



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 алгоритмы

Студент Прянишников А. Н.

Группа ИУ7-55Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Содержание

Введение	2
1 Аналитическая часть	3
1.1 Корреляция Пирсона	3
1.2 Формализация задачи	4
1.3 Алгоритм поиска столбцов с максимальной корреляцией . .	4
1.4 Распараллеливание основного алгоритма	5
1.5 Вывод	6
2 Конструкторская часть	7
2.1 Требования к ПО	7
2.2 Функциональная модель	7
2.3 Схемы алгоритмов	9
2.4 Типы данных для алгоритмов	11
2.5 Способ тестирования	11
2.6 Вывод	12
3 Технологическая часть	13
3.1 Средства реализации программного обеспечения	13
3.2 Листинг кода	13
3.3 Функциональные тесты	16
3.4 Вывод	17
4 Исследовательская часть	18
4.1 Демонстрация работы программы	18
4.2 Технические характеристики	18
4.3 Тестирование программы	19
4.4 Вывод	20
5 Заключение	22
Список литературы	23

Введение

В машинном обучении и аналитике данных часто встаёт задача нахождения взаимосвязи между двумя переменными.[1] Для этого используется такая метрика как корреляция Пирсона.

Актуальность работы заключается в том, нахождение этого значения тратит много системных ресурсов, в первую очередь времени. Для сокращения временных затрат можно применить параллельные вычисления.

Целью данной работы является разработка программы, которая реализует два способа нахождения двух самых коррелирующих признаков в признаковом пространстве: классический, и с использованием потоков.

Для достижения поставленной цели необходимо выполнить следующее:

- рассмотреть алгоритм нахождения корреляции Пирсона;
- рассмотреть способы распараллеливания алгоритма;
- привести схемы реализации рассматриваемых алгоритмов;
- определить средства программной реализации;
- реализовать рассматриваемые алгоритмы;
- протестировать разработанное ПО;
- провести модульное тестирование всех реализаций алгоритмов;
- оценить реализацию алгоритмов по времени и памяти.

1 Аналитическая часть

В этом разделе будут проведена формализация задачи, рассмотрен алгоритм нахождения корреляции Пирсона, приведены возможности для распараллеливания основного алгоритма.

1.1 Корреляция Пирсона

Корреляция – статистическая связь двух и более случайных величин [2]. При этом изменение значений одной или нескольких из этих величин сопутствуют систематическому изменению значений другой или других величин. Сама корреляция не говорит о том, что между случайными величинами есть связь, она лишь говорит, что между ними есть статистическая связь. В качестве одного из видов корреляции используется корреляция Пирсона.

Коэффициент корреляции Пирсона характеризует существование линейной зависимости между двумя величинами. Пусть даны две выборки $X^m = (x_1, x_2, \dots, x_m)$ и $Y^m = (y_1, y_2, \dots, y_m)$. Тогда коэффициент корреляции Пирсона рассчитывается по формуле 1.1:

$$r_{xy} = \frac{cov(X, Y)}{D(X) * D(Y)} \quad (1.1)$$

где $D(X)$ – дисперсия случайной величины, $cov(X, Y)$ – ковариация случайной величины.

Дисперсия считается по формуле 1.2:

$$D(X) = \sum_{i=1}^M (x_i - \bar{x})^2 \quad (1.2)$$

Ковариация случайных величин X и Y находится по следующему соотношению 1.3:

$$cov(X, Y) = \mathbf{E}((X - \mathbf{E}(X)) * (Y - \mathbf{E}(Y))) \quad (1.3)$$

где $\mathbf{E}(X)$ – математическое ожидание случайной величины X .

Коэффициент корреляции Пирсона может принимать значения от -1 до 1. Если $|r|$ близок к 1, это значит, что между двумя случайными величинами отслеживается статистическая линейная связь.

По условию задачи нужно найти два признака с максимальной корреляцией, поэтому учитываться будет только модуль корреляции.

Чтобы применять корреляцию Пирсона, должно выполняться несколько требований:

1. Исследуемые переменные X и Y должны быть распределены нормально.
2. Исследуемые переменные X и Y должны быть измерены в интервальной шкале или шкале отношений.
3. Количество значений в исследуемых переменных X и Y должно быть одинаковым.

1.2 Формализация задачи

Реализуемое ПО должно найти два признака с максимальной корреляцией. Пусть $L^m(X_1, X_2, X_3, \dots, X_m)$ – признаковое пространство. Тогда результатом работы программы будет следующее представление 1.4:

$$i, j : X_i, X_j = \operatorname{argmax}(|r_L|) \quad (1.4)$$

1.3 Алгоритм поиска столбцов с максимальной корреляцией

Основной алгоритм состоит из нескольких этапов:

1. Подсчёт математического ожидания для каждого признака.
2. Нахождение корреляции для каждой пары признаков, отталкиваясь от известного математического ожидания.

3. Поиск максимального значения коэффициента корреляции.

Математическое ожидание для каждого признака производится через цикл по всем строкам. Результаты заносятся в массив размером с количеством признаков.

