Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе № 4 по курсу «Анализ алгоритмов»

Тема Параллельные вычисления на основе нативных потоков

Студент Косарев А.А.

Группа ИУ7-51Б

Оценка (баллы) ____________________

Преподаватель Волкова Л. Л., Строганов Ю.В.

Москва — 2023 г.

Содержание

Введение

Зачастую в сфере информационных технологий используют параллельную обработку данных, которая позволяет уменьшить время работы программы. Одним из примеров такой обработки является конвейерная обработка. Суть та же, что и при работе реальных конвейрных лент - материал (данное) поступает на обработку, после окончания обработки материал передается на место следующего обработчина, при этом предыдыдущий обработчик не ждет полного цикла обработки материала, а получает новый материал и работает с ним.

Целью данной работы является изучение принципов конвейрной обработки данных. Для достижения поставленной цели необходимо выполнить следующие задачи:

— изучить основы конвейрной обработки данных;

— описать алгоритмы обработки матрицы, которые будут использоваться в текущей лабораторной работе;

— привести схемы конвейрной и линейной обработок;

— описать используемые типы и структуры данных;

— описать структуру разрабатываемого программного обеспечения;

— реализовать разработанный алгоритм;

— провести функциональное тестирование разработанного алгоритма;

— провести сравнительный анализ по времени для реализованного алгоритма;

— подготовить отчет по лабораторной работе.

Многопоточность [1] — способность центрального процессора (CPU) или одного ядра в многоядерном процессоре выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой. Этот подход отличается от многопроцессорности, так как многопоточность процессов и потоков совместно использует ресурсы одного или нескольких ядер: вычислительных блоков, кэш-памяти центрального процессора (ЦП) или буфера перевода с преобразованием (TLB).

В тех случаях, когда многопроцессорные системы включают в себя несколько полных блоков обработки, многопоточность направлена на максимизацию использования ресурсов одного ядра, используя параллелизм на уровне потоков, а также на уровне инструкций. Поскольку эти два метода являются взаимодополняющими, их иногда объединяют в системах с несколькими многопоточными ЦП и в ЦП с несколькими многопоточными ядрами.

Ниже описаны достоинства и недостатки многопоточности.

Преимущества:

— использование общего адресного пространства программы для набора потоков;

— меньшие затраты на создание потока в сравнении с процессами;

— повышение производительности процесса за счёт распараллеливания процессорных вычислений;

— если теряется кэш, выделенный одному потоку, другие потоки могут продолжать использовать неиспользованные вычислительные ресурсы.

Недостатки:

— один поток может использовать адресное пространство другого потока при совместном использовании аппаратных ресурсов;

— с программной точки зрения, аппаратная поддержка многопоточности более трудоемка для программного обеспечения;

— проблема планирования потоков.

Целью данной работы является получить навык организации параллельных вычислений на примере алгоритма нахождения определителя матрицы через миноры.

Для достижения поставленной цели необходимо выполнить следующие задачи:

— описать основы многопоточного программирования;

— описать алгоритм нахождения определителя матрицы через миноры;

— разработать параллельный алгоритм нахождения определителя матрицы через миноры;

— реализовать оба алгоритма нахождения определителя матрицы через миноры;

— провести сравнительный анализ по времени на одинаковых матрицах и различном количестве потоков;

— провести сравнительный анализ затрат реализаций последовательного и параллельного алгоритмов по времени при разных размерах матриц с использованием многопоточности и без;

— описать и обосновать полученные результаты.

1 Аналитическая часть

В этом и последующих разделах будет рассмотрен алгоритм нахождения определителя матриц через миноры.

Матрица [2] — это набор чисел, записываемый в виде прямоугольной таблицы. Строки и столбцы матрицы можно считать векторами. Матрицы, у которых число строк равно числу столбцов, называют квадратными.

Обычно для обозначения элемента матрицы A, стоящего в *i-той* строке и в *j-ом* столбце используется следующая запись: A[i][j].

