Distributed Data Analytics Lab

Exercise Sheet 1

Sebastián Pineda Arango Matrikel-Nr: 246098

Topic Distributed Computing with message Passing Interface (MPI)

Exercise 0: System

Characteristic	Description
Processor	I7-4510U CPU @ 2.6 GHz
Number of cores	2
Number of virtual cores	4
RAM	8 Gb
OS	Windows 10
Programming language version	Python 3.6.3

Exercise 1: Basic Parallel Vector Operations with MPI

Part a. Addition of vectors

- **Problem formulation:**

We want to perform the following mathematical operation

$$\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$$

Where $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in R^N$ (it is, the vectors have length N).

- **Distributed reformulation of the problem**

If we have $\boldsymbol{M}$ workers, which can compute using some part of the vectors then we can divide the vector $\boldsymbol{a}$ and $\boldsymbol{b}$ in M parts, such as:

$$\boldsymbol{a}^{(j)}, \boldsymbol{j} = \boldsymbol{1} \ldots \boldsymbol{M}$$

$$\boldsymbol{b}^{(j)}, \boldsymbol{j} = \boldsymbol{1} \ldots \boldsymbol{M}$$

Where $\boldsymbol{a}^{(j)}$ is the fraction of the vector $\boldsymbol{a}$ that the worker $\boldsymbol{j}$ receives and $\boldsymbol{b}^{(j)}$ is the fraction of the vector $\boldsymbol{b}$ that the worker $\boldsymbol{j}$ receives. These fractions can be calculated in the following way:

$$\boldsymbol{a}^{(j)} = \left(a_{(j-1)\cdot d}, a_{(j-1)\cdot d+1}, \dots, a_{j\cdot d}\right), j = 1 \dots M-1$$

$$\boldsymbol{a}^{(j)} = \left(a_{(M-1)\cdot d}, \dots, a_N\right), j = M$$

Where$d = \lfloor N/M \rfloor$. The same distribution fractions are assigned for the vector $\boldsymbol{b}$.

$$\boldsymbol{b}^{(j)} = \left(b_{(j-1)\cdot d}, b_{(j-1)\cdot d+1}, \dots, b_{j\cdot d}\right), j = 1 \dots M-1$$

$$\boldsymbol{b}^{(j)} = \left(b_{(M-1)\cdot d}, \dots, b_N\right), j = M$$

The data fractions are computed and sent by the **master** to every worker using MPI. Once each worker receives the data, they can compute the output as:

$$\boldsymbol{c}^{(j)} = \boldsymbol{a}^{(j)} + \boldsymbol{b}^{(j)}$$

Afterwards, every worker sends back the data to the master which can concatenate the output to obtain the final value for $\boldsymbol{c}$.

$$\boldsymbol{c} = [\boldsymbol{c}^1, \dots, \boldsymbol{c}^M]$$

Both in the notation and in the implemented program, the master receives the data for $j = M$.

We represent graphically the way the data is distributed in the figure 1, where we use *N*=6 and *M*=3. Here we show four different parts:

1) The master generates the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$.
2) The master segments the vectors and se ndseach fraction to the respective worker.
3) Each worker receives the data (two vectors segments) and add them. The result is sent back to the master.
4) The master receives the data and concatenates to obtain the final result.

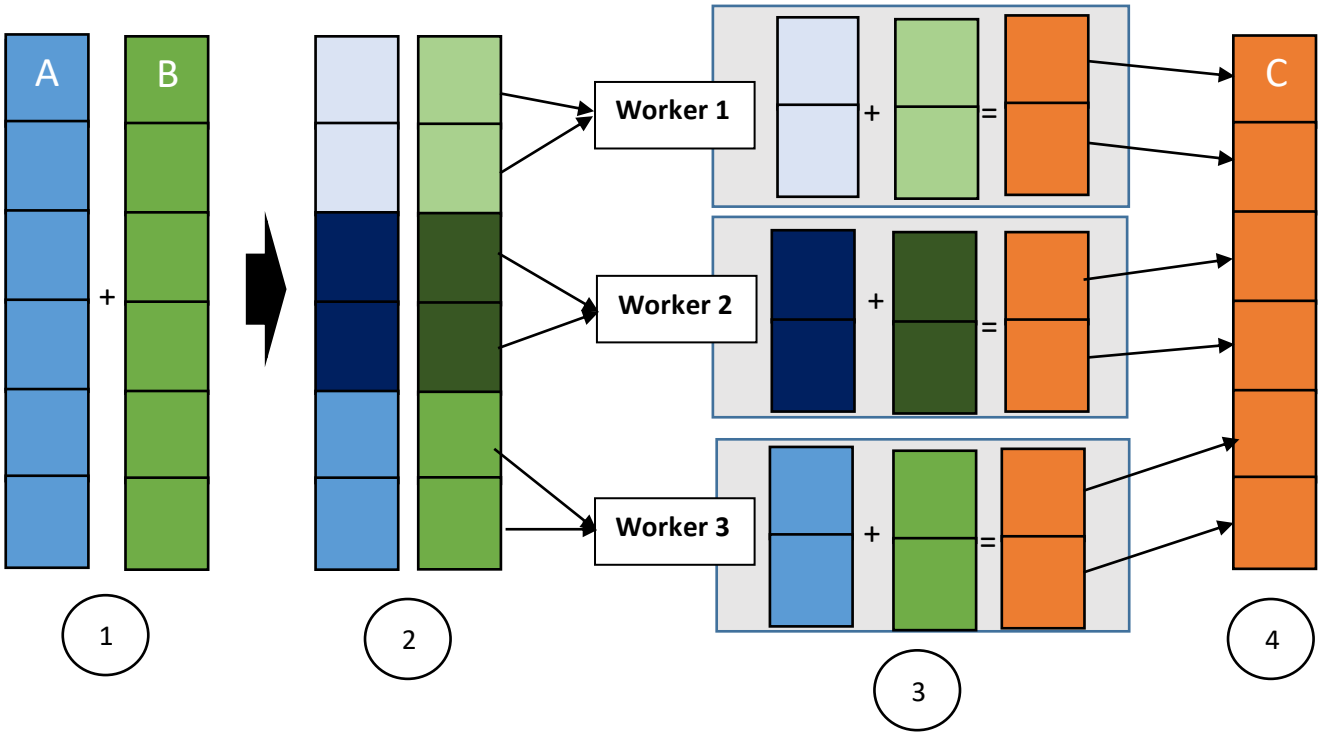


Figura 1. Strategy to divide data between workers for vector addition

- **Amount of information exchanged:**

Given that the master sends two vectors of size N and the workers together sends also two vectors of size N, then the amount of data exchange is in the order of **4N**. If every element is 1 Byte, then the information amount would be at least 4N bytes (plus some protocol information).

