T05: OpenMP - Write a parallel code to perform two NxN Matrix Addition - Each element of the matrix is double precision number.

CS24M1005 – SINDHIYA R

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

1) Serial Code - 5 Marks

```
#include <iostream>
#include <cstdlib>
#include <chrono>

using namespace std;

#define N 10000  // Matrix size

// Function to allocate memory for an NxN matrix
double** allocate_matrix(int size) {
   double** matrix = new double*[size];
   for (int i = 0; i < size; i++) {
      matrix[i] = new double[size];
   }
   return matrix;
}

// Function to initialize matrix with random values
void initialize_matrix(double** matrix, int size) {
   for (int i = 0; i < size; i++) {
      for (int j = 0; j < size; j++) {
         matrix[i][j] = (double)(rand() % 1000) / 100.0;  // Values in range [0,10]
      }
   }
}

// Function to free matrix memory
void free_matrix(double** matrix, int size) {
   for (int i = 0; i < size; i++) {
      delete[] matrix[i];
   }
   delete[] matrix;
}

// Serial matrix addition
void serial_matrix_addition(double** A, double** B, double** C, int size) {
```

```
    auto start = chrono::high_resolution_clock::now();

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }

    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> duration = end - start;
    cout << "Serial Execution Time: " << duration.count() << " seconds" << endl;
}

// Main function
int main() {
    // Allocate matrices
    double** A = allocate_matrix(N);
    double** B = allocate_matrix(N);
    double** C = allocate_matrix(N);

    // Initialize matrices
    initialize_matrix(A, N);
    initialize_matrix(B, N);

    // Run serial matrix addition
    serial_matrix_addition(A, B, C, N);

    // Free allocated memory
    free_matrix(A, N);
    free_matrix(B, N);
    free_matrix(C, N);
    return 0;
}
```

Output:

```
sindhiya@MSI:/mnt/c/SEM_02/HPC/t05$ g++ add_s.cpp -o add_s -fopenmp
sindhiya@MSI:/mnt/c/SEM_02/HPC/t05$ ./add_s
Serial Execution Time: 0.439476 seconds
```

2) Parallel Code - 5 Marks

```
#include <iostream>
#include <cstdlib>
#include <omp.h>

using namespace std;

#define N 10000  // Matrix size

// Function to allocate memory for an NxN matrix
```

```
double** allocate_matrix(int size) {
   double** matrix = new double*[size];
   for (int i = 0; i < size; i++) {
      matrix[i] = new double[size];
   }
   return matrix;
}

// Function to initialize matrix with random values
void initialize_matrix(double** matrix, int size) {
   for (int i = 0; i < size; i++) {
      for (int j = 0; j < size; j++) {
         matrix[i][j] = (double)(rand() % 1000) / 100.0;  // Values in range [0,10]
      }
   }
}

// Function to free matrix memory
void free_matrix(double** matrix, int size) {
   for (int i = 0; i < size; i++) {
      delete[] matrix[i];
   }
   delete[] matrix;
}

// Parallel matrix addition using OpenMP
void parallel_matrix_addition(double** A, double** B, double** C, int size, int num_threads) {
   omp_set_num_threads(num_threads);  // Set number of threads
   double start = omp_get_wtime();

   #pragma omp parallel for collapse(2)  // Parallelize both loops
   for (int i = 0; i < size; i++) {
      for (int j = 0; j < size; j++) {
         C[i][j] = A[i][j] + B[i][j];
      }
   }

   double end = omp_get_wtime();
   cout << "Parallel Execution Time with " << num_threads << " threads: " << (end - start) << " seconds"
<< endl;
}

// Main function
int main() {
   // Allocate matrices
   double** A = allocate_matrix(N);
   double** B = allocate_matrix(N);
   double** C = allocate_matrix(N);
```

```
    // Initialize matrices
    initialize_matrix(A, N);
    initialize_matrix(B, N);

    // Run parallel matrix addition with different thread counts
    int threads[] = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64};
    for (int t : threads) {
        parallel_matrix_addition(A, B, C, N, t);
    }

    // Free allocated memory
    free_matrix(A, N);
    free_matrix(B, N);
    free_matrix(C, N);

    return 0;
}
```

