CSE 6220, Spring 2015

Programming Assignment 2 - Report

Ankit Srivastava & Cansu Tetik

February 15, 2017

1 Problem Description

Given a system of linear equations, in the form of a full rank $n \times n$ matrix of coefficients A, an n element vector of constants b, and accuracy l. We want to obtain an approximate solution x, using *Jacobi's method*, such that $\|Ax - b\| \leq l$ where $\|x\|$ is the L2-norm.

2 Algorithm Design Description

In this section, we give a brief description of the algorithms that we implemented for solving the given problem.

2.1 Sequential Algorithm

For solving the problem sequentially, we followed the algorithm given in the assignment. The matrix A is given to the sequential calculator in row-major representation, along with the vector of constants b, accuracy l and maximum number of iterations. We decompose A into diagonal matrix D and remainder component R. We then update the solution vector x in each iteration until it reaches the given accuracy or up to the given maximum number of times.

Pseudocode for the sequential algorithm is shown in Algorithm 1.

Algorithm 1 Sequential Jacobi

```
1: result ← φ
2: procedure MatrixVectorMultiply(n, A, x)
3:     vector y(n)
4:     for i from 0 to n − 1 do
5:         y[i] ← 0
6:         for j from 0 to n − 1 do
7:             y[i] ← y[i] + (A[i × n + j] × x[j])
       return y
8: procedure CalculateL2Norm(n, A, b, x, y)
9:     l2Norm ← 0
10:    for i from 0 to n − 1 do
11:        l2Norm ← l2Norm + (y[i] − b[i])^2
       return √l2Norm
12: procedure Jacobi(n, A, b, l, maxIteration)
13:    vector x(n)
14:    vector D(n)
15:    R ← A
16:    for i from 0 to n − 1 do
17:        D[i] ← A[i × (n + 1)]
18:        R[i] ← 0
19:        x[i] ← 0
20:    for iter from 1 to maxIteration do
21:        y ← MatrixVectorMultiply(n, A, x)
22:        l2Norm ← CalculateL2Norm(n, A, b, x, y)
23:        if l2Norm ≤ l then
24:            break
25:        for i from 0 to n − 1 do
26:            x[i] = (b[i] − y[i])/D[i]
        return x
```

2.2 Parallel Algorithm

For solving the problem in parallel with Jacobi method, we are given the problem size n, the input matrix A, the input vector b, accuracy l and maximum number of iterations max_iter. We use the wrapper function mpi_jacobi to solve the problem with given arguments. We distribute the input vector b and the output vector x within the first column of processors, and the input matrix A within the whole grid; using the formulas given for the number of elements in each processor after distribution. After distribution, we follow the parallel Jacobi algorithm given in the description.

***DistributeVector*:** Using $MPI\_Scatterv$, we distribute the vector from the first processor in the grid within the first column. We decide the number of elements to send/receive per processor with $Block\_Decompose$ helper functions.

***GatherVector*:** We implemented the $MPI\_Gatherv$ version of vector distribution method. Again, we calculate the size of each local vector in the first column processors by using the $Block\_Decompose$ functions in utils.

***DistributeMatrix*:** First we distribute the matrix within the first column of the grid, now each processor in the first column has the matrix elements for its row. However, the elements that belong to different processors in each row are distributed across columns of the matrix. We therefore transpose the matrix and then send it across the rows using $MPI\_Scatterv$. Since we get a transposed matrix in each processor after the scatter, we transpose it back to get the correct distributed matrix on each processor.

***TransposeVector*:** First each processor in the first column (i, 0) sends its local vector to the diagonal processor (i, i) in its row. Then, the diagonals broadcast the local vectors within their column subcommunicators.

***MatrixVectorMultiply*:** We transpose the input vector on all processors using $TransposeVector$ method. Then we locally multiply the vectors and matrices. The resulting local results are summed to first processor in each row with MPI_Reduce.

***Jacobi*:** We first split the local matrix A into the diagonal component D and the residual component R. The split is required only on diagonal processors, since we are distributing a square matrix on a square grid of processors. We then follow the same algorithm as sequential solver and use the parallel implementation of the above functions for updating local x in each iteration. We calculate L2-norm using $MPI\_AllReduce$ and all the processors break if the calculated value is less than the termination value.

Pseudocode for the parallel algorithm is shown in Algorithm 2.

