**Class:** Final Year (Computer Science and Engineering)

**Year:** 2024-25 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 6**

**Exam Seat No:**

**Title of practical: Implementation of OpenMP programs.**

Implement following Programs using OpenMP with C:

1. Implementation of Matrix-Matrix Multiplication.
2. Implementation of Matrix-vector Multiplication.

**Problem Statement 1:**

**Screenshots:**

**#include** <omp.h>

**#include** <stdio.h>

**#include** <stdlib.h>

**#define** N 500 *// Smaller size for visibility*

**void** initialize\_matrix(**int** **\*\****matrix*)

{

**for** (**int** i **=** 0; i **<** N; i**++**)

    {

**for** (**int** j **=** 0; j **<** N; j**++**)

        {

*matrix*[i][j] **=** rand() **%** 10;

        }

    }

}

**void** matrix\_multiply(**int** **\*\****a*, **int** **\*\****b*, **int** **\*\****c*)

{

**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];

            }

        }

    }

}

**void** matrix\_multiply\_parallel(**int** **\*\****a*, **int** **\*\****b*, **int** **\*\****c*)

{

**#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];

            }

        }

    }

}

**void** print\_matrix(**int** **\*\****matrix*)

{

**for** (**int** i **=** 0; i **<** 5; i**++**)

    { *// Print only first 5 rows for brevity*

**for** (**int** j **=** 0; j **<** 5; j**++**)

        {

            printf("%d ", *matrix*[i][j]);

        }

        printf("\n");

    }

}

**int** main()

{

**int** **\*\***a **=** malloc(N **\*** **sizeof**(**int** **\***));

**int** **\*\***b **=** malloc(N **\*** **sizeof**(**int** **\***));

**int** **\*\***c **=** malloc(N **\*** **sizeof**(**int** **\***));

**for** (**int** i **=** 0; i **<** N; i**++**)

    {

        a[i] **=** malloc(N **\*** **sizeof**(**int**));

        b[i] **=** malloc(N **\*** **sizeof**(**int**));

        c[i] **=** malloc(N **\*** **sizeof**(**int**));

    }

    initialize\_matrix(a);

    initialize\_matrix(b);

*// Sequential*

**double** start **=** omp\_get\_wtime();

    matrix\_multiply(a, b, c);

**double** end **=** omp\_get\_wtime();

**double** seq\_time **=** end **-** start;

    printf("Sequential Time: %f seconds\n", seq\_time);

    printf("Result Matrix (Sequential):\n");

    print\_matrix(c);

*// Parallel*

    start **=** omp\_get\_wtime();

    matrix\_multiply\_parallel(a, b, c);

    end **=** omp\_get\_wtime();

**double** par\_time **=** end **-** start;

    printf("Parallel Time: %f seconds\n", par\_time);

    printf("Result Matrix (Parallel):\n");

    print\_matrix(c);

*// Calculate Speedup and Efficiency*

**double** speedup **=** seq\_time **/** par\_time;

**int** num\_threads **=** omp\_get\_max\_threads();

**double** efficiency **=** speedup **/** num\_threads;

    printf("Speedup: %f\n", speedup);

    printf("Efficiency: %f\n", efficiency);

**for** (**int** i **=** 0; i **<** N; i**++**)

    {

        free(a[i]);

        free(b[i]);

        free(c[i]);

    }

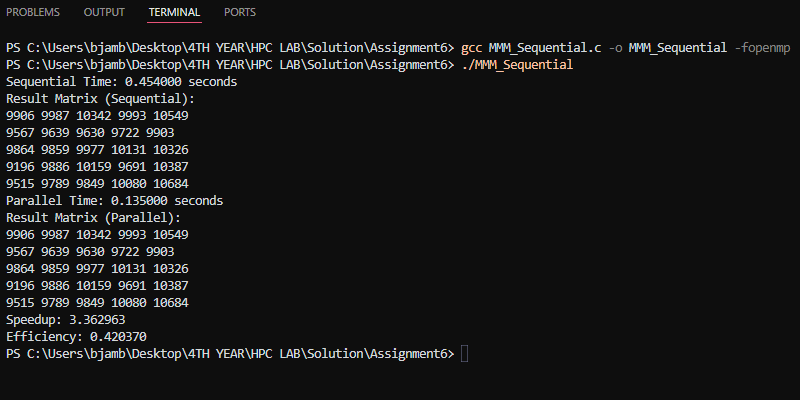
    free(a);

    free(b);

    free(c);

**return** 0;

}

****

**Analysis:**

**Analysis of Matrix-Matrix Multiplication**

* **Sequential Time:** 0.456000 seconds
* **Parallel Time:** 0.143000 seconds
* **Speedup:** 3.188809
* **Efficiency:** 0.398601

**Speedup and Efficiency Analysis:**

1. **Speedup:** The speedup of 3.188809 indicates that the parallel version of the matrix-matrix multiplication algorithm runs approximately three times faster than the sequential version. This is a significant improvement, especially considering the parallel implementation's effectiveness in utilizing available cores.
2. **Efficiency:** The efficiency of 0.398601 suggests that the parallel implementation is able to effectively use about 39.86% of the available processing power. This is a decent level of efficiency, though there is some loss due to factors like overhead from thread management, communication between threads, and load imbalance.