- **Amount of operations in sequential an parallel settings:**

In the sequential setting, N additions will take place. In parallel setting, at least each worker will perform **N/M** additions (M is the number of workers).

- **Code**

To implement the above mentioned strategy, we use python and the library MPI4py. We can observe the process in the code as follows:

- *Importing libraries and initializing important variables*

```
from mpi4py import MPI
import numpy as np
from random import random
import time
import sys

#initializing important variables
comm = MPI.COMM_WORLD
num_workers = comm.Get_size()
worker = comm.Get_rank()
N= int(sys.argv[1])
```

- *Computations in the worker*: the worker receives information from master as a python dictionary. Then it sends back the addition of the received vectors. The workers also send back the ID (rank) so that the output can be reconstructed in the correct order by the master.

```
if worker !=0:

    #receiving data from master
    data = comm.recv()
    V1_parallel = data['V1']
    V2_parallel = data['V2']

    #adding received vectors and sendind back to master
    comm.send({'SUM' : V1_parallel+ V2_parallel,
               'owner' : worker}, dest=0)
```

- *Computations in the master (first part)*: the master creates two random vectors and send to every worker a segment of these vectors (as explained before).

```
else:

    #vector of random numbers between 0 and 10
    V1 = np.random.randint(1,10,N).astype(int)
    V2 = np.random.randint(1,10,N).astype(int)
    V3 = np.zeros(N)

    #fraction of data for each worker
    data_frac = int(N/(num_workers))

    i=0
    #sending data to workers
    for i in range(1,num_workers):
        data  = {'V1': V1[data_frac*(i-1):data_frac*i],
                 'V2': V2[data_frac*(i-1):data_frac*i]}
        comm.send(data, dest=i)
```

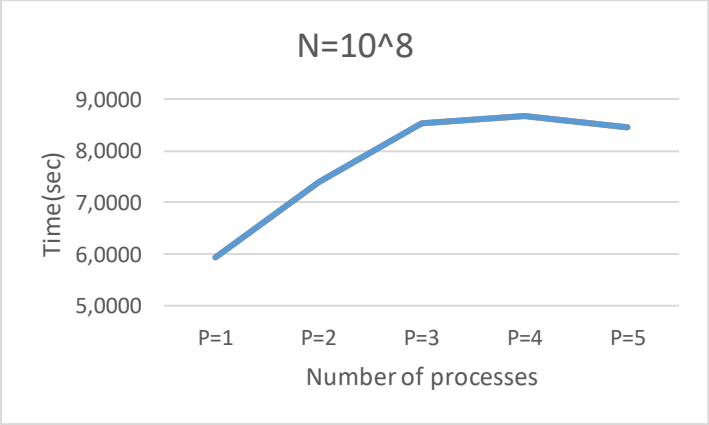
- *Computations in the master (second part):* after sending the data to workers, the master compute the partial outputs with the segment it kept. Then, it receives the output sent by the workers an, finally, concatenates them for a final result.

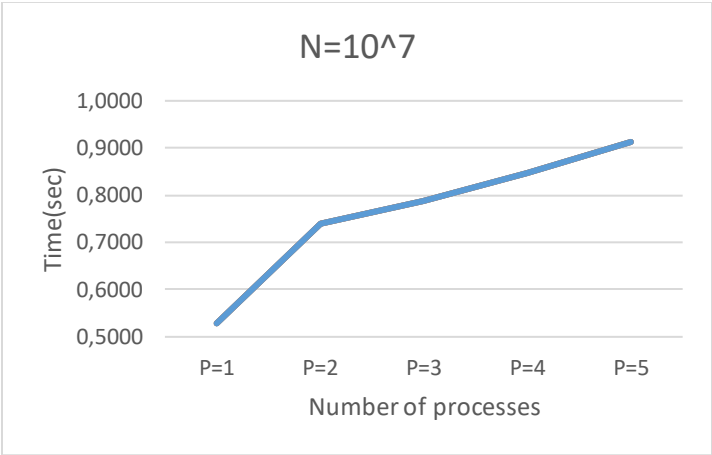
```
#data that the master keeps
V1_parallel = V1[data_frac*i:]
V2_parallel = V2[data_frac*i:]

V3[data_frac*i:] = V1_parallel + V2_parallel

#receiving and merging information from workers
for i in range(1,num_workers):
    data = comm.recv()
    j = data['owner']
    V3[data_frac*(j-1):data_frac*j]=data['SUM']
```

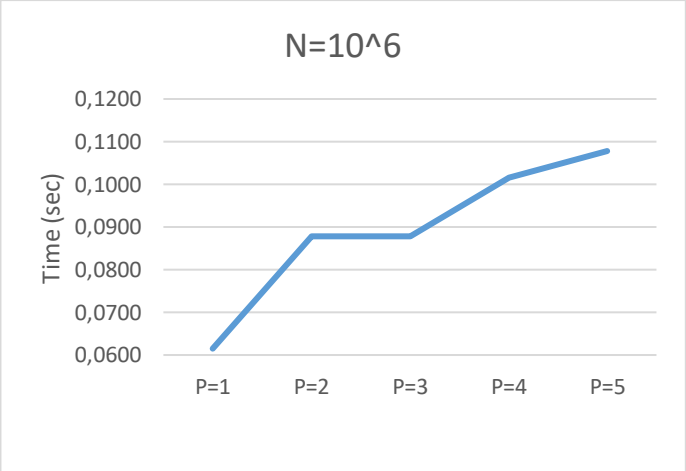
- **Results**

We ran experiments for different values of N and for different numbers of processes. Each configuration was run five times and we tabulate and plot the mean of the execution time among these experiments. We can see that when we increase the number of processes, there are an increment on the processing time for all the vector sizes. This is explained due to the communication **overload**, which actually increase the time. However, the increment is less strong for a really big vector (10e-8). At this moment, the communication overload starts to be balanced by the computation time gained through the multi-process task. Since the computer has only two cores, it is not possible to see how it would behave with more process.

