Programación para Física y Astronomía

Departamento de Física.

June 13, 2022

Universidad Andrés Bello®

python powered

Resumen

Clases

Clusters de Computadores

GPU Computing

MPI4PY

Actividades

Clases

Resumen

- Clases : Estructuras ’abstractas’ que nos sirven para crear objetos.
- Objetos : Elementos creados a partir de una clase que tienen propiedades y métodos.
- Cuando creamos clases, simplificamos nuestros programas. Reutilizamos códigos y estructuramos de mejor forma el desarrollo de un/el software.
- Revisión la guía

```python
#Este programa lee un fichero de datos y grafica
#la primera y segunda columna de esos datos (X vs Y)
import numpy as np
import matplolib.pyplot as plt

#Cargamos los datos del fichero
datos = np.loadtxt("datos.dat")

#Asignamos las columnas a x e y
x = datos[:,0]
y = datos[:,1]

#Graficamos y Desplegamos el grafico
plt.plot (x, y)
plt.show()
```

Cambiemos este simple programa, y construyamos una clase que podamos reutilizar cada vez que queramos graficar un par de columnas desde un archivo.

- Renombremos este programa a : *miclase.py*
- Y cambiemos el contenido a lo siguiente:

```
1  #Esta clase grafica dos columnas de un archivo
2  import numpy as np
3  import matplolib.pyplot as plt
4
5  class Grafico():
6      ca, cb = 0, 1
7      fichero = 'datos.txt'
8      def __init__(self, a, b, f):
9          self.ca, self.cb = a, b
10         fichero=f
11     def Show(self):
12         datos = np.loadtxt(self.fichero)
13         X, Y = datos[:,self.ca], datos[:,self.cb]
14         plt.plot (X, Y)
15         plt.show()
```

Usemos nuestra clase

Ahora que ya grabamos nuestro fichero ***miclase.py*** procedamos a utilizar la clase. Para ello creamos un nuevo programa llamado ***analisis1.py***

```
1  #Cargamos nuestra clase
2  from miclase import Grafico
3
4  #Graficamos columnas 0 y 1 del archivo datos.txt
5  g = Grafico(0, 1, 'datos.txt')
6
7  #Mostramos el grafico
8  g.Show()
```

Como pueden observar, si construimos esta clase, la podemos reutilzar en todos nuestros códigos de ahora en adelante. Y podemos graficar cualquiera de las dos columnas que deseemos, es muy util aprender a utilziar clases.

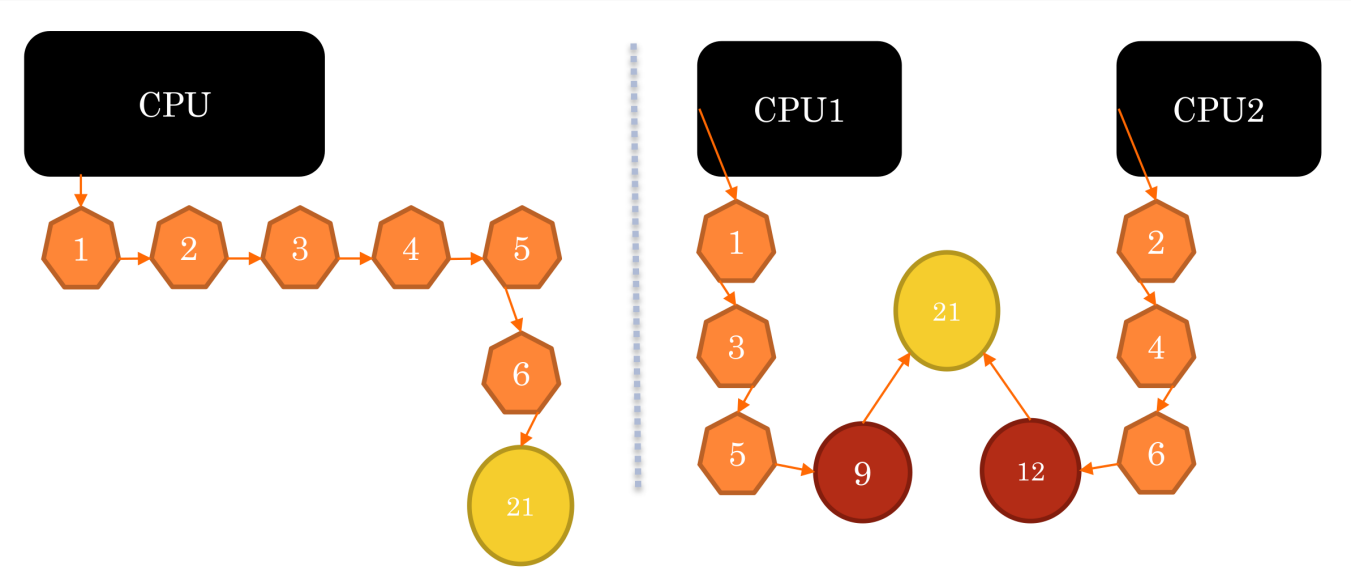
Clusters de Computadores

El rol de los clústers de cómputo

- Un clúster de cómputo es un grupo de computadores acoplados que trabajan juntos, de tal forma que pueden ser vistos como un solo computador.
- Usualmente son conectados a través de redes locales 1Gbps (slow) / Infiniband $\geq$ 56Gbps (fast)
- Su aplicabilidad es variada, desde *e-commerce*, HPC-*databases*, HPC-*scientific software*, etc.
- Su importancia
  - Precio/Performance
  - Disponibilidad
  - Escalabilidad

