CS 566: Parallel Processing
Homework 2 - Design Document

Zohair Hashmi – UIN 668913771

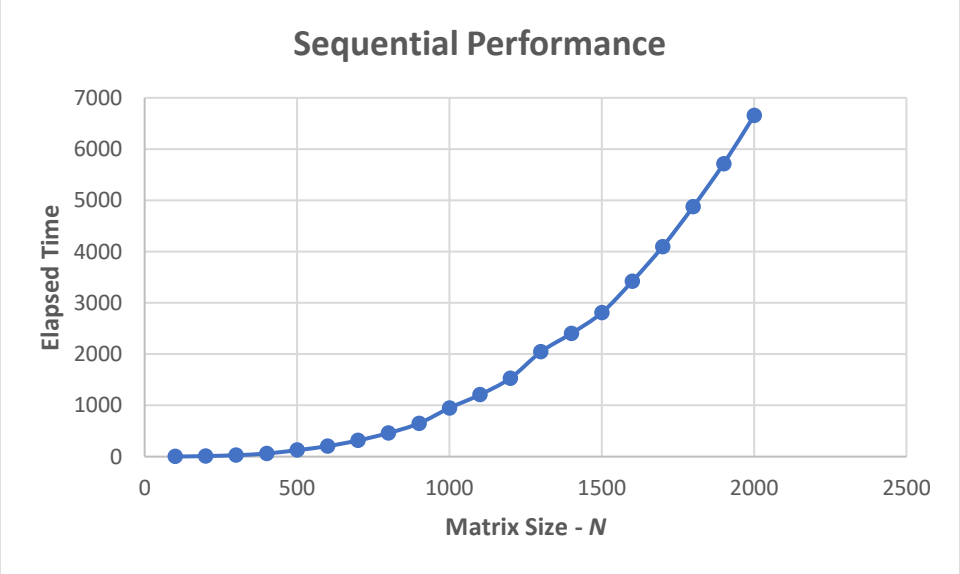
Introduction:

In this experiment, we run the gaussian elimination code sequentially and parallelly to compare their performance in terms of **Elapsed Time** and evaluate the results on varying number of threads and matrix size (N).

The following sections describe the tests that were conducted for each type of processing method and their respective observed results. We discuss the difference in the results observed for each of the processing methods and the optimization and enhancements made over time.

Sequential Processing:

The given "gauss.c" code was executed on Chameleon's Virtual Machine for varying iterations of N, ranging from N = 100 to 2000 with an interval of 100. The following is the graph displaying the performance of our sequential code w.r.t the size of the matrix.



As we can see, increase in matrix size leads to an exponential increase in elapsed time to run the executable file. For analysis in the next sections, we have used N=1000 to conduct experiments on different number of threads using the parallel processing method. This is because, N=1000 takes around 950ms to execute, so that observing performance over varying threads will not be very time taking. N is also large enough for the algorithm to scale and demonstrate visible performance differences.

Pthread Parallel Processing:

The Pthread Parallel Processing task of this assignment is carried out in the following steps:

1. Modify the Gaussian Elimination routine for parallel processing using Pthread.
2. Compile the code for varying number of threads (1 to 20, step size=1) and plot the results.
3. Identify the optimal number of threads, specifically the one with the lowest elapsed time.
4. Fix the NTHREADS to the optimal number of threads recorded and compile the code using varying matrix size ranging from N=100 to 2000 with a step size of 100.
5. Plot the results to compare with Sequential Processing.

Code Modification:

Libraries included: #include <pthread.h>

Global variables introduced:

- CHUNK_SIZE – Used to distribute the execution of 'for' loop in Gaussian Elimination task into chunks that can be fed to a single thread.
- Global_row - to keep track of the row that is being processed.
- NTHREADS – to define number of threads to be used.
- Global_row_lock – initialized a *pthread_mutex_t* to lock the global_row variable.
- Barrier – initialized a *pthread_barrier_t* to synchronize the threads.