Затем через всю матрицу значений производится проход по всем парам признаков, и рассчитывается корреляция, используя формулы 1.1, 1.2 и 1.3.

Полученное на каждой итерации значение сравнивается с максимальным на текущий момент. Сохраняются также и индексы признаков, коэффициент корреляции Пирсона для которых постоянен.

1.4 Распараллеливание основного алгоритма

В основном алгоритме есть возможность внедрить параллельность для каждого этапа:

- в этапе подсчёта среднего арифметического можно подсчитывать значение одновременно для каждого признака, так как здесь используются только значения из конкретного столбца;
- в этапе подсчёта корреляции можно также подсчитывать одновременно коэффициент для нескольких пар признаков.

Так как на втором этапе постоянно нужно обновлять максимум, то здесь последовательность действий для каждого потока другая:

1. Поток подсчитывает максимальное значение и соответствующие индексы в своём пространстве. Все вычисления происходят внутри функции.
2. Поток записывает получившийся результат в глобальную для потоков переменную, если его результат больше, чем значение этой переменной.

На последнем этапе возможна гонка потоков, поэтому могут возникнуть некоторые ошибки. Чтобы их избежать, должен быть использован мьютекс, который блокирует доступ к глобальным для потоков переменным на время записи туда нового значения. Мьютекс нужно запускать ещё до сравнения, иначе возможны ошибки.

1.5 Вывод

Были произведена формализация задачи, рассмотрен основной алгоритм нахождения корреляции Пирсона, а также обнаружены возможности для распараллеливания основного алгоритма.

2 Конструкторская часть

В этом разделе будут приведены требования к ПО, схемы реализации алгоритмов, а также выбранные классы эквивалентности для тестирования ПО.

2.1 Требования к ПО

Ниже будет представлен список требований к разрабатываемому программному обеспечению.

Требования к входным данным:

- на вход подаётся матрица, состоящая из вещественных чисел;
- в матрице как минимум два столбца и две строки.

Требования к выводу:

- программа должна вывести индексы самых коррелирующих между собой столбцов, а также значение коэффициента корреляции Пирсона для неё.

2.2 Функциональная модель

На рисунке 2.1 будет представлена функциональная модель IDEF0 уровня 1.



Рисунок. 2.1: Схема классического алгоритма поиска наиболее коррелирующих признаков

2.3 Схемы алгоритмов

На рисунке 2.2 будет приведена схема реализации классического алгоритма поиска максимальной корреляции в признаковом пространстве. На рисунке 2.3 будет приведена схема реализации алгоритма поиска максимальной корреляции в признаковом пространстве с использованием параллельности.

