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 №5 по курсу "Анализ алгоритмов"

Тема Конвейерные вычисления

Студент Прянишников А. Н.

Группа ИУ7-55Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Содержание

Введение

Многие задачи требуют огромных вычислительных мощностей, но даже суперкомпьютеры не всегда справляются с поставленными задачами. Вычислительные конвейеры предоставляют возможность распараллелить исходный алгоритм для оптимального использования системных ресурсов. Вычислительные конвейеры используются в процессорах для ускорения выполнения инструкций.

Целью данной работы является разработка программы, которая реализует два способа реализации вычислительного конвейера: линейный и с использованием параллельных вычислений.

Для достижения поставленной цели необходимо выполнить следующее:

- рассмотреть методологию конвейерной разработки;
- рассмотреть способы распараллеливания конвейерных вычислений;
- привести схемы реализации конвейера;
- определить средства программной реализации;
- реализовать рассматриваемые способы реализации вычислительного конвейера;
- протестировать разработанное ПО;
- провести модульное тестирование всех реализаций алгоритмов;
- оценить реализацию алгоритмов по времени и памяти.

1 Аналитический раздел

В этом разделе будут проведена формализация задачи, рассмотрена методология вычислительных конвейеров и возможности для распараллеливания.

1.1 Вычислительный конвейер

Конвейер - система поточного производства [1]. Вычислительный конвейер — способ организации вычислений, используемый в современных процессорах и контроллерах с целью повышения их производительности, а также технология, используемая при разработке компьютеров и других цифровых электронных устройств [2].

Идея заключается в параллельном выполнении нескольких инструкций процессора. Сложные инструкции процессора представляются в виде последовательности более простых стадий. Вместо выполнения инструкций последовательно (ожидания завершения конца одной инструкции и перехода к следующей), следующая инструкция может выполняться через несколько стадий выполнения первой инструкции. Это позволяет управляющим цепям процессора получать инструкции со скоростью самой медленной стадии обработки, однако при этом намного быстрее, чем при выполнении эксклюзивной полной обработки каждой инструкции от начала до конца.

На рисунке 1 представлен пример реализации вычислительного конвейера, где каждая буква означает одно из действий при выполнении процессором команды.

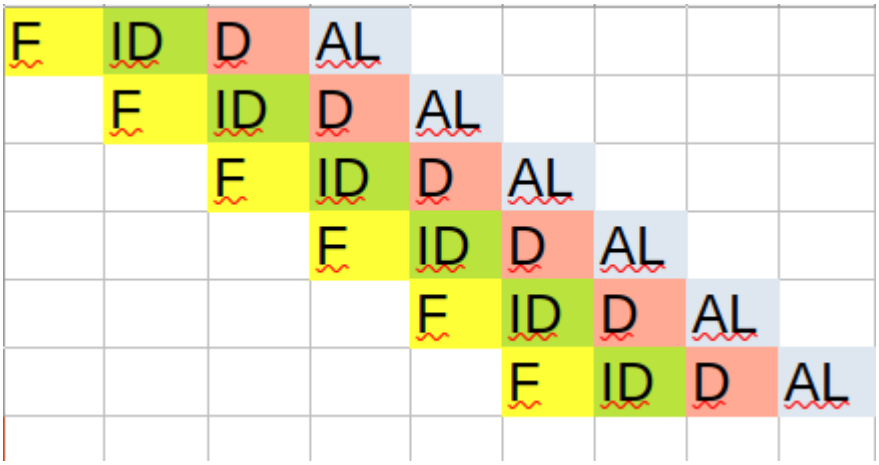

Рисунок. 1.1: Демонстрация вычислительного конвейера

1.2 Основной алгоритм

В качестве задачи для реализации выбрана следующая. Пусть дан массив arr. Требуется произвести стандартизацию данных. Это означает, что каждый элемент массива должен быть преобразован по формуле 1.1:

$$Arr[i] = \frac{Arr[i] - \mu}{\sigma} \tag{1.1}$$

где μ - среднее арифметическое массива, а σ - стандартное отклонение.

1.3 Реализация конвейера для основного алгоритма

Выполнение задачи можно разбить на несколько этапов:

1. Вычисление среднего арифметического массива.
2. Вычисление дисперсии по элементам массива. Для этого требуется знать среднее арифметическое из первого этапа.
3. Стандартизация по формуле 1.1 на основе дисперсии и среднего арифметического массива.

Вычислительный конвейер может быть построен на этих трёх этапах, так как три задачи выполняются за один порядок времени, соответственно, конвейер не будет задерживаться на одном отрезке.

Каждый этап конвейера записывает результат в очередь следующего этапа, соответственно, требуется использовать мьютексы для того, чтобы исключить ошибки по памяти.

1.4 Вывод

Была рассмотрена методология вычислительного конвейера, произведена формализация задачи и рассмотрены способы распараллеливания основного алгоритма.

2 Конструкторский раздел

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 массив, состоящий из вещественных чисел;
- в массиве минимум два элемента;
- в массиве минимум два элемента отличаются друг от друга.

Требования к выводу:

- программа должна вывести логи обработки данных при выполнении различных заданий.

2.2 Схемы алгоритмов

На рисунке 2.1 будет приведена схема реализации линейного алгоритма стандартизации данных.

На рисунках 2.2-2.5 будут приведены схемы реализации алгоритма стандартизации данных с использованием конвейерных вычислений.

