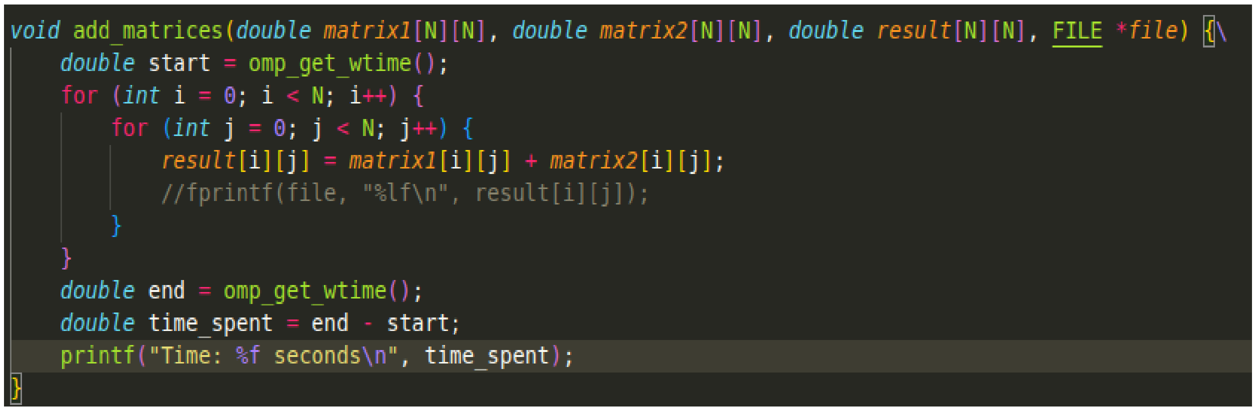
HPC OPENMP TUTORIAL 5 REPORT

CS22B2015 – HARSHITH B

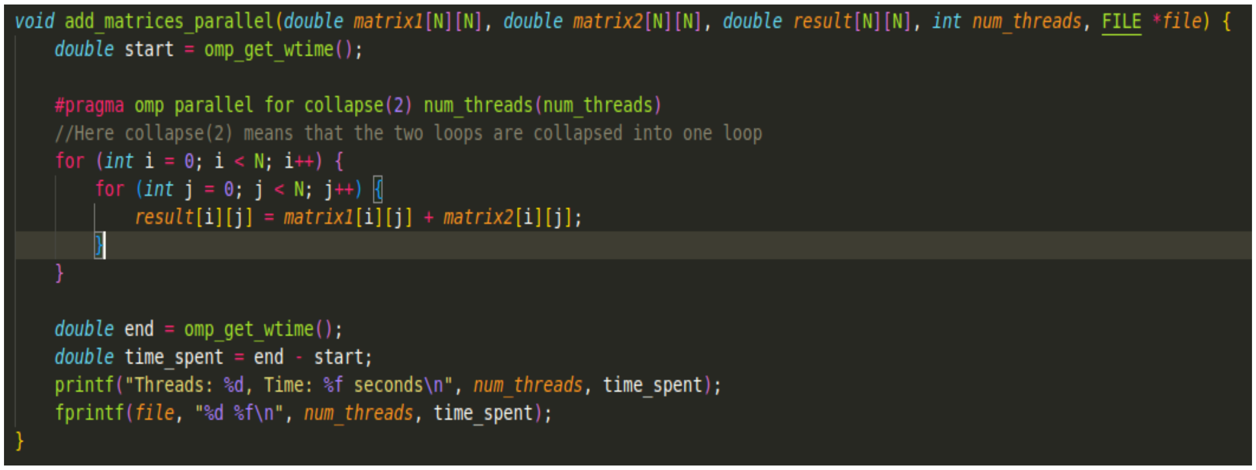
The serial and parallel code for matrix addition is provided in the serial.c and parallel.c file. The key section of the code performs the following steps:

1. Threads are allocated in powers of 2 (1, 2, 4, 8, 16, 32, 64, 128) within a loop, and the sum is calculated using the OpenMP pragma reduction. The serial code runs by default in 1 thread and therefore OpenMP is not being used.
2. Two datasets consisting of approximately 10^8 double-precision floating-point numbers is generated using Python and stored in the output1.txt and output2.txt files.
3. The serial and parallel code is converting this 10^8 into 10^4 x 10^4 matrices A and B and OpenMP is being implemented only for the addition part of A+B.
4. The threads versus time data is recorded into a text file. This data is then processed, and the respective results are plotted and analyzed using results.py.

