match")
        z = [0 for _ in range(0, len(x))]
    except ValueError as e:
        print(e)

    for i in range(0, len(x)):
        z[i] = x[i]+y[i]

    return z


def main():
```

```
    # define some constants
    ROOT = 0
    LOWER_LIMIT = 1
    UPPER_LIMIT = 10
    VECTOR_LENGTH = 10

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

    assert size > 1

    try :
        if rank == 0 :
            if VECTOR_LENGTH % size != 0 :
                raise ValueError("elements are not evenly partitioned between\
    processors")
    except ValueError as e:
        print(e)

    chunk = VECTOR_LENGTH//size

    if rank == 0 :

        # produce two random vectors
        x , y = rand_vector(LOWER_LIMIT, UPPER_LIMIT,
VECTOR_LENGTH),rand_vector(LOWER_LIMIT, UPPER_LIMIT, VECTOR_LENGTH)

        # partition them or shard them, one shard for each process including root!
        partitioned_x, partitioned_y = partition(x, list, list, chunk), partition(y,
list, list, chunk)

        # build a list of tuples, the first element is x chunk, second y chunk
        x_y = [(x_chunk, y_chunk) for (x_chunk, y_chunk)
               in zip(partitioned_x, partitioned_y)]
    else:
        x_y = None

    x_y = comm.scatter(x_y, root=ROOT)
    local_x, local_y = x_y[0], x_y[1]
    sum = vector_sum(local_x, local_y)
```

```
    result = comm.gather(sum, root=0)
    if rank == 0:
        assert flatten(list, result) == vector_sum(x, y)
        print(f"{x} + {y} =", flatten(list, result))

    return result


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

Follow the code in **main().** There are really only two pieces that should trip you up a bit. The first is the code that instantiates two random vectors, then partitions each of the vectors into chunks that evenly distribute across processes (including root!). The partitions take the form of a list of lists. The program then creates a list of tuples. The first element of the tuple is the x chunk, and the second the y chunk. So we have *a* tuple for each process, and that process can unpack that tuple and add the first and second parts of the tuple. We scatter these tuples across the processes, including once again process 0 ,the root , and then we unpack and add those elements of the tuple. We sum them using our **vector_sum()** function and then we gather the results back up into a list. This will be a list of lists, not a single list, so we then check the result and flatten the list of lists into a list.

### 3.3.2 Broadcast

Now broadcast is a much simpler operation. It sends *a* piece of data to *all* the processes within the communicator. This is depicted in figure 3.3.2.1.

*Figure 3.3.2.1*

There is in fact an opportunity to use broadcast in our riemann sum program. After we get a bunch of inputs from the user stored in process 0 we have to transmit certain inputs to *all* processes. We have to transmit the function string to be compiled on the process, and we have to transmit the spacing between points in the interval (of course we could recompute that spacing on the process as well). We can in fact replace:

```
# 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)
```

With a simple :

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

Notice however, we are only sending those inputs that are the *same* for all processes. The local interval is in fact specific to each process and we need scatter-gather for that. We will get back to that.

### 3.3.3 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?

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. Look at the figure below:

*Figure 3.3.2.1*

However, it would be extremely difficult to design and implement the tree data structure and even more difficult to write an optimal algorithm which could perform the global sum. The efficiency of the algorithm would depend on the operating-system and hardware of the computer. We can leave all that nasty detail to the people who designed and implemented MPI. MPI provides precisely a **reduce()** function whose signature is :

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

Reduce is a method of the **Comm** class. The argument of the parameter **sendobj** is any object on which an operation will be applied between the data on each separate process e.g. summing local data into a global sum. The **op** argument is the operation that will be applied. Now MPI defines an **Op** class which is what one would expect, a class which defines a function to be applied across processes. A simple example of reduce is given below:

```
"""
Shows the use of reduce. Each process produces
a random integer and then all are summed and sent to
process 0 (root).

Date: 05/16/2024
Author: Djamil Lakhdar-Hamina
```

```
"""

from random import randint

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()
```

Which produces a random integer for each process, then reduces them by summation and gives an example (non-deterministic) output:

```
rank 0: 5
rank 3: 9
rank 2: 1
rank 1: 3
the global sum is: 18
```

We can actually define our own instances of the **Op** class , we can define our own user-defined operations! An example is given below:

```
"""
Shows the use of Op class to define user reduction operation
A single random integer is produced across processes, the same one,
then the magnitude is found.

Date: 05/16/2024
Author: Djamil Lakhdar-Hamina

"""

from random import randint, seed

from mpi4py import MPI


def magnitude(x, y, float):
   return (x**2+y**2)**(1/2)


Magnitude = MPI.Op.Create(magnitude, True)


def main():

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

   LOWER_LIMIT = 1
   UPPER_LIMIT = 10

   seed(42)

   random_int = randint(LOWER_LIMIT, UPPER_LIMIT)
   print(f"rank {rank}: {random_int}")
   global_magnitude = comm.reduce(random_int, op=Magnitude, root=0)

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

```
        assert global_magnitude == 2 * random_int

    Magnitude.Free()



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

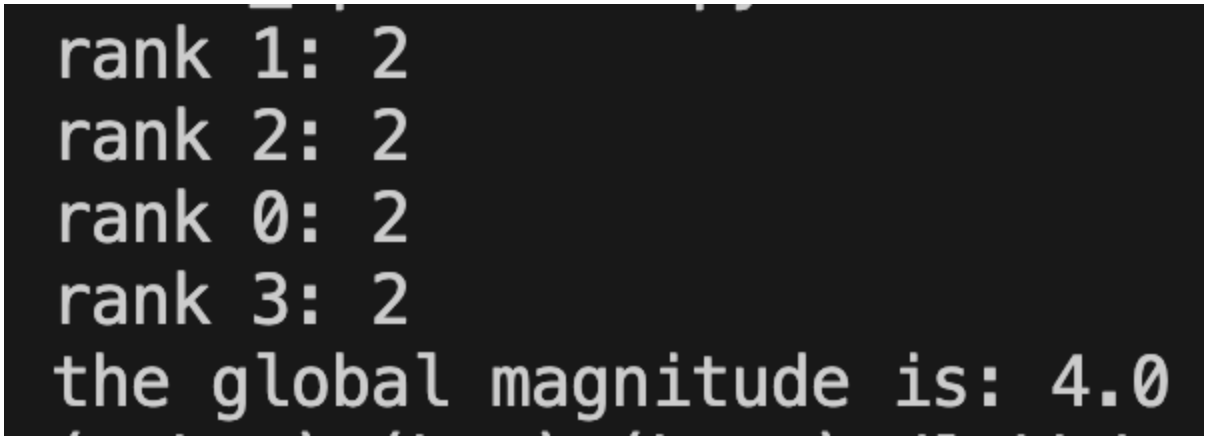
The program simply produces a single number across each process e.g. if there are four processes then each process has a copy of say integer 2. Now the magnitude M of a set of numbers a,b,c…n is:

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

So for our example we have:

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

Precisely the output of this program on *my machine* is:




The heart of the program is:

```
def magnitude(x, y, float):
    return (x**2+y**2)**(1/2)
```

```
Magnitude = MPI.Op.Create(magnitude, True)
```

**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
```

This method creates a new **Op** object. That object itself needs to have two arguments: **function** and **commute**. The **function** argument is of type Callable (basically the type of function). That **function** has to itself follow a certain signature. That function must take two data buffers and the type of those buffers and returns a buffer of the same type. If you do not specify the type the operation will not proceed. Now a **Buffer** type is MPI talk and we will come back to it, but in general computer science it means a predefined store of data, of information, that is used temporarily before that data is moved from one place to another. Here in our case it is just two floats, but it could have been really any other python object. **Commute** is a boolean (true or false) and for a function to commute simply means that if we define an operation like O(x+y) = O(y+x) e.g. addition commutes x+y = y + x.

### 3.3.4 Back to Riemann Sums

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 replacing:

```
    # 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)

    # Perform the local summation then send back
    if rank != 0:
        local_sum = riemann_sum_left(f, local_interval, delta_x)
        comm.send(local_sum, dest=0)
    else:
        global_sum = local_sum
        for process in range(1, size):
            local_sum = comm.recv(source=process)
            global_sum += local_sum
        print(f"sum: {global_sum}")
```

With a very succinct:

```
    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)
```

Wow! Collective functions are indeed very powerful and hide a lot of the complexity needed to implement optimal global operations.

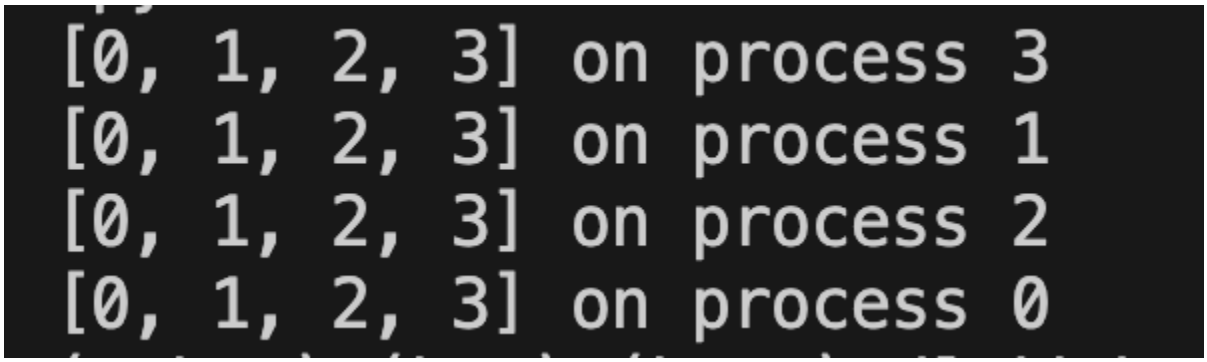
### 3.3.5 Allgather

There is a final collective function I would like to introduce **allgather()** which takes a piece of data on a process , gathers all the pieces of data and puts it in a list , and then distributes that list to all processes. Here is an example:

```
"""
Show use of allgather, have each process record its
rank in a variable then gather all these variables in list.

Date: 05/16/2024
```

```
Author: Djamil Lakhdar-Hamina

"""
from mpi4py import MPI


def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()

    ranks = comm.allgather(rank)

    print(f"{ranks} on process {rank}")


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

Which has output for program run with 4 processes :




### 3.3.6 Advanced Application Allgather: Matrix-Matrix Multiplication

To exhibit the use of allgather let us say we want to multiply two matrices and then have the result be on *all* the processes (say for process specific analysis of each matrix). First, let us explain how matrix-matrix multiplication works.

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

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

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:

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

First we must know the dimensions. There are three rows and three columns so the dimensions are 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 notably if matrix A has dimensions mxr and B sxn then A*B = C some new matrix with dimensions 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]. Now this is how the multiplication works:

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] (first number is row number, second number is column number). Now take row vector 1 and multiply by column vector 2, add all the elements up. This gives the entry [1,2]. Now take row vector 1 and multiply by column vector 3,add all the elements up. This gives the entry [1,3]. Now take row vector 2 and multiply by column vector 1, add the elements up. This gives entry [2,1]. See the pattern ? For every row vector i and column vector j, we multiply the ith row

by the jth column , add the elements up, and produce the number in position [i,j] in the new matrix. Now how can we parallelize this matrix multiplication?

There are many strategies, we depict them in the following figures:

*Figure 3.6.1 :*

Let us show the program which implements both serial and parallel multiplication. We will deploy a row partition:

```
"""
Perform matrix-matrix multiplication then have matrix distributed
to each process.

Date: 05/21/2024
Author: Djamil Lakhdar-Hamina

"""
from mpi4py import MPI


class Matrix:
   """
   A class representing a matrix including various attributes
   and methods.

   Attributes:
       dimensions (tuple(int,int)): dimension of matrix

   Methods:
       setitem: set the item using a tuple
       getitem: get the item using a tuple or slice
       str: produce a string rep for print
       T: produce transpose of matrix
       equal: dimensional and elementwise
```

```
    add: add two matrices
    multiply: multiply two matrices

"""

def __init__(self, A=[[]]) -> None:
    self.mat = A
    self.dimensions = (len(self.mat), len(self.mat[0]))

def __setitem__(self, index, value):
    if isinstance(index, tuple) and len(index) == 2:
        row, col = index
        self.mat[row][col] = value
    else:
        raise IndexError("Invalid index format")

def __getitem__(self, index: tuple) -> "Matrix":
    if isinstance(index, tuple):
        if all(isinstance(k, int) for k in index):
            row, col = index
            return self.mat[row][col]
        elif isinstance(index[0], int) and isinstance(index[1], slice):
            # Row access with column slicing
            row = index[0]
            cols = self._process_slice(index[1], len(self.mat[0]))
            sliced_row = [self.mat[row][j] for j in cols]
            return Matrix([sliced_row])
        elif isinstance(index[0], slice) and isinstance(index[1], int):
            # Column access with row slicing
            rows = self._process_slice(index[0], len(self.mat))
            col = index[1]
            sliced_matriB = [[self.mat[i][col]] for i in rows]
            return Matrix(sliced_matriB)
        elif all(isinstance(k, slice) for k in index):
            rows = self._process_slice(index[0], len(self.mat))
            cols = self._process_slice(index[1], len(self.mat[0]))
            sliced_matriB = [
                [self.mat[i][j] for j in range(len(self.mat[0])) if j in cols]
                for i in rows
            ]
            return Matrix(sliced_matriB)
    else:
```

```
            raise TypeError("Invalid key type")

    def _process_slice(self, slic, maB_size) -> range:
        start = slic.start if slic.start is not None else 0
        stop = slic.stop if slic.stop is not None else maB_size
        step = slic.step if slic.step is not None else 1
        return range(start, stop, step)

    def __str__(self):
        strings = [" ".join(list(map(str, row))) for row in self.mat]
        values = "\n".join(strings)
        m, n = self.dimensions
        header = f"Matrix: {m}x{n}"
        return "\n".join([header, values])

    def T(self) -> "Matrix":
        A_T = []
        m, n = self.dimensions
        for j in range(n):
            v = []
            for i in range(m):
                v.append(self[i, j])
            A_T.append(v)
        return Matrix(A_T)

    def __eq__(self, B: "Matrix") -> "Matrix":
        assert B.dimensions == self.dimensions
        m, n = self.dimensions
        for i in range(m):
            for j in range(n):
                if self[i, j] != B[i, j]:
                    return False
        return True

    def __add__(self, B: "Matrix") -> "Matrix":
        C = []
        assert self.dimensions == B.dimensions
        m, n = self.dimensions
        for i in range(m):
            v = []
            for j in range(n):
                v.append(self[i, j] + B[i, j])
```

```
            C.append(v)
        return Matrix(C)

    def __mul__(self, B: "Matrix") -> "Matrix":
        m, n = self.dimensions
        r, s = B.dimensions
        assert n == r
        C = []
        for i in range(m):
            v = []
            for j in range(s):
                val = 0
                for k in range(r):
                    val += self[i, k] * B[k, j]
                v.append(val)
            C.append(v)
        return Matrix(C)


def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    A = Matrix([[1, 2, 3], [3, 4, 5]])
    B = Matrix([[1, 2], [3, 4], [4, 5]])
    m, _ = A.dimensions
    _, n = B.dimensions

    if rank == 0:
        try:
            m, r = A.dimensions
            s, n = B.dimensions
            if m != size and n != size:
                raise ValueError("# rows does equal # process")
            if r != s:
                raise ValueError("dimensions of matrices invalid")
        except MPI.Exception as e:
            print(f"{e}: the number of rows does not equal number of processors")

    n, row_partition = comm.bcast((n, A[rank, :]), root=0)
```

```
    local_row = []
    for i in range(n):
        local_row.append((A[rank, :] * B[:, i])[0, 0])

    y = comm.allgather(local_row)

    assert A * B == Matrix(y)

    return y


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

Ignore the code for the **Matrix** class. Just treat this as a blackbox which defines a new object **Matrix**. This **Matrix** object has an attribute **dimension** which returns the integers rowxcolumn in tuple form (row,column). Now there are a number of methods and those that matter here will be **__init__** , **__setitem__**, **__getitem__** ,**__eq__** , and **__mul__.** The **__init__** method defines the *constructor* of the class, how we instantiate an instance of the class. The line:

```
    A = Matrix([[1, 2, 3], [3, 4, 5]])
    B = Matrix([[1, 2], [3, 4], [4, 5]])
```

Shows its use. We simply input a list of lists. Each list is *a row*. The **__setitem__** method allows us to set the value at a position [i,j] of the matrix via an index for instance:

```
>> A= Matrix([[1,2],[1,2]])
>> A[0,0]=0
>> print(A[0,0])
0
```

Now **__getitem__** allows us to get an item from the matrix using either a tuple index like [i,j] *or* a slice such as [1:,j] for instance:

```
>>A= Matrix([[1,2],[1,2]])
```

```
>> print(A[1:,:])
1x2 Matrix
2
2
```

The [1:,:] index reads “all elements in the row including and after 1 across all columns” i.e. the second column of elements. Now **__eq__** and **__mul__** simply define when two matrices are equal A=B and how to multiply two matrices A*B. So how does the parallel code work ? The heart of the program is:

```
  n, row_partition = comm.bcast((n, A[rank, :]), root=0)

  local_row = []
  for i in range(n):
      local_row.append((A[rank, :] * B[:, i])[0, 0])

  y = comm.allgather(local_row)
```

Before this section , first we do some error checking. The dimensions of the two matrices must be valid. The number of rows must equal the number of processes. Now each row is sent to its specific process along with the column number to iterate over via a broadcast. The row in the process is then multiplied by each column and a list is created for [rank,j] e.g. the 0th row with all columns gives the 0th row of the new matrix. The allgather then gathers the results in a list so we get a list of lists. Each process also has access to the new result y.

### 3.3.7 Excercises

## 3.4 Numpy Objects

### 3.4.2 NumPy and mpi4py:

In mpi4py there is a distinction between methods relating to general python objects , which are written lowercase, such as **send** and those relating to memory or data buffers such as ndarrays , which are written in uppercase, such as **Send**. 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 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.

*Figure 3.4.2.1: A sending memory buffer is used when sending information, and then this memory buffer transfers its contents to a receiving buffer*

Below we translate some of the simple programs written in section 3.3 into numpy versions. Where the example is very simple we give no explanation.

### 3.4.3 Send and Receive

```
"""
Exhibit the use of memory-buffer from np in mpi4py.

Date: 05/22/2024
Author: Djamil Lakhdar-Hamina

"""

import numpy as np
from mpi4py import MPI


def main():
   ROOT = 0
   m = 1
   n = 3
   comm = MPI.COMM_WORLD
   rank = comm.Get_rank()
   size = comm.Get_size()
   # Send , Receive
   if rank == 0:
```

```
        for p in range(1, size):
            sendbuf = np.random.rand(m, n)
            comm.Send([sendbuf, MPI.FLOAT], dest=p)
            print(f"sending {sendbuf} to {p} from root")

    else:
        rcvbuf = np.zeros((m, n), dtype=float)
        comm.Recv([rcvbuf, MPI.FLOAT], source=ROOT)
        print(f"received {rcvbuf} at {rank}")


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

Notice that the **Recv** method requires a pre-initialized memory buffer here called **rcvbuf** which is the same size and type as **sendbuf**. What happens is that the **rcvbuf** is *modified in place*. Furthermore, we utilized the 2-length buffer description [buffer, datatype].

### 3.4.4 Scatter-Gather

```
"""
Exhibit use of scatter-gather in numpy.

Date: 05/22/2024
Author: Djamil Lakhdar-Hamina

"""

import numpy as np
from mpi4py import MPI


def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    # Define the data to be scattered from the root process
    if rank == 0:
```

```
        send_data = np.array(
            [[i + j for j in range(size)] for i in range(size)], dtype=int
        )
        print(send_data.shape)
    else:
        send_data = None

    # Allocate memory for receive buffer in each process
    recv_data = np.empty(size, dtype=int)

    # Scatter data from root process to all other processes
    comm.Scatter(send_data, recv_data, root=0)

    print(f"Process {rank} received scattered data: {recv_data}")

    # Perform some computation (e.g., multiply the received data by scalar)
    local_sum = recv_data * 12

    # Gather results back to the root process
    if rank == 0:
        global_sums = np.empty((3, size), dtype=int)
    else:
        global_sums = None

    comm.Gather(local_sum, global_sums, root=0)

    if rank == 0:
        print("Global sums gathered by root process:")
        print(global_sums)


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

We did not even have to specify the data type of the memory buffers here. This can be done, but for explicitness this is not recommended, specifying types can make debugging easier.

### 3.4.4 Riemann Sum: Broadcast-Reduce

```
"""
```

```
Implement left-handed riemann sum using NumPy.

Date: 05/22/2024
Author: Djamil Lakhdar-Hamina

"""

from typing import Callable, Type

import numpy as np
from mpi4py import MPI


def is_single_line(expression: str) -> bool:
   """
   Takes a string expression and checks if the string is one-line by counting
   /n (newline) characters.