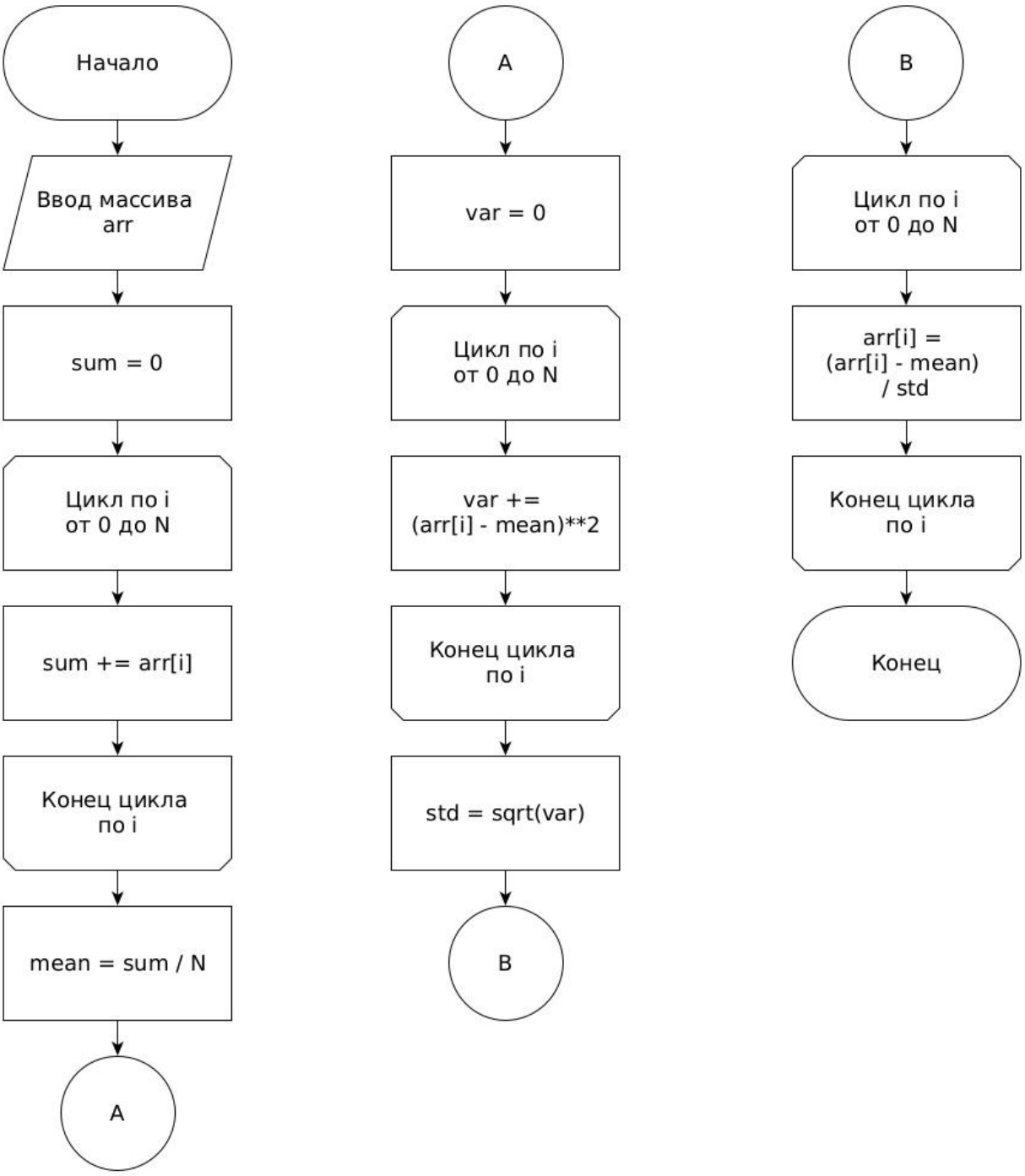

Рисунок. 2.1: Схема линейного алгоритма стандартизации данных

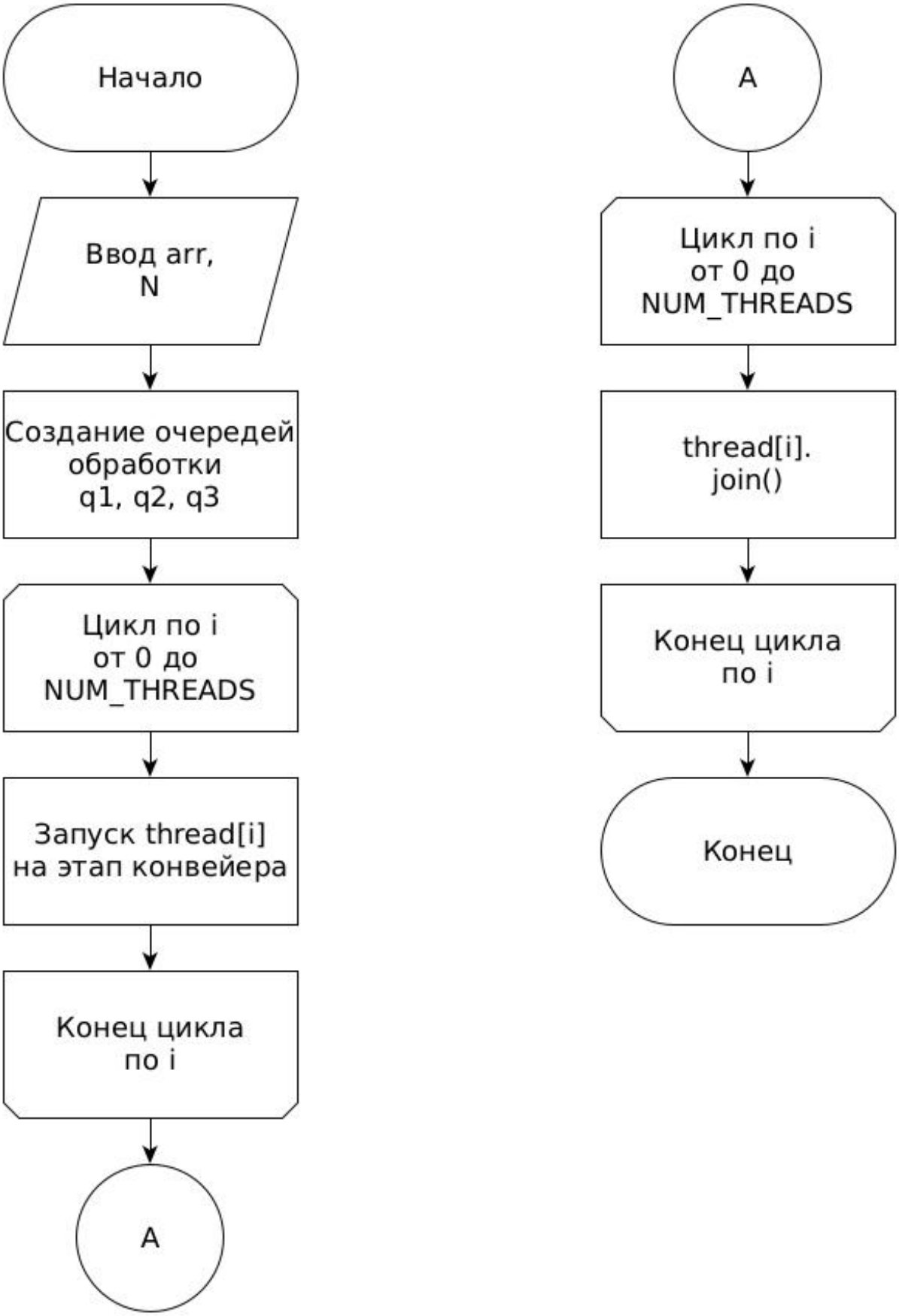

Рисунок. 2.2: Схема работы главного потока для реализации алгоритма стандартизации данных с использованием конвейера

