Name : Saurabh Nitnaware
Experiment No : 6
Write a parallel program to multiply two matrices using openMP library and compare the execution time with its serial version. Also change the number of threads using omp_set_num_threads() function and analyze how thread count affects the execution time.
Date of Performance : 14/03/24
Date of Submission : 19/04/24

Aim: Write a parallel program to multiply two matrices using openMP library and compare the execution time with its serial version. Also change the number of threads using omp_set_num_threads() function and analyse how thread count affects the execution time.

Objective: To understand the omp_set_num_threads() function and analyse how thread count affects the execution time.

Theory:

Algorithm for a parallel program to multiply two matrices using OpenMP library:

1. *Initialize the matrices A, B, and C.*
2. *Set the number of threads using omp_set_num_threads() function.*
3. *Set up a parallel region using the OpenMP library.*
4. *Within the parallel region, each thread should calculate a subset of the total number of elements in matrix C.*
5. *For each element (i, j) in matrix C, calculate its value by summing the products of the corresponding elements in matrices A and B.*
6. *Use an OpenMP reduction to combine the results from each thread.*
7. *Output the resulting matrix C.*
8. *Calculate the execution time for the parallel version.*
9. *Implement the serial version of the algorithm and calculate its execution time.*
10. *Compare the execution times of the parallel and serial versions to determine the speedup achieved by parallelization.*
11. *Change the number of threads using the omp_set_num_threads() function and analyze how the thread count affects the execution time.*

Note that the execution time may be affected by various factors, including the number of threads used, the size of the matrices, and the hardware on which the program is running. Experimentation and benchmarking may be necessary to determine the optimal thread count and other parameters for a given problem.

In general, increasing the number of threads may improve performance up to a certain point, after which the overhead of thread creation and synchronization may begin to outweigh the benefits of parallelism. Additionally, larger matrices may require more threads to achieve a significant speedup, while smaller matrices may not benefit from parallelization at all.

It is important to note that OpenMP does not guarantee a deterministic order of execution for threads, which means that the output of a parallel program may differ from the output of the corresponding serial program even when given the same input. To ensure correct results, it may be necessary to use synchronization primitives such as critical sections or atomic operations.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <vector>
#include <chrono>
#include <iostream>
using namespace std;
const int N = 1000;
int main(int argc, char** argv) {
    vector<vector<int>> A(N, vector<int>(N));
    vector<vector<int>> B(N, vector<int>(N));
    vector<vector<int>> C(N, vector<int>(N));
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = rand() % 100;
            B[i][j] = rand() % 100;
        }
    }

    auto start_serial = chrono::high_resolution_clock::now();
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            int sum = 0;
```

```
            for (int k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
    auto end_serial = chrono::high_resolution_clock::now();
        auto duration_serial = chrono::duration_cast<chrono::milliseconds>(end_serial - start_serial);
    auto start_parallel = chrono::high_resolution_clock::now();
#pragma omp parallel for
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            int sum = 0;
            for (int k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
    auto end_parallel = chrono::high_resolution_clock::now();
        auto duration_parallel = chrono::duration_cast<chrono::milliseconds>(end_parallel - start_parallel);
     cout << "Time taken for serial matrix multiplication: " << duration_serial.count() << " milliseconds" << endl;
```

```
    cout << "Time taken for parallel matrix multiplication: " << duration_parallel.count() << "
milliseconds" << endl;
```

Output:

Time taken for serial matrix multiplication: 33251 milliseconds

Time taken for parallel matrix multiplication: 32909 milliseconds

Conclusion:In summary, OpenMP enhances matrix multiplication by distributing work across threads, especially beneficial for large matrices and intensive tasks. It optimizes multi-core processor use, potentially reducing execution time. However, achieving peak performance demands attention to workload balance, thread overhead, and memory access. Experimentation and tuning allow developers to exploit OpenMP's potential, accelerating matrix operations and enhancing application performance.