Parallel Algorithm - Solve system of linear equations with Jacobi method with MPI

D'Souza, Ajay
ajaydsouza@gatech.edu

May 23, 2017

Abstract

Implement Parallel Algorithm to solve a system of linear equations us the Jacobi method

Contents

List of Figures

1 Algorithm Description

1. We are given a square matrix of size n and p the available processors where p is a perfect square. We will embed a logical square matrix of size $\sqrt{p} \times \sqrt{p}$ on to the hypercubic processors p using `MPI_Cart_Create`.

2. A property of this embedding is that the processors in a row or a column of the logical $\sqrt{p} \times \sqrt{p}$ matrix are hypercubic connected processors

3. The $(0,0)$ processor or the rank 0 processors reads the input matrix A,b and has to block distribute it to the other relevant processors

4. The processors in the first column of the matrix have to get the vector b block distributed to them from the rank 0 processor in that column. This can be done by a `MPI_Scatterv` along the first column with the count and displacement arrays set to the appropriate $\lceil \frac{n}{\sqrt{p}} \rceil$ or $\lfloor \frac{n}{\sqrt{p}} \rfloor$ offset and count depending on if the rank of the processor is $< (n \mod \sqrt{p})$ or not. The complexity of this operation is $\mathcal{O}(\tau \log \sqrt{p} + \mu n)$

5. Next we have to distribute the $n \times n$ matrix A from processor of rank 0 to every processor in the matrix so each processor has a local A block matrix of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$.

6. This is implemented using `MPI_Scatterv`. The count and displacement arrays let us provide varying size of rows and columns that need to be block distributed to a processor based on its rank and of n not being a multiple of $\sqrt{p}$. Matrix A is stored as an one dimensional array and non contiguous elements from the array would need to be scattered. This can be accomplished with creating a `MPI_Type_vector` which takes as input a count,block length and stride

7. Since this involved scattering the elements along column and then along rows of processors which are hypercubic the run time complexity is

$$\begin{aligned} &= \mathcal{O}(\tau \log \sqrt{p} + \mu n^2) + \mathcal{O}(\tau \log \sqrt{p} + \mu \frac{n^2}{\sqrt{p}}) \\ &= \mathcal{O}(\tau \log \sqrt{p} + \mu n^2) \end{aligned} \tag{1}$$

8. Once the processors in the first column receive the block distributed rows of matrix A, those processors can keep an array local $local_D$ of $\frac{n}{\sqrt{p}}$ elements from the block distributed $local_A$, which correspond to the diagonal elements of the $n \times n$ matrix A.

9. Or these diagonal elements of the $n \times n$ matrix A, could also be got by the processors in the first column, from the local block matrix A $local_A$ in the diagonal processors in the row of each column (i,i). The sending and receiving can be performed by `MPI_Send` and `MPI_Recv` for a complexity of $\mathcal{O}(1)$

10. Finally to perform the matrix multiplication Ax, each processor in column i would need to have the bock distributed $local_x$ vector of size $\frac{n}{\sqrt{p}}$ from the i^{th} processor in the first column

11. This transpose broadcast of local x can be performed using the following steps

    (a) Each ith processor in the left column will send its local $local_x$ array of size $\frac{n}{\sqrt{p}}$ to the diagonal processor (i, i) in that row using `MPI_Send` and `MPI_Recv`, this is of complexity $\mathcal{O}(1)$

    (b) The diagonal processor (i, i) in every row will then broadcast these $\frac{n}{\sqrt{p}}$ local $local_x$ elements to each processor in its column using `MPI_Bcast`, this is of complexity $\mathcal{O}(\tau + \mu\frac{n}{\sqrt{p}}) \log \sqrt{p}$

12. Now each processor in the processor matrix of size $\sqrt{p} \times \sqrt{p}$ has a block allocated local A matrix $local_A$ and a local x vector $local_x$ and can perform a matrix vector multiplication

13. Matrix multiplication is performed in each processor with a complexity of $\mathcal{O}((\frac{n}{\sqrt{p}})^2)$

14. The multiplication results in a local y $local_y$ vector of size $\frac{n}{\sqrt{p}}$ on each processor.

15. Now the corresponding elements of local y, in every processor in a row are added and stored in the local y vector of size $\frac{n}{\sqrt{p}}$ of the processor in the first column in each row using. This operation can be performed with `MPI_Reduce` with a complexity of $\mathcal{O}(\tau + \mu\frac{n}{\sqrt{p}}) \log \sqrt{p}$

16. The processors in the first column now have the local $local_y$,$local_D$ and $local_b$ arrays of size $\frac{n}{\sqrt{p}}$. Then can now compute the the new x as $x_i = \frac{(local_{b_i} - (localy_i - local_{D_i} * local_{x_i}))}{local_{D_i}}$

17. The new x computed in the processors in the first column in each row is stored in the $local_x$ vector on that processor

18. Now this new $local_x$ is transpose distributed back from the i^{th} processor in the first row to the processors in the i^{th} column using transpose broadcast as described earlier

19. With the new $\frac{n}{\sqrt{p}}$ local $local_x$ values at each processor will perform another matrix multiplication $local\_y = local\_Alocal_x$ as described above.

20. The results from local y from the processors in a row are summed up back to the first processors in that row as described earlier

21. Each processor in the first column can then compute the 2 norm of $|local_b - local_y|$

22. We can then do a `MPI_AllReduce` to sum the local 2 norm of all processors in the grid to get the 2 norm of $|b - AX_{new}|$. The value of the 2 norm for the whole matrix will now be available on each processor in the matrix.

23. This way every processor can check if the 2 norm is within the tolerance specified and exit the loop in the same way as other processors. This step of computing the norm using `MPI_AllReduce` should be of complexity

$$\begin{aligned} &= \mathcal{O}(\tau + \mu)\log\sqrt{p} + \mathcal{O}(\tau + \mu)\log\sqrt{p} \\ &= \mathcal{O}(\tau + \mu)\log\sqrt{p} \end{aligned} \tag{2}$$

24. If the 2 norm of $b - AX_{new}$ is beyond the tolerance specified and if we have still not reached the cut off iteration then each processor will continue the iteration

25. On continuing the iteration the first column processors already have the value of the new $local_y$ from the last AX_{new}.So the first column processors can compute the new $local_{x_{new}}$ and then transpose broadcast it as described earlier and continue till convergence

26. On termination of the iteration, only the processors in the first column need to send their $local_x$ values to the rank 0 processor. This can be done using `MPI_Gather`. This takes $\log\sqrt{p}$ steps and since we start by sending message of size $\frac{n}{\sqrt{p}}$ till we reach $\frac{n}{2}$. So the complexity of this operation will be $\mathcal{O}(\tau\log\sqrt{p} + \mu n)$

2 Performance Plots

The algorithm was executed for the following parameters

- size n = 500,1000,2000,4000,8000,10000,12000,14000,16000,16000,20000,25000
- Difficulty d = 0.5,0.75,0.9,0.95
- Number of parallel processors p=4,9,16,25

2.1 Runtime Performance Plots

The following are the Run Time plots

2.1.1 Runtime Performance for Serial Algorithm P=1

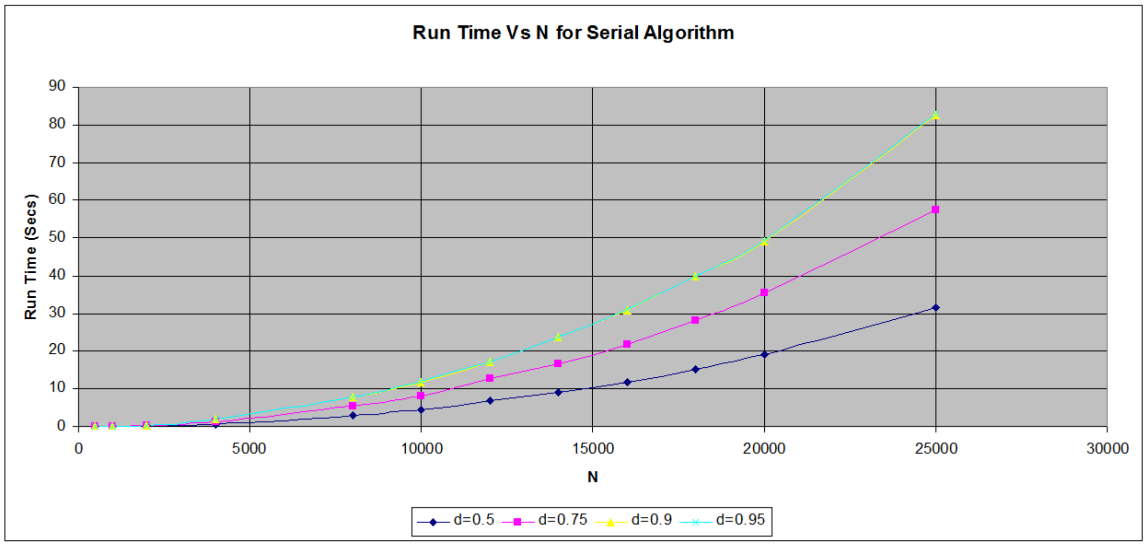

Figure 1: Serial Runtime Vs N

2.1.2 Runtime Performance - Plot of Run Time Vs N for different levels of difficulty for fixed No of Processors P

The following is the performance plot of Runtime Vs N the matrix dimension for different levels of difficulty for a fixed number of processors.