   Parameters:
   - expression: the function body in string form

   Returns:
   A boolean indicating if function body is single-lined (true)

   """
   # Count the number of newline characters
   newline_count = expression.count("\n")
   # Return True if there is only one newline character, False otherwise
   return newline_count == 0


def create_function_string_from_user_input() -> str:
   """
   Dynamically create a single-lined function string from user input.

   Parameters:

   Returns:
   A single-lined function of form f(x): return expression
   """
   # Get a string from the user
   print("Enter a single-lined function body: ")
```

```
    function_string = input()
    # Check that string is well-formed
    assert is_single_line(function_string)
    # Execute code while making sure that scope of f is not just in exec bloc
    return function_string


def create_function_on_process(function_string: str) -> Callable:
    """
    Compiles a function from a function string on a process.

    Parameters:
    - function_string: string or body of function

    Returns:
    The function with body given

    """
    locals_dict = {}
    exec(f"def f(x): return {function_string}", {}, locals_dict)
    return locals_dict["f"]


def riemann_sum_left(
    f: Callable[[float, int], float], interval: tuple, n: int, dtype: Type = float
) -> np.ndarray:
    """
    Takes function f, an interval (a,b) defined as a tuple, and a partition number n
    and approximates the integral of f on [a,b]

    Parameters:
    - f: function
    - interval: interval (a,b) where b>a
    - n : partition number, how many rectangles to approximate integral

    Returns:
    integral, the result of integration process, "area under the curve".

    Example:
    """
    a = interval[0]
    b = interval[1]
```

```
    delta_x = (b - a) / n
    x = np.arange(a, b, delta_x)
    integral = sum(f(x) * delta_x)
    return np.array(integral, dtype=dtype)


def main():
    root = 0
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    global_sum = np.empty(1, dtype=float)
    if rank == 0:
        # Define function on root process
        function_string = create_function_string_from_user_input()
        f = create_function_on_process(function_string)
        # Check that it was "compiled"
        assert callable(f)

        # User input for integral
        print(
            """Enter start of interval,\
end of interval, and partition number: """
        )
        a, b, n = input().split()

        # Type convert into appropriate format
        info = np.array([a, b, n], dtype=float)

        # Check that user inputs fulfill conditions
        try:
            if info[2] % size != 0:
                raise ValueError(
                    "Number of partitions \
isn't evenly split between processes"
                )
            if info[0] >= info[1]:
                raise ValueError(
                    "The lower limit of the interval\
greater or equal to upper"
                )
        except ValueError as e:
```

```
            print(e)
    else:
        info = np.empty(3, dtype=float)
        function_string = None

    comm.Bcast([info, MPI.FLOAT], root=0)
    function_string = comm.bcast(function_string, root=0)
    a, b, n = info
    if rank != 0:
        f = create_function_on_process(function_string)

    # use rank number to create a interval in tuple form
    partition_size = (b - a) / size
    partition_number = n // size
    local_interval = (
        a + rank * partition_size,
        a + (rank + 1) * partition_size,
    )
    local_sum = riemann_sum_left(f, local_interval, partition_number)
    # we use a double to allow for greater precision otherwise overflow can happen
    comm.Reduce(
        [local_sum, MPI.DOUBLE], [global_sum, MPI.DOUBLE], op=MPI.SUM, root=root
    )

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

    return global_sum

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

Much of this function is the same. However, there are two principal differences between the previous pure python and the current numpy version. First off, we do not create an interval in the form of a nested list or a nested array and then scatter parts of the list to the various processes. Instead, the tuple which represents the sub-interval is defined for each process using the value of **rank**.

Secondly, we utilize the uppercase **Bcast** and **Reduce** methods which require a different treatment of memory in general.

### 3.4.5 Vector Addition: Broadcast-Scatter-Gather

We present here the NumPy version of the vector-vector addition program from before. The strategy though is a bit different. Two vectors are generated of length n. Here vector 1 and vector 2 are split into p partitions of length n/p to be distributed to p processes. For each process p, the vector 1 and vector 2 are *concatenated* , stapled together, so that for instance vector1=[1,2,3] and vector2=[1,2,3] are concatenated into vector=[1,2,3,1,2,3]. The concatenated vector is then of length 2n/p. Then each concatenated partition is scattered to a process. The process then splits the concatenated vector in half and adds the first half to the second half of the vector. The result is gathered back into the array of arrays and then flattened.

```
"""
Use numpy array and mpi4py to add two vectors together.

Date: 05/22/2024
Author: Djamil Lakhdar-Hamin
"""

from itertools import islice

import numpy as np
from mpi4py import MPI



def part_gen(iterable: iter, n: int) -> list[list]:
   # batched('ABCDEFG', 3) → ABC DEF G
   if n < 1:
       raise ValueError("n must be at least one")
   it = iter(iterable)
   while batch := list(islice(it, n)):
       yield batch
```

```
def partition(iterable: iter, n: int) -> list[list]:
   "partition an iterable into chunks of n size"
   return np.array(list(part_gen(iterable, n)))


def main():
   ROOT = 0

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

   if rank == 0:
       print("input vector length:")
       n = int(input())
       chunk = n // (size)
       info = np.array([n, chunk], dtype=int)
       try:
           if n % size != 0:
               raise ValueError(
                   "The values are not evenly partitioned between n workers\
processors"
               )
       except ValueError as e:
           print(e)
           comm.Abort()
       # define 2 random vectors
       x, y = np.random.rand(n), np.random.rand(n)
       partitions = np.array(partition(x, chunk), dtype=float), np.array(partition(y,
chunk), dtype=float)
       # split each vector into p chunks of n size
       sendbuf = np.array(
           [np.concatenate((part_x, part_y)) for (part_x, part_y) in zip(*partitions)]
       )
       print(f"sendbuf is : \n {sendbuf}\n")
   else:
       info = np.empty(2, dtype=int)
        sendbuf = None

   comm.Bcast([info, MPI.INT], root=0)
   n, chunk = info
   recvbuf = np.empty(chunk * 2, dtype=float)
```

```
    comm.Scatter([sendbuf, MPI.FLOAT], [recvbuf, MPI.FLOAT], root=ROOT)
    midpoint = len(recvbuf) // 2
    local_sum = recvbuf[:midpoint] + recvbuf[midpoint:]

    finalrecvbuf = np.empty((size, chunk))

    comm.Gather([local_sum, MPI.FLOAT], [finalrecvbuf, MPI.FLOAT], root=ROOT)
    if rank == 0:
        print(f" {x} + {y} = {final recvbuf.flatten()}")
        assert np.all(x + y == final recvbuf.flatten())


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

Let us follow this program rather closely as it makes use of **Bcast, Scatter** and **Gather**. We ask the user for input on how long the vector is. Then we create the **sendbuf** , which is the data buffer which is an array of arrays, each array is a partitioned and concatenated vector. We also create an array called **info** which when broadcasted allows us to create receiving buffers of certain length for the other processes.

Now rank 0 does the partitioning and concatenation. If the rank is not 0, the **sendbuf** memory buffer must be initialized to **None** for all processes except root if the program is to work. The same goes for **info**. *Notice that the empty receive buffers are of the same size and type*. Now we use **Bcast** which modifies in place the previously empty initialized vector **info**. We use the contents of **info** to create receiving buffers on each process of the right size, the size being n//size (the number of elements evenly distributed between the processes).

Here :

```
    comm.Scatter([sendbuf, MPI.FLOAT], [recvbuf, MPI.FLOAT], root=ROOT)
    midpoint = len(recvbuf) // 2
    local_sum = recvbuf[:midpoint] + recvbuf[midpoint:]
```

We scatter the **sendbuf**, collecting the portion for each process into an associated **recvbuf**. We then add the left half of the vector (which is part of the vector 1) and add it to the right half (which is part of vector2). Finally:

```
finalrecvbuf = np.empty((size, chunk))


comm.Gather([local_sum, MPI.FLOAT], [finalrecvbuf, MPI.FLOAT], root=ROOT)
```

We initialize an empty **finalrecvbuf** which has the *same shape* as **sendbuf** and we then gather the contents of our calculations (a vector which is of size n//size) on each process into root. Here is the interactive output run with 5 inputs across 5 processes:

```
input vector length:
5
number elements per process: 1
vector x: [0.38322905 0.98861993 0.8473237  0.07336761 0.86653903]
vector y: [0.85950019 0.42502985 0.2853473  0.52586603 0.92193138]
split it up and concatenated, sendbuf is :
 [[0.38322905 0.85950019]
 [0.98861993 0.42502985]
 [0.8473237  0.2853473 ]
 [0.07336761 0.52586603]
 [0.86653903 0.92193138]]

rank 0: empty receiving buffer per process:[0.38322905 0.85950019]
rank 4: empty receiving buffer per process:[ 9.94589822e+092 -1.51890927e-240]
rank 2: empty receiving buffer per process:[-6.11530373e+304  5.21934357e-126]
rank 1: empty receiving buffer per process:[ 1.29896799e+079 -5.80057416e+230]
rank 3: empty receiving buffer per process:[-3.96405417e+206  2.31368346e-256]
rank 3: each array sent to a process, split up and added:[0.59923364]
rank 0: each array sent to a process, split up and added:[1.24272924]
rank 4: each array sent to a process, split up and added:[1.78847041]
rank 1: each array sent to a process, split up and added:[1.41364978]
rank 2: each array sent to a process, split up and added:[1.132671]
```

```
rank 0: empty final receiving buffer: [[0.85950019]
 [0.42502985]
 [0.2853473 ]
 [0.52586603]
 [0.92193138]]
rank 3: empty final receiving buffer: [[ 1.  ]
 [ 2.75]
 [ 6.  ]
 [10.75]
 [17.  ]]
rank 4: empty final receiving buffer: [[ 1.  ]
 [ 2.75]
 [ 6.  ]
 [10.75]
 [17.  ]]
rank 2: empty final receiving buffer: [[ 1.  ]
 [ 2.75]
 [ 6.  ]
 [10.75]
 [17.  ]]
rank 1: empty final receiving buffer: [[ 1.  ]
 [ 2.75]
 [ 6.  ]
 [10.75]
 [17.  ]]
```

```
final receiving buffer after gather: [[1.24272924]
 [1.41364978]
 [1.132671  ]
 [0.59923364]
 [1.78847041]]

 [0.38322905 0.98861993 0.8473237  0.07336761 0.86653903] + [0.85950019 0.42502985 0.2853473
0.52586603 0.92193138] = [1.24272924 1.41364978 1.132671   0.59923364 1.78847041]
```

### 3.4.6. Advanced Application: Numpy Riemann Sum

```
"""
Implement left-handed riemann sum using NumPy.

Date: 05/22/2024
Author: Djamil Lakhdar-Hamina

"""

from typing import Callable, Type

import numpy as np
from mpi4py import MPI



def is_single_line(expression: str) -> bool:
   """
   Takes a string expression and checks if the string is one-line by counting
```

```
    /n (newline) characters.

    Parameters:
    - expression: the function body in string form

    Returns:
    A boolean indicating if function body is single-lined (true)

    """
    # Count the number of newline characters
    newline_count = expression.count("\n")
    # Return True if there is only one newline character, False otherwise
    return newline_count == 0


def create_function_string_from_user_input() -> str:
    """
    Dynamically create a single-lined function string from user input.

    Parameters:

    Returns:
    A single-lined function of form f(x): return expression
    """
    # Get a string from the user
    print("Enter a single-lined function body: ")
    function_string = input()
    # Check that string is well-formed
    assert is_single_line(function_string)
    # Execute code while making sure that scope of f is not just in exec bloc
    return function_string


def create_function_on_process(function_string: str) -> Callable:
    """
    Compiles a function from a function string on a process.

    Parameters:
    - function_string: string or body of function

    Returns:
    The function with body given
```

```
    """
    locals_dict = {}
    exec(f"def f(x): return {function_string}", {}, locals_dict)
    return locals_dict["f"]


def riemann_sum_left(
    f: Callable[[float, int], float], interval: tuple, n: int, dtype: Type = float
) -> np.ndarray:
    """
    Takes function f, an interval (a,b) defined as a tuple, and a partition number n
    and approximates the integral of f on [a,b]

    Parameters:
    - f: function
    - interval: interval (a,b) where b>a
    - n : partition number, how many rectangles to approximate integral

    Returns:
    integral, the result of integration process, "area under the curve".

    Example:
    """
    a = interval[0]
    b = interval[1]
    delta_x = (b - a) / n
    x = np.arange(a, b, delta_x)
    integral = sum(f(x) * delta_x)
    return np.array(integral, dtype=dtype)


def main():
    root = 0
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    global_sum = np.empty(1, dtype=float)
    if rank == 0:
        # Define function on root process
        function_string = create_function_string_from_user_input()
        f = create_function_on_process(function_string)
```

```
        # Check that it was "compiled"
        assert callable(f)

        # User input for integral
        print(
            """Enter start of interval,\
end of interval, and partition number: """
        )
        a, b, n = input().split()

        # Type convert into appropriate format
        info = np.array([a, b, n], dtype=float)

        # Check that user inputs fulfill conditions
        try:
            if info[2] % size != 0:
                raise ValueError(
                    "Number of partitions \
isn't evenly split between processes"
                )
            if info[0] >= info[1]:
                raise ValueError(
                    "The lower limit of the interval\
greater or equal to upper"
                )
        except ValueError as e:
            print(e)
    else:
        info = np.empty(3, dtype=float)
        function_string = None

    comm.Bcast([info, MPI.FLOAT], root=0)
    function_string = comm.bcast(function_string, root=0)
    a, b, n = info
    if rank != 0:
        f = create_function_on_process(function_string)

    # use rank number to create a interval in tuple form
    partition_size = (b - a) / size
    partition_number = n // size
    local_interval = (
        a + rank * partition_size,
```

```
        a + (rank + 1) * partition_size,
    )
    local_sum = riemann_sum_left(f, local_interval, partition_number)
    # we use a double to allow for greater precision otherwise overflow can happen
    comm.Reduce(
        [local_sum, MPI.DOUBLE], [global_sum, MPI.DOUBLE], op=MPI.SUM, root=root
    )

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

    return global_sum


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

This is left as an exercise to study and understand.

### 3.4.7 Exercises

1. Study the riemann sum script above. Run it, and comment on it. Make sure you understand what is happening line by line. Try changing things up to break and change behavior of the script.
2.

## 3.5 Blocking/Non-blocking and One-sided Communication

We are now getting into more advanced topics. Good job! We will explore the distinction between blocking and nonblocking communication here.

Send and receive are forms of **blocking** communication. What that means is that the sender and receiver must explicitly coordinate, the sender must talk to the receiver, the receiver must get the message, and then the sender and receiver can proceed to another task. We have already worked with blocking communications. There are two-types i.e. point-to-point and collective communication and we worked with both.

As opposed to blocking communication we have, surprise, **non-blocking** communication. Non-blocking communication means processes communicate by directly accessing memory on remote processes *without having to explicitly communicate with one another*. Let us think through this via analogy. Imagine there are two persons. One person wants to send a message out to the world, let us say they are lost in the woods, it doesn't really matter who gets it, they just need help. That person goes to a tree, and carves out their message. Now a fellow hiker finds that tree , and based on the message starts a search for that person.

### 3.5.1 Get-put Interface

MPI-2 added a certain interface for non-blocking communication and it called **get-put** or **Random Memory Access (RMA)** model. The reason one would utilize this interface is because it can be more efficient for certain network configurations , taking advantage of the features of that network. It also applies more readily to a shared-memory set-up by lowering latency and software overhead.

The MPI interface revolves around an object called **windows**, which essentially specify regions of remote memory of processes which can be accessed by processes via read and write operations. These remote blocks of memory can be accessed via put (remote send) , get (remote receive) and accumulate (remote reduce). In python, this means the main actor is a class called **Win,** and get, put , get, and accumulate are in fact methods of that class. Now MPI, and mpi4py support different synchronization styles, but these are fairly complex and it is hoped by working through the basic put-get model that the basis is established for the reader to delve into such styles on their own.

Now how would one utilize this get-put interface and make use of the **Win** class. The following program shows how:

```
"""
Show use of get put model. Program instantiates a window of
shared memory which only exists in process 0. Process 0
fills a buffer, or puts data, and then the other processes
```

```
get that data from shared memory.

Date: 05/22/2024
Author: Djamil Lakhdar-Hamina

"""

import numpy as np
from mpi4py import MPI
from mpi4py.util import dtlib


def main():
   comm = MPI.COMM_WORLD
   rank = comm.Get_rank()

   dtype = MPI.FLOAT
   np_dtype = dtlib.to_numpy_dtype(dtype)
   itemsize = dtype.Get_size()

   N = 42
   win_size = N * itemsize if rank == 0 else 0
   win = MPI.Win.Allocate(win_size, comm=comm)
   buf = np.empty(N, dtype=np_dtype)

   if rank == 0:
       buf.fill(42)
       win.Lock(rank=0, lock_type=MPI.LOCK_EXCLUSIVE, assertion=MPI.MODE_NOCHECK)
       win.Put(buf, target_rank=0)
       win.Unlock(rank=0)
       comm.Barrier()
   else:
       comm.Barrier()
       win.Lock(rank=0, lock_type=MPI.LOCK_SHARED, assertion=MPI.MODE_NOCHECK)
       win.Get(buf, target_rank=0)
       win.Unlock(rank=0)
       assert np.all(buf == 42)


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

Let us closely explain what is happening. The program begins as usual , and then converts an MPI datatype **MPI.FLOAT** to a numpy datatype via a sub-module method **dtlib.to_numpy_type.** The size of an item of the datatype is then queried. All of this via:

```
    dtype = MPI.FLOAT
    np_dtype = dtlib.to_numpy_dtype(dtype)
    itemsize = dtype.Get_size()
```

Next the size of the memory buffer is dictated and then the window size calculated. The size of the window will be the itemsize*number of items in process 0 and it will be zero for the other processes. Finally, the window , or shared memory, is allocated and the memory buffer within that shared memory is instantiated all via :

```
    N = 42
    win_size = N * itemsize if rank == 0 else 0
    win = MPI.Win.Allocate(win_size, comm=comm)
    buf = np.empty(N, dtype=np_dtype)
```

Notice the class **MPI.Win** for which **Allocate** is a method with the signature:

```
(method) def Allocate(
    size: int,
    disp_unit: int = 1,
    info: Info = INFO_NULL,
    comm: Intracomm = COMM_SELF
) -> Win
```

This is a method of the **Win** class. **Size** is the size of the window in *bytes*. The **disp_unit** means the number of bytes per item. The argument **info** takes an **MPI.Info** object and modifies it. Finally , **comm** is a communicator, here the **MPI.COMM_WORLD** object. Here, the only arguments we use are **size** and **comm**. We will return to the others soon.

Now comes the fun part.

```
    if rank == 0:
```

```
        buf.fill(42)
        win.Lock(rank=0, lock_type=MPI.LOCK_EXCLUSIVE, assertion=MPI.MODE_NOCHECK)
        win.Put(buf, target_rank=0)
        win.Unlock(rank=0)
        comm.Barrier()
    else:
        comm.Barrier()
        win.Lock(rank=0, lock_type=MPI.LOCK_SHARED, assertion=MPI.MODE_NOCHECK)
        win.Get(buf, target_rank=0)
        win.Unlock(rank=0)
        assert np.all(buf == 42)
```

If the rank of the process is 0, we fill our memory buffer with the value 42. Then we put a *lock* on the shared memory window via **Lock**. The **Lock** method has the following signature:

```
(method) def Lock(
    rank: int,
    lock_type: int = LOCK_EXCLUSIVE,
    assertion: int = 0
) -> None
```

The **rank** specifies the part of the shared memory that we put a lock on, the part of the shared memory allocated only to that process. The **lock_type** tells us in what way this part of the shared memory can be accessed. Can different processes access this part of the window? If so this is specified by **MPI.LOCK_EXCLUSIVE.** Otherwise, **MPI.LOCK_SHARED** specifies that different processes can access that part of the shared memory concurrently.

#### *3.5.1.1 Assertions*

The **assertion** argument requires a bit of hashing out. In MPI and mpi4py, an "assertion" is a hint that you can provide to the MPI implementation to optimize the performance of certain operations. These assertions are optional flags that convey specific information

about the context in which the operation is being performed, potentially allowing the MPI library to make optimizations based on the guarantees you provide.

Assertions are typically constants specified by MPI.MODE_*. They include:

- MPI.MODE_NOCHECK: Indicates that the process knows the lock will be granted immediately. This can save some overhead since the MPI implementation can skip some of the internal checks.
- MPI.MODE_NOPRECEDE: Indicates that the lock will not be preceded by another lock on the same window in the same process.
- MPI.MODE_NOPUT: Indicates that the process will not perform any Put or Accumulate operations during the lock epoch.
- MPI.MODE_NOSTORE: Indicates that the process will not update any local window copies in the current epoch.
- MPI.MODE_NOSUCCEED: Indicates that the lock will not be followed by another lock on the same window in the same process.

These flags can be combined using the bitwise OR operator if multiple assertions apply.

Now that we have put a lock on the part of shared memory that is reserved for process 0, specified that only a process can access that part of the window at a time, and that the process to access the window knows it is immediately locked we can actually *put* data in that window. That is precisely what the **Put** command does which has the signature:

```
(method) def Put(

   origin: BufSpec,

   target_rank: int,

   target: TargetSpec | None = None

) -> None
```

Evidently, **origin** is the data that will be transferred to the window. **Target_rank** specifies the part of the window reserved for that rank where the data will be stored. Finally, on top of rank number we can also use **target** which has the form (rank, count , datatype). We then synchronize the processes via **comm.Barrier()**.

Next we access the worker processes. We synchronize the processes, then lock the portion of shared memory dedicated to process 0. We specify this time **LOCK_SHARED**. This allows the various worker processes to access the 0-part of the window concurrently, they can share access. We then utilize the **Get** method to retrieve the data. The method has the same signature as **Put.** The script ends by checking that the memory has indeed been transferred from process 0, to the window, from the window to the various worker nodes.

### 3.5.2 Get-put with Target

As mentioned before there is a modification that can be made to the **Get-Put** scripts above in that we can deploy the **target** arguments in **Get** and **Put**. An example:

```
"""
Show the get put model of MPI through basic example.

