Performance Report - Multithreading Capabilities of a Machine

In this study, we are scrutinizing the multithreading capabilities of a given machine. This report is focused on analyzing the performance of a machine by observing CPU utilization and computation time to run the algorithm in different scenarios. We are going to perform matrix multiplication and Array Summation on different data sets.

Machine used:
Macbook Pro (Apple M1 Pro)
RAM: *16 GB*
Hard Drive: *512 GB SSD*
Operating System: *macOS Monterey (version 12.3)*

Performed by: Mahima Gupta (T22055) and Prashant D Kulkarni (T22058)

Parallel Matrix Multiplication

1. Calculating size of matrix for multiplication

For Estimating size of matrix, we used *iterative improve* method. We used square matrices for multiplication. For multiplication, Matrix A has been split into two parts whereas matrix B is used as is. Values were matched against computation time as mentioned below -

Size of Matrix	Computation Time (seconds)
[3500 x 3500]	52.034
[6500 x 6500]	594.619
[7000 x 7000]	750.25
[7500 x7500]	939.379

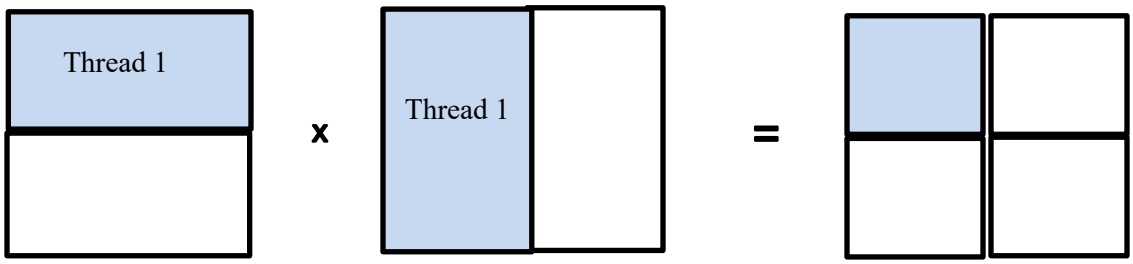
*It can be concluded from above experiment that **[6500x6500]** is the optimal choice for conducting further experiments. Since, it took nearly 10 minutes to perform matrix multiplication.*

2. Matrix Multiplication by Splitting both Matrices Using 4 Threads

Here, we are multiplying Matrix A by Matrix B. Splitting Matrix A into two parts, horizontally. Splitting Matrix B into two parts, vertically. We are using 4 threads for implementation.

⇒ Thread 1 will multiply the first half of Matrix A with the first half of Matrix B.
⇒ Thread 2 will multiply the first half of Matrix A with the second half of Matrix B.

⇒ Thread 3 will multiply the second half of Matrix A with the first half of Matrix B.
⇒ Thread 4 will multiply the second half of Matrix A with the second half of Matrix B.



Similarly, remaining thread will perform multiplication on remaining halves respectively.

Size of Matrix	Computation Time for Matrix Multiplication (seconds)	CPU Utilization
[6500 x 6500]	285.944	51.4%
[6500 x 6500]	304.756	54.8%
[6500 x 6500]	282.288	52.85%

3. Matrix Multiplication by Splitting both Matrices using 2 Threads

Here, we are multiplying Matrix A by Matrix B. Splitting Matrix A into two parts, vertically. Splitting Matrix B into two parts, horizontally. We are using 2 threads for implementation.

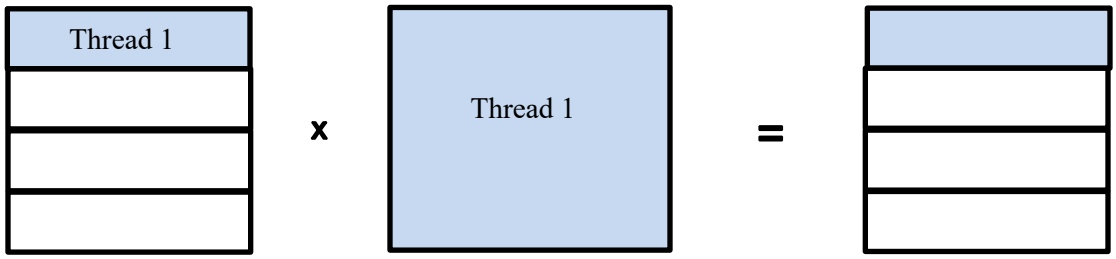
⇒ Thread 1 will multiply the first half of Matrix A with the first half of Matrix B.
⇒ Thread 2 will multiply the second half of Matrix A with the second half of Matrix B.