1. P=4 (2)

2. P=9 (3)

3. P=16 (4)

4. P=25 (5)

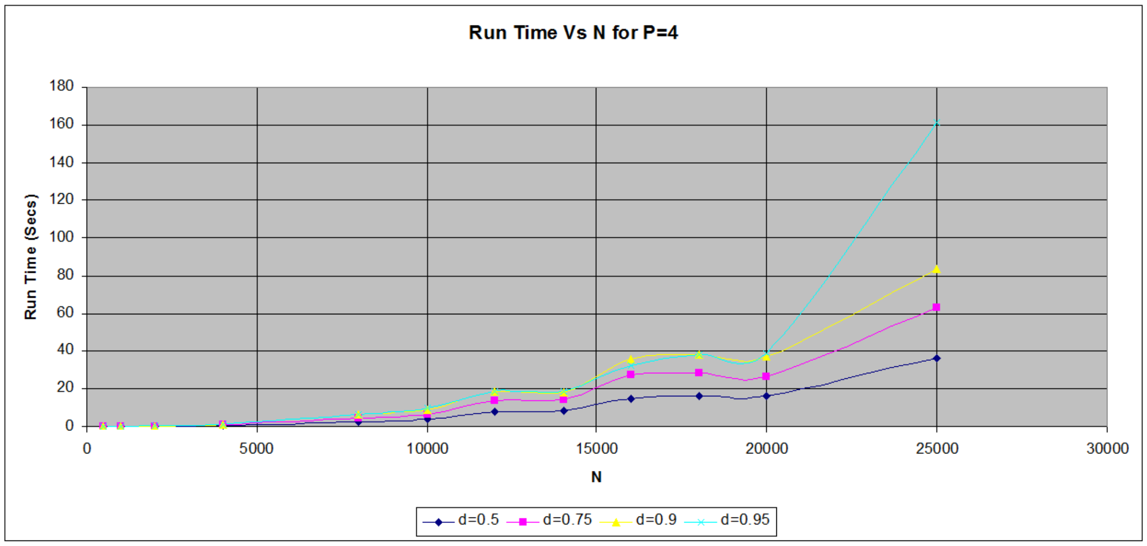

Figure 2: Run Time Vs N for P=4

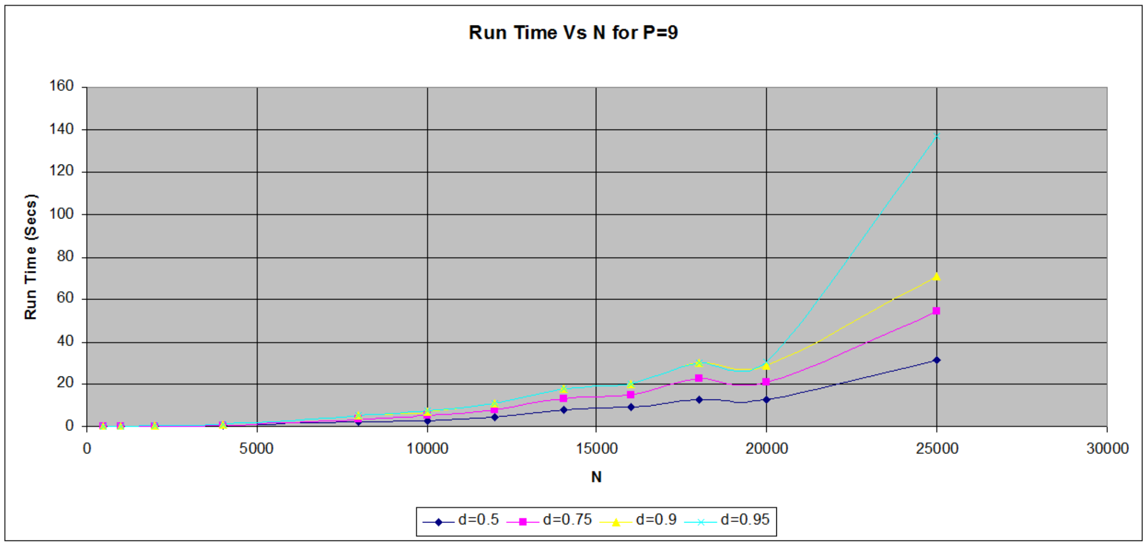

Figure 3: Run Time Vs N for P=9

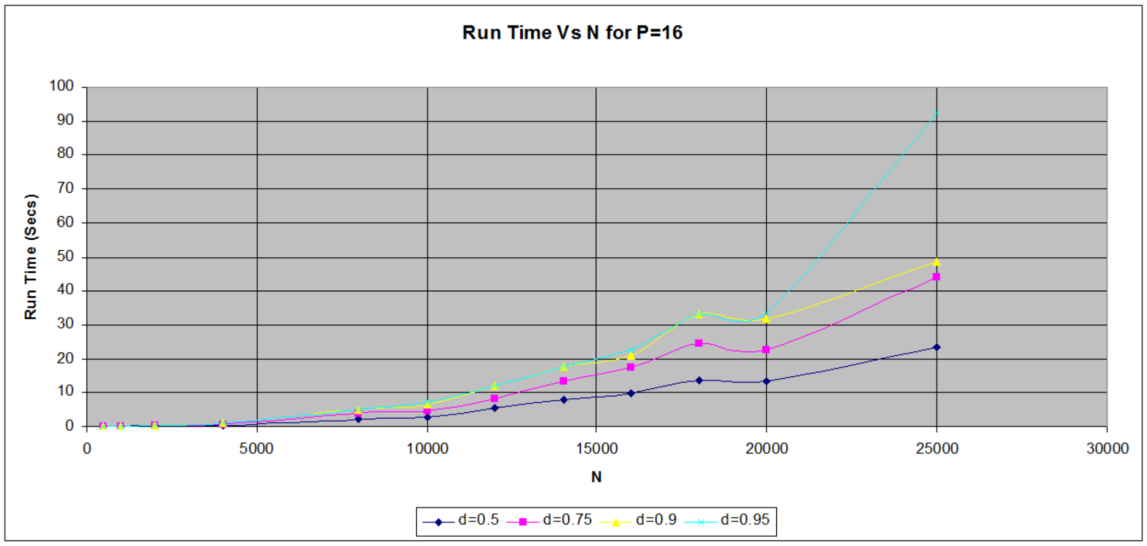

Figure 4: Run Time Vs N for P=16

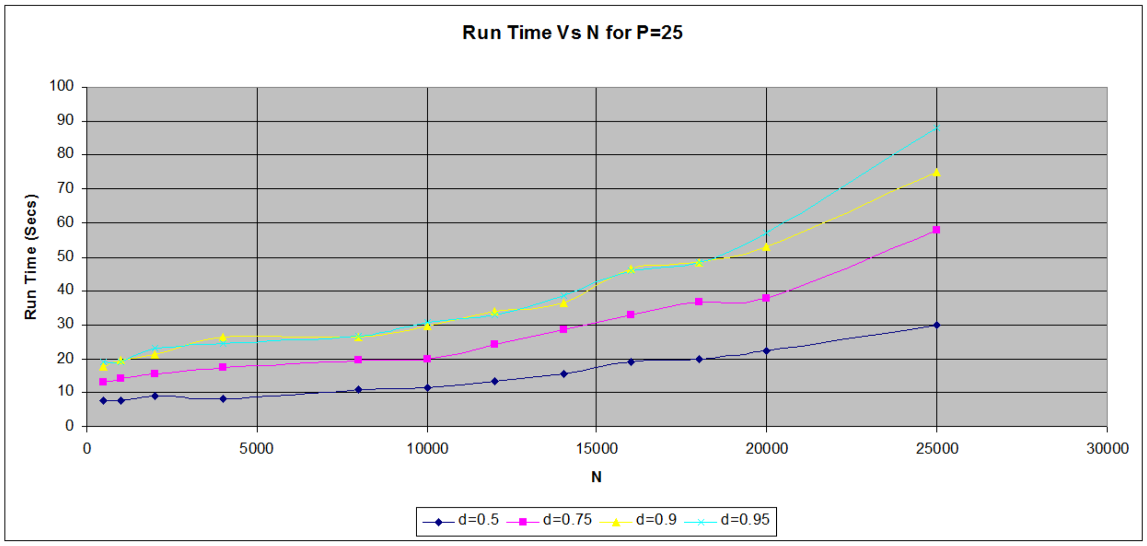

Figure 5: Run Time Vs N for P==25

2.1.3 Runtime Performance - Plot of Run Time Vs Number of Processors P for different levels of difficulty for fixed size N

The following is the performance plot of Run Time Vs Number of Processors P for different levels of difficulty for fixed matrix size N

