CSE 6220 Programming Assignment 3

Team members: Huaidong Yang, Yingdan Wu

1. Algorithm[1]

1.1 Sequential Jacobi Method (jacobi.cpp)

To solve a system of linear equations $Ax=b$, we adopted iterative numerical method-Jacobi algorithm as following. Firstly, we obtain a matrix D which is the diagonal of A and subsequently a matrix $R=A-D$. Then we choose an initial guess for x and update $x=D^{-1}(b-Rx)$ iteratively until $||Ax-b||$ is reduced to an acceptable value.

1.2 Parallel Jacobi Method (mpi_jacobi.cpp)

To implement parallel Jacobi method, we constructed the following help functions.

distribute_vector: Equally distributes a vector stored on processor (0,0) onto processors (i,0) [i.e., the first column of the processor grid]. Since we only need to communicate among the processors in the first column, we **split the communicator** and get a column communicator with the processors in the first column only. The size of the sub-vector to be sent to other processors from (0,0) can be decided such that ceil(n/q) elements go to the `n mod q` first processes and floor(n/q) elements go to the remaining processes. After determining the sendCounts, displs and recvCounts, etc., we adopted **MPI_Scatterv** for distributing the vector to first column processors from processor (0,0).

gather_vector; Reverts the operation of **distribute_vector()**. Split the communicator into each column community. Use **MPI_Gatherv** to send vetors from (i, 0) to (0,0). The definition of sendCounts, displs, and recvCounts are the same in **distribute_vector**.

distribute_matrix: Equally distributes a matrix stored on processor (0,0) onto the whole grid (i,j). Block distributes the input matrix of size n-by-n, stored in row-major format onto a 2d communicator grid of size q-by-q with a total of p = q*q processes. The matrix is decomposed into blocks of sizes n1-by-n2, where both `n1` and `n2` can be either ceil(n/q) or floor(n/q). The `n` rows of the matrix are block decomposed among the `q` rows of the 2d grid. The first `n mod q` processes will have ceil(n/q) rows, whereas the remaining processes will have floor(n/q) rows. The same applies for the distribution of columns. Since the partitioned blocks in input_matrix to be sent are not connected with respect to indices, I stored these blocks in an **index-connected manner** in another matrix ***A***. Additionally, we need to arrange the matrix A such that the stored blocks are **in the sequence as the destination of the processors**. After identifying the sendCounts, displs and recvCounts, etc., we adopted **MPI_Scatterv** for distributing the matrix ***A*** to all processors from processor (0,0).

transpose_bcast_vector: Given a vector distributed among the first column of processor, send the column vector to its diagonal processor (i,i) using **MPI_Gatherv.** The diagonal processor receives the column vector as a row vector, which achieves the transpose. Split the communicator into each column community. Then use the **MPI_Bcast** to broadcast the row vector to its column community.

distributed_matrix_vector_mult: Calculates y = A*x for a distributed n-by-n matrix A and distributed, size `n` vectors x and y on a q-by-q processor grid. The matrix A is distributed on the q-by-q grid as explained for the `distribute_matrix()` function. The vectors are distributed on the q-by-q grid as explained for the `distribute_vector()` function. The matrix multiplication is solved by first transposing the input vector `x` onto all processors as described for the `transpose_bcast_vector()` function, then locally multiplying the row decomposed vector by the local matrix. Then, the resulting local vectors are summed by using MPI_Reduce along rows by using row sub-communicators onto processors of the first column, which yields the result `y`.