Date: 05/22/2024
Author: Djamil Lakhdar-Hamina

"""

import numpy as np
from mpi4py import MPI
from mpi4py.util import dtlib


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

```
    info = MPI.Info()

    datatype = MPI.FLOAT
    np_dtype = dtlib.to_numpy_dtype(datatype)
    itemsize = datatype.Get_size()

    N = size + 1
    win_size = N * itemsize if rank == 0 else 0
    win = MPI.Win.Allocate(
        size=win_size,
        disp_unit=itemsize,
        info=info,
        comm=comm,
    )

    if rank == 0:
        mem = np.frombuffer(win, dtype=np_dtype)
        mem[:] = np.arange(len(mem), dtype=np_dtype)
        print(f"buffer instantiated in process 0 : {mem}")
    comm.Barrier()

    buf = np.zeros(2, dtype=np_dtype)
    target = (rank, 2, datatype)
    win.Lock(rank=0)
    win.Get(buf, target_rank=0, target=target)
    win.Unlock(rank=0)
    print(f"{rank}: {buf}")
    assert np.all(buf == [rank, rank + 1])


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

This gives output for 4 processes:

```
buffer instantiated in process 0 : [0. 1. 2. 3. 4.]
0: [0. 1.]
3: [3. 4.]
2: [2. 3.]
1: [1. 2.]
```

So what happened? The first crucial part:

```
if rank == 0:
    mem = np.frombuffer(win, dtype=np_dtype)
    mem[:] = np.arange(len(mem), dtype=np_dtype)
    print(f"buffer instantiated in process 0 : {mem}")
comm.Barrier()
```

Process 0 instantiates a buffer from the window. The function **np.frombuffer()** is an interesting and low-level function. It takes any kind of buffer and converts it into a 1-d ndarray. So for instance:

```
>>> buffer_string = b"bioinformatics"
>>> np.frombuffer(buffer=buffer_string, dtype="|S1", count=len(buffer_string))
array([b'b', b'i', b'o', b'i', b'n', b'f', b'o', b'r', b'm', b'a', b't',
       b'i', b'c', b's'], dtype='|S1')
```

Where b"" is a *byte-string* and "|S1" is the name given in numpy dtypes to a byte string of length 1 where | signifies that byte-order does not matter. Within the get-put program the fact that we can use the frombuffer method on win shows that **Win** class is ,under the hood , simply a fancy buffer, nothing more and nothing less. We then replace the elements of that 1-d buffer with an **arange()** which in its most basic form takes a length l and a step size n and produces an array of length l in increments of n. So here if mem has length 5, and the step size is by default 1, the array is [0,1,2,3,4]. Next we have :

```
buf = np.zeros(2, dtype=np_dtype)
target = (rank, 2, datatype)
win.Lock(rank=0)
win.Get(buf, target_rank=0, target=target)
```

```
win.Unlock(rank=0)
print(f"{rank}: {buf}")
assert np.all(buf == [rank, rank + 1])
```

Notice process 0 never *puts* any data in the window explicitly. Instead the window is converted into a target buffer and data is transferred from the target buffer to the remote process buffer here called **buf**. The target buffer is accessed via a **Get** command utilizing the **target** argument. The **target** argument specifies which process is *getting* the data from the target buffer, and then tells the process *how much and what kind* of data to access from the target buffer. Here process r is accessing 2 elements of datatype **MPI.FLOAT**, that is 2 floats are read into the local buffer.

### 3.5.3 The I-methods : Isend-Irecv

The old-school form of non-blocking communication can be summarized in what I call the I-methods. To understand this we have to understand what send and receive are doing as forms of blocking communication.

When I send a message from a process, that process blocks, it cannot perform any more computation until the message is sent and received. For example:

```
if rank == 0 :
    comm.send(...)
    # can't access this until message receive
else:
    comm.recv(...)
    # can't access this until message received
```

But what if I want to send a message but also continue to perform further computation on the process i.e. the computations I need to do do not interfere with the results of the communication. That is precisely what isend and irecv are for and for every blocking method e.g. send, receive, reduce etc. there is an I-equivalent.

Check out the following code:

```
"""
This program demonstrates basic MPI (Message Passing Interface) asynchronous
communication usingthe mpi4py library. The example involves two processes
(ranks 0 and 1) exchanging messages.

Usage:
   Run the script with an MPI execution command such as
  `mpiexec -n 2 python script_name.py`.

Date: 05/23/2024
Author: Djamil Lakhdar-Hamina

"""

"""
This program demonstrates basic MPI (Message Passing Interface) asynchronous
communication usingthe mpi4py library. The example involves two processes
(ranks 0 and 1) exchanging messages.

Usage:
   Run the script with an MPI execution command such as
  `mpiexec -n 2 python script_name.py`.

Date: 05/23/2024
Author: Djamil Lakhdar-Hamina

"""

from mpi4py import MPI


def main():
   comm = MPI.COMM_WORLD
   rank = comm.Get_rank()
   size = comm.Get_size()
   statuses = [MPI.Status() for _ in range(size//2+1)]
   assert size == 2
   # non numpy-example
   if rank == 0:
       data = bytearray(source="Hello, world!".encode())
```

```
        request = comm.Isend(data, dest=1)
        print(f"rank {rank}: line 28 {request.Test(statuses[0])}")
        print(f"rank {rank} :line 29 {2 + 3} ")
    elif rank == 1:
        buf = bytearray(46)
        request = comm.Irecv(buf, source=0)
        print(f"rank {rank}: line 33 {request.Test(statuses[0])}")
        print(f"rank {rank} : line 34 {6 + 3} ")
        request.Wait(status=statuses[0])
        print(f"rank {rank}: line 36 {request.Test(statuses[0])}")
        print(f"{buf.decode()} From process {rank}")


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

Let us go through this program.

```
        data = bytearray("Hello, world!".encode())
        request = comm.Isend(data, dest=1)
        print(f"rank {rank}: line 28 {request.Test(statuses[0])}")
        print(f"rank {rank} :line 29 {2 + 3} ")
```

So first, the root process generates a **bytearray** for “Hello, world!”. A **bytearray** sounds like what it is , it is a *native* python object which follows for the **source** argument :

| Type | Description |
|---|---|
| String | Converts the string to bytes using `str.encode()` Must also provide encoding and optionally errors |
| Integer | Creates an array of provided size, all initialized to null |
| Object | A read-only buffer of the object will be used to initialize the byte array |

| | |
|---|---|
| Iterable | Creates an array of size equal to the iterable count and initialized to the iterable elements Must be iterable of integers between `0 <= x < 256` |
| No source (arguments) | Creates an array of size 0. |

We can transform the string “Hello, world!” into a series of bytes via the method **encode**. The reason we do this is to create a *data buffer* of the string data. We then pass this to **Isend** which has a simple signature:

```
(method) def Isend(
   buf: BufSpec,
   dest: int,
   tag: int = 0
) -> Request
```

Notice we did not need to specify the item type within the data buffer but simply passed data to argument **buf**. This is because the byte type is in a sense the lowest-level data type and so wears its type on its sleeve. Now the **dest** is obviously the destination and the **tag** what we learned before. But what is this return type **Request**?

The **Request** type is in fact the heart of the non-blocking and asynchronous model of computation for imethods. It is a class object within the python and mpi4py universe and has many methods and attributes. It is produced by **Isend** and **IRecv** methods. So:

```
      request = comm.Isend(data, dest=1)
```

Created an instance of the request class. Obviously the first argument is the object sent, and the second the destination. Now, because this is asynchronous and non-blocking, process 0 does not hang until the message sent is received by process 1. The computations after the **isend** continue unabated.

As for process 1, the process 1 must create a buffer to store the data, here we choose a comfortable 20 bytes. If a buffer of insufficient size is passed you will get an error message, or else the contents of the message will not be fully stored in process 1. Next, process 1 fills the buffer and produces a request object of its own:

```
    request = comm.Irecv(buf, source=0)
```

Now the request can be queried as to whether or not the message was received, we actually *explicitly* block the communication via this **request.Wait** command:

```
request.Wait(status=statuses[0])
```

Here we introduce the **Status**() object , it does exactly what one would expect. It shows us the **Status** of the communication and includes self-explanatory attributes as **count, error, cancelled**. This **Status** object is instantiated with `MPI.Status()` . The output of this program, and it is not deterministic, is:

```
rank 1: line 33 False
rank 1 : line 34 9
rank 1: line 36 True
Hello, world! From process 1
rank 0: line 28 True
rank 0 :line 29 5
```

Notice that when the message is not received and then the request "closed" that the **Test** method of request returns **False**. The **Test** method *tests if a request for message reception is complete*. Study the output above closely.

### 3.5.4 Bcast,Scatter,Gather, Waitall and Advanced Application 1: Vector Addition

The next program is a doozy. It will show though how we can implement broadcast, scatter, and gather using our **Isend** and **Irecv**.

```
"""
This program demonstrates builds an asynchronous bcast, scatter, and gather
and demonstrates their use via vector-vector addition.

Usage:
   Run the script with an MPI execution command such as
  `mpiexec -n 2 python script_name.py`.

Date: 05/23/2024
Author: Djamil Lakhdar-Hamina

"""

from itertools import islice
from typing import Generator, Iterable, List, Sequence, Type

import numpy as np
from mpi4py import MPI

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


def part_gen(
   iterable: Iterable, iterable_type: Type, n: int or float
) -> "Generator[Iterable]":
   """
   Takes an iterable and breaks it up into iterable_type iterators of n size .

   Parameters:
   - iterable: the iterator to break up
   - iterable_type : the type of iterator to be broken up into
   - n : the number of elements in the iterator

   Returns:
```

```
    A list of iterators with number of elements n

    """
    # batched('ABCDEFG', 3) → ABC DEF G
    if n < 1:
        raise ValueError("n must be at least one")
    it = iter(iterable)
    while batch := iterable_type(islice(it, n)):
        yield batch


def partition(
    n: int,
    inner_iterable: Iterable,
    outer_iterable_type: Iterable = List,
    inner_iterable_type: Iterable = List,
) -> Iterable[List]:
    """
    Partition an iterable into chunks of a specified size.

    This function divides an input iterable into smaller chunks, each of size `n`.
    The resulting chunks are stored in an iterable specified by `outer_iterable`,
    and each chunk itself is of the type specified by `inner_iterable_type`.

    Parameters:
    n (int): The size of each chunk.
    inner_iterable (iter): The iterable to be partitioned.
    outer_iterable (iter): The type of the outer iterable that will hold the chunks.
    Default is `list`. inner_iterable_type (iter): The type of the inner iterables that
    represent the chunks. Default is `list`.

    Returns:
    iter[list]: An iterable containing the partitioned chunks.

    Example:
    >>> list(partition(3, range(10)))
    [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]]

    >>> tuple(partition(2, 'abcdef', outer_iterable=tuple, inner_iterable_type=str))
    (('a', 'b'), ('c', 'd'), ('e', 'f'))
    """
    return outer_iterable_type(part_gen(inner_iterable, inner_iterable_type, n))
```

```
def async_bcast(
   buf: object, buf_size: int, dtype: Type, comm: "MPI.Comm", size: int, root: int = 0
) -> object:
   """
   Asynchronously broadcasts data from the root process to all other processes in the
   given MPI communicator.

   Parameters:
   buf (object): The buffer containing the data to be broadcast. This is only used by
   the root process.
   buf_size (int): The size of the buffer (number of elements) to be broadcast.
   dtype (Type): The data type of the elements in the buffer.
   comm (MPI.Comm): The MPI communicator over which the data is to be broadcast.
   size (int): The total number of processes in the communicator.
   root (int, optional): The rank of the root process from which data will be
   broadcast. Default is 0.

   Returns:
   object: The buffer containing the received data on non-root processes. On the root
   process, this is the original buffer.

   Example:
   >>> from mpi4py import MPI
   >>> import numpy as np
   >>> comm = MPI.COMM_WORLD
   >>> rank = comm.Get_rank()
   >>> size = comm.Get_size()
   >>> if rank == 0:
   >>>    buf = np.array([1, 2, 3, 4], dtype=np.int)
   >>> else:
   >>>    buf = None
   >>> recvbuf = async_bcast(buf, 4, np.int, comm, size, root=0)
   >>> print(f"Process {rank} received: {recvbuf}")

   Notes:
   - This function uses non-blocking send (Isend) and receive (Irecv) operations to
   achieve asynchronous communication.
   - The root process sends the buffer to all other processes.
   - Non-root processes allocate an empty buffer of the specified size and type to
     receive the data.
```

```
    - The function returns the received buffer on non-root processes and the original
    buffer on the root process.
    """

    if rank == root:
        recvbuf = buf
        requests = []
        for p in range(0, size):
            if p != root:
                request = comm.Isend(buf=recvbuf, dest=p, tag=0)
                requests.append(request)
        MPI.Request.Waitall(requests)
    else:
        recvbuf = np.empty(buf_size, dtype=dtype)
        request = comm.Irecv(buf=recvbuf, source=0, tag=0)
        request.Wait()
    return recvbuf


def async_scatter(
    sendbuf: Sequence[object],
    recv_size: int,
    dtype: Type,
    comm: "MPI.Comm" = comm,
    size: int = size,
    root: int = 0,
) -> object:
    """
    Asynchronously scatters data from the root process to all other processes in the
    given MPI communicator.

    Parameters:
    sendbuf (Sequence[object]): A sequence of buffers, where each buffer contains the
    data to be sent to a specific process. This is only used by the root process.
    recv_size (int): The size of the buffer (number of elements) to be received by each
    process.
    dtype (Type): The data type of the elements in the buffer.
    comm (MPI.Comm): The MPI communicator over which the data is to be scattered.
    size (int): The total number of processes in the communicator.
    root (int, optional): The rank of the root process from which data will be
    scattered. Default is 0.
```

```
    Returns:
    object: The buffer containing the received data on each process.

    Example:
    >>> from mpi4py import MPI
    >>> import numpy as np
    >>> comm = MPI.COMM_WORLD
    >>> rank = comm.Get_rank()
    >>> size = comm.Get_size()
    >>> if rank == 0:
    >>>     sendbuf = [np.array([i, i+1, i+2], dtype=np.int) for i in range(size)]
    >>> else:
    >>>     sendbuf = None
    >>> recvbuf = async_scatter(sendbuf, 3, np.int, comm, size, root=0)
    >>> print(f"Process {rank} received: {recvbuf}")

    Notes:
    - This function uses non-blocking send (Isend) and receive (Irecv) operations to
    achieve asynchronous communication.
    - The root process sends a specific portion of the send buffer to each process.
    - Non-root processes allocate an empty buffer of the specified size and type to
    receive their portion of the data.
    - The function returns the received buffer on all processes.
    """
    if rank == root:
        recvbuf = sendbuf[0]
        requests = []
        for p in range(0, size):
            if p != root:
                request = comm.Isend(sendbuf[p], p, p)
                requests.append(request)
        MPI.Request.Waitall(requests)
    else:
        recvbuf = np.empty(recv_size, dtype=dtype)
        request = comm.Irecv(recvbuf, source=root, tag=rank)
        request.Wait()
    return recvbuf


def async_gather(
    sendbuf: Sequence[object],
    recv_size: int,
```

```
    dtype: Type,
    comm: "MPI.Comm" = comm,
    size: int = size,
    root: int = 0,
) -> List[object]:
    """
    Asynchronously gathers data from all processes to the root process in the given MPI
    communicator.

    Parameters:
    sendbuf (Sequence[object]): The buffer containing the data to be sent from each
    process.
    recv_size (int): The size of the buffer (number of elements) to be received
    from each process.
    dtype (Type): The data type of the elements in the buffer.
    comm (MPI.Comm): The MPI communicator over which the data is to be gathered.
    size (int): The total number of processes in the communicator.
    root (int, optional): The rank of the root process to which data will be gathered.
    Default is 0.

    Returns:
    List[object]: A list of buffers containing the received data on the root process.
    On non-root processes, returns None.

    Example:
    >>> from mpi4py import MPI
    >>> import numpy as np
    >>> comm = MPI.COMM_WORLD
    >>> rank = comm.Get_rank()
    >>> size = comm.Get_size()
    >>> sendbuf = np.array([rank, rank + 1, rank + 2], dtype=np.int)
    >>> gathered_data = async_gather(sendbuf, 3, np.int, comm, size, root=0)
    >>> if rank == 0:
    >>>     for i, data in enumerate(gathered_data):
    >>>         print(f"Process {i} sent: {data}")

    Notes:
    - This function uses non-blocking send (Isend) and receive (Irecv) operations to
    achieve asynchronous communication.
    - The root process receives data from all processes and gathers it into a list.
    - Non-root processes send their data to the root process.
    - The function returns a list of received buffers on the root process, and returns
```

```
    None on non-root processes.
    """

    if rank == root:
        requests = []
        gather_arr = []
        gather_arr.append(sendbuf)
        for p in range(0, size):
            if p != root:
                recv_buf = np.empty(recv_size, dtype=dtype)
                request = comm.Irecv(recv_buf, p, p)
                requests.append(request)
                gather_arr.append(recv_buf)
        MPI.Request.Waitall(requests)
        return gather_arr
    else:
        request = comm.Isend(sendbuf, dest=root, tag=rank)
        request.Wait()


def main():
    assert size > 2

    if rank == 0:
        print("input vector length:")
        n = int(input())
        chunk = n // (size)
        chunk_buffer = np.array([chunk], dtype="i")
        # print(f"number elements per process: {chunk}")
        try:
            if n % size != 0:
                raise ValueError(
                    "The values are not evenly partitioned between n workers\
processors"
                )
        except ValueError as e:
            print(e)
            comm.Abort()
        # define 2 random vectors
        x, y = np.random.rand(n), np.random.rand(n)
        partitions = partition(chunk, x, list, list), partition(chunk, y, list, list)
        # split each vector into p chunks of n size
```

```
        sendbuf = np.array(
            [np.concatenate((part_x, part_y)) for (part_x, part_y) in zip(*partitions)]
        )
        # print(f"split it up and concatenated, sendbuf is : \n {sendbuf}\n")

    else:
        n = chunk_buffer = sendbuf = None

    # broadcast chunk length
    buf_size = 1  # Size of the buffer
    chunk = async_bcast(
        buf=chunk_buffer, buf_size=buf_size, dtype="i", comm=comm, size=size
    )[0]
    recvbuf = async_scatter(
        sendbuf=sendbuf, recv_size=chunk * 2, dtype=float, comm=comm, size=size
    )
    local_sum = recvbuf[:chunk] + recvbuf[chunk:]
    finalrecvbuf = np.array(
        async_gather(sendbuf=local_sum, recv_size=chunk, dtype=float, comm=comm)
    )
    if rank == 0:
        assert np.all(x + y == final recvbuf.flatten())


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

### 3.5.5 Accumulate and Advanced Application 2: Back to Riemann Sums

```
"""
This program computes the Riemann sum of a user-defined function over a specified
interval using MPI for parallel processing. The main components of the program include
functions for partitioning iterables, generating parts of iterables,
validating single-line expressions,creating functions dynamically from user input, and
performing the Riemann sum calculation. The MPI framework is utilized to distribute
the
computation across multiple processes.

Functions:
```

```
----------
- part_gen(iterable: Iterable,
          iterable_type: Type,
          n: int or float) -> "Generator[Iterable]":
   Takes an iterable and breaks it up into `iterable_type` iterators of size `n`.

- partition(n: int,
inner_iterable: Iterable, outer_iterable_type: Iterable = List,
inner_iterable_type: Iterable = List) -> Iterable[List]:
   Partitions an iterable into chunks of a specified size.

- is_single_line(expression: str) -> bool:
   Checks if a string expression is a single line.

- create_function_string_from_user_input() -> str:
   Dynamically creates a single-lined function string from user input.

- create_function_on_process(function_string: str) -> Callable:
   Compiles a function from a function string on a process.

- riemann_sum_left(f: Callable[[(float | int)], float], interval: Tuple[(float |
int)],
delta_x: (float | int)) -> float | int:
   Approximates the integral of `f` over an interval using the left Riemann sum
method.

MPI Process:
-------------
- main():
   Manages the MPI processes, including collecting user input for the function and
   interval, distributing the computation, and aggregating the results.