The following modifications are made in the implementation of the routine:

Thread Function:

- Introduced a new function *gauss_thread* that serves as the entry point for each thread. Each thread performs part of the Gaussian elimination, performing back substitution after the Gaussian elimination phase outside the parallelized section, as it does not require parallel execution.
- The *global_row* variable is used to keep track of the row being processed globally. A mutex (*global_row_lock*) is used to ensure atomic updates to this global variable.

Parallelization Strategy:

- The outer loop (*norm* loop) of the Gaussian elimination is parallelized by dividing the work among threads. Each thread processes a chunk of rows (*CHUNK_SIZE*) determined by the global row variable.
- A mutex is used to lock and update the *global_row* variable among threads to avoid data races.
- Created an array of *pthread_t* to represent the threads.
- Used *pthread_create* to spawn threads, and *pthread_join* to wait for the completion of all threads.

Barrier Synchronization:

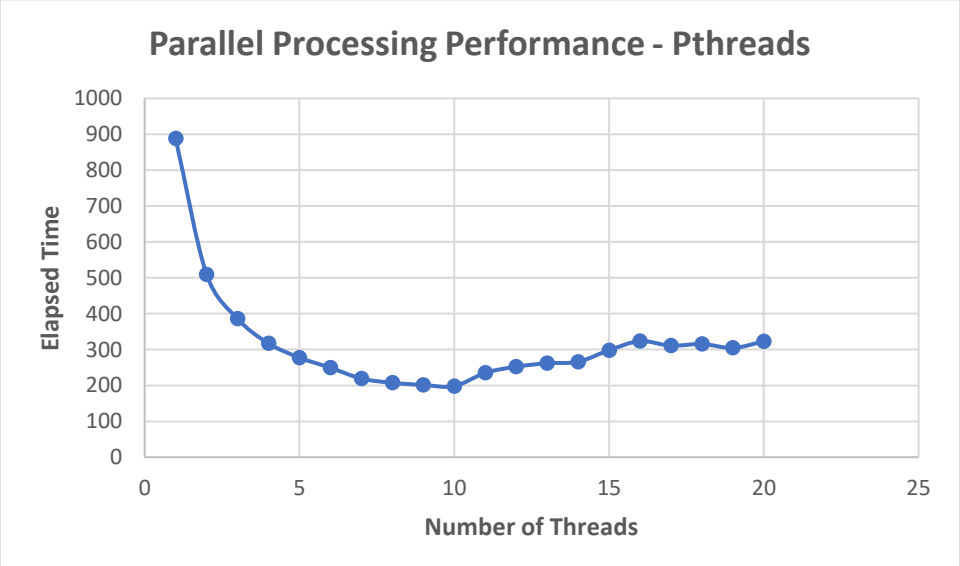
- Introduced a barrier (*pthread_barrier_t*) to synchronize threads after completing the Gaussian elimination phase. This ensures that all threads have finished their portion before proceeding to the next step.
- Cleaned up thread-related resources using *pthread_mutex_destroy* for the mutex and *pthread_barrier_destroy* for the barrier.

Performance Evaluation:

After executing the code for number of threads from 1 to 20, we obtain the following graph which shows a rapid decrease in elapsed time when the number of threads is initially increased as the benefits of parallelism outweigh the overhead.

The decrease in the elapsed time slows down with increase in threads before reaching optimal thread count and increasing again. This is in accordance with Amdahl's Law and the resource contention between threads.

The observed optimal number of threads for using Pthreads is 10 with an elapse time of 198ms.



OpenMP Parallel Processing:

The OpenMP Parallel Processing task of this assignment is carried out in the following steps:

1. Modify the Gaussian Elimination routine for parallel processing using OpenMP.
2. Compile the code for varying number of threads (1 to 60, step size=2) and plot the results.
3. Identify the optimal number of threads, specifically the one with the lowest elapsed time.
4. Fix the NTHREADS to the optimal number of threads recorded and compile the code using varying matrix size ranging from N=100 to 2000 with a step size of 100.
5. Plot the results to compare with Sequential Processing.

Code Modification:

Libraries included: #include <omp.h>

Global variables introduced:

- NTHREADS – to define number of threads to be used.

