TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

VIỆN TOÁN ỨNG DỤNG VÀ TIN HỌC

BÁO CÁO MÔN

TÍNH TOÁN SONG SONG

Giảng viên hướng dẫn:

TS. Đoàn Duy Trung

Thành viên nhóm: 05

Họ và Tên	MSSV
Nguyễn Hữu Thuật	20185410
Trang Hải Long	20185382
Trần Xuân Hiếu	20185354
Lại Tiến Long	20185376

CHƯƠNG TRÌNH

```
/*
INPUT:  Matrix of m rows, n cols
OUTPUT: The average value of each column's highest value for all columns of
the input matrix


Ex: Given a 3x3 matrix such as:
        [ 100   200  300
          400   200  500
          100   900  100 ]
--> the highest value of each column respectively: 400, 900, 500
    --> the expected result of the calculation: (400 + 900 + 500) / 3 = 600
*/


#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <chrono>
#include <iostream>
using namespace std;
using namespace std::chrono;


/*  CONSTANTS  */
constexpr auto NUM_THREADS = 8;


/*  FUNCTION DECLARATIONS  */
// Memory allocation and definition of objects' elements
```

```
void ProcessInitialization(double*& Matrix, double*& tempResults, int& Row,
int& Col);

// Generating random values for the input matrix

void DummyDataInitialization(double* Matrix, int Row, int Col);

// Calculations

double SerialResultCalculation(double* Matrix, double*& tempResults, int Row,
int Col);

double ParallelResultCalculation(double* Matrix, double* tempResults, int
Row, int Col);

// Extra Parallel function using only one parallel code block

double xParallelResultCalculation(double* Matrix, double* tempResults, int
Row, int Col);

// Function for computational process termination

void ProcessTermination(double* Matrix, double* tempResults);



int main() {

    /*  DECLARE VARIABLES */

    int Row, Col;          // # of Rows and Columns of the input matrix

    double* Matrix;        // Matrix input

    double* tempResults;   // Vector to store intermediate results

    double ParallelResult; // result calculated by Parallel algorithm

    double SerialResult;   // Result calculated by Serial algorithm


    /*  INIT  */

    omp_set_num_threads(NUM_THREADS);

    ProcessInitialization(Matrix, tempResults, Row, Col);


    /*  PARALLEL EXECUTION  */

    auto pStart = high_resolution_clock::now();
```

```
    ParallelResult = xParallelResultCalculation(Matrix, tempResults, Row,
Col);
    auto pFinish = high_resolution_clock::now();
    auto pDuration = duration_cast<microseconds>(pFinish - pStart);

    /*  TEST RESULT WITH A SERIAL ALGORITHM  */
    auto sStart = high_resolution_clock::now();
    SerialResult = SerialResultCalculation(Matrix, tempResults, Row, Col);
    auto sFinish = high_resolution_clock::now();
    auto sDuration = duration_cast<microseconds>(sFinish - sStart);

    /*  PRINT RESULT  */
    // Printing serial and parallel results
    printf("Serial Result = %lf\n", SerialResult);
    printf("Parallel Result = %lf\n", ParallelResult);
    // Printing the time spent by the calculations
    cout << "Time of parallel execution: " << pDuration.count() << "
microseconds" << endl;
    cout << "Time of serial execution: " << sDuration.count() << "
microseconds" << endl;

    /* COMPUTATIONAL PROCESS TERMINATION */
    ProcessTermination(Matrix, tempResults);
    return 0;
}

/*  FUNCTION DEFINITIONS  */

void DummyDataInitialization(double* Matrix, int Row, int Col)
{
    int i, j;
```

```
    srand(unsigned(clock()));
    for (i = 0; i < Row; i++)
    {
        for (j = 0; j < Col; j++)
            Matrix[i * Col + j] = 3.0 + rand() % 2000000;   // Using random
number for matrix elements
    }
}


void ProcessInitialization(double*& Matrix, double*& tempResults, int& Row,
int& Col)
{
    do {
        printf("\nEnter row of the initial objects: ");
        scanf_s("%d", &Row);
        printf("\nChosen objects row= %d\n", Row);
        if (Row <= 0)printf("\nRow of objects must be greater than 0!\n");
    } while (Row <= 0);


    do {
        printf("\nEnter Colum of the initial objects: ");
        scanf_s("%d", &Col);
        printf("\nChosen objects Colum= %d\n", Col);
        if (Col <= 0)printf("\nColume of objects must be greater than 0!\n");
    } while (Col <= 0);


    // Declare matrix and tempResults array
    Matrix = new double[Row * Col];
    tempResults = new double[Col];
```

```
    // Initialize matrix elements
    DummyDataInitialization(Matrix, Row, Col);
}


double SerialResultCalculation(double* Matrix, double*& tempResults, int Row,
int Col)
{
    int i, j;
    double result = 0;


    for (j = 0; j < Col; j++)
    {
        tempResults[j] = Matrix[j];
        for (i = 0; i < Row; i++)
        {
            if (Matrix[i * Col + j] > tempResults[j])
                tempResults[j] = Matrix[i * Col + j];
        }
        result += tempResults[j] / Col;
    }


    return result;
}


double ParallelResultCalculation(double* Matrix, double* tempResults, int
Row, int Col)
{
    int i, j;
    double sum[NUM_THREADS] = { 0.0 };
    double result = 0.0;
```

```
    // get largest number of each column, store in tempResults
#pragma omp parallel for private(i)
    for (j = 0; j < Col; j++)
    {
        tempResults[j] = Matrix[j];
        for (i = 0; i < Row; i++)
        {
            if (Matrix[i * Col + j] > tempResults[j])
                tempResults[j] = Matrix[i * Col + j];
        }
    }

#pragma omp parallel
    {
        int i, nthreads;
        nthreads = omp_get_num_threads();

        for (i = omp_get_thread_num(); i < Col; i += nthreads)
            sum[omp_get_thread_num()] += tempResults[i];
    }

    for (i = 0; i < NUM_THREADS; i++)
        result += sum[i];

    return result / Col;
}

double xParallelResultCalculation(double* Matrix, double* tempResults, int
Row, int Col)
```

```
{
    int i, j, nthreads;
    double Result[NUM_THREADS] = { 0.0 };
    double result = 0;

#pragma omp parallel private(i, j)
    {
        nthreads = omp_get_num_threads();
        for (j = omp_get_thread_num(); j < Col; j += nthreads)
        {
            tempResults[j] = Matrix[j];
            for (i = 0; i < Row; i++)
            {
                if (Matrix[i * Col + j] > tempResults[j])
                    tempResults[j] = Matrix[i * Col + j];
            }
            Result[omp_get_thread_num()] += tempResults[j];
        }
    }

    for (i = 0; i < NUM_THREADS; i++)
    {
        result += Result[i] / Col;
    }

    return result;
}

void ProcessTermination(double* Matrix, double* tempResults) {
    delete[] Matrix;
```

```
    delete[] tempResults;
    }
}
```