Algorithm 2 Parallel

```
1: result ← φ
2: procedure DistributeVector(n, b, bLocal, comm)
3:     colRank ← rank in the column
4:     q ← √p
5:     if colRank == 0 then
6:         Set sendCounts and displacements with BlockDecompose()
7:     MPI_Scatterv(b, sendCounts, displacements, bLocal) return
8: procedure GatherVector(n, b, bLocal, comm)
9:     colRank ← rank in the column
10:    q ← √p
11:    if colRank == 0 then
12:        Set sendCounts and displacements with BlockDecompose()
13:    MPI_Gatherv(bLocal, b, sendCounts, displacements) return
14: procedure DistributeMatrix(n, A, ALocal, comm)
15:    colRank ← rank in the column
16:    rowRank ← rank in the row
17:    q ← √p
18:    if rank == 0 then
19:        Set sendCounts and displacements with BlockDecompose()
20:    Distribute within first column with MPI_Scatterv(ALocal, A, sendCounts, displacements)
21:    if rowRank == 0 then
22:        Set sendCounts and displacements with BlockDecompose()
23:    Distribute within rows with MPI_Scatterv(bLocal, b, scounts, displs)
   return
24: procedure TransposeVector(n, colVector, rowVector, comm)
25:    colRank ← rank in the column
26:    rowRank ← rank in the row
27:    q ← √p
28:    if rowRank == 0 then
29:        Set sendCount with BlockDecompose()
30:        MPI_Send(colVector, sendCount, colRank, rowComm)
31:    else if colRank == rowRank then
32:        Set recvCount with BlockDecompose()
33:        MPI_Recv(rowVector, recvCount, 0, rowComm)
34:    Set bcastCount with BlockDecompose()
35:    MPI_Bcast(rowVector, bcastCount, rowRank, colComm) return
```

```
36: procedure MatrixVectorMultiply(n, Alocal, xLocal, yLocal comm)
37:     colRank ← rank in the column
38:     rowRank ← rank in the row
39:     q ← √p
40:     localVec ← 0
41:     TransposeVector(n, xLocal, localVec, comm)
42:     yLocal ← ALocal * localVec
43:     MPI_Reduce(yLocal, yLocal, rowCount, MPI_SUM, 0, rowComm)
    return
44: procedure Jacobi(n, ALocal, bLocal, xLocal, comm, maxIteration, l)
45:     q ← √p
46:     if rowRank == colRank or rowRank == 0 then
47:         if rowRank == colRank then
48:             Set DLocal and RLocal
49:         if rowRank! = 0 then
50:             MPI_Send(DLocal, rowCount, 0, rowComm)
51:         else if colRank! = 0 then
52:             MPI_Recv(DLocal, rowCount, colRank, rowComm)
53:     Initialize xLocal in first column
54:     for iter from 1 to maxIteration do
55:         MatrixVectorMultiply(n, ALocal, xLocal, temp, comm)
56:         if rowRank == 0 then
57:             Set l2Norm
58:         MPI_Allreduce(l2Norm, 1, MPI_SUM, comm)
59:         l2Norm ← √l2Norm
60:         if l2Norm ≤ l then
61:             break
62:         Update x withMatrixVectorMultiply(n, RLocal, xLocal, temp, comm)
63:         if rowRank == 0 then
64:             xLocal ← (bLocal − temp)/Dlocal
65:     MPI_Gatherv(bLocal, b, sendCounts, displacements) return
```

3 Plots and Observations

For running the experiments, we had three parameters that we could play with: problem size (n), number of processors (p), and difficulty level (d). We obtained timings for various combinations of the aforementioned parameters and plotted and analyzed the timings so obtained. In the sections below, we discuss how we varied the parameters and the effect that the variation had on the performance.

We used *jinx7* node, which has 4×6 cores, for running our experiments up to $p = 16$ and used 5 *sixcore* nodes for running the experiments for $p = 25$.

3.1 Problem Size (n)

First of all, we had to choose a suitable problem size for running our experiments. We observed that even though the serial Jacobi implementation was able to handle problem sizes of up to $n = 39000$, the parallel part started failing for problem sizes of $n = 30000$. After some deliberation, we concluded that this was happening because *jinx* cluster has low memory per node (maximum of 24 GB on one node). Therefore, this was expected with extra memory requirements in the parallel implementation.

We restricted our experiments to a maximum problem size of $n = 28000$, which worked fine on smaller number of processors, but didn't work when run with $p = 25$. Problem sizes were varied in the set $\{10000, 15000, 20000, 25000, 28000\}$.

