T06: OpenMP - Write a parallel code to perform two NxN Matrix Multiplication - Each element of the matrix is double precision number.

CS24M1005 – SINDHIYA R

Write a parallel code to perform two NxN Matrix Multiplication - Each element of the matrix is double precision number. Consider N values sufficiently larger number at least 10000.

Assumption: 3000 matrix size is taken as input because 10000 input matrix size took 15000 seconds for execution which is roughly 2.5hrs. Attached image for your reference.

```
9992 9993 9994 9995 9996 9997 9998 9999 Serial Execution Time: 15332.328169 seconds
sindhiya@MSI:/mnt/c/SEM_02/HPC/t06$
```

1) Serial Code - 5 Marks

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
using namespace std;
#define N 3000 // Matrix size

// Function to allocate a matrix
double** allocate_matrix(int n) {
    double** matrix = (double**)malloc(n * sizeof(double*));
    for (int i = 0; i < n; i++) {
        matrix[i] = (double*)malloc(n * sizeof(double));
    }
    return matrix;
}

// Function to initialize a matrix with random values
void initialize_matrix(double** matrix, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i][j] = (double)rand() / RAND_MAX;
        }
    }
}

// Serial Matrix Multiplication
void serial_matrix_mult(double** A, double** B, double** C, int n) {
    for (int i = 0; i < n; i++) {
```

```
        // cout<<i<< " ";
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    double** A = allocate_matrix(N);
    double** B = allocate_matrix(N);
    double** C = allocate_matrix(N);

    initialize_matrix(A, N);
    initialize_matrix(B, N);

    clock_t start = clock();
    serial_matrix_mult(A, B, C, N);
    clock_t end = clock();

    printf("Serial Execution Time: %f seconds\n", (double)(end - start) / CLOCKS_PER_SEC);

    // Free memory
    for (int i = 0; i < N; i++) {
        free(A[i]); free(B[i]); free(C[i]);
    }
    free(A); free(B); free(C);

    return 0;
}
```

Output:

```
sindhiya@MSI:/mnt/c/SEM_02/HPC/t06$ g++ mul.cpp -o mul -fopenmp
sindhiya@MSI:/mnt/c/SEM_02/HPC/t06$ ./mul
Serial Execution Time: 254.150323 seconds
```

2) Parallel Code - 5 Marks

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 3000 // Matrix size
```

```
// Function to allocate a matrix
double** allocate_matrix(int n) {
    double** matrix = (double**)malloc(n * sizeof(double*));
    for (int i = 0; i < n; i++) {
        matrix[i] = (double*)malloc(n * sizeof(double));
    }
    return matrix;
}

// Function to initialize a matrix with random values
void initialize_matrix(double** matrix, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i][j] = (double)rand() / RAND_MAX;
        }
    }
}

// Parallel Matrix Multiplication using OpenMP
void parallel_matrix_mult(double** A, double** B, double** C, int n) {
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    double** A = allocate_matrix(N);
    double** B = allocate_matrix(N);
    double** C = allocate_matrix(N);

    initialize_matrix(A, N);
    initialize_matrix(B, N);

    int threads[] = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64};
    int num_tests = sizeof(threads) / sizeof(threads[0]);

    for (int t = 0; t < num_tests; t++) {
        omp_set_num_threads(threads[t]);
```

```
        double start = omp_get_wtime();
        parallel_matrix_mult(A, B, C, N);
        double end = omp_get_wtime();
        printf("Parallel Execution with %d threads: %f seconds\n", threads[t], end - start);
    }

    // Free memory
    for (int i = 0; i < N; i++) {
        free(A[i]); free(B[i]); free(C[i]);
    }
    free(A); free(B); free(C);
    return 0;
}
```

Output:

```
sindhiya@MSI:/mnt/c/SEM_02/HPC/t06$ g++ mul_p.cpp -o mul_p -fopenmp
sindhiya@MSI:/mnt/c/SEM_02/HPC/t06$ ./mul_p
Parallel Execution with 1 threads: 261.819542 seconds
Parallel Execution with 2 threads: 124.467051 seconds
Parallel Execution with 4 threads: 63.176933 seconds
Parallel Execution with 6 threads: 44.522700 seconds
Parallel Execution with 8 threads: 38.269970 seconds
Parallel Execution with 10 threads: 36.955305 seconds
Parallel Execution with 12 threads: 32.439435 seconds
Parallel Execution with 16 threads: 31.529269 seconds
Parallel Execution with 20 threads: 32.671955 seconds
Parallel Execution with 32 threads: 31.242556 seconds
Parallel Execution with 64 threads: 31.632853 seconds
```