1. N=1000 (6)
2. N=8000 (7)
3. N=16000 (8)
4. N=20000 (9)
5. N=25000 (10)

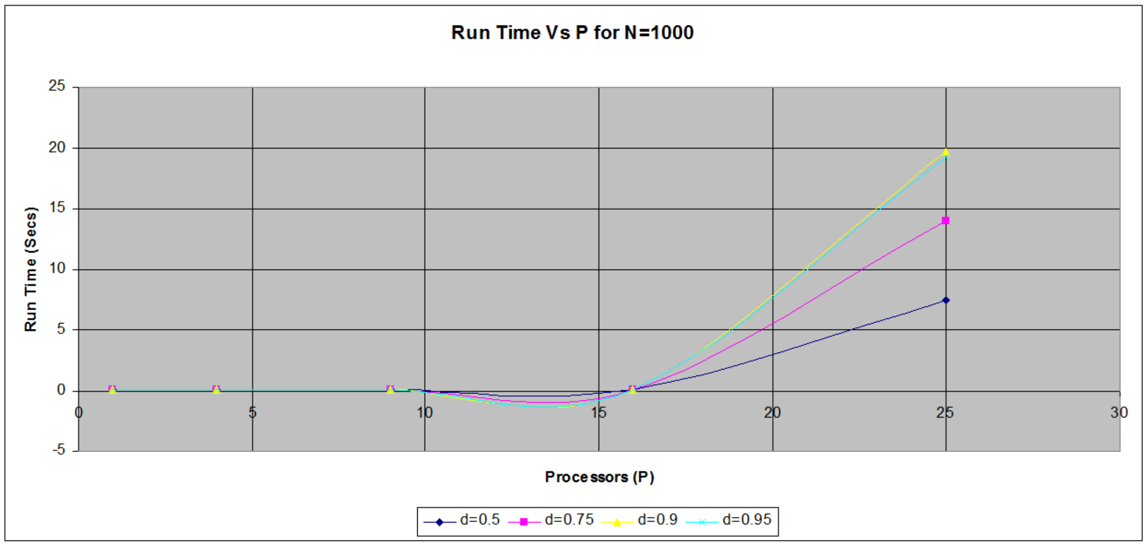

Figure 6: Run Time Vs P for N=1000

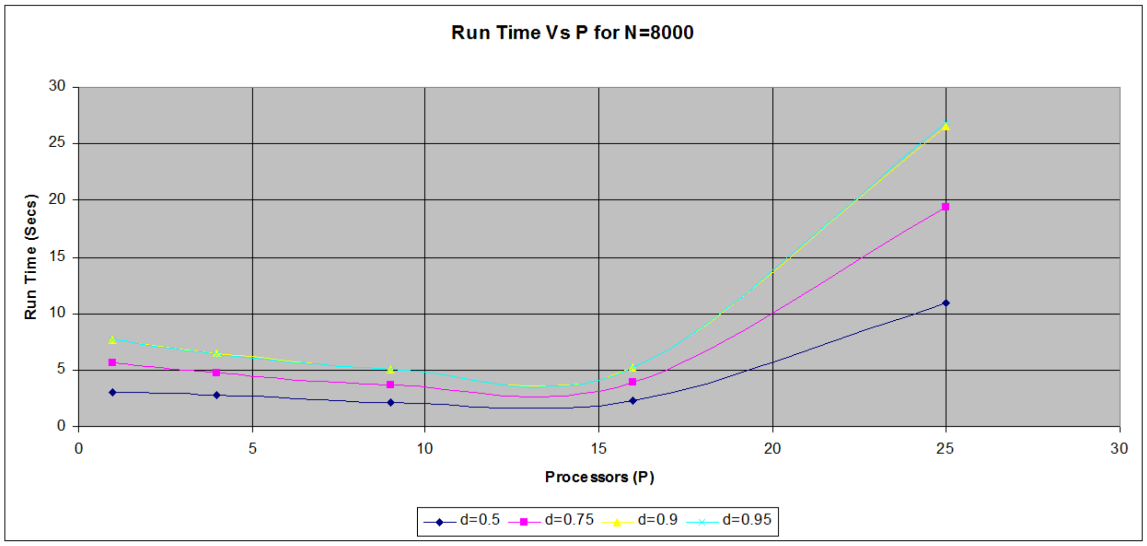

Figure 7: Run Time Vs P for N=8000

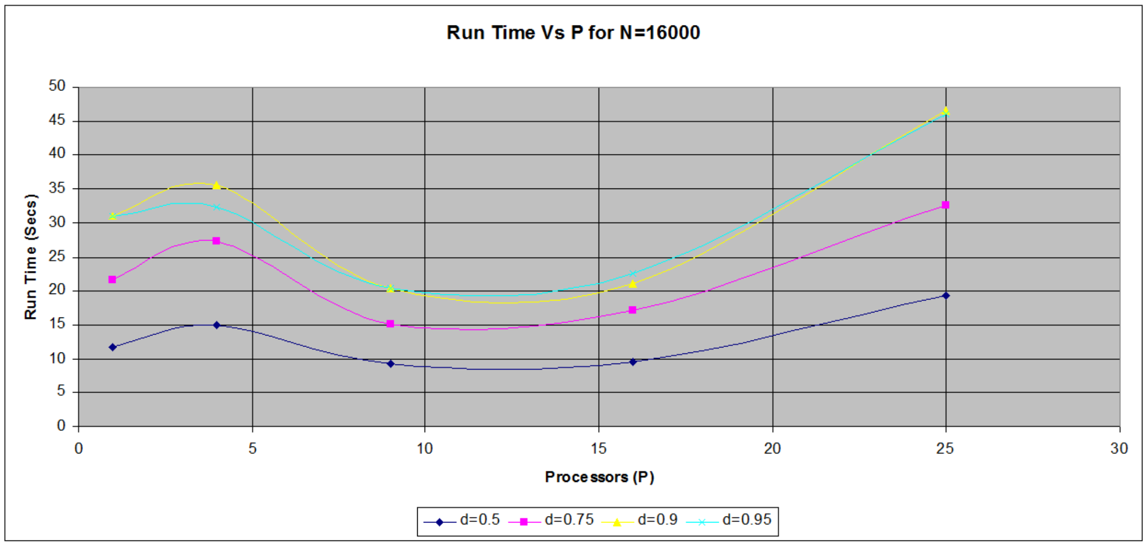

Figure 8: Run Time Vs P for N=16000

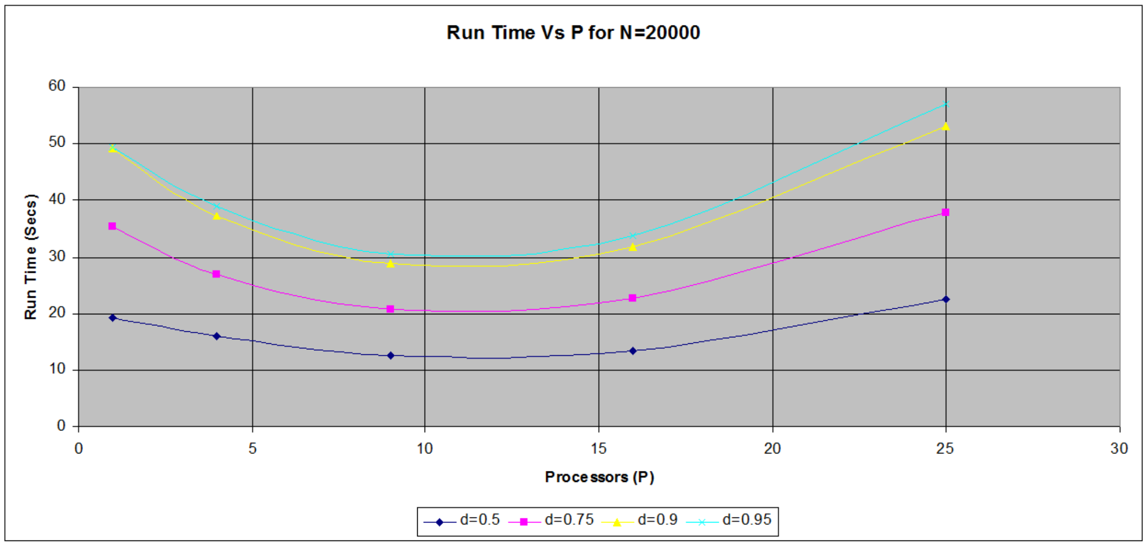

Figure 9: Run Time Vs P for N=20000

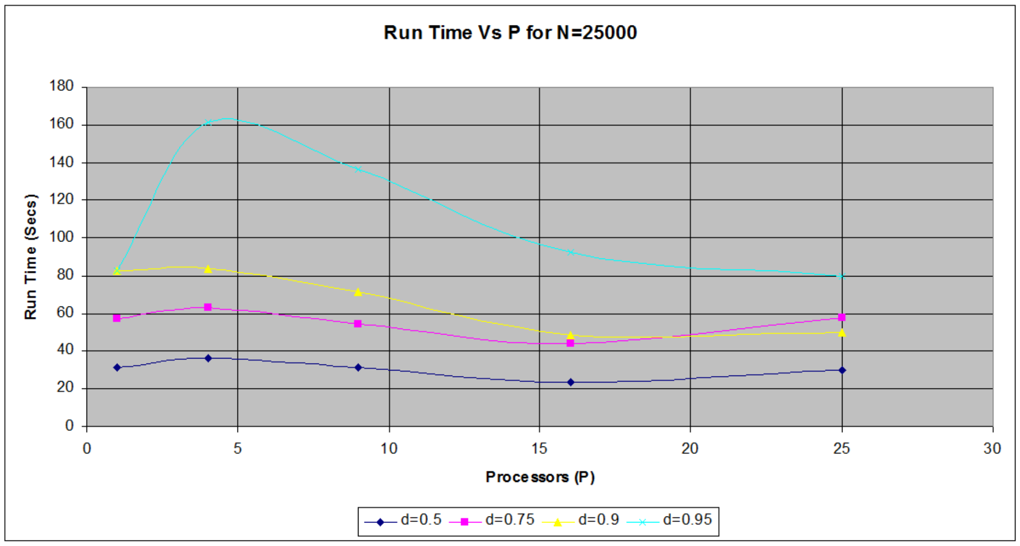

Figure 10: Run Time Vs P for N=25000

2.1.4 Runtime Performance - Plot of Run Time Vs Number of Processors P for different N for fixed Difficulty d

The following is the performance plot of Run Time Vs Number of Processors P for different N for fixed Difficulty d

1. d=0.5 (11)
2. d=0.75 (12)
3. d=0.9 (13)
4. d=0.95 (14)