Clusters de Computadores

- Generalmente, se requieren software personalizados, y enfocados a ciertos problemas particulares, por lo que programar un código propio es “casi siempre” una excelente alternativa.
- ¿Cómo se organizan las CPU para trabajar en paralelo?
  - Sumando de uno a 6



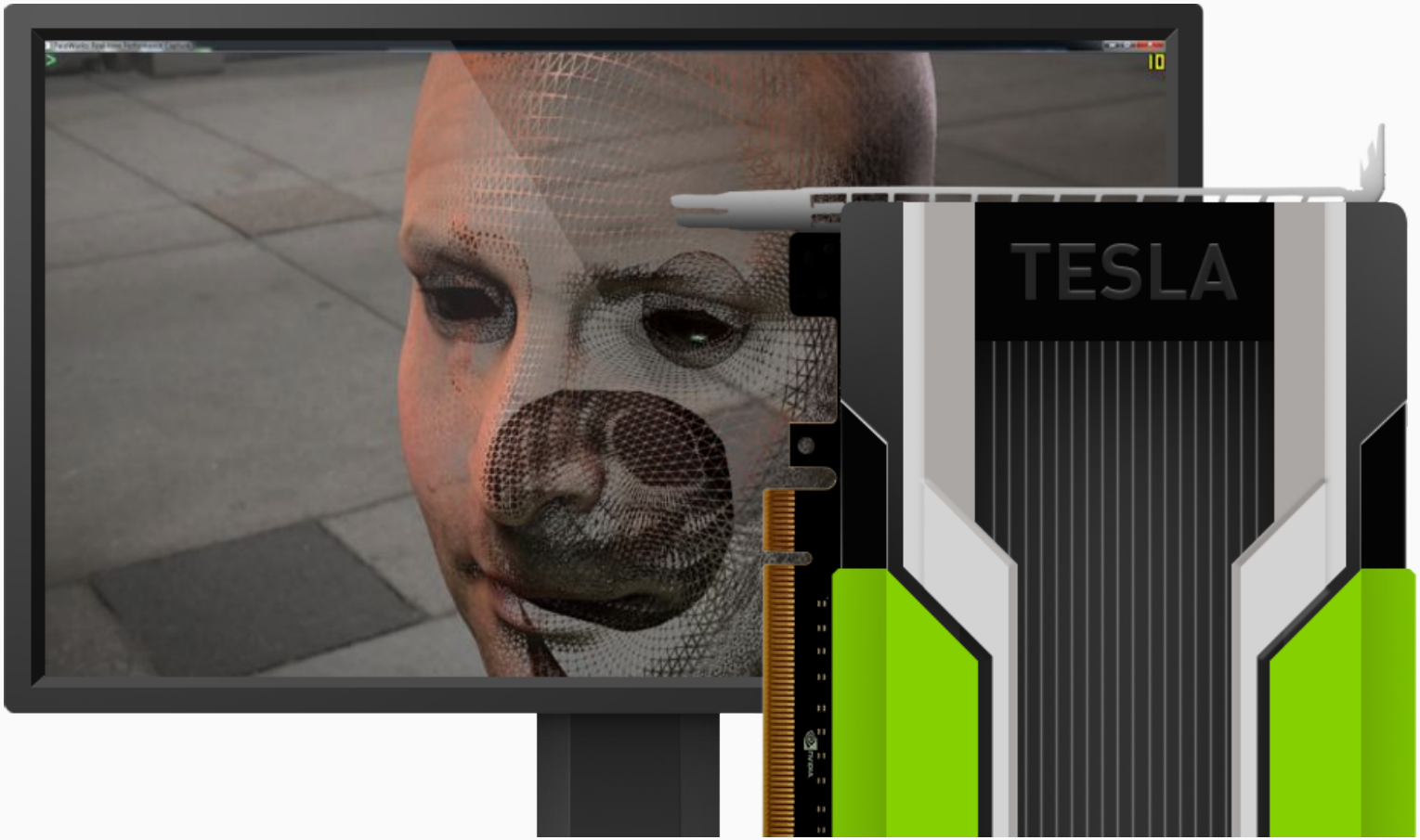
Sólo unir más y más computadores . . .

