Homework 4

Parallel Distributed Num Algorithms

<u>Name:</u> Saurabh Kumar
<u>UIN:</u> 926009924

1. You are required to develop a parallel C/C++ implementation of the above algorithm that uses OpenMP directives to parallelize the `compute_inverse` routine. You should use the task directive for the recursive tasks.

Ans: The parallelized code using OpenMP has been submitted on eCampus. OpenMP "task" directive has been used to parallelize the recursive portion of the code.

1.1 Steps to Run

Ans: The submission contains the following files:

a. **inverse.cpp :** The source code with parallelized function for calculation of inverse

b. **grid.job :** Job file for submission of the parallel job to ada cluster. The job file has commands to run the parallel code for 1, 2, 4, 10 & 20 threads automatically.

c. **run.sh :** A bash script that compiles the c code and submits it to the ada grid. It has commands to set the environment in the shell for openMP compilation and job submission.

d. **Command to run :** ***./run.sh*** (This command will automatically submit the job for 1, 2, 4, 10 & 20 threads)

e. **output.txt :** Submission output report generated by my submission.

2. Describe your strategy to parallelize the algorithm. Discuss any design choices you made to improve the parallel performance of the code.

Ans: I have implemented the solution in C. The c*ompute_inverse* function is analogous in behavior as that in the matlab code. It takes a dynamically allocated 2D matrix as a primary input and computes the inverse of the matrix in place. The matrix is passed by reference to the function and hence, the return type is void.

The *compute_inverse* function checks if the number of rows and columns of the input matrix are the same. It errors out if they don't match because inverse of a non-square matrix cannot be calculated.

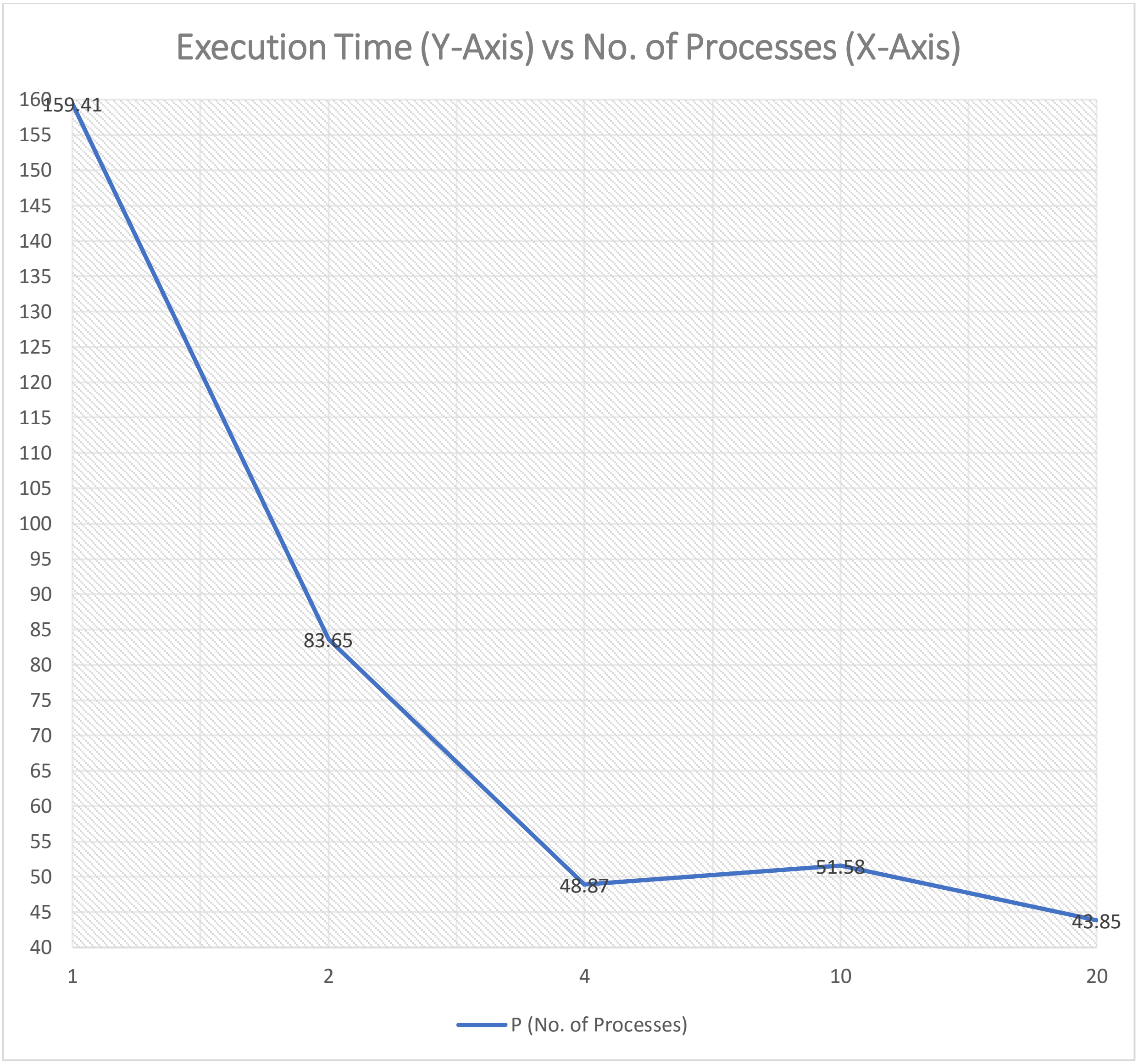
During each function call, the *compute_inverse* function divides the matrix into 4 parts. It ignores the bottom left matrix as it is filled up with 0's (Input matrix is an upper triangular matrix). It then recursively calls the *compute_inverse* function of upper left and bottom right matrices and calculates the value of upper right matrix using the return values from the recursive calls. The same strategy has been used in the matlab implementation. If the number of elements in the matrix is 1, the *compute_inverse* function simply returns the reciprocal of the number as its inverse. The design choice made in order to improve parallel efficiency is that during each recursive call, the matrix is divided into 4 parts which are roughly equivalent is size so that each parallel process is almost equally loaded. Secondly, the functionality has been implemented in a way to eliminate dependency amongst for loop statements so that all of them can be run in parallel without compromising the correctness of the result.