Summary Mean Computation Time			
N. of Processes	**N=10^8**	**N=10^7**	**N=10^6**
P=1	5,9340	0,5286	0,0615
P=2	7,4069	0,7411	0,0879
P=3	8,5340	0,7869	0,0877
P=4	8,6709	0,8485	0,1015
P=5	8,4427	0,9129	0,1078







Part b. Average of numbers in a vector

- **Problem formulation**

We want to perform the following mathematical operation:

$$average(\boldsymbol{a}) = \left(\frac{1}{N}\right)\sum_{\boldsymbol{i=0}}^{\boldsymbol{N}} a_j$$

Where $\boldsymbol{a} \in R^N$ (it is, the vector has length N).

- **Distributed reformulation of the problem**

If we have ***M*** workers, which can compute using some part of the vectors then we can divide the vector $\boldsymbol{a}$ in M parts, such as:

$$\boldsymbol{a^{(j)}, j = 1 \dots M}$$

Where $\boldsymbol{a^{(j)}}$ is the fraction of the vector $\boldsymbol{a}$ that the worker $\boldsymbol{j}$ receives. These fractions can be calculated in the following way:

$$\boldsymbol{a^{(j)}} = \left(a_{(j-1)\cdot d}, a_{(j-1)\cdot d+1}, \dots, a_{j\cdot d}\right), j = 1 \dots M-1$$

$$\boldsymbol{a^{(j)}} = \left(a_{(M-1)\cdot d}, \dots, a_N\right), j = M$$

The data fractions are computed and sent by the **master** to every worker using MPI point-to-point communication. Once each worker receives the data, they can compute the output as:

$$\boldsymbol{sum^{(j)}} = \boldsymbol{sum}(\boldsymbol{a^{(j)}})$$

It is, the summation of all the components of the vector $\boldsymbol{a^{(j)}}$. Afterwards, every worker sends back the data to the master which can compute the final result as:

$$average(\boldsymbol{a}) = \sum_{\boldsymbol{j=1}}^{\boldsymbol{M}} \boldsymbol{sum^{(j)}}$$

Both in the notation and in the implemented program, the master keeps the data for $j = M$.

We represent graphically the way the data is distributed in the figure 1, where we use *N*=6 and *M*=3. Here we show four different parts:

1) The master generates the vector $\boldsymbol{a}$.
2) The master segments the vectors and sends each fraction to the respective worker.
3) Each worker receives the data (vector segments) and add them. The result is sent back to the master.
4) The master receives the partial summatory, add them and divide by the size of the original vector to obtain the average.

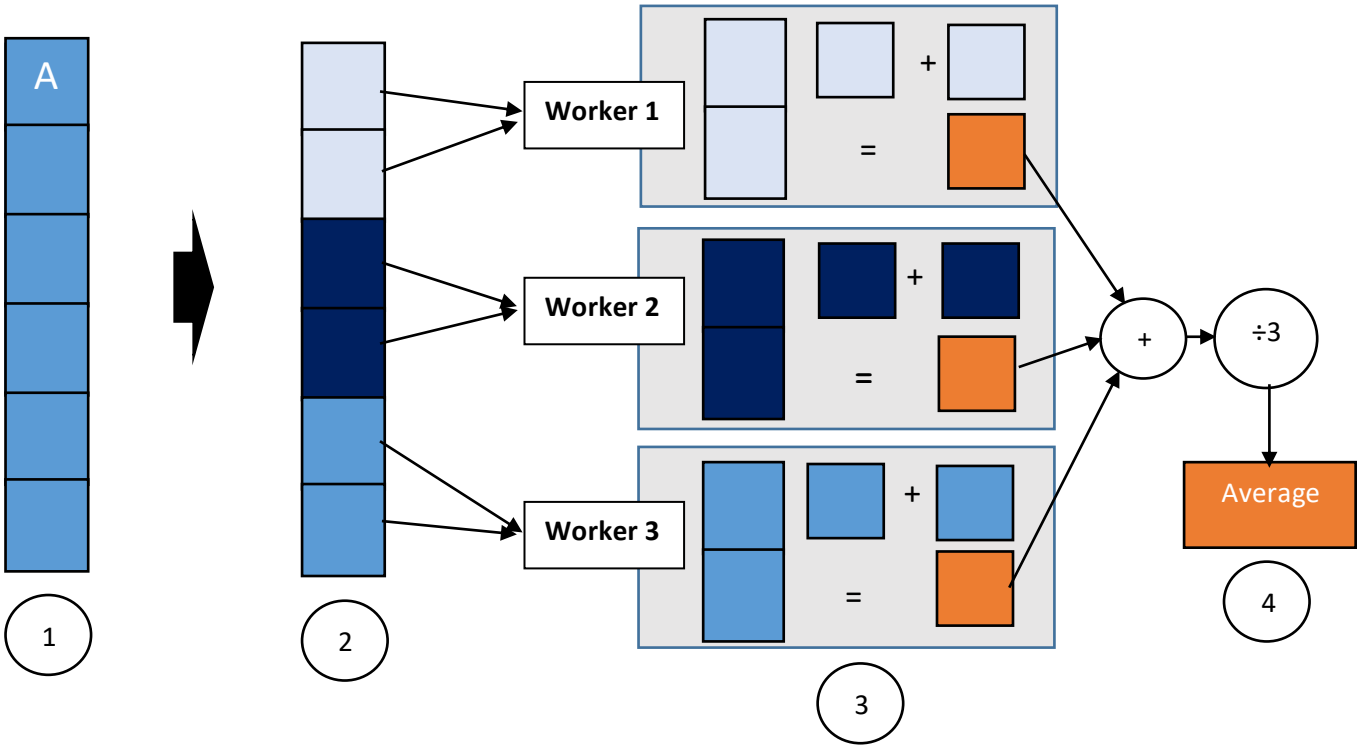


Figura 2. Strategy to divide data between workers for vector mean calculation

- **Amount of information exchanged:**

Given that the master sends a vector of size N and the workers together sends sent back only a number (M values), then the amount of data exchange is in the order of **N+M**. If every element is 1 Byte, then the information amount would be at least (**N+M**) bytes (plus some protocol information).

- **Amount of operations in sequential an parallel settings:**

In the sequential setting, N additions will take place. In parallel setting, at least each worker will perform **N/M** additions (M is the number of workers), and the master must perform **M** additional auditions.