We have divided the resultant matrix into four regions. *Only one thread can access only one region at a time.*

Size of Matrix	Computation Time for reading Input Values (seconds)	Computation Time for Matrix Multiplication (seconds)	CPU Utilization
[6500 x 6500]	1.065	> 2615.77	43.9 %

4. Matrix Multiplication by Splitting only Matrix A using N Threads

Here, we are multiplying Matrix A by Matrix B. Splitting Matrix A into n equal parts, horizontally. No Splitting on Matrix B. We are using n threads for implementation, one thread for each part.

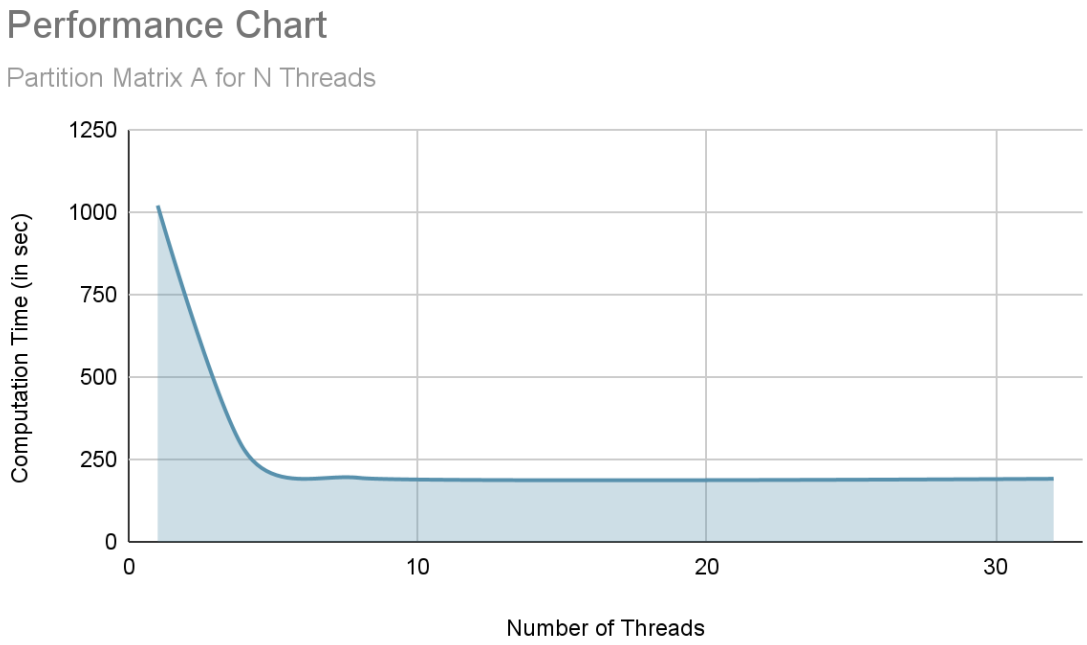


Size of Matrix	N - Number of Threads	Average computation Time	Computation Time for Matrix Multiplication (seconds)	CPU Utilization
[6500 x 6500]	1	1020.11	1028.06 | 1014.23 | 1018.04	13.36 %
[6500 x 6500]	4	278.543	286.123 | 276.978 | 272.528	54.38%
[6500 x 6500]	8	193.729	193.218 | 187.707 | 200.263	96.52%
[6500 x 6500]	32	191.085	193.587 | 192.44 | 187.23	98.97 %

The **speedup** of a parallel algorithm over a corresponding sequential algorithm is the ratio of the compute time for the sequential algorithm to the time for the parallel algorithm. We are achieving parallelism using multithreading over conventional single-core programming.
Speedup = (Computation Time using 1 Thread / Computation Time using N threads)

Number of Threads	Computation Time	SpeedUp
4	278.543	3.633
8	193.729	5.225
32	191.085	5.338

From above readings, we can conclude that as the degree of multi-threading increases, speedup increases as well. Henceforth, resulting in significant reduction in computation time for the given task. CPU Utilization increases with increasing degree of multi-threading. Maximum utilization has been achieved when number of threads is equal to number of cores present in the machine used. After that threshold, the CPU utilization doesn't increase significantly.