2.2 SpeedUp Plots

The following are the SpeedUp Ratio plots

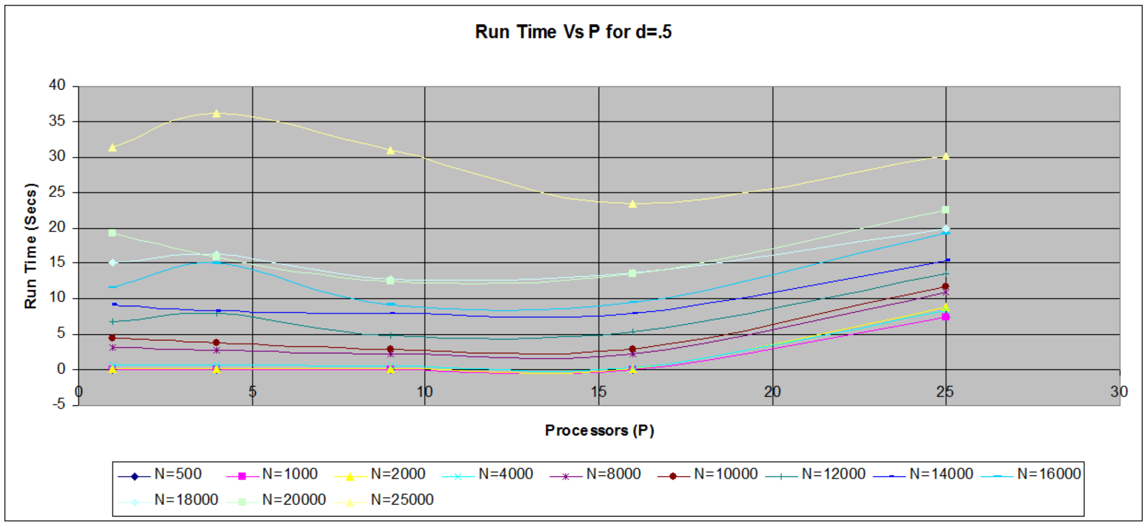

Figure 11: Run Time Vs P for d=0.50

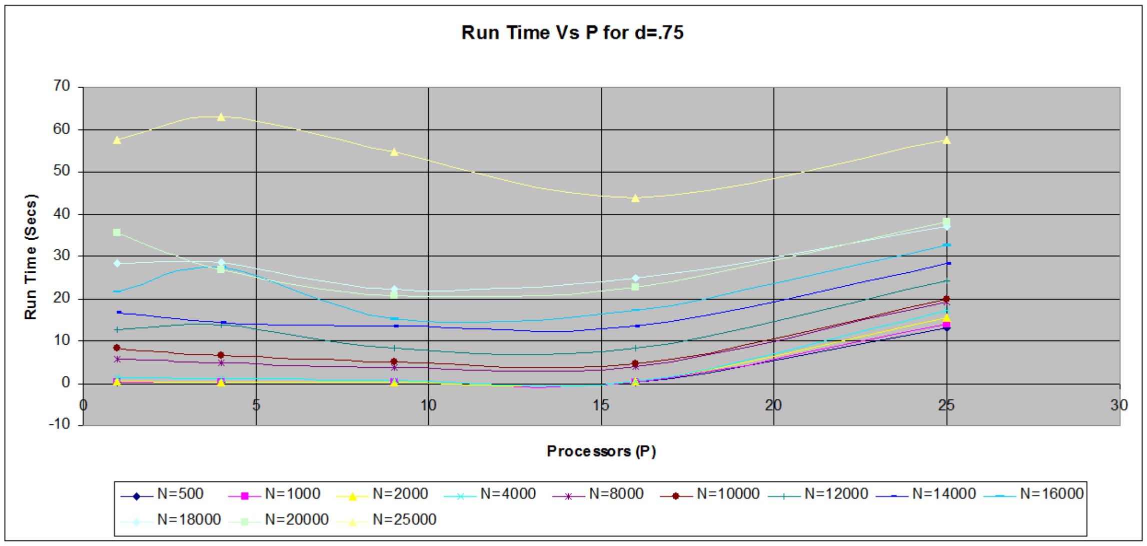

Figure 12: Run Time Vs P for d=0.75

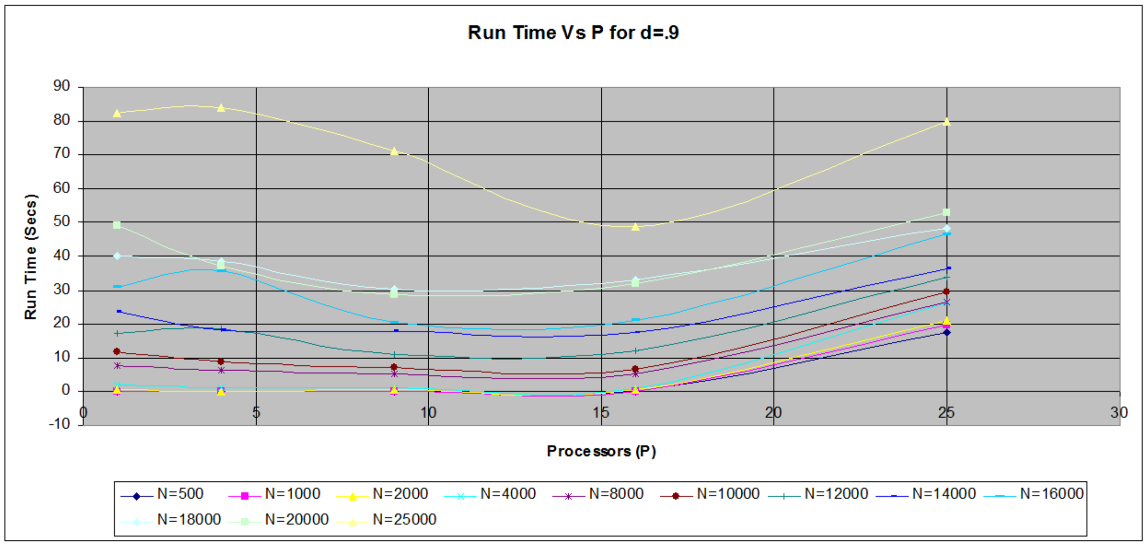

Figure 13: Run Time Vs P for d=0.90

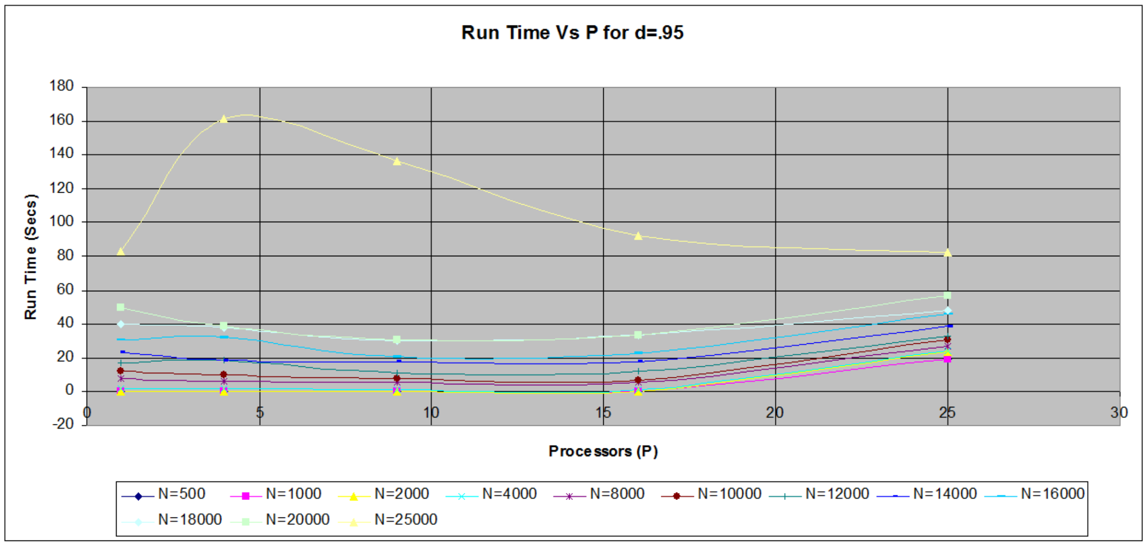

Figure 14: Run Time Vs P for d=0.95

2.2.1 SpeedUp - Plot of SpeedUp Vs N for different levels of difficulty for fixed No of Processors P

The following is the performance plot of SpeedUp Vs N the matrix dimension for different levels of difficulty for a fixed number of processors.