Самыми распространенными операциями над матрицами являются сложение, вычитание, транспонирование, умножение и нахождение определителя [2].

Способ вычисления определителя матрицы через миноры реализует формулу, описанную далее.

Минором Mij элемента aij матрицы A *n-го* порядка называется определитель $(n-1)$*-го* порядка, полученного из исходного определителя вычеркиванием *i-ой* строки и *j-го* столбца:

$$M_{ij} = \begin{pmatrix} a_{11} & \dots & a_{1,j-1} & a_{1,j+1} & \dots & a_{1,n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{i-1,1} & \dots & a_{i-1,j-1} & a_{i-1,j+1} & \dots & a_{i-1,n} \\ a_{i+1,1} & \dots & a_{i+1,j-1} & a_{i+1,j+1} & \dots & a_{i+1,n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{n1} & \dots & a_{n,j-1} & a_{n,j+1} & \dots & a_{n,n} \end{pmatrix}. \tag{1.1}$$

Замечание: определитель можно считать только для квадратных матриц, то есть тех матриц, у которых количество строк равно количеству столбцов.

Алгебраическим дополнением *Aij* элемента aij матрицы A *n-го* порядка называется число, равное произведению минора Mij на $(1)^{i+j}$:

$$Aij = (1)^{i+j} \cdot M_{ij}. \tag{1.2}$$

Определители *n-го* порядка вычисляются с помощью метода понижения порядка — по формуле $detA{=}\sum_{j=1}^{n} aij\,Aij$ (i фиксировано) — разложение по i-*ой* строке.

Метод приведения к треугольному виду заключается в преобразовании определителя, когда все элементы, лежащие по одну сторону главной диагонали рассматриваемой матрицы, становятся равными нулю. В этом случае определитель равен произведению элементов главной диагонали [3].

Вывод

В аналитической части был описан последовательный алгоритм поиска определителя матриц через миноры. Далее необходимо произвести оценку его эффективности и проверить её экспериментально.

2 Конструкторская часть

В данном разделе будут рассмотрены схемы алгоритмов нахождения определителя с распараллеливанием и без него.

2.1 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие типы данных:

— количество строк — целое число типа *int*;

— количество столбцов — целое число типа *int*;

— матрица — двумерный массив значений типа *int*.

2.2 Требования к программному обеспечению

Выдвинут ряд требований к программе:

— на вход подается матрица;

— выполняется проверка на предмет того, является ли матрица квадратной;

— если матрица квадратная, то на выходе необходимо получить определитель матрицы и время (в с), потраченное на его вычисление, в противном случае сообщить о том, что определитель не может быть вычислен.

2.3 Разработка алгоритмов

На рисунках 2.1–2.2 представлены схемы алгоритмов вычисления определителя матрицы без распараллеливания и с ним.