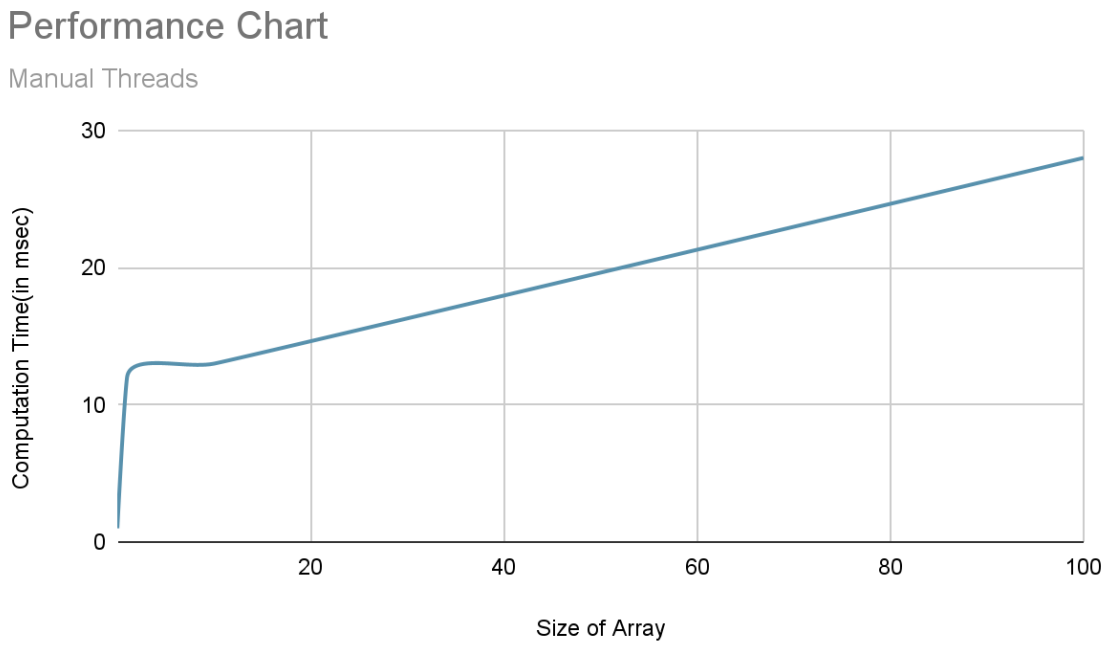


Array Summation

5. Summation of n elements in an Array Using 4 Threads

We are adding n elements present in a 1D Array. We are dividing the array into 4 parts. We are storing summation of each part in another array. And then finally, summing up the values stored in the summation array.

Size of Array	Computation Time for Summation (milliSeconds)	CPU Utilization
100	1	4.5%
1000000	12	5.7%
10000000	13	7.8%
100000000	28	17.45%