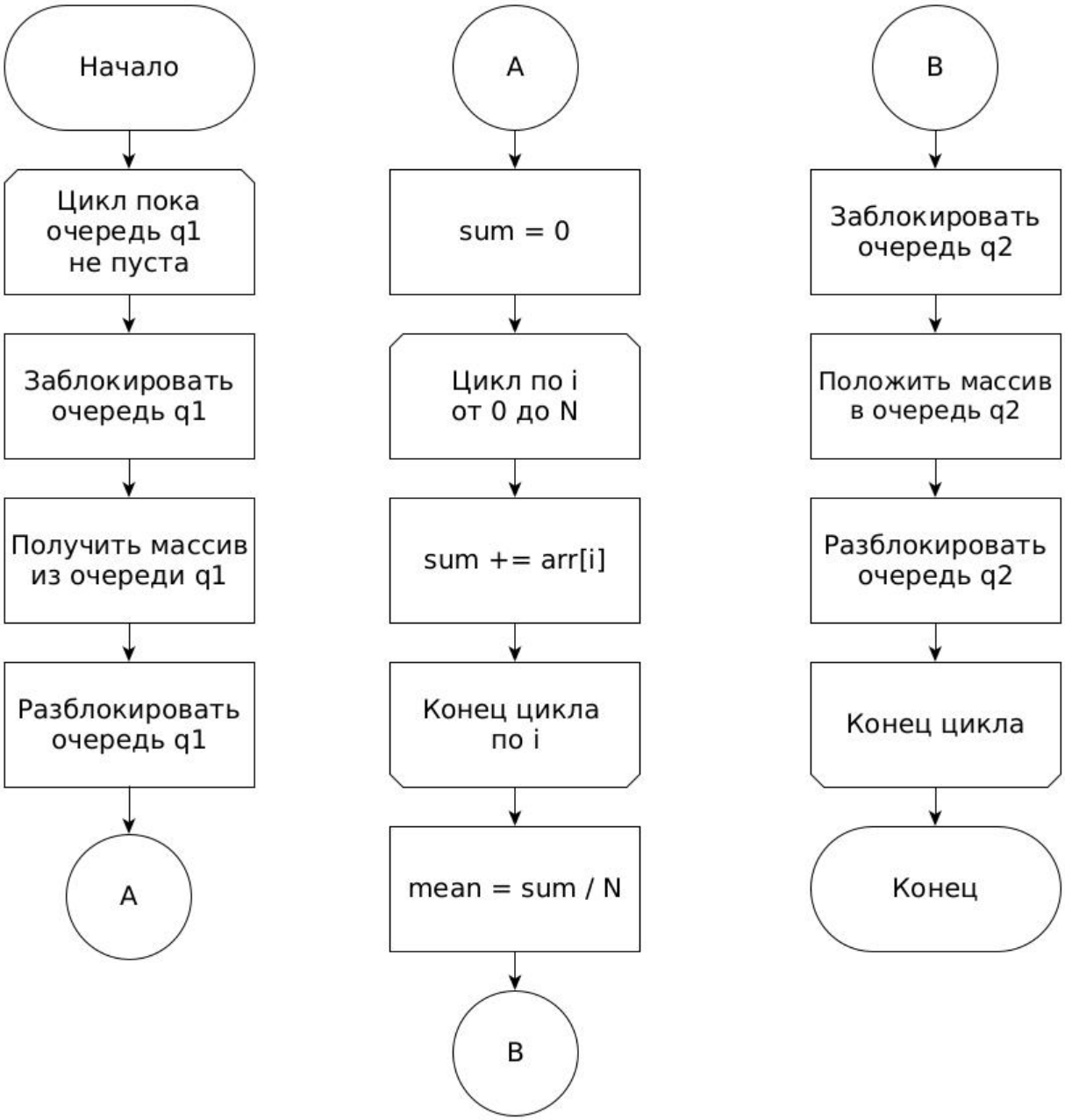

Рисунок. 2.3: Схема работы потока на первом этапе конвейерных вычислений

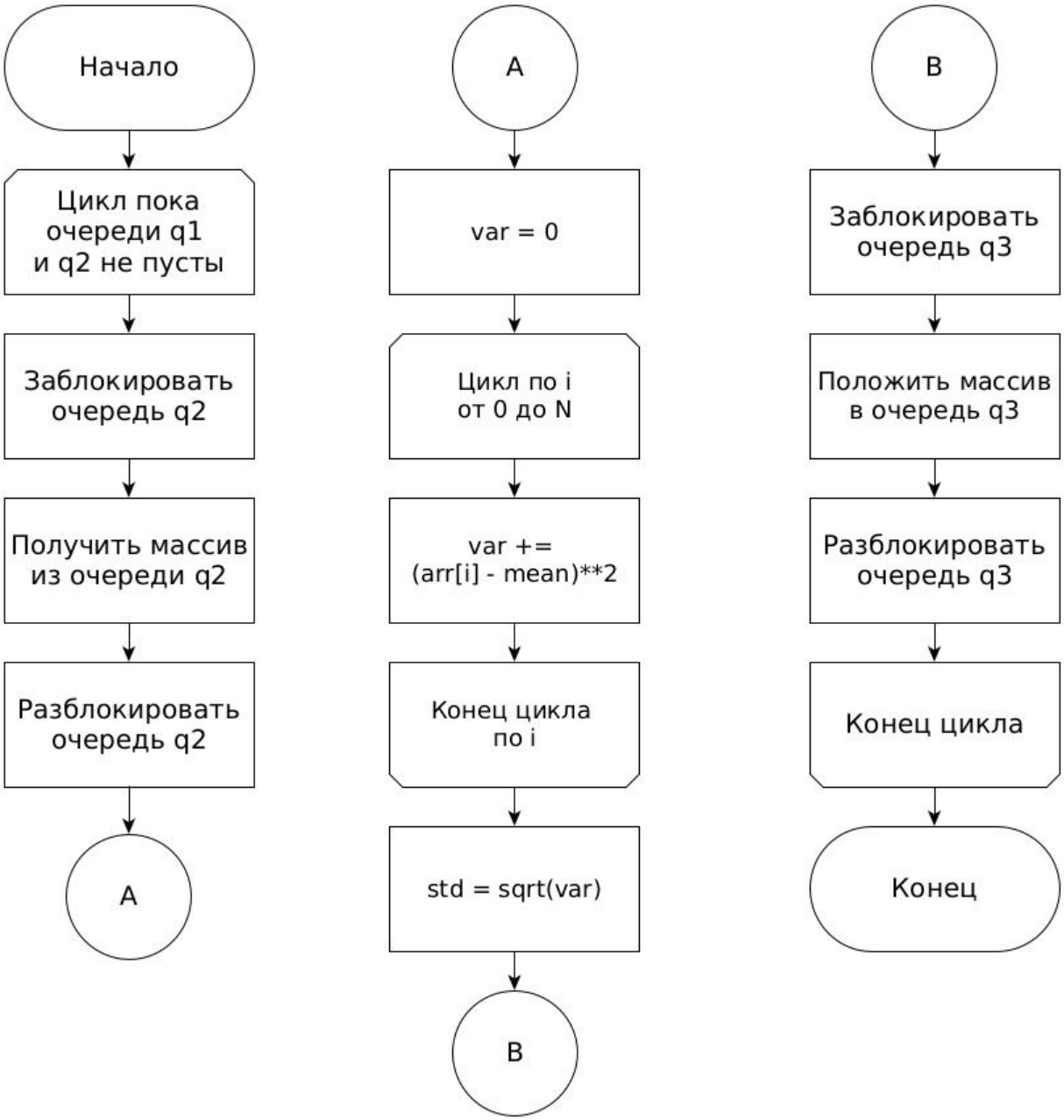

Рисунок. 2.4: Схема работы потока на втором этапе конвейерных вычислений

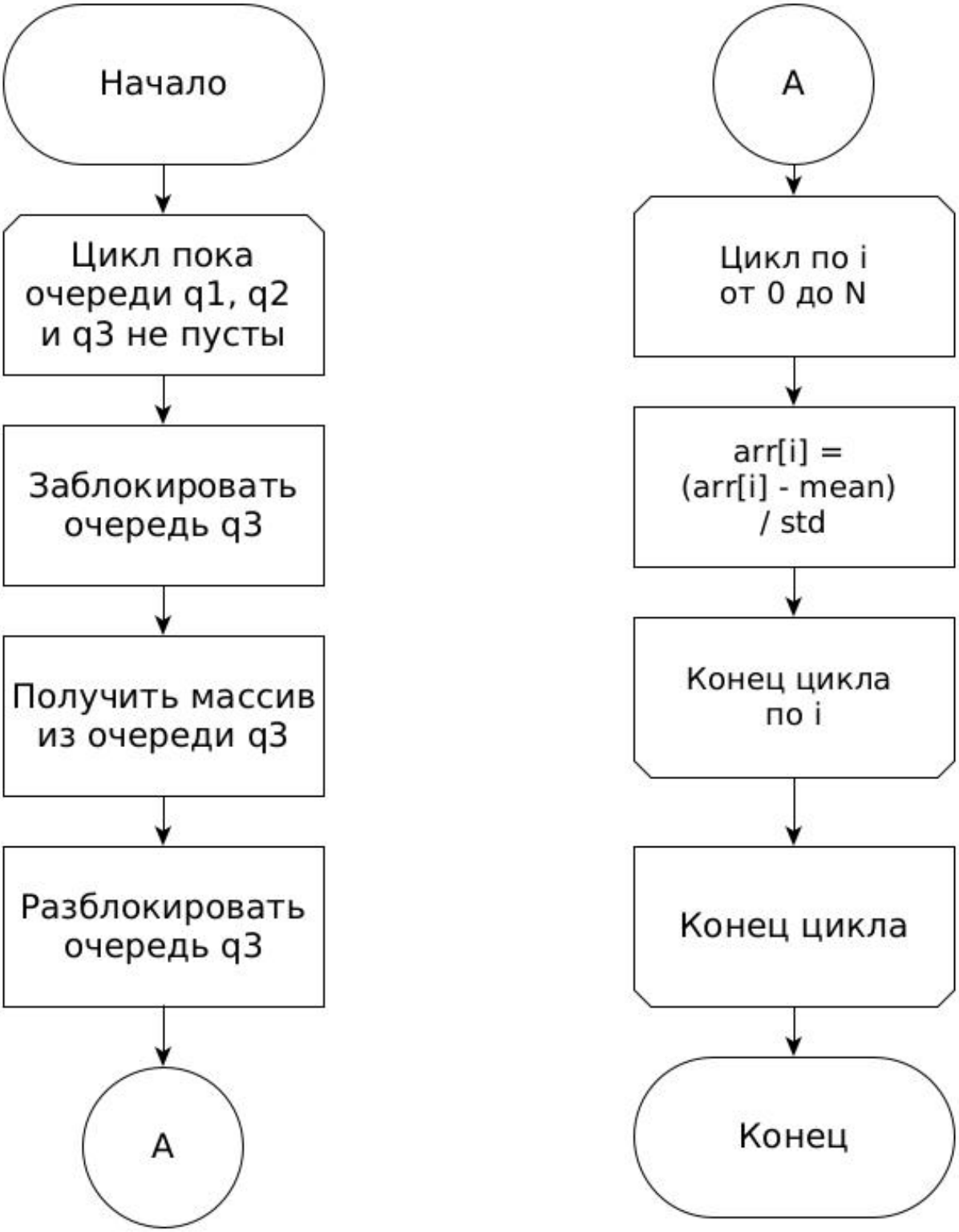

Рисунок. 2.5: Схема работы потока на третьем этапе конвейерных вычислений

2.3 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 массива может быть произвольным из тех, что допустимы по требованиям ко вводу.

2.4 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:

- массив состоит из двух элементов;
- массив состоит из множества разных элементов;
- массив состоит из целых чисел, следующих на числовой прямой друг за другом (например, -1 0 1).
- массив состоит из чисел, очень близких друг к другу.
- массив состоит из 5 чисел от 0 до 1000

Для тестирования требуется сравнить полученное значение с эталонным. Для этого использовалась библиотека Sklearn и метод StandartScaler, который выполняют стандартизацию входного массива.

2.5 Вывод

Были приведены требования к ПО, схемы реализации алгоритмов. Был 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 листингах 4.1-4.4 представлена реализация фрагментов разработанного алгоритма.

