CSE 6220 Programming Assignment 3
Jacobi's Method

Qi Zheng (qzheng61) Wenqing Shen (wshen35)

Overview

The **Jacobi method** (or **Jacobi iterative method**) is an algorithm for solving a system of linear equations $Ax = b$, with A having no zeros along its main diagonal. Each diagonal element is solved for, and an approximate value is plugged in. The process is then iterated until it converges. It can be shown that for diagonally dominant matrices, Jacobi's method is guaranteed to converge. The goal of this assignment is to develop a parallel algorithm for Jacobi's method, and compare the performance with sequential algorithm.

Algorithm Design

Sequential Implementation

Given a full rank $n \times n$ matrix $A \in R^{n \times n}$, a right-hand size vector $b \in R^n$, a termination accuracy $l2\_termination \in R$, a maximum iteration count iter_max. Jacobi's method iteratively approximates $x \in R^n$ for $Ax = b$ following the steps:

Initialize $x = [0, 0, ...0]$
$D = diag(A)$
$InvD = D^{-1}$
$R = A - D$
$iter\_count = 0$
$residual = ||Ax - b||$ // ||•|| is the L2 norm

while ($residual > l2\_termination$ && $iter\_count < iter\_max$) *do* {
    $x = invD * (b - Rx)$ // update x
    $residual = ||Ax - b||$ // update residual
}

Parallel Implementation

In this project, we use a 2-D mesh (grid) as the communication network. We assume that the number of processors is a perfect square: $p = q \times q$, q is an integer. p processors will be arranged into a grid of size $q \times q$. The inputs are distributed on the grid in a pattern we define

as block distribute method. The method distributes n components onto q processors following a pattern of:

$$\begin{cases} \lceil \frac{n}{q} \rceil & \text{if } i < (n \mod q) \\ \lfloor \frac{n}{q} \rfloor & \text{if } i \geq (n \mod q) \end{cases}$$

Initially, the matrix A and vector b both are both stored in processor (0, 0) in the grid. The parallel algorithm is described as follows:

1. **Vector Distribution**

   Block distributes the input vector of size n from process (0, 0) to processes (i, 0), i.e., the first column of the 2D grid communicator. Elements of the vector are equally distributed, such that ceil(n/q) elements go to the (n *mod* q) first processes and floor(n/q) elements go to the remaining processes in a pattern described in previous texts. Vector b is distributed in this pattern.

2. **Matrix Distribution**

   Equally block distribute the matrix A stored on processor (0,0) onto the whole grid (i, j). First equally distribute the row blocks of A onto the first column of processors using MPI Scatterv through the column sub-communicator created by MPI_Cart_sub. Then for each row in the grid, equally scatter the elements onto each processor in the same row using row sub-communicator.

3. **Transpose a vector**

   Define a function that 'transpose' the input vector x distributed among the first column (i,0) of processors to the whole grid, such that it is block decomposed by row of processors. To accomplish this, first, each processor (i, 0) alone the first column sends its local vector to the diagonal processor (i, i). Then the diagonal processor (i, i) broadcasts the message among its column using a column sub-communicator.

4. **Solves $Ax = b$ for x using Jacobi's method,**

   Implement parallel Jacobi's algorithm as follows:

```
D = diag(A)          // block distribute to first column (i,0)
invD = 1/ D
R = A - D            // make a copy of A except setting diagonal to zero
x = [0, ..., 0]      // initialize x to zero, block distributed on first column

for (iter in 1:max_iter):
    w = R*x          // The matrix multiplication is solved by first transposing the input
                     // vector x as defined earlier, then locally multiplying the row
```

```
                    // decomposed vector by the local matrix. Then, the resulting local
                    // vectors reduce along rows onto the first column.
x = invD * (b - P)      //  local on first column
w = A*x                 // using distributed matrix vector multiplication as described
                        // in w = R*x
l2 = ||b - w||          // calculate L2-norm in a distributed fashion
if (l2 <= l2_termination):
        return          // exit if termination criteria is met, make sure
                        // all processor know that they should exit (-> MPI_Allreduce)
```

Results and Discussions

To analyze the performance of parallel computing and compare it with sequential program, we ran both algorithms with different problem sizes and difficulty levels, respectively. Runtime plots and speedup plots have been generated for performance analysis and comparison.