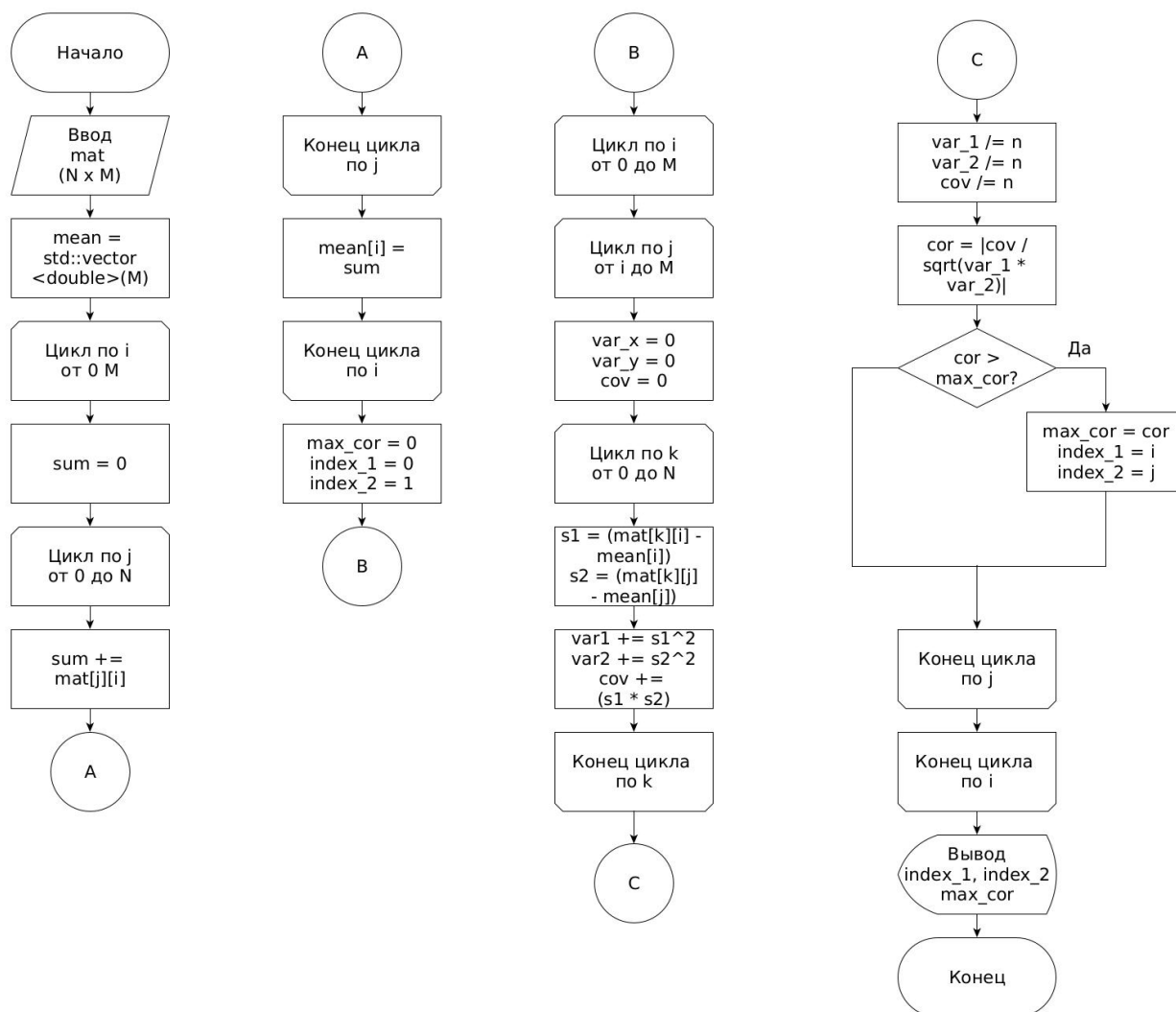


Рисунок. 2.2: Схема классического алгоритма поиска наиболее коррелирующих признаков

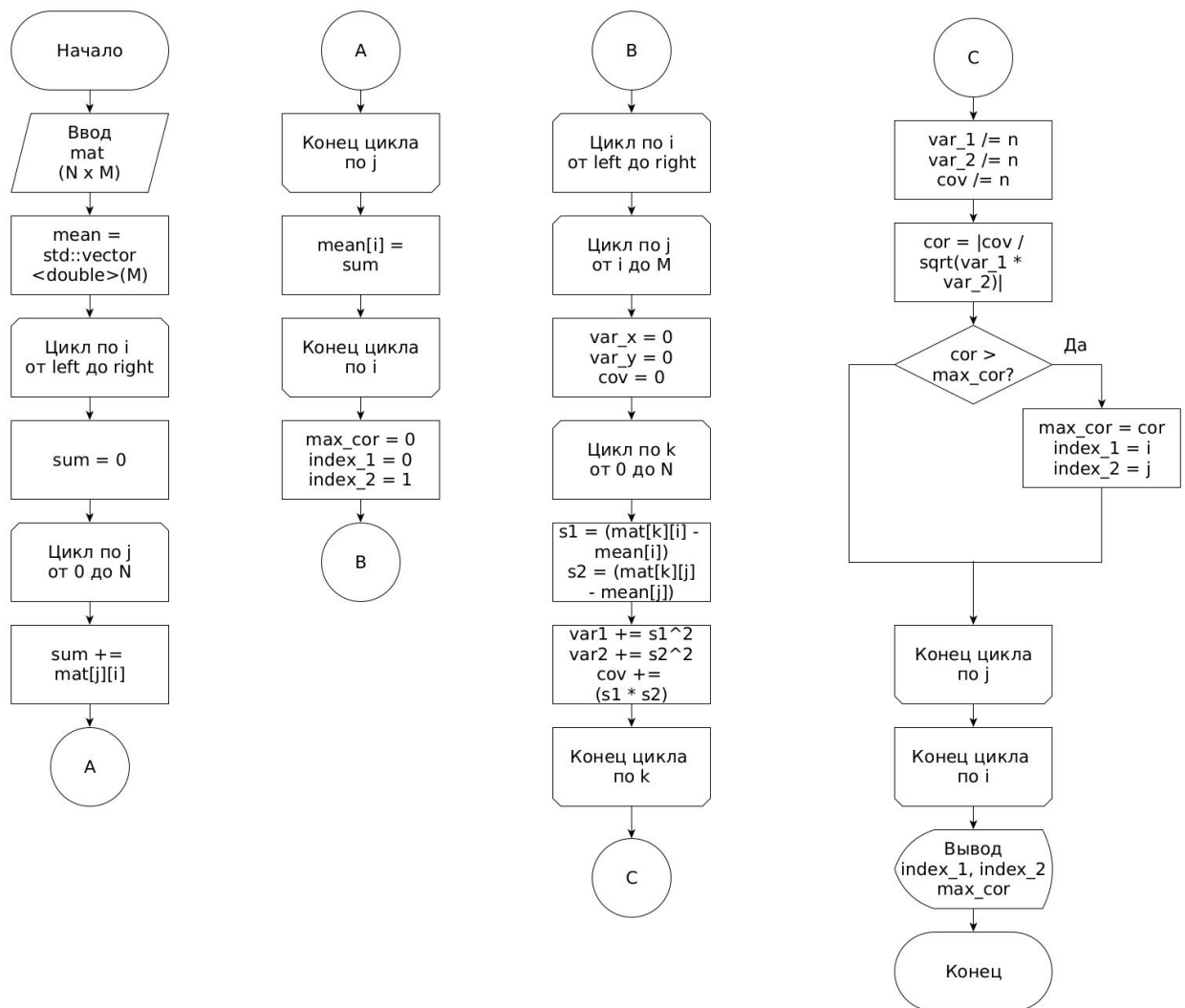


Рисунок. 2.3: Схема алгоритма поиска наиболее коррелирующих признаков с использованием параллельности

Распараллеливание вычислений реализовано благодаря добавлению двух новых переменных `left` и `right`, которые указывают на диапазон строк, которые необходимо рассчитать.

