HPC Tutorial 3 Report
CS22B2012
K Aditya Sai

The code for Addition and Multiplication of vectors is implemented in C++ with OpenMP.

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

2. **Dataset Generation**: A C++ code generates two files containing 1 million double-precision floating-point values randomly which are stored in input1.txt and input2.txt.

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

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

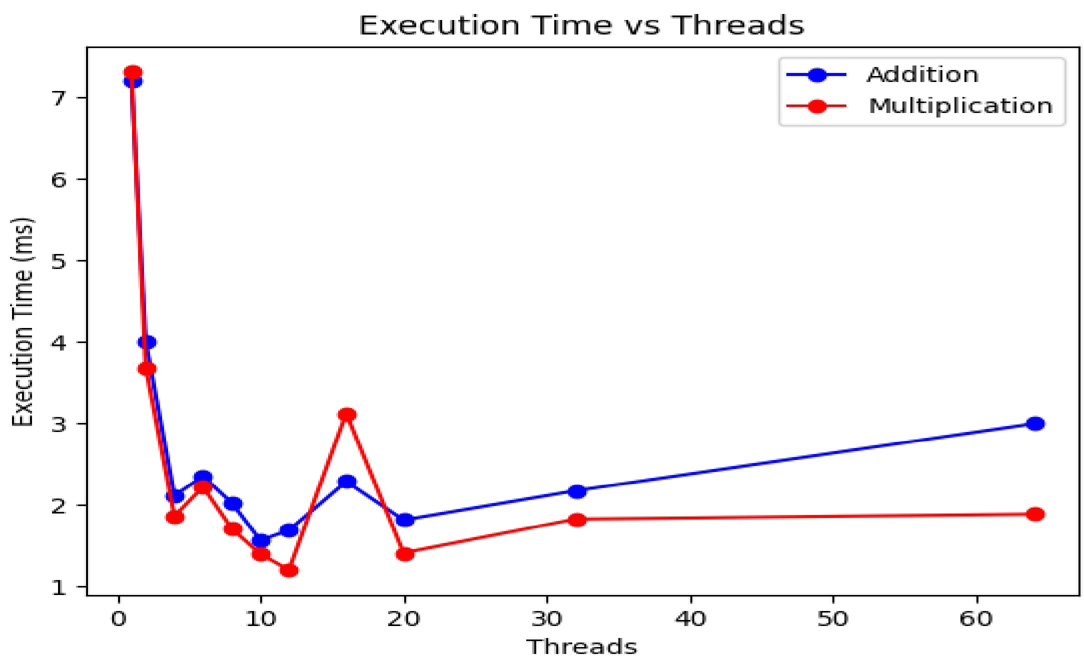
Parallel Code for Addition :

```
void parallelVectorAddition(const vector<double> &A, const vector<double> &B, vector<double> &C, double &sum)
{
    sum = 0.0;
    #pragma omp parallel for
    for (size_t i = 0; i < A.size(); i++)
    {
        C[i] = A[i] + B[i];
        // sum += C[i];
    }
}
```

Parallel Code for Multiplication :

```
void parallelVectorMultiplication(const vector<double> &A, const vector<double> &B, vector<double> &C, double &prod)
{
    prod = 0.0;
    #pragma omp parallel for
    for (size_t i = 0; i < A.size(); i++)
    {
        C[i] = A[i] * B[i];
        // prod += C[i];
    }
}
```