- **Code**

To implement the above mentioned strategy, we use python and the library MPI4py. We can observe the process in the code as follows:

- *Importing libraries and initializing important libraries*

```
#importing libraries
from mpi4py import MPI
import numpy as np
from random import random
import time
import sys

#initializing important variables
comm = MPI.COMM_WORLD
num_workers = comm.Get_size()
worker = comm.Get_rank()
N= int(sys.argv[1])
```

- *Computations in the worker:* the worker receives information from master as a python dictionary (using ***comm.recv()***). Then it sends back the addition of the elements of the received vector (with **comm.send()**). Note that the worker send back its ID (rank) so that the data can be collected without error.

```
if worker !=0:
    #receiving data from master
    data = comm.recv()
    V_parallel = data['V']
    sum_V = sum(V_parallel)
    comm.send({'SUM':sum_V,
               'owner': worker}, dest=0)
```

- *Computations in the master (first part):* the master creates a random vector and send to every worker a segment of this vector (as explained before – using **comm.send()**).

```
else:
    #vector of random numbers between 1 and 10
    V = np.random.randint(2,5,N).astype(int)

    #fraction of data for each worker
    data_frac = int(N/(num_workers))

    i=0

    #sending data to workers
    for i in range(1,num_workers):
        data  = {'V': V[data_frac*(i-1):data_frac*i]}
        comm.send(data, dest=i)
```

- *Computations in the master (second part)*: after sending the data to workers, the master compute the partial sum of the components with the segment it kept. Then, it receives the output sent by the workers (***comm_recv()***) and, finally, adds these outputs using an accumulator (*sum_V)* to get a final result.

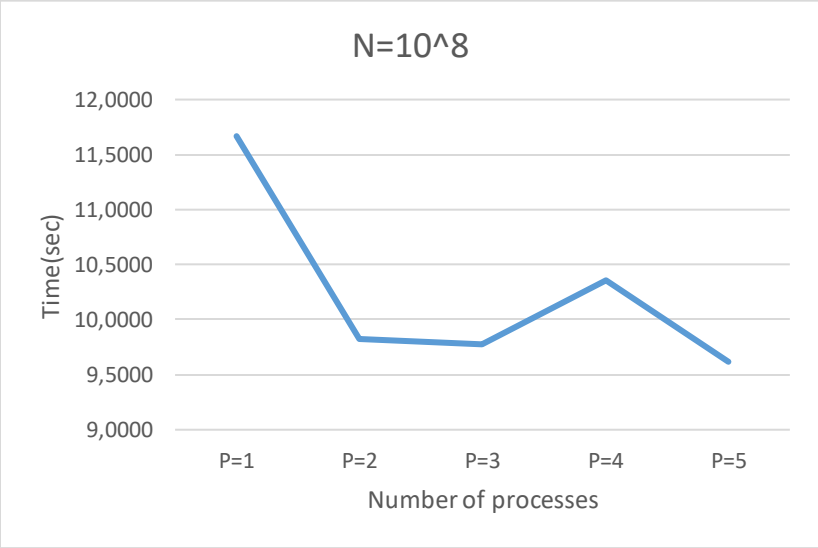
```
#sum of vector part in master
sum_V = sum(V[data_frac*i:])

#receiving sums from workers and adding to the accumulator
for i in range(1,num_workers):
    data = comm.recv()
    sum_V += data['SUM']
```

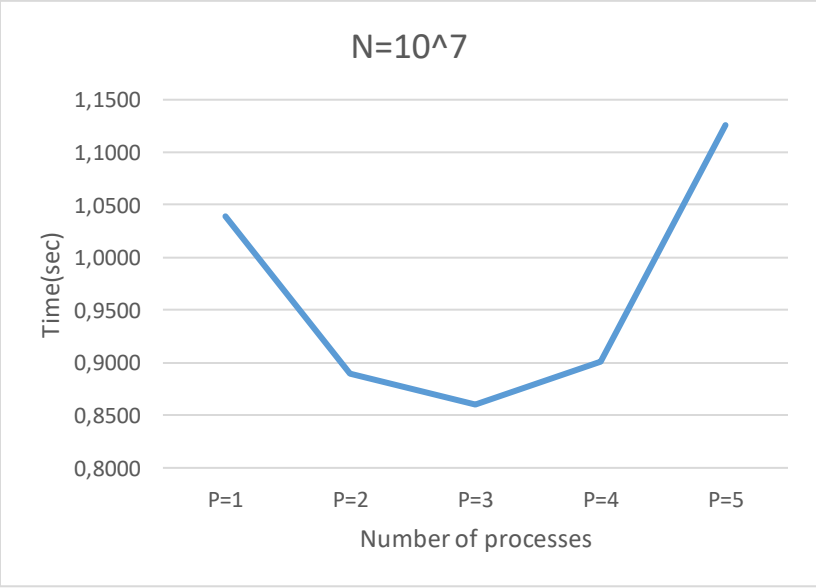
- **Results**

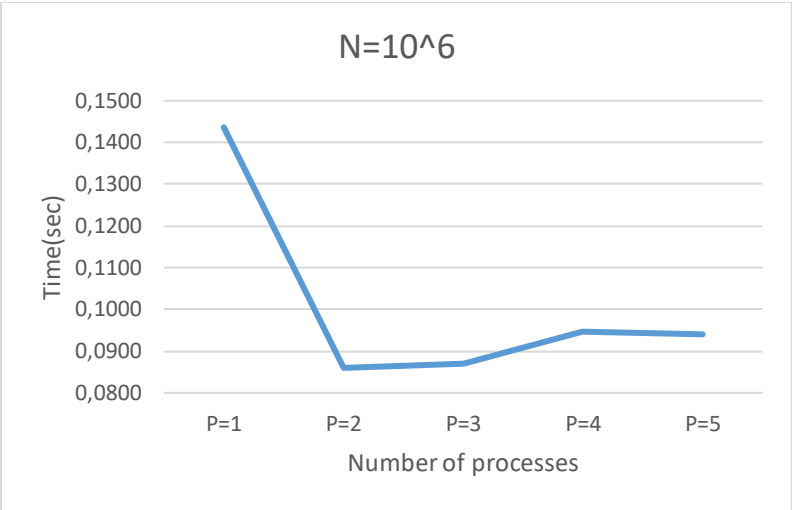
We ran experiments for different values of N and for different numbers of processes. Each configuration was run five times and we tabulate and plot the mean of the execution time among these experiments. In contrast to the vector addition, increasing the number of process for finding the average of the vector decrease the computation time. In this case, the amount of exchanged information is less and therefore the communication overload is

not so harmful. For N=10e-7, however, having more than 4 processes may not be so beneficial since the time gained through parallel computing does not compensate the communication overload.

