Laboratory 1

Miguel Angel Caldas Villamizar

0. System specs

Hardware	MacBook Pro early 2011
Software	macOS High Sierra 10.13.4
RAM	8GB 1600MHz DDR3 (Personally upgraded)
Python version	Python 3.5.2 :: Anaconda custom (86_64)
Processor	2.3GHz Intel i5
# Cores	2

1. Adding vectors and finding average of result vector

   For this exercise it was used the *point-to-point* communication technique.
   The algorithm is basically the following:
   The root creates both random vectors. Takes a slice of both vectors and sends the remaining of both vectors to the next rank. The next rank does the same, takes a slice of both and sends the remaining to the next rank. The final rank has to take whatever is left from both vectors (The sizes should be very similar in all ranks given the fact that the portions are based on the division of the total length of the vector with the amount of ranks).
   The root processes its share and sends it to the next rank.
   Each rank has to process its share, append what received from the previous rank and send to the next rank its result (a sub-vector with the values of adding the slices of the original vectors).
   The final rank gets the whole result vector.
   Part B is done exactly the same, but instead of creating a sub-vector to send to the next rank, the send the sum of all numbers inside the slice. Finally the last rank divides the result with the size of the original vector to get the average.

Part	Number of workers	Size of data	Time in seconds
A	4	10^6	0.51
B	4	10^6	1.36
A	3	10^6	0.49
B	3	10^6	0.96
A	2	10^6	0.33
B	2	10^6	0.59

   The program is designed to do task A followed by task B on one run. It is possible to observe in the previous table that for a problem as simple as

adding two vectors the use of parallel programming is not only not necessary but it makes it worse than doing it in simply one thread.
In the following table it is confirmed that using parallel computing for this problem is just a bad idea. The next one uses the maximum dimension that the device described on part zero could hold.

Part	Number of workers	Size of data	Time in seconds
A	4	10^7	5.1
B	4	10^7	13.86
A	3	10^7	4.49
B	3	10^7	9.74
A	2	10^7	4.35
B	2	10^7	7.7

2. Dot product of matrix and vector

   For this problem it was used a very similar algorithm to the one on problem 1. The root creates the matrix and the vector. This time the slices will be an array of two elements: an array with some rows from the matrix and a sub-vector from the original one.
   As the previous problem, the root sends to the next rank so it can takes its share and send to the next one whatever is left until done.
   The main difference with the previous problem is that this time instead of calculating the sum of two numbers each one from the sub-vectors; it will calculate the linear combination of the whole row from the matrix with the number of the vector.

$$Result = \sum_{i=0}^{N} x_i * v$$

   'V' being the number from sub-vector and 'N' being the size of the rows. This number is going to be stored in a vector that will be send to the next rank so that the last rank receives the concatenation of all results.

Number of ranks	Dimension of data	Time in seconds
2	10^3	0.16
3	10^3	0.23
4	10^3	0.33

   In this problem similar to the previous one, it's to simple for a thread to multiply all values of the row with the same number, so it is not necessary to do parallel computing.

Number of ranks	Dimension of data	Time in seconds
2	104	17.86
3	10^4	30.61
4	10^4	57.01

The results confirm the conclusion of not using parallel computing for this simple problem. (The maximum dimension that the hardware could hold was 10^4)

3. Dot product of two matrixes

In this final problem it was allowed to use collective communication, which makes the process more efficient.

The dot product of two matrixes is the following:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} x \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + cg & af + ch \\ ce + dg & cf + dh \end{pmatrix}$$

Each element of the result matrix is the result of the linear combination of each row from the first matrix and the column of the second matrix.

This means two important points:

- It is impossible for a worker to do its job without at least one full matrix. Each worker needs minimum one matrix to do the job.
- Because python defines a matrix as a vector of vector, normally is not easy to go through a matrix in other way than horizontally (advancing through the rows). It's better if before starting the calculations, the second matrix is rotated in order for the columns to become the rows.

The algorithm does these two things, and then slices the first matrix. The root creates a vector of 4 elements. Each element is a tuple that contains a slice of the first matrix and the whole second matrix rotated.

Using scatter every rank gets it own slice and second matrix. The slice of the matrix as in exercise 2 is a list of rows. Each rank calculates the linear combinations and stores them in a sub-matrix (array of arrays).

Finally the root gathers all sub-matrixes and appends them into one final result matrix.

Number of workers	Dimension of matrixes	Time in seconds
2	10^2	0.09
3	10^2	0.121
4	10^2	0.117
2	10^3	85.45
3	10^3	85.08
4	10^3	88.33

This last task shows two different results. It is indeed a complicated operation that may benefit from parallel computing. However, probably because the hardware used only has two cores on its processor; the use of more than three workers stops improving the execution time. Also if the size

of the matrixes is not so big, it applies as a simple task that is better to perform without using parallel programming.
If there are any doubts about the algorithm, the code is well commented and should solve them.