distributed_jacobi: Solves A*x = b for `x` using Jacobi's method, where A, b, and x are distributed among a q-by-q processor grid. The vector *D* is the diagonal of matrix *A*. Hence, it can only be extracted from the local_A at processors (i,i) and then sent to its corresponding column processor (i,0) via **MPI_Send** and **MPI_Recv**. For processors (i,i), we obtain the local_R matrix by copying local_A and setting the diagonal to zero; for the other processors (i,j), where i and j are not identical, we obtain the local_R matrix by copying local_A. After the **distributed_matrix_vector_ mult** operation, we have the calculated local_y at column processors (i,0). Then we can get the local error as a **vector local_err**. Adopting **gather_vector** for local_err, we can get all the errors as a vector of length *n* at processor (0,0). At processor (0,0), we can compute the **l2 error** and then send it to all the other processors using **MPI_Bcast**. Then all the processors can check the termination criteria and determine whether to terminate the **while loop**. The steps for the Jacobi methods are roughly as follows (pseudo-code):

```
D = diag(A)     // block distribute to first column (i,0)
R = A - D       // copy A and set diagonal to zero
x = [0,...,0]   // init x to zero, block distributed on first column
while (iter < max_iter and l2 > l2_termination):
    w = R*x         // using distributed_matrix_vector_mult()
    x = (b - P)/D   // purely local on first column, no communication necessary!
    w = A*x         // using distributed_matrix_vector_mult()
    l2 = ||b - w||  // calculate L2-norm in a distributed fashion
    iter++      // iteration counts
```

2. Main Results

The run-time complexity in the main function can be analyzed as follow:

Distribute matrix, $scatterv\left(\frac{n^2}{p}, p\right) \rightarrow O(\tau logp + \mu n^2)$ (1)

Distribute vector, $scatterv\left(\frac{n}{\sqrt{p}}, \sqrt{p}\right) \rightarrow O(\tau logp + \mu n)$ (2)

Send/recev D, $send\left(\frac{n}{\sqrt{p}}, \sqrt{p}\right) \rightarrow O(\tau + \mu \frac{n}{\sqrt{p}})$ (3)

While loop, $O\left(\frac{n^2}{p}\right) + O(\tau logp + \mu n)$ (4)

Gather, $gatherv\left(\frac{n}{\sqrt{p}}, \sqrt{p}\right) \rightarrow O(\tau logp + \mu n)$ (5)

Totally, the run time complexity is: $O(\tau logp + \mu n^2) + Iteration * O(\frac{n^2}{p} + \tau logp + \mu n)$ (6)

2.1 Effect of number of processors (p)

Figure 1 shows the runtime of the algorithm as a function of number of processors at different n*n matrix with difficulty level=1. (n=100 means the matrix size is 100*100). As indicted in Equation 6, when n is small, the running time is dominated by the communication time: $O(\tau logp + \mu n^2)$, the larger the p, the longer the running time. When n is large, the running time is dominated by the computational time:$Iteration * O(\frac{n^2}{p})$. It is true in Figure 1. Since the difficulty=1 for all cases, Iterations are the same. When n=100, the communication time is dominant, the runtime increases with p. When n=40000, the computational time is dominant, the runtime decreases with p. When n=5000, there is a threshold of p=16. When p<16, the computation time is dominant, the runtime increases with p. When p>16, the communication time is dominant, the runtime increases with p. Similar situation happens for n=1000 and n=10000.

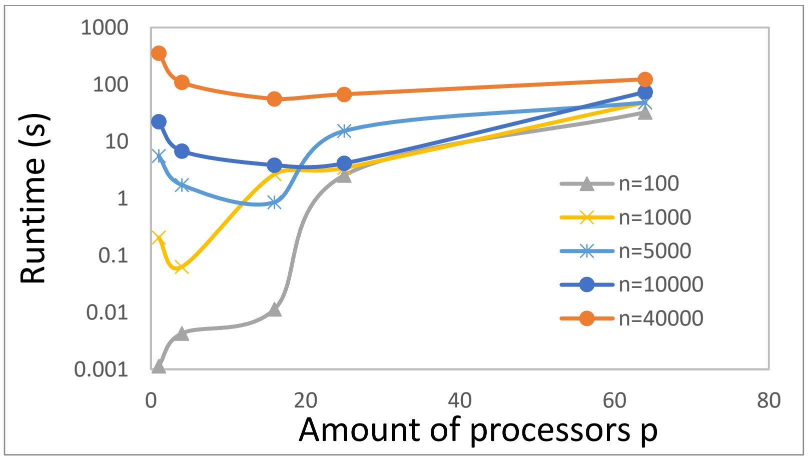

Figure 1. Runtime as a function of number of processors, p, at different n*n matrix with difficulty level=1.