Summary Mean Computation Time			
N. of Processes	**N=10^8**	**N=10^7**	**N=10^6**
P=1	11,6674	1,0392	0,1437
P=2	9,8194	0,8899	0,0860
P=3	9,7783	0,8602	0,0871
P=4	10,3580	0,9007	0,0948
P=5	9,6174	1,1259	0,0940







Exercise 2: Parallel Matrix Vector Multiplication using MPI

- **Problem formulation:**

We want to perform the following mathematical operation

$$\boldsymbol{c} = A\boldsymbol{b}$$

Where $\boldsymbol{b}, \boldsymbol{c} \in R^N$, $A \in R^{N \times N}$ (it is, two vectors that have length N, and square matrix with size $N \times N$).

- **Distributed reformulation of the problem**

If we have ***M*** workers, which can compute using some part of the vectors then we can divide the matrix A in M parts, such as:

$$A^{(j)}, j = 1 \dots M,$$

Where $A^{(j)}$ is the fraction of the matrix A that the worker ***j*** receives and. These fractions can be calculated in the following way:

$$A^{(\boldsymbol{j})} = \begin{pmatrix} A_{(j-1)\cdot d,1} & \cdots & A_{(j-1)\cdot d,N} \\ \vdots & \ddots & \vdots \\ A_{j\cdot d,1} & \cdots & A_{j\cdot d,N} \end{pmatrix}, \qquad if\ j = 1 \dots M-1$$

$$A^{(j)} = \begin{pmatrix} A_{(j-1)\cdot d,1} & \cdots & A_{(j-1)\cdot d,N} \\ \vdots & \ddots & \vdots \\ A_{N,1} & \cdots & A_{N,N} \end{pmatrix}, \qquad if\ j = M$$

Where $d = \lfloor N/M \rfloor$. The vector $\boldsymbol{b}$ is not fractionated. **The whole vector is sent to all workers.**

The data fractions are computed and sent by the **master** to every worker using MPI point to point communication. Once each worker receives the data, they can compute the output as a matrix-vector multiplication (assuming $\boldsymbol{b}$ is a column vector):

$$\boldsymbol{c}^{(j)} = A^{(j)}\boldsymbol{b}$$

Here we have the partial results of the final vector $\boldsymbol{c}$**.** Afterwards, every worker sends back the data to the master which can concatenate the output to obtain the final value for $\boldsymbol{c}$.

$$\boldsymbol{c} = \begin{pmatrix} \boldsymbol{c}^1 \\ \dots \\ \boldsymbol{c}^M \end{pmatrix}$$

Both in the notation and in the implemented program, the master receives the data for $j = M$.

We represent graphically the way the data is distributed in the figure 1, where we use *N*=3 and *M*=3. Here we show four different parts:

1) The master generates the vectors $\boldsymbol{b}$ and the matrix A and segments the matrix. Afterwards, it sends each fraction of the matrix and the whole vector to the respective worker.
2) Each worker receives the data (the vector and the matrix) and multiply them. The result is sent back to the master.
3) The master receives the data and concatenates to obtain the final result.

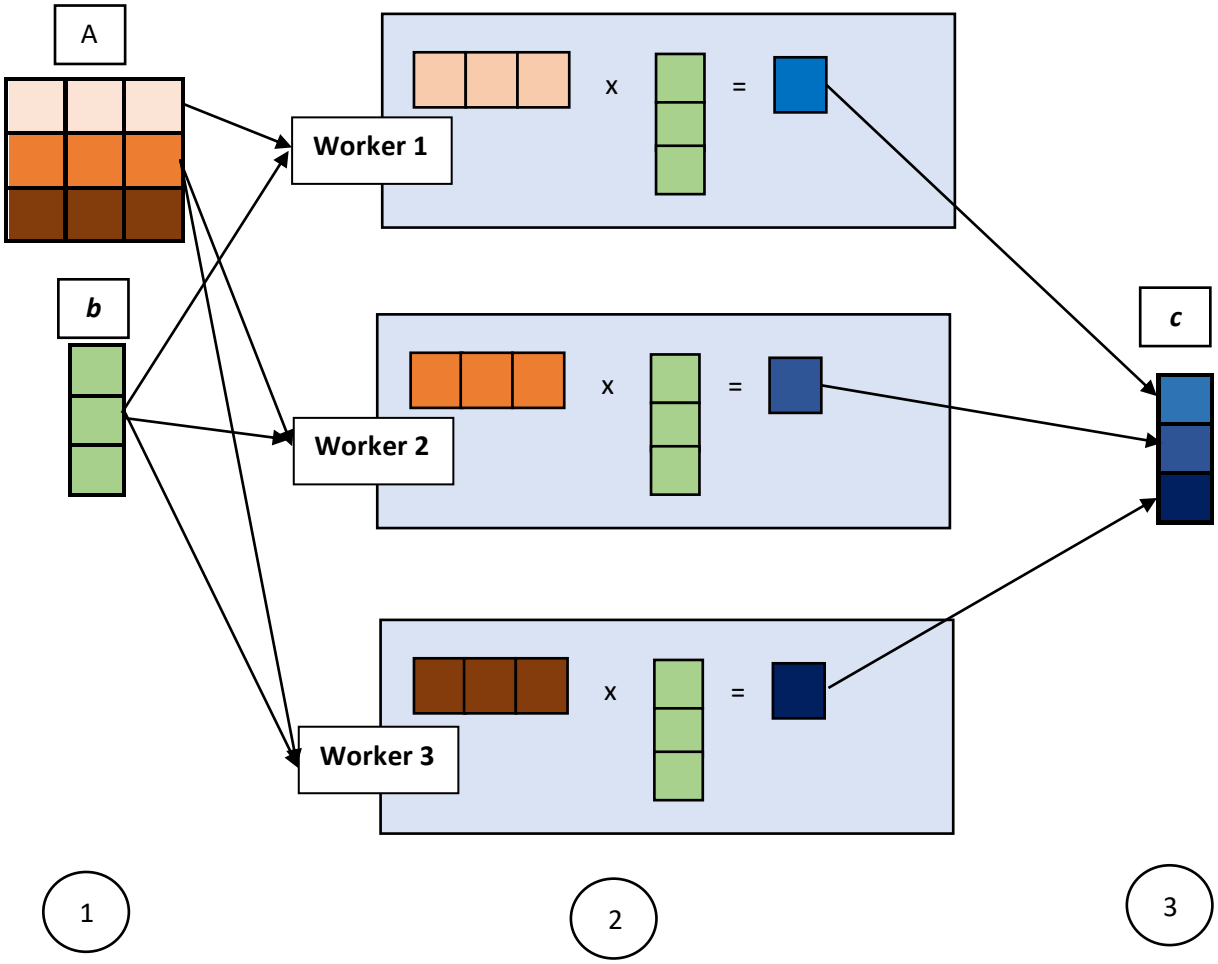


Figura 3. Strategy to divide data between workers for vector – matrix multiplication

- **Amount of information exchanged and amount of operations:**

Given that the master sends a vector of size N and matrix of size NXN. The workers send back together N values, therefore the amount of data exchanged is in the order of $2N + N^2$. On the other hand, the amount of operations for each worker is approximately N/M multiplications and N/M additions.

