CSE 6220 Programming Assignment 3
Parallel Jacobi's Method

Wei Zhao (wzhao41@gatech.edu), Shushu Zhao (szhao317@gatech.edu),
Wenyue Wang (wwang413@gatech.edu)

April 24, 2020

1 Introduction

Jacobi's method is an iterative, numerical method for solving a system of linear equations. In this project, we first implemented a sequentially solver of linear system by using Jacobi's method. After successfully tested and verified our sequential solver, we started to implement and exploit the parallel algorithm to adapt the Jacobi's method. The implementation details is discussed in Method section and an evaluation of performance is also included in this report.

2 Method

This project aims to learn how to implement Jacobi's Method to solve the problem of matrix-vector multiplication. Here we have different method for Sequential of one processor and Parallel of multiple processors through MPI.

2.1 Sequential Jacobi Implementation

To solve x from equation $Ax = b$, Jacobi's method needs several inputs: A full rank $n * n$ matrix A, A $n * 1$ vector b, a termination accuracy *I2_termination* and a max iteration number that terminates the whole loop.

Then, the general method can be written as follows:

Initialize $x : \leftarrow [0\ 0\ ...\ 0]$
$D = diag(A)$
$R = A - D$
$D_{invert} = D^{-1}$
$cnt = 0$
$residual = ||Ax - b||$
$while(residual > I2\_termination\ \&\&\ cnt < max\_iter)\ do\{$
    $x = D_{invert} * (b - Rx$
    $residual = ||Ax - b||$
$\}$

2.2 Parallel Jacobi Implementation

To implement Jacobi's method to calculate matrix vector multiplication, there are several assumptions we need to make. The most important one is that the number of processor p should be a perfect square such that $p = q * q$, and q is an integer. Then, we will build a $q * q$ 2D grid map as the communication network.

Initially, matrix and vector will be distributed in sub-functions vector-distribution and matrix-distribution below. One thing to notice is that, matrix A and vector b were initially stored in processor $(0, 0)$ of the $n * n$ mesh.

2.2.1 Vector Distribution

The general idea of vector distribution is to equally distribute the vector stored on processor $(0, 0)$ onto processors $(i, 0)$, which is the first column of the 2D $n * n$ communicator as figure 1.

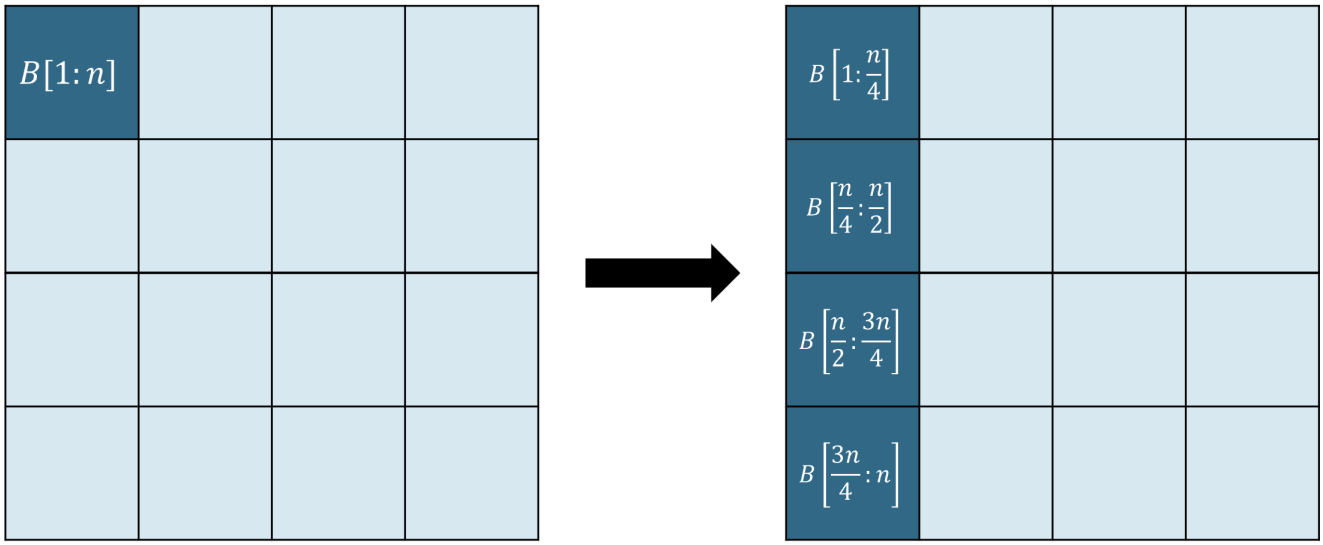

Figure 1: Runtime vs Number of Data at p = 4 from d = 0 to d = 1

This distribution should follow the block distribution as distributes n components onto q processors in a pattern of:

$$\begin{cases} \lceil n/p \rceil & if \;\; i < (n \; mod \; q) \\ \lfloor n/p \rfloor & if \;\; i >= (n \; mod \; q) \end{cases} \tag{1}$$

2.2.2 Matrix Distribution

The matrix A on (0,0) is distributed to each processor by using following procedures. First each row elements/block is distributed by using $MPIScatterv$. It's transited through column communicator created by MPI_Cart_sub. Then all elements in each row are evenly scatter to each processor in the same row using the row communicator and these two step serve the distribution of matrix.

2.2.3 Transpose Bcast Vector

The input vector x is distributed in the first column of grid (i,0) and we will send the elements from a processor (i,0) to its corresponding diagonal processor (i, i), using a single MPI Send and the sub-communicator for the row of processors. We then broadcast the received elements from (i, i)along each column of the grid using a sub-communicator for the column of processors.[1]

2.2.4 Matrix Vector Multiplication Distribution

After the transposing, each processor has the elements it needs for its local matrix-vector multiplication. We just need to let the local vector timies it's the local matrix and then use a parallel reduction to get the sum of the resulting vectors along the rows of the grid back onto the processors of the first column. And the final result of the multiplication thus ends up distributed among the first column in the same way that the input x was. [2]

2.2.5 Jacobi Distribution

In this Parallel Jacobi, matrix A and R is distributed using above procedures and the calculation result of Rx is found by using Matrix Vector Multiplication. Same as described in the project description, the vectors x, and b, are distributed among the first column of the processor grid. The diagonal elements of A need to be collected along the first column of the processor grid. So we can update x by $x_i = 1/d_i\ (b_i - (Rx)_i)$ using only local operations.[3] And the last step is to determine when to stop, as suggested in the programming assignment description, we use L2 norm to check the accuracy of calculating result. We first find the sum of square in the local and perform the parallel reduction in the first column of processors.The processor will exit if L2 norm is met.

3 Results and Discussion

3.1 Runtime

In this part, we discuss the running time with number of processors, value of n, and value of d. We divied the whole part into several cases: case 1: when p = 4, from n = 100 to n = 30000; case 2: when p = 1, from n = 100 to n = 30000; case 3: when p = 4, from d = 0 to d = 1; case 4: when p = 1, from d = 0 to d = 1.