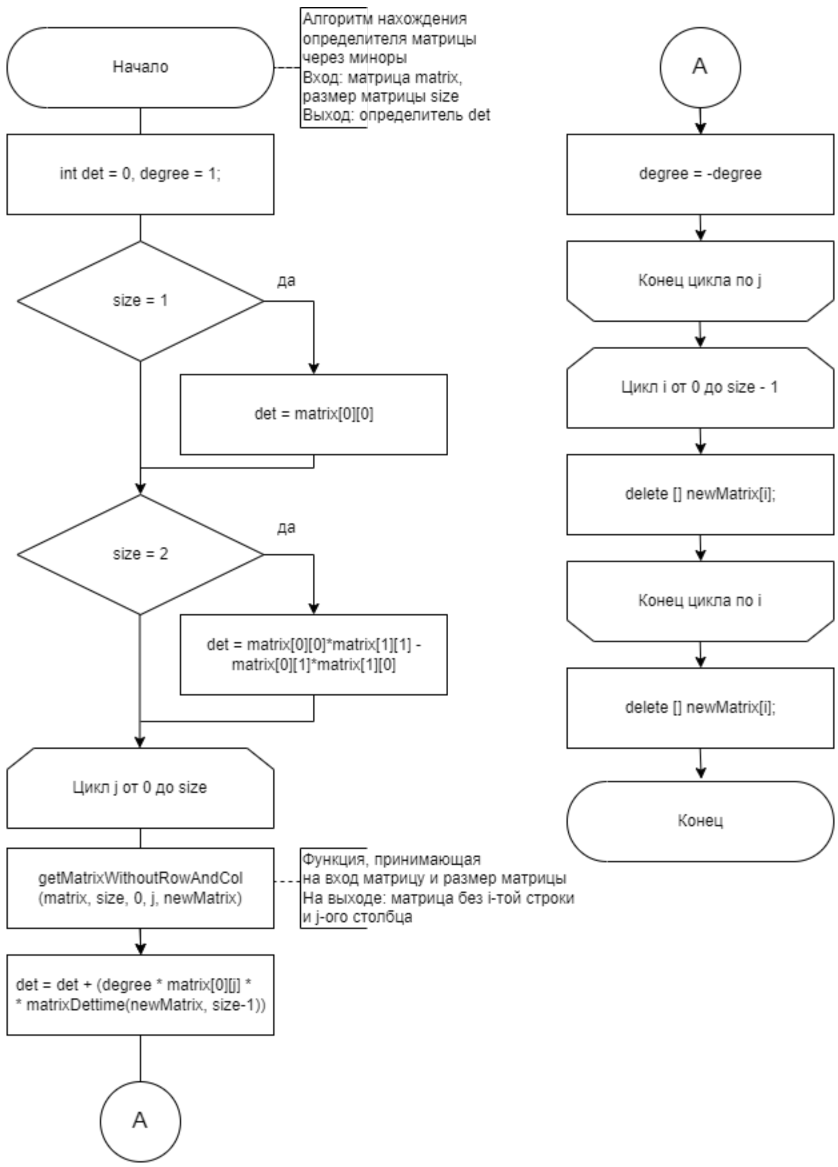

Рисунок 2.1 – Последовательный алгоритм поиска определителя матрицы

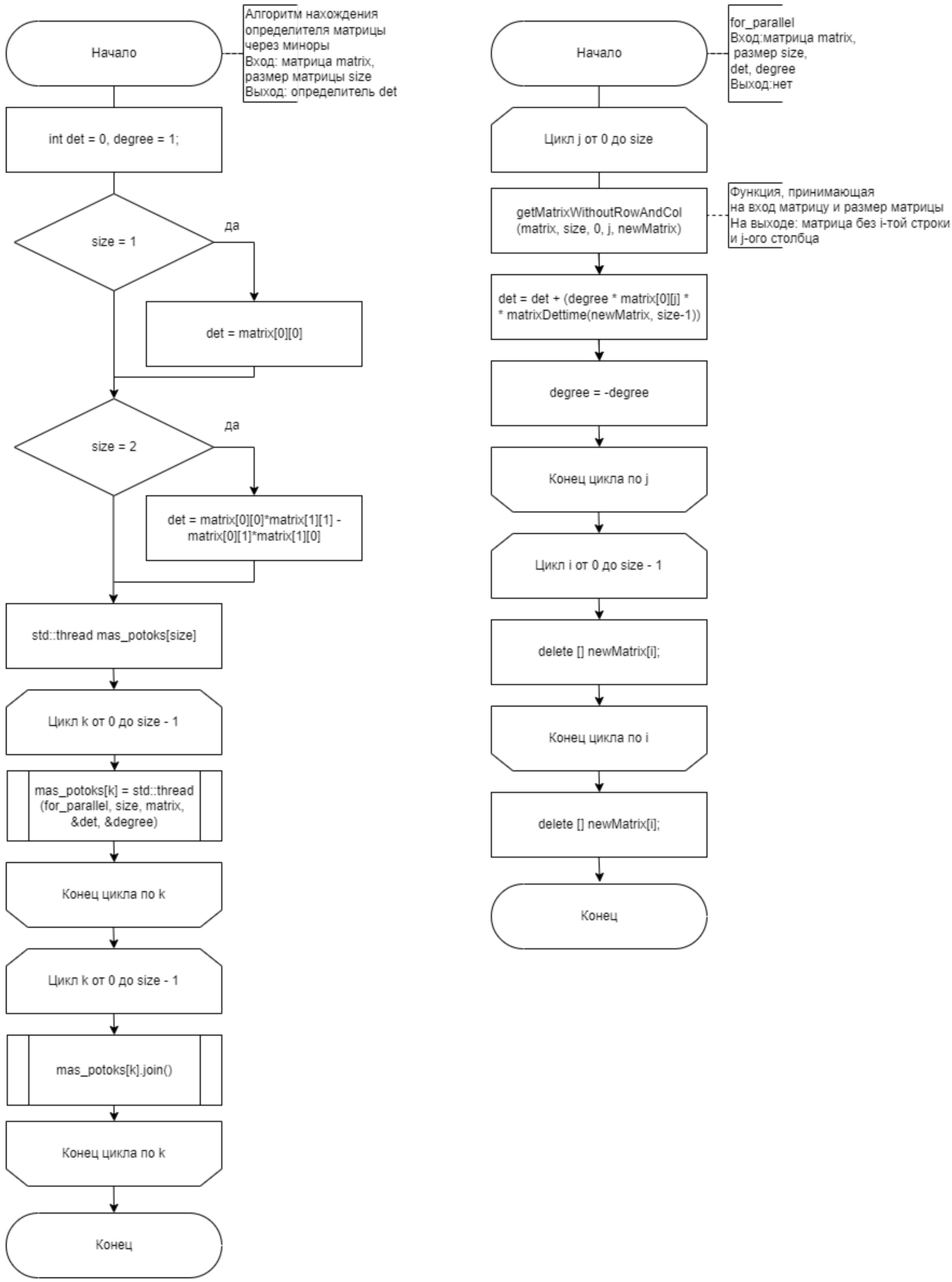

Рисунок 2.2 – Алгоритм поиска определителя матрицы с распараллеливанием

Вывод

В данном разделе были построены схемы алгоритма нахождения определителя матриц через миноры с распараллеливанием и без.

3 Технологическая часть

В данном разделе будут выбраны язык программирования, среда разработки, а также представлены листинги кода реализации алгоритмов поиска определителя.

3.1 Средства реализации

Для реализации алгоритмов был выбран язык программирования C++, в котором существует библиотека time.h [4] для замера процессорного времени. С помощью полученных результатов времени будут построены графики для наглядного отображения временной эффективности алгоритмов. Будет использована среда разработки CLion.

3.2 Сведения о модулях программы

Программа состоит из одного модуля — *main.c* — файл, содержащий код алгоритмов, создания и заполнения матрицы, вывода замеренного процессорного времени и весь второстепенный код.

В листингах 3.1, 3.2 и 3.3 представлены реализации функции удаления матрицы *i-ой* строки и *j-ого* столбца и алгоритма поиска определителя с получением замеров процессорного времени, затраченного на их выполнение.

Листинг 3.1 – Функция удаления *i-ой* строки и *j-ого* столбца

```
void getMatrixWithoutRowAndCol(int **matrix, int size, int row, int
    col, int **newMatrix)
{
    int offsetRow = 0;
    int offsetCol = 0;
    for(int i = 0; i < size-1; i++)
    {
        if(i == row)
        {
            offsetRow = 1;
        }
```

```
        offsetCol = 0;
        for(int j = 0; j < size-1; j++)
        {
            if(j == col)
            {
                offsetCol = 1;
            }
            newMatrix[i][j] = matrix[i + offsetRow][j + offsetCol];
        }
    }
}
```