Runtime Analysis

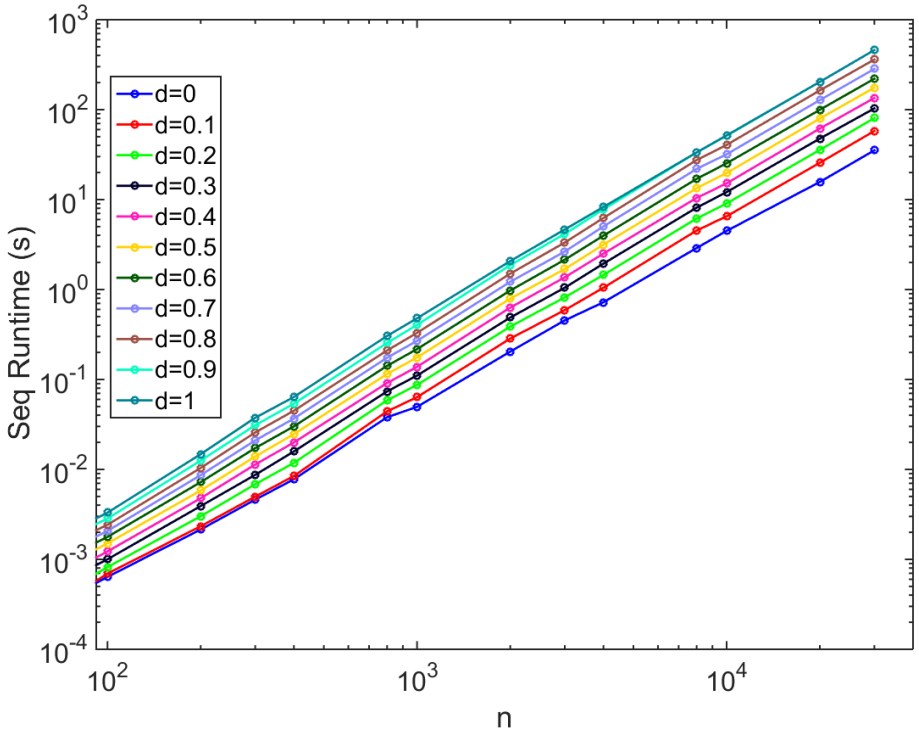

Figure 1. Sequential Jacobi runtime as a function of problem size (n) for various difficulty levels

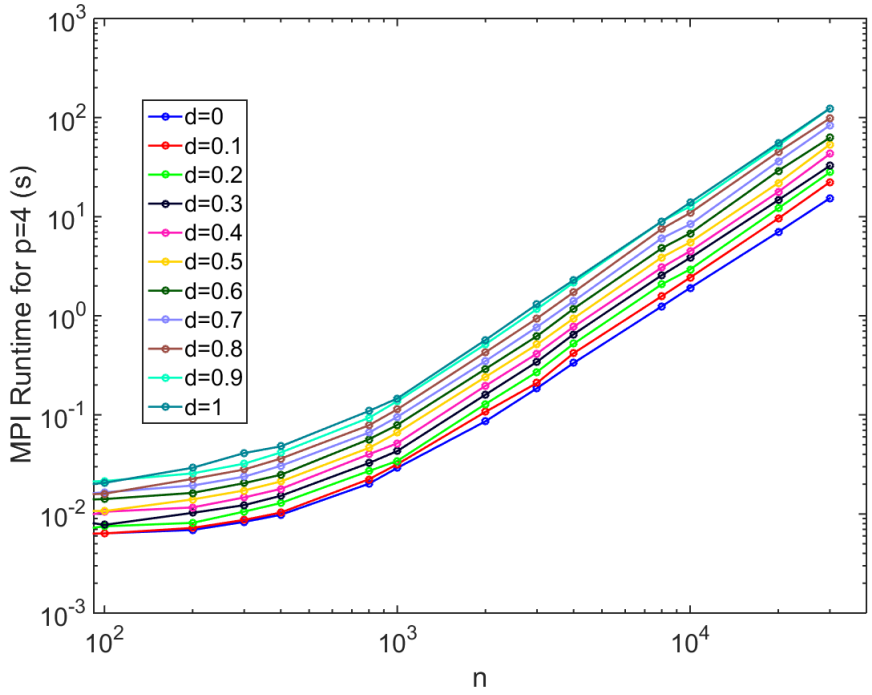

Figure 2. Parallel Jacobi runtime as a function of problem size (n) for various difficulty levels