Example Usage:
--------------
1. The program prompts the user to enter a single-line function body and the interval
(start, end) and number of partitions.
2. The main function handles the MPI initialization, input distribution,
and the parallel computation of the Riemann sum.
3. The computed integral is printed by the root process.

Note:
-----
```

```
- Ensure MPI is set up in your environment to run this program.
- The user input for the function should be in the form of a valid single-line Python
expression.

Date: 05/28/2024
Author : Djamil Lakhdar-Hamina

"""

from itertools import islice
from typing import Callable, Generator, Iterable, List, Tuple, Type

import numpy as np
from mpi4py import MPI
from mpi4py.util import dtlib


def part_gen(
    iterable: Iterable, iterable_type: Type, n: int or float
) -> "Generator[Iterable]":
    """
    Takes an iterable and breaks it up into iterable_type iterators of n size .

    Parameters:
    - iterable: the iterator to break up
    - iterable_type : the type of iterator to be broken up into
    - n : the number of elements in the iterator

    Returns:
    A list of iterators with number of elements n

    """
    # batched('ABCDEFG', 3) → ABC DEF G
    if n < 1:
        raise ValueError("n must be at least one")
    it = iter(iterable)
    while batch := iterable_type(islice(it, n)):
        yield batch


def partition(
    n: int,
```

```
    inner_iterable: Iterable,
    outer_iterable_type: Iterable = List,
    inner_iterable_type: Iterable = List,
) -> Iterable[List]:
    """
    Partition an iterable into chunks of a specified size.

    This function divides an input iterable into smaller chunks, each of size `n`.
    The resulting chunks are stored in an iterable specified by `outer_iterable`,
    and each chunk itself is of the type specified by `inner_iterable_type`.

    Parameters:
    n (int): The size of each chunk.
    inner_iterable (iter): The iterable to be partitioned.
    outer_iterable (iter): The type of the outer iterable that will hold the chunks.
    Default is `list`. inner_iterable_type (iter): The type of the inner iterables that
    represent the chunks. Default is `list`.

    Returns:
    iter[list]: An iterable containing the partitioned chunks.

    Example:
    >>> list(partition(3, range(10)))
    [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]]

    >>> tuple(partition(2, 'abcdef', outer_iterable=tuple, inner_iterable_type=str))
    (('a', 'b'), ('c', 'd'), ('e', 'f'))
    """
    return outer_iterable_type(part_gen(inner_iterable, inner_iterable_type, n))


def is_single_line(expression: str) -> bool:
    """
    Takes a string expression and checks if the string is one-line by counting
    /n (newline) characters.

    Parameters:
    - expression: the function body in string form

    Returns:
    A boolean indicating if function body is single-lined (true)
```

```
    """
    # Count the number of newline characters
    newline_count = expression.count("\n")
    # Return True if there is only one newline character, False otherwise
    return newline_count == 0


def create_function_string_from_user_input() -> str:
    """
    Dynamically create a single-lined function string from user input.

    Parameters:

    Returns:
    A single-lined function of form f(x): return expression
    """
    # Get a string from the user
    print("Enter a single-lined function body: ")
    function_string = input()
    # Check that string is well-formed
    assert is_single_line(function_string)
    # Execute code while making sure that scope of f is not just in exec bloc
    return function_string


def create_function_on_process(function_string: str) -> Callable:
    """
    Compiles a function from a function string on a process.

    Parameters:
    - function_string: string or body of function

    Returns:
    The function with body given

    """
    locals_dict = {}
    exec(f"def f(x): return {function_string}", {}, locals_dict)
    return locals_dict["f"]


def riemann_sum_left(
```

```
    f: Callable[[(float | int)], float],
    interval: Tuple[(float | int)],
    delta_x: (float | int),
) -> float | int:
    """
    Takes function f, an interval (a,b) defined as a tuple, and a partition number n
    and approximates the integral of f on [a,b]

    Parameters:
    - f: function
    - interval: interval (a,b) where b>a
    - n : partition number, how many rectangles to approximate integral

    Returns:
    integral, the result of integration process, "area under the curve".

    Example:
    """
    fvec = np.vectorize(f)
    return np.sum(
        fvec(np.arange(start=interval[0], stop=interval[1], step=delta_x)) * delta_x
    )


def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    dtype = MPI.FLOAT
    np_dtype = dtlib.to_numpy_dtype(dtype)
    itemsize = dtype.Get_size()

    if rank == 0:
        # Define function on root process
        function_string = create_function_string_from_user_input()
        f = create_function_on_process(function_string)
        # Check that it was "compiled"
        assert callable(f)

        # User input for integral
        print(
            """Enter start of interval,\
```

```
end of interval, and partition number: """
        )
        a, b, n = input().split()

        # Type convert into appropriate format
        a, b = tuple(map(float, [a, b]))
        n = int(n)

        # Check that user inputs fulfill conditions
        try:
            if n % size != 0:
                raise ValueError(
                    "Number of partitions \
isn't evenly split between processes"
                )
            if a >= b:
                raise ValueError(
                    "The lower limit of the interval\
greater or equal to upper"
                )
        except ValueError as e:
            print(e)

    else:
        function_string = a = b = n = None

        # Evenly partition the interval into subintervals
        # of size partition space, one per process

    comm.Barrier()

    function_string, a, b, n = comm.bcast((function_string, a, b, n), root=0)
    if rank != 0:
        f = create_function_on_process(function_string)

    partition_size = (b - a) / size
    partition_delta = n // size
    n = partition_size / partition_delta
    local_interval = (
        a + rank * partition_size,
        a + (rank + 1) * partition_size,
    )
```

```
    local_sum = np.array([riemann_sum_left(f, local_interval, n)], dtype=np_dtype)
    global_sum = np.empty(1, dtype=np_dtype)
    win = MPI.Win.Allocate(itemsize, comm=comm)
    win.Lock(rank=0, lock_type=MPI.LOCK_SHARED, assertion=MPI.MODE_NOCHECK)
    target_rank = 0
    win.Accumulate([local_sum, dtype], target_rank=target_rank, op=MPI.SUM)
    win.Unlock(rank=0)
    comm.Barrier()
    win.Lock(rank=0, lock_type=MPI.LOCK_SHARED, assertion=MPI.MODE_NOCHECK)
    win.Get(global_sum, target_rank=0)
    win.Unlock(rank=0)

    if rank == 0:
        print(f"The integral of {function_string} = {global_sum[0]}")

    win.Free()


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

### 3.5.6 Advanced Application 3: Matrix Transpose

```
"""
This program partitions a square matrix into smaller square submatrices, transposes
each submatrix in parallel using MPI, and reassembles the transposed blocks into the
final matrix. The main steps include dividing the matrix, scattering the submatrices
to MPI processes, transposing locally, and gathering the transposed blocks.

Functions:
----------
- block_partition(matrix: np.array, n: int) -> np.array:
    Divides a square matrix into nxn submatrices.

- unblock(block_matrix: np.array, n: int) -> np.array:
    Reconstructs the original matrix from nxn submatrices.

- transpose(matrix: np.array) -> np.array:
```

```
   Transposes a given matrix.

MPI Process:
-------------
- main():
   Manages the MPI processes, including matrix input, partitioning, scattering,
transposing, and gathering results.

Example Usage:
--------------
1. The user is prompted to enter the dimensions of the square matrix and the block
size.
2. The matrix is partitioned, and submatrices are scattered to MPI processes.
3. Each process transposes its submatrix and the results are gathered and reassembled
by the root process.

Note:
-----
- Ensure MPI is set up in your environment to run this program.

Example:



"""

import numpy as np
from mpi4py import MPI
from mpi4py.util import dtlib


def block_partition(matrix: np.array, n: int) -> np.array:
   """
   Takes a square matrix of dimensions mxm and breaks into
   square submatrices of dimensions nxn.

   Parameters:
   - matrix: 2d np.array matrix to be divided
   - n: dimension of the matrix

   Returns:
   np.array with dimensions (mxm/(nxn),n,n)
```

```
    Example:

    """
    m = matrix.shape[0]
    blocks = []
    for i in range(0, m, n):
        for j in range(0, m, n):
            block = matrix[i : i + n, j : j + n]
            blocks.append(block)
    return np.array(blocks)

#TODO : write this function
def unblock(block_matrix: np.array, n: int) -> np.array:
    """
    Takes a block matrix of sub dimensions mxm and builds
    square matrix of dimension nxn.

    Parameters:
    - matrix: 3d np.array matrix to be collapsed
    - n: dimension of the matrix

    Returns:
    np.array with dimensions (nxn)

    Example:

    """
    b, m, _ = block_matrix.shape
    matrix = np.empty((n, n))
    for i in range(0, b):
        for j in range(0, m):
            for k in range(0, m):
                matrix[j, k] = block_matrix[i, j, k]
    return matrix


def transpose(matrix: np.array) -> np.array:
    """
    Takes a matrix of dimensions mxn and gives the
    tranpose nxm.
```

```
    Parameters:
    - matrix: 2d np.array matrix to be tranposed

    Returns:
    np.array with dimensions (nxm)

    Example:

    """
    m, n = matrix.shape
    new_matrix = np.empty((n, m))
    for row in range(m):
        new_matrix[:, row] = matrix[row, :]
    return new_matrix


def main():
    ROOT = 0
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    dtype = MPI.FLOAT
    np_dtype = dtlib.to_numpy_dtype(dtype)
    itemsize = dtype.Get_size()

    dim = blocks = local_block = None
    if rank == 0:
        print("dimensions of the square matrix:")
        dim = int(input())
        print("block dimension:")
        block_dimension = int(input())
        A = np.array([np.arange(dim) for _ in range(dim)])
        blocks = block_partition(A, block_dimension)
        final_buf = np.empty((blocks.shape), dtype=np_dtype)
        try:
            if dim % size != 0 and dim % 2 != 0 and size % 2 != 0:
                raise ValueError(
                    f"Dimensions {dim} cannot be block partitioned across {size}\
processes and dimensions and size are not even numbers"
                )
        except ValueError as e:
```

```
            print(e)
            exit(1)

    local_block = comm.scatter(blocks, root=ROOT)
    dim = comm.bcast(dim, root=ROOT)
    block_dimension = local_block.shape[1]
    win_size = dim**2 * itemsize
    win = MPI.Win.Allocate(win_size, comm=comm)
    local_buf = np.empty((block_dimension, block_dimension), dtype=np_dtype)

    win.Lock(rank=rank)
    local_buf = transpose(local_block)
    win.Put(local_buf, target_rank=rank)
    win.Unlock(rank=rank)
    comm.Barrier()

    if rank == 0:
        win.Lock(rank=ROOT)
        for i in range(0, size):
            win.Get(local_buf, target_rank=i)
            final_buf[i] = local_buf
        win.Unlock(rank=ROOT)

    if rank == 0:
        print(final_buf)
        # TODO: finish unblocking the result end

    win.Free()


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

### 3.5.7 Exercises

1. **** Implement asynchronous scatter-gather using only Irecv and Isend plus Wait and Waitall. Make sure it can communicate the

basic python objects (e.g. int, float ,str, list, dict, set, tuple).

## 3.6 Communicators and Topologies

Let us explore the concept of communicators and of topologies. Communicators are objects which contain processes and which coordinate communication between these processes. A topology is a specific way of ordering these processes, we want to organize processes in certain ways in order to organize our thinking and to optimize computation.

Now before we were working with MPI.COMM_WORLD, which simply represents a communicator containing all processes spawned at the command line. However, there are applications where we want to treat subsets of all processes as independent communicators, independent universes of communication.

In MPI, there are two types of communicators , **intra-communicators and inter-communicators**. Intra-communicators are in essence a collection of processes that can send messages to each other and engage in collective communication. Inter-communicators are in essence used to send messages between processes belonging to disjoint, independent intra-communicators. We already confronted the paradigmatic intra-communicator MPI.COMM_WORLD, which contains all processes spawned by the program.

### 3.6.1 Intra-communicators

Now a barebones intra-communicator is made of up two things:

1. A group
2. A context

A **group** is an ordered collection of processes, each assigned a unique **rank** , which is just a non-negative integer. A **context** is a system-defined object that uniquely picks out a communicator. Two

distinct communicators will have a different context even if they share the same group! Why contexts then? Communication occurs only if a process in one communicator sends a message to a process in the *same* communicator, and the way to ensure that the communicator is the *same* is to have a property, the context, which ensures we can identify the communicator.

The following shows operations with groups:

```
"""
This program demonstrates the creation and manipulation of MPI groups and communicators
using mpi4py. It divides the processes into two groups: the first half and the second half of the available ranks.

The key steps include:
1. Initializing the MPI environment and determining the rank and size.
2. Creating process ranks for the first half of the available ranks.
3. Creating two new groups: one including the first half and the other excluding the first half.
4. Duplicating the first group.
5. Demonstrating various group operations such as intersection, union, and comparison.
6. Creating a communicator for the first group and checking its existence for each process.

The program outputs the size of each group, the results of group operations, and whether the new communicator exists for each process.
"""
from mpi4py import MPI

# 0 1 2 ... n
# n n+1 ... m
# where n == m//2


def main():

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

```
    # first row ranks to add to new group
    process_ranks = [i for i in range(size//2)]

    group_world = comm.Get_group()

    # create new group
    first_row_group = group_world.Incl(process_ranks)
    print(f"first row size: {first_row_group.size}")

    # create new group from second row
    second_row_group = group_world.Excl(process_ranks)
    print(f"second row size: {second_row_group.size}")

    # duplicate first row group
    first_row_copy = first_row_group.Dup()

    # exhibit some methods
    print("\nrelations between groups:")
                        print(f"intersect:       {group_world.Intersect(first_row_group,
second_row_group).size}")
    print(f"union: {group_world.Union(first_row_copy, second_row_group).size}")
    print(f"compare: {group_world.Compare(first_row_copy, second_row_group)}\n")

    first_row_comm = comm.Create(first_row_group)

    if first_row_comm != MPI.COMM_NULL:
        print(f"first_row_comm does exist in process: {rank}")
    elif first_row_comm == MPI.COMM_NULL:
        print(f"first_row_comm does not exist in process: {rank}")


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

Which gives for output with 4 processes:

```
first row size: 2
second row size: 2

relations between groups:
intersect: 0
union: 4
compare: 3

first row size: 2
second row size: 2

relations between groups:
intersect: 0
union: 4
compare: 3

first row size: 2
second row size: 2

relations between groups:
intersect: 0
union: 4
compare: 3

first row size: 2
second row size: 2

relations between groups:
intersect: 0
union: 4
compare: 3

first row size: 2
second row size: 2

relations between groups:
intersect: 0
union: 4
compare: 3

first_row_comm does not exist in process: 2
```

First the program initializes our environment in the usual way. We then define a number of process ranks in a list, the first half of the processes by rank. Next we get the group object from the **COMM_WORLD** universe. We use this group object **Group** to create a new *sub-group* via one of the methods of **Group** i.e. **Incl**. This command simply creates a new sub-group by *including* a list of processes you want to make into a new sub-group. Then we use the opposite of **Incl** i.e. **Excl**. This takes a list of the processes you want to exclude from a new sub-group. We then exhibit another method of group, we copy the new sub-group of "first row" processes using **Dup.**

Next, we exhibit some "relational" methods of the **Group** object. Notably, we take the **COMM_WORLD** group and we perform **Intersect, Union, and Compare**. **Intersect** and **Union** simply take two arguments, a first and a second group and checks the intersection and union of both groups. The following figure shows what those operations mean in set-theoretic terms where each group is treated as a set:

*Figure 3.6.1.1: A union is a function which takes two sets of elements, two groups of processes and pumps out a new set of the elements in the first or in the second. A intersection is a function which takes two sets of elements and spits out a new set of the elements in the first and in the second.*

**Compare** is a bit different. It takes two groups but it returns an integer between 0-3 for which each integer is itself mapped to a meaningful MPI constant. The table below shows this:

| | | |
|---|---|---|
| **MPI.IDENT** | 0 | communicators are identical; they are the same object. |
| **MPI.CONGRUENT** | 1 | congruent, meaning they have the same group of processes with the same ranks but are different objects. |
| **MPI.SIMILAR** | 2 | The communicators have the same group of processes, but the ranks of processes within the groups may differ |

| | | |
|---|---|---|
| **MPI.UNEQUAL** | 3 | The communicators are completely different; they do not have the same group of processes. |

*Table 3.6.1.1 : Table showing the Compare possible results, the integer mapped to them, and the meaning of each integer and result*

Finally, we create a new communicator using a method of **COMM_WORLD** i.e. Create() , which takes a group and pumps out a new intra-communicator. We then check whether, for each process, the new communicator is valid or exists via the **MPI.COMM_NULL** and **MPI.COMM_SELF** objects.

```
if first_row_comm != MPI.COMM_NULL:
    print(f"first_row_comm does exist in process: {rank}")
elif first_row_comm != MPI.COMM_SELF:
    print(f"first_row_comm does not exist in process: {rank}")
```

This code snippet checks if **first_row_comm** is a valid communicator in the current process. If first_row_comm is valid (not MPI.COMM_NULL), it prints that the communicator exists for this process. If first_row_comm is not **MPI.COMM_SELF** (indicating it could be invalid), it prints that the communicator does not exist for this process. These objects are useful for debugging and verifying the state of communicators across different processes in an MPI program.

What if we had 1000 processes ? Are we going to create intra-communicators by writing lists [1,2,3,4,5… etc.] explicitly? That would be a waste of time. Fortunately, the global communicator has a built-in method, called Split() which takes two parameters, color and key. Color will determine which processes are in which

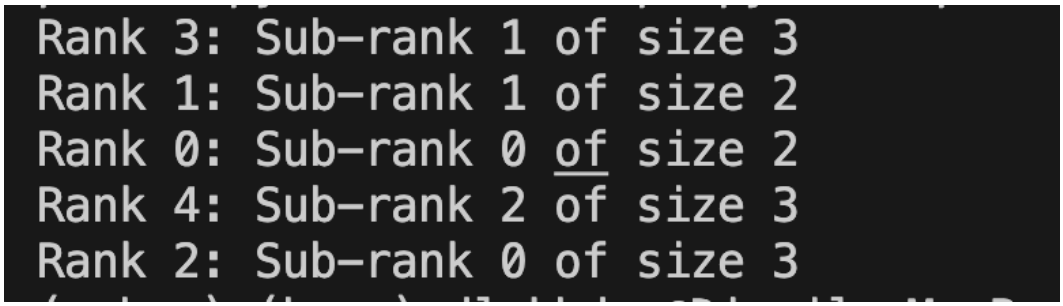
subgroup. Key will give a sort of identifier to each process *within* each subgroup.

For instance, say I have 4 processes. I want to put rank 0,1 in one subgroup and ranks 2,3 in another. I assign a color of 0 to the ranks 0,1 and a color 1 to the ranks 2,3. Then let us say I want to make sure 0,1 have subranks 0,1 respectively. I can pass a key that orders them, it can be any ordered integers. I could pass say 1,2 and the subranks will be 0,1. The following program generalizes this logic below:

```
"""
This program demonstrates the use of MPI to split a communicator into two sub-communicators
based on the rank of the processes. It initializes MPI, determines the rank and size of the
world communicator, and then divides the processes into two groups.

Processes with ranks in the first half of the communicator are assigned to one group, while
those in the second half are assigned to another. Each group performs communication within
its sub-communicator and prints out its sub-rank and the size of the sub-communicator.

The key steps include:
1. Initializing the MPI environment and obtaining the rank and size.
2. Splitting the communicator into two sub-communicators based on rank.
3. Performing communication within each sub-communicator.
4. Printing the sub-rank and sub-communicator size for each process.
5. Freeing the sub-communicator resources.

Date: 05/30/2024
Author: Djamil Lakhdar-Hamina
"""
from mpi4py import MPI


def main():

    # Initialize MPI
    comm = MPI.COMM_WORLD
```

```
    rank = comm.Get_rank()
    size = comm.Get_size()
    assert size > 2

    # Split the communicator into two sub-communicators
    if rank < size // 2:
        color = 0
        key = rank
    else:
        color = 1
        key = rank

    # put into one of two groups if data defined in one or the other is 0 or 1
    sub_comm = comm.Split(color, key)

    # Perform communication within the sub-communicator
    sub_rank = sub_comm.Get_rank()
    sub_size = sub_comm.Get_size()

    print(f"Rank {rank}: Sub-rank {sub_rank} of size {sub_size}")

    sub_comm.Free()


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

The program splits the processes into two halves, it takes the first half of the processes puts them in a sub-group and the same for the second half. It identifies each of these sub-groups via a color, and then it assigns a key to each process within the sub-group. We then check the rank and sub-rank of each process and the size of each sub-group. This gives output:




A note. We free the sub communicator, although python manages cleanup and memory it is cleaner to simply free the memory ourselves, especially when the programs get very complex this is best.