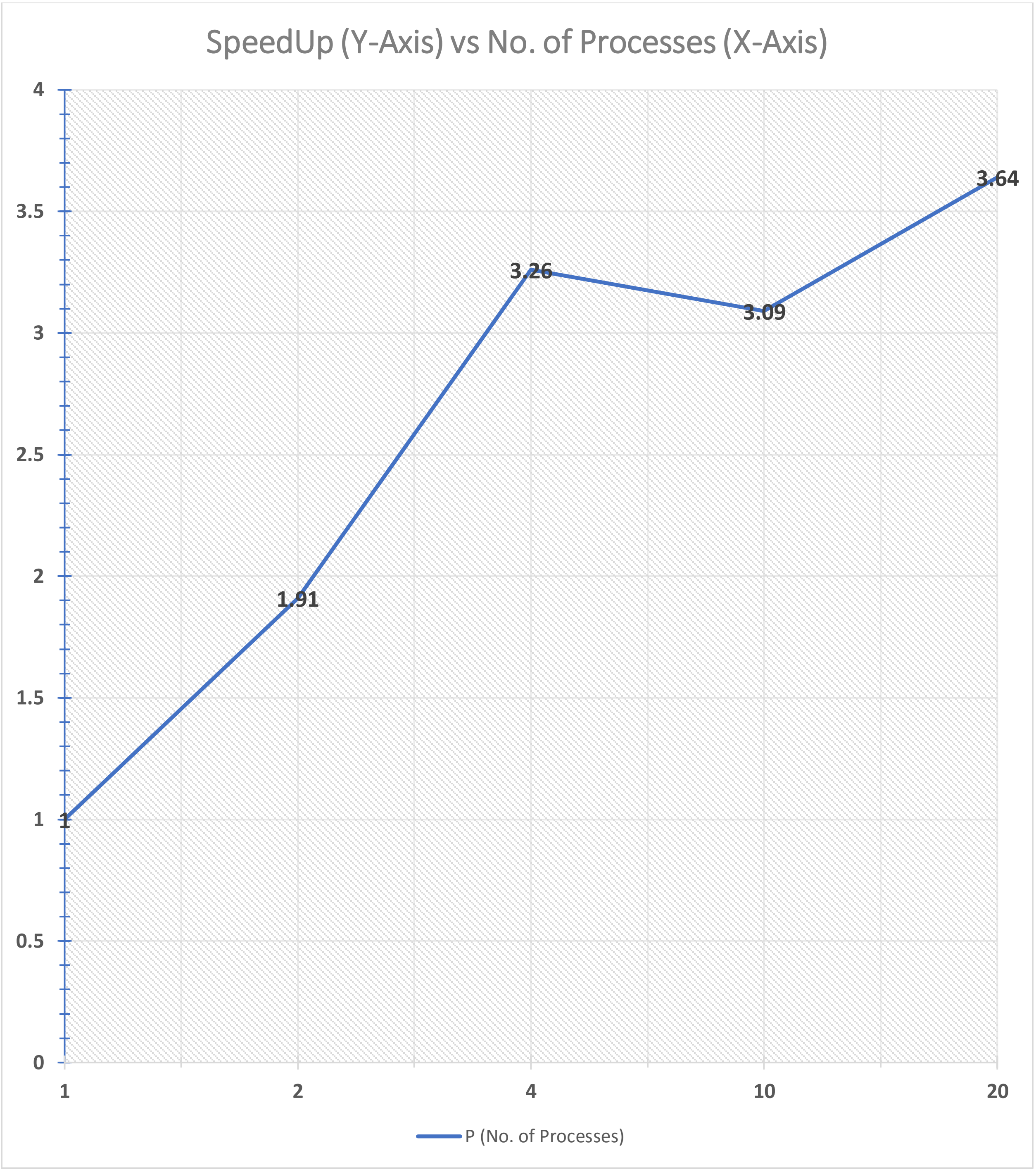
3. Determine the speedup and efficiency obtained by your routine on 1, 2, 4, 10, and 20 processors. You may choose appropriate values for the matrix size to illustrate the features of your implementation.