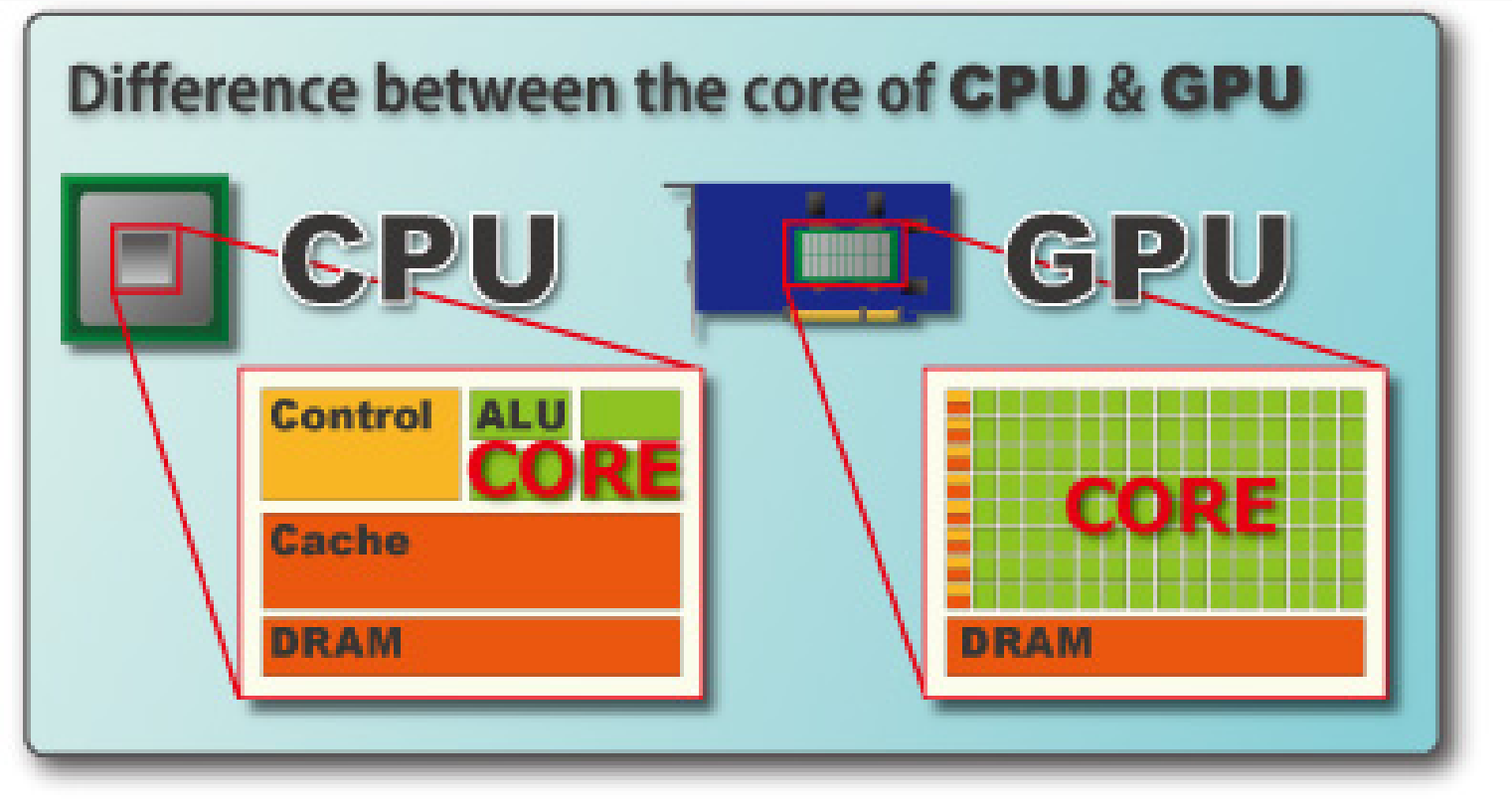
¿?