### 3.6.2 Inter-communicators

Inter-communicators are sub-groups of distinct processes which can only communicate with one another directly. Different inter-communicators then communicate with each other only via a parent or head process. The following is a very good pattern for building inter-communicators:

```
"""
This MPI program demonstrates the creation and use of intercommunicators
to facilitate communication between two subgroups of processes. The program
is designed for a minimum of 4 processes and splits the processes into two
subgroups. Each subgroup has a local leader that communicates with its
children within the subgroup.

The steps performed are:
1. Initialize MPI and obtain rank and size.
2. Split the processes into two subgroups based on rank.
3. Create an intercommunicator between the two subgroups.
4. Perform intra-group communication: local leader sends messages to its
  children within the subgroup.
5. Free the intercommunicators and sub-communicators.
6. Finalize MPI.

Run the program with at least 4 processes to observe the communication
patterns.

Functions:
- main: Main function to execute the MPI communication.
```

```
"""
from mpi4py import MPI


# Initialize MPI
def main() -> None:
   comm = MPI.COMM_WORLD
   rank = comm.Get_rank()
   size = comm.Get_size()

   local_leader = 0
   remote_leader = 0
   assert size >= 4

   if rank < size // 2:
       color = 0
       key = rank
   else:
       color = 1
       key = rank

   sub_comm = comm.Split(color, key)

   # Create an intercommunicator
   if color == 0 :
       intercomm = comm.Create_intercomm(local_leader, sub_comm, remote_leader, 0)
   else:
           intercomm  =  comm.Create_intercomm(local_leader,  sub_comm,  remote_leader,
size//2)

   local_rank = sub_comm.Get_rank()

   if color == 0:
       if local_rank == local_leader:
           for p in range(1, size//2):
               intercomm.send(None, dest=p)
               print(f"rank:local_rank : sending from {rank}:{local_rank} \
to {p}:{local_rank}...")
       else:
           intercomm.recv(source=local_leader)
           print(f"rank:local_rank : {rank}:{local_rank} \
received from {local_leader}:{local_leader}",
```

```
        end="\n")
    else:
        if local_rank == local_leader:
            for p in range(size//2+1, size):
                intercomm.send(None, dest=p)
                print(f"rank:local_rank : sending from {rank}:{local_rank}\
to {p}:{local_rank}...")
        else:
            intercomm.recv(source=size//2)
            print(f"rank:local_rank : {rank}:{local_rank} \
received from {size//2}:{local_leader}",
        end="\n")


    intercomm.Free()
    sub_comm.Free()
    MPI.Finalize()


if __name__ == "__main__":
    main()


# Perform communication within each subgroup if needed
# global_data = intercomm.gather(local_data, root=0)
# print(global_data)
```

First , we split the processes into two *intra-communicators*. The first half of the processes are put into one and the second half into another. Now we use the **comm** object to create an inter-communicator via :

```
    # Create an intercommunicator
    if color == 0:
        intercomm = comm.Create_intercomm(local_leader, sub_comm, remote_leader, 0)
    else:
        intercomm = comm.Create_intercomm(
            local_leader, sub_comm, remote_leader, size // 2
        )
```

With signature:

```
(method) def Create_intercomm(
    local_leader: int,
    peer_comm: Intracomm,
    remote_leader: int,
    tag: int = 0
) -> Intercomm
```

The **Create_intercomm** method takes three necessary arguments. The **local_leader** is the *local rank* that will be considered the root process using the *local rank* of the process. If rank 3 is the parent of the second inter-comm, then local_rank=0 is the value to be used. Next is **peer_comm** which takes the name of the *other* inter-comm here both being called **sub_comm** in their distinct processes. Finally, **remote_leader** is the *global rank* of the head node of the *other* inter-comm. This is extremely important. So in this block of code, we simply create a new inter-comm by checking if the process is part of a certain sub-group via **color** and then creating the intercom as per the signature above.

Then in the part where we send messages from the two distinct "head" processes we use *global ranks*. We make sure that inter-comms communicate internally by checking color.

### 3.6.3 Topologies

We mentioned briefly that a communicator can have additional structure imposed on it besides the group and the context. That additional information is said to be **cached** with the communicator, and cached information are called **attributes** (not to be confused with Python's attributes). One of the most important attributes is a topology, which imposes an order on processes within a group by associating different addressing schemes with the processes.

Essentially there are two different topologies that can be used in MPI i.e. a Cartesian or grid versus a graph topology. A Cartesian grid is

exactly what it sounds like, it is a line, or square, or cube with discrete cells making it up, just like in our mathematics courses when we deal with coordinate systems. A graph is a bit more abstract of a data structure, or less familiar. It consists of nodes and vertices, each node has certain attributes associated with it, and the vertices connect the nodes. We depict both below:

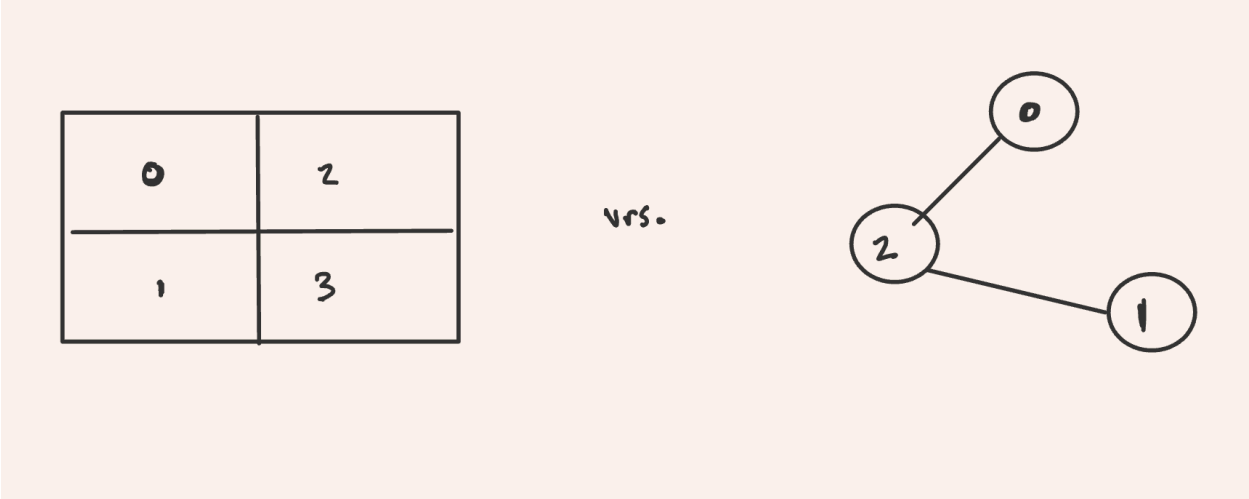



*Figure 3.6.1 : Grid versus Graph. On the left we have a 2-dimensional Cartesian grid, and within each cell is a process identified by its rank. Communication can occur between any of these processes and the processes are associated with a coordinate [n,m] where n is the row and m the column. On the right is a graph, the circles are the processes identified by rank, and the vertices connect the processes allowing communication between them. So 0 and 1 do not communicate. This is in fact a directed graph.*

### 3.6.1.1 Cartesian or Grid Topologies

As one might expect, the Cartesian or grid topology orders the processes into a grid, a coordinate-system.
In order to associate a square grid with **MPI_COMM_WORLD** or in fact with any intra-communicator we need to specify the following:

1. The *dimension* of the grid. If the grid has say 1 dimension it is a line, if it has 2 its a square , and if 3 its a cube. Anything more is called a hyper-cube.

2. The *size* of the dimension , so for instance for a line how many points in the line, and for a square how many rows and columns.
3. *Periodicity* of dimension. This tells us if for instance the last point in the line wraps around to the first point. Or if the last entry in the row or column of the square wraps around to the first.
4. *Reordering* of processes, the system can actually optimize the ordering of processes in the grid and we can tell the program that for the sake of optimization.

Let us see how this works for a 2x2 cartesian grid with periodicity both column-wise and row-wise.

```
"""
This program demonstrates the use of MPI (Message Passing Interface) to create
a 2D Cartesian grid topology with periodic boundaries using the mpi4py library.

The program checks if the number of processes (MPI ranks) is a perfect square,
ensuring it can form a square grid. Each process determines its coordinates
within this grid and identifies its neighboring processes, facilitating
communication between them.

The program involves the following key steps:
1. Initialize the MPI environment and obtain the rank and size of the communicator.
2. Ensure the number of processes is a perfect square, allowing for an n x n grid.
3. Create a Cartesian communicator with periodic boundaries.
4. Print the coordinates of each rank within the grid.
5. Perform shift communications to identify left-right and up-down neighbors.
6. Create and use a sub-communicator for communication along a specified dimension.
7. Clean up by freeing the Cartesian communicators.

The program showcases essential MPI functionalities for parallel computing
applications, particularly in simulations and computations requiring structured
grid topologies.

Date: 05/30/2024
Author: Djamil Lakhdar-Hamina
"""
from mpi4py import MPI
```

```
def check_perfect_square(x: int) -> bool:
    """
    Check if a given integer is a perfect square.

    This function takes an integer as input and determines if it is a perfect
    square. A perfect square is an integer that is the square of another integer.

    Parameters:
    x (int): The integer to be checked.

    Returns:
    bool: True if x is a perfect square, False otherwise.

    Example:
    >>> check_perfect_square(16)
    True
    >>> check_perfect_square(18)
    False
    """
    square = x**(1/2)
    if square % 1 == 0:
        return True
    else:
        return False


def main() -> None:
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    assert size >= 4 and check_perfect_square(size)
    # want to create a nxn square grid with periodicity
    n = size//2
    cart_comm = comm.Create_cart(dims=[n, n], periods=[True, True], reorder=True)

    print(f"rank {rank} has coordinates {cart_comm.Get_coords(rank)}")

    comm.Barrier()
    # Perform communication within the Cartesian communicator
    # (e.g., shift communication)
    left_neighbor, right_neighbor = cart_comm.Shift(1, 1)
```

```
        print(f"Rank {rank}: Left neighbor {left_neighbor}, Right neighbor
{right_neighbor}")

    up_neighbor, down_neighbor = cart_comm.Shift(0, 1)
    print(f"Rank {rank}: Up neighbor {up_neighbor}, Down neighbor {down_neighbor}")

    comm.Barrier()
    sub_cart_comm = cart_comm.Sub([False, True])

    print(f"Rank {rank}: has sub-coordinates {sub_cart_comm.coords}")

    # Free the Cartesian communicators
    sub_cart_comm.Free()
    cart_comm.Free()


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

When we run:

```
Unset
 mpiexec -n 4 python -m mpi4py  ../examples/communicators/cartesian_comm.py
```

This gives output:

```
rank 2 has coordinates [1, 0]
rank 0 has coordinates [0, 0]
rank 1 has coordinates [0, 1]
rank 3 has coordinates [1, 1]
Rank 2: Left neighbor 3, Right neighbor 3
Rank 2: Up neighbor 0, Down neighbor 0
Rank 1: Left neighbor 0, Right neighbor 0
Rank 1: Up neighbor 3, Down neighbor 3
Rank 0: Left neighbor 1, Right neighbor 1
Rank 0: Up neighbor 2, Down neighbor 2
Rank 3: Left neighbor 2, Right neighbor 2
Rank 3: Up neighbor 1, Down neighbor 1
Rank 2: has sub-coordinates [0]
Rank 3: has sub-coordinates [1]
Rank 1: has sub-coordinates [1]
Rank 0: has sub-coordinates [0]
```

Let us break this down line by line. First , we posit the global communicator **COMM_WORLD**. We next check that the number of processes is a perfect square so that we can create a square cartesian grid.

From the **COMM_WORLD** object we define a topology *for* **COMM_WORLD** using the method **COMM_WORLD.Create_Cart**(). Create_cart takes 3 parameters i.e. dims, periods, reorder and then returns a Cartcomm, a cartesian communicator. The first, **dims**, means dimensions. A line has 1 dimension, a square 2, a cube 3, and so on. A dimension is expressed as a list or sequence of integers, an integer for the size of each dimension. So let us say we want a grid with 3 rows and 4 columns then dims=[3,4], the first 3 is the number of rows and the second is the number of columns.

The second argument, **periods**, defines the “adjacency” of the first and last member in a dimension. In python it is expressed as a sequence of Booleans (**True** and **False**), one spot for each dimension. **True** means we “wrap-around” and **False** means there is no wrap-around. The figure below will clarify what this means. Finally, **reorder** is a Boolean , it dictates whether we allow MPI to cleverly organize the grid to maximize efficiency based on the specificities of the system in terms

of hardware and software. The figure below depicts the output for the 2x2 grid, reference the figure when looking at the output above:

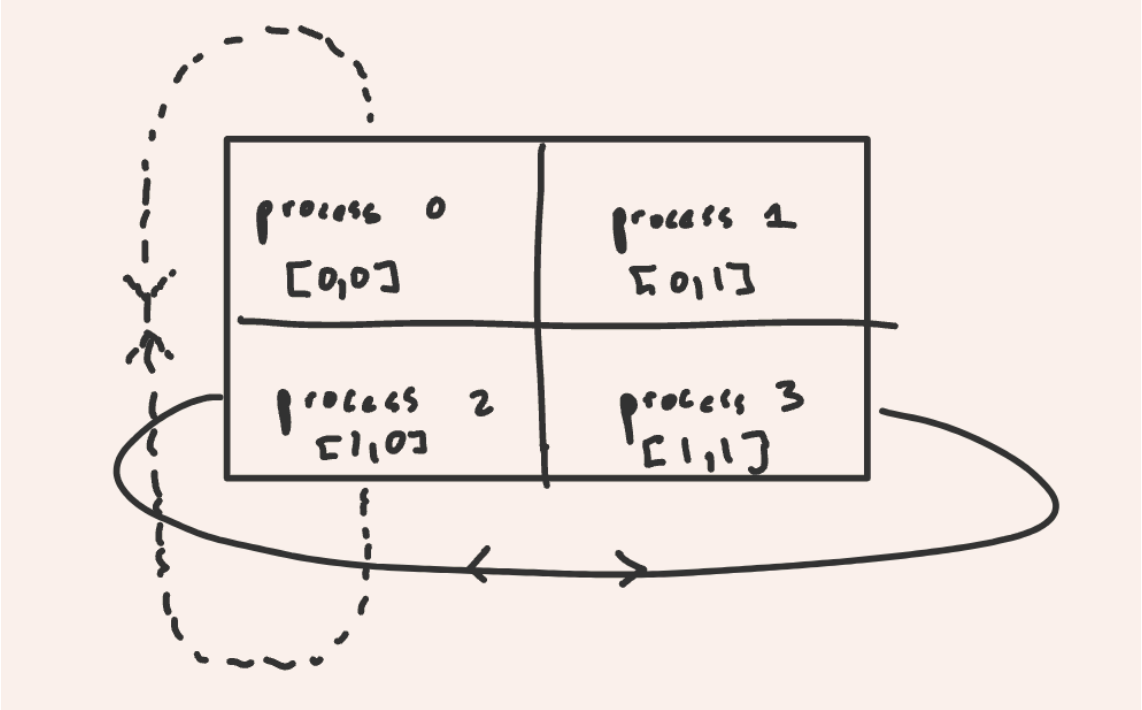



*Figure 3.6.1.1.1 This is a 2x2 Cartesian grid. Each cell contains a process identified by rank. The [m,n] brackets define the coordinate of the process within the grid. The loop around arrows shows the periodicity. For instance, as per output, the up neighbor of process 0 is process 2 , but so is the down neighbor.*

The next lines leverage attributes of the **Cartcomm**. The **Cartcomm** has a number of attributes , for instance each process with its rank is assigned a coordinate in the grid system and here we print those coordinates. Then we use the **Shift**() method of the Cartcomm to check the left and right, up and down neighbors, for each process. **Shift** takes two parameters , the first **direction** is actually more like dimension, it specifies the dimension we want to check. The second is **disp** which specifies the way we want to go within the dimension. So for instance when we specify direction=1 and disp=1 we are saying, “check across the column (1) to the right (1)”. Another example is if

we specify direction=0 and disp=-2 we would be saying “check across rows (0) down 2 (-2)”.

Now finally, it is possible to fragment the Cartesian grid into *sub-grids*. We use the **Sub**() method of the Cartcomm to do so. Sub() takes a sequence of Booleans, one for each dimension. The booleans specify whether the dimension is kept , retained. So when I specify Sub([False,True]) above I am saying “make two lines across columns in this square” which means make rows their own sub-grid. This process is depicted below:

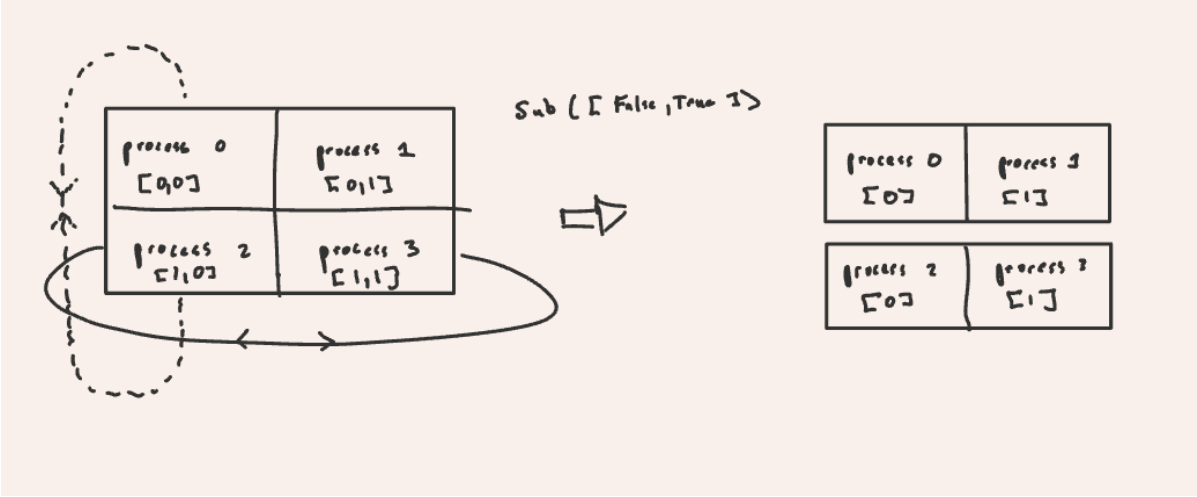



*Figure 3.6.1.1.2 Compare this figure to the output below. The single brackets in the left “line” grids are sub-coordinates.*

### 3.6.1.2 Graph Topologies

A graph is a data structure which consists of nodes, aka vertices, and edges. A node is a process with attributes. And an edge is a connection between processes. Graphs are ubiquitous and highly general data structures and they appear everywhere in computer science. The very python interpreter, its parser, makes use of a data structure called a tree and each expression entered into the interpreter is broken up into an abstract syntax tree. We can create and manipulate a graph similar to a Cartesian communicator, except the relevant object or class here is a **Graphcomm**.

```
"""
   Demonstrates the creation of a graph topology using MPI with mpi4py.

   This function initializes the MPI environment, creates a graph topology
   with predefined neighbor relationships, and prints the neighbors of each
   process. The program assumes there are exactly 3 processes, each with a
   specific set of neighbors.

   Steps:
   1. Initialize the MPI environment and obtain the rank and size of the communicator.
   2. Ensure the number of processes is exactly 3.
   3. Define the index and edges arrays to specify the graph topology.
   4. Create a graph communicator based on the index and edges.
   5. Retrieve and print the neighbors for each process.
   6. Free the graph communicator.

   This example showcases the use of MPI's graph topology capabilities to define
   and manage custom communication patterns in parallel applications.

