Bilkent University
Department of Computer Engineering

CS 426
Project 1

Nisa Yılmaz
22001849

Description of Implementation

For the implementation, I have defined a struct called matrix and functions for creating, freeing, multiplying, adding, and applying sigmoid function to the matrix struct. This struct has a float** which stores the actual matrix and 2 other attributes for storing the dimensions.

The serial implementation is straightforward. One process multiplies the two matrices W and X, adds the bias vector by extending the vector to span all columns and applies the sigmoid function element wise.

In the parallel implementation, if the process is the master process it reads the text files for weight and input matrices as well as the bias vector. Then this process calculates the workload (number of rows each process will be working on) and sends this workload to every process except itself. It also sends the relative rows of the weight matrix and bias matrix, which start at the offset workload * process_id, as well as the whole input matrix. After sending the data to other processes, the master process calculates the rows it is responsible for. Then it waits for the other processes to send their output matrix for the rows they are responsible for and gathers all the rows in a single matrix. Then writes this matrix to a text file.

If the process is a worker process, it receives the workload, the rows of weight and bias that it is responsible for and the whole input matrix. Then it multiplies the matrices, adds the bias vectors relative rows and applies the sigmoid function. After obtaining the output matrix, it sends this to the master process to be combined with the other rows to form the final output matrix.

Tests

I have carried out tests with 3 different input sizes, 100x100 matrices (both weight and input has the same size) 500x500 matrices and 1000x1000 matrices. For all the different input sizes I ran the serial program as well as the parallel one with 2, 4, 5, and 10 processes.

Obtained Data and Graphs

The obtained data from the test runs I performed can be seen below in the table.

	Serial	2 Proc	4 Proc	5 Proc	10 Proc
100x100	0.010271	0.009586	0.010231	0.011718	0.029613
500x500	0.914746	0.596442	0.494437	0.473899	1.044057
1000x1000	7.16829	4.668509	3.532995	3.636937	4.022025

Table 1. Runtime Data

The graphs below show the runtime with different input sizes (Figure 1, Figure 2 and Figure 3) as well as the whole Table 1 (Figure 4).

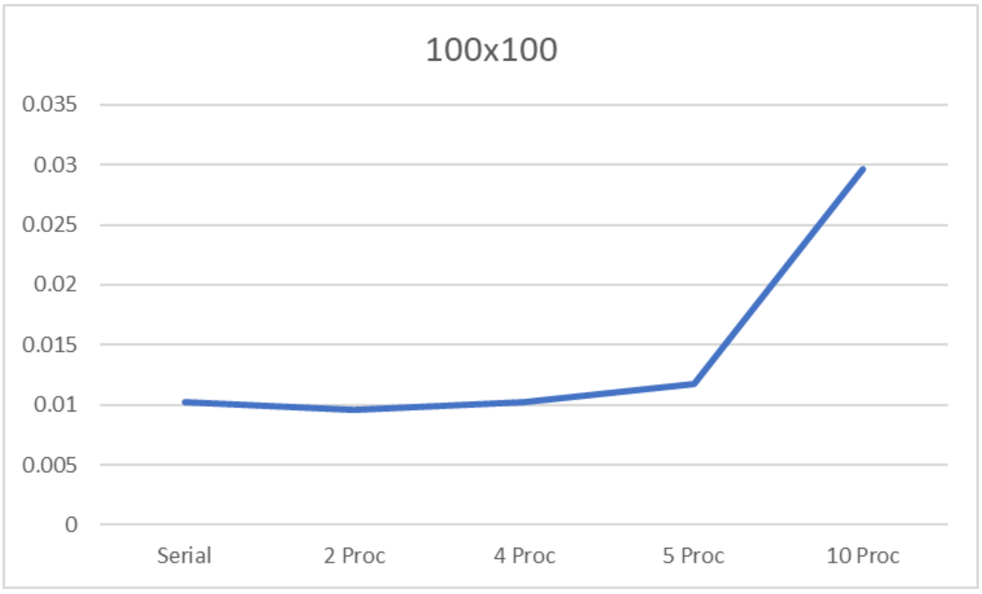

Figure 1. Runtime Graph for 100x100 Matrices

Looking at this graph, it can be observed that the parallel program with 2 processes runs as almost as fast as the serial implementation. This is because the data size is small and serial implementation works very fast. Even if there is speedup by parallel implementation, communication overhead may diminish the speedup. This can be seen from the grap. As the number of processes increase, the runtime of the parallel program exceeds that of the serial. Although in theory we expect to lower the runtime, communication overhead causes an increase in runtime. When this overhead is more than the speedup, the overall runtime increases. In this case, it means that the subproblems we have created are too small.