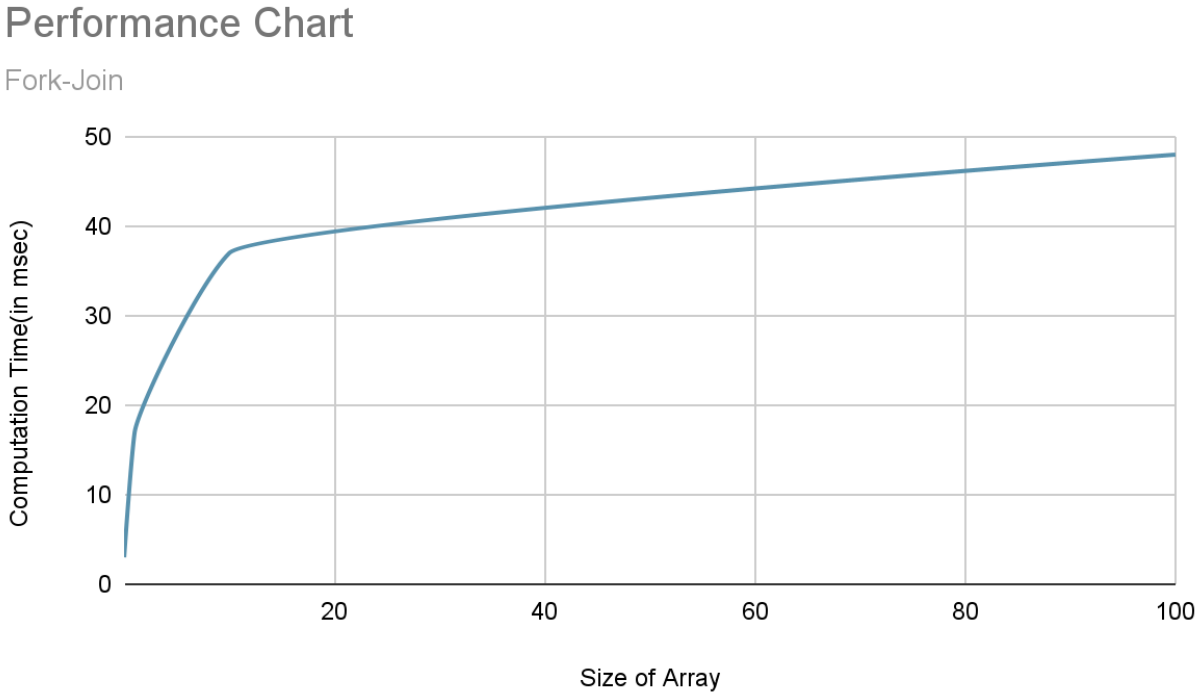


6. Summation of n elements in an Array Using Fork-Join

We are using Fork-Join Pool to compute the sum of n elements in an array.

Size of Array	Computation Time for Summation (milliSeconds)	CPU Utilization
100	3	4.65%
1000000	17	8.47%
10000000	37	13.28%
100000000	48	15.69%

Size	Computation Time using Fork Join	Computation Time using Threads	SpeedUp
100	3	1	3
1000000	17	12	1.41
10000000	37	13	2.84
100000000	48	28	1.71