Листинг 3.1: Реализация линейного алгоритма

```
void stage1_linear(array_t &matrix, int task_num, bool is_print)
{
    log_linear(matrix, task_num, 1, get_avg, is_print);
}



void stage2_linear(array_t &matrix, int task_num, bool is_print)
{
    log_linear(matrix, task_num, 2, get_var, is_print);
}


void stage3_linear(array_t &matrix, int task_num, bool is_print)
{
    log_linear(matrix, task_num, 3, fill_array, is_print);
}



void parse_linear(int count, size_t size, bool is_print)
```

```
{

    time_now = 0;

    std::queue<array_t> q1;
    std::queue<array_t> q2;
    std::queue<array_t> q3;

    queues_t queues = {.q1 = q1, .q2 = q2, .q3 = q3};

    for (int i = 0; i < count; i++)
    {
        array_t res = generate_array(size);

        queues.q1.push(res);
    }

    for (int i = 0; i < count; i++)
    {
        array_t matrix = queues.q1.front();
        stage1_linear(matrix, i + 1, is_print);
        queues.q1.pop();
        queues.q2.push(matrix);

        matrix = queues.q2.front();
        stage2_linear(matrix, i + 1, is_print); // Stage 2
        queues.q2.pop();
        queues.q3.push(matrix);

        matrix = queues.q3.front();
        stage3_linear(matrix, i + 1, is_print); // Stage 3
        queues.q3.pop();
    }
}
```