2.2 Effect of Matrix size (n)

Figure 2 shows the runtime of the program as a function of matrix size at difficulty=1 with different number of processors. (n=100 means the matrix is 100*100) As indicated in Equation 6, when p is small, the runtime is dominant by the computational time $Iteration * O(\frac{n^2}{p})$. When p is large, the runtime is dominant by the communication time $O(\tau logp + \mu n)$. It is also true in figure 2. For p=4, p is relative small, the runtime is dominant by computational time $Iteration * O(\frac{n^2}{p})$. Thus, the runtime increases with matrix size n. For p=64, p is relatively large, the runtime is dominant by communication time $O(\tau logp + \mu n)$. Since in the lecture notes [2], $O(\frac{\tau}{\mu}) \approx 10000$. With n=$O$(10000) in the data, the communication time can be eliminated to $O(\tau logp)$. Thus, the runtime is not influenced by the matrix size n, but only decided by the constant number of processors p=64. As the n is getting larger to rise θ(10000), the term $O(\mu n)$ starts to count, and the runtime starts to increase with the matrix size n. It is true for p=64 with n=10000 and n=40000.

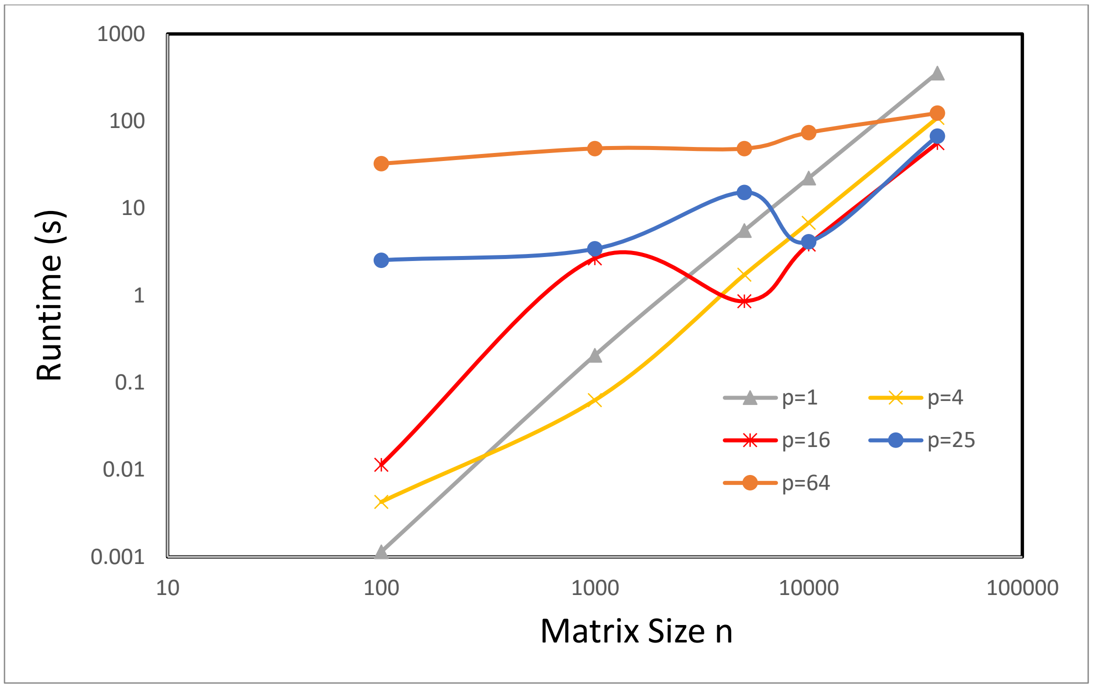

Figure 2. Runtime as a function of number of matrix size n at difficulty=1 with different number of processors.