1. Serial and Parallel Code Segment for Matrix Addition



```c
void add_matrices(double matrix1[N][N], double matrix2[N][N], double result[N][N], FILE *file) {
    double start = omp_get_wtime();
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
            //fprintf(file, "%lf\n", result[i][j]);
        }
    }
    double end = omp_get_wtime();
    double time_spent = end - start;
    printf("Time: %f seconds\n", time_spent);
}
```

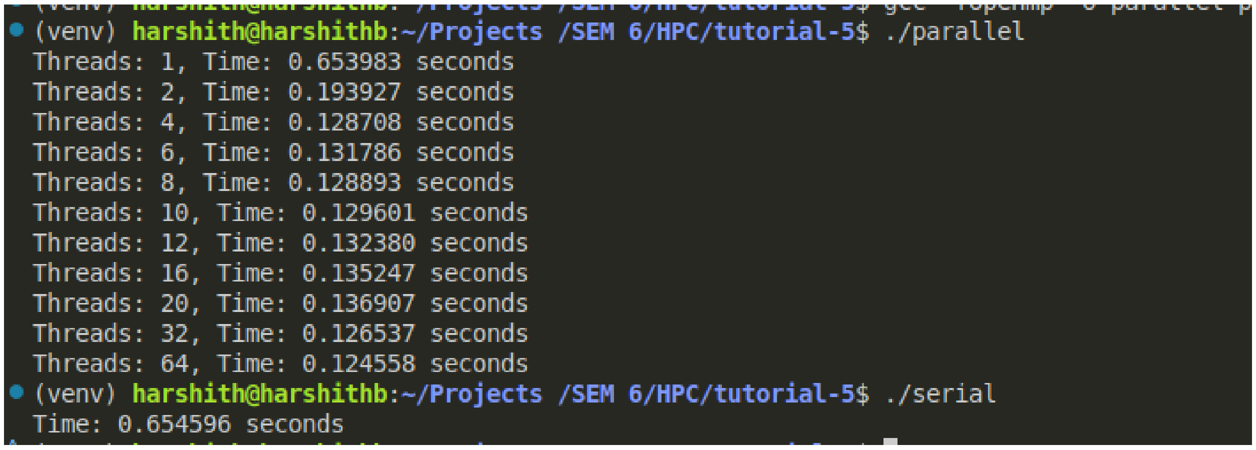


```c
void add_matrices_parallel(double matrix1[N][N], double matrix2[N][N], double result[N][N], int num_threads, FILE *file) {
    double start = omp_get_wtime();

    #pragma omp parallel for collapse(2) num_threads(num_threads)
    //Here collapse(2) means that the two loops are collapsed into one loop
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }

    double end = omp_get_wtime();
    double time_spent = end - start;
    printf("Threads: %d, Time: %f seconds\n", num_threads, time_spent);
    fprintf(file, "%d %f\n", num_threads, time_spent);
}
```