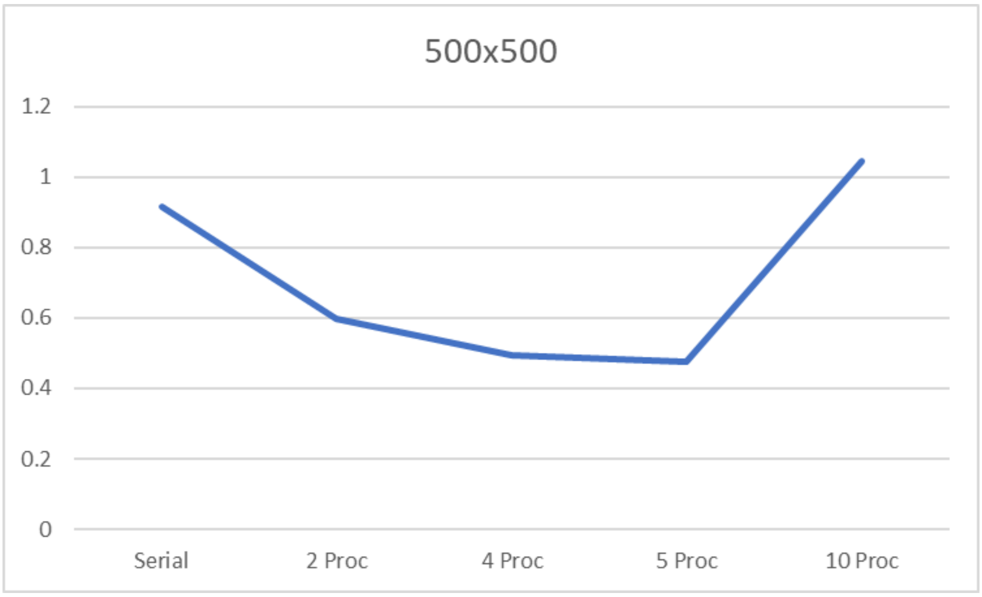

Figure 2. Runtime Graph for 500x500 Matrices

The graph for 500x500 matrices display an expected behavior. For the parallel implementation with 2, 4, and 5 processes the runtime decreases compared to the serial implementation. But as the number of processes increase to 10, the overhead also increases making the overall runtime longer.