GPU Computing

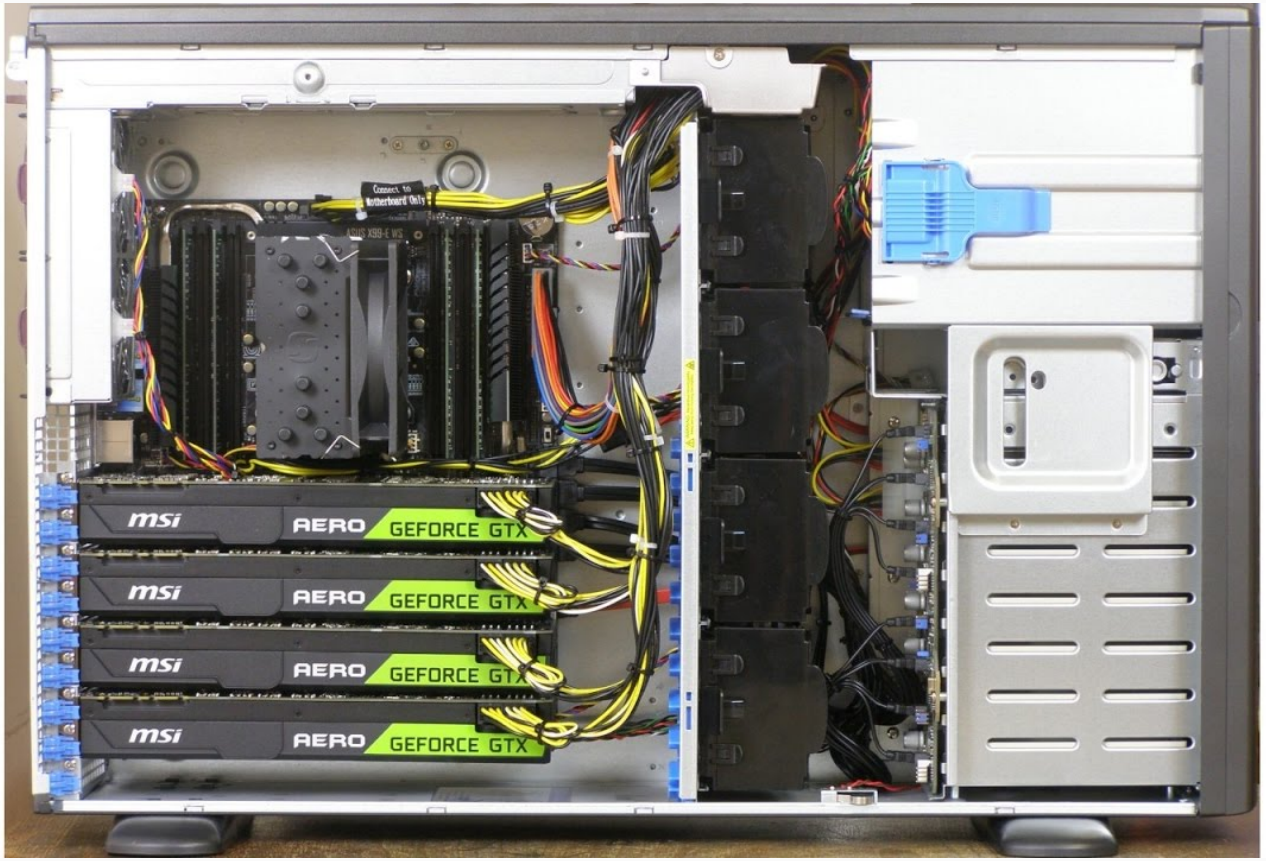
Graphics Processing Units

Graphics Processing Units




Difference between the core of CPU & GPU
CPU
GPU
Control
ALU
CORE
Cache
DRAM
CORE
DRAM

Graphics Processing Units



Graphics Processing Units

- Si bien es muy rentable (hasta 200X más rápido, para casos especiales), requiere más herramientas de programación en distintos lenguajes.
- Una ventaja es su bajo costo, pero lleva también un alto consumo eléctrico.
- Podemos convertir un PC de escritorio, en un verdadero cluster de cómputo con una sola tarjeta GPU.
- No tiene tanta madurez como MPI.

Graphics Processing Units

- Recientemente se ha revelado su potencial como arquitectura de cálculo para computación en general (GPGPU, General Purpose GPU) y por sobre todo para computación científica!
- Con las capacidades de memoria y velocidad de cómputo de las recientes generaciones de GPU, es posible conseguir ganancias en eficiencia de un orden de magnitud o más con respecto a los tiempos de CPU.
- Usos de GPU en computación científica
  - Simulaciones de elementos finitos.
  - Simulaciones Monte Carlo.
  - Dinámica Molecular
  - Calculos de Estructura Electrónica
  - Y muchos más.

MPI4PY

MPI: Message Passing Interface

- MPI : Message Passing Interface
  - Es una de las librerías principales para trabajar en paralelo con procesadores (CPU).
  - Entre sus implementaciones más utilizadas se encuentra OpenMPI e IntelMPI
  - Tienen ambas un gran soporte, tanto por la comunidad como por la empresa.
- Python posee variadas **interfaces** para el uso de MPI. Una de ellas es *MPI4PY*, que utilizaremos hoy.
  - Para comenzar, debe instalar el paquete python3-mpi4py y libopenmpi-dev
  - Al instalar python3-mpi4py se deberán instalar todos los paquetes necesarios para el uso de MPI.
  - Si el computador NO POSEE más de un CORE (o hilo de CPU) entonces el uso de MPI no será de gran utilidad.
  - Es posible que no vea mejoras, si utiliza una máquina virtual, configurada con sólo 1 CPU.
  - Lo más rentable es utilizar una máquina con múltiples cores reales para una verdadera alta performance.