1. P=4 (15)

2. P=9 (16)

3. P=16 (17)

4. P=25 (18)

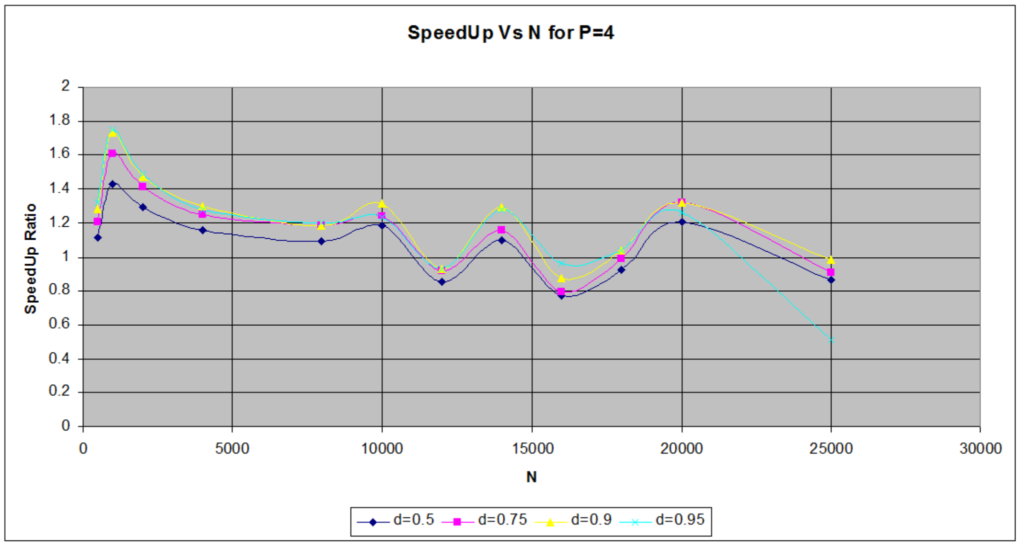

Figure 15: SpeedUp Vs N for P=4

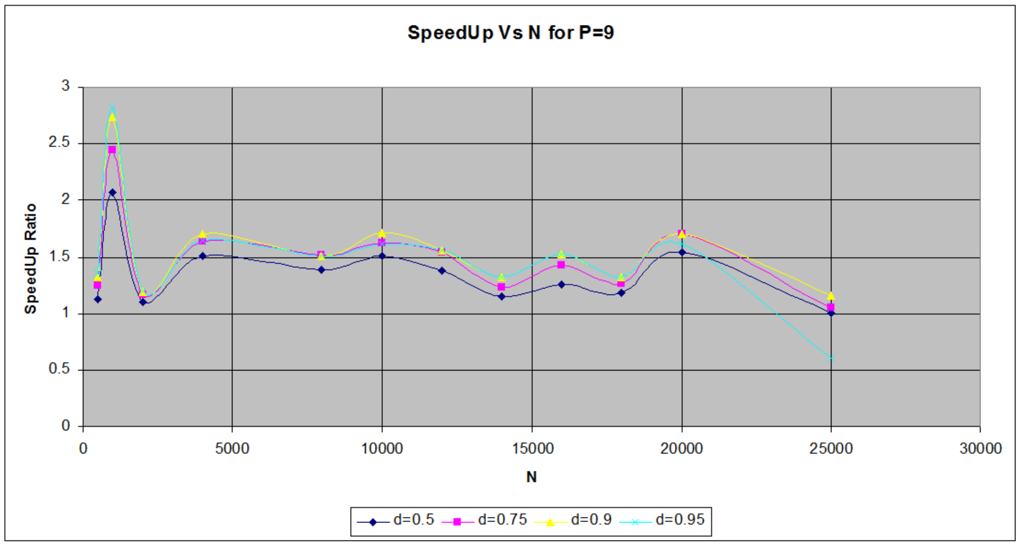

Figure 16: SpeedUp Vs N for P=9

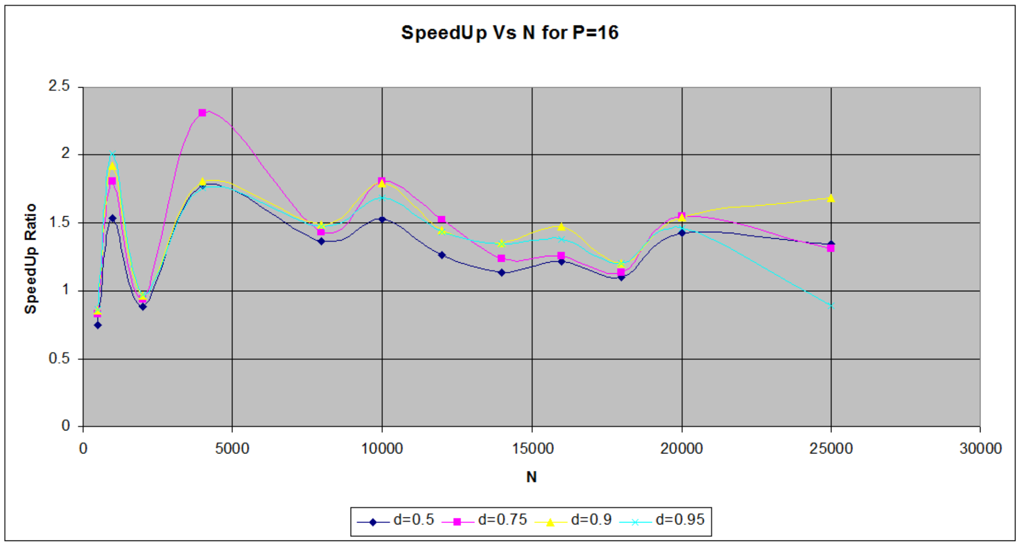

Figure 17: SpeedUp Vs N for P=16

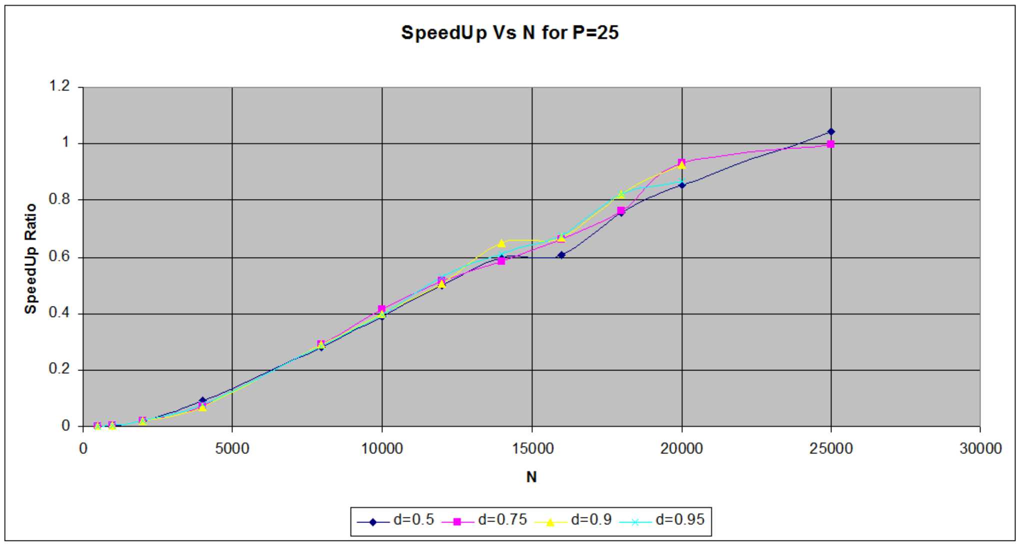

Figure 18: SpeedUp Vs N for P==25

2.2.2 Runtime Performance - Plot of SpeedUp Vs Number of Processors P for different levels of difficulty for fixed size N

The following is the performance plot of SpeedUp Vs Number of Processors P for different levels of difficulty for fixed matrix size N