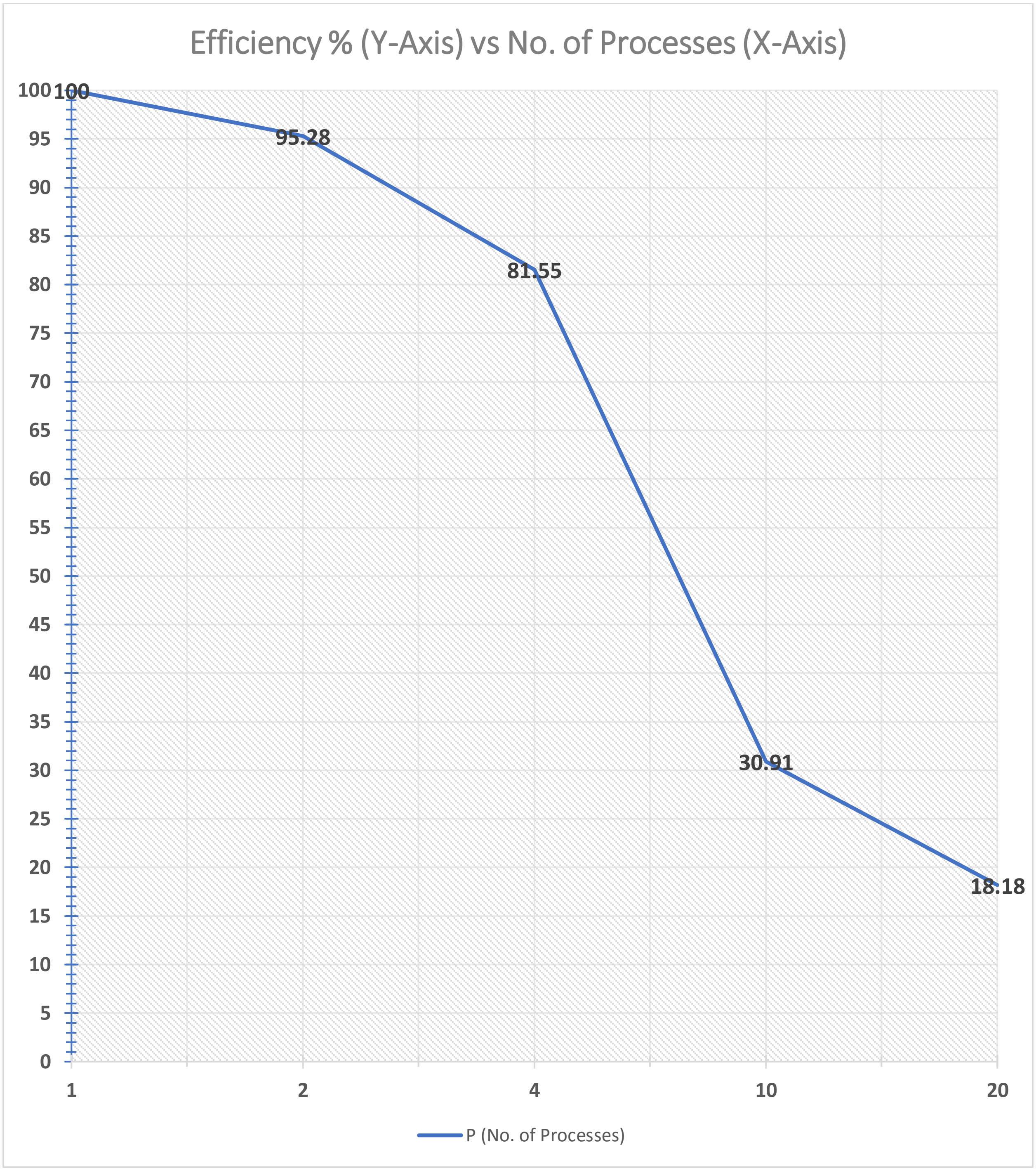
Ans: <u>SpeedUp/Efficiency Calculations:</u>

The matrix size used for Speedup/Efficiency calculations is 4096*4096.

Processors	Execution time (secs)	Speedup	Efficiency
1	159.41 secs	1	100%
2	83.65 secs	1.91	95.28%
4	48.87 secs	3.26	81.55%
10	51.58 secs	3.09	30.91%
20	43.85 secs	3.64	18.18%




SpeedUp (Y-Axis) vs No. of Processes (X-Axis)
4
3.5
3
2.5
2
1.5
1
0.5
0
1
2
4
10
20
1
1.91
3.26
3.09
3.64
P (No. of Processes)


Efficiency % (Y-Axis) vs No. of Processes (X-Axis)
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
100
95.28
81.55
30.91
18.18
1
2
4
10
20
P (No. of Processes)