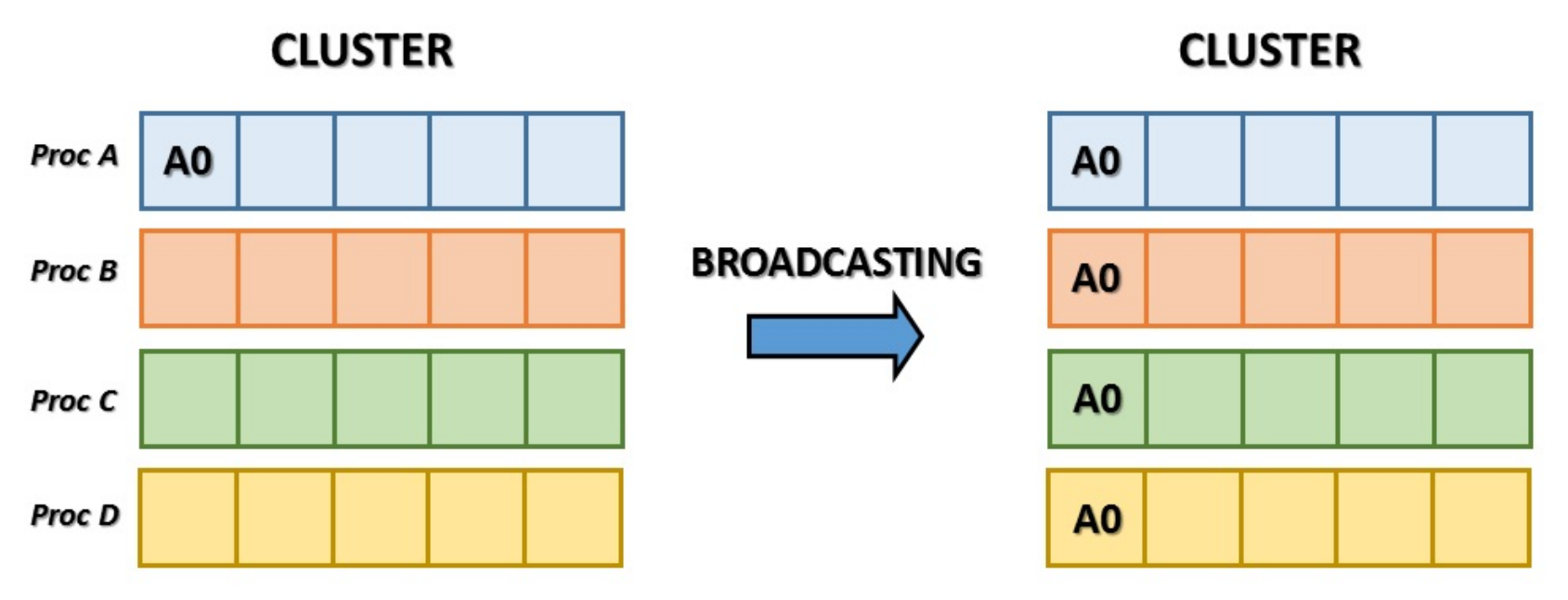
MPI4PY Hola Mundo

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
print(f'Hola Mundo!, Mi rank es: {rank}')
```

Ejecutamos el codigo con :

```
mpirun -np 2 python3 codigo.py
```

BROADCASTING



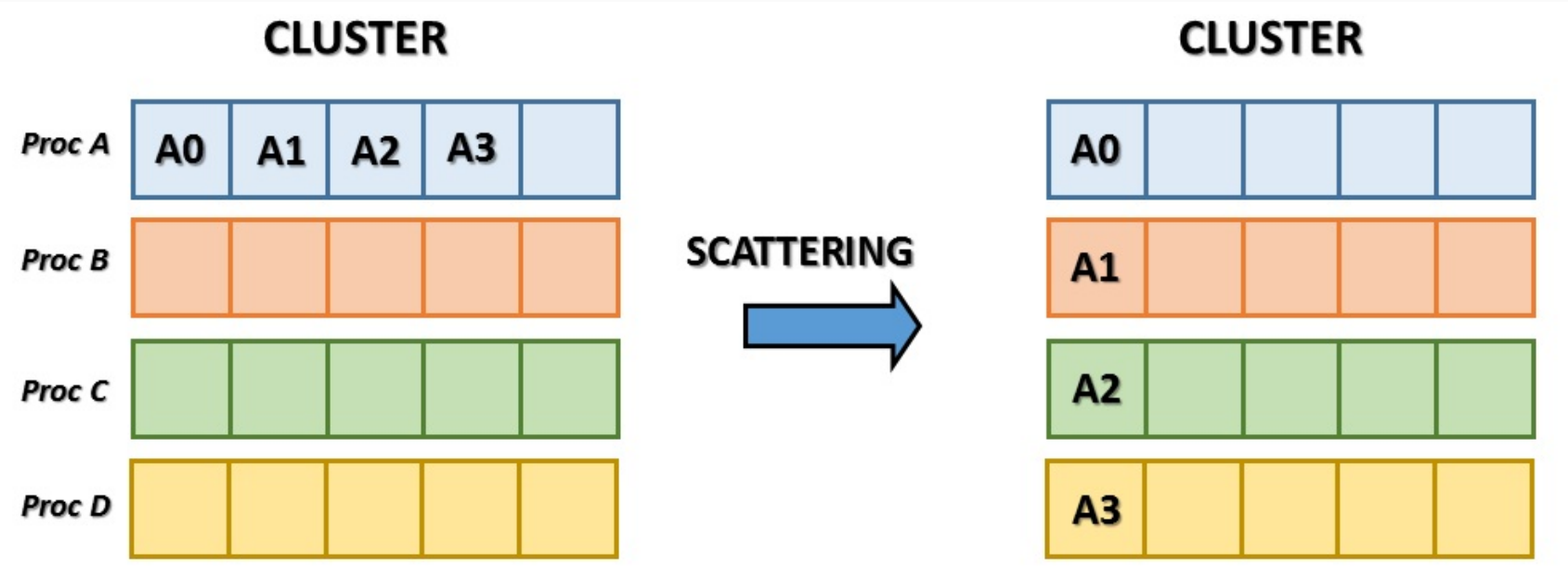
BROADCASTING

```python
from mpi4py import MPI

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

if rank == 0:
    data = {'key1' : [1,2, 3],
            'key2' : ( 'abc', 'xyz')}
else:
    data     = None