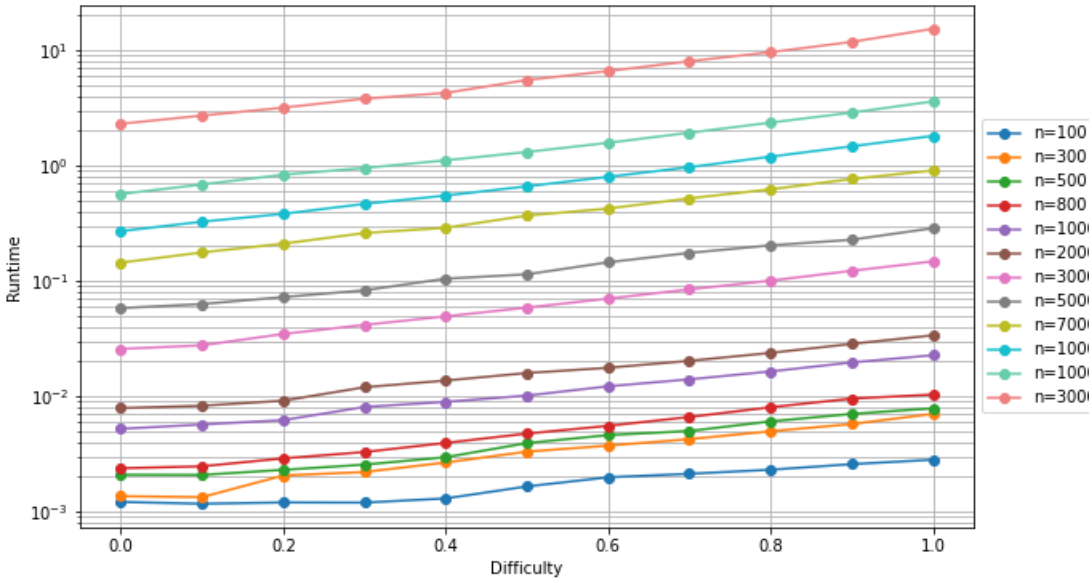

Figure 2: Runtime vs Difficulty at p = 4 from n = 100 to n = 30000

The first figure visualize the relationship between number of processors, difficulty, and running time. As we can see, with the fix of difficulty, as n increases, the running time increases orderly. Most of the gaps between the curves are uniform, but from three curves which are n = 300, 500, 800 are neared. Occupy of the running cluster and distribution of data may be potential reasons. With fixed n, as difficulties increases, the running time lifts almost follow the linear pattern. Since we have 4 processors, which is still parallel programming, so even the curve looks like linear, but different from serial pattern.

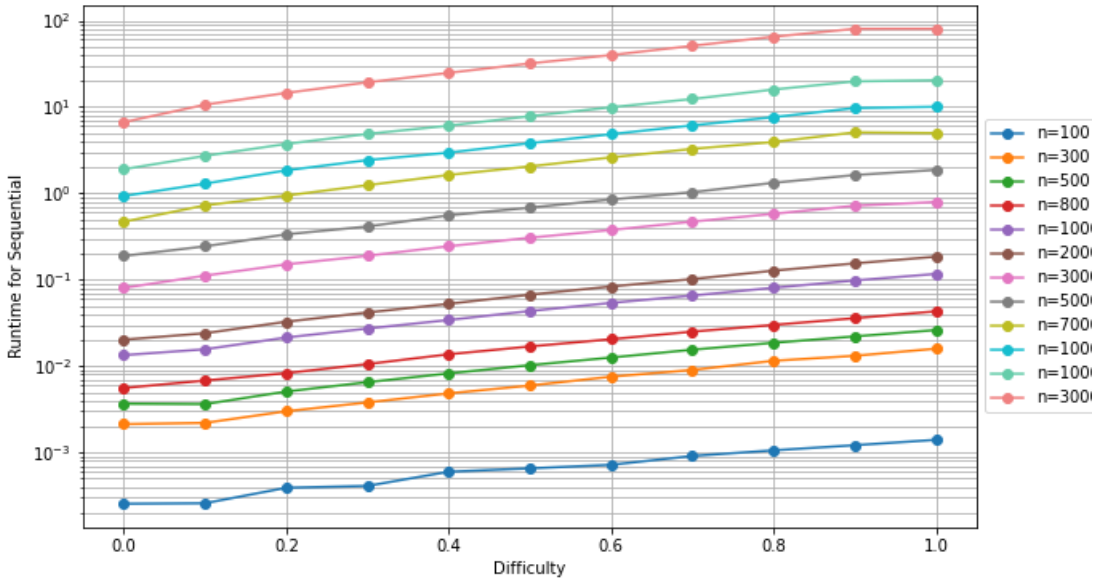

Figure 3: Runtime vs Difficulty at p = 1(Sequential) from n = 100 to n = 30000

The second figure visualize the relationship between number of processors, difficulty, and running time. Since this time we only have 1 processor, which is serial programming, so the shape of curves is similar to linear. As we can see, with fix difficulty, the running time is increased gradually, when n bigger than 3000, the gaps between each curve is more equal while the gaps between rest curves is uneven.

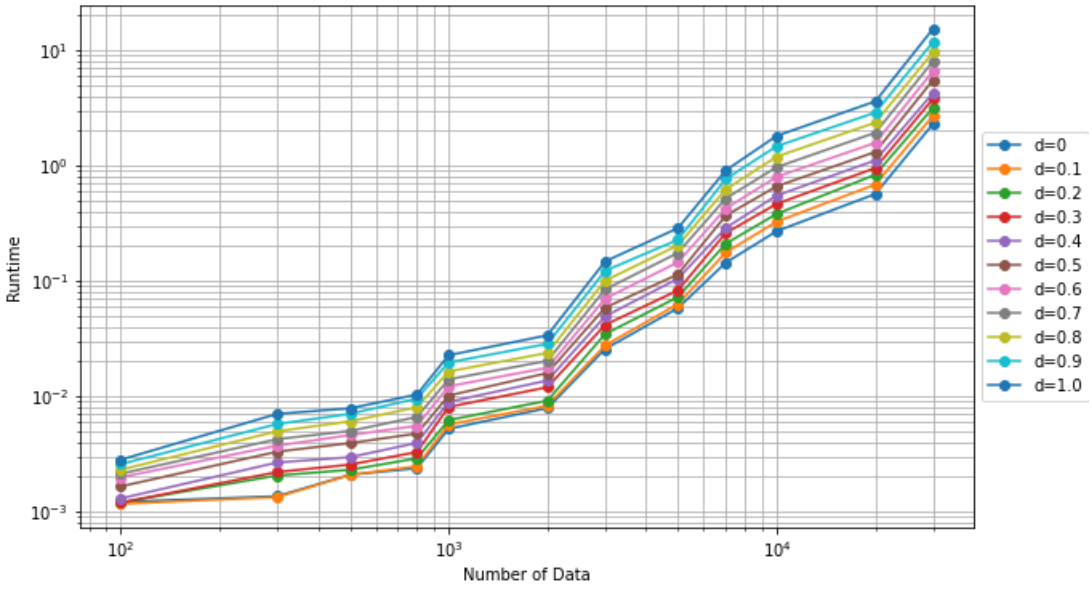

Figure 4: Runtime vs Number of Data at p = 4 from d = 0 to d = 1

For this figure, we can find that with increases of number of data, the running time is also increasing, and the shape almost follows the exponential curve. For value of d, the pattern of curves is not strictly uniform with d. When d equals to 1.0, the curve is on the top, and when d equals to 0.1, the curve is on the bottom. The curve in the middle is occasionally misaligned. Since the number of processors is 4, for parallel programming, the efficiency may be influenced by computation and communication.

