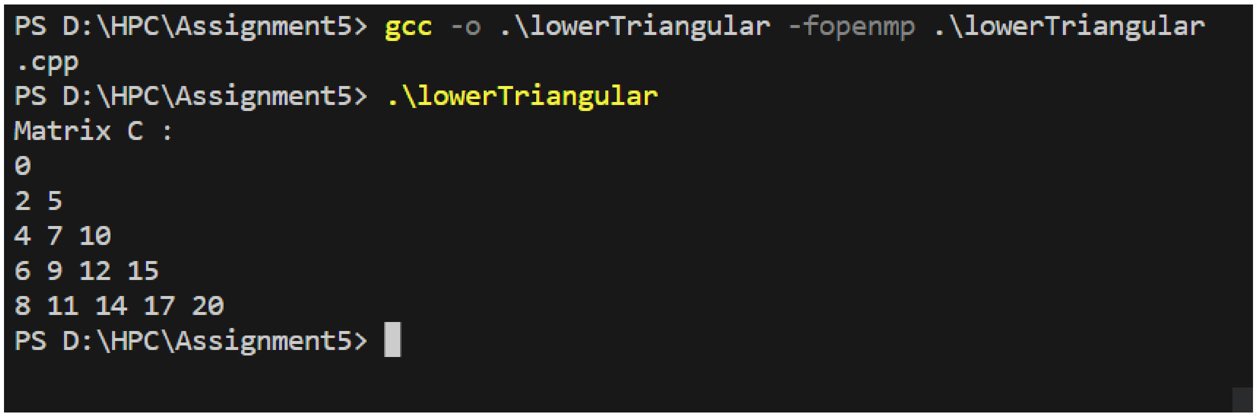
Assignment No. 5

PRN: 2020BTECS00025
Name: Sahil Santosh Otari
Course: High Performance Computing Lab
Title of practical: Study and implementation of Open MP program.

1. Implementation of sum of two lower triangular matrices.

```
#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define N 5

void sumLowerTriangularMatrices(int A[][N], int B[][N], int C[][N]) {

    #pragma omp parallel for collapse(2)

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

        for (int j = 0; j <= i; j++) {

            C[i][j] = A[i][j] + B[i][j];

        }

    }

}


int main() {

    int A[N][N], B[N][N], C[N][N];
    for (int i = 0; i < N; i++) {

        for (int j = 0; j <= i; j++) {

            A[i][j] = i+j;

            B[i][j] = i+2*j;

        }

    }

    sumLowerTriangularMatrices(A, B, C);



    // Display the result matrix C (sum of A and B)

    printf("Matrix C :\n");

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

        for (int j = 0; j <= i; j++) {

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

        }

        printf("\n");

    }



    return 0;

}
```
Output:

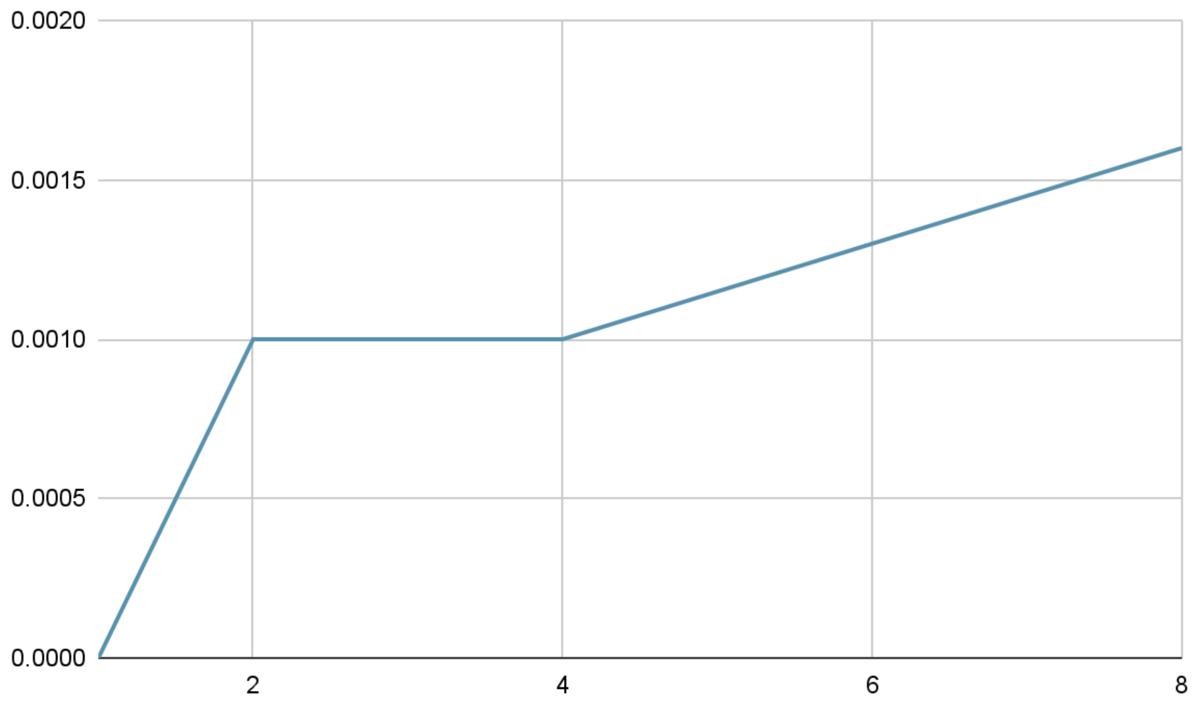

```
PS D:\HPC\Assignment5> gcc -o .\lowerTriangular -fopenmp .\lowerTriangular
.cpp
PS D:\HPC\Assignment5> .\lowerTriangular
Matrix C :
0
2 5
4 7 10
6 9 12 15
8 11 14 17 20
PS D:\HPC\Assignment5>
```