2.4 Типы данных для алгоритмов

Тестирование алгоритмов будет производиться на вещественных числах, которые могут быть и отрицательными, и равны нулю. Несмотря на это, сами реализации алгоритмов универсальны и предназначены для любых численных типов данных.

Размер матрицы может быть произвольным из тех, что допустимы по требованиям ко вводу.

2.5 Способ тестирования

Тестирование программы будет производиться методом чёрного ящика. Такой подход выбран, так как от реализаций алгоритмов требуется в первую очередь правильность работы. Сама по себе реализация не требует тестировки, так как в точности повторяет теоретические принципы, сформированные в аналитическом разделе.

В качестве классов эквивалентности были выбраны следующие сущности:

- матрица состоит из двух столбцов;
- матрица состоит из двух строк и нескольких столбцов;
- между двумя столбцами матрицы прослеживается явная линейная зависимость;
- между двумя столбцами матрицы прослеживается явная отрицательная линейная зависимость;
- матрица состоит из случайных значений;

Для корректного сравнения требуется для каждой реализации умножения матриц сравнить полученный на выходе результат с эталонным. Для этого будет использоваться библиотека `numpy`, то есть предварительно эталонное значение для каждой матрицы будет рассчитано.

2.6 Проектирование работы потоков

В разрабатываемом ПО предполагается, что один из потоков является основным. Он будет раздавать задания остальным потоком, после этого ждать выполнения каждым потоком задачи. Так как параллельность выполняется в двух местах, то основной поток будет раздавать задания два раза.

На рисунке 2.4 представлена схематичная временная диаграмма работы потоков программы на примере 8-поточного алгоритма.