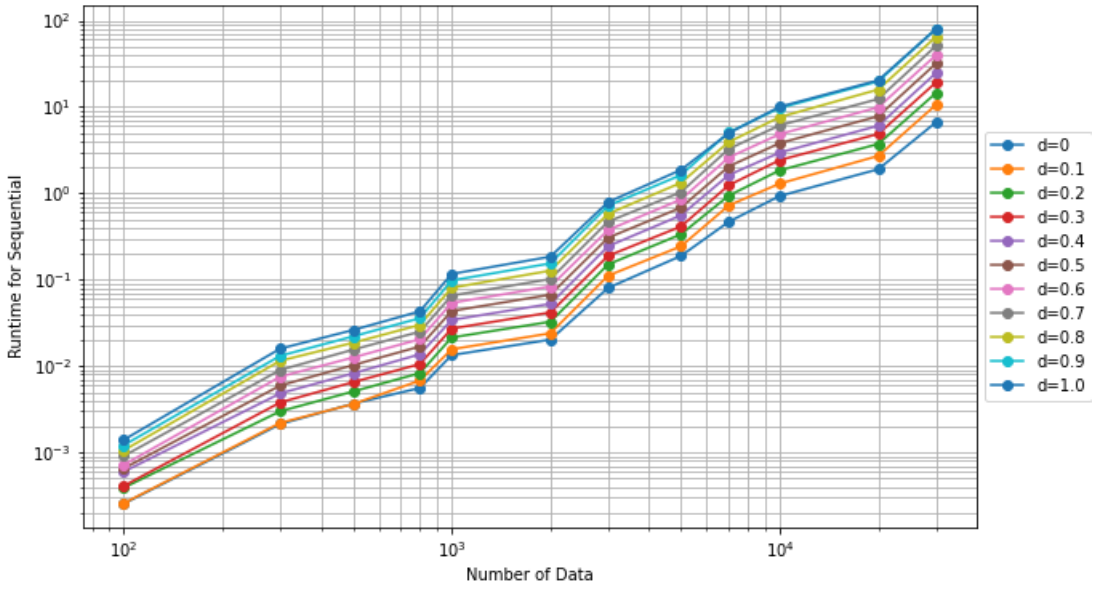

Figure 5: Runtime vs Number of Data at p = 1(Sequential) from d = 0 to d = 1

For this figure, we could find that the shape of curves is similar to linear. And With increasing of number of data, the running time is also increasing. Since we only have one processor, which is a serial programming, the distribution of curves with various value of d almost follow the rule that the smaller of d, the line is lower; the bigger of d, the line is upper. The curve which d equals to 1.0 is on the top while the curve which d equals to 0 is on the bottom.

3.2 Speedup

In this part we discuss the speed up for our programming. we discuss the situation when P = 4, the relation between number of input data with different d and difficulties with different N.

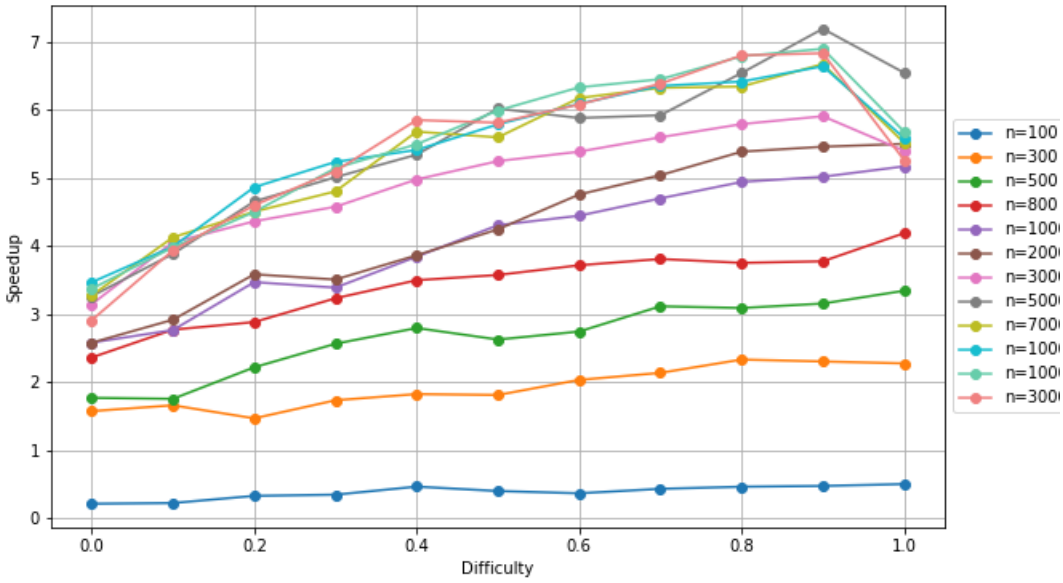

Figure 6: Speed Up vs Difficulties with Different N when P = 4

For speed up, as we can see in this figure, with fix n, the increase of difficulty also brings the increase of speed up. Although part of the curve fluctuates, the overall trend is still upward. For distribution of curves, when n equals to 100, the line is on the bottom, and when n increases, the line also set upper, and when n equals to 30000, the line is on the top.

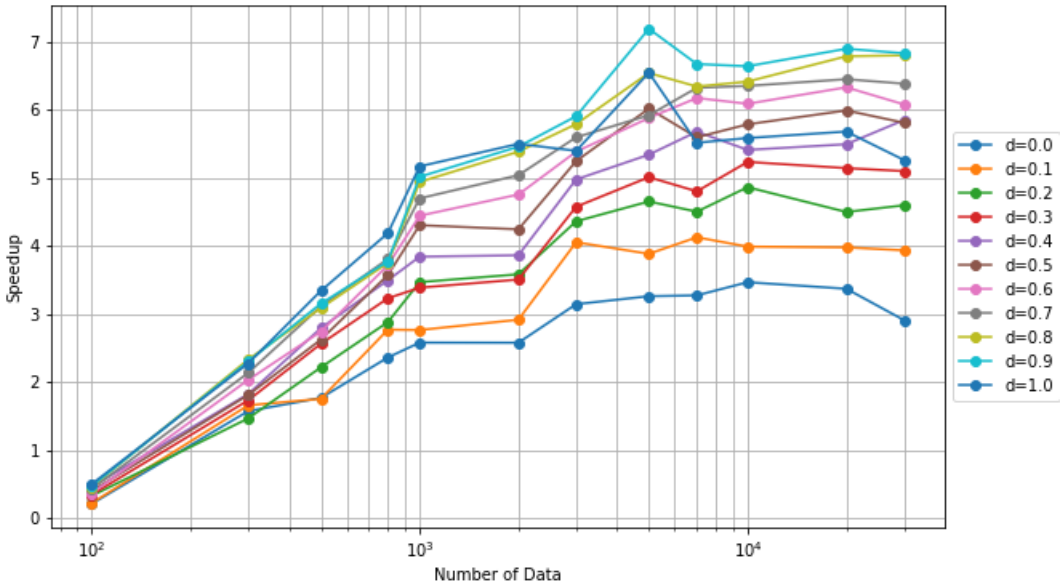

Figure 7: Speed Up vs Number of Input Data with Different d at P = 4

As we can see in the figure 6, the distribution of curves with various of d almost follow the pattern of figure 5. The smaller of d, the lower of related curve. When d equals to 0.0, the curve is on the bottom, and when d equals to 1.0, the curve is on the top. With fix d, the the pattern of curves is also upward with increasing of number of data. Although fluctuations exits in some corner, the overall situation is increasing.

3.3 Efficiency

In this section, we studied the efficiency of this program by varying different input size and difficulties. Figure 8 shows the efficiency is slightly increasing with difficulties for larger input data size but it isn't obvious for small input dataset. Figure 9 shows efficiency vs Number of input data and they are positively related, we observes similar behaviors of Efficiency as that of Speed-up. In the Figure 16, we plotted the Efficiency VS Input Data with Different P. It's interesting to observe that the P = 4 has the highest Efficiency regardless of input data size. This means p = 4 is an optimal number of processor because it makes the best trade off between communication time and computation time.