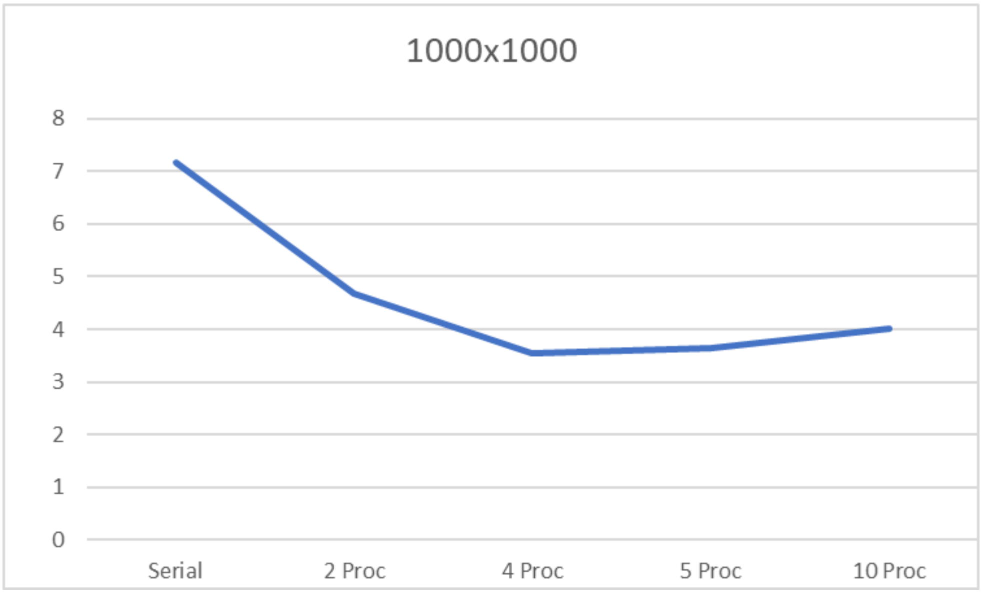

Figure 3. Runtime Graph for 1000x1000 Matrices

From the graph of 1000x1000 matrices it can be seen that there has been a considerable speedup. The maximum speedup is achieved by using 4 processes. Increasing the number of processors further slows the execution because of communication overheads. Compared to the other input sizes, the speedup with larger data is more significant, in this case 2 times.

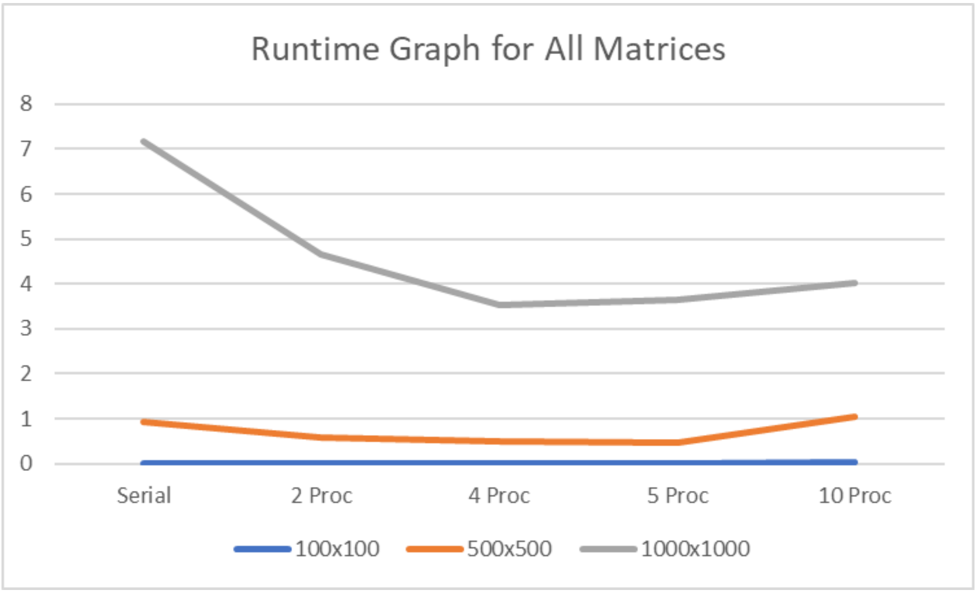

Figure 4. Runtime Graph for All Matrices

By looking at this graph it can be seen that parallel implementation with 10 processes has increased the runtime for all inputs. This is an unwanted situation and is caused by dividing the problem into very small subproblems. In this case sending and receiving the messages between processes is greater than the speedup caused by parallel implementation. In theory, there should be a 10 times speedup since the workload is divided by this number, but in practice the overhead causes a longer runtime.

Another thing that can be observed is that the runtime reduction in small and medium sized inputs are almost negligible compared to the bigger sized input. Since executing the serial program with small and medium sized inputs already takes a little time, the speedup gain is not very considerable. But with the bigger input size, there is a noticeable speedup.

Considering the results of this experiment, to gain maximum speedup from a parallel implementation it is very important to choose the subproblem size so that the overhead caused by communication is much less than the speedup. Also, parallel computing does not have a great effect on problems with smaller data but as the data gets bigger parallel programs can be used to decrease runtime and utilize multicore/multiple cpu architectures more efficiently.