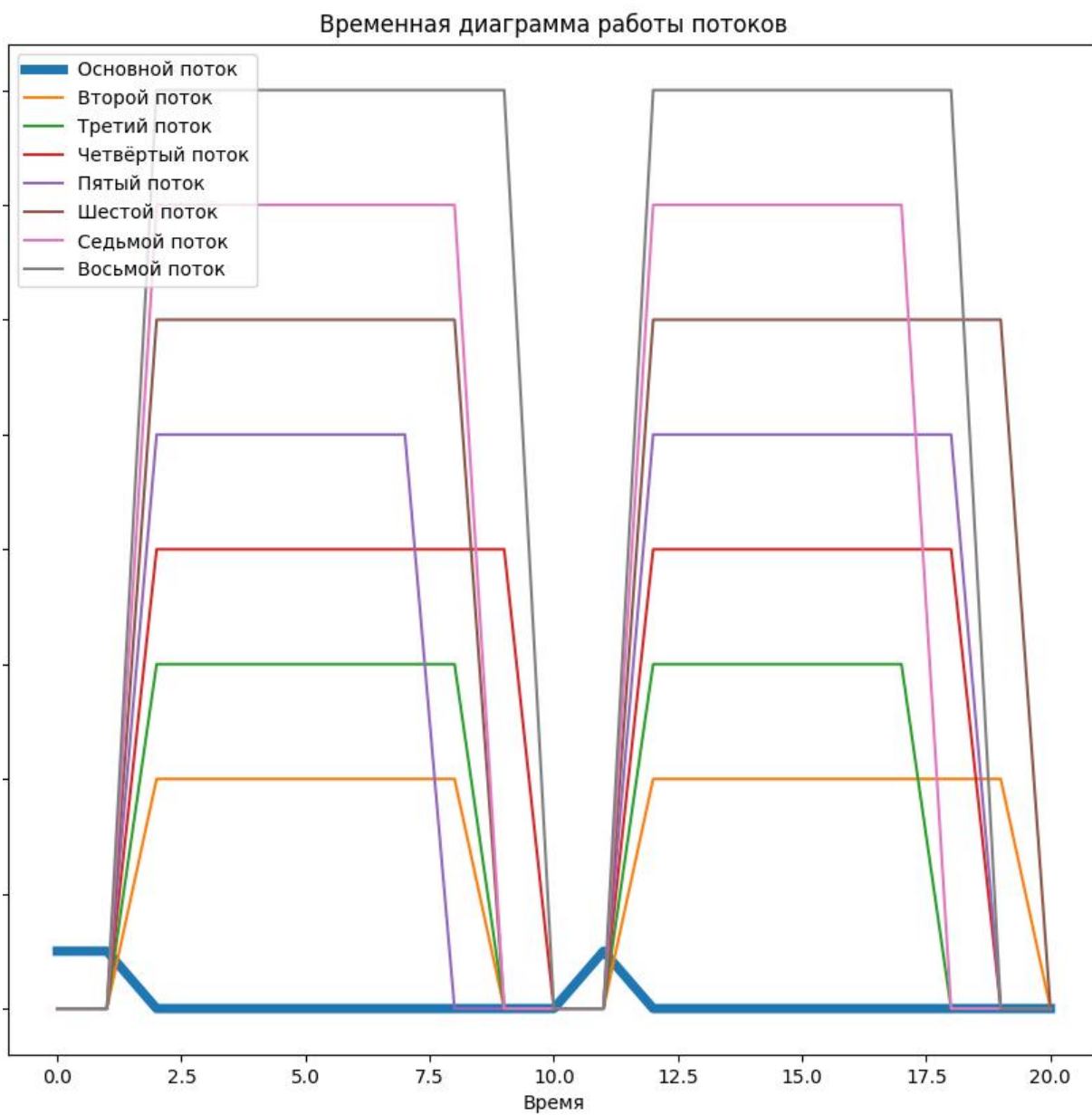


Рисунок. 2.4: Временная диаграмма работы потоков программы

Из этого можно сделать вывод, что в случае работы 2-поточного алгоритма, один поток будет основным, а ещё один поток будет выполнять один все действия, поэтому время работы будет схоже с временем работы классического алгоритма.

Также для доступа к переменной глобального максимума будет использоваться мьютекс.

2.7 Вывод

Были приведены требования к ПО, схемы реализации алгоритмов. Были определен способ тестирования алгоритмов.

3 Технологическая часть

В этом разделе будут приведены листинги кода и результаты функционального тестирования.

3.1 Средства реализации программного обеспечения

В качестве языка программирования выбран C++, так как имеется опыт разработки проектов на этом языке, а также есть возможность использования нативных потоков [3]. Для замеры общего времени используется библиотека chrono [4].

3.2 Листинг кода

На листинге 4.1 представлен код работы классического алгоритма.

На листинге 4.2 представлен код работы подсчёта среднего арифметического для каждого потока.

На листинге 4.3 представлен код работы подсчёта корреляции Пирсона для каждого потока.

На листинге 4.4 представлен общий код работы алгоритма с использованием параллельных вычислений.

Листинг 3.1: Реализация классического алгоритма

```
void find_most_corr(std::vector<std::vector<double>> matrix, int n, int m)
{
    std::vector<double> mean(m, 0);
    for (int i = 0; i < m; i++)
    {
        double sum = 0;
        for (int j = 0; j < n; j++)
            sum += matrix[j][i];
        mean[i] = (sum / n);
    }
    int idx_1 = 0;
    int idx_2 = 0;
```



```

double max_cor = 0;
for (int i = 0; i < m; i++)
{
    for (int j = i + 1; j < m; j++)
    {
        double var_x = 0;
        double var_y = 0;
        double cov = 0;
        for (int k = 0; k < n; k++)
        {
            double slag_1 = (matrix[k][i] - mean[i]);
            double slag_2 = (matrix[k][j] - mean[j]);
            var_x += (slag_1 * slag_1);
            var_y += (slag_2 * slag_2);
            cov += (slag_1 * slag_2);
        }
        var_x /= n;
        var_y /= n;
        cov /= n;
        double cor = fabs(cov / sqrt(var_x * var_y));
        if (cor > max_cor)
        {
            idx_1 = i;
            idx_2 = j;
            max_cor = cor;
        }
    }
}

```

Листинг 3.2: Реализация схемы подсчёта среднего арифметического для каждого потока

```

void scheme_one_for_thread(std::vector<std::vector<double>> matrix, std::vector<double> &mean,
                           int start_index, int end_index, int n)
{
    for (int i = start_index; i < end_index; i++)
    {
        float sum = 0;
        for (int j = 0; j < n; j++)
            sum += matrix[j][i];
        mean[i] = (sum / n);
    }
}

```

Листинг 3.3: Реализация схемы подсчёта корреляции Пирсона

```

void scheme_two_for_thread(std::vector<std::vector<double>> matrix, std::vector<double> mean,

```

```

        int start_index, int end_index, int n, int m,
        volatile double &max_cor,
        volatile double &idx_1, volatile double &idx_2)
{
    int local_idx_1 = 0;
    int local_idx_2 = 0;
    double local_max_cor = 0;
    for (int i = start_index; i < end_index; i++)
    {
        for (int j = i + 1; j < m; j++)
        {
            double var_x = 0;
            double var_y = 0;
            double cov = 0;
            for (int k = 0; k < n; k++)
            {
                double slag_1 = (matrix[k][i] - mean[i]);
                double slag_2 = (matrix[k][j] - mean[j]);
                var_x += (slag_1 * slag_1);
                var_y += (slag_2 * slag_2);
                cov += (slag_1 * slag_2);
            }
            var_x /= n;
            var_y /= n;
            cov /= n;
            double cor = fabs(cov / sqrt(var_x * var_y));
            if (cor > local_max_cor)
            {
                local_idx_1 = i;
                local_idx_2 = j;
                local_max_cor = cor;
            }
        }
    }
    write_max(local_max_cor, local_idx_1, local_idx_2, max_cor, idx_1,
        idx_2);
}

