HPC OPENMP TUTORIAL 3 REPORT

CS22B2015 – HARSHITH B

The parallel code for reduction and critical sections is provided in the main.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.
2. Two datasets consisting of approximately 15 million double-precision floating-point numbers is generated using Python and stored in the output1.txt and output2.txt files.
3. 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. Parallel Code Segment for Addition

```
printf("Parallel Vector Addition\n");
for(int i=0;i<num_options;i++){
    int T = thread_counts[i];
    omp_set_num_threads(T);
    start = omp_get_wtime();

    #pragma omp parallel for
    for(int j=0;j<count;j++){
        vec.sum_result[j] = vec.arr1[j] + vec.arr2[j];
    }

    end = omp_get_wtime();
    double time_taken = end - start;
    printf("Threads: %d, Time: %lf\n", T, time_taken);
    fprintf(f_add, "%d %lf\n", T, time_taken);
}
```

2. Parallel Code Segment for Critical Section

```c
printf("Parallel Vector Multiplication\n");
for(int i=0;i<num_options;i++){
    int T = thread_counts[i];
    omp_set_num_threads(T);
    start = omp_get_wtime();

    #pragma omp parallel for
    for(int j=0;j<count;j++){
        vec.product_result[j] = vec.arr1[j] * vec.arr2[j];
    }

    end = omp_get_wtime();
    double time_taken = end - start;
    printf("Threads: %d, Time: %lf\n", T, time_taken);
    fprintf(f_mul, "%d %lf\n", T, time_taken);
}
```

3. Terminal Output

```
(venv) harshith@harshithb:~/Projects /SEM 6/HPC/tutorial-3$ gcc -fopenmp -o main main.c
(venv) harshith@harshithb:~/Projects /SEM 6/HPC/tutorial-3$ ./main
Parallel Vector Addition
Threads: 1, Time: 0.122005
Threads: 2, Time: 0.049216
Threads: 4, Time: 0.041992
Threads: 6, Time: 0.039007
Threads: 8, Time: 0.039357
Threads: 10, Time: 0.031059
Threads: 12, Time: 0.043587
Threads: 16, Time: 0.030443
Threads: 20, Time: 0.034529
Threads: 32, Time: 0.027351
Threads: 64, Time: 0.029720
Parallel Vector Multiplication
Threads: 1, Time: 0.121978
Threads: 2, Time: 0.048556
Threads: 4, Time: 0.039551
Threads: 6, Time: 0.040747
Threads: 8, Time: 0.037327
Threads: 10, Time: 0.038504
Threads: 12, Time: 0.048182
Threads: 16, Time: 0.035565
Threads: 20, Time: 0.030276
Threads: 32, Time: 0.027972
Threads: 64, Time: 0.023994
```

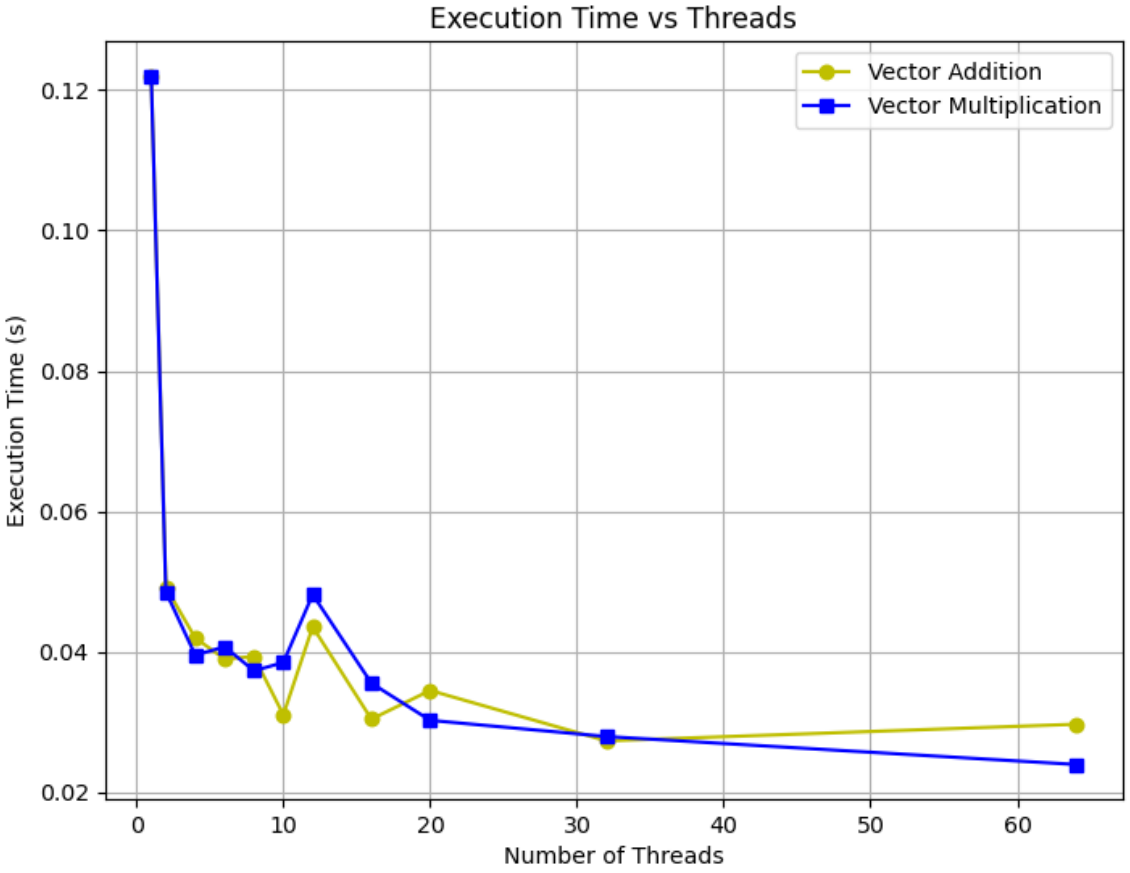
4. Thread vs Time Plots and Observations