Листинг 3.2: Реализация этапов вычислительного конвейера

```
void stage1_parallel(std::queue<array_t> &q1, std::queue<array_t> &q2,
                     std::queue<array_t> &q3, bool is_print)
{
    int task_num = 1;

    std::mutex m;

    while(!q1.empty())
    {
        m.lock();
        array_t matrix = q1.front();
        m.unlock();
```

```
        log_conveyor(matrix, task_num++, 1, get_avg, is_print);

        m.lock();
        q2.push(matrix);
        q1.pop();
        m.unlock();
    }
}


void stage2_parallel(std::queue<array_t> &q1, std::queue<array_t> &q2,
                     std::queue<array_t> &q3, bool is_print)
{
    int task_num = 1;

    std::mutex m;

    do
    {
        m.lock();
        bool is_q2empty = q2.empty();
        m.unlock();

        if (!is_q2empty)
        {
            m.lock();
            array_t matrix = q2.front();
            m.unlock();

            log_conveyor(matrix, task_num++, 2, get_var, is_print);

            m.lock();
            q3.push(matrix);
            q2.pop();
            m.unlock();
        }
    } while (!q1.empty() || !q2.empty());
}


void stage3_parallel(std::queue<array_t> &q1, std::queue<array_t> &q2,
                     std::queue<array_t> &q3, bool is_print)
{
    int task_num = 1;

    std::mutex m;
```

```
    do
    {
        m.lock();
        bool is_q3empty = q3.empty();
        m.unlock();

        if (!is_q3empty)
        {
            m.lock();
            array_t matrix = q3.front();
            m.unlock();

            log_conveyor(matrix, task_num++, 3, fill_array, is_print);

            m.lock();
            q3.pop();
            m.unlock();
        }
    } while (!q1.empty() || !q2.empty() || !q3.empty());
}
```