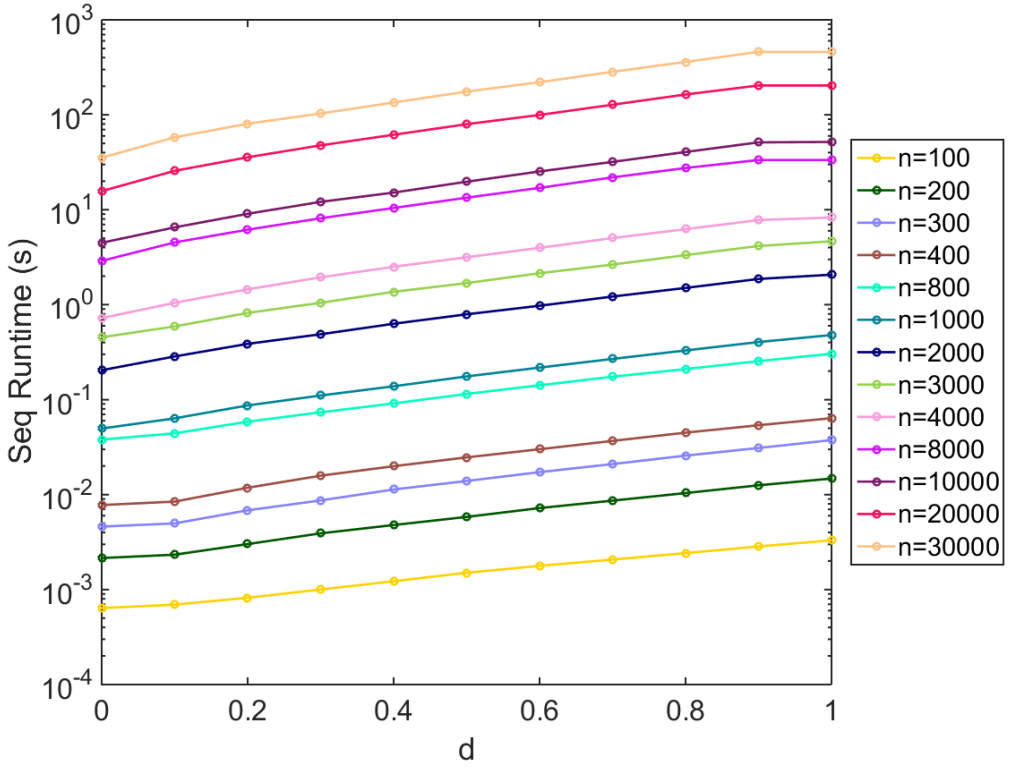

Figure 3. Sequential Jacobi runtime as a function of difficulty (d) for different problem sizes

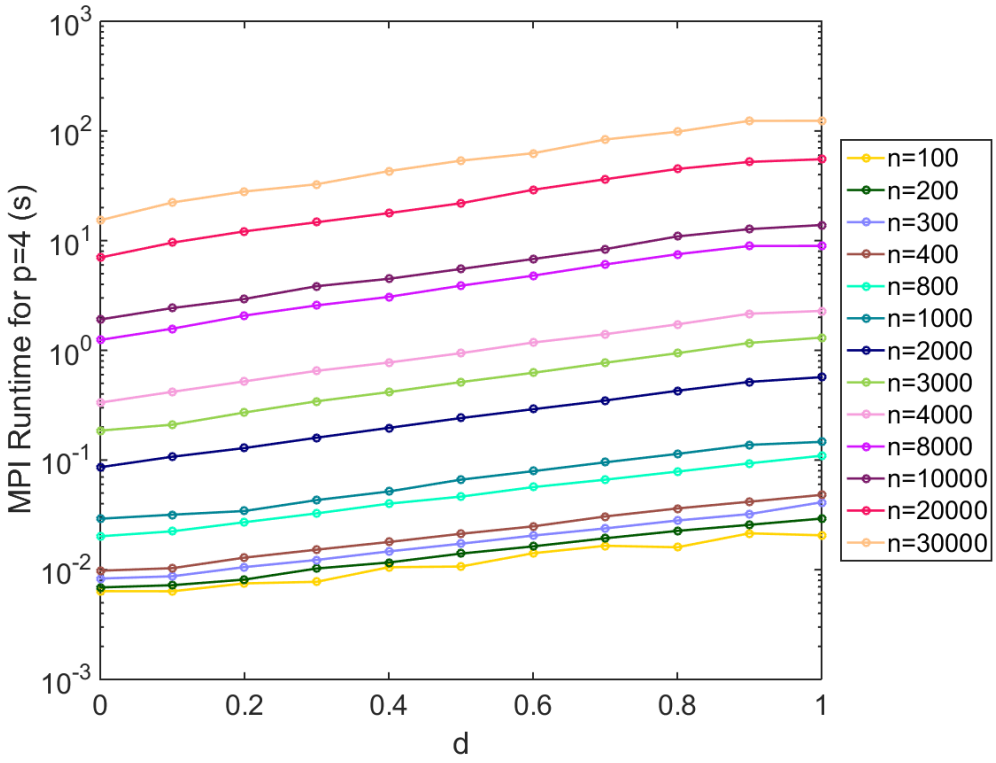

Figure 4. Parallel Jacobi runtime as a function of difficulty (d) for different problem sizes