The following modifications are made in the implementation of the routine:

Gauss Function:

- The outer loop of Gaussian elimination, which iterates over the normalization step, is parallelized using OpenMP. The *#pragma omp parallel for* directive is used to distribute the iterations of the outer loop among multiple threads.
- Parallelizing only the Gaussian Elimination loop while performing back substitution later outside the parallelized section, as it does not require parallel execution.

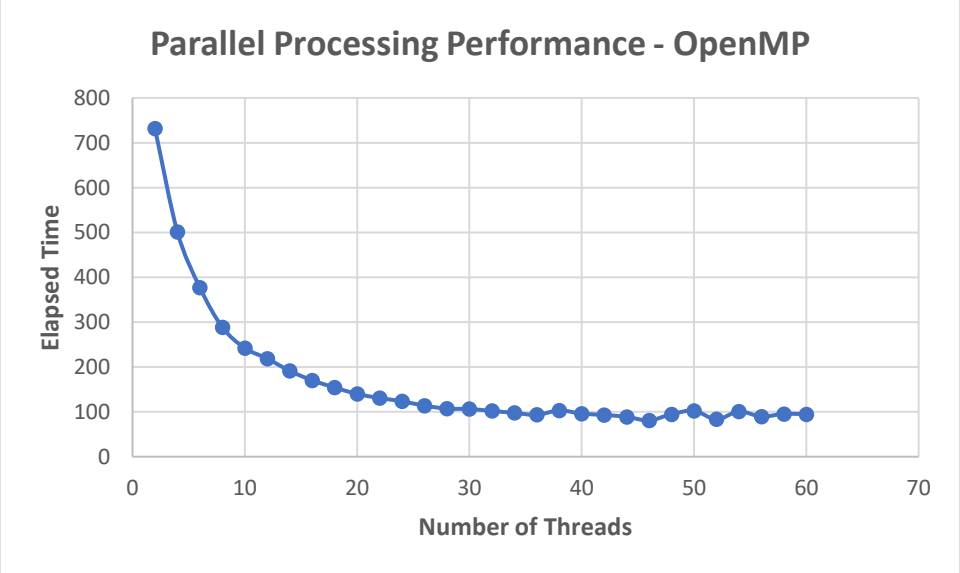
Private and Shared Variables:

- A private clause is added to the *#pragma omp parallel* directive to specify that each thread should have its private copies of the loop index variables *(norm, row, col, multiplier*). The shared clause is used to indicate that the arrays *A, B,* and *X* are shared among all threads.

Performance Evaluation:

After executing the code for number of threads from 1 to 60, we obtain the following graph which shows a similar declining trend to the Phtread experiment graph, only that after about 28 threads any increase in number of threads has insignificant change to the execution time.