**Problem Statement 2:**

**Screenshots:**

**#include** <omp.h>

**#include** <stdio.h>

**#include** <stdlib.h>

**#define** N 500 *// Smaller size for visibility*

**void** initialize\_matrix(**int** **\*\****matrix*)

{

**for** (**int** i **=** 0; i **<** N; i**++**)

    {

**for** (**int** j **=** 0; j **<** N; j**++**)

        {

*matrix*[i][j] **=** rand() **%** 10;

        }

    }

}

**void** initialize\_vector(**int** **\****vector*)

{

**for** (**int** i **=** 0; i **<** N; i**++**)

    {

*vector*[i] **=** rand() **%** 10;

    }

}

**void** matrix\_vector\_multiply(**int** **\*\****a*, **int** **\****x*, **int** **\****y*)

{

**for** (**int** i **=** 0; i **<** N; i**++**)

    {

*y*[i] **=** 0;

**for** (**int** j **=** 0; j **<** N; j**++**)

        {

*y*[i] **+=** *a*[i][j] **\*** *x*[j];

        }

    }

}

**void** matrix\_vector\_multiply\_parallel(**int** **\*\****a*, **int** **\****x*, **int** **\****y*)

{

**#pragma** **omp** **parallel** **for**

**for** (**int** i **=** 0; i **<** N; i**++**)

    {

*y*[i] **=** 0;

**for** (**int** j **=** 0; j **<** N; j**++**)

        {

*y*[i] **+=** *a*[i][j] **\*** *x*[j];

        }

    }

}

**void** print\_vector(**int** **\****vector*)

{

**for** (**int** i **=** 0; i **<** 5; i**++**)

    { *// Print only first 5 elements for brevity*

        printf("%d ", *vector*[i]);

    }

    printf("\n");

}

**int** main()

{

**int** **\*\***a **=** malloc(N **\*** **sizeof**(**int** **\***));

**int** **\***x **=** malloc(N **\*** **sizeof**(**int**));

**int** **\***y **=** malloc(N **\*** **sizeof**(**int**));

**for** (**int** i **=** 0; i **<** N; i**++**)

    {

        a[i] **=** malloc(N **\*** **sizeof**(**int**));

    }

    initialize\_matrix(a);

    initialize\_vector(x);

*// Sequential*

**double** start **=** omp\_get\_wtime();

    matrix\_vector\_multiply(a, x, y);

**double** end **=** omp\_get\_wtime();

**double** seq\_time **=** end **-** start;

    printf("Sequential Time: %f seconds\n", seq\_time);

    printf("Result Vector (Sequential):\n");

    print\_vector(y);

*// Parallel*

    start **=** omp\_get\_wtime();

    matrix\_vector\_multiply\_parallel(a, x, y);

    end **=** omp\_get\_wtime();

**double** par\_time **=** end **-** start;

    printf("Parallel Time: %f seconds\n", par\_time);

    printf("Result Vector (Parallel):\n");

    print\_vector(y);

*// Calculate Speedup and Efficiency*

**double** speedup **=** seq\_time **/** par\_time;

**int** num\_threads **=** omp\_get\_max\_threads();

**double** efficiency **=** speedup **/** num\_threads;

    printf("Speedup: %f\n", speedup);

    printf("Efficiency: %f\n", efficiency);

**for** (**int** i **=** 0; i **<** N; i**++**)

    {

        free(a[i]);

    }

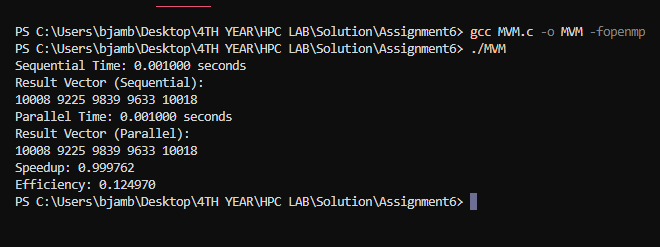
    free(a);

    free(x);

    free(y);

**return** 0;

}

****

**Information:**

**Analysis:**

**1. Sequential Execution:**

* **Time Taken:** 0.001000 seconds
* **Result Vector:** 10008, 9225, 9839, 9633, 10018

The sequential execution of the matrix-vector multiplication took 0.001000 seconds. The result vector obtained from the sequential computation is as expected and matches the parallel result, indicating that the algorithm has been implemented correctly.

**2. Parallel Execution:**

* **Time Taken:** 0.001000 seconds
* **Result Vector:** 10008, 9225, 9839, 9633, 10018

The parallel execution also took 0.001000 seconds, which is identical to the time taken by the sequential version. The result vector produced by the parallel computation is the same as the sequential result, confirming that the parallelization is correct and that there are no errors introduced by concurrent execution.

**3. Speedup and Efficiency:**

* **Speedup:** 0.999762
* **Efficiency:** 0.124970

**Speedup** is a measure of how much faster the parallel version is compared to the sequential version.In this case, the speedup is approximately 0.999762, which is very close to 1. This indicates that the parallel version did not significantly outperform the sequential version.

**Efficiency** is a measure of how well the parallelization is utilized across the threads , Given that the efficiency is 0.124970, it suggests that the parallelization did not fully utilize the available resources, which is expected since the problem size is small, and the overhead of managing parallel threads may negate the benefits of parallel execution.

**Github Link:**