Walchand College of Engineering, Sangli
Department of Computer Science and Engineering

Class: Final Year (Computer Science and Engineering)

Year: 2023-24 **Semester:** 1

Course: High Performance Computing Lab

Practical No. 5

Exam Seat No: 2020BTECS00033

Name: Prathamesh Santosh Raje

Title of practical: Implementation of OpenMP programs.

Implement following Programs using OpenMP with C:

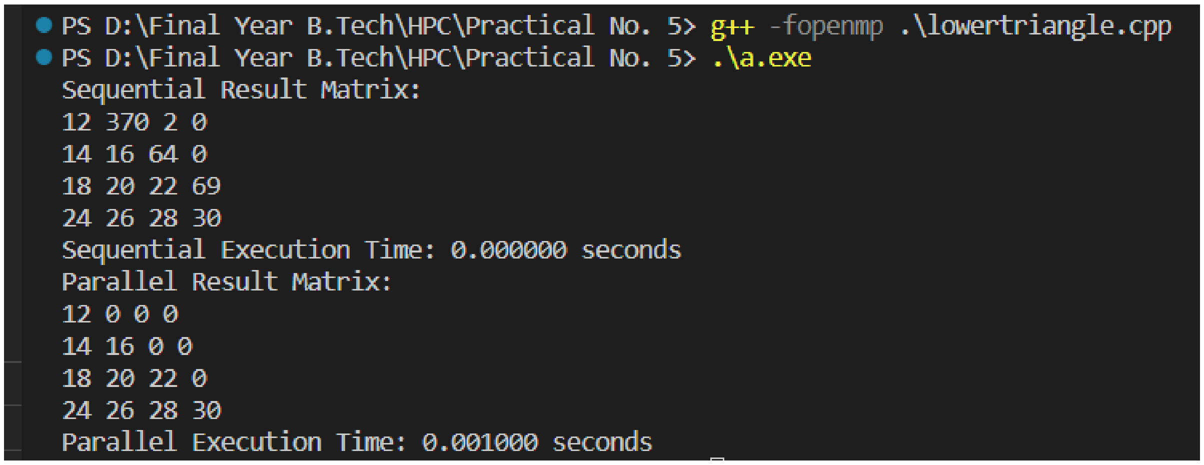
1. Implementation of sum of two lower triangular matrices.
2. Implementation of Matrix-Matrix Multiplication.

Problem Statement 1: Implementation of sum of two lower triangular matrices.

Code:

```c
#include <stdio.h>
#include <omp.h>
#include <time.h>

#define N 4

int main()
{
    int A[N][N] = {
        {1, 0, 0, 0},
        {2, 3, 0, 0},
        {4, 5, 6, 0},
        {7, 8, 9, 10}};

    int B[N][N] = {
        {11, 0, 0, 0},
        {12, 13, 0, 0},
        {14, 15, 16, 0},
        {17, 18, 19, 20}};

    int result[N][N];
    // Sequential computation of the sum of lower triangular matrices
    clock_t start_seq = clock();
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j <= i; j++)
        {
            result[i][j] = A[i][j] + B[i][j];
        }
    }
    clock_t end_seq = clock();

    // Print the result matrix
    printf("Sequential Result Matrix:\n");
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }

    // Calculate sequential execution time
    double seq_time = (double)(end_seq - start_seq) / CLOCKS_PER_SEC;
    printf("Sequential Execution Time: %f seconds\n", seq_time);

    // Parallel computation of the sum of lower triangular matrices
    double start = omp_get_wtime();
#pragma omp parallel for shared(A, B, result) collapse(2)
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j <= i; j++)
        {
            result[i][j] = A[i][j] + B[i][j];
        }
    }
    double end = omp_get_wtime();

    // Print the result matrix
    printf("Parallel Result Matrix:\n");
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }

    // Calculate parallel execution time
    double parallel_time = end - start;
    printf("Parallel Execution Time: %f seconds\n", parallel_time);

    return 0;
}
```