2. Terminal Output

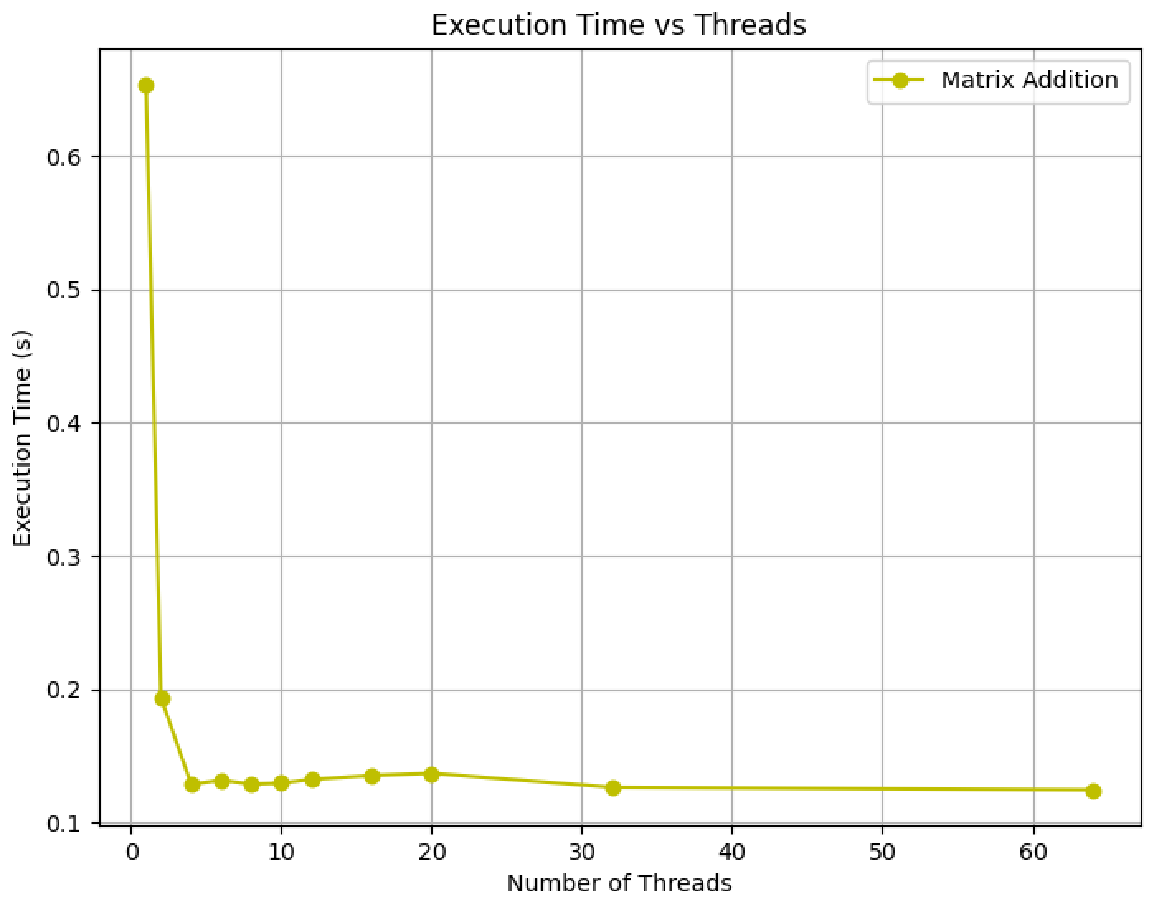


```
(venv) harshith@harshithb:~/Projects /SEM 6/HPC/tutorial-5$ ./parallel
Threads: 1, Time: 0.653983 seconds
Threads: 2, Time: 0.193927 seconds
Threads: 4, Time: 0.128708 seconds
Threads: 6, Time: 0.131786 seconds
Threads: 8, Time: 0.128893 seconds
Threads: 10, Time: 0.129601 seconds
Threads: 12, Time: 0.132380 seconds
Threads: 16, Time: 0.135247 seconds
Threads: 20, Time: 0.136907 seconds
Threads: 32, Time: 0.126537 seconds
Threads: 64, Time: 0.124558 seconds
(venv) harshith@harshithb:~/Projects /SEM 6/HPC/tutorial-5$ ./serial
Time: 0.654596 seconds
```