3.2 Number of Processors (p)

Since we could only use number of processors which are perfect squares, we varied number of processors in the set $\{1, 4, 9, 16, 25\}$ and obtained the runtimes for solving the problem for all the different problem sizes (discussed in Section 3.1), for three different difficulty levels: 0.0, 0.5 & 1.0. The corresponding runtime and speedup plots for different difficulty levels are shown in Figure 1 & Figure 2, Figure 3 & Figure 4, and Figure 5 & Figure 6. We observed that the experiments were not very reliable while running on the cluster for $p > 25$ and therefore we decided to exclude the results. Also, as discussed in Section 3.1, we didn't have any data for $p = 25$ and $n = 28000$ and therefore the corresponding data points are missing from the plots.

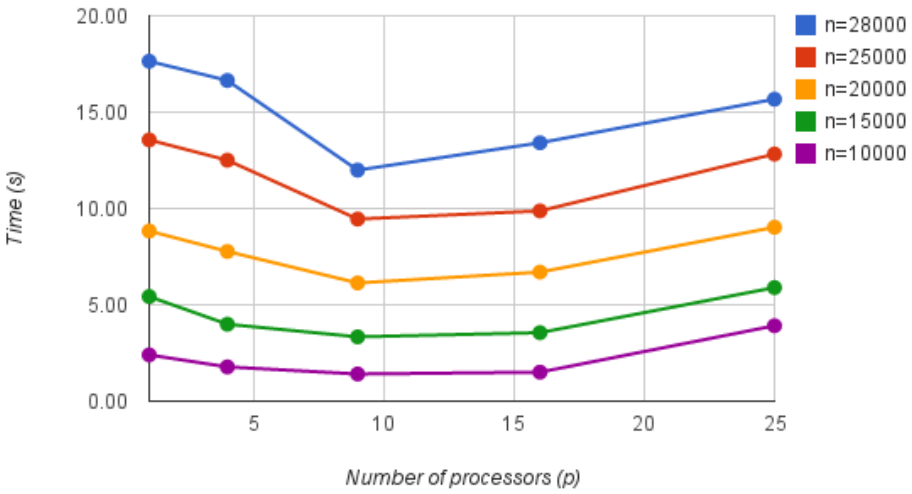

Figure 1: Runtime v/s Number of Processors ($d = 0.0$)

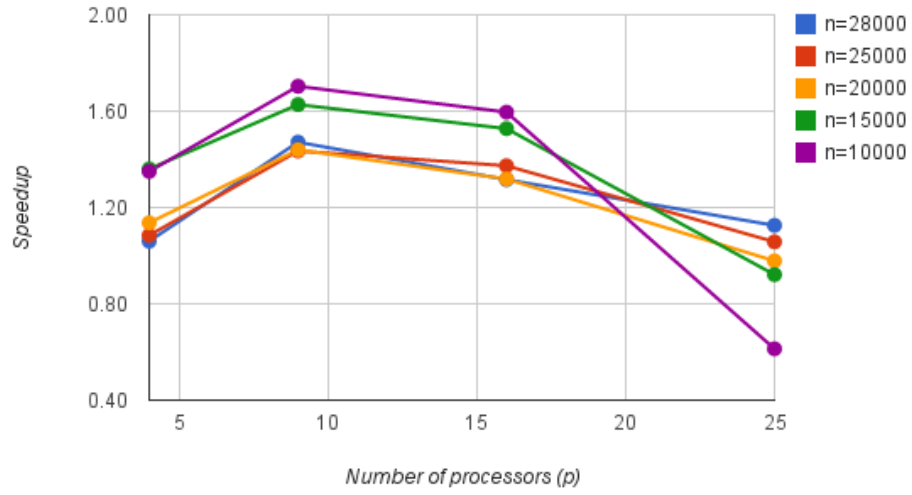

Figure 2: Speedup v/s Number of Processors ($d = 0.0$)