Figure 1 and Figure 2 are runtime plots for sequential and parallel Jacobi methods respectively, plotted as a function of problem size (n) for various difficulty levels. From the plots, it is apparent to see that as the problem size increases, both algorithms take longer to terminate. Meanwhile, for a fixed size problem, the more 'difficult' (i.e., less diagonal dominant and thus more difficult to converge) the problem is, the longer our algorithms take to compute the results. However, the changing behaviors of two algorithms are slightly different in terms of problem size.

For the sequential algorithm, time increases with increasing problem size (n) almost linearly on the log-log plot, indicates that with a fixed difficulty value, sequential runtime is a power function of problem size. If we analyze the problem further, the complexity of the matrix-vector multiplication is $O(n^2)$, and is executed once in each iteration, which is consistent with what we observe on the plots.

For the parallel algorithm, although the runtime also increase with the problem, the increase is much slower when the problem size is small, then the trend goes steeper when the size becomes large. Also, if we compare the exact runtime value for the same problem size and difficulty among sequential and parallel programs, we can see that the runtime for the parallel algorithm is much larger than that of sequential when n is small. This is because when n is small, the communication time dominates the overall runtime for

parallel computing, while for sequential Jacobi, it has no such communication overhead. In the implementation of parallel Jacobi's method, there are quite a few communications going on, such as: 1) the distribution of matrix A and R onto the grid, distribution of vector x, D^{-1} (we used a vector to store D^{-1} which would be further discussed in the optimization section). 2) scatter the updated *x* from first column onto the whole grid; 3) reduce vector components in each row of processors in each iteration; 4) reduce to sum up residuals in the first column. Due to the relatively intensive communications among processors, parallel Jacobi only starts to outperform the sequential method when n is large. We will come back to that later.

We also plotted runtime as a function of difficulty (d) for different problem sizes for both algorithms, generally, the increasing trends/patterns for both algorithms in terms of increasing problem difficulty are very similar to each other.

Speedup Analysis

To analyze the speed up with changes in the difficulty level and problem size, we fixed the number of processors (p=4), ran our program, and calculated different speedups with different values of n and d.

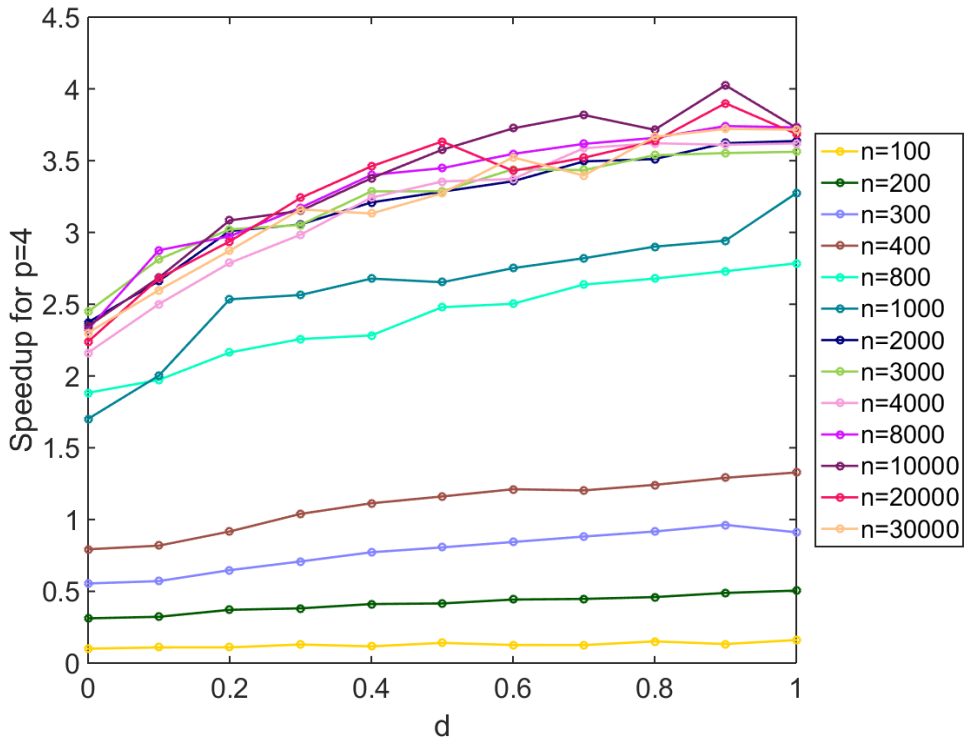

Figure 5. Parallel speedup as a function of difficulty (d) for different problem sizes