```

Листинг 3.4: Общая реализация алгоритма с использованием потоков

```

void find_most_corr_with_threads(std::vector<std::vector<double>> matrix,
    int n, int m, int num_threads)
{
    if (num_threads > m)
    {
        std::cout << "Max_enough_threads:" << m << std::endl;
        num_threads = m;
    }
    int koef = m / num_threads;
    int ost = m % num_threads;
}

```

```

auto* threads = new std::thread[num_threads];
std::vector<double> mean(m, 0);
for (int i = 0; i < num_threads; i++)
{
    int start_index = (i * koef);
    int end_index = ((i + 1) * koef);
    if (i == num_threads - 1)
        end_index += ost;
    threads[i] = std::thread(scheme_one_for_thread, matrix, std::ref(
        mean), start_index, end_index, n);
}
for (int i = 0; i < num_threads; i++)
{
    threads[i].join();
}
auto* threads_1 = new std::thread[num_threads];
volatile double max_cor = 0;
volatile double idx_1 = 0;
volatile double idx_2 = 0;

for (int i = 0; i < num_threads; i++)
{
    int start_index = i * koef;
    int end_index = (i + 1) * koef;
    if (i == num_threads - 1)
        end_index += ost;
    threads_1[i] = std::thread(scheme_two_for_thread, matrix, mean,
        start_index, end_index, n, m,
        std::ref(max_cor), std::ref(idx_1), std::
            ref(idx_2));
}
for (int i = 0; i < num_threads; i++)
{
    threads_1[i].join();
}

```

3.3 Функциональные тесты

В таблице 3.1 приведены результаты функциональных тестов. В первом столбце – исходная матрица. Во втором столбце – ожидаемый результат: наиболее коррелирующих признаков и само значение корреляции.

Таблица 3.1: Функциональные тесты

Матрица	Ожидаемый результат
1 0.5 2 15 3 9 3 11 12	2, 3, 0.867502
1 0.5 2 2 2 3 9 4 3 11 12 6 4 5 3 8	1, 4, 1
1 0.5 2 3 3 11 5 6	1, 2, 0.567236
1 0.5 2 3	1, 2, 1
5 9 1 9 2 1 3 8 5 0 5 5 6 0 6 7 8 0 1 5 8 9 1 7 6 2 3 3 3 9	1, 2, 0.879347

Все тесты алгоритмами были пройдены успешно.

3.4 Вывод

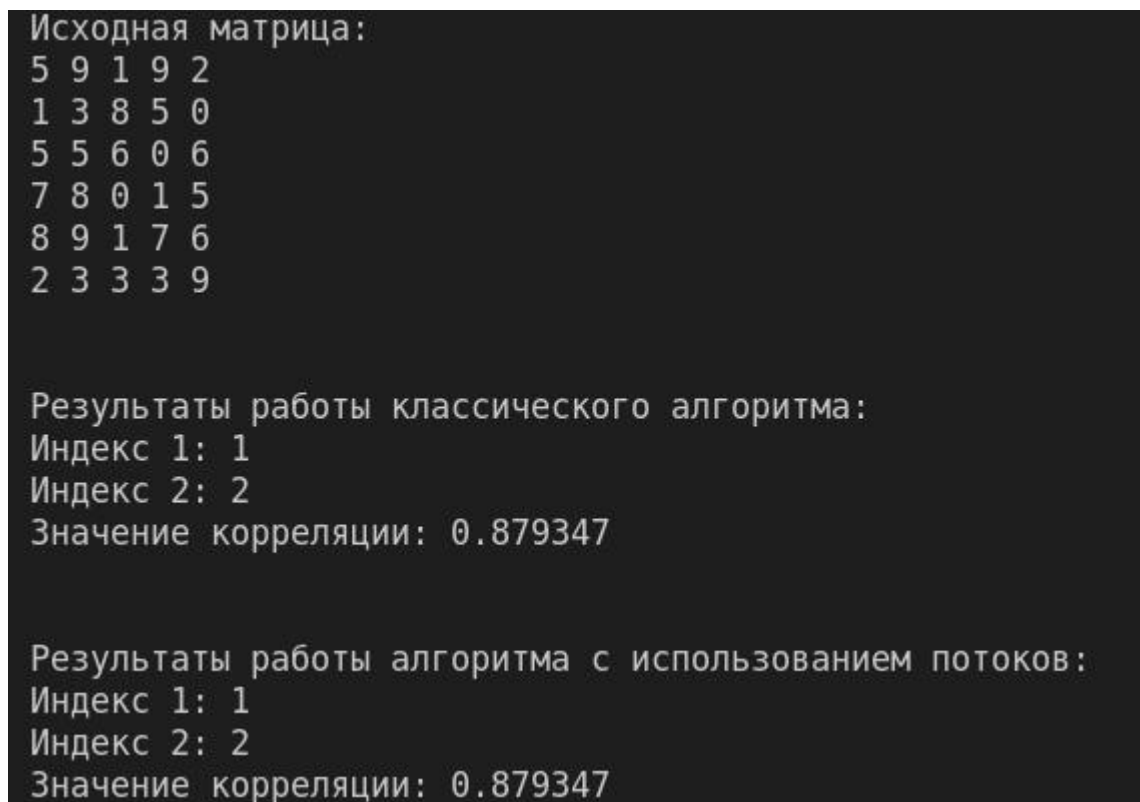
Были сформированы требования к ПО, приведены листинги кода. Были проведены функциональные тесты. Все алгоритмы справились с тестированием.

