Saint Petersburg National Research University of Information Technologies, Mechanics and Optics (ITMO University)
Faculty of Informational Technologies and Programming

REPORT

about laboratory work № 1

«Matrix Multiplication in OpenMP»

Student

Sarafanov M.I. | C4134

(Surname, initials) | Group

Saint-Petersburg, 2019

1. GOAL OF LABORATORY WORK

The purpose of this lab is to master the tool for parallelizing algorithms in the C ++ programming language - OpenMP. And also to establish how much, when using OpenMP technology, it is possible to reduce the operating time of the matrix multiplication algorithm.

2. TASK DEFINITION

The operation of matrix multiplication is one of the main tasks of matrix calculations. When multiplying square matrices of size n × n, the number of operations performed is of the order of O (n^3). Sequential matrix multiplication algorithms are known to have less computational complexity (for example, the Strassen’s algorithm), but these algorithms require some effort to master [А. А. Лабутина, Учебный курс "Введение в методы параллельного программирования", раздел "Параллельные методы матричного умножения" // Нижегородский государственный университет им. Н.И.Лобачевского, 2007]. On the other hand, the matrix multiplication algorithm is quite easy to parallelize, which will help increase the efficiency of calculations.

3. BRIEF THEORY

OpenMP (Open Multi-Processing) is an open standard for parallelizing C, C ++, and Fortran programs. Describes a set of compiler directives, library procedures, and environment variables that are designed to program multithreaded applications on multiprocessor systems with shared memory.

A sequential program is taken as the basis, and to create its parallel version the user is provided with a set of directives, functions and environment variables. It is assumed that the created parallel program will be portable between different shared memory computers that support the OpenMP API.

The OpenMP interface is designed as a standard for programming on scalable SMP systems in a shared memory model. The OpenMP standard includes specifications for a set of compiler directives, helper functions, and environment variables. OpenMP implements parallel computing using multithreading, in which the "main" thread creates a set of "subordinate" threads, and the task is distributed between them. It is assumed that threads run in parallel on a machine with multiple processors, and the number of processors does not have to be greater than or equal to the number of threads.

When using OpenMP, the SPMD model (Single Program Multiple Data) of parallel programming is assumed, within which the same code is used for all parallel threads. The program begins with a sequential area - at first one process (thread) runs, when entering the parallel area, a certain number of processes are generated, between which parts of the code are further distributed [Антонов, А.С. Технологии параллельного программирования MPI и OpenMP: Учебное пособие / А.С. Антонов. - М.: МГУ, 2012. - 344 с.].

4. ALGORITHM (METHOD) OF IMPLEMENTATION

Configuration of copmuter: AMD Ryzen 5 2400G BOX65 BT, 16Gb RAM, Ubuntu 18 x64, gcc compiler.

For calculations, the following algorithm was used:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <sys/time.h>
#include <iostream>
#include <fstream>
using namespace std;
// #define N 1000

int main(int argc, char *argv[])
{   int i,j,k; // Те индексы, что мы будем использовать - это целочисленные значения
    double run_time;
    int const threads = stoi(argv[1]); // Количество потоков
    int const N = stoi(argv[2]); // Размер матрицы N*N строк

    cout << threads << '\n';

    int **A = new int*[N]; // Объявление массива A
    int **B = new int*[N]; // Объявление массива B

    for (int count = 0; count < N; count++)
    { // Выделение памяти под массив
        A[count] = new int[N];
        B[count] = new int[N];
    }

    for (i= 0; i< N; i++)
        for (j= 0; j< N; j++)
    {
            A[i][j] = 9;
    }
```

```

    struct timeval tv1, tv2;
    struct timezone tz;
    gettimeofday(&tv1, &tz);
    omp_set_num_threads(threads);

    // Переменные i,j,k для каждого потока будут свои, а вот A и B - общие
    // Динамическое распределение итераций с фиксированным размером блока
    #pragma omp parallel for private(i,j,k) shared(A,B) schedule(dynamic, 4)
    for (i = 0; i < N; ++i) {
        for (j = 0; j < N; ++j) {
            for (k = 0; k < N; ++k) {
                B[i][j] += A[i][k] * A[k][j];
            }
        }
    }