1. N=1000 (19)

2. N=8000 (20)

3. N=16000 (21)

4. N=20000 (22)

5. N=25000 (23)

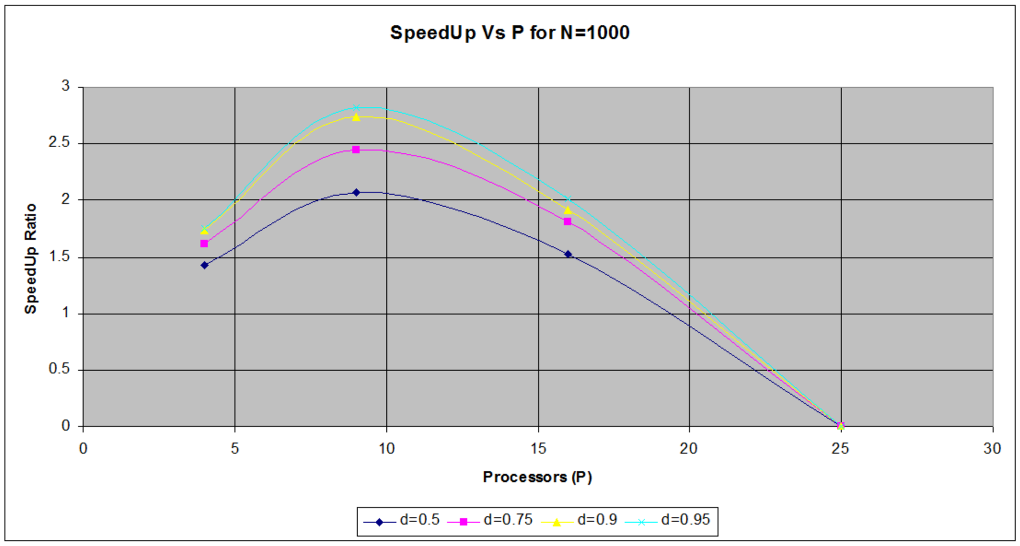

Figure 19: SpeedUp Vs P for N=1000

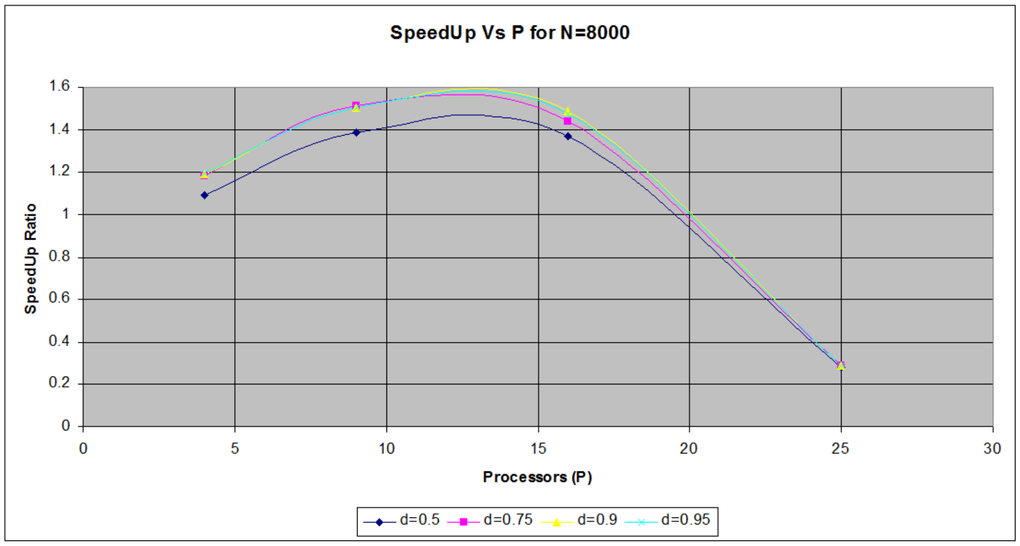

Figure 20: SpeedUp Vs P for N=8000

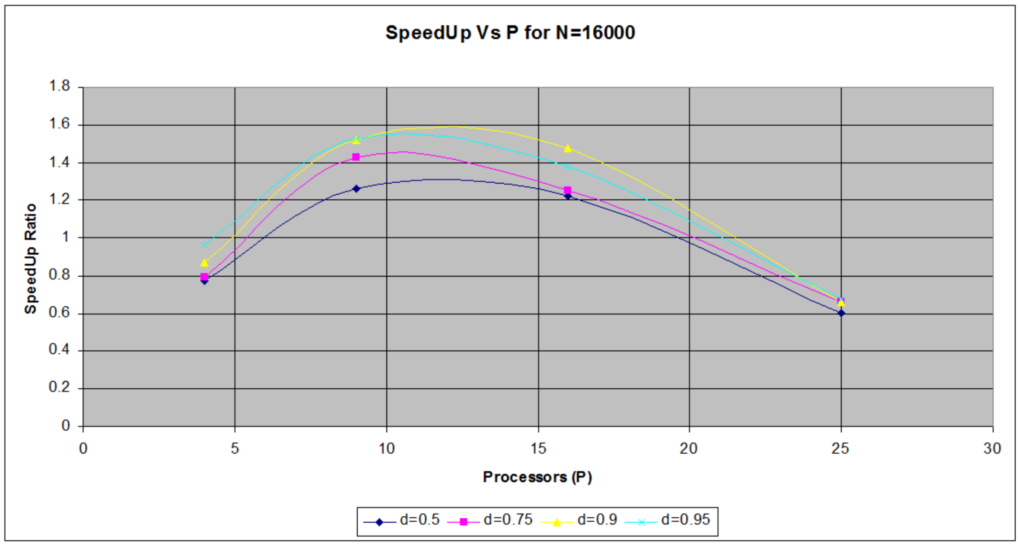

Figure 21: SpeedUp Vs P for N=16000

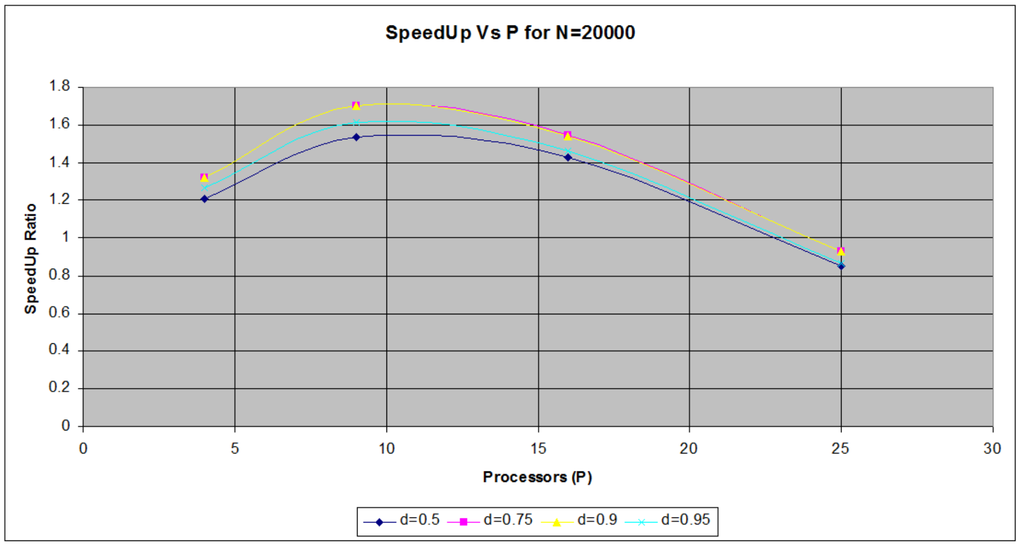

Figure 22: SpeedUp Vs P for N=20000

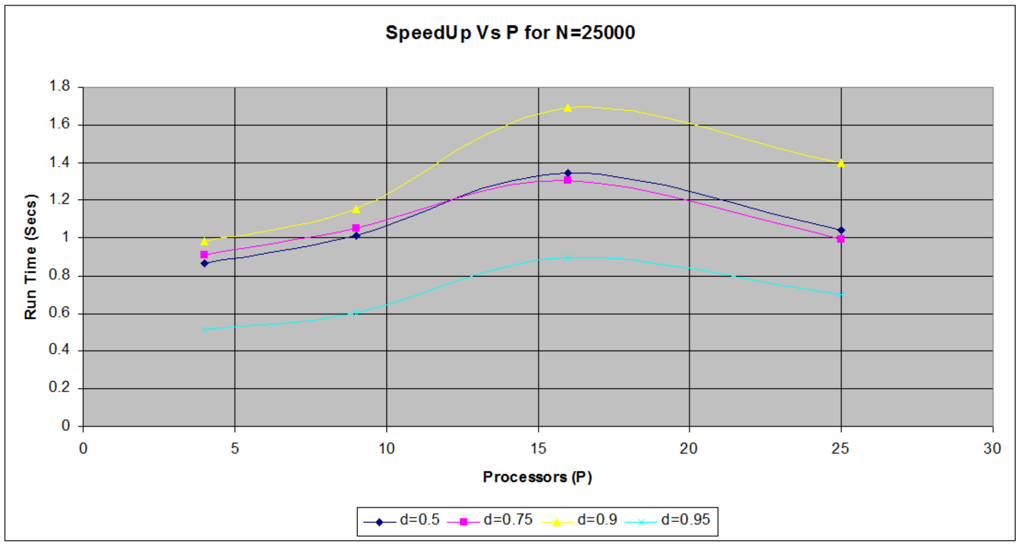

Figure 23: SpeedUp Vs P for N=25000

2.2.3 Runtime Performance - Plot of SpeedUp Vs Number of Processors P for different N for fixed Difficulty d

The following is the performance plot of SpeedUp Vs Number of Processors P for different N for fixed Difficulty d

1. d=0.5 (24)

2. d=0.75 (25)

3. d=0.9 (26)

4. d=0.95 (27)