- **Code**

To implement the above mentioned strategy, we use python and the library MPI4py. We can observe the process in the code as follows:

- *Importing libraries and initializing important libraries*

```
# #importing libraries
from mpi4py import MPI
import numpy as np
from random import random
import time
import sys

#initializing important variables
comm = MPI.COMM_WORLD
num_workers = comm.Get_size()
worker = comm.Get_rank()
N= int(sys.argv[1])
```

- *Computations in the worker:* each worker receives information with ***comm.recv()*** (matrix fragment and vector) from master as a python dictionary. Then it sends back the multiplication of the matrix with the vector – using ***comm.send().***

```
if worker !=0:

    #receiving data from master
    data = comm.recv()
    A_parallel = data['A']
    b_parallel = data['b']

    #multiplying partial result
    c_partial = A_parallel@b_parallel

    #sending data back to master
    comm.send({'c_partial':c_partial,
               'owner':worker}, dest=0)
```

- *Computations in the master (first part):* the master first create the vector and the matrix (by random sampling). Then, it fragments the data and send it to the workers as a python dictionary using MPI point-to-point communication.

```
else:

    #vector of random numbers between 1 and 10
    A = np.random.randint(1,10,(N,N)).astype(int)
    b = np.random.uniform(1,10,N).astype(int)
    c = np.zeros(N)

    #fraction of data for each worker
    data_frac = int(N/(num_workers))

    i=0 #initializing variable to avoid crashing when P=1

    #sending data to workers
    for i in range(1,num_workers):
        data  = {'A': A[data_frac*(i-1):data_frac*i,],
                 'b': b}
        comm.send(data, dest=i)
```

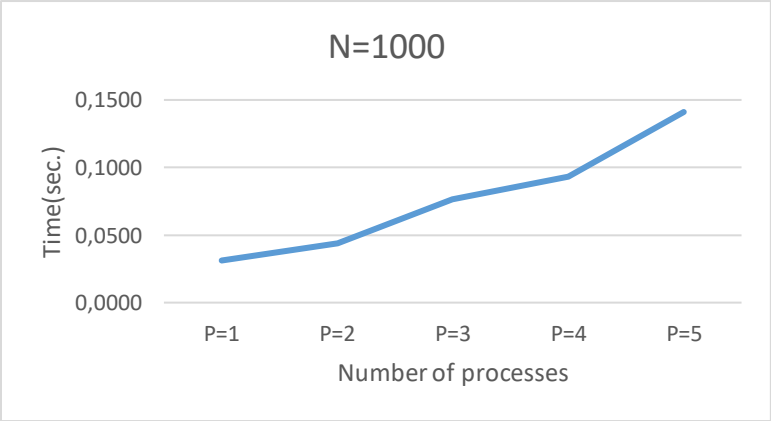
- *Computations in the master (second part):* the master computes with the data that he kept. Then he receives the output of the computations of the workers. The workers also send back their ID (rank) so that the output can be concatenate correctly.

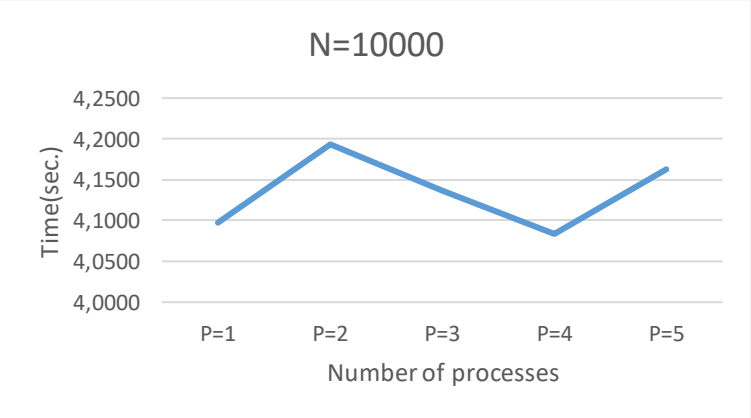
```
#multiplying data fraction in master
c[data_frac*i:] = A[data_frac*i:,]@b

#receiving data from workers and constructing final result
for i in range(1,num_workers):

   data = comm.recv()
   j = data['owner']
   c[data_frac*(j-1):data_frac*j] = data['c_partial']
```

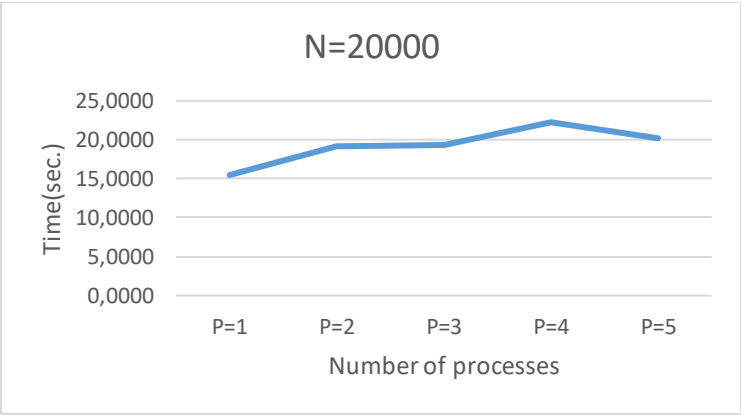
- **Results**

We ran experiments for different values of N and for different numbers of processes. Each configuration was run five times and we tabulate and plot the mean of the execution time among these experiments. As we can see from the pictures, there is not gain in parallelizing the computation, since there is an increment in time due to the communication overload.