d = comm.bcast(data, root=0)
print(f'Rank: {rank}, data: {d}')
```

Ejecutamos el codigo con :

mpirun -np 2 python3 codigo.py

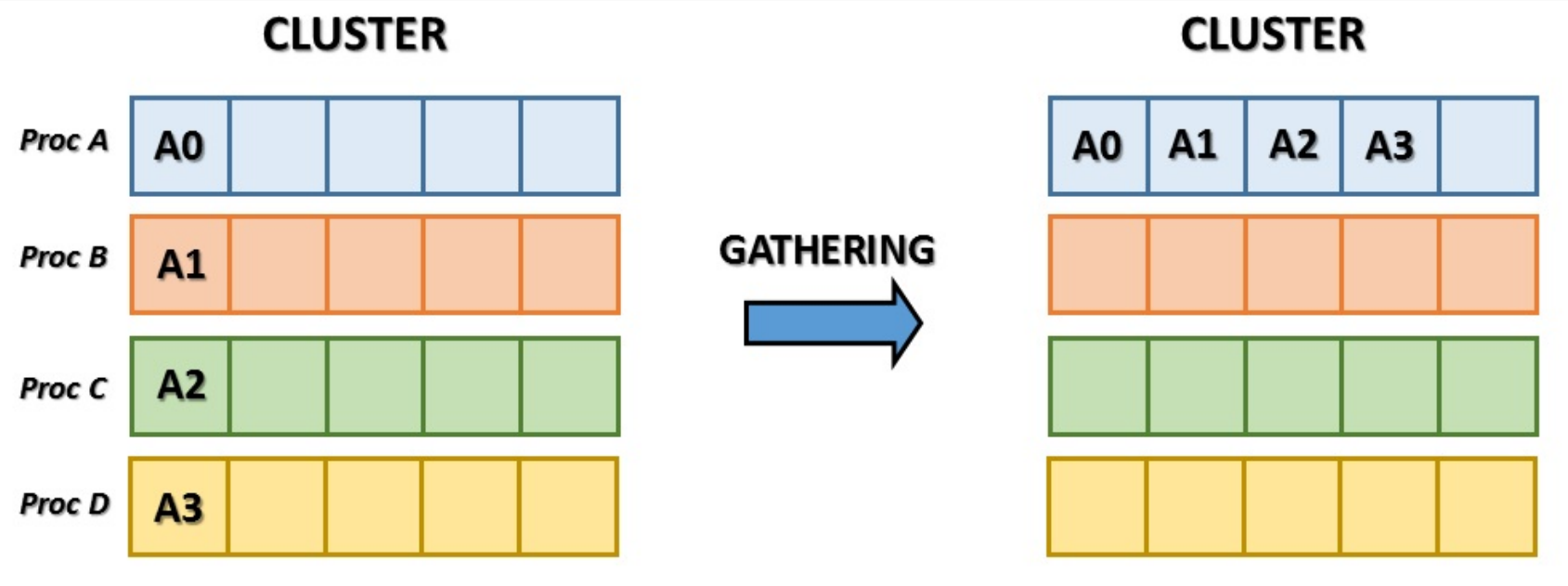
SCATTER


CLUSTER
Proc A
A0
A1
A2
A3
Proc B
Proc C
Proc D
SCATTERING
CLUSTER
A0
A1
A2
A3

SCATTER

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

comm = MPI.COMM_WORLD
size = comm.Get_size() # new: gives number of ranks in comm
rank = comm.Get_rank()

numDataPerRank = 10
data = None
if rank == 0:
    data = np.linspace(1,size*numDataPerRank,numDataPerRank*size)
    # when size=4 (using -n 4), data = [1. 2. 3. ... 38. 39. 40.]

recvbuf = np.empty(numDataPerRank, dtype='d') # allocate for recvbuf
comm.Scatter(data, recvbuf, root=0)

print(f'Rank: {rank}, recvbuf received: {recvbuf}')
```

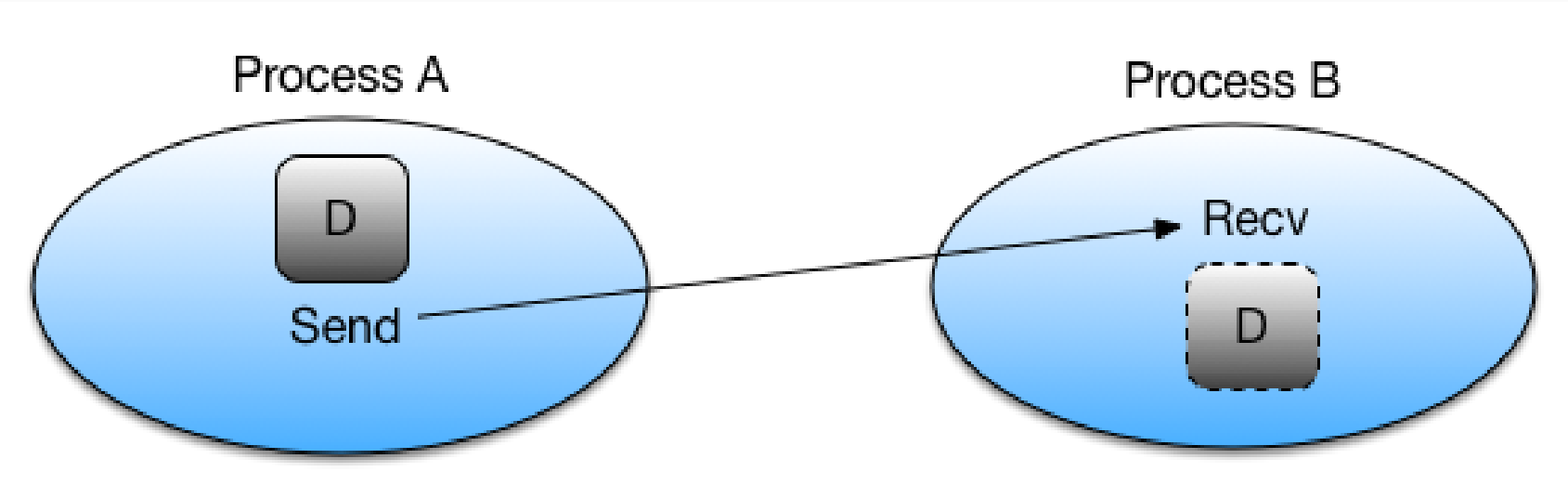
GATHER


CLUSTER
Proc A
A0
Proc B
A1
Proc C
A2
Proc D
A3
GATHERING
CLUSTER
A0
A1
A2
A3

GATHER

```
1  from mpi4py import MPI
2  import numpy as np
3
4  comm = MPI.COMM_WORLD
5  size, rank = comm.Get_size(), comm.Get_rank()
6
7  numDataPerRank = 10
8  sendbuf = np.linspace(rank*numDataPerRank+1,(rank+1)*numDataPerRank\\
9   ,numDataPerRank)
10 print(f'Rank: {rank}, sendbuf: {sendbuf}')
11
12 recvbuf = None
13 if rank == 0:
14     recvbuf = np.empty(numDataPerRank*size, dtype='d')
15
16 comm.Gather(sendbuf, recvbuf, root=0)
17 print(f'Rank: {rank}, recvbuf received: {recvbuf}')
```