Листинг 3.2 – Алгоритм нахождения определителя матрицы через миноры

```
int matrixDet(int **matrix, int size)
{
    time_t startt = 0, endt = 0;
    int det = 0;
    int degree = 1;
    startt = clock();

    if(size == 1)
    {
        return matrix[0][0];
    }
    else if(size == 2)
    {
        return matrix[0][0]*matrix[1][1] -
           matrix[0][1]*matrix[1][0];
    }
    else
    {
        int **newMatrix = new int*[size-1];
        for(int i = 0; i < size-1; i++)
        {
            newMatrix[i] = new int[size-1];
        }
        for(int j = 0; j < size; j++)
        {
            getMatrixWithoutRowAndCol(matrix, size, 0, j,
               newMatrix);
```

```
            det = det + (degree * matrix[0][j] *
                matrixDet(newMatrix, size-1));
            degree = -degree;
        }
        for(int i = 0; i < size-1; i++)
        {
            delete [] newMatrix[i];
        }
        delete [] newMatrix;
    }

    endt = clock();
    return endt - startt;
}
```

Листинг 3.3 – Функция вспомогательного потока

```
void for_parallel(int size, int **matrix, int *det, int *degree)
{
    int **newMatrix = new int*[size-1];
    for(int i = 0; i < size-1; i++)
    {
        newMatrix[i] = new int[size-1];
    }
    for(int j = 0; j < size; j++)
    {
        getMatrixWithoutRowAndCol(matrix, size, 0, j, newMatrix);
        *det = *det + (*degree * matrix[0][j] *
            matrixDet(newMatrix, size-1));
        *degree = -(*degree);
    }
    for(int i = 0; i < size-1; i++)
    {
        delete [] newMatrix[i];
    }
    delete [] newMatrix;
}

int matrixDetparallel(int **matrix, int size)
{
    clock_t startt, endt;
    int det = 0;
```

```
    int degree = 1;

    startt = clock();
    if(size == 1)
    {
        return matrix[0][0];
    }
    else if(size == 2)
    {
        return matrix[0][0]*matrix[1][1] -
            matrix[0][1]*matrix[1][0];
    }
    else
    {
        std::thread mas_potoks[size];
        for (int k = 0; k < size; k++)
        {
            mas_potoks[k] = std::thread(for_parallel, size, matrix,
                &det, &degree);
        }

        for (int k = 0; k < size; k++)
            mas_potoks[k].join();
    }

    endt = clock();
    return endt - startt;
}
```

3.3 Классы эквивалентности при тестировании

Для тестирования выделены следующие классы эквивалентности.

1. Пустая матрица.
2. Матрица не квадратная.
3. Квадратная матрица.

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для реализаций алгоритма поиска определителя матрицы.

Таблица 3.1 – Функциональные тесты

Входная матрица	Ожидаемый результат	Результат
””	матрица пустая	матрица пустая
$[1,2][3,4]$	-2	-2
$[1,2][3,4][5,6]$	матрица не квадратная	матрица не квадратная

Все тесты пройдены успешно реализациями последовательного алгоритма поиска определителя матрицы и алгоритма с распараллеливанием.

Вывод

В технологической части лабораторной работы были выбраны язык программирования и среда разработки, приведены листинги алгоритма, проведено функциональное тестирование, а также приведены технические характеристики устройства, на котором проводятся замеры времени работы алгоритма.

4 Исследовательская часть

В данном разделе лабораторной работы будут приведены примеры работы программы, а также выполнена сравнительная характеристика алгоритма с распараллеливанием и без.

4.1 Технические характеристики устройства

Ниже представлены характеристики компьютера, на котором проводилось тестирование программы:

- операционная система Windows 10 Домашняя 21H2;
- оперативная память 16 Гб;
- процессор Intel(R) Core(TM) i7-10870H CPU @ 2.20 ГГц.

Во время тестирования ноутбук был подключен к сети электропитания. Из программного обеспечения были запущены только среда разработки *PyCharm* и браузер *Chrome*.