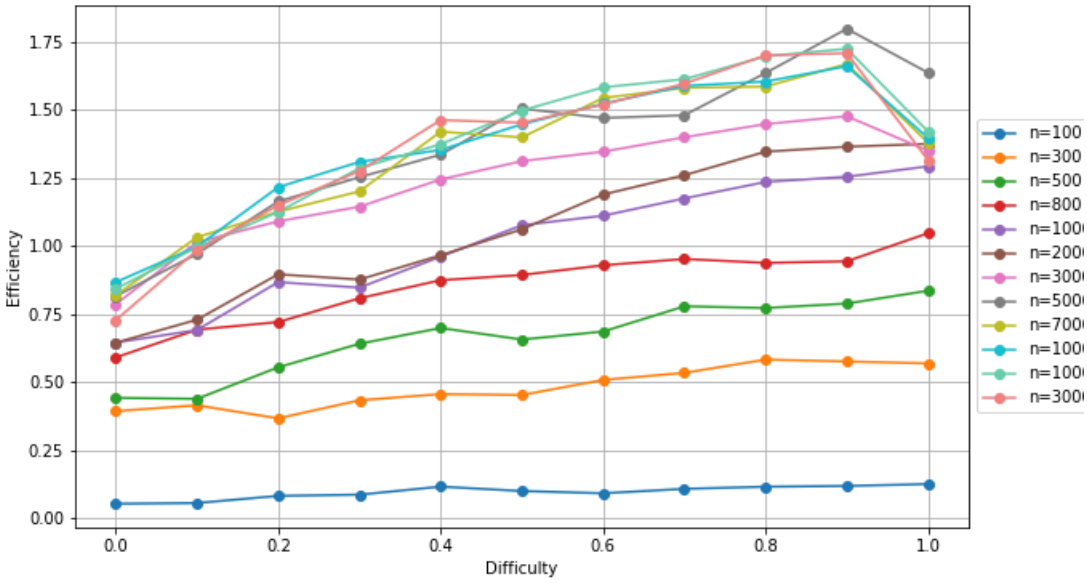

Figure 8: Efficiency vs Difficultly with Different Input when P = 4

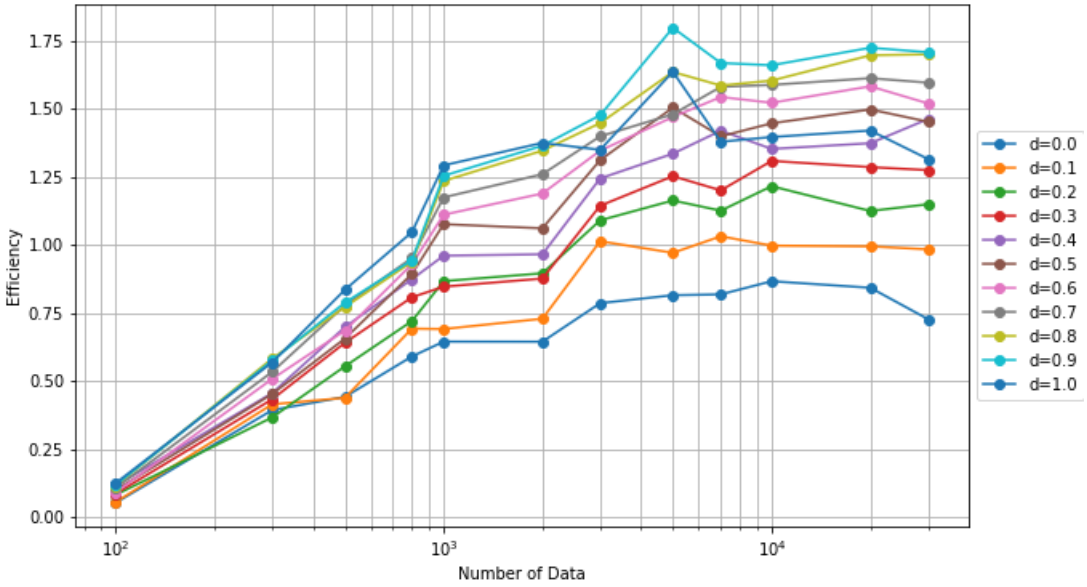

Figure 9: Efficiency vs Input Data with Different Diffculities when P = 4

3.4 Additional Analysis for Fixed Difficulty

We also studied the program performance at the fixed difficulties of input matrix. In the Figure 10, w e plot the runtime vs number of Processor with different N when d=0.5. The run time will first start to decrease as we number of processor increase but when number of processor reach P = 25, there is pike of run-time. This result is because we requested more nodes in the cluster and it significantly increase the communication time among each processor in practical.

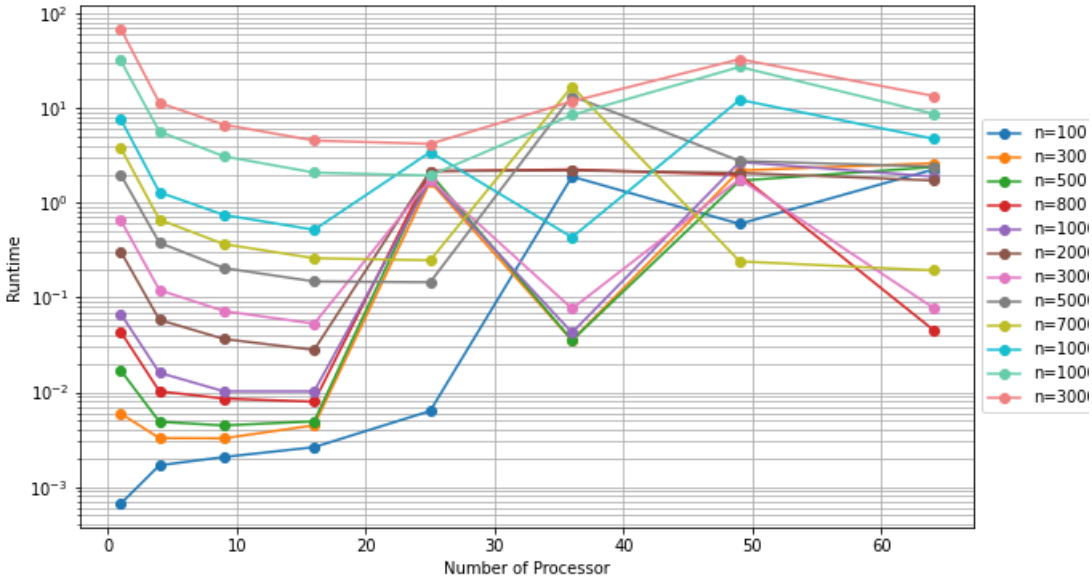

Figure 10: Runtime vs Number of Processor with Different N when d=0.5

The Figure 11 shows the run-time vs input datasize at different P. As we can see from the plot, the run-time of sequential algorithm has linear relation with input data size and we could us it as benchmark compared to other parallel algorithm, At small data size, the performance of sequential algorithm overpass the parallel one. This is as expected because communication cost will dominate the whole process. As n increase , the parallel algorithm starts it own advantages that reduce run-time by exploiting multiprocessor. For some p, the relation of run-time vs n is still linear.

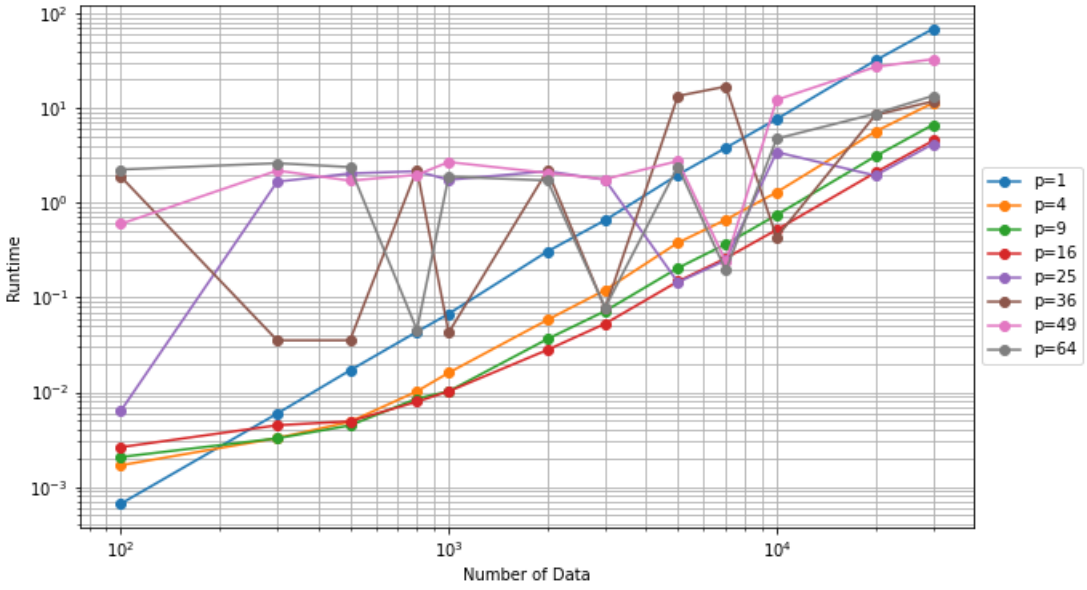

Figure 11: Runtime vs Input Data Size with different P when d = 0.5

Figure 12 and Figure 13 shows the speed up vs number of data n. It shares similar behavior as run-time. It's interesting to see that the speed up is increasing as number of processor increases until p = 16. For p larger than 16, we requested one more node to conduct this experiment and this longer communication time drag the pefromance down.