Parallel Vector Addition:

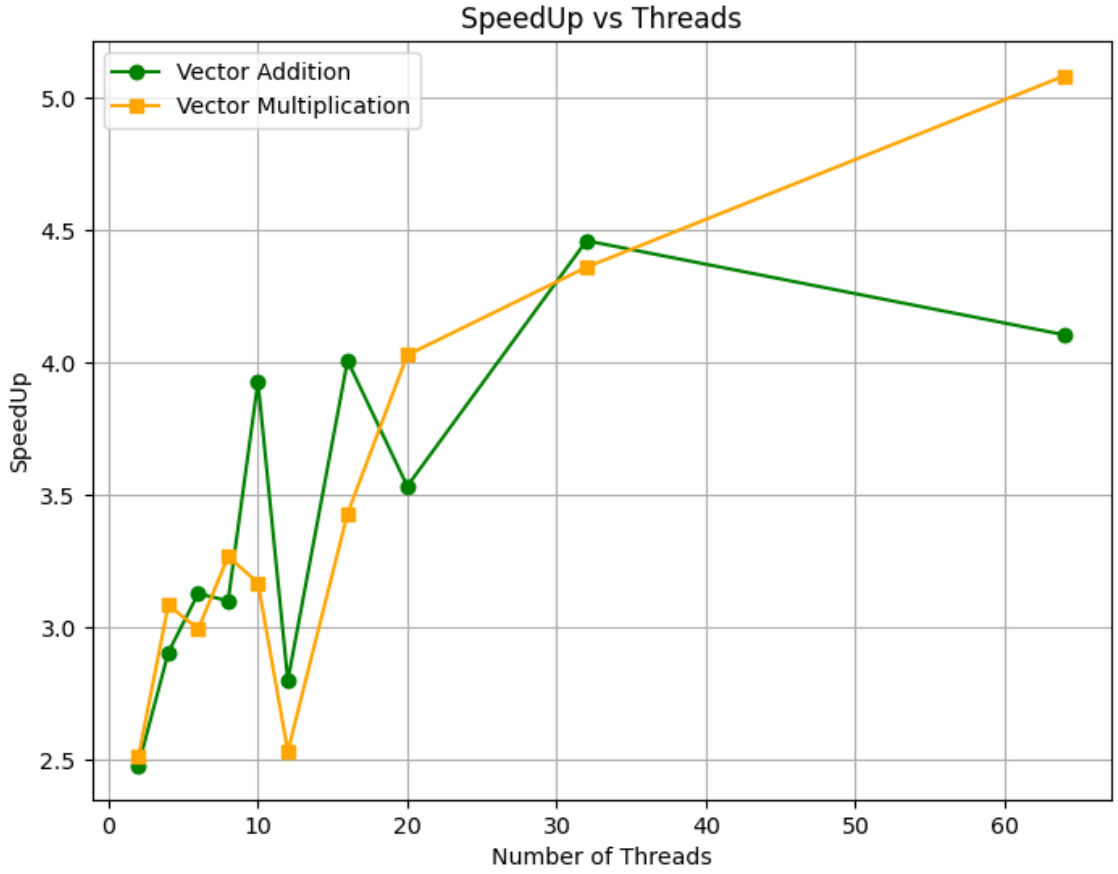
- The execution time for vector addition decreases sharply as the number of threads increases from 1 to around 8 threads, showing effective parallelization in this range.
- The **best performance** appears to occur around **32 threads**, where the execution time is at its lowest.
- Beyond 32 threads, the execution time increases slightly, likely due to overheads such as thread contention and communication costs.
- This suggests that vector addition scales well with increasing threads up to a point, after which the cost of coordination outweighs the gains from additional parallelism.