Листинг 3.3: Общая реализация вычислительного конвейера

```
void parse_parallel(int count, size_t size, bool is_print)
{

    time_now = 0;

    t1.resize(count + 1);
    t2.resize(count + 1);
    t3.resize(count + 1);

    for (int i = 0; i < count + 1; i++)
    {
        t1[i] = 0;
        t2[i] = 0;
        t3[i] = 0;
    }

    std::queue<array_t> q1;
    std::queue<array_t> q2;
    std::queue<array_t> q3;

    queues_t queues = {.q1 = q1, .q2 = q2, .q3 = q3};


    for (int i = 0; i < count; i++)
    {
        array_t res = generate_array(size);
```

```
        q1.push(res);
    }

    std::thread threads[THREADS];

    threads[0] = std::thread(stage1_parallel, std::ref(q1), std::ref(q2),
       std::ref(q3), is_print);
    threads[1] = std::thread(stage2_parallel, std::ref(q1), std::ref(q2),
       std::ref(q3), is_print);
    threads[2] = std::thread(stage3_parallel, std::ref(q1), std::ref(q2),
       std::ref(q3), is_print);

    for (int i = 0; i < THREADS; i++)
    {
        threads[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

Массив	Ожидаемый результат
1, 2	-1, 1
-3, 2, 11, 8, 0	-1.28, -0.31, 1.43, 0.85, -0.69
-2, -1, 0, 1, 2	-1.41, -0.71, 0, 0.71, 1.41
0.1, 0.12, 0.11, 0.13, 0.1	-1.03, 0.69, -0.17, 1.54, -1.03