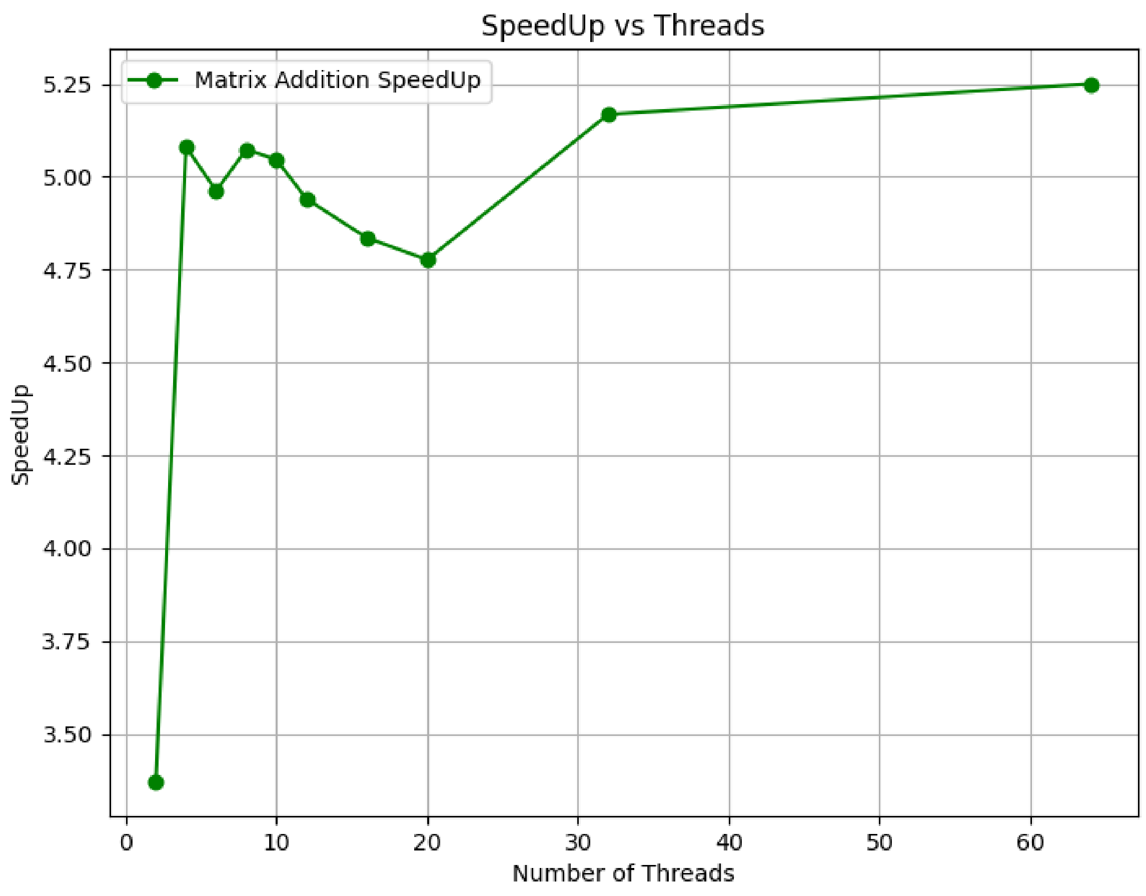
3. Thread vs Time Plots and Observations

Observations :