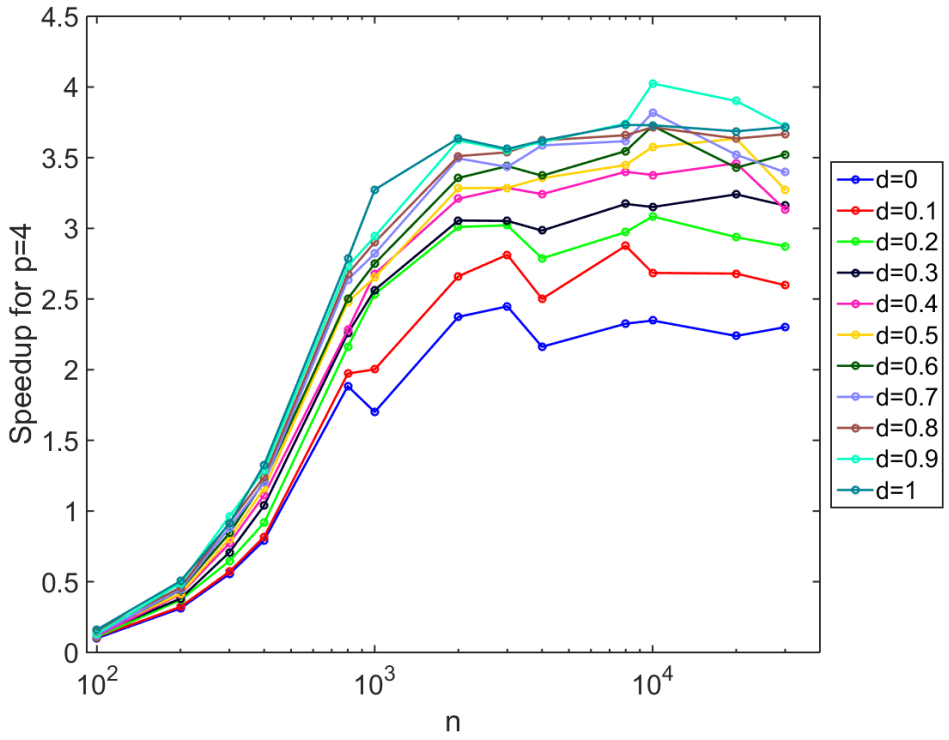

Figure 6. Parallel speedup as a function of problem size (n) for different difficulty levels

