HPC OPENMP TUTORIAL 6 REPORT

CS22B2015 – HARSHITH B

The serial and parallel code for matrix multiplication 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 Multiplication

```c
void multiply_matrices_serial(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] = 0;
            for (int k = 0; k < N; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
            // Print only for every 1000th column
            if (j % 1000 == 0) {
                printf("result[%d][%d]: %lf\n", i, j, result[i][j]);
            }
        }
    }

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

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

    #pragma omp parallel for collapse(2) num_threads(thread_count)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            result[i][j] = 0;
            for (int k = 0; k < N; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
            if (j % 1000 == 0) {
                printf("result[%d][%d] = %f, Threads: %d\n", i, j, result[i][j], thread_count);
            }
        }
    }

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

2. Terminal Output

```
result[9999][9000]: 2517257282330518.000000
Serial Time: 14939.116942 seconds
harshith@harshithb:~/Projects /SEM 6/HPC/tutorial-6$
```

```
result[6999][9000] = 2502271318989670.500000, Threads: 10
Threads: 10, Parallel Time: 2283.960016 seconds
result[0][0] = 2480267058893090.000000, Threads: 12
```

```
result[8749][9000] = 2520097921720330.000000, Threads: 8
Threads: 8, Parallel Time: 2695.886180 seconds
```

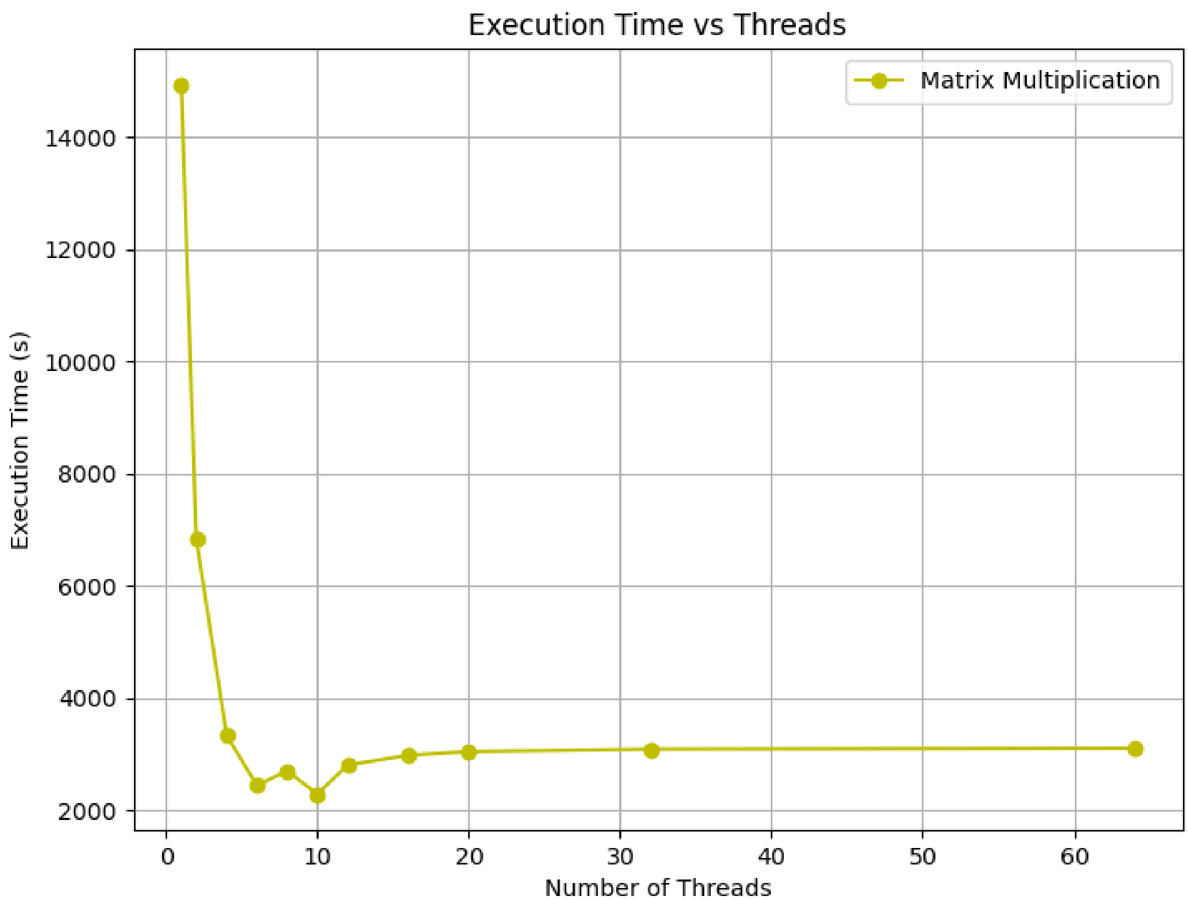
```
result[8333][3000] = 2463087898288888.500000, Threre
Threads: 6, Parallel Time: 2439.471899 seconds
```