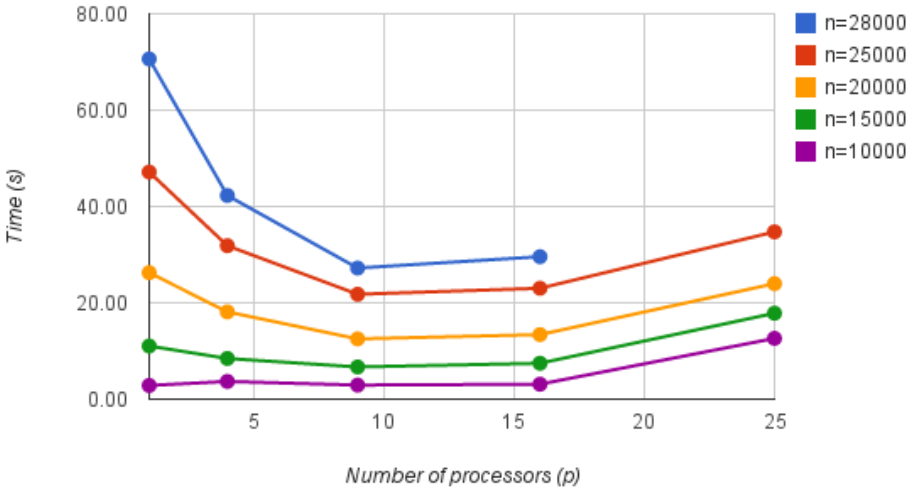

Figure 3: Runtime v/s Number of Processors ($d = 0.5$)

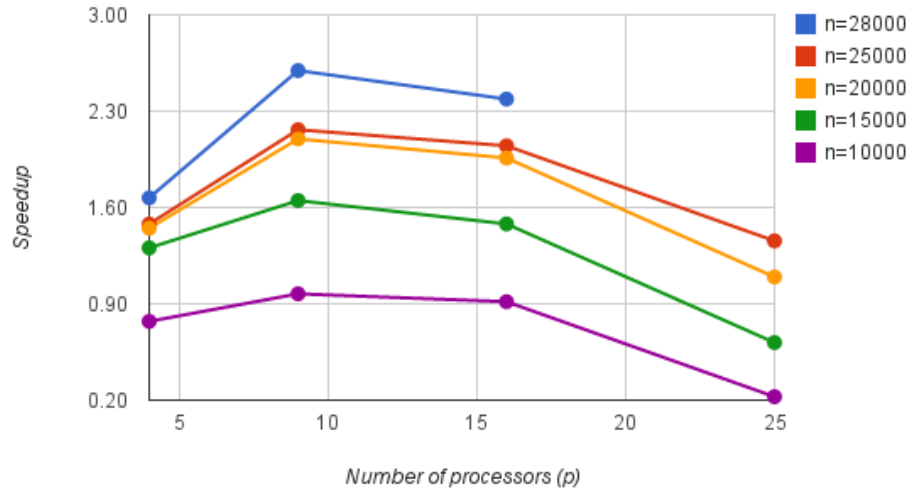

Figure 4: Speedup v/s Number of Processors ($d = 0.5$)

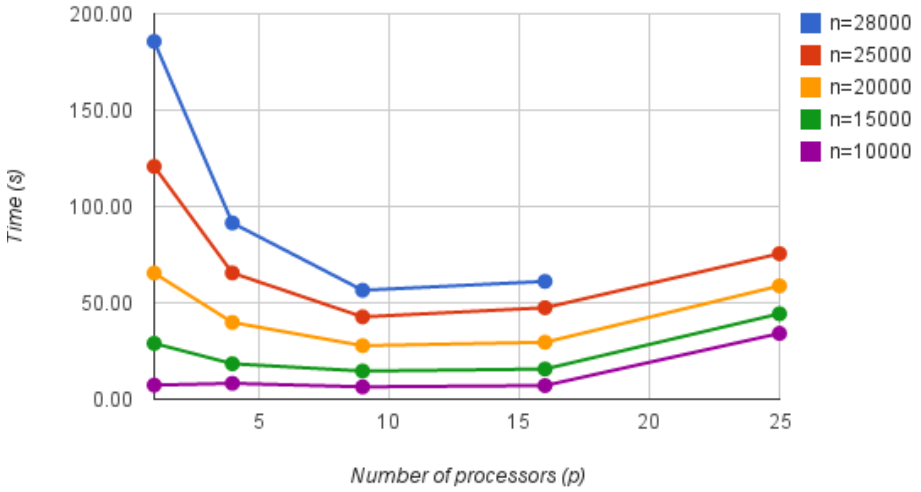

Figure 5: Runtime v/s Number of Processors ($d = 1.0$)