Parallel Vector Multiplication:

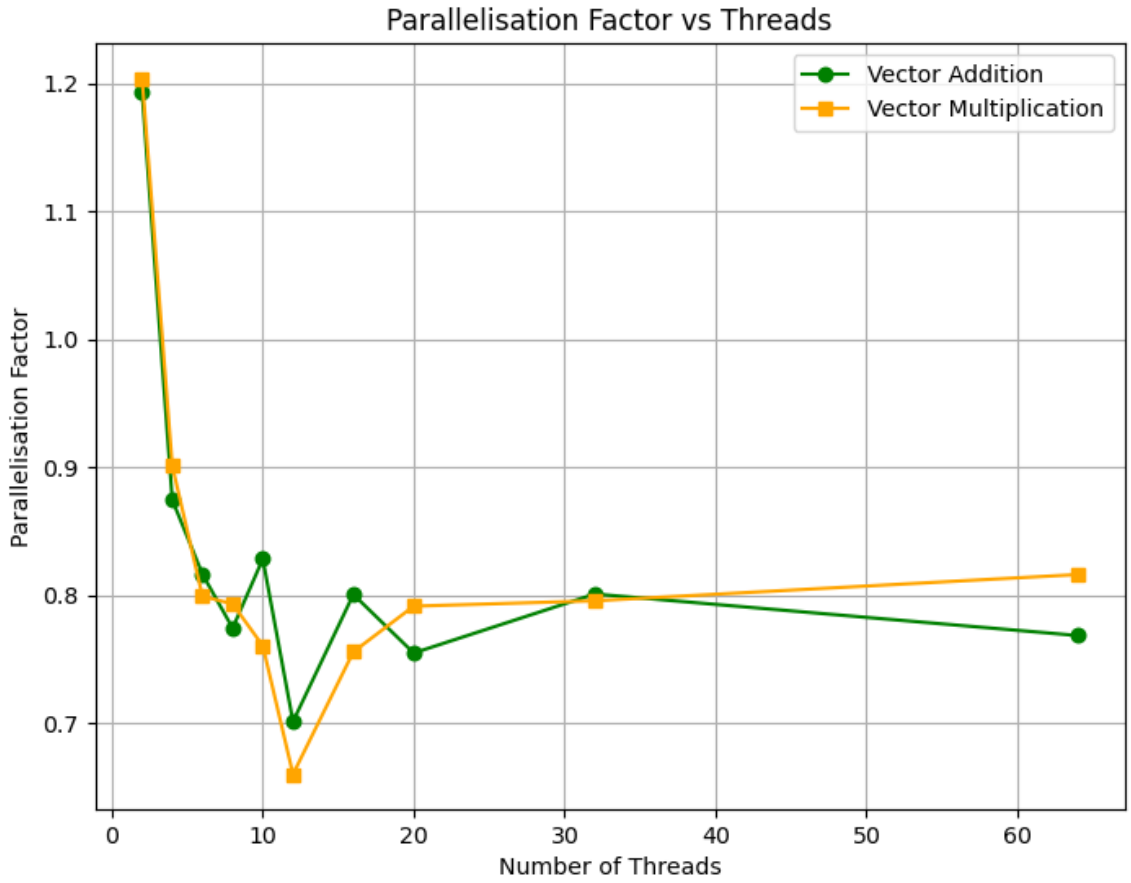
- The execution time for vector multiplication also decreases significantly from 1 thread to around 8 threads, showing initial scalability.
- After around **8 to 16 threads**, the performance flattens, and further increasing the thread count does not result in noticeable improvements.
- Slight performance degradation is observed at higher thread counts, possibly due to the overhead of managing many threads or diminishing returns when the work per thread becomes small.
- This indicates that vector multiplication benefits from parallelization but has a lower thread count threshold for optimal performance compared to vector addition.



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



6. Parallelization Fraction and Inference



- ### *Parallel Vector Addition:*

- The parallelization factor peaks initially but declines as the number of threads increases, stabilizing around **0.8**.
- This indicates diminishing returns for parallelization beyond a certain threshold (approximately **32 threads**).
- Overhead due to thread management likely impacts performance at higher thread counts.
-

- ### Parallel Vector Multiplication:

- The parallelization factor remains close to **1**, indicating highly parallelizable computation.
- The performance is optimal around **4 threads**, beyond which the overhead of managing additional threads begins to outweigh the benefits.