```
result[5100][6000] = 2491430858378203.500000, Threads: 12
Threads: 12, Parallel Time: 2802.993806 seconds
result[4375][0] = 2498118812164277.500000, Threads: 16
```

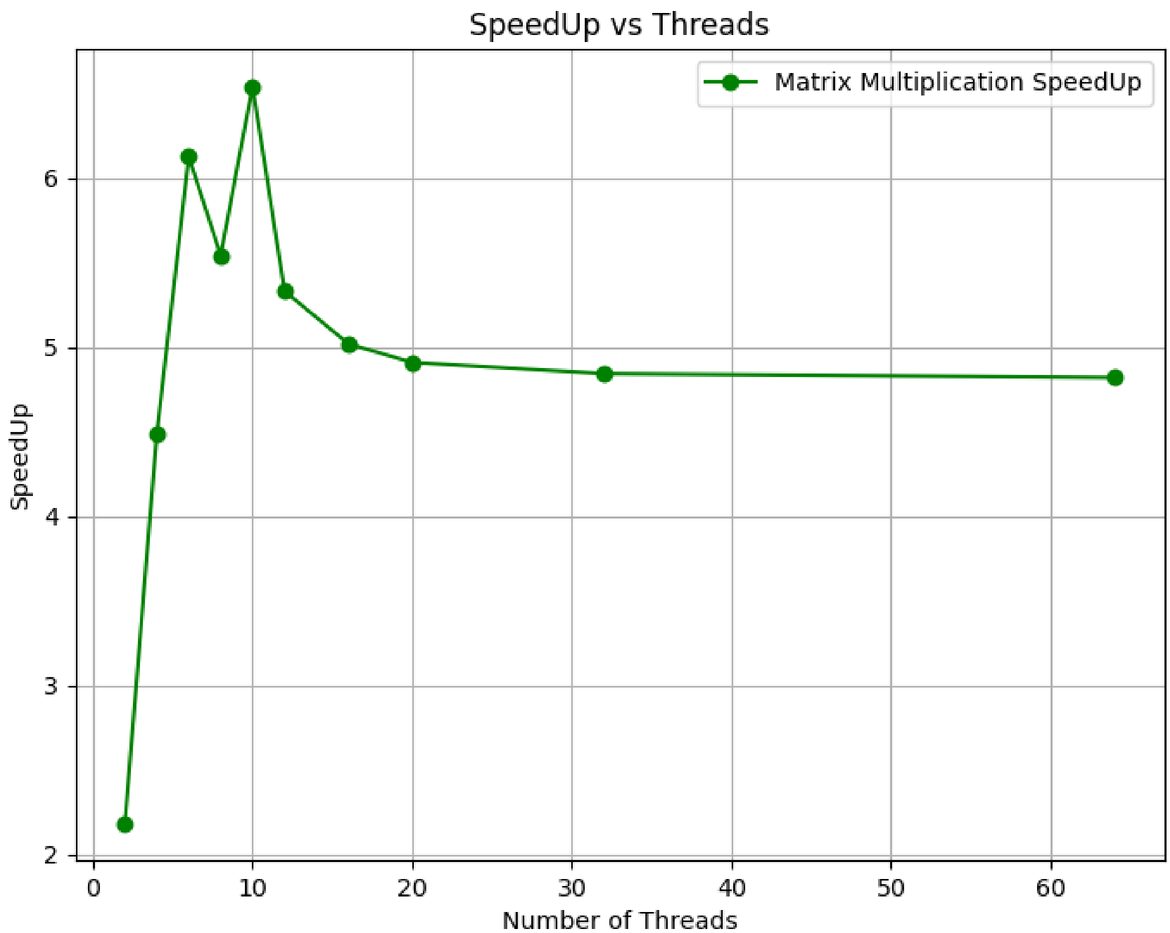
3. Thread vs Time Plots and Observations

Observations :

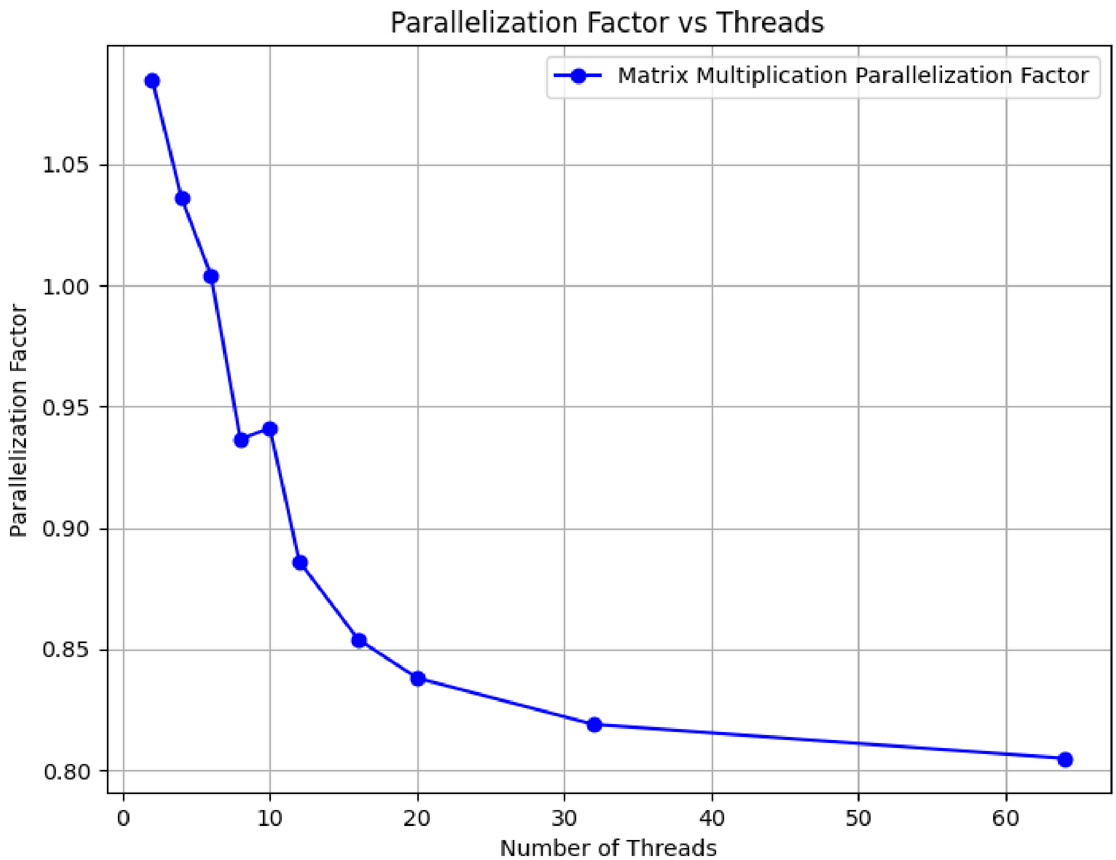
- **Initial Improvement:** There is a significant reduction in execution time as the number of threads increases from 1 to around 10 threads. This indicates effective parallelization at lower thread counts.
- **Optimal Thread Count:** The best performance is observed at a 10 threads.



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



5. Parallelization Fraction and Inference



-

Parallelisation Factor estimation : 0.93 at 10 threads