Процессор был загружен на 19%, оперативная память – на 50%.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат программы по работе алгоритма с распараллеливанием. Выбран 1 пункт меню.

4.3 Время выполнения реализаций алгоритмов

В программе используется библиотека time.h, с помощью которой можно получить время, затраченное процессором на выполнение программы, представленное типом clock_t, или -1, если оно неизвестно. Размерность возвращаемого значения определяется при помощи CLOCKS_PER_SEC, константы, которая задаёт количество единиц значения времени в одной секунде.

```
Menu
1. Simple algorithm to find det with minors
2. Parallel algorithm to find det with minors
3. Time analysis
0. Exit
Input:1

input number of cols and rows of matrix 1:3 3
1 2 3
10 -9 78
1 1 1

1 2 3
10 -9 78
1 1 1
det = 106
```

Рисунок 4.1 – Пример работы программы

Использовать функцию приходится дважды, в первый раз перед началом алгоритма, второй — после, а затем из конечного времени необходимо вычесть начальное, чтобы получить потраченное на алгоритм время.

Результаты замеров времени работы реализаций методов умножения на различных входных данных (в с) приведены в таблице 4.1. Замеры для одного и того же алгоритма с одними и теми же входными данными производились 10 раз для получения усредненного результата. Для параллельной реализации алгоритма выбрано количество потоков, равное 4, поскольку именно столько логических ядер имеет процессор ноутбука.

Таблица 4.1 – Результаты замеров времени

Размеры сцены	4 потока	без многопоточности
1x1	0.0006	5
10x10	0.001515	74
20x20	0.002379	141
30x30	0.003276	148
40x40	0.004691	164
50x50	0.004543	184

Теоретические результаты замеров и полученные практически результаты совпадают.

Также на рисунках 4.2 и 4.3 приведены графические результаты замеров работы алгоритмы на квадратных матрицах в зависимости от их размера и количества потоков. Введены следующие обозначения: *nm* — реализация алгоритма без многопоточности, *m4* — реализация алгоритма с распараллеливанием на 4 потока, *m* — реализация алгоритма с распараллеливанием на различное количество потоков.

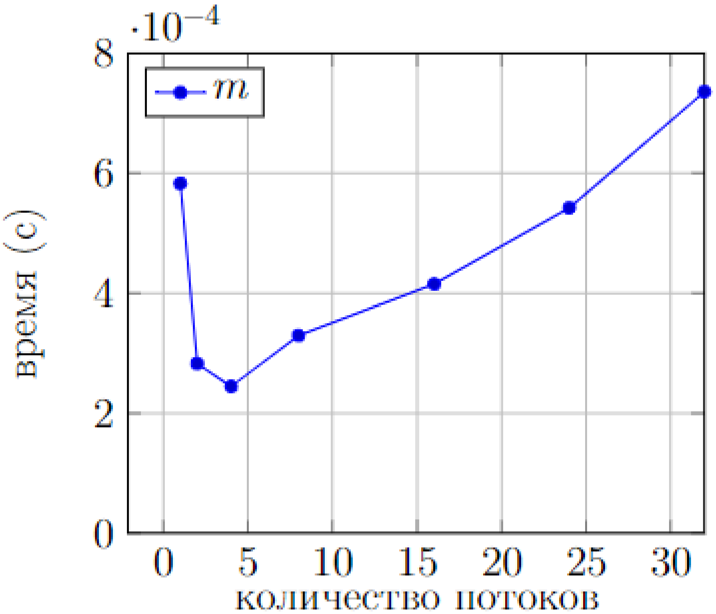

Рисунок 4.2 – Сравнение времени работы алгоритма с распараллеливанием на различное количество потоков при работе с матрицами размером 500х500