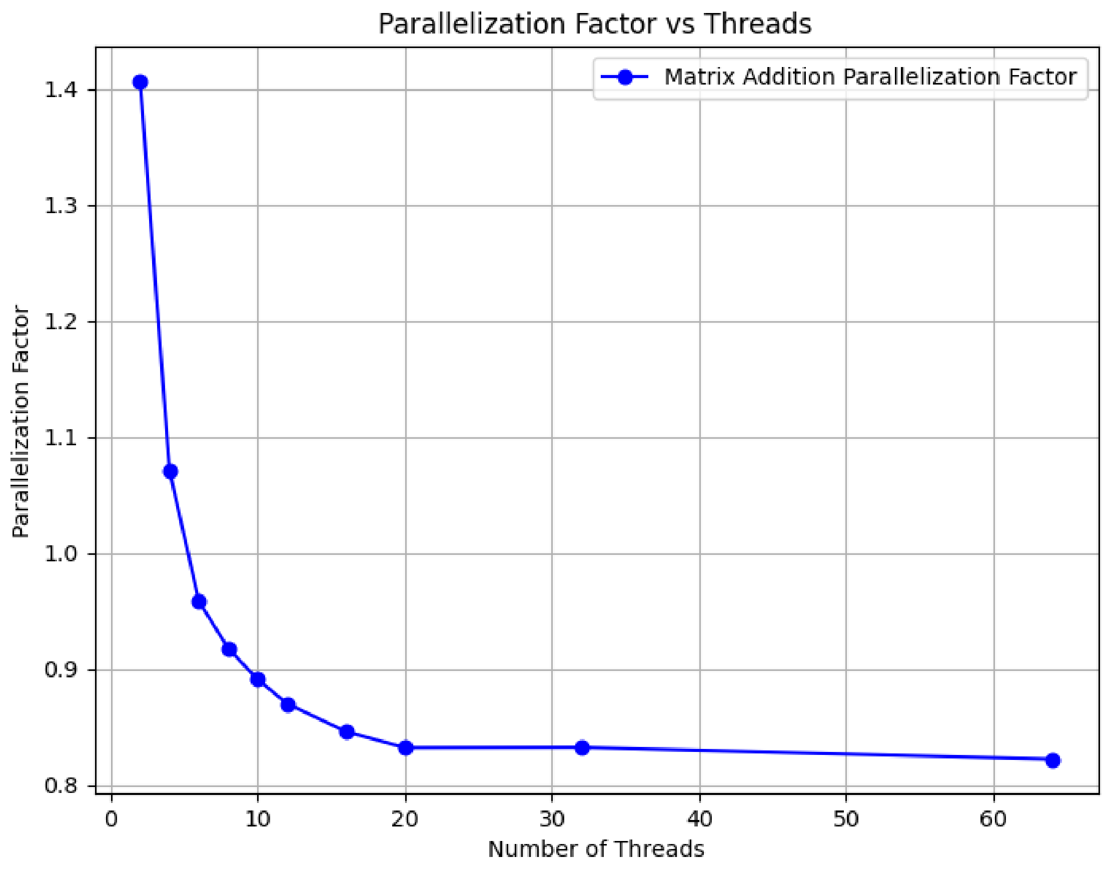
- **Initial Improvement:** There is a significant reduction in execution time as the number of threads increases from 1 to around 10 threads. This indicates effective parallelization at lower thread counts.
- **Diminishing Returns:** Beyond a certain point (e.g., around 30 threads), the decrease in execution time becomes less pronounced. This is typical due to overheads like thread management and memory contention.
- **Performance Plateau:** As the number of threads increases further (e.g., above 32), the execution time stabilizes or even slightly decreases.
- **Optimal Thread Count:** The best performance is observed at a 64 threads.
- The above observations show that the threads vs time is not too stable up until 32 threads due to the fact that there is a huge dataset with NxN matrices with N being 10000 numbers and hardware limitations.



4. SpeedUp vs Processors (SpeedUp == T(1) / T(n))



5. Parallelization Fraction and Inference



-

- ***Highest Factor at 2 Threads**: The parallelization factor starts at its maximum value when using two threads, as there is less parallel overhead. But WRT the least time (BEST CASE SCENARIO) at 64 threads the Factor is around 0.85.*

- **Drop with Increased Threads**: As the number of threads increases (e.g., 2 to 16 threads), the parallelization factor decreases, reflecting diminishing returns in performance improvement due to parallel overhead.
- **Flattening for Higher Threads**: Beyond a certain number of threads (e.g., 20 threads), the parallelization factor stabilizes, indicating that adding more threads does not significantly improve parallel efficiency.
- **Slight Recovery at 64 Threads**: A small upward trend is observed at 64 threads, possibly due to workload distribution or hardware optimization effects.