Output:

```
sindhiya@MSI:/mnt/c/SEM_02/HPC/t05$ g++ add_p.cpp -o add_p -fopenmp
sindhiya@MSI:/mnt/c/SEM_02/HPC/t05$ ./add_p
Parallel Execution Time with 1 threads: 0.442588 seconds
Parallel Execution Time with 2 threads: 0.114396 seconds
Parallel Execution Time with 4 threads: 0.0741591 seconds
Parallel Execution Time with 6 threads: 0.0692982 seconds
Parallel Execution Time with 8 threads: 0.0701256 seconds
Parallel Execution Time with 10 threads: 0.0661891 seconds
Parallel Execution Time with 12 threads: 0.065642 seconds
Parallel Execution Time with 16 threads: 0.0707848 seconds
Parallel Execution Time with 20 threads: 0.0672343 seconds
Parallel Execution Time with 32 threads: 0.0668305 seconds
Parallel Execution Time with 64 threads: 0.067135 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

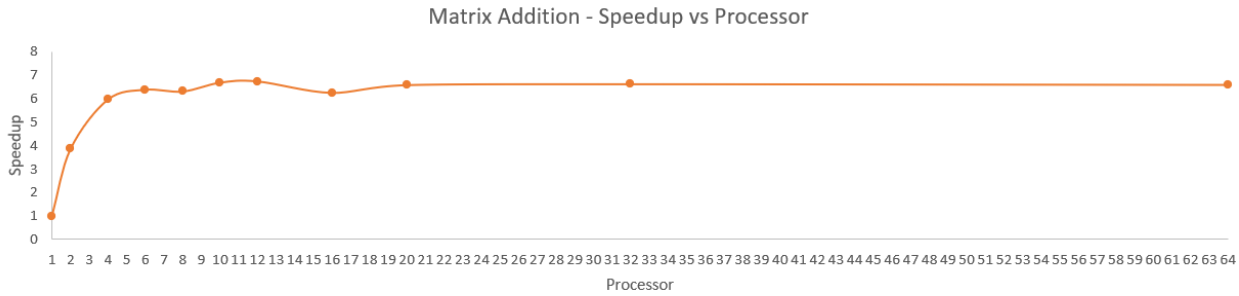
Threads	Execution Time
1	0.442588
2	0.114396
4	0.0741591
6	0.0692982
8	0.0701256
10	0.0661891
12	0.065642
16	0.0707848
20	0.0672343
32	0.0668305
64	0.067135

Observation:

1. **Significant Speedup with Parallelization**
    - The execution time drops drastically from 0.442588s (1 thread) to 0.114396s (2 threads), showing a ~3.87x speedup. This indicates that the parallel implementation effectively utilizes multiple cores.
2. **Diminishing Returns Beyond 4 Threads**
    - The speedup is substantial up to 4 threads (0.0741591s), but after that, the improvements slow down. This suggests that memory bandwidth and synchronization overhead become limiting factors.
3. **Performance Saturation Around 8–12 Threads**
    - The execution time remains almost constant from 8 to 12 threads (around 0.065-0.070s). This indicates that adding more threads does not significantly improve performance, likely due to memory access bottlenecks.
4. **No Further Speedup Beyond 16 Threads**
    - Increasing threads to 16, 20, 32, and 64 does not yield further performance gains. Instead, execution time fluctuates slightly around 0.066s-0.070s, possibly due to thread management overhead and NUMA (Non-Uniform Memory Access) effects.
5. **Optimal Thread Count Lies Between 6 and 12**
    - The lowest execution time (best performance) is around 10–12 threads, meaning the system achieves optimal resource utilization in this range. Adding more threads beyond this point results in wasted computational resources without noticeable gains.

4) Plot Speedup vs Processors - 5 Marks

Threads	Execution Time	Speedup
1	0.442588	1
2	0.114396	3.868912
4	0.0741591	5.968088
6	0.0692982	6.386717
8	0.0701256	6.311361
10	0.0661891	6.68672
12	0.065642	6.742451
16	0.0707848	6.252585
20	0.0672343	6.582771
32	0.0668305	6.622545
64	0.067135	6.592508



5) Inference - 5 Marks

1. Strong Initial Speedup with Parallelization

- The speedup jumps from 1x (1 thread) to 3.87x (2 threads) and then to 5.97x (4 threads). This shows that parallel execution significantly reduces execution time initially.

2. Diminishing Returns Beyond 4 Threads

- The speedup gain between 4 and 6 threads is only ~0.42x, indicating that increasing threads beyond this point provides smaller performance improvements**.**

3. Near-Maximum Speedup at 12 Threads

- The highest observed speedup is 6.74x at 12 threads, suggesting that this is the optimal thread count for maximizing performance in this system.

4. Performance Saturation Beyond 12 Threads

- Beyond 12 threads, the speedup starts decreasing slightly (e.g., 16 threads → 6.25x, 32 threads → 6.62x), indicating that increasing threads further does not provide proportional benefits.

5. Memory Bandwidth Becomes a Bottleneck

- The slowdown beyond 16 threads suggests that the bottleneck is likely due to memory bandwidth limitations rather than CPU processing power.

6. Minimal Difference Between 10, 12, and 16 Threads

- The execution time difference between 10, 12, and 16 threads is minimal (~0.065s-0.070s), meaning additional threads beyond 10–12 do not significantly reduce runtime.

7. No Further Gains Beyond 20 Threads

- Speedup values at 20, 32, and 64 threads remain around ~6.6x, confirming that the workload does not scale well beyond this point, likely due to thread scheduling overhead.

8. Slightly Worse Performance at 16+ Threads

- The speedup at 16 threads (6.25x) is lower than at 12 threads (6.74x), indicating that too many threads might be causing synchronization overhead, slowing down computation.

9. System Resources May Be Overloaded at 64 Threads

- The speedup at 64 threads (6.59x) is lower than at 12 threads (6.74x), suggesting that using more threads than available cores can cause thread contention and inefficiencies.

10. Theoretical Linear Speedup is Not Achieved

- Ideal speedup for 64 threads should be ~64x, but the actual achieved speedup is only 6.59x. This is because matrix addition is memory-bound rather than compute-bound, and excessive threads increase overhead instead of boosting performance.