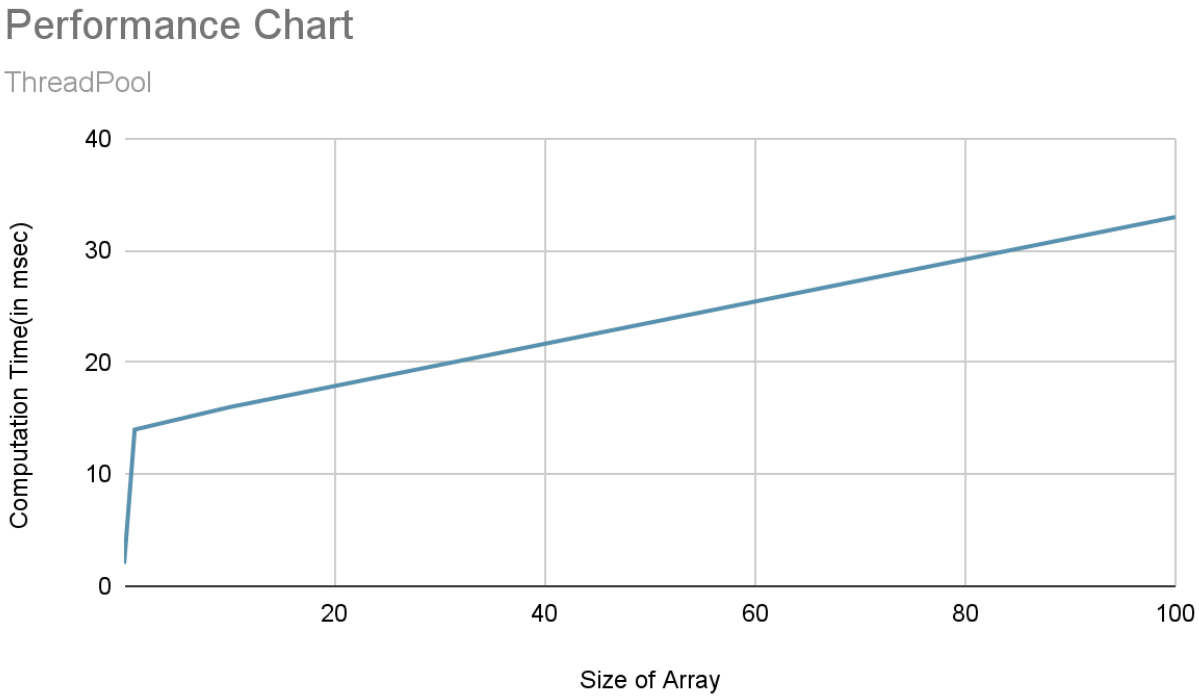


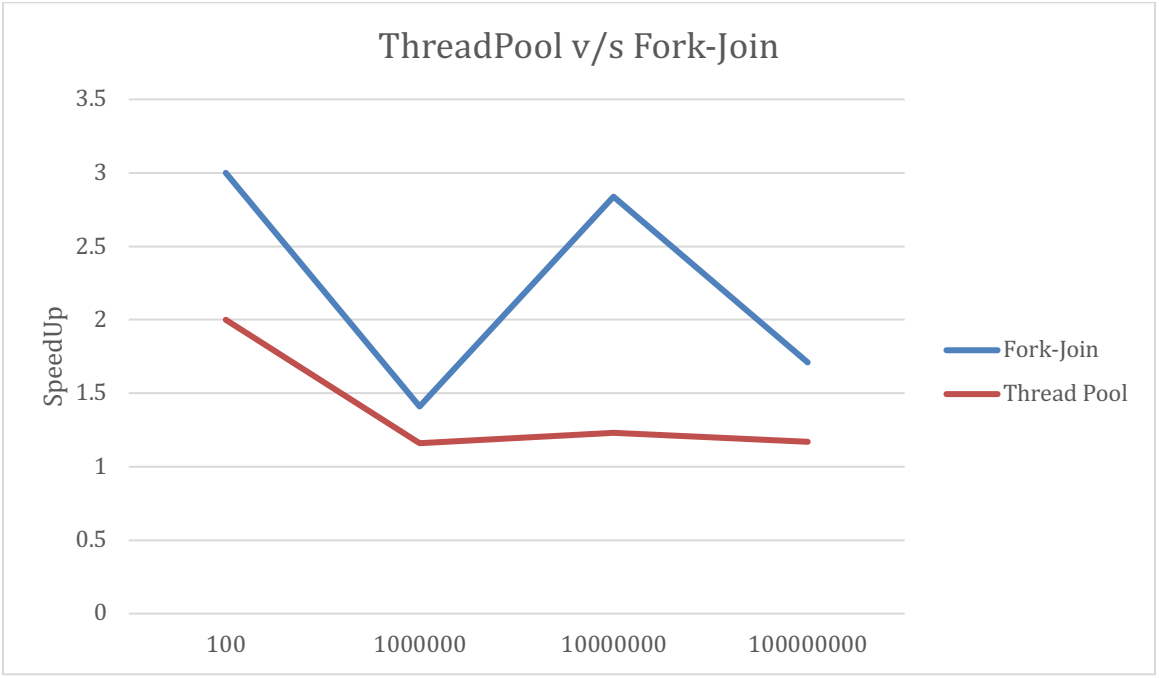
7. Summation of n elements in an Array using Thread Pool

We are using Thread pool for computing the sum of n elements in an array.

Size of Array	Computation Time for Summation (milliSeconds)	CPU Utilization
100	2	4.77%
1000000	14	5.17%
10000000	16	7.0%
100000000	33	11.69%

Size	Computation Time using Thread Pool	Computation Time using Threads	SpeedUp
100	2	1	2
1000000	14	12	1.16
10000000	16	13	1.23
100000000	33	28	1.17





It can be concluded from the above graph that fork-join is delivering better speedup than thread-pool over given datasets.

Logger:

Log4j (1.2.27) has been used to create log files. We are storing logs in app.log. We have implemented Activity ID for better tracking. Activity ID is random UUID.
Format for logs –
Activity ID: [%X{userName}]:%d{yyyy-MM-dd HH:mm:ss} (Log_type) (Message)

```
Activity ID: [66c07718-36ce-453a-86da-c1a3e14c8682]:2022-09-20 12:08:33 INFO  Main -
Strategies that user can input :
For Matrix Multiplication = HorVerSplit, VerHorSplit, HorSplit
For Array sum = ForkJoinPool, ThreadPool, ManualThread
Activity ID: [66c07718-36ce-453a-86da-c1a3e14c8682]:2022-09-20 12:08:33 INFO  Main -
User Input Details :
Computation Type : arraySum | Strategy : ThreadPool | Array Size: 10000000
Activity ID: [25679387-af3e-4ff0-850a-56a0aa34895c]:2022-09-20 12:08:33 INFO  ArraySum - Initializing an Array of size - 10000000
Activity ID: [70813147-d1fa-407e-8261-3bdbfb830ac5]:2022-09-20 12:08:34 INFO  ArraySum - Using ThreadPool
Activity ID: [70813147-d1fa-407e-8261-3bdbfb830ac5]:2022-09-20 12:08:34 INFO  ArraySum - Sum : 5005003470
Activity ID: [70813147-d1fa-407e-8261-3bdbfb830ac5]:2022-09-20 12:08:34 INFO  ArraySum - Time Taken for summation : 0.016sec
Activity ID: [718594db-29a7-476a-a5e7-71b91fd61529]:2022-09-20 12:08:52 INFO  Main -
```