8.56, 992.12, 134, 1, 567	-0.86, 1.69, -0.54, -0.88, 0.58

Все тесты алгоритмами были пройдены успешно.

3.4 Вывод

Были 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 рисунках 4.1-4.2 приведена демонстрация работы программы для линейной реализации алгоритма и параллельной.

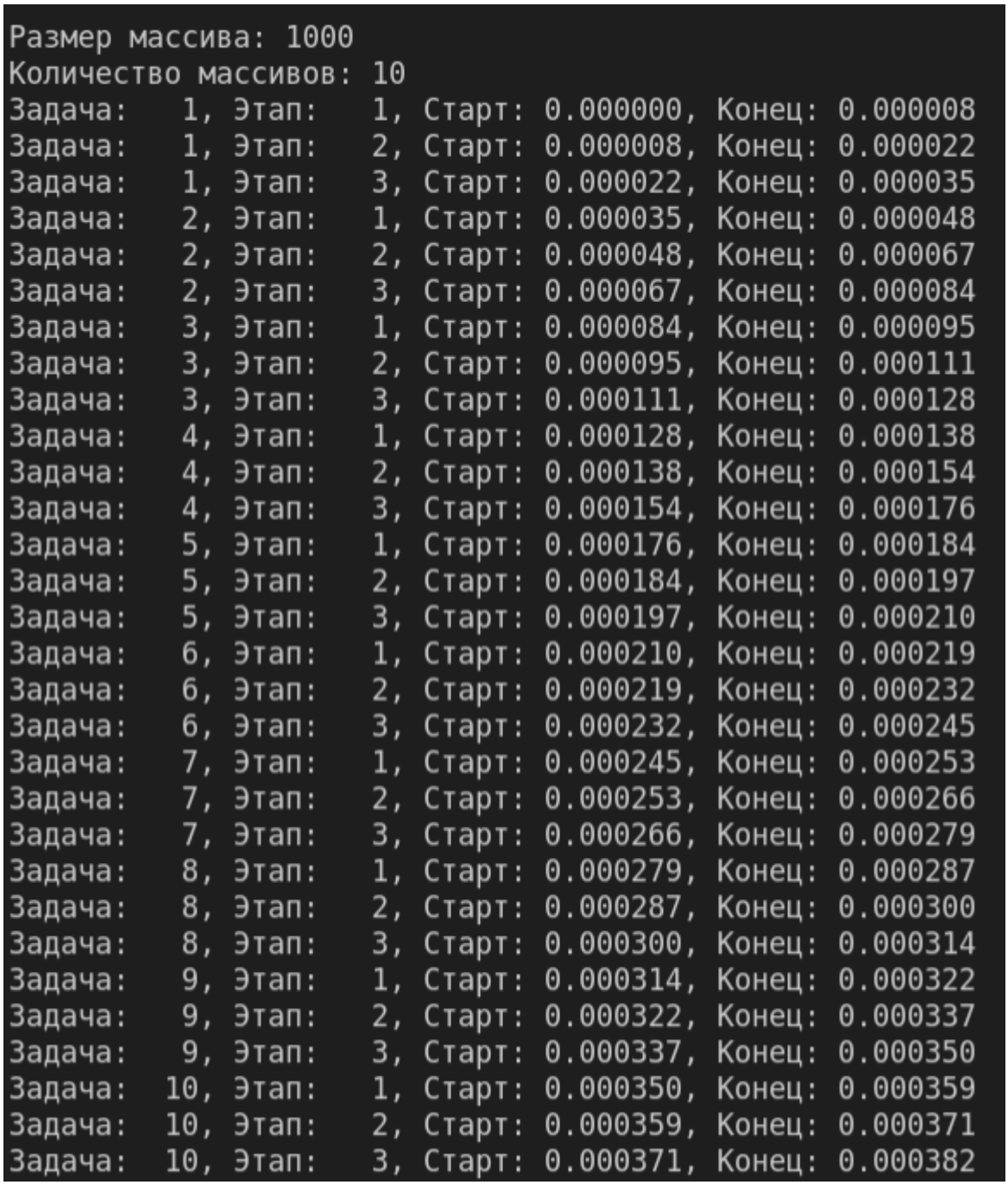

Рисунок. 4.1: Демонстрация работы линейного алгоритма

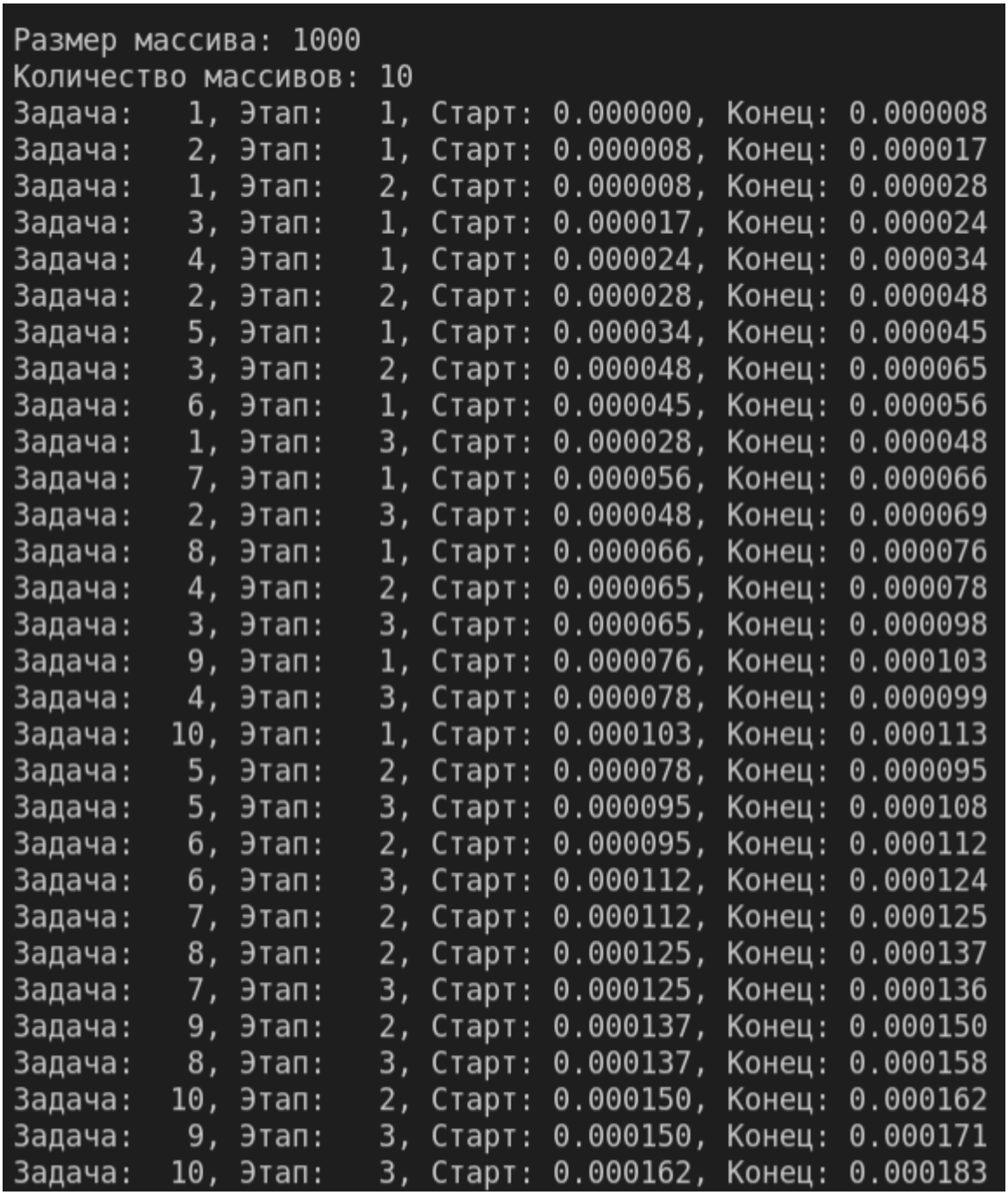

Рисунок. 4.2: Демонстрация реализации за счёт вычислительного конвейера

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

Тестирование будет производиться для двух различных ситуаций.

В первом случае во всех массивах 5000 элементов, и тестирование будет производиться на разном количестве массивов. Во втором случае будет зафиксировано 50 массивов, и тестирование будет производиться на разном числе элементов в массиве.

Это сделано для того, чтобы можно было сделать более подробный вывод насчёт конвейерных вычислений.

Результаты тестирования первого случая представлены в таблице (время - в мс). В первом столбце - размер массива, во втором - время работы линейного конвейера, в третьем - параллельного.

Таблица 4.1: Результаты тестов

Размер массива	Линейный	Параллельный
2	0.000339	0.000234