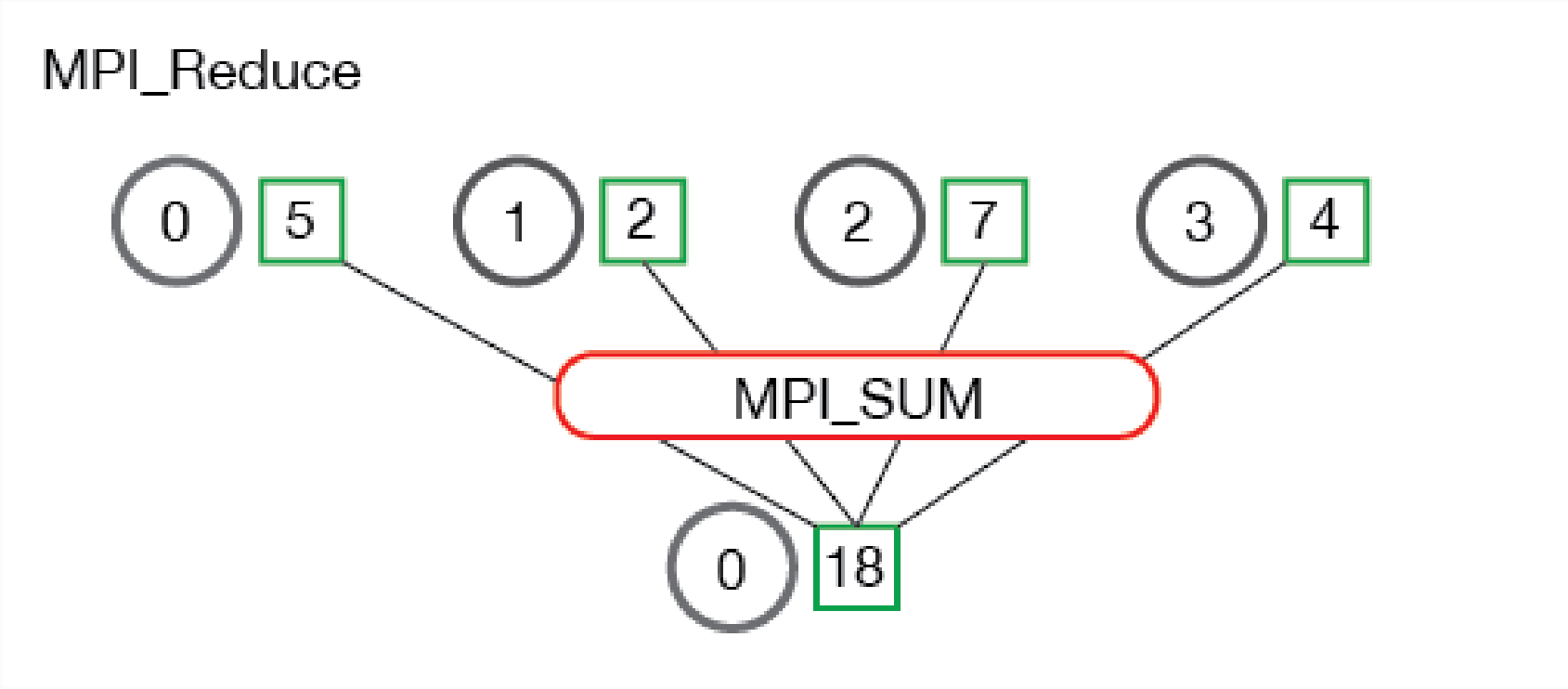
SEND/RECEIVE



SEND/RECEIVE

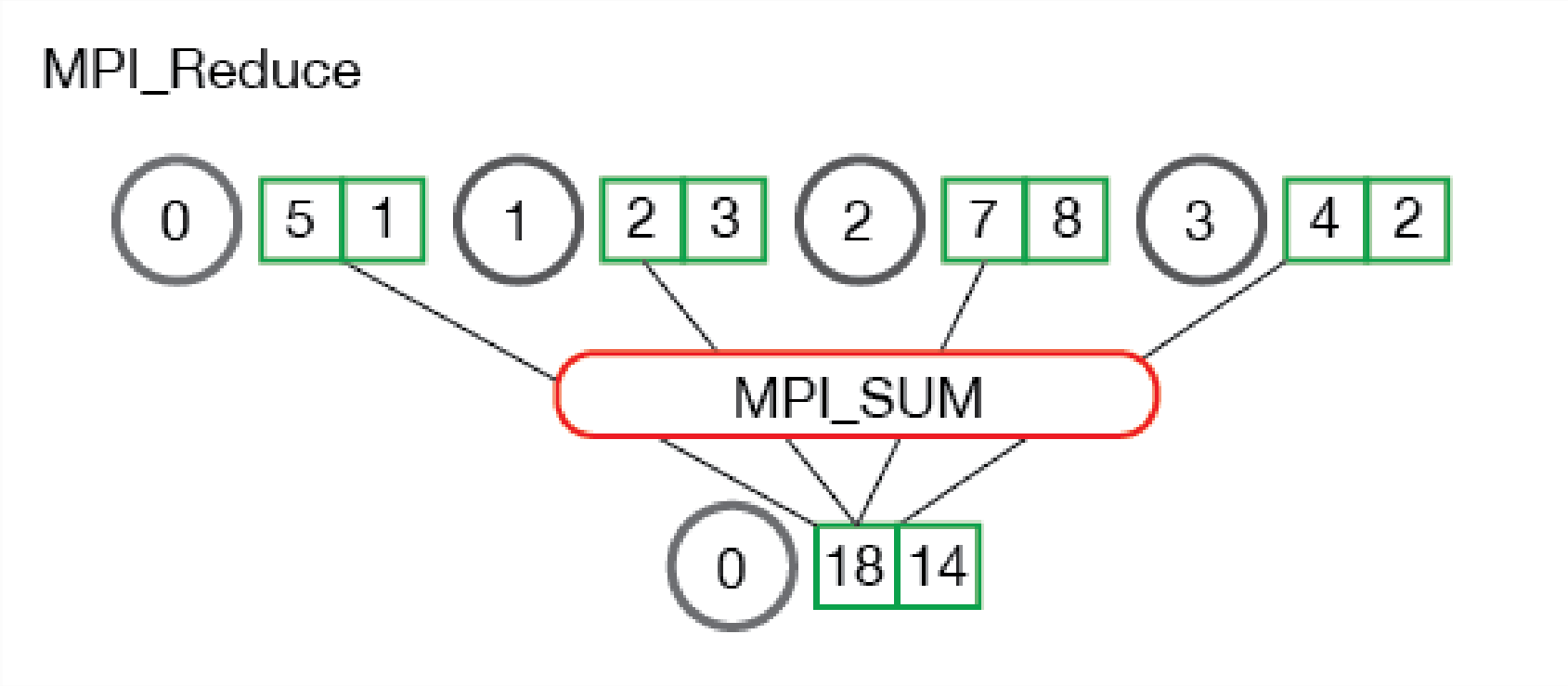
```python
from mpi4py import MPI
import numpy as np

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

if rank == 0:
    # in real code, this section might read data from a file
    numData = 10
    comm.send(numData, dest=1)
    data = np.linspace(0.0,3.14,numData)
    comm.Send(data, dest=1)

elif rank == 1:
    numData = comm.recv(source=0)
    print(f'Number of data to receive: {numData}')
    data = np.empty(numData, dtype='d')  # allocate space
    comm.Recv(data, source=0)

    print(f'data received: {data}')
```

REDUCE



send-recv

REDUCE



REDUCE

```
1  from mpi4py import MPI
2  import numpy as np
3
4  comm, rank = MPI.COMM_WORLD, comm.Get_rank()
5  # Create np arrays on each process: assigned to be the rank
6  value = np.array(rank,'d')
7  print(f' Rank: {rank} value = {value}')
8
9  # initialize the np arrays that will store the results:
10 value_sum, value_max      = np.array(0.0,'d'), np.array(0.0,'d')
11
12 # perform the reductions:
13 comm.Reduce(value, value_sum, op=MPI.SUM, root=0)
14 comm.Reduce(value, value_max, op=MPI.MAX, root=0)
15
16 if rank == 0:
17     print(f' Rank 0: value_sum = {value_sum}')
18     print(f' Rank 0: value_max = {value_max}')
```

```python
from mpi4py import MPI
