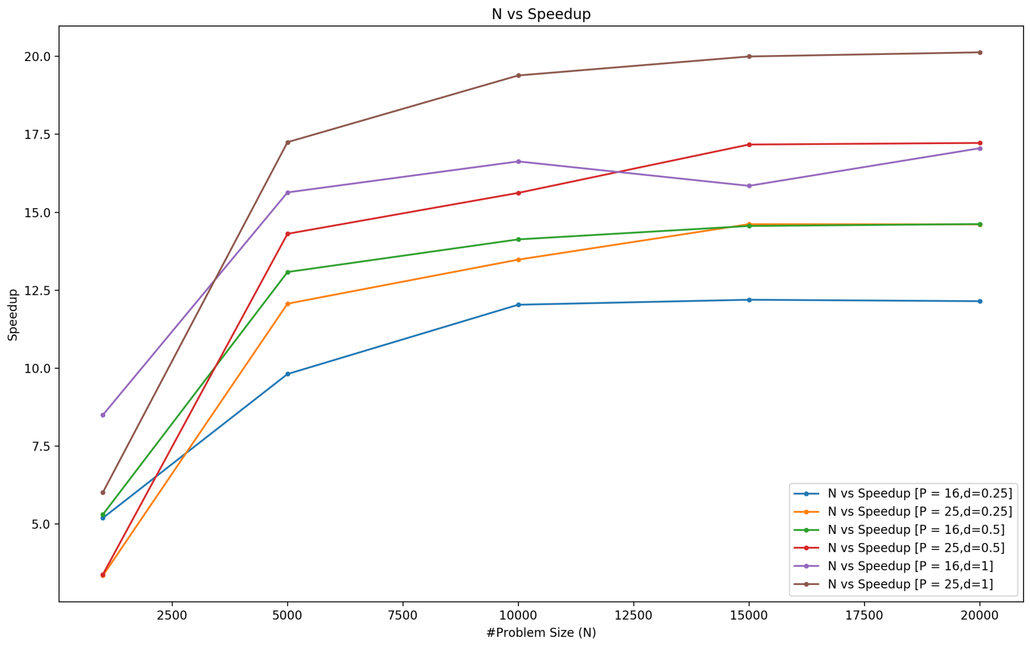
Gaurav Tarlok Kakkar - gkakkar7@gatech.edu
Sai Mohith Potluri - spotluri8@gatech.edu

1 Introduction

Initially, the matrix A and the vector b are owned by the processor with rank 0. We use Jacobi's method to compute the vector x such that $Ax = b$. This method is guaranteed to converge if A is a diagonally dominant matrix. We distribute the matrix A among the $p = q \times q$ processors by first distributing it along the first column in the mesh and then each processor in the first column distributes it further in its row. Similarly, b is distributed along the first column. Each processor locally computes R and D in $O(\frac{n^2}{p})$ time. The portion of D belonging to each row is sent to the the processor in that row that is in the first column. A subroutine for this algorithm is the matrix vector product. It is done by first transposing the vector so that each processor gets the portion of the vector required to multiply with its local matrix. The transpose is done by first sending the local portion of the vector belonging to each row to the diagonal processor in that row and then each of these processors broadcasts this part of the vector along their columns. The local portion of the matrix vector product can be computed in $O(\frac{n^2}{p})$ computation time and $O(\tau \log q + \mu \frac{n}{q} \log q)$ communication time. The product vector is reduced to the processor in the first column in each row. In each iteration of Jacobi's method, we perform two matrix vector products. We use it to compute $Ax - b$ and each processor in the first column computes the square of the norm locally which is then reduced and broadcasted so that all processors get the global norm to decide whether or not the termination condition has reached. x can then be updated locally as $D^{-1}(b - Rx)$ in $O(\frac{n}{q})$ time. This process repeats until either the norm meets a certain threshold criteria or until the execution exceeds the set number of iterations. The solution vector is distributed among the processors in the first column which is then gathered into the processor 0.

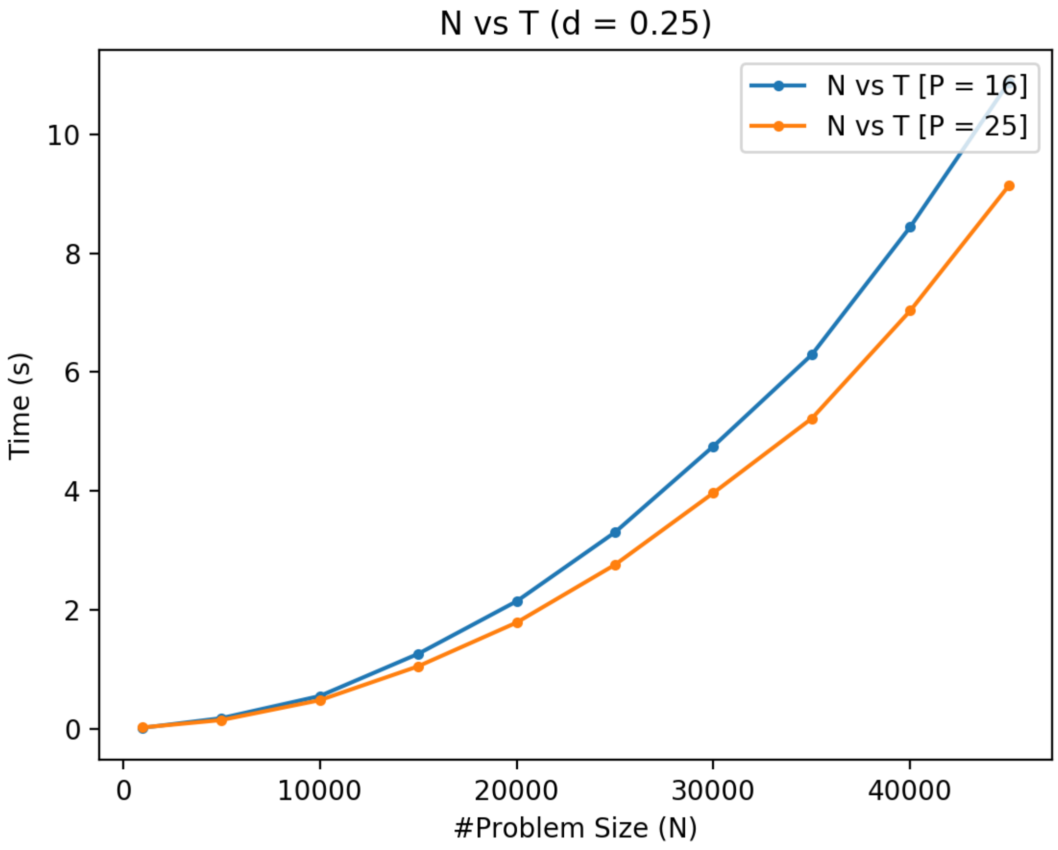
The sequential algorithm takes $O(n^2)$ time during each iteration of this algorithm whereas we expect the parallel algorithm to take $O(\frac{n^2}{p})$ time per iteration as we expect the computation time to dominate the communication time.