2.3 Effect of Difficulty (d)

Figure 3 shows the runtime of the algorithm as a function of the difficulty level with 1000*1000 matrix with 4, 9 and 16 processors. The difficulty level determines the tolerance of l2. The higher the difficulty level, the larger the iterations. As indicated in Equation 6, the runtime is proportional to the iterations due to the expression $Iteration * O(\frac{n^2}{p} + \tau logp + \mu n)$. As shown in Figure 3, the runtime increases with the difficulty for p=4, 9 and 16. It is true since the difficulty determines the iteration number, which determine the runtime. The larger the difficulty, the larger the iteration, the larger the runtime.

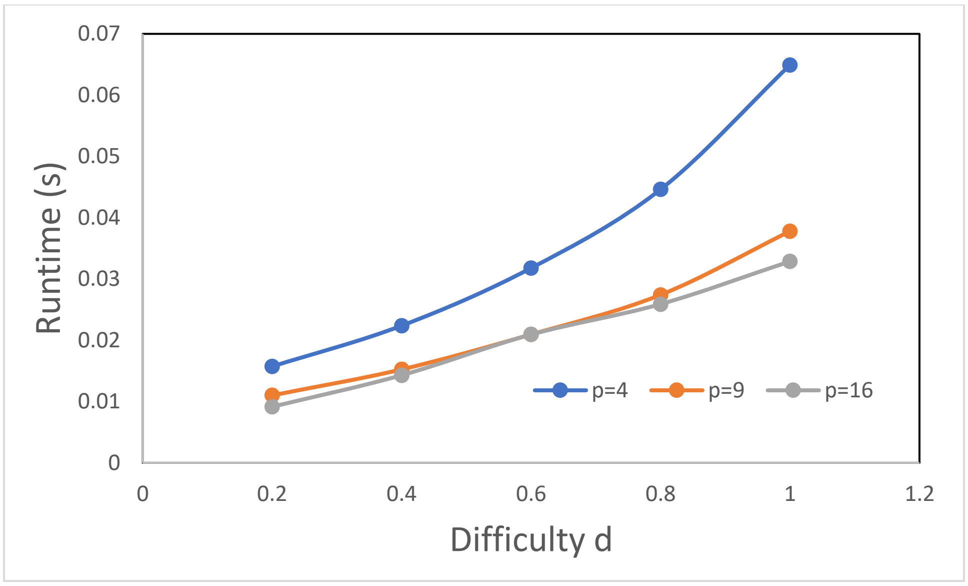

Figure 3. the runtime of the algorithm as a function of the difficulty level with 1000*1000 matrix with different number of processors.

2.4 Speedup and efficiency

The speedup and efficiency with 4 processors at different difficulty level and matrix size n are shown in Figure 4 and 5. In general, the speedup increases with the matrix size n. However, when the matrix size is large enough, the speedup starts to decrease with the matrix size. It indicates that the best speedup is achieved. For d=1, it is n=1000. For d=0.5, it is n=10000. For d=1, it is n=5000. Additionally, the speedup increases with the difficulty level. Since the larger the difficulty level, the more efforts it takes to compute, which makes the speedup more pronounced.