   Date: 05/30/2024
   Author: Djamil Lakhdar-Hamina
"""
from mpi4py import MPI


def main() -> None:
   comm = MPI.COMM_WORLD
   rank = comm.Get_rank()
   size = comm.Get_size()

   assert size == 3
   index = [
       1,
       3,
       4,
   ]  # Process 0 has 1 neighbor, process 1 has 2 neighbors, process 2 has 2 neighbors
   edges = [
       1,
       0,
       2,
       1,
```

```
    ]  # Process 0's neighbors is 1 , Process 1's neighbors are 0 and 2, Process 2's is
1


    # Create the graph topology
    graph_comm = comm.Create_graph(index=index, edges=edges)

    # Get the neighbors of the current process in the graph topology
    my_neighbors = graph_comm.Get_neighbors(rank)

    print("Rank {}: My neighbors are {}".format(rank, my_neighbors))

    # Free the graph communicator
    graph_comm.Free()


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

Running:

```
Unset
 mpiexec -n 3 python -m mpi4py  ../examples/communicators/graph_comm.py
```

The output returns:

```
Rank 0: My neighbors are [1]
Rank 2: My neighbors are [1]
Rank 1: My neighbors are [0, 2]
```

Let us break this all down. **Create_graph** has signature:

```
(method) def Create_graph(
    index: Sequence[int],
    edges: Sequence[int],
    reorder: bool = False
) -> Graphcomm
```

How to instantiate the Graphcomm via the global communicator is a bit idiosyncratic (at least I find it less than intuitive). The **index** is a list, or sequence of integers. That list tells us from process 0,1,2…p

how many neighbors there are per process, per node. This works by stating that if process 0 has neighbors n, process 1 has neighbors, and process p has o then the list specifying this is [n, n+m,n+m+p] . Now the **edges** argument specifies the vertices. To specify connections, we have to start with process 0 and write out all neighbors in a list, then we go to process zero and list all the neighbors of 1, and so on. So for instance, say we have the graph below:

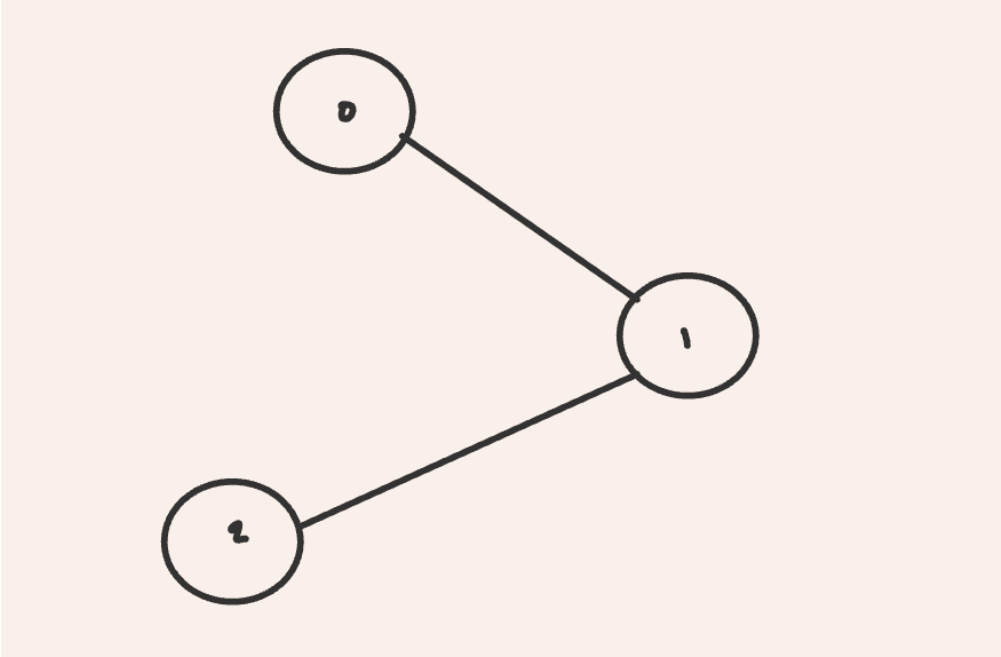



*Figure 3.6.1.2.1 We created the graph via the code block above. Process 0 has only one neighbor, process 1. Process 1 has two neighbors 0, 2. Process 2 has neighbor 1.*

The index argument would be index=[1,3,4] which tells us “process 0 has 1 neighbor, process 1 has two neighbors so add 2+1, process 2 has one neighbor so add 2+1+1”. To make the connections above we have edges=[1,0,2,1] which states “process 0 has neighbor 1, process 1 has neighbors 0 and 2, and process 2 has neighbor 1”.

### 3.6.4 Block, Cyclic, and Block-Cyclic Partitioning and darray

In the theory of parallelism, the distribution of data across processes is a deep one. We explore

## 3.8 Input-Output (IO) operations

Before we had been casually using the Python built-in print() function to print contents to screen. But how does mpi4py or MPI in general deal with stdin, stdout, and stderr? When I hit print() what is happening? 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.

### 3.8.1 File Class

Now to quote the mpi4py documentation: "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 datatypes. This data layout can be set and queried with the **File.Set_view** and **File.Get_view** methods respectively.". So the main actor on stage right now is the **File** class and we will become familiar with this class or object, along with its attributes and methods now.

#### 3.8.1.1 Point-to-Point

There is within IO for MPI, a distinction between point-to-point and collective io-operations. Point-to-point is a per process io operation, that means we actually operate on the file process by process. For example:

```
"""
Program function to demonstrate point-to-point file I/O operations using mpi4py.

Each MPI process writes its own message to a unique offset in a shared file.
The processes then read their respective messages from the file and print them.

Date: 06/02/2024
Author: Djamil Lakhdar-Hamina

"""
from mpi4py import MPI

def main() -> None:

   comm = MPI.COMM_WORLD
   rank = comm.Get_rank()
   filename = "./.point_read_write.txt"
   amode = MPI.MODE_CREATE | MPI.MODE_WRONLY
   fh = MPI.File.Open(comm=comm, filename=filename, amode=amode)

   # define massage and get file info for the read and write
   write_buffer = f"Hello world, from process {rank}!".encode("utf-8")
   buffer_size = write_buffer.__sizeof__()
   offset = buffer_size * rank

   fh.Seek(offset)
   fh.Write(write_buffer)
   fh.Sync()
   fh.Close()

   amode = MPI.MODE_RDONLY
   read_buffer = bytearray(buffer_size)
   fh = MPI.File.Open(comm, filename, amode)
   fh.Seek(offset)
   fh.Read(read_buffer)
   comm.Barrier()
   fh.Close()
```

```
    print(read_buffer.decode("utf-8"))



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

Let us break this down. We first initialize our environment as usual. Then we define an **amode** , an IO-mode. IO-modes, like read and write, are represented by MPI constants, which map to integers. There are 9 of them and they are:

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.

These modes can be combined using bitwise OR to specify multiple behaviors when opening a file and that is precisely what is done in our program. Next we create an instance of the **File** class via **MPI.File.Open** which takes three arguments. The first is **comm** , it takes the communicator that will operate with the file. The second is **filename** and the third is **amode** , the sorts of operations that will be performed with this **File** object. We then define our message, and we make sure that the message is in a buffer-form (here simply a string-bytes). The message needs to be in buffer-form whether that is string-bytes or a numpy array.

We then determine the size of the message in bytes and define an offset. An offset is simply the place in the file, the line in the file , where a message will be written to or read from. This allows each process to write to a new line so as not to read or write over the contents of the other processes. Now in order to access that offset we use the **Seek** method. Once we have accessed the line that the process will operate on, we use **Write** and then we make sure that each process has written to memory via **Sync**. After all writes are complete we close the file. We then reopen it in **MODE_RDONLY** mode and do much of the same to read the message. Run this program yourself and see how it works.

#### 3.8.1.2 Collective

Collective io means that all processes contribute to the io-operation(s). The main difference from the point of the user is that we do not need to specify the **Seek** operation. The following shows how this works:

```
"""
Demonstrates collective read and write operations using mpi4py.

This program performs the following steps:
1. Initializes MPI and obtains the rank of each process.
2. Each process prepares a unique message and calculates its offset.
3. All processes collectively write their messages to distinct offsets in
the same file using `Write_at_all`.
4. Each process reads its own message from the file using `Read_at_all`.
5. The read messages are gathered at the root process and printed in order.
```

```
6. The file is deleted after the operations are complete.

The root process coordinates the output messages to indicate the progress
of write and read operations.

Date: 06/02/2024
Author: Djamil Lakhdar-Hamina
"""

from mpi4py import MPI


def main() -> None:
   comm = MPI.COMM_WORLD
   rank = comm.Get_rank()
   status = MPI.Status()

   filename = "./.collective_read_write.txt"
   amode = MPI.MODE_CREATE | MPI.MODE_RDWR | MPI.MODE_APPEND
   fh = MPI.File.Open(comm=comm, filename=filename, amode=amode)
   message = f"Hello world, from process {rank}!\n".encode()
   chunk_size = message.__sizeof__()
   offset = chunk_size * rank

   # write section
   if rank == 0 :
       print("now writing...")
   fh.Write_at_all(offset, message, status)

   # read section
   buffer = bytearray(chunk_size)
   fh.Read_at_all(offset, buffer)

   # Gather all buffers
   all_buffer = comm.gather(buffer, root=0)
   # Print from root in order
   if rank == 0:
       print("now reading...")
       for buffer in reversed(all_buffer):
           print(buffer.decode(), end="")
       fh.Delete(filename=filename)
   fh.Close()
```

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

Notice the use of **Write_at_all** and **Read_at_all** a method which takes an **offset** and the **message**. This will write and read the messages in order. Much the same could have been accomplished with **Write_at** and **Read_at**, which takes the same arguments, and even more briefly via **Write_ordered** and **Read_ordered** which takes only the message for argument. Finally, notice the **Delete** function which takes only the filename and does exactly what it says.

Now in the opening section 3.8.1 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.

```
"""
Write data (rank number) 10 times from a numpy buffer to file in parallel.

Each MPI process writes its rank to a file in a non-contiguous pattern.
The file view is set such that each process writes to a distinct section of the file.
The program demonstrates parallel I/O using mpi4py with custom data types.

Date: 05/19/2024
Author: Djamil Lakhdar-Hamina
"""

import numpy as np
from mpi4py import MPI


def main() -> None:
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    status = MPI.Status()
```

```
    # set file mode then open the file handler
    filename = "./.noncontig_read_write.txt"
    amode = MPI.MODE_CREATE | MPI.MODE_RDWR | MPI.MODE_APPEND
    fh = MPI.File.Open(comm, filename, amode)

    item_count = 10

    buffer = np.empty(item_count, dtype="i")
    buffer[:] = rank

    # create and commit custom data structure
    filetype = MPI.INT.Create_vector(item_count, 1, size)
    filetype.Commit()

    displacement = MPI.INT.Get_size() * item_count * rank
    offset = MPI.INT.Get_size() * item_count
    fh.Set_view(displacement, filetype=filetype)
    fh.Write_at_all(offset, buffer, status)

    new_buffer = np.empty(item_count, dtype="i")
    info = fh.Get_view()
    print(info)
    fh.Read_at_all(offset, new_buffer)
    print(new_buffer)

    filetype.Free()
    fh.Close()


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

Which gives the following output for 3 processes:

```
(40, <mpi4py.MPI.Datatype object at 0x104b94030>, <mpi4py.MPI.Datatype object at 0x104b95440>, '
native')
(80, <mpi4py.MPI.Datatype object at 0x10079d440>, <mpi4py.MPI.Datatype object at 0x10079c840>, '
native')
(0, <mpi4py.MPI.Datatype object at 0x102b01440>, <mpi4py.MPI.Datatype object at 0x102b00840>, 'n
ative')
[0 0 0 0 0 0 0 0 0 0]
[2 2 2 2 2 2 2 2 2 2]
[1 1 1 1 1 1 1 1 1 1]
```

Now let us break this up. Everything is very much vanilla until we hit:

```
    # create and commit custom data structure
    filetype = MPI.INT.Create_vector(item_count, 1, size)
    filetype.Commit()
```

This code leverages the ability to define new MPI types which can be efficiently communicated across processes. Here we are creating a vector of integers where **item_count** is the number of blocks to write, **1** is the number of items to write in the block and size is the **stride** of the vector (which we explain in section 2.3). We then commit the datatype, this datatype will now be the elementary datatype that defines a per-view access of the file. Section 3.9 goes into detail as to these custom datatypes.

Next we determine the **displacement** between writes, making sure that each write accesses a non-overlapping portion of the file. We utilize this **displacement** to access the **View** that each process operates on via **Set_view** which has signature:

```
(method) def Set_view(
    disp: int = 0,
    etype: Datatype = BYTE,
    filetype: Datatype | None = None,
    datarep: str = 'native',
    info: Info = INFO_NULL
) -> None
```

**Disp** is the line where the process operates, **etype** is the elementary datatype to be accessed in the file which by default is a byte. **Filetype** is the datatype that MPI uses to define the layout of the file. The **datarep** is the representation within the file. We then do much of the same in reading and writing data. However, notice the output to the print statement of the return value of **Get_view()**. This **Get_view** returns the position of the access (per process) in terms of byte size. It then also tells us the datatypes defining the layout of the file and finally the data representation within the file.

Non-contiguous IO allows for improvements in performance as it reduces the number of I/O operations by consolidating multiple non-adjacent accesses into a single operation. It also allows for complex data access patterns without requiring the data to be reorganized contiguously in memory or storage.

#### 3.8.2 Advanced Application: CSV Reader for Datatables

Let us say we want to recreate a panda datatable csv reader. The function we

### 3.9 Custom MPI Datatypes

### 3.10 Applications to Bioinformatics

## 4. Simple Linux Utility for Resource Management: SLURM

Slurm stands for “Simple Linux Utility for Resource Management”. It is as stated in the acronym, a *resource manager* and is both open-source, and highly-scalable for linux clusters. It has three primary functions all centered on facilitating parallelism:

1. Allocates exclusive or non-exclusive resources (compute nodes) to users to perform computation
2. Provides interface to start, stop, and monitor parallel computation
3. Arbitrates conflicting demands made on resources via a queue of pending work (a queue whose structure can be defined).

Let us delve into its architecture before going onto study how to use slurm for deploying and managing parallel jobs.

## 4.1 Architecture

### 4.1.1 Slurmctld

Slurm has a central manager or daemon (demon), the **slurmctld** which coordinates the work in that it handles job queues, scheduling , resource allocation, and monitoring. It also accepts job submissions and decides which jobs should be run and in what priority. Furthermore, it keeps track of the status of all nodes and their resources (CPU, memory, energy use) and communicates with the **slurmd** daemon on each node. One can configure multiple **slurmctld** daemons in order to create a backup, this minimizes downtime and allows for seamless continuation of operation.

### 4.1.2 Slurmd

Each compute node has a **slurmd** daemon which is analogous to a remote shell in that it waits for work ,executes the work, returns status, and waits for more work. It is responsible for launching and managing jobs on the respective node, monitors the resources and reports the status of the node and job to **slurmctld**, and overseas communication with **slurmctld** in general.

### 4.1.3 Overview of *Some* User Commands

As for the user, there are many CLI tools, each of which are extremely rich in functionality.

- **srun** is the most basic way of initiating and defining job.
- **scancel** allows for termination of queued or running jobs
- **sinfo** reports system status
- **squeue** reports the status of jobs
- **sacct** allows us to get information about jobs and job steps
- **sview** command graphically reports system and job status including network topology.

### 4.1.3 Some SLURM Terminology

In terms of resources managed by the Slurm daemons: there are **nodes,** the most basic computational unit in the cluster, **partitions** which logically group nodes, **jobs** which are specified allocations of resources to a user for a definite period of time, **job steps** which are sets of jobs within a job. Let us delve some more into all the sorts of computing resources and the way they are described and organized:

- Slurm manages a **cluster** , a group of **nodes** with the capacity to interact with one another over a network. A (*compute*) **node** is a computer and is part of a larger set of nodes (a cluster). There are **compute** nodes as well as other sorts of nodes such as *login* nodes, *file server* nodes, *management* nodes, etc. A compute node offers resources such as processors, volatile memory (RAM), permanent disk space (e.g. SSD), accelerators (e.g. GPU) etc. A **processor** is the thing that , overall, actually *does* the computation.
- A **core** is the part of a processor that does the computations. A processor comprises multiple cores, as well as a memory controller, a bus controller, and possibly many other components. A processor in the Slurm context is referred to as a **socket**, which actually is the name of the slot on the motherboard that hosts the processor. A single core can have one or two **hardware threads**. This is a technology that allows virtually doubling the number of cores the operating systems perceives while only doubling part of the core components -- typically the components related to memory and I/O and not the computation components. Hardware multi-threading is very often disabled in HPC.
- a **CPU** in a general context refers to a processor, but in the Slurm context, a CPU is a consumable resource offered by a node. It can refer to a socket, a core, or a hardware thread, based on the Slurm configuration.
- A **partition** acts like a job queue which, furthermore, has structure imposed on it like job size limits, job time limit, users which have access to it etc. Once a job is assigned a set of nodes , the user is able to initiate parallel work in the form

> of job steps in any configuration within that allocation. For instance, a single job step may be started that utilizes all nodes allocated to the job, or several job steps may independently use a portion of the allocation i.e. a subset of the nodes allocated. Slurm provides resource management for the processors allocated to a job, so that multiple job steps can be simultaneously submitted and queued until there are available resources within the job's allocation.

The role of Slurm is to match those resources to **jobs**. A job comprises one or more (sequential) **steps**, and each step has one or more (parallel) **tasks**. A task is an instance of a running program, i.e. at a process, possibly along with **subprocesses** or **software threads**. Multiple tasks are dispatched on possibly multiple nodes depending on how many cores each task needs. The number of cores a task needs depends on the number of subprocesses or software threads in the instance of the running program. The idea is to map each hardware thread to one core, and make sure that each task has all assigned cores assigned on the same node.

*Figure 4.1.1-4.1.2 exhibits the basic set-up/ layout of SLURM.*

## 4.2 Installation and Set-up

Some notes on our system.

In terms of hardware,

How was this all set-up? Well first, we had to install everything needed for slurm and mpi using chroot. What chroot is it

Your home directories are mounted on a shared storage device , a storage device shared between all the nodes. A Network File System (NFS) server hosts the shared file system which can be accessed over a network and allows for read and write operations which are synchronized across all the nodes. Now let us delve into the various user commands:

### 4.3 How to Run a Job

We go over in detail how to run jobs interactively (via srun and salloc) and non-interactively (via sbatch).

#### 4.3.1 Srun

Srun is the basic means of running a job in *real-time*. The man page for srun is enormous and exhibits how rich the srun command is in features. I recommend you read through the man page *once in its entirety before proceeding*. We will exhibit the most useful and common options in srun via a series of examples from basic to complex.

##### Simple Job Submission

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun hostname
c-node1
```

We simply put a bash command after srun. This could of course be a command consisting of the **python** command and a python script or in fact any programming language.  You can inelegantly get a command to stop running via control-c.

We can run a stupid bash file such as:

```
#!/bin/bash
echo "Hello world!"
```

Remembering to make the file executable we run:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ chmod +x hello-world.sh
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun hello-world.sh
Hello world!
```

It is good practice to write out the full path, and not utilize the relative path as was done above.

### Number of Tasks

Remember that a task is synonymous basically with a process , it is the computation say “hostname” executed a certain number of times.

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -n 3 hostname
c-node1
c-node1
c-node1
```

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun --ntasks 3 hostname
c-node1
c-node1
c-node1
```

### Number of Nodes

We can of course dictate the number of nodes per **-N** or –-**nodes** option.

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -N 3 hostname
c-node1
c-node2
c-node3
```

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun --nodes 3 hostname
c-node2
c-node1
c-node3
```

We can even dictate a range of nodes to use, a min-max expression:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -N 8-16 hostname
c-node5
c-node8
c-node4
c-node3
c-node7
c-node1
c-node6
c-node2
```

Right now as of writing nodes 9-16 are down. The slurm system is smart enough to know that when I request a minimum of 8 and a maximum of 16 that because nodes 9-16 are down, then srun can only run on 8 nodes. If I had say 12 up, then the srun-job would have run 12.

Number of tasks, number of nodes , number of cpus

We can specify the number of tasks along with nodes and cpus.

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -N 4 --ntasks 3 hostname
srun: Warning: can't run 3 processes on 4 nodes, setting nnodes to 3
c-node2
c-node1
c-node3
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -N 4 --ntasks 5 hostname
c-node4
c-node2
c-node3
c-node1
c-node1
```

Notice that the number of tasks must exceed the number of nodes. Slurm determines how to balance the tasks across the nodes. What if we want to make sure that *a* task is performed by *a cpu*. We simply use the **--cpu-bind** option which is rich in functionality. The --cpu-bind option in SLURM is used to control how tasks are bound to CPUs within a node. This binding ensures that a task runs on specific CPUs or CPU cores, which can improve performance by reducing context switching and cache misses, and by ensuring that tasks don't migrate between CPUs for instance:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -N 4 --ntasks 5 --cpu-bind=cores hostname
c-node4
c-node2
c-node3
c-node1
c-node1
```

All **--cpu-bind=cores** does is specify that a task runs on a core. We could have specified **none** if we do not care to bind a task to a cpu. We can equivalently write:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -N 4 --ntasks 5 --cpus-per-task=1 hostname
c-node4
c-node2
c-node3
c-node1
c-node1
```

But what if we want to make sure that certain tasks run only on certain cores or that certain cores take up n number of tasks e.g. we would rather have c-node4 print out twice? There is a means. A hacky way is to write:

```
srun --ntasks=1 --nodes=1 -w c-node1 hostname : \
    --ntasks=1 --nodes=1 -w c-node2 hostname : \
    --ntasks=1 --nodes=1 -w c-node3 hostname : \
    --ntasks=2 --nodes=1 -w c-node4 hostname
```

Which gives :

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun --ntasks=1 --nodes=1 -w c-node1 hostname : \
>      --ntasks=1 --nodes=1 -w c-node2 hostname : \
>      --ntasks=1 --nodes=1 -w c-node3 hostname : \
>      --ntasks=2 --nodes=1 -w c-node4 hostname
srun: job 325 queued and waiting for resources

srun: job 325 has been allocated resources
c-node4
c-node2
c-node1
c-node3
c-node4
```

Which utilizes the **:** command to chain together different srun commands.

We can also be clear about the division of tasks across sockets, cores, and nodes. First, let us understand the architecture of our head node a little better (as a representative also of the other nodes) via **lscpu**:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              48
On-line CPU(s) list: 0-47
Thread(s) per core:  2
Core(s) per socket:  12
Socket(s):           2
NUMA node(s):        2
Vendor ID:           GenuineIntel
CPU family:          6
Model:               63
Model name:          Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz
Stepping:            2
CPU MHz:             3100.000
CPU max MHz:         3100.0000
CPU min MHz:         1200.0000
BogoMIPS:            4600.18
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            30720K
NUMA node0 CPU(s):   0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46
NUMA node1 CPU(s):   1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat ps
opl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx es
hf_lm abm cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpri
pln pts md_clear flush_l1d
```

So we have 2 sockets , and we have 12 cores per socket, so we have 24 cores overall on each node. So let us say we want to devote a socket per node to each task then we can use the **--ntasks-per-socket**:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -n 5 --ntasks-per-socket=2 hostname
c-node2
c-node2
c-node1
c-node1
c-node1
```

Basically what happens is that in terms of resource allocation you get:

- Node 1, Socket 1: 2 tasks
- Node 1, Socket 2: 2 tasks
- Node 2, Socket 1: 2 tasks

Now as for distributing tasks over cores we have **--ntasks-per-core:**

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -n 5 --ntasks-per-core=1 hostname
c-node1
c-node1
c-node1
c-node1
c-node1
```

If we had specified a number of processes exceeding the number of cores then more nodes would have been brought into play. Finally, we can specify distribution of tasks to nodes:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -n 5 --ntasks-per-node=1 hostname
c-node5
c-node4
c-node2
c-node3
c-node1
```

## Exact Nodes List

We can actually specify which nodes we want to utilize via the **-w** or **--nodelist** option and either a comma-delimited cli list, a range expression, or a file. We can use the cli list as:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun --nodelist=c-node1,c-node2,c-node3 hostname
c-node1
c-node2
c-node3
```

Notice that the nodelist by itself dictates the number of tasks. Here there are 3 tasks. But what if we put in a number of tasks?