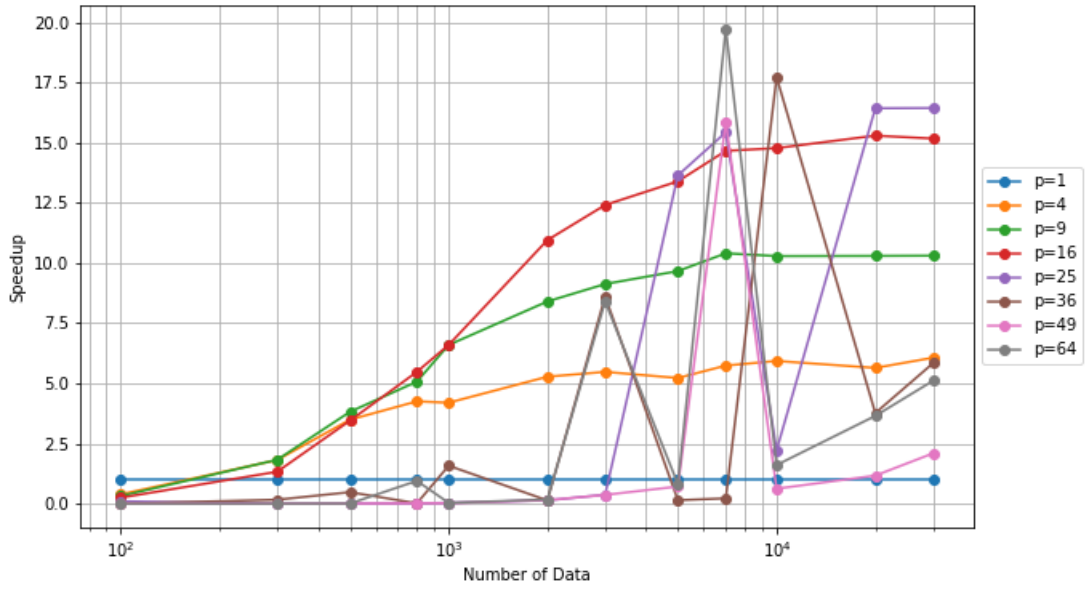

Figure 12: Speed Up vs Number of Data when d = 0.5

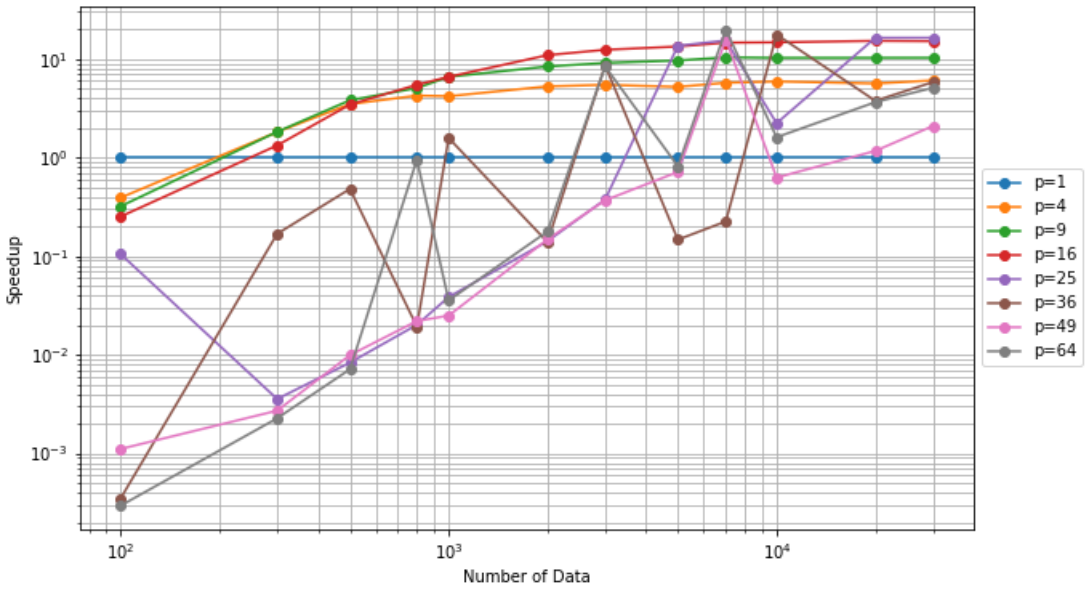

Figure 13: Speed Up vs Number of Data when d = 0.5 Log Scale

Figure 14 and 15 show speed up vs different number of P when d = 0.5. Speed up would increase with number of processor unitl p = 16. The communication times between nodes will offset the benefits of multiprocessors.

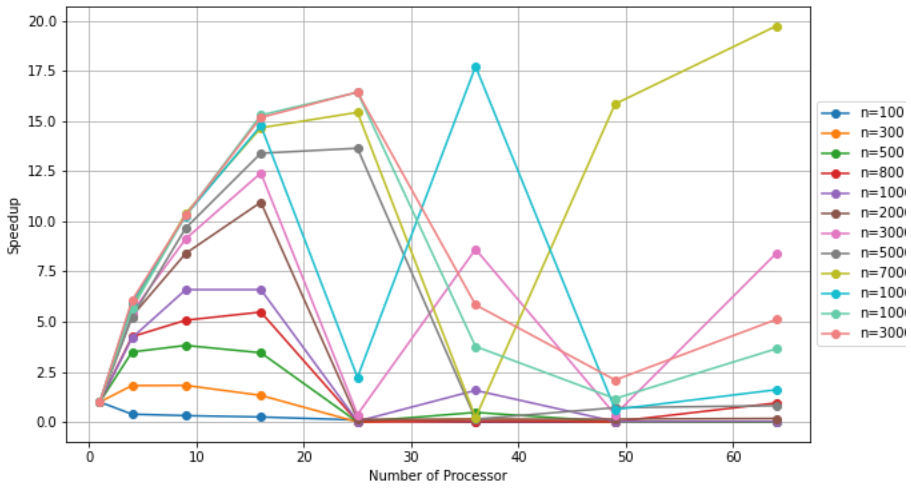

Figure 14: Speed Up vs p at Different P when d = 0.5

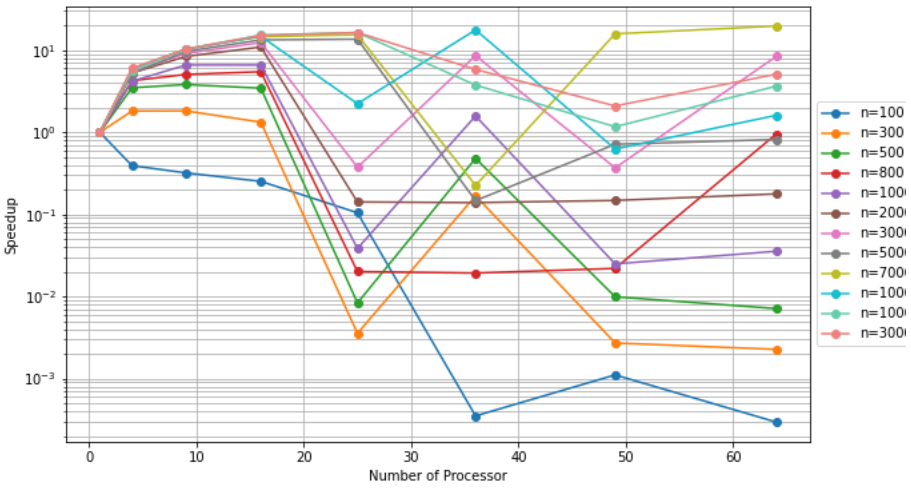

Figure 15: Speed Up vs P at Different P when d = 0.5 Log Scale

Efficiency is generally decrease as p increase and this is what we observes in Figure 16 and Figure 17. One interesting fact we found is that the efficiency go above 1 when p is 4. This is because the sequential algorithm we implemented is not the best one possible. This

is good to know that we can further improve our sequential algorithm and implementation details.