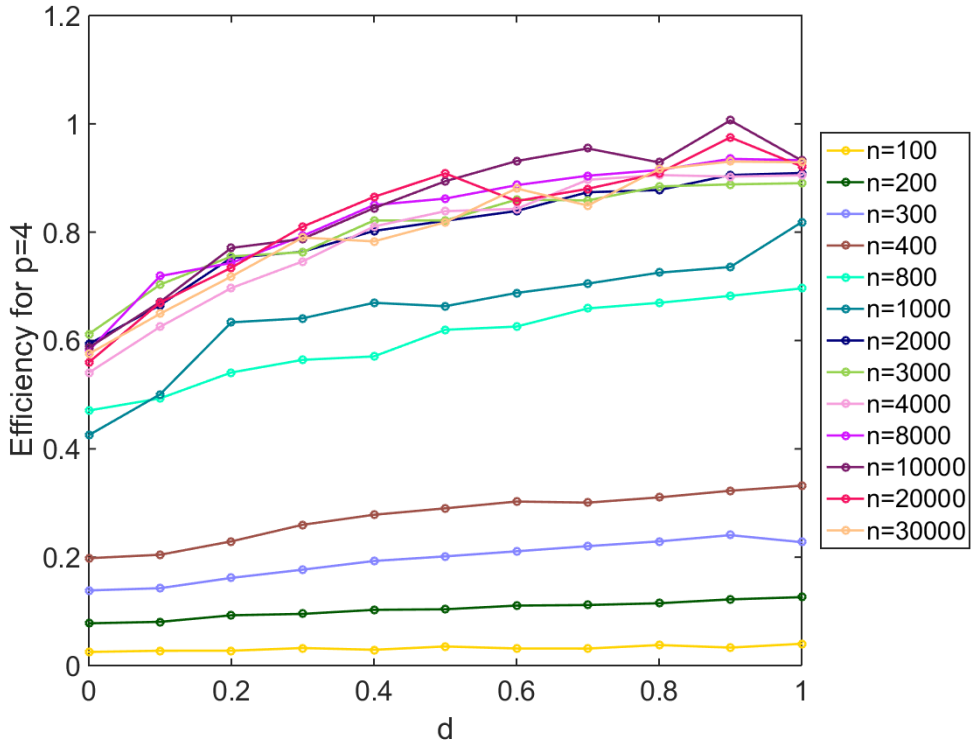

Figure 7. Efficiency as a function of difficulty (d) for different problem sizes

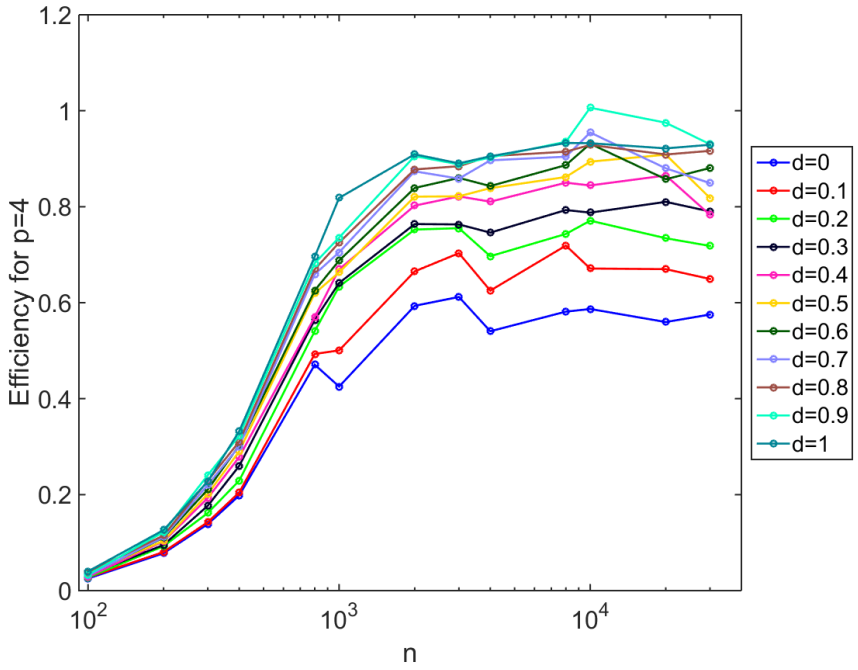

Figure 8. Efficiency as a function of problem size (n) for different difficulty levels

On Figure 5, we plotted parallel speedup as a function of difficulty level (d) for different problem sizes. From the plots we can see that for the same difficulty level, generally speedup is larger for problems of larger size, and is much smaller when n is small. However, when the problem is large enough, when the problem size continues to grow, instead of increasing, the speedup starts decreasing. We can see form the graph that the speedup for problem of size 30000 is actually lower than that of the problem of size 10000. This indicates to achieve best speedup, we have an optimal number of processors. Further increasing the number of processors will actually results in wastes.

On Figure 6, we plotted parallel speedup as a function of problem size (n) for different difficulty levels. From the plots, we observe sharp increase when n increases from 200 to about 1000, then the increase slows down and gradually, the speedup plots flatten out. Also, we can see that when n is small, different difficulty levels do not have huge differences on speedup. At that stage, problem size is the dominant factor that affect the speedup. When problem size is very large, difficulty starts to exhibit its influence on speedup. As we can see, with different difficulty value, the speedup could be quite different for the same large problem size.

Efficiency plots have also been plotted as a function of difficulty (d) and problem size (n) in Figure 7 and 8 respectively. For the two parameters we analyzed, the behavior of efficiency is similar to that of speedup.

Additional Analysis

In addition to analyzing the influence of problem size and difficulty level, we also analyzed the effects of number of processors on the performance.