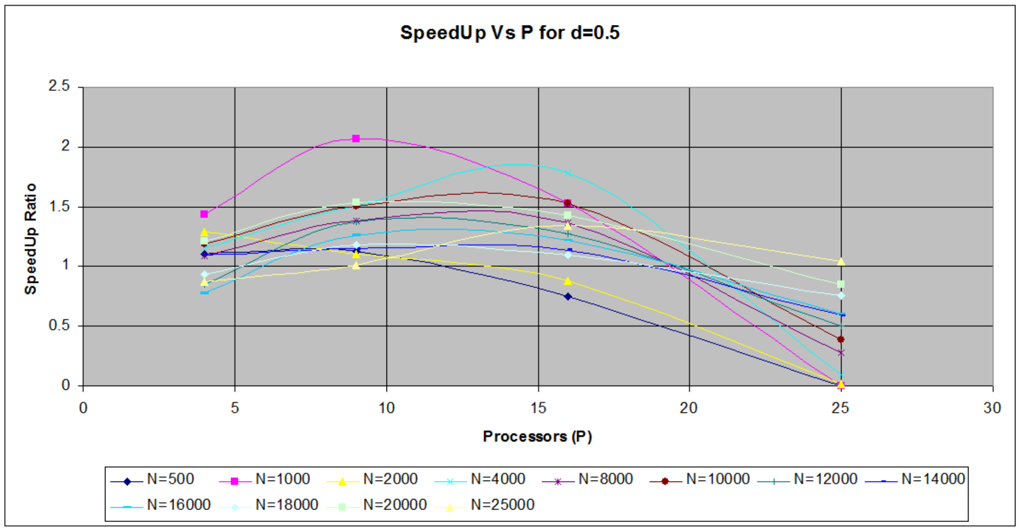

Figure 24: SpeedUp Vs P for d=0.50

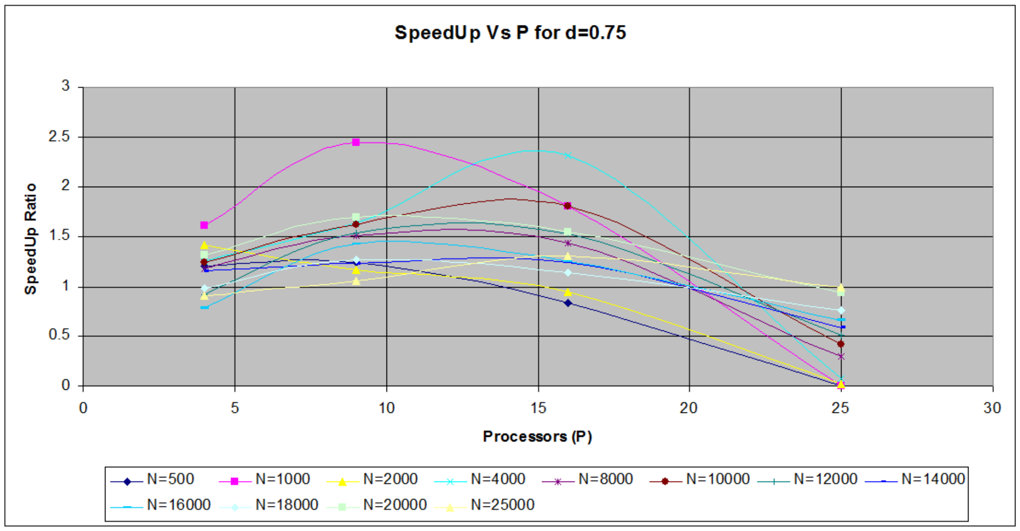

Figure 25: SpeedUp Vs P for d=0.75

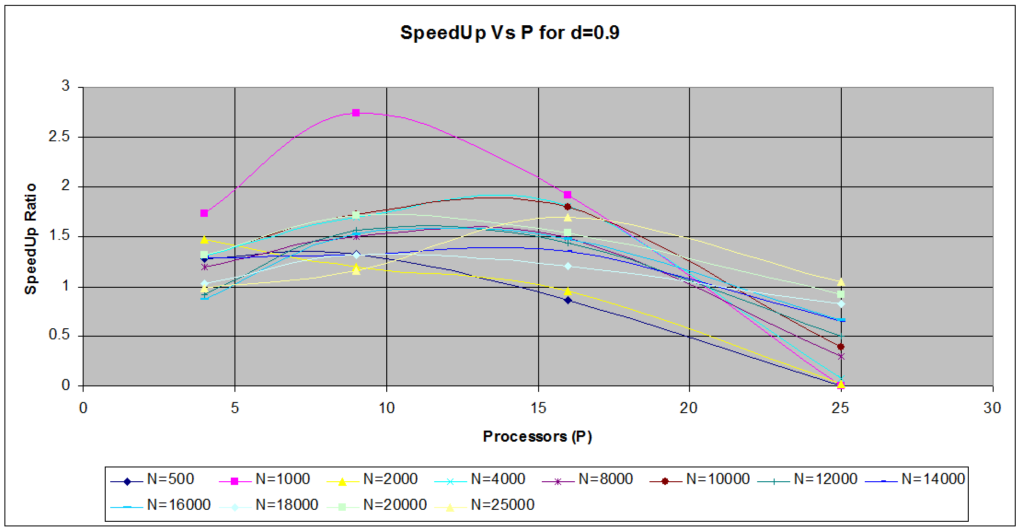

Figure 26: SpeedUp Vs P for d=0.90

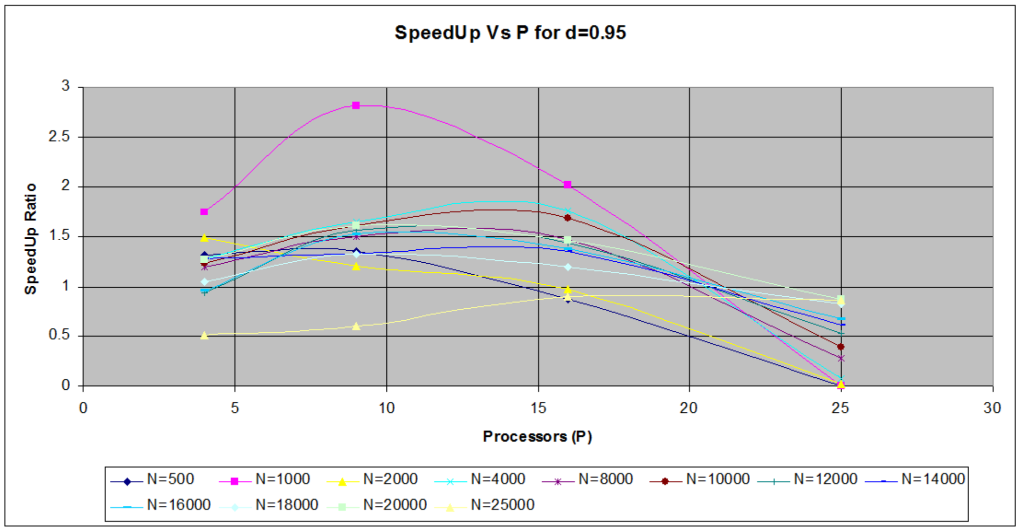

Figure 27: SpeedUp Vs P for d=0.95

3 Conclusions

- The serial runtime of the algorithm is of the order of

$$T(n,1) = iterations * \mathcal{O}(n^2) \tag{3}$$

- The parallel algorithm has a complexity of

$$T(n,p) = iterations * [\mathcal{O}(\frac{n^2}{p})] + iterations * [\mathcal{O}(\tau \log \sqrt{p} + \mu \frac{n}{\sqrt{p}})] + \mathcal{O}(\tau \log \sqrt{p} + \mu n^2) \tag{4}$$

- At very small values of n, the communication overhead given by $iterations * [\mathcal{O}(\tau \log \sqrt{p} + \mu \frac{n}{\sqrt{p}})] + \mathcal{O}(\tau \log \sqrt{p} + \mu n^2)$, especially $\tau \log \sqrt{p}$ would dominate over computation costs. So at very low values of n it might make no sense to try to parallel algorithm as it has no speed up over the serial algorithm. This is borne out by the graphs at values of $n < 1000$
- For the Serial algorithm the run time increases with N and is higher for higher difficulty levels
- For the Parallel algorithm for a given number of fixed processors, the run time
  1. Increases as N is increased
  2. Is higher with increased difficulty
  3. Increases at a higher rate at higher values on N
- For the Parallel algorithm if N is fixed then
  1. Initially for low values of p, the SpeedUp increases as we increase the number of processors p
  2. We have a point of inflexion in the curve where run time is lowest at a certain value of P and then increases again as P is increased. This indicates that for a given problem size of N there is a optimum number of processors P which give the best run time. The optimum value of P for a fixed problem size of N is the value at which the computation cost of the parallel algorithm $iterations * [\mathcal{O}(\frac{n^2}{p})]$ still dominates over the communication cost $iterations * [\mathcal{O}(\tau \log \sqrt{p} + \mu \frac{n}{\sqrt{p}})] + \mathcal{O}(\tau \log \sqrt{p} + \mu n^2)$