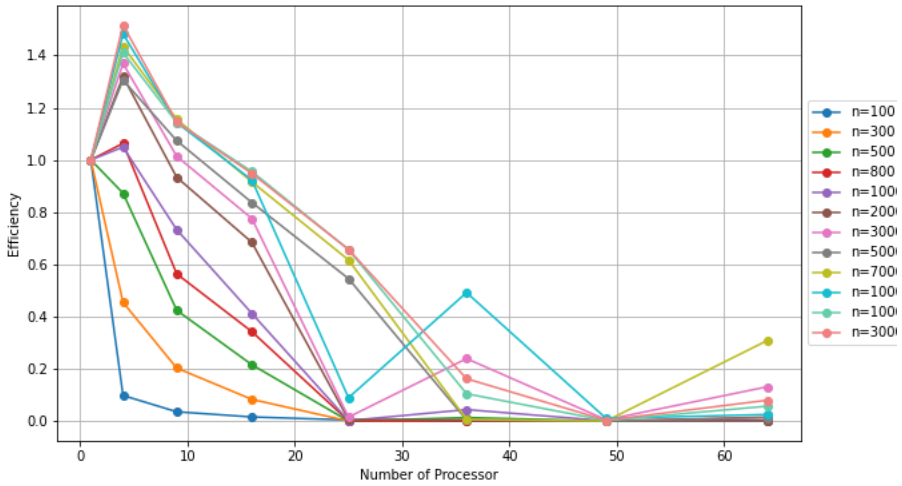

Figure 16: Efficiency vs Processor with Different Input when d = 0.5

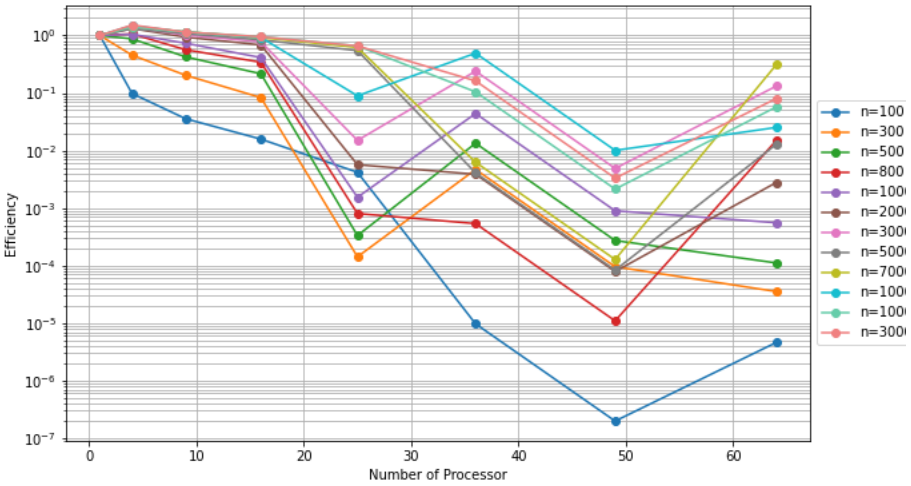

Figure 17: Efficiency vs Processor with Different Input when d = 0.5 Log Scale

Figure 18 and 19 shows the Efficiency vs Input Data with Different P. We expect that the Efficiency increase with input data size because it has similar result as we saw in the speed up. As we can see from the plot, P = 4 has the highest efficiency and overall good run-time.

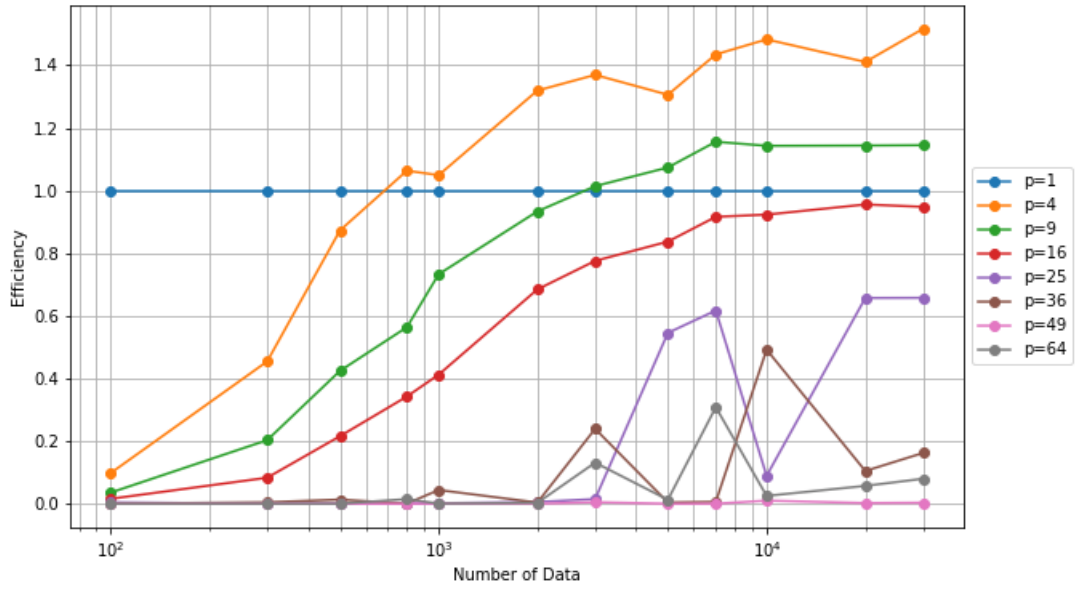

Figure 18: Efficiency vs Input Data with Different P when d = 0.5

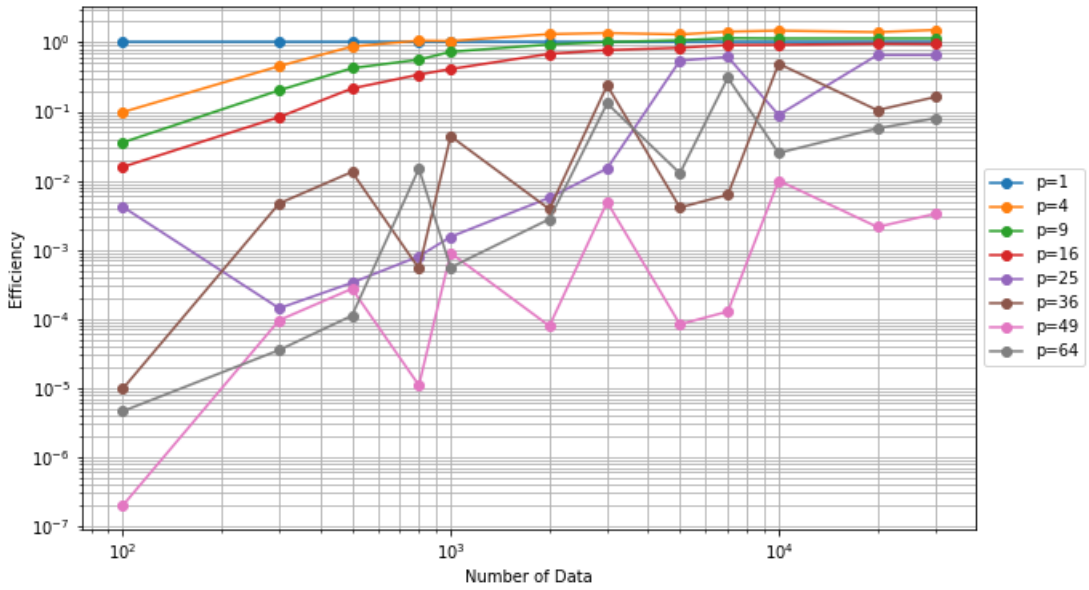

Figure 19: Efficiency vs Input Data with Different P when d = 0.5 Log

3.5 Additional Analysis for Fixed Data Size

In this part, we discuss several situation of running time, different number of processors,