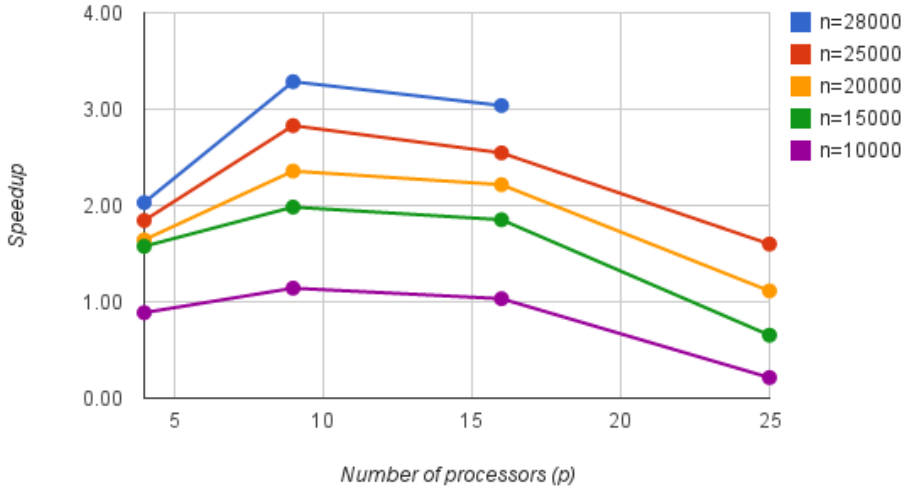

Figure 6: Speedup v/s Number of Processors ($d = 1.0$)

From the above plots, we can see that on increasing the number of processors the runtime decreases up to $n = 9$, is a little higher for $p = 16$, and then increases further for $p = 25$. Thus the observed speedup is highest, for a combination of most problem sizes and difficulty level combinations, for $p = 9$. This wasn't expected and is probably due to low memory on the cluster.

We also observed that the speedup increased with the increase in difficulty

level. This was expected since increasing the difficulty level results in more number of iterations and therefore the gains in computation time in parallel trumps the communication overhead as the difficulty level increases. The effect of difficulty level on the performance and speedup is explored in further detail in Section 3.3.

3.3 Difficulty Level (d)

Difficulty level of a problem is a real number in the range $[0, 1]$, which indicates the number of iterations it takes for Jacobi to converge to a solution. We varied the difficulty level in the range $[0.0, 1.0]$ in steps of 0.1 for different number of processors and plotted the corresponding speedup, relative to the sequential performance, in Figure 7, Figure 8, Figure 9, and Figure 10.

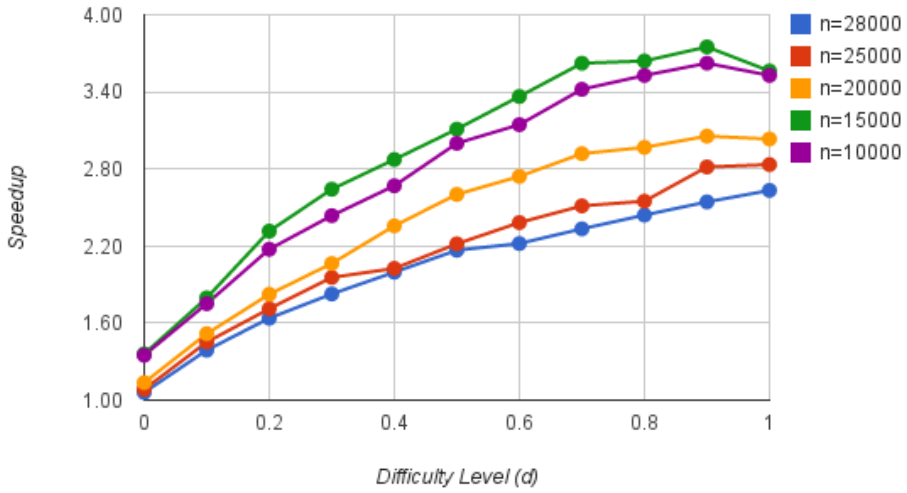

Figure 7: Speedup v/s Difficulty Level ($p = 4$)

