3116 – Lab Distributed Data Analytics

Exercise 1

Yuvaraj Prem Kumar

303384, premyu@uni-hildsheim.de

Exercise 0: Explain your system

System Specifications:
Processor: Intel i5-8250U CPU@1.80 GHz
RAM: 8 GB
Graphics: Intel UHD Graphics 620
OS: Windows 10 Home 64-bit
Python version: 3.6.7

Exercise 1: Basic Parallel Vector Operations with MPI

This is a vector and vector addition, which will output a vector of the same array size as inputs. The following are the steps in the code. The MPI send and receive code portion is taken from BDA Lecture 2, Slide 8.

Basically, the main idea here is to equally split the two vectors into "slices". This will be distributed across the worker nodes, in order to simulate a distributed execution environment. The root node acts as the 'master', to send the split datasets to the worker nodes for computation, which is then returned back to join as the final output.

Part(a)

The code is run with array size N of $\{10^2,10^5,10^7\}$. Values of $10^n>10^7$ are crashing my PC.
The workers argument used is {2,4,12} in part(a) and part (b).

1. Root node [0] initializes 2 random integer numpy array lists of size N (V1 and V2)
2. Root node [0] calls the function “slices” to divide V1 and V2 into equal slices based on number of workers “P”. The function call “slices” is used throughout the exercise.
3. The slices are sent to the worker nodes, P as per the command line input “-n”
4. The workers nodes perform the parallel sum and send it back to the root node
5. Root node receives the sum and stores in the final output vector

Array size N	# of Workers	Avg. processing time (s)
10^2	2	0.00228693
10^2	4	0.003225595
10^2	12	0.108386833
10^5	2	0.009248414
10^5	4	0.012645244
10^5	12	0.112766133
10^7	2	0.631371333
10^7	4	0.654542145
10^7	12	0.741908563

Table 1: Result set for question 1 part(a)

We can see that the average computation time actually increases with the number of workers. Vector addition is dominated by the cost to read and write the values from memory. This is mainly due to the fact that the program is actually running on one physical computer, as opposed to a ‘real’ distributed execution environment.

Array size N	# of Workers	Avg. processing time (s)
10^2	2	0.0015451
10^2	4	0.003562377
10^2	12	0.178001667
10^5	2	0.004199532
10^5	4	0.004965255
10^5	12	0.10249941
10^7	2	0.27010294
10^7	4	0.32366546
10^7	12	0.383168877

Table 2: Result set for question 1 part(b)

The same situation is observed for part(b); based on the same reasoning as in part(a). As discussed in the lectures, when multiple processes are initialized, they will actually create an interleaving effect. Hence each process may take longer to complete. However, with a true "big data" parallel execution environment across a cluster of computers, we should see better returns of parallelization.

Exercise 2: Parallel Matrix Vector multiplication using MPI

The same method in exercise 1 is applied here, with a few exceptions as noted in the following steps:

1. Root node [0] initializes one random integer numpy array lists of size N and one random integer numpy matrix array of size (N*N).
2. Root node [0] calls the function "slices" to divide the vector into equal slices based on number of workers "P".
3. Similarly, the matrix is horizontally split, using 'numpy.hsplit' into equal slices of (1x100) for each N. This is done for the regular matrix.vector dot product (numpy.matmul)
4. The slices are sent to the worker nodes, P as per the command line input "-n"
5. The workers nodes perform the dot product and send it back to the root node
6. Root node receives the sum and stores in the final output vector

A matrix * vector dot product will return a vector, so the main step here is to split it column-wise, then perform the dot operation with the vector.

$$A\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}.$$

Figure 1: Matrix-vector dot operation[1]

The program is run with Matrix:= N{10^2,10^3} using workers, P{2,4,12}. My computer is unable to handle matrices of N{10^4} as it will hang indefinitely.

Array size N	# of Workers	Avg. processing time (s)
10^2	2	0.013879162
10^2	4	0.01546124
10^2	10	0.084050369
10^3	2	1.20603192
10^3	4	1.135630883
10^3	10	1.130775992

Table 3: Result set for question 2

There is a large jump from the 10^2 x 10^2 matrix to the 10^3 x 10^3 matrix, due to the actual size of the matrix operations – accounting for the large increase in average computation time. Besides that, the difference in execution time is negligible considering the number of workers. Again, a true parallel execution environment should be a much better result.

Exercise 3: Parallel Matrix Operation using MPI

The same method is followed here, only with a matrix by matrix multiplication. Except for here, the matrices are split into the horizontal and vertical arrays each, based on the following formula[7]:

$$c_{ij} = a_{i1}b_{1j} + \cdots + a_{im}b_{mj} = \sum_{k=1}^{m} a_{ik}b_{kj},$$

1. Root node [0] initializes two random integer numpy matrix array of size (N*N).
2. Root node [0] calls the function "slices" to divide the vector into equal slices based on number of workers "P".
3. Similarly, the matrix is horizontally split, using 'numpy.hsplit' into equal slices of (1x100) for each N. This is done for the regular matrix.vector dot product (numpy.matmul)
4. The slices are sent to the worker nodes, P as per the command line input "-n"
5. The workers nodes perform the dot product and send it back to the root node
6. Root node receives the sum and stores in the final output vector

Array size N	# of Workers	Avg. processing time (s)
10^2	2	0.025510079
10^2	4	0.02604199
10^2	10	0.109240165
10^3	2	2.481332791
10^3	4	2.19969792
10^3	10	2.148043452

Table 4: Result set for question 3

Interestingly enough, the runtime for decreases when the number of workers increases. However such a small difference could be negligible or due to other factors considering the amount of RAM available at the time. On the other hand, we can see the runtime from 10^2 x 10^2 to 10^3 x 10^3 matrix multiplication actually increases by a factor of almost 100, this is in line with the expectation since the dataset also increase by 10^2 .