comm, rank, size = MPI.COMM_WORLD, comm.Get_rank(), comm.Get_size()
time = MPI.Wtime()

N = 10 #luego cambiamos a 1E6
lst = list(range(N))
split_lst = [lst[_i::size] for _i in range(size)]
my_lst = comm.scatter(split_lst)
print (f'La lista del rank {rank} es {my_lst}')

#cada rank hace su trabajo
suma = 0.0
for k in my_lst:
    suma = suma + k
print (f'Rank {rank} sumo {suma} en {MPI.Wtime()-time} segundos.')
total = comm.gather(suma)
if rank == 0:
    print (f'La suma total es de {sum(total)}')
```

Más información

https://info.gwdg.de/dokuwiki/doku.php?id=en:services:application_services:high_performance_computing:courses

Actividades

Actividades

(a) Haga un gráfico de tiempo para la suma de números enteros entre 1000 y : ”100000, 1000000, 10000000”. Grabe su trabajo, usar números tan grandes podrían hacer colapsar la RAM del equipo. Usando el **máximo** de procesadores disponibles. Para eso use el código **sumando** presentado en clases, pero cambie la línea 14:
`suma = suma + k` $\longrightarrow$ `suma +`
`exp(-(k/(suma+1))**2)/log(1/(k+1)/log(1+exp(-suma)))`

- ¿Cómo se comporta el gráfico?
- Explique el comportamiento.
- Repita el proceso usando **un procesador**

(b) Escriba un programa en Python3 usando *mpi4py* para un gran set de números (por ejemplo de 1 hasta 1e6). Determine cuáles son números primos.