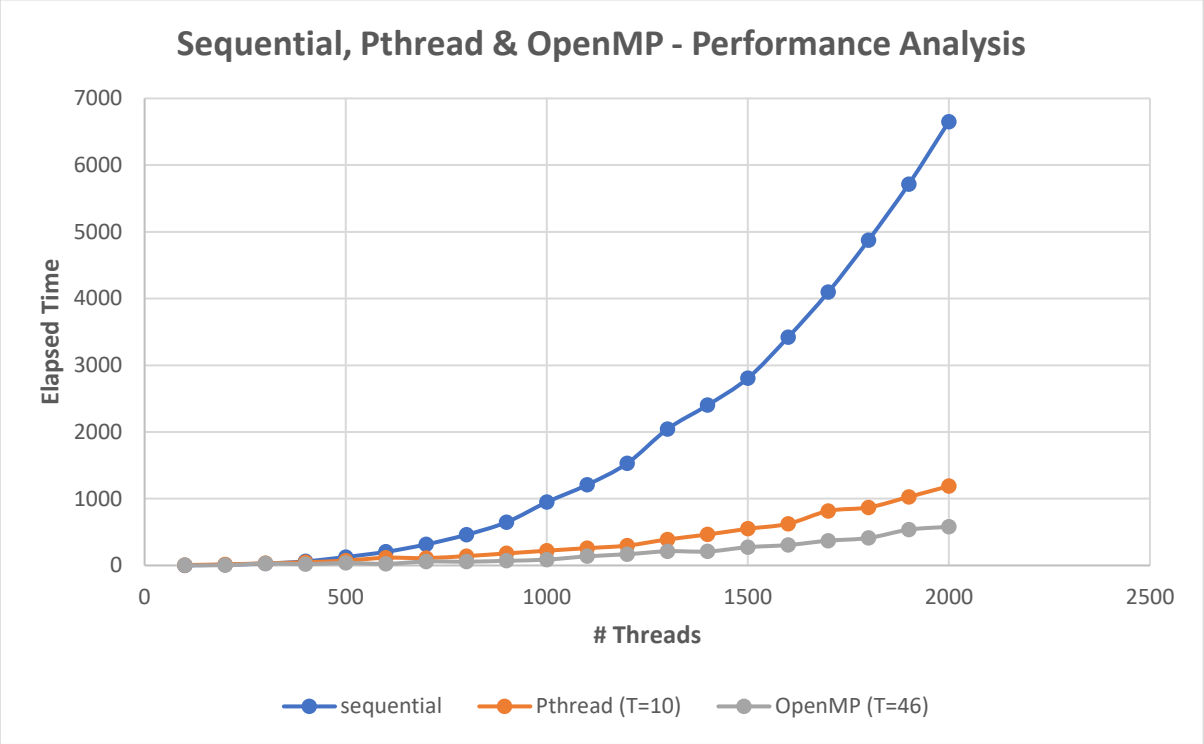
The observed optimal number of threads for using OpenMP is 46 with an elapse time of 81ms.



Conclusion:

As visualized in the graph below, the following comparisons are made about the performance of each processing type (sequential, parallel pthread, and parallel openmp) for their optimal number of threads (no threads for sequential processing):

- The comparison highlights that increasing the number of threads does increase the execution time exponentially for each of the processing types.
- The parallelization of routines has significant impact on reducing the overall execution time for each respective thread.
- OpenMP tends to perform best (even better than Pthread) when N is large.
- In my opinion OpenMP is reasonably efficient than the other two methods as it is the most convenient to code and leads to the best performance. It has better distribution of iterations to each thread without changing the structure of the sequential code itself.



Experimental Results:

Sequential, Pthread & OpenMP results for N 100 to 2000:

N (matrix size)	sequential	Pthread (T=10)	OpenMP (T=46)
100	1.087	6.316	1.985
200	11.344	15.8	4.183
300	27.611	32.6	29.988
400	61.083	47.333	21.074
500	126.672	73.434	37.868
600	203.982	114.996	25.071
700	315.405	110.748	58.576
800	460.02	140.432	56.501
900	647.93	180.161	69.777
1000	947.933	221.555	87.881
1100	1207.27	259.08	136.388
1200	1529.72	297.962	168.375
1300	2043.3	389.778	212.734
1400	2401.88	463.32	209.574
1500	2806.02	549.772	273.39
1600	3419.3	623.571	306.254
1700	4096.63	817.298	371.044
1800	4874.19	869.202	411.85
1900	5712.44	1027	536.423
2000	6651.51	1188.58	579.864

Pthreads: Parallel Processing results.

Number of threads 1 to 20.

#Threads	Elapsed Time
1	888.256
2	509.428
3	385.943
4	317.314
5	277.629
6	249.35
7	219.316
8	207.331
9	201.367
10	198.061
11	235.008
12	252.242
13	261.889
14	265.96
15	298.024
16	323.545
17	310.734
18	315.56
19	304.609
20	322.657

OpenMP: Parallel Processing results.

Number of threads 2 to 60.

#Threads	Elapsed Time
2	731.868
4	501.369
6	377.376
8	288.887
10	242.391
12	218.811
14	191.441
16	170.278
18	154.335
20	140.095
22	130.954
24	123.592
26	113.814
28	107.3
30	106.388
32	102.043
34	97.858
36	93.516
38	103.106
40	96.24
42	93.153
44	89.006
46	81.055
48	94.871
50	102.482
52	83.521
54	100.897
56	89.658
58	95.538
60	94.887