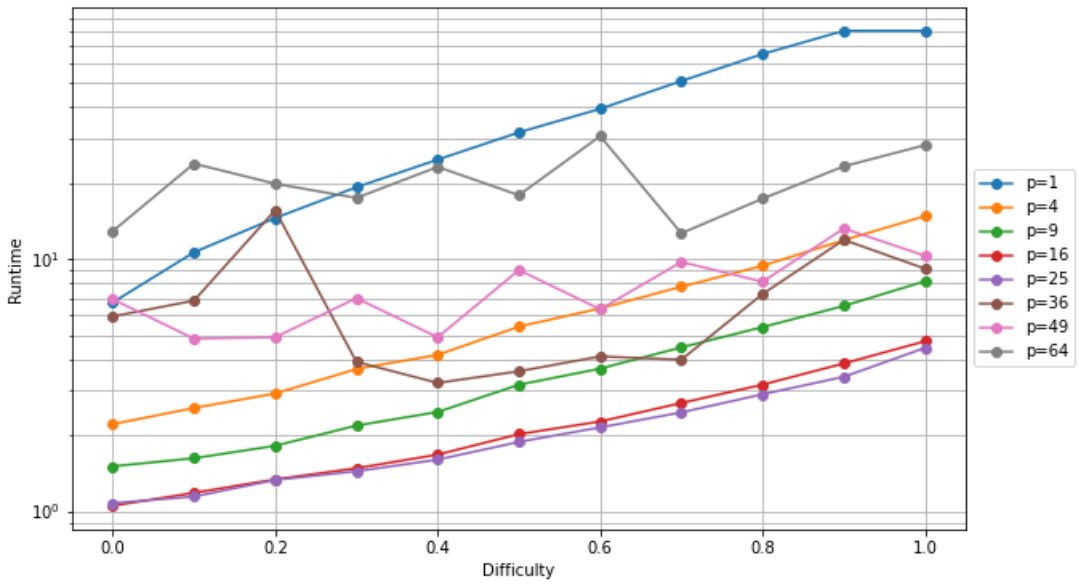

Figure 20: Runtime vs Difficulties at Different P for N = 20000

As we can see in this figure, when number of processor is smaller than 35, the shape of curves are close to linear, with increase of difficulty, the lines are upward. When number of processors over 25, the curve starts to become unstable, the curve will swing up and down. In our cases, when p equals to 25, we gains the lowest running time.

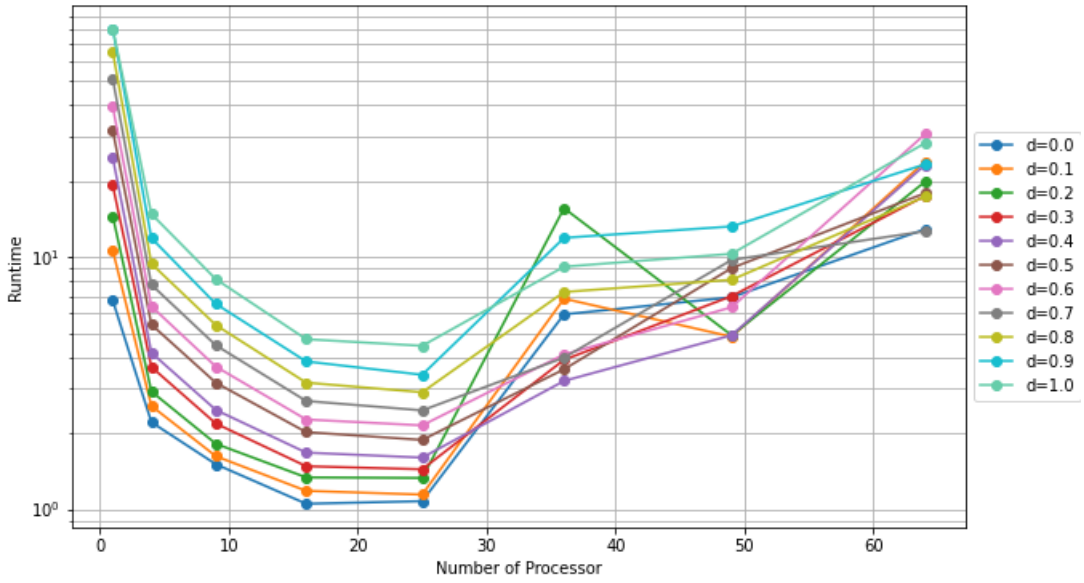

Figure 21: Runtime vs Number of Processors at Different d for N = 20000

In this graph, we could find that when number of processor is smaller than 25, the relation between running time and d is more recognizable, the curve with small value of d is in the bottom while the big value of d is on the top. But when number of p over 25, the pattern of curves with value of d is smaller than 0.3 are becoming unstable. Potential reason could be that with a large amount of processors, the communication and computation time among each processor may be delayed, which cause more running time.

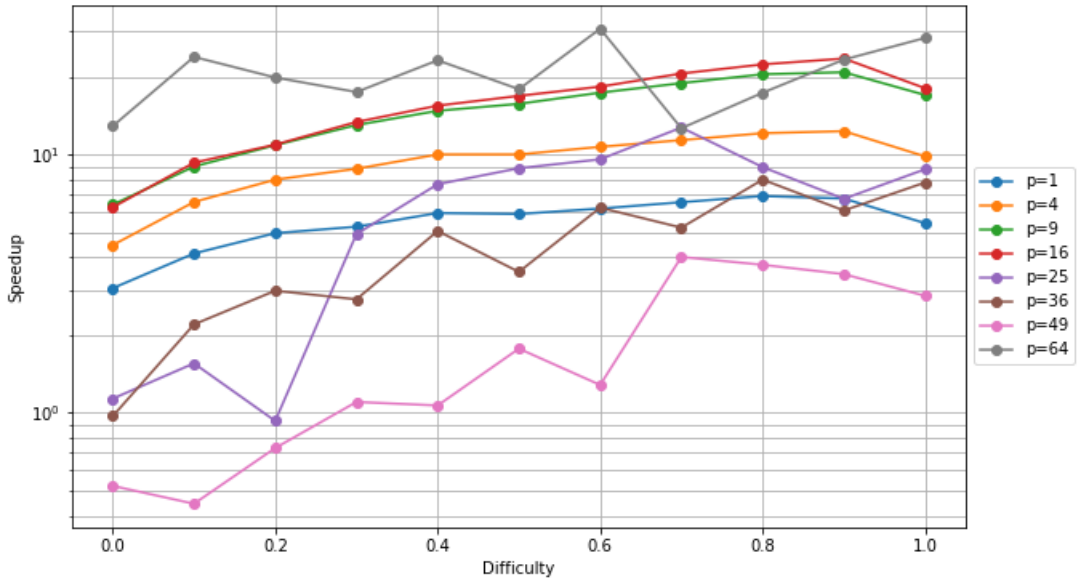

Figure 22: Speed Up vs Difficulty at Different Number of P when N = 20000

This figure shows the relation between difficulty and speed up with various number of processors. When number of p small or equals to 16, the curves are stable, with fix number of p, the increase of difficulty will brings more running time. But when number of p is over 16, the distribution of curves becomes more disordered. Since large amount of processors need the communication time, and and network between those processors will also influence the running time. For the overall trend, we can find that running time extends with increasing difficulty.