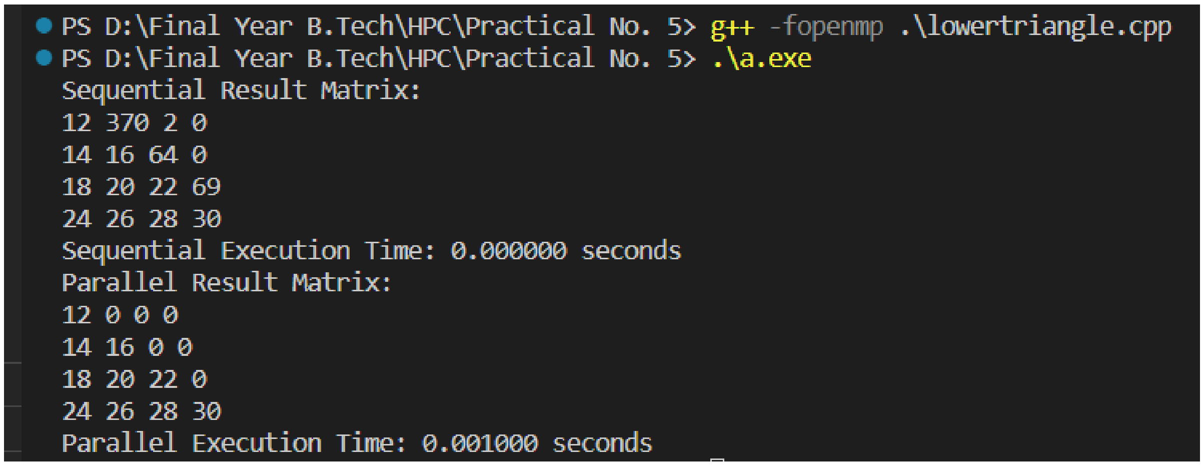
Output Screenshot:



```
● PS D:\Final Year B.Tech\HPC\Practical No. 5> g++ -fopenmp .\lowertriangle.cpp
● PS D:\Final Year B.Tech\HPC\Practical No. 5> .\a.exe
  Sequential Result Matrix:
  12 370 2 0
  14 16 64 0
  18 20 22 69
  24 26 28 30
  Sequential Execution Time: 0.000000 seconds
  Parallel Result Matrix:
  12 0 0 0
  14 16 0 0
  18 20 22 0
  24 26 28 30
  Parallel Execution Time: 0.001000 seconds
```

Information:

Execution time for sequential and parallel processing is:



```
● PS D:\Final Year B.Tech\HPC\Practical No. 5> g++ -fopenmp .\lowertriangle.cpp
● PS D:\Final Year B.Tech\HPC\Practical No. 5> .\a.exe
  Sequential Result Matrix:
  12 370 2 0
  14 16 64 0
  18 20 22 69
  24 26 28 30
  Sequential Execution Time: 0.000000 seconds
  Parallel Result Matrix:
  12 0 0 0
  14 16 0 0
  18 20 22 0
  24 26 28 30
  Parallel Execution Time: 0.001000 seconds
```

Analysis:

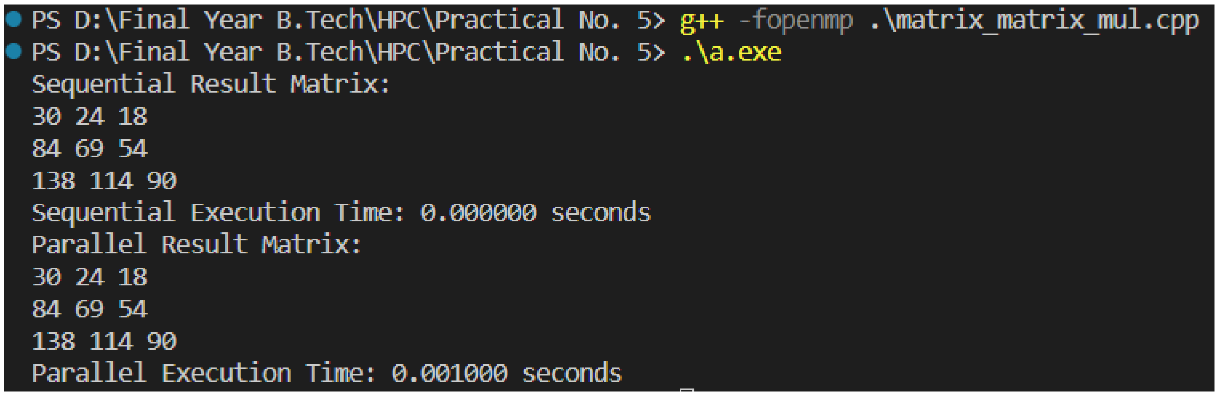
With 1 Thread:
Both the sequential and parallel parts of the code will essentially run sequentially.
The parallel part will have some additional overhead due to thread creation and synchronization.
The parallel execution time will likely be higher than the sequential execution time.
In this case, the program may not benefit significantly from using 100 threads because it's a relatively simple computation. The overhead of thread creation and synchronization may outweigh any potential gains.