Analysis:

1. Parallelism: The primary advantage of this code is the use of Open MP to parallelize the addition of lower triangular matrices. This can significantly speed up the operation, especially for larger matrices and on multicore processors.
2. Performance: The performance gain from parallelism depends on the matrix size and the number of available CPU cores. For larger matrices, the speed-up achieved by parallel execution can be more significant.
3. Load Balancing: This code does not explicitly handle load balancing among threads. Depending on the matrix size and the number of threads, some threads may complete their work earlier than others. In more complex applications, load balancing strategies may be necessary for optimal performance.
4. Data Dependency: In this code, there are no data dependencies, so threads can work independently on different parts of the matrices. However, in other matrix operations, you may need to consider data dependencies and synchronization mechanisms.
5. Overall, this code serves as a basic example of parallelizing matrix addition using Open MP, demonstrating how parallelism can be utilized to improve the efficiency of matrix operations.

Thread	1	2	4	8
Time	0.000000	0.001000	0.001000	0.001600

Graph:



2. Implementation of Matrix-Matrix Multiplication.

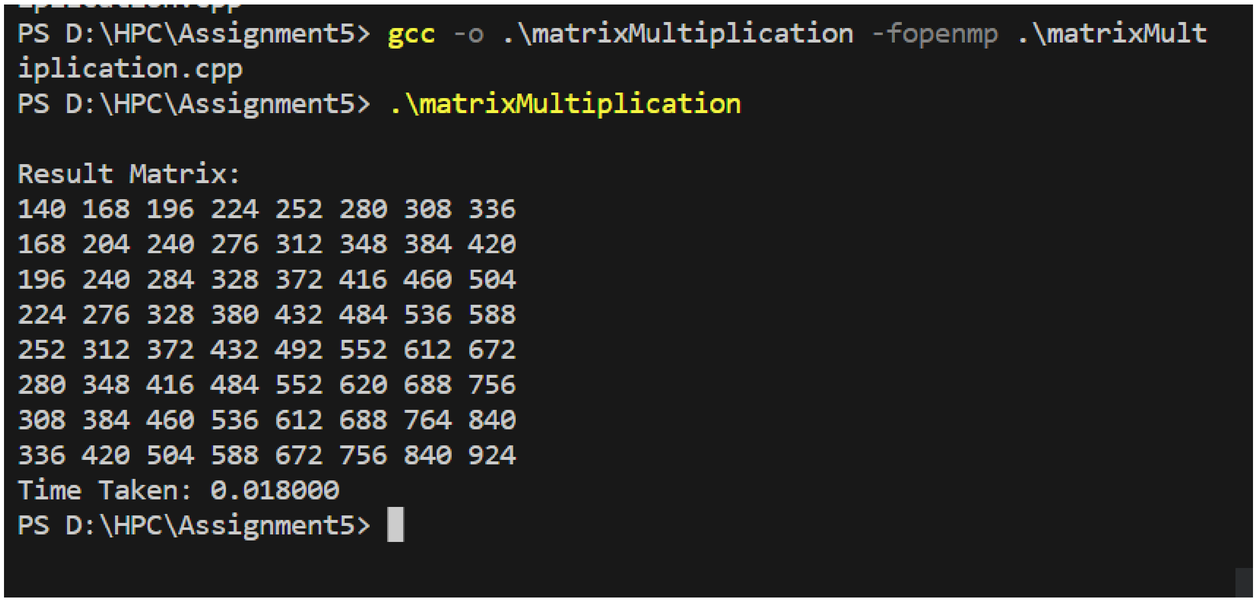
Source Code:

```
#include <stdio.h>

#include <omp.h>

#include<sys/time.h>



#define N 8  // Size of matrices
```

```c
void matrixMatrixMultiplication(int A[][N], int B[][N], int
result[][N]) {

    #pragma omp parallel for

    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];

            }

        }

    }

}


void initial(int A[][N]){

    #pragma omp parallel for

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

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

                A[i][j] = i+j;

        }

    }
```

```c
}

int main() {
    clock_t start, end;
    start = clock();
    int matrixA[N][N] = {0};
initial(matrixA);
    int matrixB[N][N] = {0};
initial(matrixB);
    int result[N][N] = {0};

    matrixMatrixMultiplication(matrixA, matrixB, result);

    printf("\nResult Matrix:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }
    end = clock();
    double duration = ((double)end-start)/CLOCKS_PER_SEC;

    printf("Time Taken: %f",duration);



    return 0;

}
```