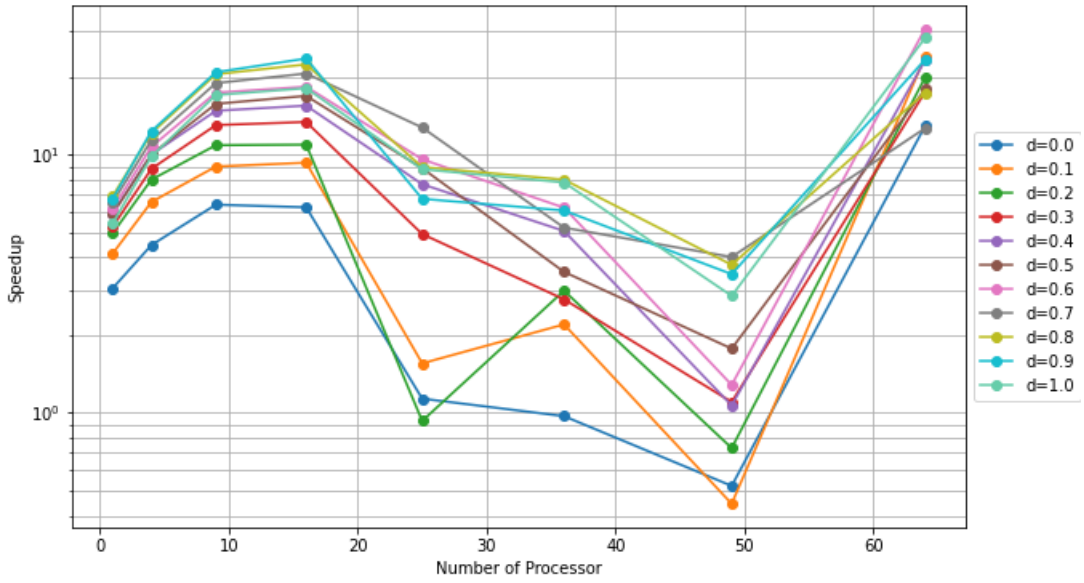

Figure 23: Speed Up vs Different Number of P at Distinct d when N = 20000

This graph express the relation between speed up and number of p at distinct d when N is fixed. In the graph, we could find two peaks exists in both curves. When number of p is close to 18, the speed up reaches the first peak, and when number of p is over 60, the speed up reaches the second peak. This follows the rule that the relation between speed up and number of processors is not strictly positively related to.

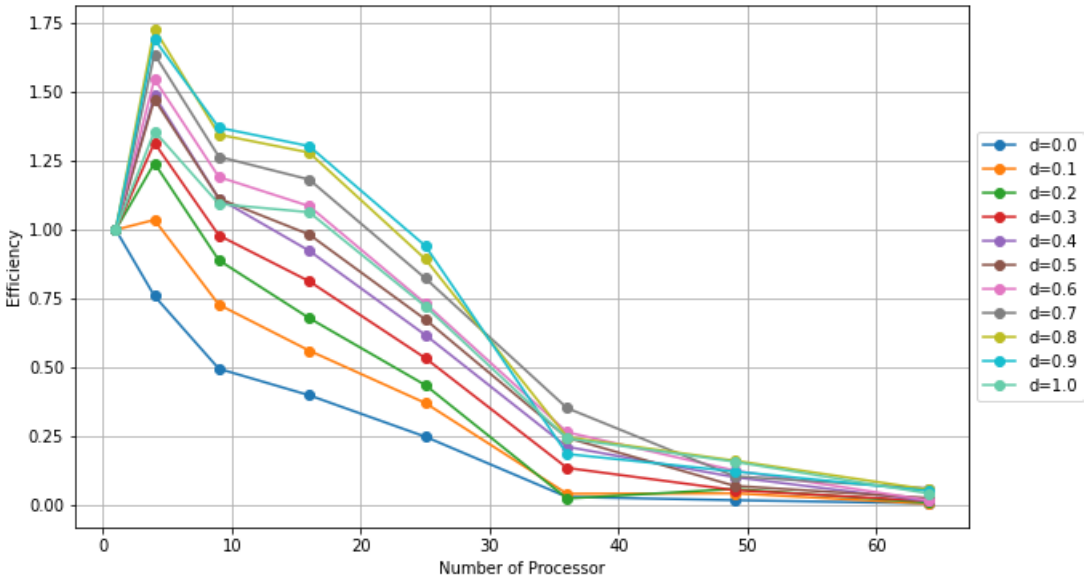

Figure 24: Efficiency vs Number of Processor with Different d for N=20000

This figure shows the straight forward relations between efficiency and number of processors. With increase of number of processors, the efficiency is gradually decreased, which follow the rules and definition of efficiency. With fixed number of processors, the bigger value of d has the higher efficiency.

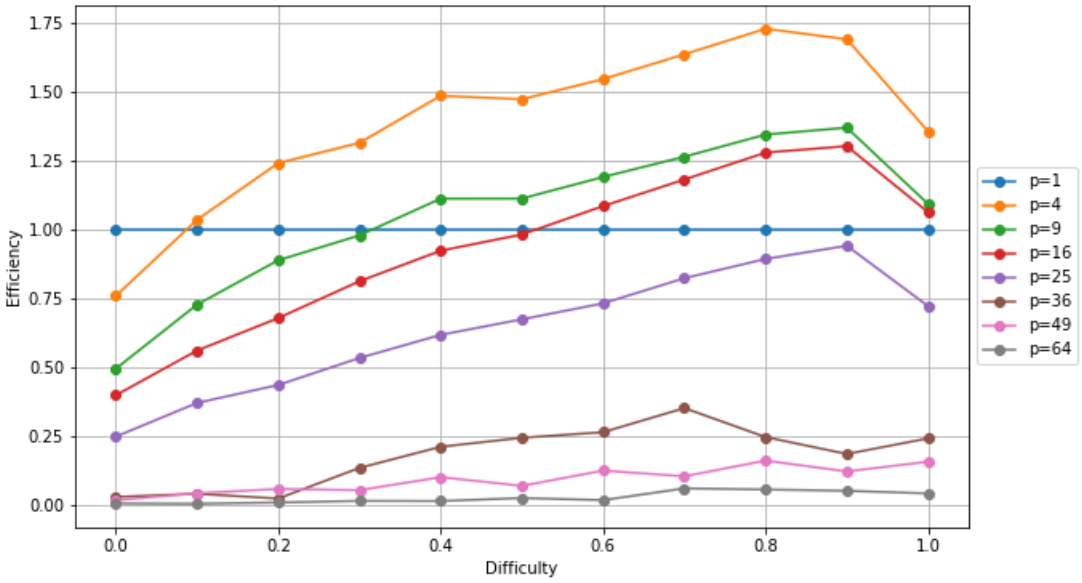

Figure 25: Efficiency vs Difficultly with Different P for N=20000

In this figure, we could find that when number of p is 1, the efficiency is always 1, which follows the definition of efficiency. The efficiency decreased with the increases of number of processors. However, some cases the efficiency is bigger than 1, and potential reason is that our serial programming is not the optimist.

4 Conclusion

In this project, we implemented both sequential and parallel Jacob's method for solving a linear system. The sequential version of Jacob's method has the linear runtime vs input size. For the parallel algorithm, the runtime will start to decrease until it reaches the optimal runtime at p = 16 a when n = 20000 as we discussed in the previous sections.

5 Possible Optimization

Since input data size parameter is n and the result input matrix would be n^2. This 2 power realtion would make input matrix extermely large even if n is small. For example, when n = 1000, n^2 would be 1 million. When we have large n such as 30000, the reult matrix would be 900,000,000 number of elements. If the matrix is sparse, then most of it would be zero, we could come up with some compression scheme to save storage. The diagnoal matrix D has all zeros except for diagonal. So we can create an array to store only the diagonal elements.

6 Reference

[1],[2],[3] CSE 6220 Introduction to High Performance Computing Spring 2020 Programming Assignment 3