HPC Tutorial 4 Report
CS22B2012
K Aditya Sai

The code for Dot Product 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.

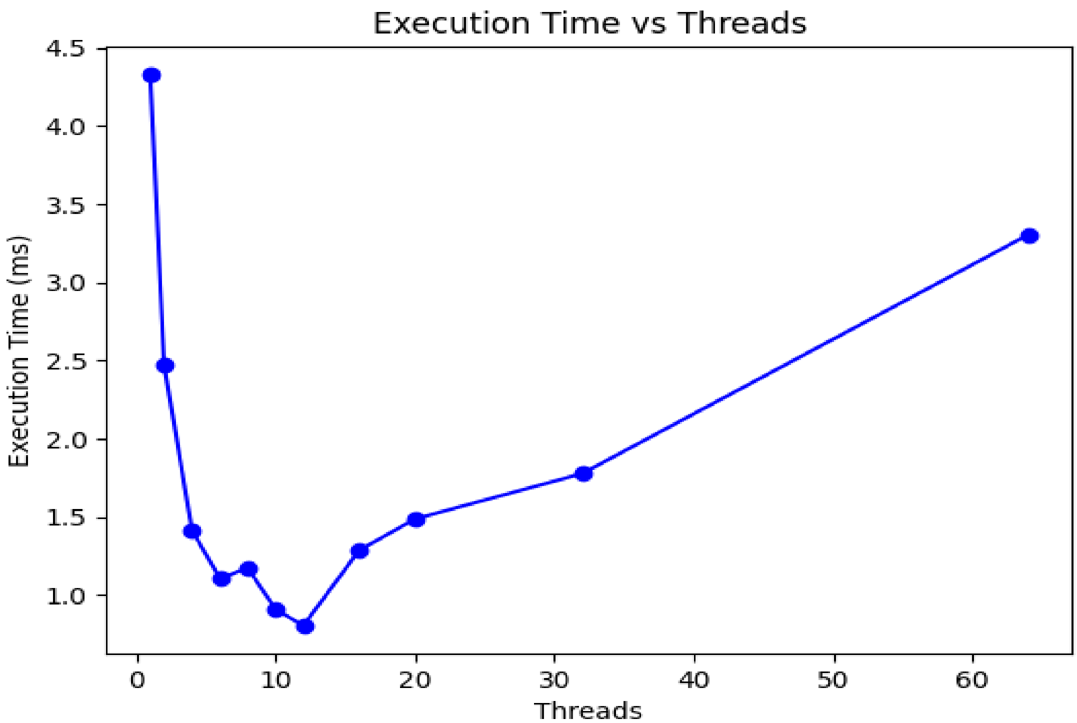
3. **Performance Analysis**: Execution times for the vector multiplication and scalar addition are recorded and speedup/parallelization fractions are calculated.

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

Parallel Code for Dot Product

```cpp
double dot_product(vector<double> &A, vector<double> &B)
{
    double sum = 0.0;
    #pragma omp parallel
    {
        double prod = 0.0;
        #pragma omp for
        for (int i = 0; i < N; i++)
        {
            prod += A[i] * B[i];
        }
        #pragma omp critical
        sum += prod;
    }
    return sum;
}
```

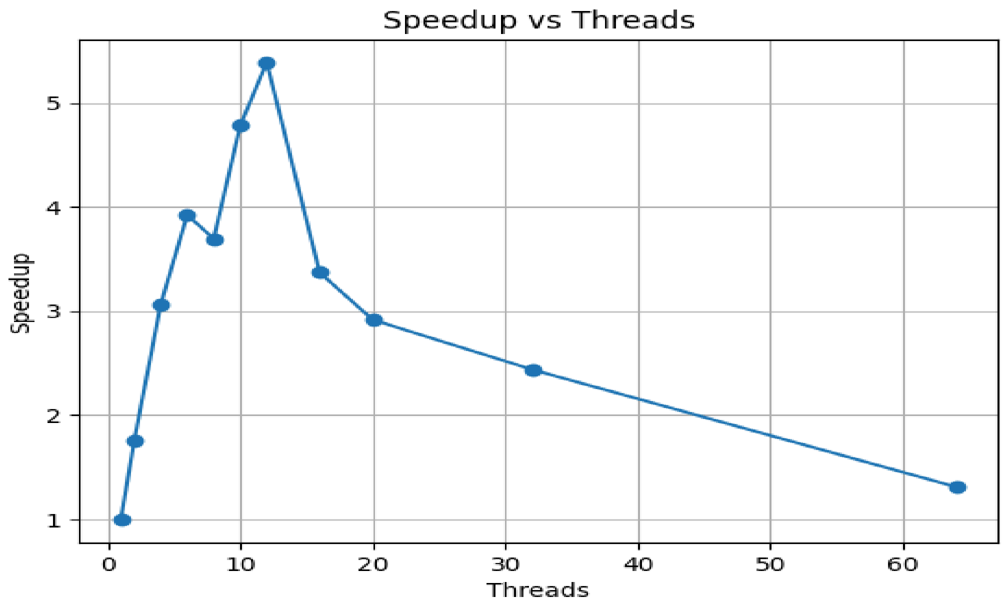
Plot Threads v/s Time :



Observations :

1. When increasing thread count from 2 to 12, execution time drops and there is a small irregularity from 6 cores to 8 cores and then to 10 cores.
2. The combined effect of both multiplication and addition and the thread management overhead results in an increase in the execution time.
3. Compared to the previous tutorial, we see that the increase after the 12 thread mark is much cheaper.
4. So we can conclude that performing multiplication and simultaneous addition is even more expensive.

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 vector dot product.
2. The speedup starts to have a steeper descent compared to both vector addition and multiplication. Since dot product involves vector multiplication and then aggregation.
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 start 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**.

Estimated Parallelization Fraction :

```
=== Parallelization Fraction Table ===
Threads    P. Fraction
----------------------
    1       1.000000
    2       0.858727
    4       0.897801
    6       0.894175
    8       0.833414
   10       0.878704
   12       0.888392
   16       0.750025
   20       0.691603
   32       0.608944
   64       0.240891
```

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:

- The optimal number of threads for this workload appears to be around **10-12**.
- Beyond this point, increasing threads leads to higher overhead, lower efficiency, and even increased execution time.
- The decrease in parallel fraction with more threads suggests that parts of the computation remain sequential or suffer from synchronization bottlenecks.
- The usage of critical causes increase in execution time due to thread synchronization overhead.