References:

[1] Math Insight. (n.d.). Retrieved from https://mathinsight.org/matrix_vector_multiplication
[2] numpy.split — NumPy v1.16 Manual. (n.d.). Retrieved from https://docs.scipy.org/doc/numpy/reference/generated/numpy.split.html
[3] MPI recv from an unknown source. (4, 8). Retrieved from https://stackoverflow.com/questions/4348900/mpi-recv-from-an-unknown-source
[4] Using. (n.d.). Retrieved from https://www.codecademy.com/en/forum_questions/54b59dcb9113cb9c56003fd8
[5] Python Parallel Matrix Vector Multiplication. (9, 4). Retrieved from https://stackoverflow.com/questions/25045176/python-parallel-matrix-vector-multiplication
[6] numpy.hsplit()_w3cschool. (n.d.). Retrieved from https://www.w3cschool.cn/doc_numpy_1_13/numpy_1_13-generated-numpy-hsplit.html
[7] Matrix multiplication. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Matrix_multiplication

Appendix A

Result set of exercises using simple vector / matrix of 10^1 and 2 worker nodes. This is to simply check the mathematical correctness of the output result set

Exercise 1A

```
C:\Users\yuvar\Documents\PythonProjects\DDA_Lab1>mpiexec -n 2 python Lab1ex1a.py
Starting program
Slices are processing on Node: 1
Starting program
Starting root Node 0
Vector 1: [[3 5 7 8 4 3 6 1 3 7]]
Vector 2: [[7 9 7 2 8 2 3 3 9 3]]
V3 current loop is ..(array([[3, 5, 7, 8, 4, 3, 6, 1, 3, 7]]), array([[7, 9, 7, 2, 8, 2, 3, 3, 9, 3]]))
Sending slices...
Receiving from Node: 1
Root node has received all the slices...
Vector 3 output: [array([[10, 14, 14, 10, 12,  5,  9,  4, 12, 10]])]
Total time taken: 0.0017572952783666551
```

Exercise 1B

```
C:\Users\yuvar\Documents\PythonProjects\DDA_Lab1>mpiexec -n 2 python Lab1ex1b.py
Processing on Node: 1
worker is doing the tasks..
Starting root Node 0
Vector 1 input: [[2 1 7 8 5 1 6 8 1 2]]
Root node has received all the slices...
Vector average output: [array([4.1])]
Total time taken: 0.0014358735134010203
```

Exercise 2

```
C:\Users\yuvar\Documents\PythonProjects\DDA_Lab1>mpiexec -n 2 python Lab1ex2.py
H-Slices are processing on Node: 1
Starting root Node 0
Matrix A input is: [[1 2 5 9 9 2 5 6 7 6]
 [1 5 2 4 3 6 5 6 3 7]
 [1 3 1 2 3 5 2 1 5 2]
 [8 8 3 2 9 9 6 8 1 8]
 [8 1 2 4 7 6 3 5 1 6]
 [1 8 2 1 3 9 6 2 7 4]
 [8 8 7 3 2 6 2 2 8 2]
 [2 4 8 5 1 6 5 6 3 6]
 [4 9 4 5 6 1 4 4 8 6]
 [6 2 4 3 8 9 8 2 5 7]]
Vector B input is: [[4 6 5 3 9 6 3 3 9 6]]
Root node has received all the slices...
Vector C output: [array([303, 246, 218, 260, 298])]
Total time taken: 0.0018841573219106067
```

Exercise 3

```
C:\Users\yuvar\Documents\PythonProjects\DDA_Lab1>mpiexec -n 2 python Lab1ex3.py
H-Slices and V-Slices are processing on Node: 1
Starting root Node 0
Matrix A input is: [[5 6 7 9 6 2 3 7 7 9]
 [9 5 5 9 5 1 9 1 7 3]
 [5 2 3 3 2 3 4 2 7 9]
 [1 3 7 9 9 2 2 5 3 1]
 [7 8 1 1 2 6 2 2 6 8]
 [2 6 1 1 4 6 3 1 1 9]
 [3 9 3 5 2 9 1 8 7 7]
 [7 7 5 3 7 5 7 1 6 9]
 [8 1 2 3 4 1 8 9 4 3]
 [6 7 2 5 9 6 7 2 4 7]]
Matrix B input is: [[9 4 3 8 9 5 2 5 5 8]
 [8 3 2 2 4 6 5 8 8 7]
 [2 3 3 7 3 4 4 7 3 7]
 [6 4 1 5 6 2 9 1 7 9]
 [1 5 4 6 3 5 6 9 8 6]
 [1 1 8 3 7 9 2 5 6 7]
 [2 1 9 7 8 8 4 2 5 5]
 [6 4 4 9 3 1 7 3 1 1]
 [1 5 6 9 6 7 9 8 4 4]
 [3 9 5 5 5 3 5 8 4 3]]
Root node has received all the slices...
Matrix C output: [array([185, 156, 221, 192, 187]), array([268, 296, 314, 270, 290])
, array([309, 282, 354, 309, 331]), array([199, 228, 205, 190, 217]), array([296, 27
9, 324, 316, 326])]
Total time taken: 0.0020258104523236398
```