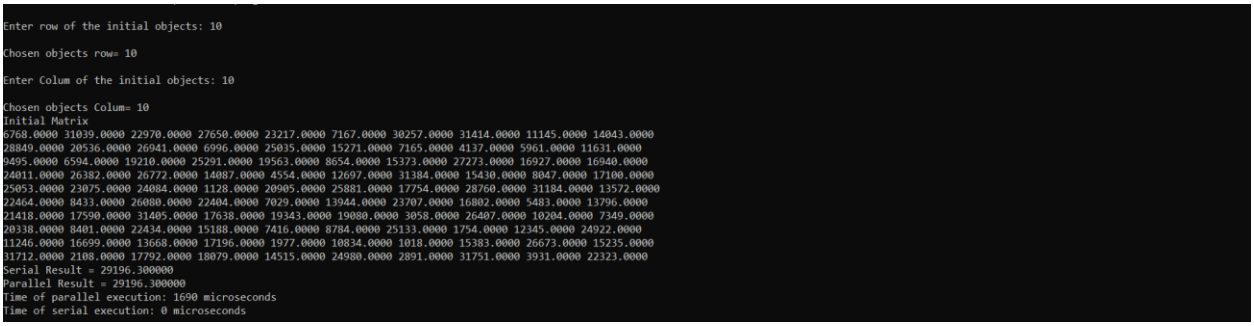
KẾT QUẢ CHẠY CHƯƠNG TRÌNH

I. Chạy chương trình với các trường hợp:

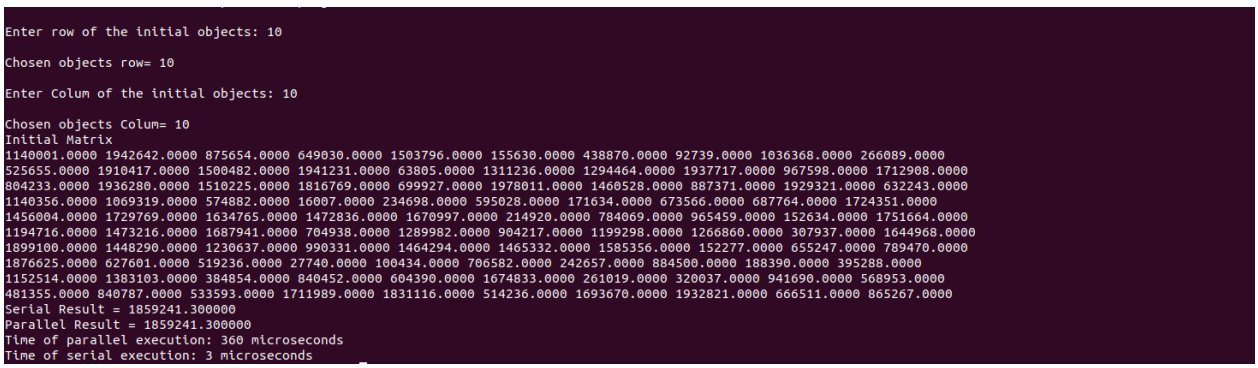
1. Số luồng = 4; Số hàng = 10 ; Số cột =10 (In ma trận ở trường hợp này)
2. Số luồng = 4 ; Số hàng = 10 ; Số cột =1.000.000
3. Số luồng = 8 ; Số hàng = 10; Số cột 1.000.000
4. Số luồng = 8 ; Số hàng = 1.000.000; Số cột =10
5. Số luồng = 16 ; Số hàng =1.000.000; Số cột =10
6. Số luồng = 16; Số hàng = 111; Số cột = 123
7. Số luồng = 32; Số hàng = 1000; Số cột =1000