1.1 N vs Speedup



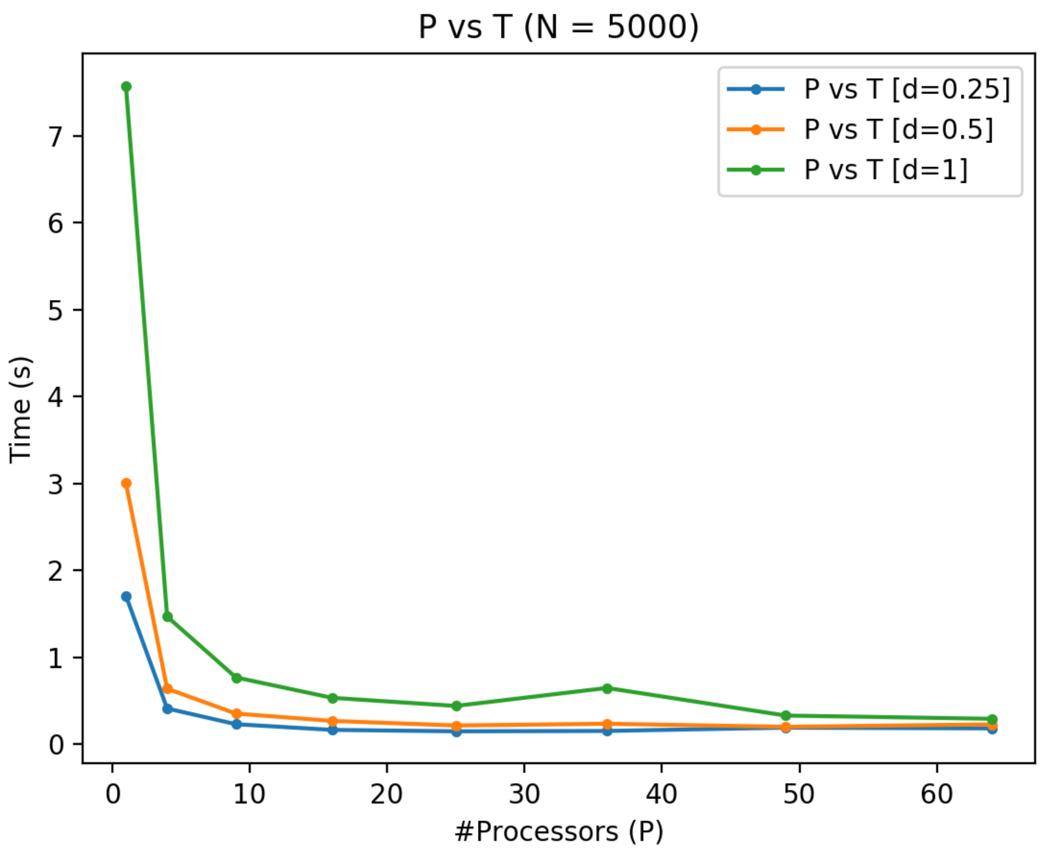
Based on the above paragraph, the expected speedup during the Iterate & Update phase of the algorithm is p. Since, that is the most intensive part of the algorithm we expect the overall speedup to be close to p as well, especially if it takes longer to converge. This expectation is backed up by the results where we can see that speedup increases with difficulty. This also explains the low speedup values for small problem sizes as the algorithm converges relatively fast and hence the overheads become significant in comparison. We can also observe that the speedup is higher when using more processors, as expected.

1.2 N vs T



We expect time to increase in proportion with the size of the matrix A (N^2). The parabola like shape in the graph is therefore as expected. Also as expected is that using more processors makes the program run faster and hence the $P = 25$ curve is below the $P = 16$ curve.

1.3 P vs T



We expect the time taken to be inversely proportional to the number of processors used. Hence, the graph is expected to look like the $y = \frac{1}{x}$ curve as observed. The parameter d determines the diificulty in convergence of the algorithm, hence we see that the curves corresponding to higher values of d are above those corresponding to lower values of d. We suspect the slight bump at $P = 36$ in the curves might be because of the overhead due to communication across different nodes, as before that all processors belong to the same node. We ran on nodes with 28 processors per node on the cluster.