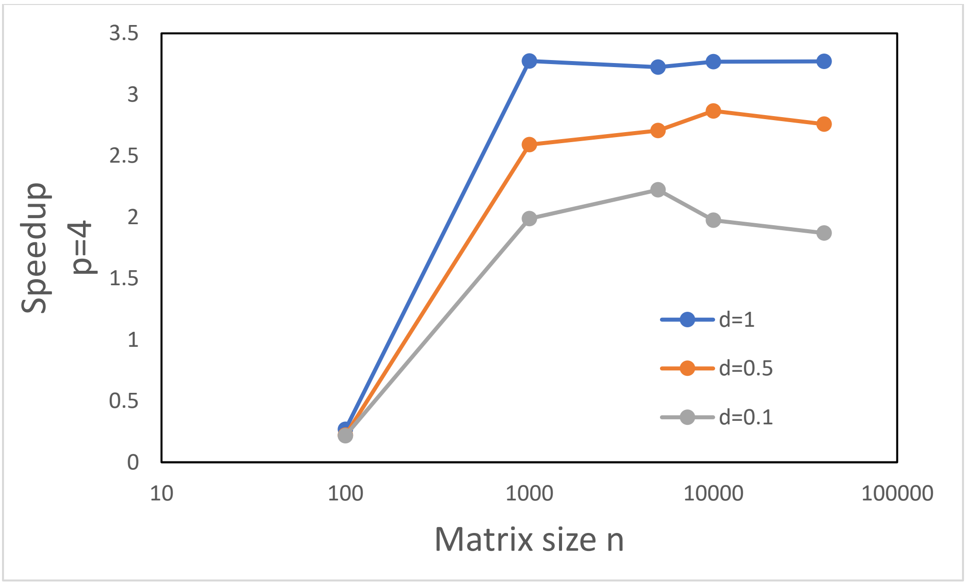

Figure 4: the speedup as a function of the matrix size n with different difficulty level d using 4 processors.

Figure 5 shows the efficiency with 4 processors at different matrix size and difficulty level. In all cases, no super speed up is achieved (Efficiency <1). According the figure, the optimal problem size is achieved when n>1000, since with n>1000, the efficiency>0.1 and can be treated as O(1).

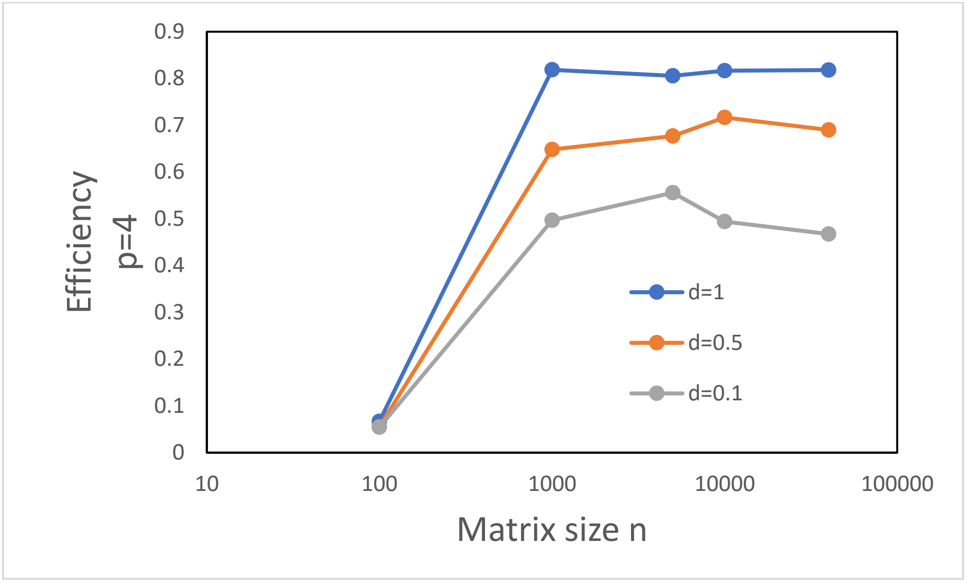

Figure 5: the efficiency as a function of the matrix size n with different difficulty level d using 4 processors.

3. Conclusions

In light of the above analysis, we can draw the following conclusions: 1) For constant n: At small n, runtime increases with p. At large n, runtime decreases with n. 2) For constant p: With small p, runtime increases with n. With large p, runtime is constant at the beginning and finally increases with n. 3) The runtime increases with the difficulty level. 4) The speedup increases with n and difficulty level. The optimal problem size is about n=1000.

Reference

[1] Umit V. Catalyurek, 2020 Spring, *CSE 6220 Programming Assignment Statement*.

[2] Umit V. Catalyurek, 2020 Spring, *CSE 6220 Lecture Notes*. A PARALLEL ALGORITHM EXAMPLE &MODEL OF PARALLEL COMPUTATION