1. Trường hợp 1: Số luồng = 4; Số hàng = 10 ; Số cột =10

1.1. window



1.2. Linux



2. Trường hợp 2: Số luồng = 4 ; Số hàng = 10 ; Số cột =1.000.000
    2.1. window

    2.2. Linux

3. Trường hợp 3: Số luồng = 8 ; Số hàng = 10; C Số cột 1.000.000
    3.1. window

    3.2. Linux

4. Trường hợp 4: Số luồng = 8 ; Số hàng = 1.000.000; Số cột =10
    4.1. window

4.2. Linux

5. Trường hợp 5: Số luồng = 16 ; Số hàng =1.000.000; Số cột =10

5.1. window

5.2. Linux

6. Trường hợp 6: Số luồng = 16; Số hàng = 111; Số cột = 123

6.1. window

6.2. Linux

7. Trường hợp 7: Số luồng = 32; Số hàng = 1000; Số cột =1000

7.1. window

7.2. Linux

II. Thống kê

Bảng số liệu thống kê kết quả khi thực hiện giải thuật tuần tự và song song

STT	Kích thước ma trận		Số luồng (T)	Thời gian thực hiện tuần tự (ms)		Thời gian thực hiện song song (ms)		Hiệu suất (tuần tự/song song)	
	Số hàng (R)	Số cột (C)		Win	Linux	Win	Linux	Win	Linux
Test 1	10	10	4	0	3	1690	360	0.0000	0.0021
Test 2	10	1.000.000	4	98298	53004	27299	34556	0.9002	0.3882
Test 3	10	1.000.000	8	90567	54412	37957	19049	0.2982	0.3570
Test 4	1.000.000	10	8	98055	59839	31007	13819	0.3953	0.5413
Test 5	1.000.000	10	16	85703	59330	27168	13387	0.1971	0.2770
Test 6	111	123	16	65	236	8946	1609	0.0005	0.0092
Test 7	1000	1000	32	8031	5668	14400	3965	0.0174	0.0447

III. Nhận xét

- Khi tăng số lượng luồng xử lí, thời gian được xử lí giảm đi
- Thời gian chạy ở quá trình song song ít hơn ở quá trình tuần tự. Tuy nhiên khi kích thước ma trận nhỏ thì quá trình chạy song song mất nhiều thời gian hơn quá trình chạy tuần tự (bởi vì thời gian chờ trong quá trình chạy song song lâu hơn quá trình tính toán trong quá trinh chạy tuần tự).
- Chương trình thực hiện tính toán song song theo cột, nên với ma trận có cùng số phần tử (tích Row*Column bằng nhau) thì ma trận có nhiều hàng hơn sẽ xử lí lâu hơn vì quá trình duyệt các phần tử trong từng cột là duyệt tuần tự.

STT	Kích thước ma trận		Thời gian thực hiện tuần tự (ms)		Thời gian thực hiện song song (ms)		Hiệu suất (tuần tự/song song)	
	Số hàng (R)	Số cột (C)	Win	Linux	Win	Linux	Win	Linux
Test 1	10	10	0	3	1690	360	0.0000	0.0021
Test 2	10	1.000.000	98298	53004	27299	34556	0.9002	0.3882
Test 3	10	1.000.000	90567	54412	37957	19049	0.2982	0.3570
Test 4	1.000.000	10	98055	59839	31007	13819	0.3953	0.5413
Test 5	1.000.000	10	85703	59330	27168	13387	0.1971	0.2770
Test 6	111	123	65	236	8946	1609	0.0005	0.0092
Test 7	1000	1000	8031	5668	14400	3965	0.0174	0.0447

IV. Báo cáo bài toán ma trận nhân vector

STT	Kích thước ma trận	Thời gian thực hiện tuần tự	Thời gian thực hiện song song	Hiệu suất
Test № 1	10	0.000001	0.004443	2.81341E-05
Test № 2	100	0.000029	0.003529	0.001027203
Test № 3	1000	0.004089	0.005657	0.09035266
Test № 4	2000	0.020137	0.007410	0.339692982
Test № 5	3000	0.067213	0.014322	0.586623726
Test № 6	4000	0.079390	0.038741	0.256156269
Test № 7	5000	0.151565	0.063098	0.30025714
Test № 8	6000	0.217916	0.109199	0.249448255
Test № 9	7000	0.148064	0.081744	0.226414171
Test № 10	8000	0.224541	0.074702	0.375727892
Test № 11	9000	0.357941	0.088998	0.502737421
Test № 12	10 000	0.350015	0.118220	0.370088606