Output:



```
PS D:\HPC\Assignment5> gcc -o .\matrixMultiplication -fopenmp .\matrixMult
iplication.cpp
PS D:\HPC\Assignment5> .\matrixMultiplication

Result Matrix:
140 168 196 224 252 280 308 336
168 204 240 276 312 348 384 420
196 240 284 328 372 416 460 504
224 276 328 380 432 484 536 588
252 312 372 432 492 552 612 672
280 348 416 484 552 620 688 756
308 384 460 536 612 688 764 840
336 420 504 588 672 756 840 924
Time Taken: 0.018000
PS D:\HPC\Assignment5>
```

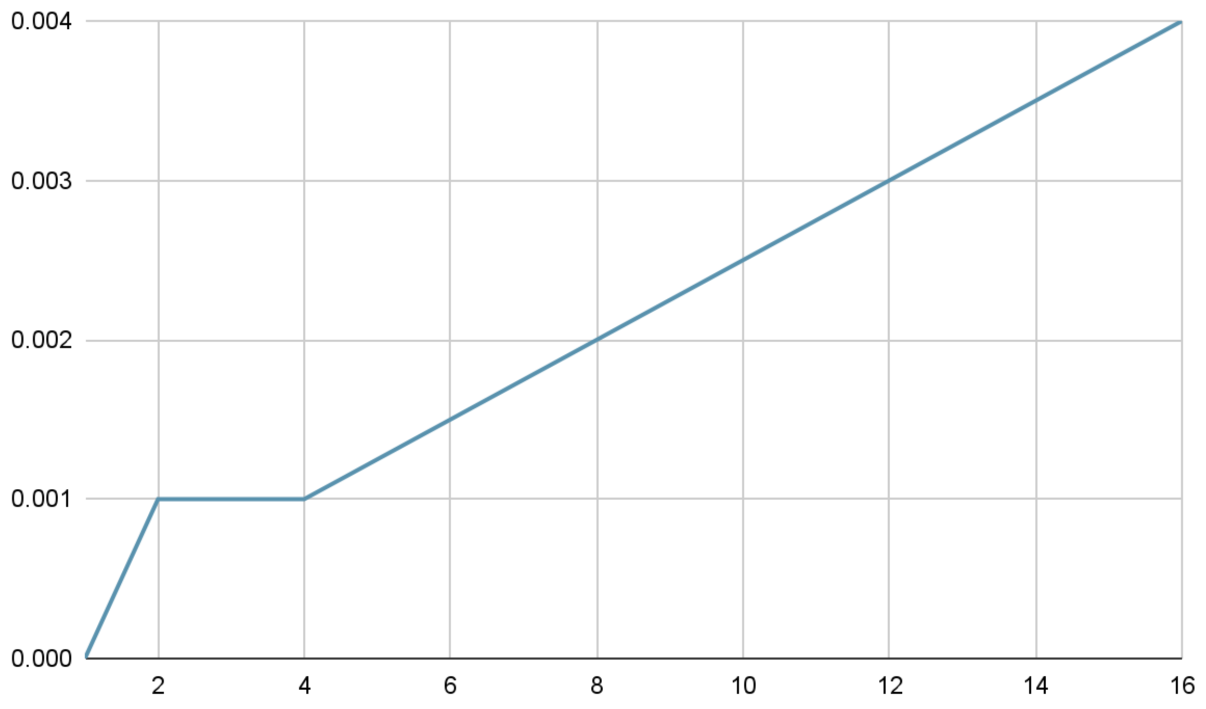
Information:

1. Matrix Sizes: The code defines the sizes of the matrices `A`, `B`, and `C` using constants `N`, `M`, and `P`. These constants determine the number of rows and columns in each matrix. You can adjust these values according to your specific matrix dimensions.
2. Matrix Initialization: The code initializes matrices `A` and `B` with sample values. It uses nested loops to assign values to each element of the matrices. You can replace these initializations with your own matrices or input method.
3. Matrix Multiplication Function: The `matrixMultiply` function is responsible for performing matrix multiplication. It uses nested loops to iterate through the matrices and calculate the product matrix `C`. Open MP parallelism is applied to the outer loops, enabling concurrent execution of matrix multiplication for improved performance.

4. Displaying the Result: After matrix multiplication, the code prints the result matrix `C` to the console.

Threads	1	2	4	16
Time	0.000000	0.001000	0.001000	0.004000

Graph:


0.004
0.003
0.002
0.001
0.000
2
4
6
8
10
12
14
16