Summary Mean Computation Time			
N. of Processes	**N=1000**	**N=10000**	**N=20000**
P=1	0,0312	4,0969	15,4726
P=2	0,0437	4,1933	19,0907
P=3	0,0765	4,1372	19,3263
P=4	0,0933	4,0835	22,2514
P=5	0,1411	4,1622	20,1542







Exercise 3: Parallel Matrix Operation using MPI

- **Problem formulation:**

We want to perform the following mathematical operation

$$C = AB$$

Wher, $A, B, C \in R^{N \times N}$ (it is, square matrices with size $N \times N$).

- **Distributed reformulation of the problem**

If we have ***M*** workers, which can compute using some part of the vectors then we can divide the matrix A in M parts, such as:

$$A^{(j)}, j = 1 \dots M,$$

Where $A^{(j)}$ is the fraction of the matrix A that the worker ***j*** receives and. These fractions can be calculated in the following way:

$$A^{(j)} = \begin{pmatrix} A_{(j-1)\cdot d,1} & \cdots & A_{(j-1)\cdot d,N} \\ \vdots & \ddots & \vdots \\ A_{j\cdot d,1} & \cdots & A_{j\cdot d,N} \end{pmatrix}, \qquad if\ j = 1 \dots M-1$$

$$A^{(j)} = \begin{pmatrix} A_{(j-1)\cdot d,1} & \cdots & A_{(j-1)\cdot d,N} \\ \vdots & \ddots & \vdots \\ A_{N,1} & \cdots & A_{N,N} \end{pmatrix}, \qquad if\ j = M$$

Where $d = \lfloor N/M \rfloor$. The matrix B is not fractionated. **The whole matrix B is sent to all workers.**

The data fractions are computed and sent by the **master** to every worker using MPI point to point communication. Once each worker receives the data, they can compute the output as a matrix multiplication:

$$C^{(j)} = A^{(j)}B$$

Here we have the partial results of the final matrix C**.** Afterwards, every worker sends back the data to the master which can concatenate the output to obtain the final value for C. Every $C^{(j)}$ is a row vector, therefore, after concatenation, we get a matrix.

$$C = \begin{pmatrix} C^1 \\ \dots \\ C^M \end{pmatrix}$$

Both in the notation and in the implemented program, the master receives the data for $j = M$.

We represent graphically the way the data is distributed in the figure 1, where we use *N*=3 and *M*=3. Here we show four different parts:

1) The master generates matrices A and B, the master does not segment the matrices, but send them both using a ***broadcast***.
2) Each worker receives the data (two matrices), select only the assigned fraction to process and multiply.
3) The master ***gather*** the data and concatenates to obtain the final result.

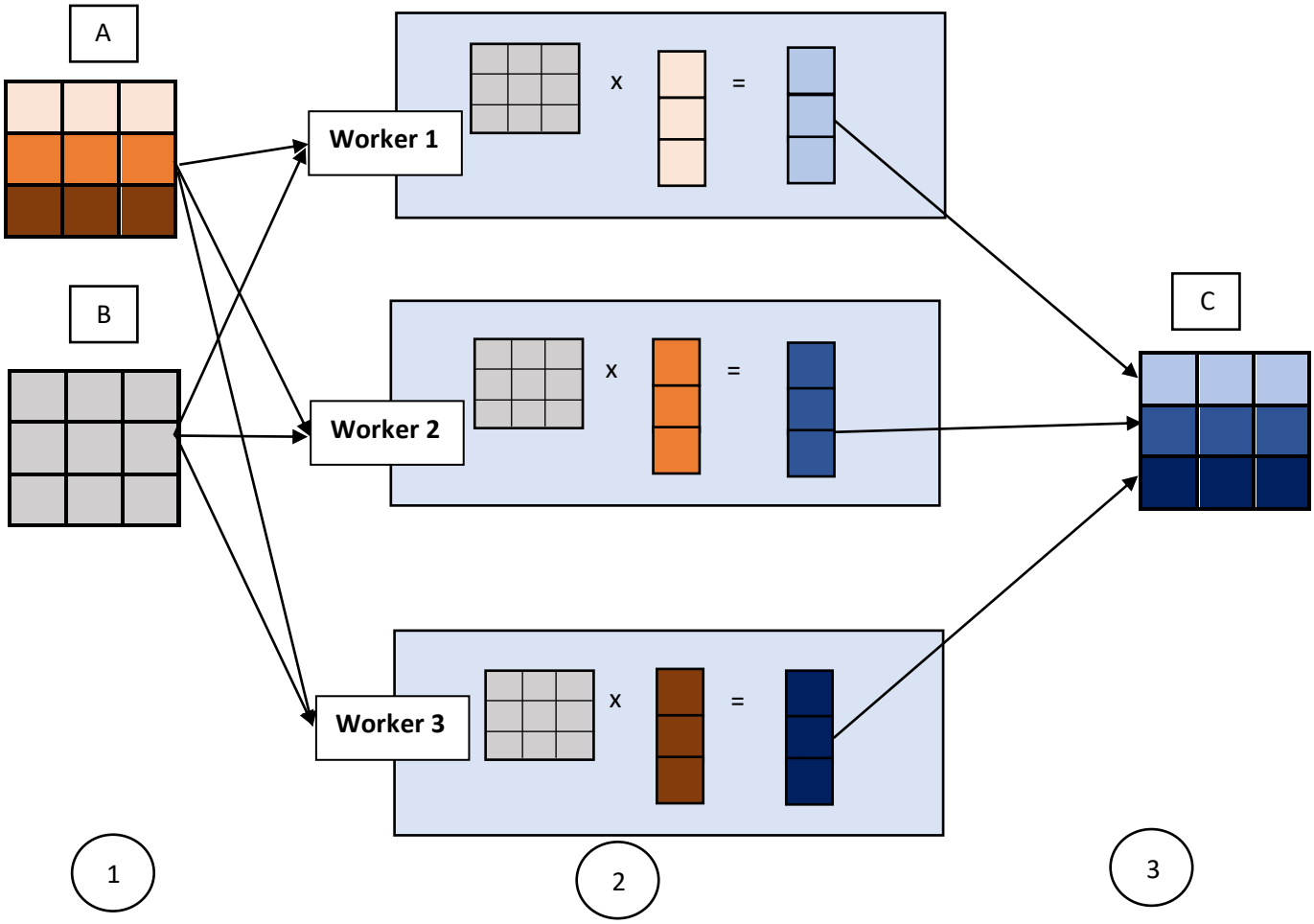


Figura 4. Strategy to divide data between workers for matrix multiplications

- **Amount of information exchanged and amount of operations:**

Given that the master sends two matrices of size $N \times N$. The workers send back together $N \times N$ values, therefore the amount of data exchanged is in the order of $3N^2$. On the other hand, the amount of operations for each worker is approximately N^2/M multiplications and N^2/M additions.