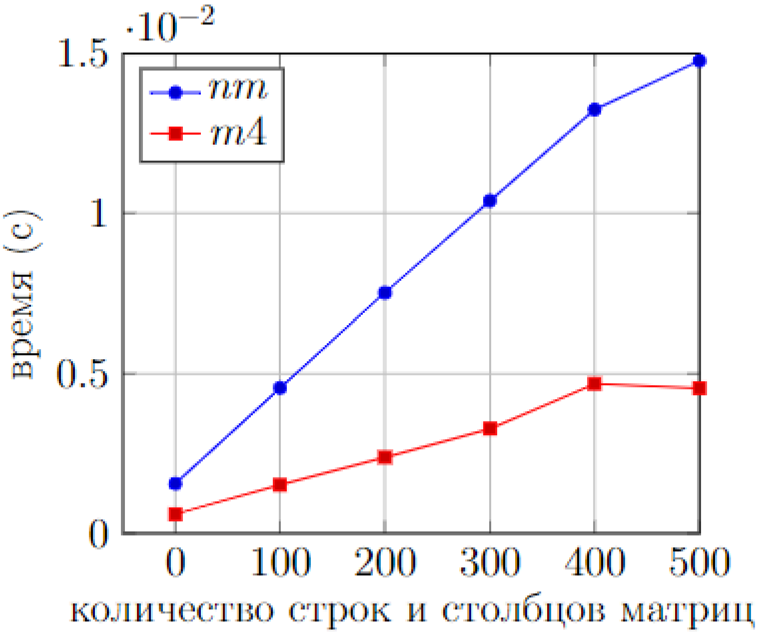

Рисунок 4.3 – Сравнение времени работы алгоритма без распараллеливания и с 4 вспомогательными потоками при работе с квадратными матрицами

Вывод

По графикам видно, что при использовании 4 вспомогательных потоков, многопоточная реализация алгоритма значительно эффективнее по времени реализации без многопоточности при работе с матрицами размером 500x500. Данное количество потоков обусловлено тем, что на ноутбуке, на котором проводились замеры времени ПО, имеется всего 4 логических ядра, а следовательно, количество потоков, при котором потоки будут распределены между всеми ядрами равномерно, равно 4. Именно поэтому лучшие результаты достигаются именно на 4 потоках, даже несмотря на ресурсы, которые дополнительно затрачиваются на содержание потоков. Исходя из построенных графиков, можно сделать вывод, что распараллеливание кода значительно увеличивает эффективность алгоритма поиска определителя матрицы по времени.

Заключение

В ходе лабораторной работы поставленная ранее цель была достигнута: были получены навыки организации параллельных вычислений на примере алгоритма нахождения определителя матрицы через миноры. Выяснилось, что при работе с матрицами размерами до 500x500 элементов наиболее рационально использовать именно многопоточную реализацию алгоритма. Также в ходе выполнения лабораторной работы были решены следующие задачи:

— описаны основы многопоточного программирования;

— описан алгоритм нахождения определителя матрицы через миноры;

— разработан параллельный алгоритм нахождения определителя матрицы через миноры;

— реализованы оба алгоритма нахождения определителя матрицы через миноры;

— проведен сравнительный анализ по времени на одинаковых матрицах и различном количестве потоков;

— проведен сравнительный анализ затрат по времени при разных размерах матриц с использованием многопоточности и без;

— описаны и обоснованы полученные результаты.

Список использованных источников

1. Многопоточность [Электронный ресурс]. — URL: https://ru.bmstu.wiki (дата обращения: 10.10.2022)

2. Матрицы. Понятие. Применение [Электронный ресурс]. — URL: https://pro-prof.com/forums/topic/matrix_definition_using (дата обращения: 10.10.2022)

3. Вычисление определителей 2-го и 3-го порядков. Миноры, алгебраические дополнения. [Электронный ресурс]. — URL: http://mathportal.net/index.php/linejnaya-algebra/vychislenie-opredelitelej-minori-algebraicheskie-dopolneniya-rang (дата обращения: 10.10.2022)

4. Заголовочный файл ctime (time.h) [Электронный ресурс]. — URL: http://cppstudio.com/cat/309/326/ (дата обращения: 10.10.2022)