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun --ntasks 5 --nodelist=c-node1,c-node2,c-node3 hostname
c-node3
c-node2
c-node2
c-node1
c-node1
```

We can also make use of range-expressions as follows, say we want to print the hostname for nodes 1-8:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun --nodelist=c-node[1-8] hostname
c-node8
c-node4
c-node5
c-node7
c-node1
c-node2
c-node3
c-node6
```

We can even break up this range-expressions as follows, say we want to print the hostname for nodes 1-3, then for nodes 4-6:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun --nodelist=c-node[1-3,4-6] hostname
c-node4
c-node5
c-node1
c-node3
c-node2
c-node6
```

Finally we can utilize a *file* , it should itself be a comma-delimited list. Let us say we want to run our command in nodes 1,3, 5, and 7. I defined a script:

```
#!/bin/bash

################################################################################
# Script: generate_comma_delimited_list.sh
#
# Description:
#   This script generates a comma-delimited list of strings based on input
#   parameters (start, end, step) and writes it to a specified output file.
#   Each element in the list is formatted as "c-node<i>", where <i> is a
#   sequential number within the specified range.
#
# Usage:
#   ./generate_comma_delimited_list.sh <start> <end> <step> <output_file>
#
#   Parameters:
#     <start>: Starting number of the sequence.
#     <end>: Ending number of the sequence.
#     <step>: Step size for incrementing the sequence.
#     <output_file>: File where the comma-delimited list will be saved.
#
# Example:
#   To generate a list from 1 to 10 with a step of 2 and save it to "nodes.txt":
#   ./generate_comma_delimited_list.sh 1 10 2 nodes.txt
#
# Author: Djamil-Lakhdar-Hamina
# Date: 06/05/2024
#
################################################################################

set -e

# Check if all required arguments are provided
if [ "$#" -ne 4 ]; then
   echo "Usage: $0 <start> <end> <step> <output_file>"
   exit 1
fi

# Assign command-line arguments to variables
```

```
start=$1
end=$2
step=$3
output_file=$4

# Initialize an empty string to store the comma-separated list
output=""

# Check if all required arguments are provided
# Loop through the sequence from start to end with the specified step
for (( i=start; i<=end; i+=step )); do
   # Append "c-node${i}" followed by a comma (except after the last element)
   if [ -n "$output" ]; then
       output+=","  # Add comma only if output is not empty
   fi
   output="c-node${i}"
done

# Write the output string to the specified file
echo -n "$output" > "$output_file"

echo "Comma-delimited list written to $output_file"

exit 0
```

So to generate a node-list we run the first command as shown, and then to use srun we run the second command as shown:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ ./generate_nodelist.sh 1 8 2 nodelist.txt
Comma-delimited list written to nodelist.txt
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun --nodelist /home/dlakhdar/excludes.txt  hostname
c-node5
c-node1
c-node3
c-node7
```

Utilize the file *full-path* to the nodelist as slurm recognizes this as a file only with the inclusion of the “/” symbol.

You can also *exclude* which nodes to not use via the **-x** or **--exclude** option. You can specify which to exclude via explicit list, range, or file.

Memory Options

We can actually control the RAM memory utilized in jobs. The most basic option is via **--mem** which controls the memory requested in *megabytes* . But we can of course specify the units, I recommend using **GB.** So for example let us say we have a highly memory bound operation. A good example is IO operations. We have a script below :

```
#!/bin/bash
################################################################################
# Script: sum_numbers.sh
#
# Description:
#   This script calculates the sum of numbers from 1 to 1000 and prints the result.
#
# Usage:
#   ./sum_numbers.sh 1 100 2 out.txt
#
# Notes:
#   This script uses a simple loop to iteratively add numbers from 1 to 1000.
#   It demonstrates basic arithmetic operations and variable usage in Bash scripting.
#
# Example output:
#   sum of 1-1000 is 500500
################################################################################

set -e

# Check if all required arguments are provided
if [ "$#" -ne 4 ]; then
   echo "Usage: $0 <start> <end> <step> <output_file>"
   exit 1
fi

# Assign command-line arguments to variables
start=$1
end=$2
step=$3
output_file=$4
counter=0
```

```
for (( i=start; i<=end; i+=step )); do
   counter=$(($counter+$i))
   echo -n "${i}" >> ${output_file}.txt
done
echo "sum of ${start}-${end} is $counter"

rm ${output_file}.txt

exit 0
```

This script accumulates numbers in some range specified by {start..stop..end} and at each step of the iteration it prints it to a file. If say start-stop defines a range of 1000 numbers this becomes a highly expensive operation. Let us see how to run this script and then how to run **srun** overall:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -N 3 --mem=5GB ./sum_numbers.sh 1 5000 1 small_memory_allocation
sum of 1-5000 is 12502500
sum of 1-5000 is 12502500
sum of 1-5000 is 12502500
```

**NOTE**: A memory size specification of zero is treated as a special case and grants the job access to all of the memory on each node for newly submitted jobs and all available job memory to new job steps. So let us say that I execute job 42 , if I specify –-mem=0GB across nodes 1,2, and 3 then my job has access to *all* the memory in 1,2,3. Now this option in fact has a system-configured environmental default which we can view via **scontrol show config.** It is called **DefMemPerNode,** (default memory per node) and is linked to **MaxMemPerNode,** which is the maximum memory one can request per node. We will return to this topic of system-variables later but for now we can investigate any given environment via the following command:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ scontrol show config | grep "DefMemPerNode"
DefMemPerNode           = UNLIMITED
```

You can search for any variable by replacing “DefMemPerNodes” with the variable of your choice.

We can in fact specify which sort of memory we care about, let us say we care about CPU memory then we would use the -**-mem-per-cpu** option. GPU would be -**-mem-per-gpu.**

### Partition Options

We can specify which partition to use via **-p** or **--partition** :

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun --nodelist=c-node[1-8] --partition=normal hostname
c-node5
c-node8
c-node4
c-node1
c-node2
c-node7
c-node3
c-node6
```

We can specify multiple partitions via a comma-delimited list.

### Time options

Let us say we want to dictate when a job starts, when it ends, how long it runs.

First we must be clear on what the time is for our system, utilize the **date** to make sure:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ date
Thu Jun  6 03:04:17 GMT 2024
```

We can dictate when a job begins via the **-b** or **--begin** option. From the documentation :

“It accepts times of the form *HH:MM:SS* to run a job at a specific time of day (seconds are optional). (If that time is already past, the next day is assumed.) You may also specify *midnight*, *noon*, *elevenses* (11 AM), *fika* (3 PM) or *teatime* (4 PM) and you can have a time-of-day suffixed with *AM* or *PM* for running in the morning or the evening. You can also say what day the job will be run, by specifying a date of the form *MMDDYY* or *MM/DD/YY YYYY-MM-DD*. Combine date and time using the following format *YYYY-MM-DD[THH:MM[:SS]]*. You can also give times like *now + count time-units*, where the time-units can be *seconds* (default),

*minutes*, *hours*, *days*, or *weeks* and you can tell Slurm to run the job today with the keyword *today* and to run the job tomorrow with the keyword *tomorrow*. The value may be changed after job submission using the **scontrol** command"

So a series of examples:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ \
> srun --begin=16:00  hostname : \
> srun --begin=now+1hour hostname : \
> srun --begin=now+60 hostname : \
> srun --begin=2024-06-05T12:30:30 hostname : \
> srun --begin=fika hostname
```

We can do the same with **--deadline** which removes a job if the job is not completed by the specified time set:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ \
> srun --deadline=16:00  hostname : \
> srun --deadline=now+1hour hostname : \
> srun --deadline=now+60 hostname : \
> srun --deadline=2024-06-05T12:30:30 hostname : \
>
```

Finally we can use --**time** or **-t** command. As per documentation :

```
A time limit of zero requests that no time limit be imposed.
Acceptable time formats include "minutes", "minutes:seconds",
"hours:minutes:seconds", "days-hours", "days-hours:minutes" and
"days-hours:minutes:seconds". This option applies to job and step
allocations.
```

Here is an example. Say we have an infinite loop, and we want to stop it after 1 second, then we would write the program **endless-loop.sh**:

```
#!/bin/bash
while :; do :; done
```

Then we make it executable and the execute it via srun with the time option via:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -N 3 --time 00:00:01 ./endless-loop.sh
srun: Job step aborted: Waiting up to 32 seconds for job step to finish.
slurmstepd-c-node1: error: *** STEP 433.0 ON c-node1 CANCELLED AT 2024-06-06T03:29:26 DUE TO TIME LIMIT ***
srun: error: c-node3: task 2: Terminated
srun: error: c-node2: task 1: Terminated
srun: error: c-node1: task 0: Terminated
```

The default limit is the partition default time limit. When the limit is reached, each task in each job step is sent SIGTERM followed by SIGKILL. The interval between signals is specified by the Slurm configuration parameter **KillWait**. So we find that :

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ scontrol show config | grep OverTimeLimit
OverTimeLimit           = 0 min
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ scontrol show config | grep KillWait
KillWait                = 30 sec
```

This means that even though the job ended after 1 second via a SIGTERM, it took an extra 30 seconds for the overall job to be terminated.

### Environmental Options

Say we want to make sure that an environmental variable is defined in the head node, registered in the compute node, and then somehow utilized. There is an option **--export** which does exactly that. It's signature is:

**--export**={[ALL,]<*environment_variables*>|ALL|NONE}

The default is **ALL** so that the following works:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -N 3 echo $foo
bar
bar
bar
```

But we could also define a new variable within the command:

```
(.vhpc) (.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -N 3 --export=ALL,quuz=qaaz bash -c 'echo ${quuz}'
qaaz
qaaz
qaaz
```

We had to use the goofy **bash -c 'echo ${quuz}'** because otherwise if we use **echo ${quuz}** then the expression **${quuz}** expands in the current before **srun** executes so that **echo ""** is what is run on each node.

What if we want to broadcast say a whole *executable file* to the various compute nodes? Then we would utilize the **--bcast** option. For instance,

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 tmp]$ srun --bcast=/tmp/mine -N3 a.out
```

Will transfer **a.out** to all 3 nodes and then execute it. If this is not an executable the command will raise errors about exec format.

Finally **--chdir** does what it says, it changes the present working directory within each process.

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -N 3 --chdir=/tmp --bcast=/mine a.out
```

Job Dependency, Prolog, Epilog

Let us say you want to run a job before some job runs, or any job runs. How do we accomplish this? Via the **--dependency** option which apparently applies only to job allocations and not steps. In fact this option is the easiest means to create jobs with multiple steps.

The signature of the command is:

**-d**, **--dependency**=*<dependency_list>*

Now the dependency list takes the form :

*<type:job_id[:job_id][,type:job_id[:job_id]]>* or
*<type:job_id[:job_id][?type:job_id[:job_id]]>*

Now only two dependency types can be joined by a **,** which means and or a **?** which means or.

The various types , which I consider most relevant for our uses are, as per the documentation:

- **after:job_id[[+time][:jobid[+time]...]]**

After the specified jobs start or are cancelled and 'time' in minutes from job start or cancellation happens, this job can begin execution. If no 'time' is given then there is no delay after start or cancellation.

- **afterany:job_id[:jobid...]**

This job can begin execution after the specified jobs have terminated. This is the default dependency type.

- **afternotok:job_id[:jobid...]**

This job can begin execution after the specified jobs have terminated in some failed state (non-zero exit code, node failure, timed out, etc). This job must be submitted while the specified job is still active or within **MinJobAge** seconds after the specified job has ended.

- **afterok:job_id[:jobid...]**

This job can begin execution after the specified jobs have successfully executed (ran to completion with an exit code of zero). This job must be submitted while the specified job is still active or within **MinJobAge** seconds after the specified job has ended.

- **singleton**

This job can begin execution after any previously launched jobs sharing the same job name and user have terminated. In other words, only one job by that name and owned by that user can be running or suspended at any point in time. In a federation, a singleton dependency must be fulfilled on all clusters unless DependencyParameters=disable_remote_singleton is used in slurm.conf.

So for instance let us say that we want to execute a job if job 21 and 23 complete, *or* any of jobs 25-28 are completed. We would write:

```
srun --N 3 --dependency=afterok:20:23?afterany:25:26:27:28 hostname
```

But what if we had hundreds of jobs? It would hardly be useful to specify them like this. The answer is a job array but if we are stubborn and wish to not use that functionality we can generate a a range within a dependency string via bash arrays. The following script does just that:

```
#!/bin/bash
: '
```

```
Generate a Slurm dependency string for a sequence of job IDs.

Usage:
   ./script.sh <start> <step> <end> <type>

Arguments:
   <start> : Starting job ID.
   <step>  : Step increment for the job IDs.
   <end>   : Ending job ID.
   <type>  : Dependency type (e.g., afterok, afterany).

Example:
   ./script.sh 20 1 23 afterok

This script outputs a Slurm dependency string formatted as:
<type>:<start>:<start+step>:...:<end>
'

set -e

# initialize cli arguments
start=$1
step=$2
end=$3
type=$4

# generate job id array, turn to string, strip of extra : at end
job_ids=($(seq ${start} ${step} ${end}))
dependency_string="${type}:"
dependency_string+=$(printf "%s:" "${job_ids[@]}")
dependency_string=${dependency_string%:}

# echo the dependency string
echo ${dependency_string}

exit 0
```

Now we can run in the cli:

```
dependency_string=$(bash ./generate-dependency-string.sh 25 1 28 afterany)
srun --N 3 --dependency=afterok:20:23?${dependency_string} hostname
```

To get the same result as above.

What if instead we wanted to run an executable file *on the head node* before or after some job? That is what the **--prolog or --epilog** options , respectively, do:

```
#include <stdio.h>
int main(){
   printf("Hello World!");
   return 0 ;
}
```

Which we compile and run finally via:

```
gcc hello-world.c
chmod +x a.out
srun -N 3 --epilog=./a.out hostname
```

In thinking through dependency scripts I utilize a form of diagraming which I depict by example below in figure 4.3.1:

*Figure 4.3.1 :*

## Distribution: Block, Cyclic, Block-Cyclic of Tasks, Sockets

We have discussed modes of partitioning arrays and matrices many times by now. It turns out that we can utilize this same theory and terminology to specify resource allocation via the **-m or --distribution** option. We can control distribution via a command like :

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun -N 4 --ntasks 5 --distribution=block:cyclic:fcyclic hostname
c-node4
c-node3
c-node1
c-node1
c-node2
```

Let us break this down. First we run a command on 4 nodes, and we spawn 5 tasks. How to distribute these 5 tasks across 4 nodes? That is what **--distribution** handles. A distribution-string has 3 or 4 parts to it and so a signature like:

**--distribution**={*|block|cyclic|arbitrary|plane=<*size*>}[:{*|block|cyclic|fcyclic}[:{*|block|cyclic|fcyclic}]][,{Pack|NoPack}]

The first * is the distribution of tasks to nodes. When we specify block that means consecutive tasks share a node up until

Debugging Options

We simply mention two options **-e or --error** and **--output or -o**. Specify these in order to generate an output file that takes standard output and places it in the file and standard error and places it in the error file.

### 4.3.2 Salloc

Let us say you want to be able to allocate 3 nodes but you want to work interactively with that allocation e.g. you might be doing some exploratory work on each node and you are not sure what commands you will be running. You can specify an interactive job like this, and demand resources in the same way you did for **srun**:

Salloc is simply an interactive way to specify a number of resources and we provide an example here:

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ salloc -N 5 --ntasks=5 --ntasks-per-node=1 --time=00:30:00 --contiguous --comment="example of salloc" --job-name="salloc_example"
salloc: Granted job allocation 471
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun hostname
c-node5
c-node4
c-node1
c-node2
c-node3
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ srun echo "hello world!"
hello world!
hello world!
hello world!
hello world!
hello world!
```

### 4.3.3 Sbatch

We have spoken about running jobs “non-interactively” i.e. submitting a job to slurm and having it run in the background. This is the function of **sbatch.** We will explore how to utilize this ClI tool now.

#### Simple Job Submission

**Sbatch** as a command requires that it be executed with a **batch** script. A **batch** script is simply a shell script where the top section is population with pre-processor directives **#SBATCH --option=value .** These are the same options we explored in the **srun** section. A basic example is given below, we write the following script:

```
#!/bin/bash
#SBATCH --job-name=sbatch-hello-world
#SBATCH --nodes=4
#SBATCH --ntasks=8
#SBATCH --ntasks-per-node=2
#SBATCH --output=%x-%j.out
#SBATCH --error=%x-%j.err

# This script submits a Slurm job that runs across 4 nodes with a total of 8 tasks.
# Each node runs 2 tasks. The job prints "Hello world!" along with the job name.
# Output and error messages are saved in files named after the job name and job ID.

working_dir=$(pwd)
job_id=$SLURM_JOB_ID
job_name=$SLURM_JOB_NAME

if [ -d ${working_dir}/tmp ]; then
   echo "directory exists"
else
   echo "creating directory..."
   mkdir ${working_dir}/tmp
fi

echod "redirecting to stderr"
```

```
srun echo "Hello world! From ${job_name}"

mv *.out $(pwd)/tmp/ &&  mv *.err $(pwd)/tmp/

exit 0
```

Let us break this down. The first lines past the shebang are **#SBATCH** directives. The directives make use of file-name patterns. These file-name patterns can be utilized in options but not in the body of the script itself. **%x** and **%j** ,respectively, mean job name and job id. You can look at the full list here. We then leverage the slurm environmental variable **SLURM_JOB_NAME**. There is a full list of environmental variables available to **sbatch** here in the section "Input Environmental Variables" or in the appendix.

We create a temporary file in the present working directory, a directory given by the **pwd** command. We then run an expression **echod** which is not a valid command so that the error gets redirected to our error file. Finally , we run an **srun** command, all the options specified in the **#SBATCH** directives are picked up by the **srun** command. Finally, we move all our error and output files to a temporary directory within the present working directory. The output is given here:

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ sbatch --time=00:10:00 sbatch-hello-world.sh
Submitted batch job 483
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ cat *483*
/var/spool/slurmd/job00483/slurm_script: line 24: echod: command not found
directory exists
Hello world! From sbatch-hello-world
Hello world! From sbatch-hello-world
Hello world! From sbatch-hello-world
Hello world! From sbatch-hello-world
Hello world! From sbatch-hello-world
Hello world! From sbatch-hello-world
Hello world! From sbatch-hello-world
Hello world! From sbatch-hello-world
```