- **Code**

  - *Importing libraries and initializing important libraries*

```
#importing libraries
from mpi4py import MPI
import numpy as np
from random import random
import time
import sys

#initializing important variables
comm = MPI.COMM_WORLD
num_workers = comm.Get_size()
worker = comm.Get_rank()
N= int(sys.argv[1])
```

  - *Initializing data in worker:* The worker creates two random matrices to be broadcasted: A and B. He also creates an empty matrix which will store the result of the multiplication. The workers initialize two variables A and B (empty), which will store the broadcasted matrices.

```
#initializing data in master and workers
if worker ==0:

    #Creating the random matrices to multiply
    A = np.random.randint(1,10,(N,N)).astype(int)
    B = np.random.randint(1,10,(N,N)).astype(int)
    C = np.zeros ((N,N)).astype(int)

else:
    A = None
    B = None
```

- *Broadcasting data*: the created matrix are broadcasted by the master. Now, every worker can have access to them.

```
#broadcasting matrices
A = comm.bcast(A, root=0)
B = comm.bcast(B, root=0)
```

- *Computing matrix multiplications:* Every worker access to the assigned fragment (as explained before) and computes the partial matrix output.

```
if worker != (num_workers-1):
    C_partial = {'C':A[worker*data_frac:(worker+1)*data_frac,]@B,
                 'owner':worker}
else:
    C_partial = {'C':A[worker*data_frac:,]@B,
                 'owner':worker}
```

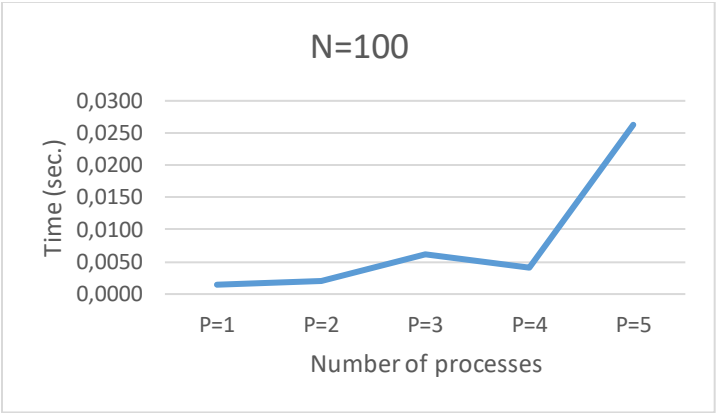
- *Gathering data in master:* the master gathers the data processed by the different workers. The gathered data is organized so that a final matrix C can be output.

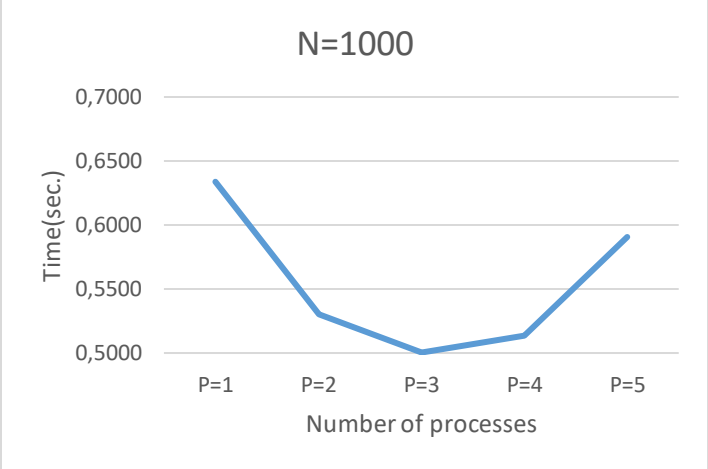
```
#gathering data on master
data = comm.gather(C_partial, root=0)

#processing gathered data on master
if worker==0:

    #iterating over the gathered data
    for element in data:

        #interpreting the gathered dat
        owner = element['owner']
        C_partial = element['C']

        if owner != (num_workers-1):
            C[owner*data_frac:(owner+1)*data_frac,:]=C_partial
        else:
            C[owner*data_frac:,:]=C_partial
```

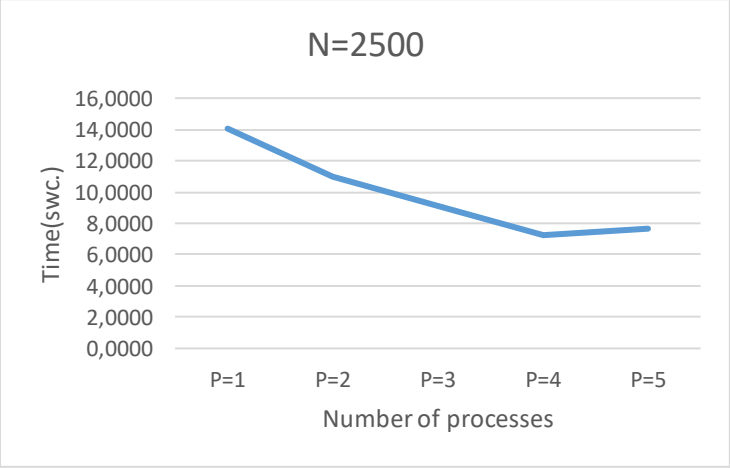
- **Results**

We ran experiments for different values of N and for different numbers of processes. Each configuration was run five times and we tabulate and plot the mean of the execution time among these experiments. The parallel matrix multiplication seems to be harmful for small matrix sizes (N=100), however for N=1000 and N=2500, there is a gain in computational time. Surprisingly, for N=2500 ,having 4 processes reduces the computation time in almost 50%.

Summary Mean Computation Time			
N. of Processes	**N=100**	**N=1000**	**N=2500**
P=1	0,0015	0,6338	14,0525
P=2	0,0020	0,5299	10,9479
P=3	0,0061	0,5004	9,0824
P=4	0,0041	0,5134	7,2414
P=5	0,0263	0,5911	7,6851







Notes

All the measurement tables used to get the graphs and the mean are provided in the zip file with the source code.

References

- Lecture on MPI from Big Data Analytics: https://www.ismll.uni-hildesheim.de/lehre/bd-19s/script/bd-02new-openmpi-python.pdf
- Tutorial of MPI4PY: https://mpi4py.readthedocs.io/en/stable/tutorial.html