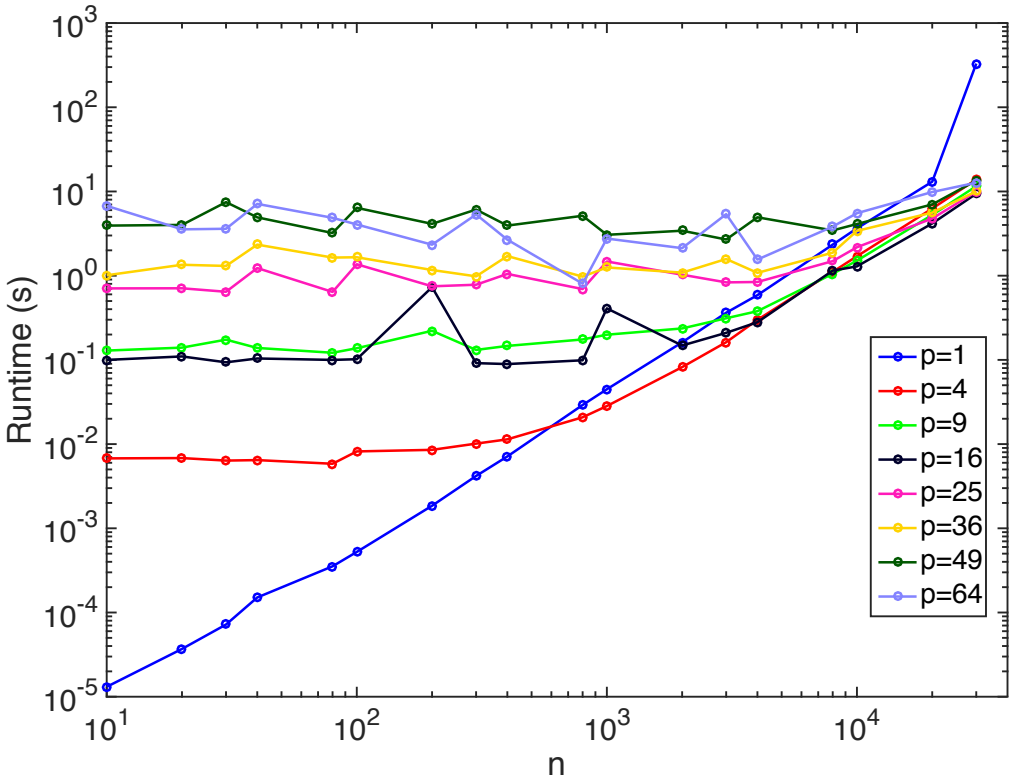

Figure 9. Runtime as a function of n for different numbers of processors

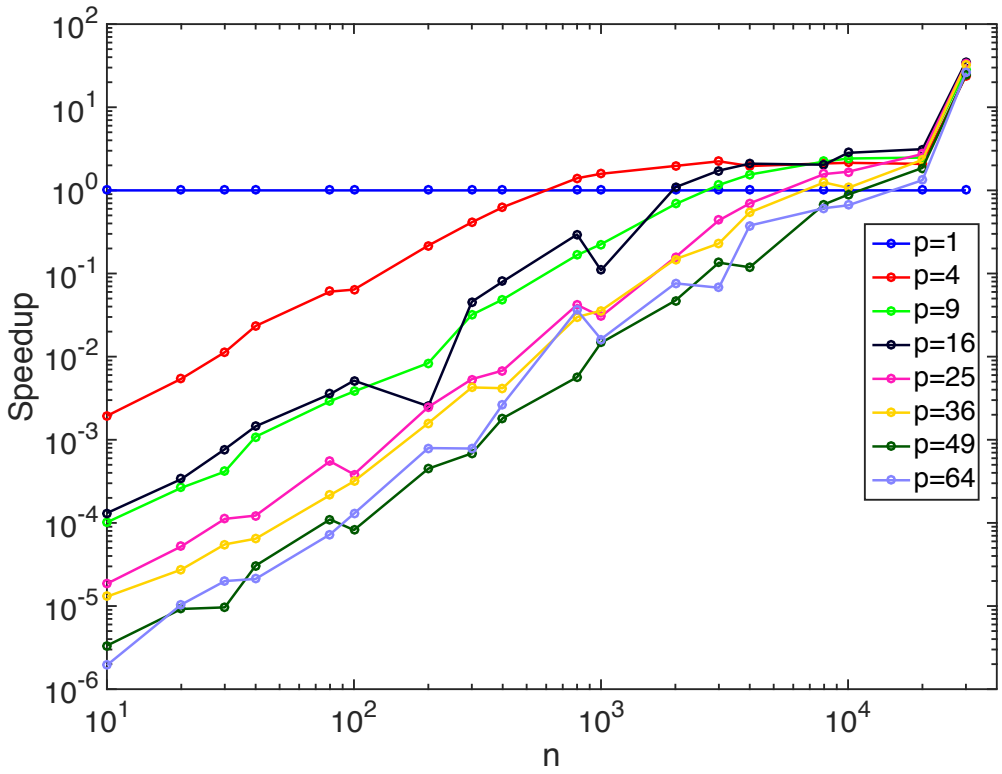

Figure 10. Speedup as a function of n for different numbers of processors

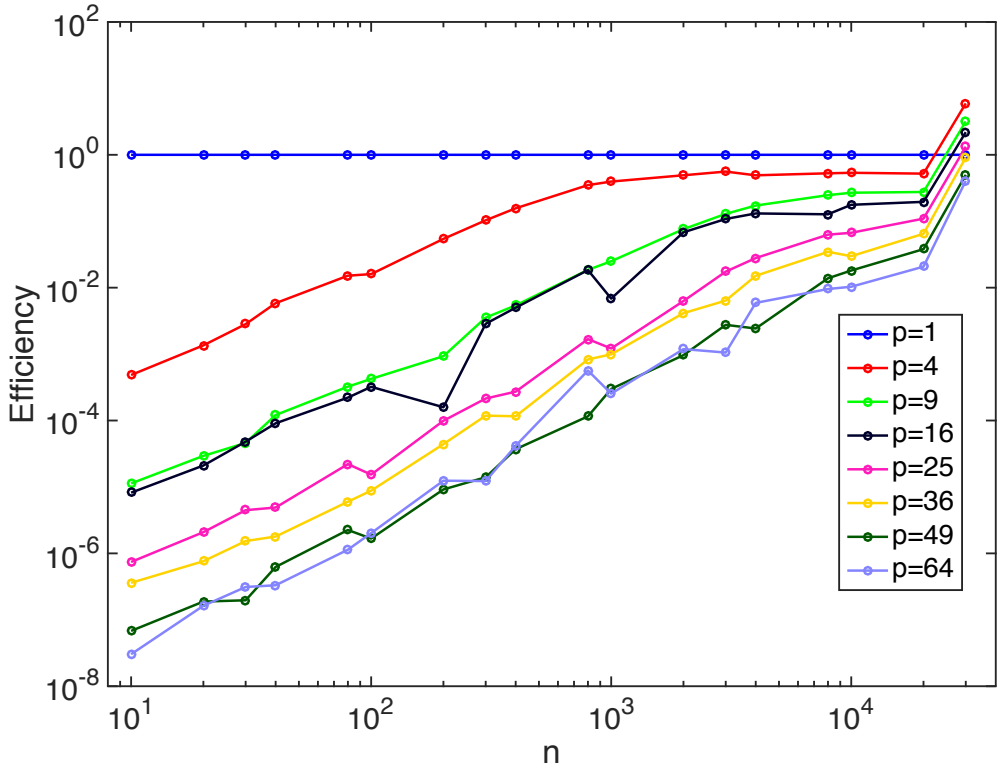

Figure 11. Efficiency as a function of n for different numbers of processors

In Figures 9, 10 ,11, runtime, speedup and efficiency are plotted as a function of problem size n with p ranging from 1 to 64. As we stated in previous texts, in this problem, there are quite a few communication processes going on. Hence, when the problem is not computational intensive (that is to say, n is small), parallel computing does not exhibit much advantage. Only when n is large enough (in this case, when n approaches 10^3, some parallel cases start run faster than sequential, and when n exceeds 10^4, most parallel cases outperforms sequential), parallel computing excels sequential in performance. Also, the dominant power of communication also exhibits itself with flat plots on the graph when a small n is varying. When the problem size n is small, the runtime is dominated by communication time, hence it increases with the increase in p. When problem size n is large, computing takes much longer time, hence when increasing p from 1, the computing power of the cluster increases, and thus reduces the runtime. However, as p increases, communication starts playing a more and more important role. When p exceeds certain threshold, communication takes the lead, and thus offsets the runtime reduce from the increase of computing capability. Both speedup and efficiency plots conform with our previous conclusion. Only when the problem size is large enough, using parallel computing with decent number of processors is valuable. In the end, plotting and analyzing the runtime, speedup and efficiency plots will enable us to determine the optimal number of processors to use for problems of certain sizes with a goal of improving the efficiency, reducing runtime, and using smaller number of processors if possible.

Optimizations

1. In the original algorithm described in the assignment, D is stored as matrix. For a large size A, D is very sparse with non-zero values only on diagonal. To save the memory, for the diagonal matrix D, we used an array of size n to hold the data, and block decompose it onto the first column in the same pattern as vector b.
2. Since matrix (vector) D is only used for the calculations of R and D^{-1}. During the iterative Jacobi's method, only D^{-1} is used, hence we actually don't need D. D^{-1} could be obtained directly from taking the inverse of the diagonal values of A, while R could be obtained by copying A and setting diagonal values to 0. This could further save storage memory, especially

for solving large size problems with limited number of processors. (When n=30000, the memory required for storing A has already reached 7.2GB!)