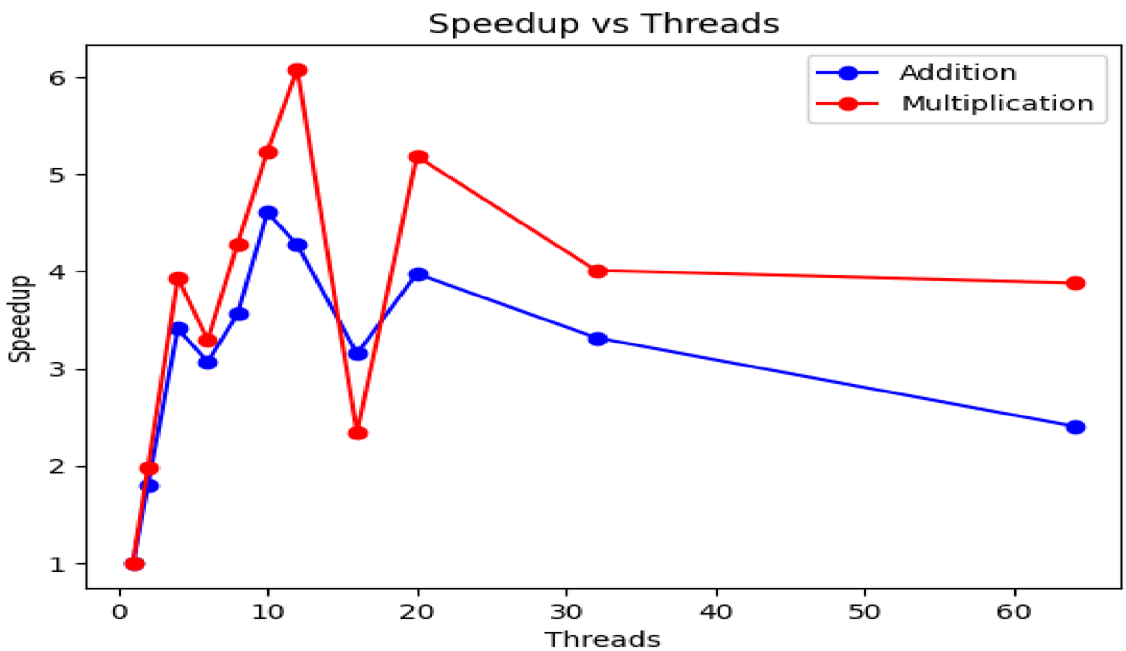
Plot Threads v/s Time :



Observations :

1. When increasing thread count from 2 to 10, execution time drops sharply for both addition (blue) and multiplication (red) this shows that parallelization is working efficiently.
2. After 10 threads, multiplication has comparatively lower execution time suggesting that multiplication is more efficiently parallelized.
3. After 32 threads it is observed that the execution time takes a slight increase, due to the context switch overheads.

Plot Threads v/s Speedup:



Observations :

1. Between **10 and 20 threads**, the speedup shows irregular behavior may be due to thread scheduling overheads, memory bandwidth limitations, or load imbalance.
2. Multiplication has more sharp fluctuations since it's more computationally intensive compared to addition

Inferences :

1. Since speedup is calculated by Amdahl's law, the speedup is inversely proportional to parallelized execution time.
2. Parallelized execution time is negatively affected by increase in number of threads after some number of threads due to the overhead of context switch.
3. This overhead is existent is all cases but becomes dominant as the number of threads keeps increasing.

Estimated Parallelization Fraction :

=== Parallelization Fraction Table ===

Threads	P. Fraction (Addition)	P. Fraction (Multiplication)
2	0.889526	0.992779
4	0.942251	0.994274
6	0.809033	0.835855
8	0.821862	0.875927
10	0.869850	0.898875
12	0.835584	0.911391
16	0.728354	0.611589
20	0.787853	0.849552
32	0.721136	0.774833
64	0.593851	0.754017

Observations :

1. The parallelization fraction reaches its peak at 10 threads for addition and 12 threads for multiplication..
2. Addition parallelization drops more sharply beyond 10 threads likely because it encounters more parallelization bottlenecks sooner.
3. Up to 10 threads, multiplication maintains a higher fraction (≥ 0.9) compared to addition.
4. This suggests multiplication scales better with increasing threads, meaning it has less synchronization or memory bottlenecks.

Conclusion:

- Multiplication has a consistently better parallelization fraction than addition, indicating it scales more efficiently.
- Parallelization efficiency declines significantly beyond 16 threads, making additional threads less beneficial.
- Addition faces stronger parallelization bottlenecks than multiplication, likely due to memory contention.
- Optimizations like dynamic scheduling or memory access improvements could improve performance at high thread counts.