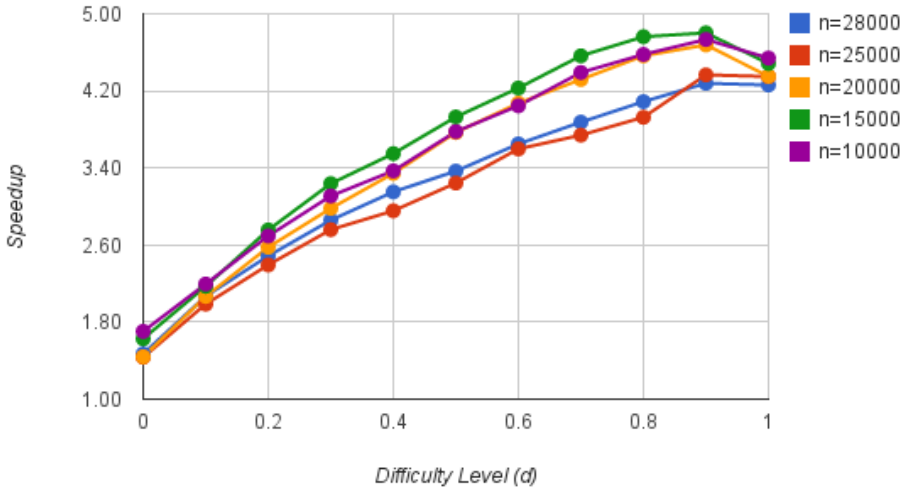

Figure 8: Speedup v/s Difficulty Level ($p = 9$)

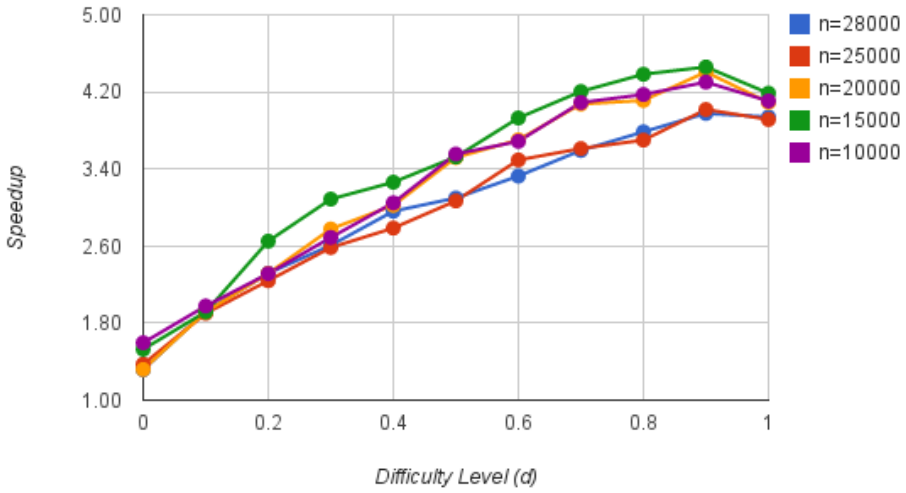

Figure 9: Speedup v/s Difficulty Level ($p = 16$)

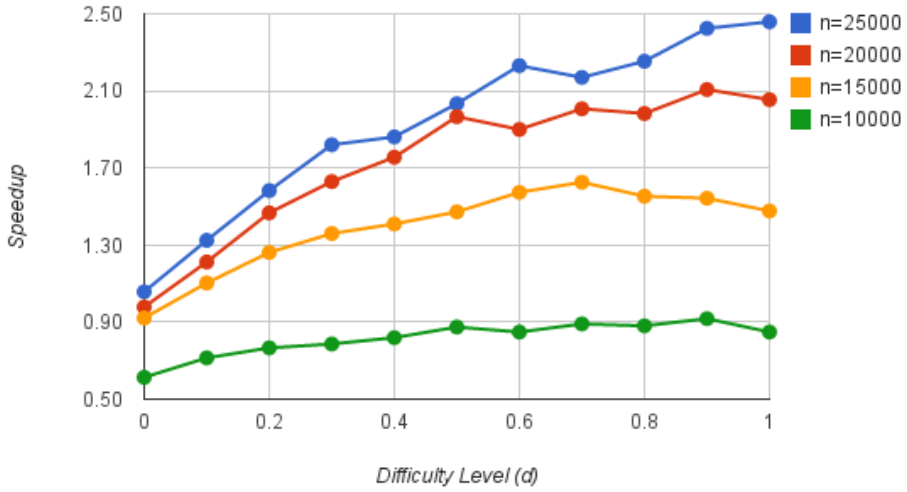

Figure 10: Speedup v/s Difficulty Level ($p = 25$)

From the above plots, we can verify what we observed in Section 3.2: speedup generally increases with increase in the difficulty level for all values of the number of processors and all the problem sizes. However, observed speedup is highest for $n = 15000$ for $p = 4, 9$, and 16 with $n = 10000$ being the second best. The difference between the relative speedup of different problem sizes though constantly keeps decreasing as the number of processor increases. For $p = 25$, we see that the maximum speedup is achieved for $n = 25000$, indicating again that gains in computation trumps communication overhead when the problem size is big, on large number of processors, even though the speedup is smaller than that obtained for $p < 25$.