  3. It can also seen that the value of the point of inflexion , ie P the number of processors at which runtime is the least, gets higher as the fixed value of N is increased. This indicates that as N is increased the computation cost of the parallel algorithm $iterations * [\mathcal{O}(\frac{n^2}{p})]$ is higher and thus can absorb the larger

communication cost $iterations * [\mathcal{O}(\tau \log \sqrt{p} + \mu \frac{n}{\sqrt{p}})] + \mathcal{O}(\tau \log \sqrt{p} + \mu n^2)$ of a larger N

- Higher the difficulty level higher is the run time. The spread of the run time is higher at higher values of N
- As seen with Run Time, SpeedUp is also fairly stable in a small range but falls off gradually as either N decreases below a threshold or is increased beyond a threshold for a fixed number of processors
  1. SpeedUp decreases as N is increased beyond a certain threshold for a given fixed number of processors, showing that for a given fixed number of processors as N is increased communication (specifically the brandwidth overhead in $iterations * [\mathcal{O}(\tau \log \sqrt{p} + \mu \frac{n}{\sqrt{p}})] + \mathcal{O}(\tau \log \sqrt{p} + \mu n^2)$) begins to dominate over computation cost for the Parallel Algorithm.
- It can be seen from the graph that SpeedUp < 1 for values of $N < 1000$. This is again as expected as communication costs of the parallel algorithm will dominate over computation costs at low values of N
- Thus we can conclude that
  1. At low values of N, parallel algorithm does not have a utility because of communication overheads
  2. For a given fixed value of N, SpeedUp of the parallel algorithm increases initially as P is increased from a low value of P. However on reaching a optimum value of SpeedUp for a certain value of P, the SpeedUp then begins to fall gradually as P is increased further
  3. So for a given problem size of N there is a optimum value of P to get the best SpeedUp