3) Report - Thread vs Time - (run the parallel code with 1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64 Processors) - 10 Marks

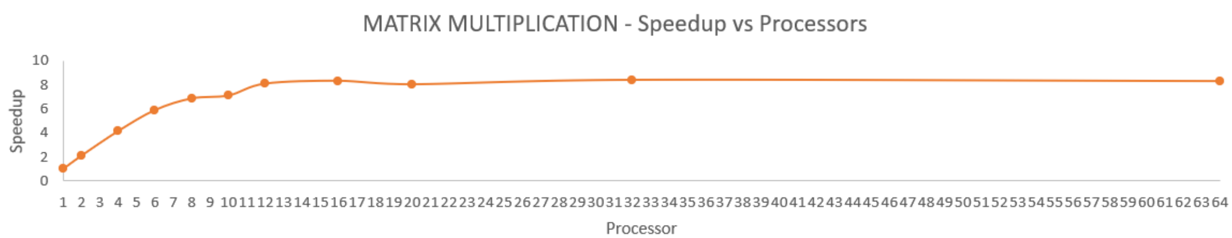
Threads	Execution Time
1	261.819542
2	124.467051
4	63.176933
6	44.5227
8	38.26997
10	36.955305
12	32.439435
16	31.529269
20	32.671955
32	31.242556
64	31.632853

Observation:

- Beyond **12 threads**, the **performance gain is minimal**, indicating a **saturation point**.
- **CPU** and **memory bandwidth bottlenecks limit** further improvement.
- After 12 threads, **increased contention for cache** and **RAM bandwidth** prevents further speedup.
- **Synchronization overhead** and **thread management** costs increase at 20+ threads, leading to no significant speedup.
- Threads beyond available **physical cores (hyperthreading)** do not contribute much to performance.

4) Plot Speedup vs Processors - 5 Marks

Threads	Execution Time	Speedup
1	261.819542	1
2	124.467051	2.103525
4	63.176933	4.144227
6	44.5227	5.880585
8	38.26997	6.841384
10	36.955305	7.084762
12	32.439435	8.071027
16	31.529269	8.304016
20	32.671955	8.013587
32	31.242556	8.380222
64	31.632853	8.276824



5) Inference - 5 Marks

1. Near-Linear Speedup Up to 12 Threads

- Speedup improves significantly up to 12 threads (8.07× speedup).
- This indicates that **matrix multiplication is highly parallelizable**, benefiting from multi-threading.

2. Performance Saturation Beyond 12 Threads

- The improvement flattens after 12-16 threads, suggesting a hardware bottleneck.
- This is caused by memory **bandwidth limitations** and **cache conflicts**.

3. Memory Bandwidth Becomes the Limiting Factor

- Beyond 12 threads, the processor struggles to fetch data from memory efficiently.
- Large matrices require high memory throughput, leading to **contention** among threads.

4. Synchronization and Overhead Reduce Efficiency

- At 20+ threads, execution time fluctuates, showing diminishing returns.
- **Thread creation, scheduling, and data synchronization** introduce significant overhead.

5. CPU Hyperthreading Provides Minimal Gains

- Increasing threads beyond available physical cores (16 in modern i7 CPUs) does not improve speedup.
- Hyperthreading only helps if computations involve **frequent memory stalls**.

6. Matrix Size Plays a Crucial Role in Parallel Performance

- For very large matrices ($N \geq 10{,}000$), multi-threading is highly effective.
- However, for smaller matrices, overhead dominates, making **parallelization inefficient**.

7. Tiling and Cache Optimization Can Improve Performance

- Using cache-aware optimizations like tiling (**block matrix multiplication**) reduces cache misses.
- This can improve performance even when increasing threads beyond 12-16.

8. GPU Acceleration Would Be More Effective

- GPUs are optimized for massively parallel operations, unlike CPUs, which suffer from thread contention.

9. Thread Scheduling and Load Balancing Affect Performance

- Uneven work distribution among threads can cause some to remain idle while others work harder.
- Dynamic scheduling may improve performance in some cases.

10. Parallel Matrix Multiplication is Memory-Bound, Not Just Compute-Bound

- While matrix multiplication involves intense computation ($O(N^3)$), memory access is a major bottleneck.