Dependencies

Above in the srun section we mentioned dependencies, but best practice is to utilize the dependency feature to create jobs with steps via sbatch. The way I do so is by writing a dependency coordinating script. Let us say we run 5 jobs, then we run another job which depends on *any* of those other jobs completing, before finally running a job that depends on that job completing. We utilize our dependency string generator as per above in the section on Job dependency in srun.

```
#!/bin/bash
# Script: coordinate-dependency-sbatch.sh
# Description: This script coordinates the submission of two SLURM jobs.
#              The first job submits multiple instances of sbatch-hostname.sh
#              and collects their job IDs. It then calculates a dependency
#              string based on the IDs of the first and last jobs.
#              The second job is submitted with dependencies on the first job,
#              and runs a simple 'echo' command on SLURM.
# Usage: ./coordinate-dependency-sbatch.sh <NODE_NUMBER> <TASK_NUMBER>
# Example: ./coordinate-dependency-sbatch.sh 4 8

set -e   # Exit immediately if a command exits with a non-zero status

STEP=1
NODE_NUMBER=$1
TASK_NUMBER=$2
num_arguments=$#

# Check if exactly two CLI arguments are passed
if [[ ${num_arguments} != 2 ]]; then
   echo "Not enough CLI arguments passed. Expected 2, got ${num_arguments}."
   exit 1
fi

# Generate sbatch-hostname.sh script with SLURM directives
cat <<EOF > sbatch-hostname.sh
#!/bin/bash
#SBATCH --nodes=${NODE_NUMBER}
#SBATCH --ntasks=${TASK_NUMBER}
srun hostname
EOF
```

```
chmod +x sbatch-hostname.sh

ids=()
#First steps
for i in {1..4};
do
  ids+=($(sbatch sbatch-hostname.sh | awk '{print $4}'))
done

echo "Submitted batch jobs ${ids[@]}"

first_index=${ids[0]}
array_last_index=$((${#ids[@]} - 1))
last_index=${ids[array_last_index]}

dependency_string=$(bash ./generate-dependency-string.sh ${first_index} ${STEP}
${last_index} afterany)

# Second step
job_id=$(sbatch -N ${NODE_NUMBER} --dependency=${dependency_string} --wrap='srun
--nodes='"${NODE_NUMBER}"' --ntasks='"${TASK_NUMBER}"' echo "hello world"' | awk
'{print $4}')

echo "Submitted batch job ${job_id}"

sbatch --dependency=afterok:${job_id} sbatch-hostname.sh

rm sbatch-hostname.sh

exit 0
```

When we run this:

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ bash ./coordinate-dependency-sbatch.sh 3 3
Submitted batch jobs 583 584 585 586
Submitted batch job 587
Submitted batch job 588
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ rm slurm-581.out
.bash_history                    coordinate-dependency-sbatch.sh   exclude_nodes.tx
.bash_logout                     cpu_info.txt                      excludes.txt
.bash_profile                    cpu-intensive.sh                  generate-depende
.bashrc                          .dotnet/                          generate_nodelis
.cache/                          .emacs                            hello-world.c
.conda/                          endless-loop.sh                   hello-world.sh
coordinate-dependency-sbatch.out  .esmtp_queue/                    .python_history
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ rm slurm-581.out
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ rm slurm-582.out
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ cat slurm*.out
c-node1
c-node3
c-node2
c-node4
c-node5
c-node6
c-node4
c-node5
c-node6
c-node3
c-node1
c-node2
hello world
hello world
hello world
c-node1
c-node2
c-node3
```

So let us break down this script. First we set bash options **-set e** which dictates that on the first command error the program is to stop running. Then we unpack a number of arguments and check the number of arguments there are via the special character **$#**. Next we check that the requisite number of arguments are passed.

Next we generate what is called a **heredoc** via the :

```
# Generate sbatch-hostname.sh script with SLURM directives
cat <<EOF > sbatch-hostname.sh
#!/bin/bash
#SBATCH --nodes=${NODE_NUMBER}
#SBATCH --ntasks=${TASK_NUMBER}
srun hostname
EOF
```

This will create a file named **sbatch-hostname.sh** in the present working directory. We then make sure that the file is executable. We then initialize an empty array via **id=()** and then run 4 sbatch jobs in a loop making sure to save the job ids in the array via:

```
  ids+=($(sbatch sbatch-hostname.sh | awk '{print $4}'))
```

Now:

```
awk '{print $4}’
```

Now awk, a data extraction and manipulation linux tool, will pick out the output piped from the sbatch command and the {print $4} will pick out the 4th space-delimited string.

We then generate a dependency string and run an sbatch job.

```
job_id=$(sbatch -N ${NODE_NUMBER} --dependency=${dependency_string} --wrap='srun --nodes='"${NODE_NUMBER}"' --ntasks='"${TASK_NUMBER}"' echo "hello world"' | awk '{print $4}')
```

Now I have used this a lot before but with var=$(command) we are saving the result of a command as a variable (bash command substitutions). We then run an **sbatch** command but we use the nifty **--wrap** option which takes bash commands as a *string* and makes it a file that will be executed by **sbatch**. Here we wrap an **srun** command. One needs to specify externally the number of nodes in the **sbatch** command and this must agree with the number of nodes in the **srun** command or else the program will fail. Also, notice the way that we interpolate variables with single strings, we have to escape the single string via a ‘ and then wrap the variable interpolation via “. The rest of the script is rather straightforward.

### Job Array

One of the coolest features of **sbatch** is the ability to create job arrays i.e. a collection of similar jobs. Now as per the documentation:

All jobs must have the same initial options (e.g. size, time limit, etc.), however it is possible to change some of these options after the job has begun execution using the scontrol command specifying the *JobID* of the array or individual *ArrayJobID*. (here)

The interface to job arrays is simply the **--array** or **-a** option. For instance we have script:

```
#!/bin/bash
#SBATCH --job-name=sbatch-hello-world
#SBATCH --nodes=4
#SBATCH --ntasks=8
#SBATCH --ntasks-per-node=2
#SBATCH --output=%x-%j.out
#SBATCH --error=%x-%j.err

# This script submits a Slurm job that runs across 4 nodes with a total of 8 tasks.
# Each node runs 2 tasks. The job prints "Hello world!" along with the job name.
# Output and error messages are saved in files named after the job name and job ID.

working_dir=$(pwd)
job_id=$SLURM_JOB_ID
job_name=$SLURM_JOB_NAME

if [ -d ${working_dir}/tmp ]; then
   echo "directory exists"
else
   echo "creating directory..."
   mkdir ${working_dir}/tmp
fi

srun echo "Hello world! From ${job_name}"

mv *.out $(pwd)/tmp/ &&  mv *.err $(pwd)/tmp/

exit 0
```

Then we can run

```
# Submit a job array with index values between 0 and 31
$ sbatch --array=0-3 sbatch-hello-world.sh
# Submit a job array with index values of 1, 3, 5 and 7
```

```
$ sbatch --array=1,3 sbatch-hello-world.sh

# Submit a job array with index values between 1 and 7
# with a step size of 2 (i.e. 1, 3, 5 and 7)
$ sbatch --array=1-4:2  sbatch-hello-world.sh
```

Now array jobs produce certain environmental variables. Notably we should be aware of **SLURM_ARRAY_JOB_ID** which is just the job id of the first job within the array (similar to the address of an array in say C which is just the address of the first element of the array).

### 4.3.4 How to Monitor and Affect Jobs: Scontrol, Sinfo, Squeue, Scancel

We have learned to *run* jobs but now we wish to either *alter, stop, or somehow act on these running jobs*. Scontrol and scancel allow us to do all this and much more.

First, let us be clear on some terms. A node is **idle** when it is not executing any process and when it is available for use. A node is **drained** when it is currently in use but it is not accepting future jobs. A node is **down** when … well it is down i.e. not operational. And **unknown** means precisely the state is not reported or known.

#### How to Monitor SLURM Configuration and State: Scontrol

**Scontrol** allows us to monitor the Slurm configuration and state. Right off the bat **scontrol** is a command like any other, it has **options**. However, on top of **options** it includes **commands**. The form of usage is:

```
scontrol --option command
```

The most important command is **show**. The following shows all the sorts of entities in the slurm universe you can monitor and gather data on (from documentation):

**show** <*ENTITY*>[=<*ID*>] or <*ENTITY*> [<*ID*>]
Display the state of the specified entity with the specified identification.

- **config**

Displays parameter names from the configuration files in mixed case (e.g. SlurmdPort=7003) while derived parameters names are in upper case only (e.g. SLURM_VERSION). For instance:

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ scontrol show config | head
Configuration data as of 2024-06-07T20:32:41
AccountingStorageBackupHost = (null)
AccountingStorageEnforce = none
AccountingStorageHost   = localhost
AccountingStorageExternalHost = (null)
AccountingStorageParameters = (null)
AccountingStoragePort   = 0
AccountingStorageTRES   = cpu,mem,energy,node,billing,fs/disk,vmem,pages
AccountingStorageType   = accounting_storage/none
AccountingStorageUser   = root
```

Notice only the **head** is shown since the output is in fact very long. These are all the configuration parameters that are specified in the slurm.conf file.

- **daemons**

Reports which daemons should be running on this node.
ed list use **hostlistsorted** (e.g. tux2,tux1,tux2 = tux[1-2,2]).

- **job**

Displays statistics about all jobs by default. If an optional jobid is specified, details for just that job will be displayed. If the job does not specify socket-per-node, cores-per-socket or threads-per-core then it will display '*' in the ReqS:C:T=*:*:* field.

Let us say we run:

```
sbatch -N 2 --ntasks 2 --wrap='sleep 100'
```

Then we can check the status of the job via :

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ sbatch -N 2 --ntasks 2 --wrap='sleep 100'
Submitted batch job 595
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ scontrol show job=595
JobId=595 JobName=wrap
   UserId=dlakhdar(364015931) GroupId=domain users(364000513) MCS_label=N/A
   Priority=4294901620 Nice=0 Account=(null) QOS=(null)
   JobState=RUNNING Reason=None Dependency=(null)
   Requeue=1 Restarts=0 BatchFlag=1 Reboot=0 ExitCode=0:0
   RunTime=00:00:21 TimeLimit=UNLIMITED TimeMin=N/A
   SubmitTime=2024-06-07T20:44:42 EligibleTime=2024-06-07T20:44:42
   AccrueTime=2024-06-07T20:44:42
   StartTime=2024-06-07T20:44:44 EndTime=Unknown Deadline=N/A
   SuspendTime=None SecsPreSuspend=0 LastSchedEval=2024-06-07T20:44:44 Scheduler=Main
   Partition=normal AllocNode:Sid=bko-ac-hpc-21:1916002
   ReqNodeList=(null) ExcNodeList=(null)
   NodeList=c-node[1-2]
   BatchHost=c-node1
   NumNodes=2 NumCPUs=96 NumTasks=2 CPUs/Task=1 ReqB:S:C:T=0:0:*:*
   TRES=cpu=96,mem=250G,node=2,billing=96
   Socks/Node=* NtasksPerN:B:S:C=0:0:*:* CoreSpec=*
   MinCPUsNode=1 MinMemoryNode=0 MinTmpDiskNode=0
   Features=(null) DelayBoot=00:00:00
   OverSubscribe=NO Contiguous=0 Licenses=(null) Network=(null)
   Command=(null)
   WorkDir=/mnt/ace_sequencing/home/dlakhdar
   StdErr=/mnt/ace_sequencing/home/dlakhdar/slurm-595.out
   StdIn=/dev/null
   StdOut=/mnt/ace_sequencing/home/dlakhdar/slurm-595.out
   Power=
```

- **node**

Displays statistics about all nodes by default. If an optional nodename is specified, details for just that node will be displayed.

We can check say a specific list of nodes via:

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ scontrol show node=c-node[1-2]
NodeName=c-node1 Arch=x86_64 CoresPerSocket=12
   CPUAlloc=0 CPUEfctv=48 CPUTot=48 CPULoad=0.00
   AvailableFeatures=(null)
   ActiveFeatures=(null)
   Gres=(null)
   NodeAddr=c-node1 NodeHostName=c-node1 Version=22.05.10
   OS=Linux 4.18.0-513.9.1.el8_9.x86_64 #1 SMP Wed Nov 29 18:55:19 UTC 2023
   RealMemory=128000 AllocMem=0 FreeMem=126415 Sockets=2 Boards=1
   State=IDLE ThreadsPerCore=2 TmpDisk=0 Weight=1 Owner=N/A MCS_label=N/A
   Partitions=normal
   BootTime=2024-06-06T15:43:36 SlurmdStartTime=2024-06-06T16:06:39
   LastBusyTime=2024-06-07T20:46:24
   CfgTRES=cpu=48,mem=125G,billing=48
   AllocTRES=
   CapWatts=n/a
   CurrentWatts=0 AveWatts=0
   ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s

NodeName=c-node2 Arch=x86_64 CoresPerSocket=12
   CPUAlloc=0 CPUEfctv=48 CPUTot=48 CPULoad=0.00
   AvailableFeatures=(null)
   ActiveFeatures=(null)
   Gres=(null)
   NodeAddr=c-node2 NodeHostName=c-node2 Version=22.05.10
   OS=Linux 4.18.0-513.9.1.el8_9.x86_64 #1 SMP Wed Nov 29 18:55:19 UTC 2023
   RealMemory=128000 AllocMem=0 FreeMem=126465 Sockets=2 Boards=1
   State=IDLE ThreadsPerCore=2 TmpDisk=0 Weight=1 Owner=N/A MCS_label=N/A
   Partitions=normal
   BootTime=2024-06-06T15:43:53 SlurmdStartTime=2024-06-06T16:06:39
   LastBusyTime=2024-06-07T20:46:24
   CfgTRES=cpu=48,mem=125G,billing=48
   AllocTRES=
   CapWatts=n/a
   CurrentWatts=0 AveWatts=0
   ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s
```

- **partition**

Displays statistics about all partitions by default. If an optional partition name is specified, details for just that partition will be displayed. An example:

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ scontrol show partition=knl
PartitionName=knl
   AllowGroups=ALL AllowAccounts=ALL AllowQos=ALL
   AllocNodes=ALL Default=NO QoS=N/A
   DefaultTime=NONE DisableRootJobs=NO ExclusiveUser=NO GraceTime=0 Hidden=NO
   MaxNodes=UNLIMITED MaxTime=UNLIMITED MinNodes=0 LLN=NO MaxCPUsPerNode=UNLIMITED
   Nodes=c-node[9-16]
   PriorityJobFactor=1 PriorityTier=1 RootOnly=NO ReqResv=NO OverSubscribe=EXCLUSIVE
   OverTimeLimit=NONE PreemptMode=OFF
   State=UP TotalCPUs=2176 TotalNodes=8 SelectTypeParameters=NONE
   JobDefaults=(null)
   DefMemPerNode=UNLIMITED MaxMemPerNode=UNLIMITED
   TRES=cpu=2176,mem=771096M,node=8,billing=2176
```

## How to Monitor State: Sinfo

We can also use **sinfo** to get some more succinct information about the state of nodes and partitions.

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
normal*      up   infinite      8   idle c-node[1-8]
knl          up   infinite      8   idle c-node[9-16]
```

We can get a node based view via:

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ sinfo -N
NODELIST   NODES PARTITION STATE
c-node1        1   normal* idle
c-node2        1   normal* idle
c-node3        1   normal* idle
c-node4        1   normal* idle
c-node5        1   normal* idle
c-node6        1   normal* idle
c-node7        1   normal* idle
c-node8        1   normal* idle
c-node9        1       knl idle
c-node10       1       knl idle
c-node11       1       knl idle
c-node12       1       knl idle
c-node13       1       knl idle
c-node14       1       knl idle
c-node15       1       knl idle
c-node16       1       knl idle
```

How to Affect State: Scancel, Supdate

We can affect jobs such as canceling or updating the information of a job. Let us show how.

We can cancel a job, or jobs based on certain criterion or options. For instance we can cancel a job based on id or job name or based on

partition.

```
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ sbatch -N 2 --ntasks 2 --wrap='sleep 100'
Submitted batch job 596
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ scancel 596
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ sbatch -N 2 --ntasks 2 --jobname=scancel_example --wrap='sleep 100'
sbatch: unrecognized option '--jobname=scancel_example'
Try "sbatch --help" for more information
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ sbatch -N 2 --ntasks 2 --job-name=scancel_example --wrap='sleep 100'
Submitted batch job 597
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ scancel --jobname=scancel_example
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ sbatch -N 2 --ntasks 2 --partition=nknl --wrap='sleep 100'
sbatch: error: invalid partition specified: nknl
sbatch: error: Batch job submission failed: Invalid partition name specified
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ sbatch -N 2 --ntasks 2 --partition=knl --wrap='sleep 100'
Submitted batch job 598
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ scancel --partition=knl
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
               598       knl     wrap dlakhdar CG       0:10      1 c-node9
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ scancel --partition=knl
(.vhpc) [dlakhdar@bko-ac-hpc-21 ~]$ squeue
```

For a full list of options look at the documentation here.

What if we want to say suspend a job, and then resume it. Well we can use **scontrol suspend** followed by **scontrol resume.** Unfortunately by default only slurm root can utilize these commands.

Supdate

**Scontrol supdate** lets us modify any number of features of running jobs. The signature of the command is:

**update** <SuspendExc*>[=|+=|-=]<*LIST*>

Update *SuspendExcNodes*, *SuspendExcParts*, or *SuspendExcStates*. <*LIST*> is either a NodeList, list of partitions, or list of node states respectively. Use +=/-= to add/remove nodes, partitions, or states to/from the currently configured list. Use = to replace the current list. SuspendExcNodes does not support "+="/"-=" when the ":" option is used, however, direct assignment "=" is always supported. Consider using "scontrol show config | grep SuspendExc" to see current state of these settings

Unfortunately by default only slurm root can utilize these commands. So we would suspend, update, then resume jobs.

### 4.5 MPI and Slurm

We have learned *ALOT*. Now we get to the pinnacle of all of it. We will combine numpy+mpi+slurms sbatch scripts! First let us explain

#### 4.5.1 mpirun /mpiexec

To be written

## 5. Testing, Debugging, and Performance

First, I want to develop a very,very simple framework to assess speed-up and performance. This is by no means the standard in the fields in which parallel programming is ubiquitous but for our purposes I think it's sufficient to make predictions about the benefits of parallel programming.

When we assess speed-up , I break up the speed-up into two parts. There is the time it takes for a task to complete and there is the time it takes for processes to communicate. We call these unoriginally **task-time** and **communication-time**. Now say we have a serial program and it takes t units of time to compute on data of size d (it does not matter for now the units). Now let us say that we split up the computation into processes that take time t1,t2…tn. If all tasks run concurrently, then the time it takes to complete the computation on d can be taken to be the upper limit of the times t1,t2…tn :

```
T := max(t1,t2…tn)
```

We would obviously hope though that:

```
T << t
```

But, and here is the big but, the truth is that 1) processes do not always run strictly concurrently and 2) communication itself can take time, and in fact can be very costly computationally i.e. *communication isn't free*. This puts a limit

# Bibliography

## Websites

## Books

Python Data Science Handbook

# Glossary

# Appendix

## Common Bash Patterns

### Bash array commands

| Syntax | Result |
| --- | --- |
| `arr=()` | Create an empty array |
| `arr=(1 2 3)` | Initialize array |
| `${arr[2]}` | Retrieve third element |
| `${arr[@]}` | Retrieve all elements |

| | |
|---|---|
| `${!arr[@]}` | Retrieve array indices |
| `${#arr[@]}` | Calculate array size |
| `arr[0]=3` | Overwrite 1st element |
| `arr+=(4)` | Append value(s) |
| `str=$(ls)` | Save `ls` output as a string |
| `arr=( $(ls) )` | Save `ls` output as an array of files |
| `${arr[@]:s:n}` | Retrieve n elements `starting at index s` |

Common Slurm Variables

| Variable | Description |
|---|---|
| $SLURM_JOB_ID | The Job ID. |
| $SLURM_JOBID | Deprecated. Same as $SLURM_JOB_ID |

| | |
|---|---|
| $SLURM_SUBMIT_DIR | The path of the job submission directory. |
| $SLURM_SUBMIT_HOST | The hostname of the node used for job submission. |
| $SLURM_JOB_NODELIST | Contains the definition (list) of the nodes that is assigned to the job. |
| $SLURM_NODELIST | Deprecated. Same as SLURM_JOB_NODELIST. |
| $SLURM_CPUS_PER_TASK | Number of CPUs per task. |
| $SLURM_CPUS_ON_NODE | Number of CPUs on the allocated node. |
| $SLURM_JOB_CPUS_PER_NODE | Count of processors available to the job on this node. |
| $SLURM_CPUS_PER_GPU | Number of CPUs requested per allocated GPU. |
| $SLURM_MEM_PER_CPU | Memory per CPU. Same as `--mem-per-cpu` . |
| $SLURM_MEM_PER_GPU | Memory per GPU. |

| | |
|---|---|
| $SLURM_MEM_PER_NODE | Memory per node. Same as `--mem` . |
| $SLURM_GPUS | Number of GPUs requested. |
| $SLURM_NTASKS | Same as -n, –ntasks. The number of tasks. |
| $SLURM_NTASKS_PER_NODE | Number of tasks requested per node. |
| $SLURM_NTASKS_PER_SOCKET | Number of tasks requested per socket. |
| $SLURM_NTASKS_PER_CORE | Number of tasks requested per core. |
| $SLURM_NTASKS_PER_GPU | Number of tasks requested per GPU. |
| $SLURM_NPROCS | Same as `-n`, `--ntasks`. See $SLURM_NTASKS. |
| $SLURM_NNODES | Total number of nodes in the job’s resource allocation. |
| $SLURM_TASKS_PER_NODE | Number of tasks to be initiated on each node. |
| $SLURM_ARRAY_JOB_ID | Job array’s master job ID number. |

| | |
|---|---|
| $SLURM_ARRAY_TASK_ID | Job array ID (index) number. |
| $SLURM_ARRAY_TASK_COUNT | Total number of tasks in a job array. |
| $SLURM_ARRAY_TASK_MAX | Job array’s maximum ID (index) number. |
| $SLURM_ARRAY_TASK_MIN | Job array’s minimum ID (index) number. |