    gettimeofday(&tv2, &tz);
    ofstream myfile;
    myfile.open ("example.csv", ios::app); // Открываем файл на запись
    run_time = (double) (tv2.tv_usec-tv1.tv_usec) / 1000000 + (double) (tv2.tv_sec-tv1.tv_sec); // Время работы алгоритма (Милисекунды плюс секунды)
    printf("\n %f \n", run_time);
    myfile << threads << ";" << N << ";" << run_time << "\n"; // Записываем строку - Размер матрицы
    myfile.close(); // Заканчиваем запись
    // Освобождаем память
    delete[]A;
    delete[]B;
    return 0;
}

```

The program was run through the command line in the Linux operation system a number of times.

5. RESULT AND EXPERIMENTS

The results of measurements of the speed of the algorithm of single-threaded and multi-threaded matrix multiplication are presented in table 1.

Table 1.

Acceleration relative to a single-threaded algorithm

Amount of threads	Matrix dimensions	Run time	Acceleration relative to a single-threaded algorithm
1	100	0,005	-
1	200	0,04	-
1	300	0,15	-
1	400	0,37	-
1	500	0,74	-
1	600	1,44	-
1	700	2,07	-
1	800	3,19	-
1	900	4,20	-
1	1000	6,22	-
1	1100	9,86	-
1	1200	14,31	-
6	100	0,002	3,4
6	200	0,01	3,5
6	300	0,04	3,9
6	400	0,09	3,9

6	500	0,19	3,8
6	600	0,32	4,5
6	700	0,51	4,1
6	800	0,83	3,8
6	900	1,24	3,4
6	1000	1,51	4,1
6	1100	2,78	3,5
6	1200	3,85	3,7
11	100	0,002	3,0
11	200	0,01	3,1
11	300	0,04	3,4
11	400	0,09	4,1
11	500	0,18	4,1
11	600	0,32	4,5
11	700	0,50	4,2
11	800	0,76	4,2
11	900	1,29	3,2
11	1000	1,52	4,1
11	1100	2,80	3,5
11	1200	3,72	3,9

As can be seen from table 1, using parallelization of the algorithm, it was possible to reduce the operating time by an average of 4 times.

The results of the algorithm with a different number of threads are shown in Figures 1-3.

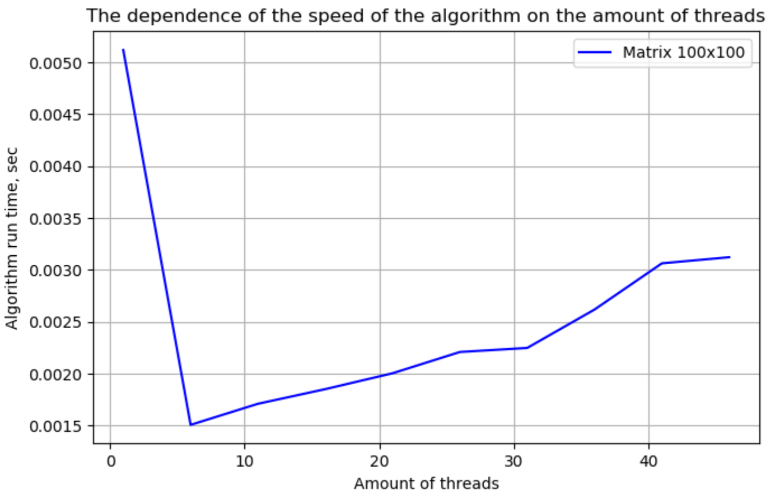

Figure 1. Runtime for matrix multiplication 100x100

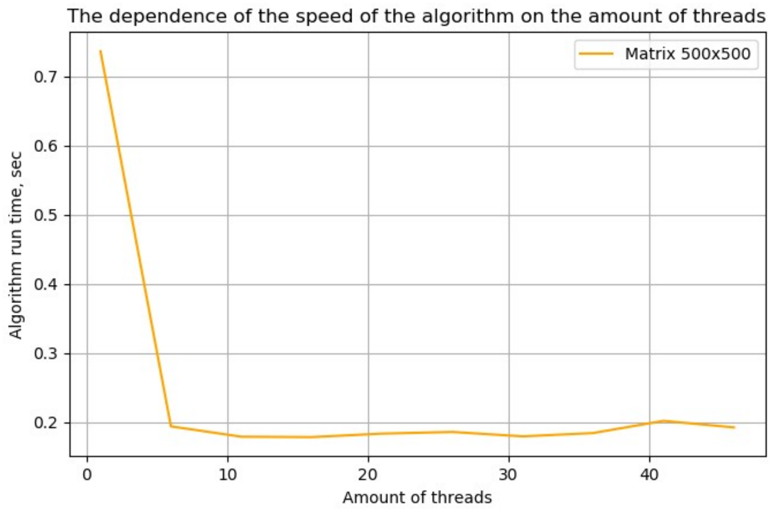

Figure 2. Runtime for matrix multiplication 500x500

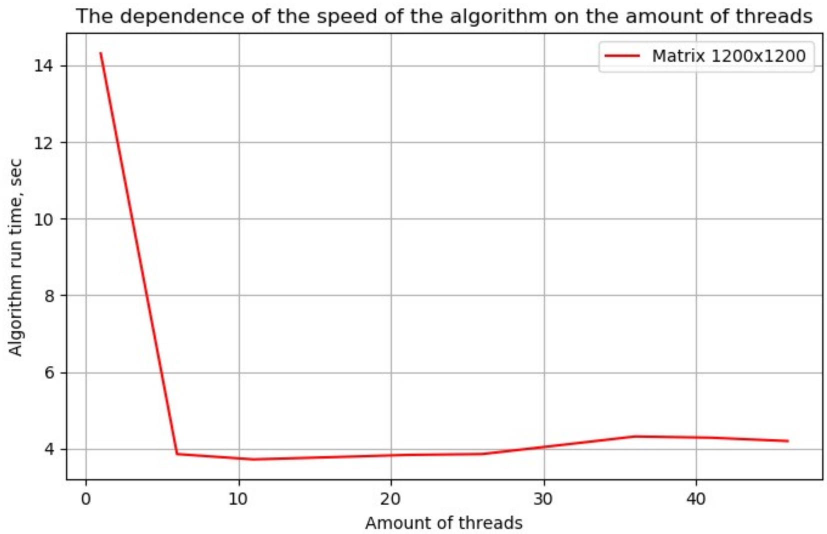

Figure 3. Runtime for matrix multiplication 1200x1200

It can be assumed that the best number of threads is the number of cores we have at our disposal. Since there are 4 cores on my computer - I think that the algorithm will work best with just so many threads, as can be seen from the graph. Otherwise, a lot of threads are allocated to one core and more time can be spent on their separation and synchronization.

6. CONCLUSION

As a result of the laboratory work, experience was gained in writing algorithms in C ++, the skills of parallelizing algorithms using OpenMP directives were mastered. By the example of matrix multiplication, an increase in the productivity of calculations by 3-4 times was demonstrated. This allows us to conclude that similar parallelization can be applied to other more complex algorithms, which allows us to do more calculations in the same or less time as a single-threaded algorithm.