4 Исследовательская часть

В этом разделе будет продемонстрирована работа программы, а также приведены результаты тестирования алгоритмов.

4.1 Демонстрация работы программы

На рисунке 4.1 приведена демонстрация работы программы.



```
Исходная матрица:
5 9 1 9 2
1 3 8 5 0
5 5 6 0 6
7 8 0 1 5
8 9 1 7 6
2 3 3 3 9

Результаты работы классического алгоритма:
Индекс 1: 1
Индекс 2: 2
Значение корреляции: 0.879347

Результаты работы алгоритма с использованием потоков:
Индекс 1: 1
Индекс 2: 2
Значение корреляции: 0.879347
```

Рисунок. 4.1: Демонстрация работы программы

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование, следующие:

- операционная система: Ubuntu 20.04.1 LTS;
- память: 8 GB;

- процессор: Intel Core i5-1135G7 @ 2.40GHz [5].
- количество ядер процессора: 8

Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.3 Тестирование программы

Для определения быстродействия работы алгоритмов будет проведено исследование зависимости на квадратных матрицах, так как её размер однозначно определяется по одной переменной.

В таблице 4.1 представлены результаты тестирования времени работы алгоритма в зависимости от количества потоков и размера матрицы. Время - в микросекундах.

Таблица 4.1: Результаты тестов (с учётом того, что один поток - главный)

Размер массива	Классический	2 потока	4 потока	8 потоков	16 потоков
2	1	44	79	219	226
5	4	52	103	294	287
10	11	81	157	324	835
25	104	240	225	444	892
50	673	1014	690	741	1162
75	2181	2811	2025	1676	2195
100	5242	6575	4328	3491	4291
200	45820	51614	32406	22376	21584
500	777738	798849	491472	378796	316299
750	3765498	4036732	2285193	1535681	1419432
1000	7883329	8551277	4943219	3264990	3070650

На рисунке 4.1 показан график зависимости времени работы алгоритмов от размера матрицы и количества потоков.