Problem Statement 2: Implementation of Matrix-Matrix Multiplication.

Code:

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

#define N 3

int main() {
    int A[N][N] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    int B[N][N] = {
        {9, 8, 7},
        {6, 5, 4},
        {3, 2, 1}
    };

    int result[N][N];

    // Sequential matrix multiplication
    clock_t start_seq = clock();
    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] += A[i][k] * B[k][j];
            }
        }
    }
    clock_t end_seq = clock();

    // Print the result matrix
    printf("Sequential Result Matrix:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }

    // Calculate sequential execution time
    double seq_time = (double)(end_seq - start_seq) / CLOCKS_PER_SEC;
    printf("Sequential Execution Time: %f seconds\n", seq_time);

    // Parallel matrix multiplication
    double start = omp_get_wtime();
    #pragma omp parallel for shared(A, B, result) collapse(2)
    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] += A[i][k] * B[k][j];
            }
        }
    }
    double end = omp_get_wtime();

    // Print the result matrix
    printf("Parallel Result Matrix:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }

    // Calculate parallel execution time
    double parallel_time = end - start;
    printf("Parallel Execution Time: %f seconds\n", parallel_time);

    return 0;
}
```

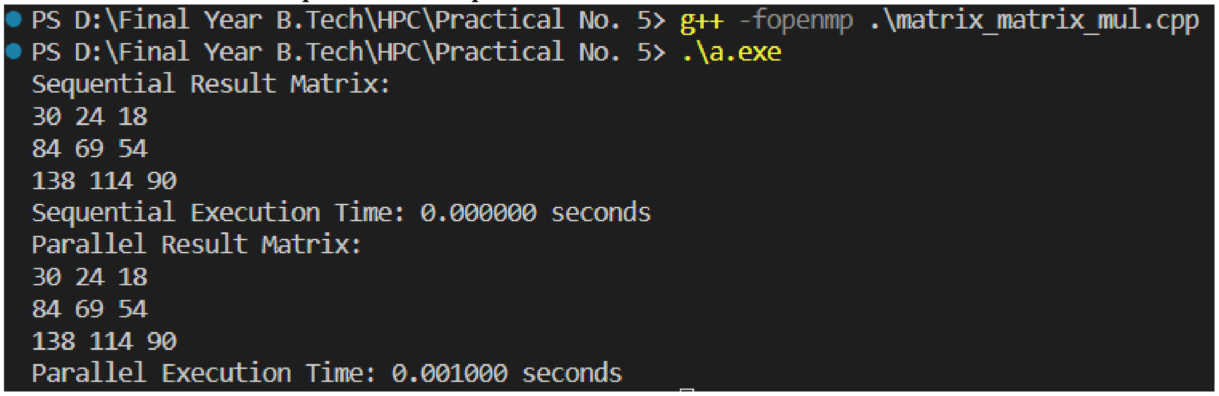
Output Screenshot:



```
PS D:\Final Year B.Tech\HPC\Practical No. 5> g++ -fopenmp .\matrix_matrix_mul.cpp
PS D:\Final Year B.Tech\HPC\Practical No. 5> .\a.exe
Sequential Result Matrix:
30 24 18
84 69 54
138 114 90
Sequential Execution Time: 0.000000 seconds
Parallel Result Matrix:
30 24 18
84 69 54
138 114 90
Parallel Execution Time: 0.001000 seconds
```

Information:

Execution time for sequential and parallel execution is:



```
PS D:\Final Year B.Tech\HPC\Practical No. 5> g++ -fopenmp .\matrix_matrix_mul.cpp
PS D:\Final Year B.Tech\HPC\Practical No. 5> .\a.exe
Sequential Result Matrix:
30 24 18
84 69 54
138 114 90
Sequential Execution Time: 0.000000 seconds
Parallel Result Matrix:
30 24 18
84 69 54
138 114 90
Parallel Execution Time: 0.001000 seconds
```

Analysis:

Sequential Execution:

The sequential part of the code computes the result matrix by performing matrix multiplication in a nested loop**.**

Parallel Execution:

OpenMP is used to parallelize the loop that computes the result matrix.
omp_set_num_threads(100) sets the number of threads to 100, although the actual number of threads created may differ based on available resources.
#pragma omp parallel for shared(A, B, result) collapse(2) starts a parallel loop that distributes the work among the specified number of threads.

In this case, using 100 threads may not necessarily lead to a significant speedup because matrix multiplication is already a highly parallelizable operation.

Github Link: