HPC Tutorial 5 Report
CS22B2012
K Aditya Sai

The code for Matrix Addition is implemented in C++ with OpenMP.

1. **Thread Allocation**: The code is executed with thread counts ranging from 1 to 64

2. **Matrix Generation**: A C++ code generates matrices of dimension 10000 x 10000 with double precision floating point elements ranging from 1-100

3. **Performance Analysis**: Execution times for the addition are recorded and speedup/parallelization fractions are calculated.

4. **Visualization**: Python scripts generate plots for execution time and speedup.

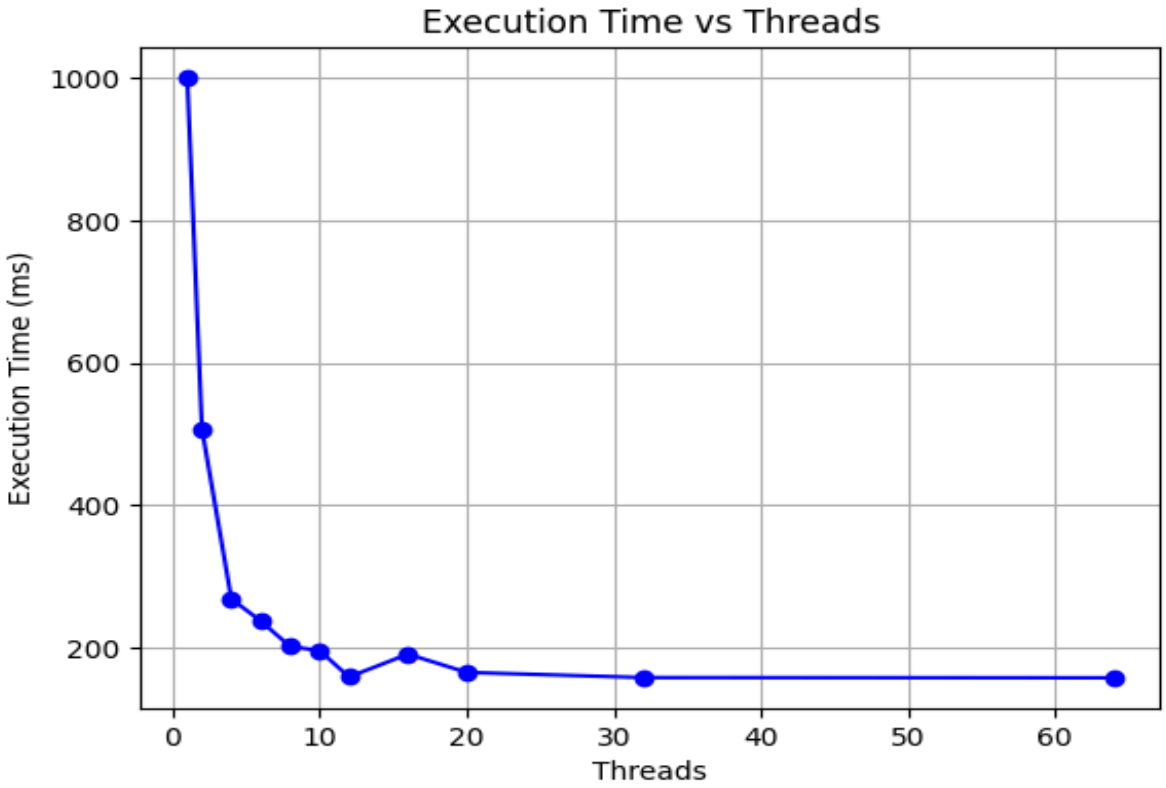
Parallel Code for Matrix Addition:

```
void addMatricesParallel(const vector<vector<double>> &matrix1, const vector<vector<double>> &matrix2, vector<vector<double>> &result)
{
    #pragma omp parallel for
    for (int i = 0; i < SIZE; ++i)
    {
        for (int j = 0; j < SIZE; ++j)
        {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }
}
```

Serial Code for Matrix Addition:

```
void addMatricesSerial(const vector<vector<double>> &matrix1, const vector<vector<double>> &matrix2, vector<vector<double>> &result)
{
    for (int i = 0; i < SIZE; ++i)
    {
        for (int j = 0; j < SIZE; ++j)
        {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }
}
```

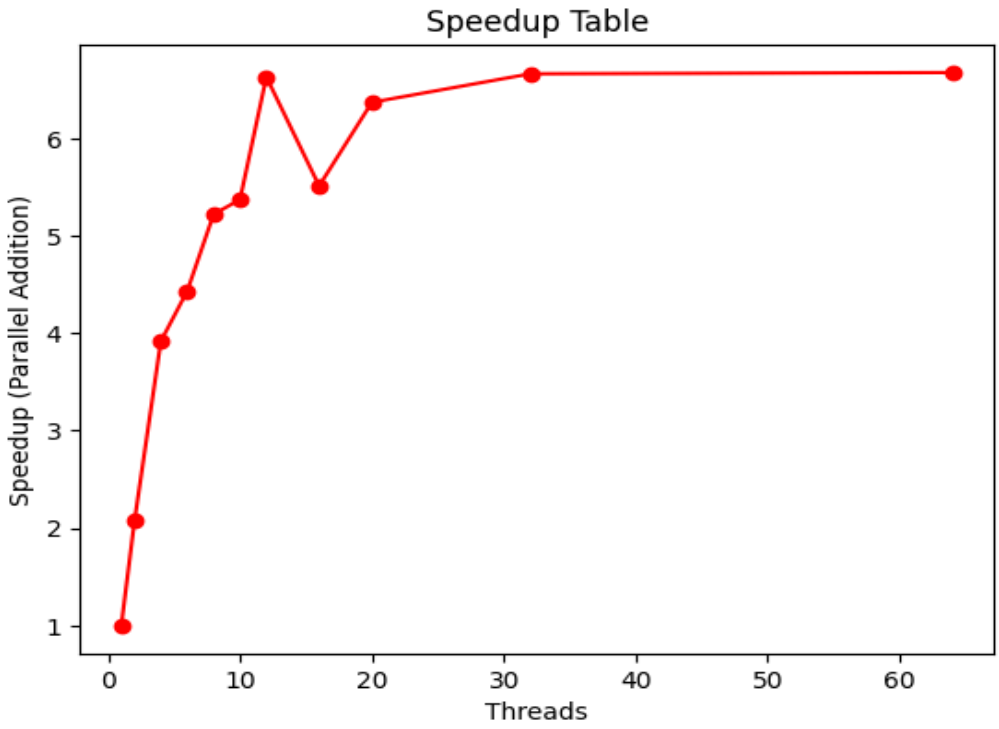
Plot Threads v/s Time :



Observations :

1. When increasing thread count from 2 to 12, execution time drops until it reaches 12 threads.
2. The addition and the thread management overhead would result in an increase in the execution time. However, we see that due to the large size of the problem, the execution time remains similar for threads ranging from 20 to 64.
3. This suggests that the thread context switch is not as dominant compared to the computational cost of matrix addition.

Plot Threads v/s Speedup:



Observations :

1. The sudden spike in the overall speedup from 1 thread to 12 threads shows that for the system on which this was performed, 12 threads is the most optimal in case of matrix addition.
2. The speedup does not appear to have a major descent as the number of threads increases.
3. The parallel construct seems to efficiently parallelize and reduce the execution time up-to 12 threads after which the CPU and thread management overheads are expected to negatively affect the speedup.

Inferences :

1. Since speedup is calculated by Amdahl's law, the speedup is inversely proportional to parallelized execution time.
2. If we look at the speedup and execution time graphs side by side we would notice that the slope of the descent in **speedup** looks similar to the magnitude of slope of ascent in **execution time**.
3. Both slopes seem to have flatlined after a certain number of threads.

Estimated Parallelization Fraction :

```
=== Parallelization Fraction Table ===
   Threads            P. Fraction (Parallel Addition)
-----------------------------------
         1                 0.000000
         2                 1.034820
         4                 0.992952
         6                 0.929279
         8                 0.923927
        10                 0.904207
        12                 0.926319
        16                 0.873093
        20                 0.887338
        32                 0.877276
        64                 0.863653
```

Observations :

1. The parallelization fraction reaches its peak at 8 and 12 threads...
2. The parallelization fraction does drop slightly but not at an alarming rate.
3. Up to 10 threads, the program maintains a parallelization fraction (≥ 0.9).
4. This suggests that parallelization is advantageous in case of matrix addition.

Conclusion:

- The optimal number of threads for this workload appears to be around **10-12**.
- Beyond this point, increasing threads doesn't seem to affect the execution time or the parallelization factor much negatively.
- The consistency in parallel fraction with more threads suggests that parts of the computation have been efficiently parallelized..