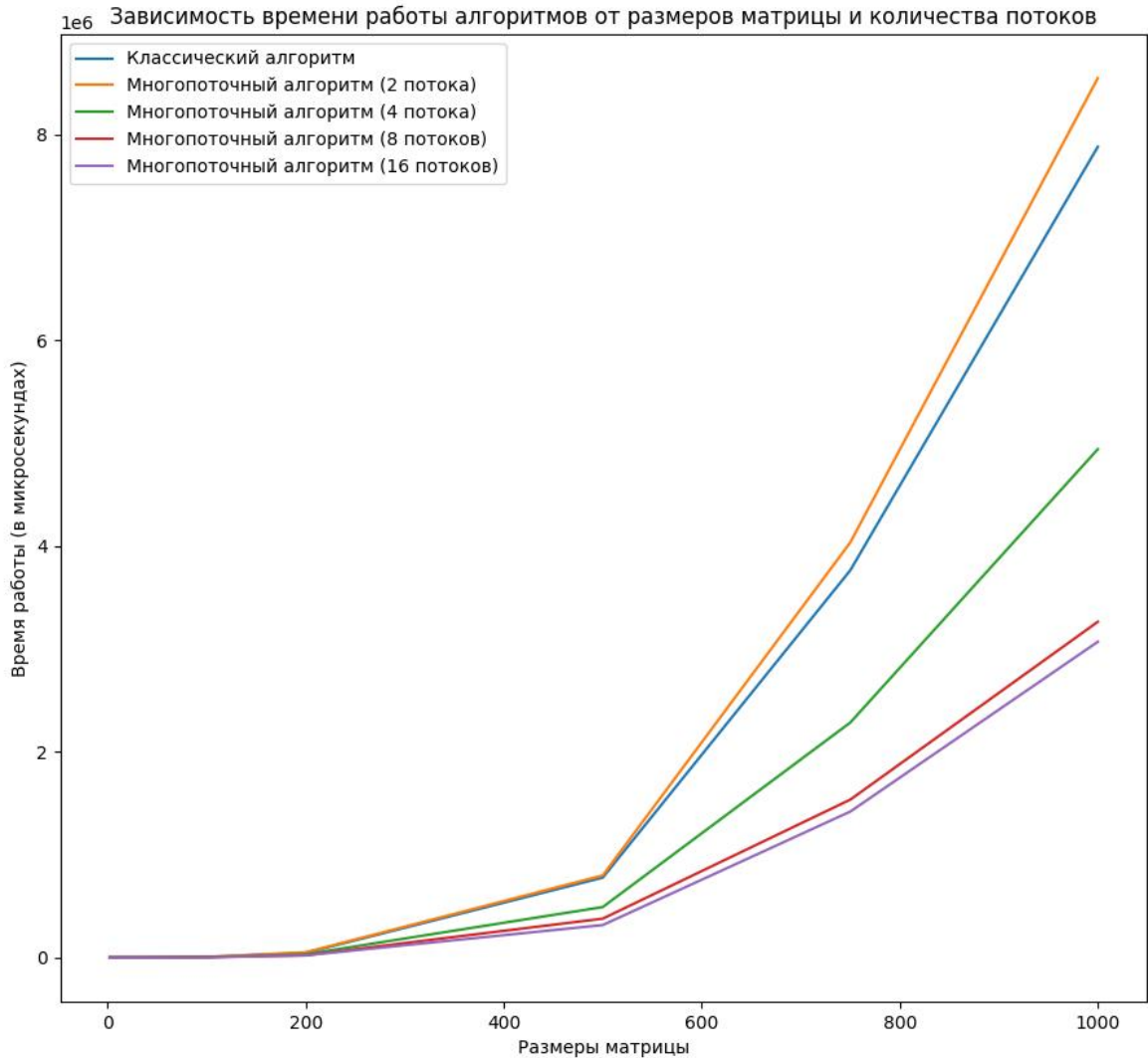


Рисунок. 4.2: График зависимости времени работы алгоритмов от размера матрицы и количества потоков

На 16 потоках программа показала наилучшие результаты.

4.4 Вывод

Распараллеливание программы действительно привело к ускорению работы программы.

На матрице размером $N = 1000$ 16-поточный алгоритм показал наилучшие результаты, опередив на 20% 8-поточный алгоритм, в полтора раза алгоритм с $M = 4$ потоками, и в 2.5 раза - классический и двухпоточный

алгоритмы. Такие же пропорции наблюдаются при $N = 500$, $N = 200$. При $N = 100$ 8-поточный алгоритм занял первое место.

На матрице размером $N \leq 50$ классический алгоритм показал наилучшие результаты. Это связано с тем, что поддержка потоков также требует дополнительных системных ресурсов, и для маленьких матриц это оказалось неэффективным. Но при матрицах $N > 50$ многопоточные алгоритмы уже оправдывают ожидания.

Алгоритм с двумя потоками оказался самым медленным при $N \geq 50$. Это объясняется тем, что в реализации один из потоков - главный, и он лишь ждёт выполнения работы другим потоком. То есть это был такой же классический алгоритм, но при этом он поддерживал многопоточность, то есть системные затраты были выше.

Увеличения числа потоков в два раза не привело к уменьшению работы программы в два раза. Для $M_1 = 2$ и $M_2 = 4$ в среднем четырёхпоточный показал время в 1.8 раз меньше. Для $M_1 = 4$ и $M_2 = 8$ разница составила 50 %. Такого ускорения не было, потому что потоки чаще обращаются к сравнению глобальной переменной, и мьютекс блокирует доступ к переменным. Более того, у самих потоков становится меньше системных ресурсов, поэтому выполнение потока по отдельности происходит медленнее.

В случае $M_1 = 8$ и $M_2 = 16$ вообще не наблюдается выигрыша до $N = 200$, а затем 16-поточный алгоритм стал работать быстрее на 20%. То есть можно сделать вывод, что создание потоков внутри одного ядра тоже даёт выигрыш.

Затраты на память у обоих алгоритмов имеют один и тот же порядок $O(N)$.

5 Заключение

В ходе работы были рассмотрены два алгоритма поиска наиболее коррелирующих столбцов: классический алгоритм и многопоточный. Были составлены схемы алгоритмов. Было реализовано работоспособное ПО, удалось провести анализ зависимости затрат по времени от размера матрицы и количества потоков.

Многопоточный алгоритм показал более высокие результаты, начиная с $N = 50$. 16-поточный алгоритм показал наилучшие результаты на матрицах размером $N \geq 200$. Классический алгоритм на небольших размерах матрицы оказался быстрее, не требуя при этом поддержки поточности. Количество потоков, большее, чем количество ядер, не дало прироста до $N = 500$, а затем увеличило результаты на 20%.

Литература

- [1] Как использовать корреляцию для понимания взаимосвязи между переменными. Режим доступа <https://www.machinelearningmastery.ru/how-to-use-correlation-to-understand-the-relationship-between-variables/> (дата обращения: 29.10.2021).
- [2] Корреляционный анализ. Режим доступа <https://statanaliz.info/statistica/korrelyaciya-i-regressiya/linejnyj-koefficient-korrelyacii-pirsona/> (дата обращения: 29.10.2021).
- [3] `std::thread`. Режим доступа <https://en.cppreference.com/w/cpp/thread/thread> (дата обращения: 29.10.2021).
- [4] Date and time utilities. Режим доступа <https://en.cppreference.com/w/cpp/chrono> (дата обращения: 29.10.2021).
- [5] Процессор Intel® Core™ i5-1135G7. Режим доступа <https://www.intel.ru/content/www/ru/ru/products/sku/208658/intel-core-i51135g7-processor-8m-cache-up-to-4-20-ghz/specifications.html> (дата обращения: 29.10.2021).