5	0.000850	0.000547
10	0.002557	0.001240
25	0.002948	0.001512
50	0.004570	0.001927
75	0.009906	0.005649
100	0.008375	0.003910
150	0.012908	0.005777
200	0.018446	0.010047
300	0.027001	0.015216
500	0.042561	0.020329
700	0.081352	0.038265
850	0.092877	0.032452
1000	0.094582	0.041348

На рисунке 4.1 показан график зависимости времени работы алгоритмов от количества массивов.

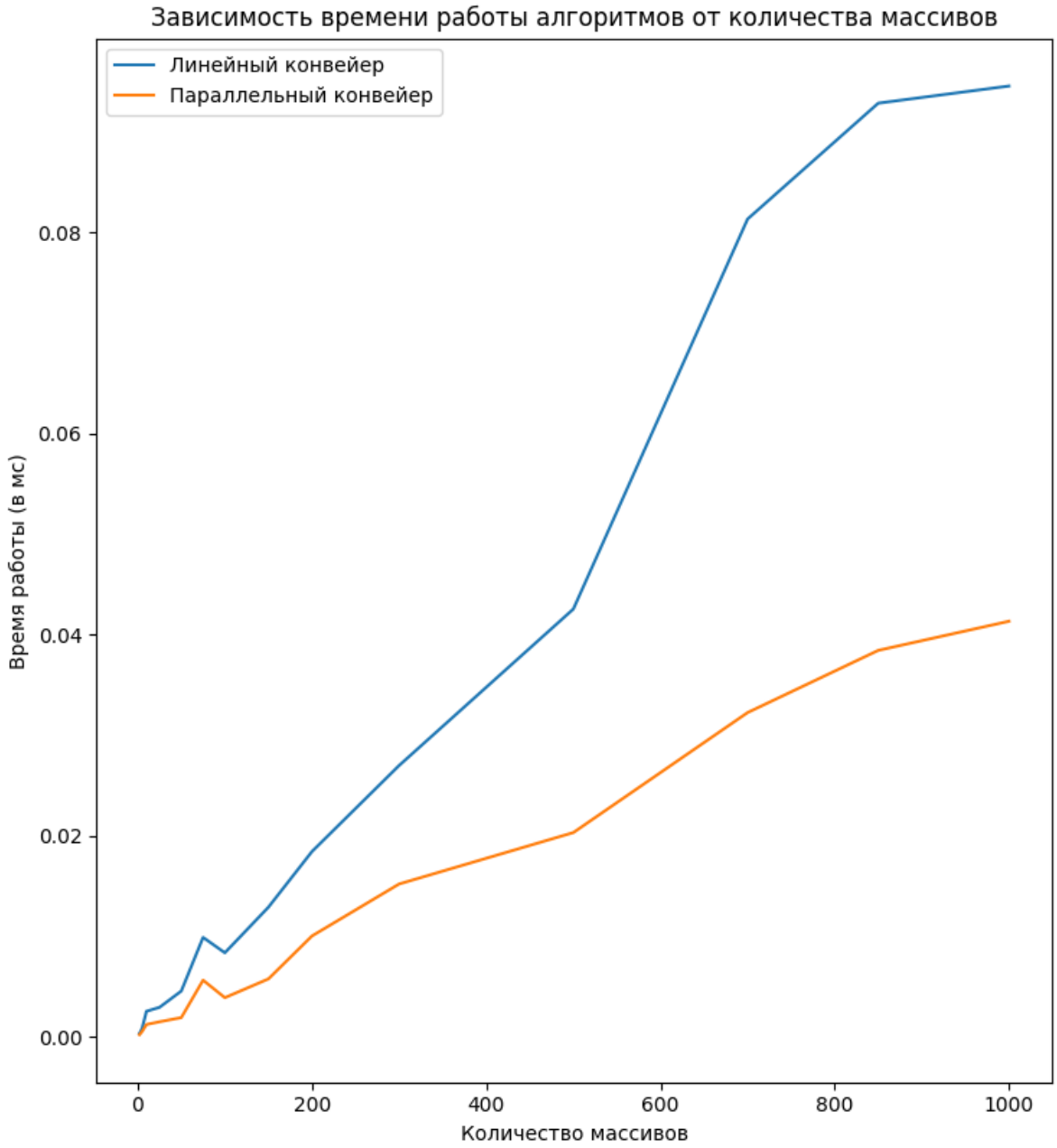

Рисунок. 4.3: График зависимости времени работы алгоритмов от количества массивов

Результаты тестирования второго случая представлены в таблице (время - в мс). В первом столбце - размер массива, во втором - время работы линейного конвейера, в третьем - параллельного.

Таблица 4.2: Результаты тестов

Количество элементов массива	Линейный	Параллельный
2	0.000179	0.000036
5	0.000192	0.000066
10	0.000371	0.000105
50	0.000908	0.000185
100	0.002483	0.000395
200	0.003450	0.000996
300	0.003985	0.001116
500	0.005747	0.001290
750	0.006763	0.002194
1000	0.008604	0.003437
2000	0.012041	0.007453
3000	0.017266	0.012255
5000	0.030266	0.018450
6000	0.038714	0.018948
7000	0.040697	0.021442
8000	0.052930	0.027142
10000	0.072837	0.031553

На рисунке 4.1 показан график зависимости времени работы алгоритмов от размера массивов.

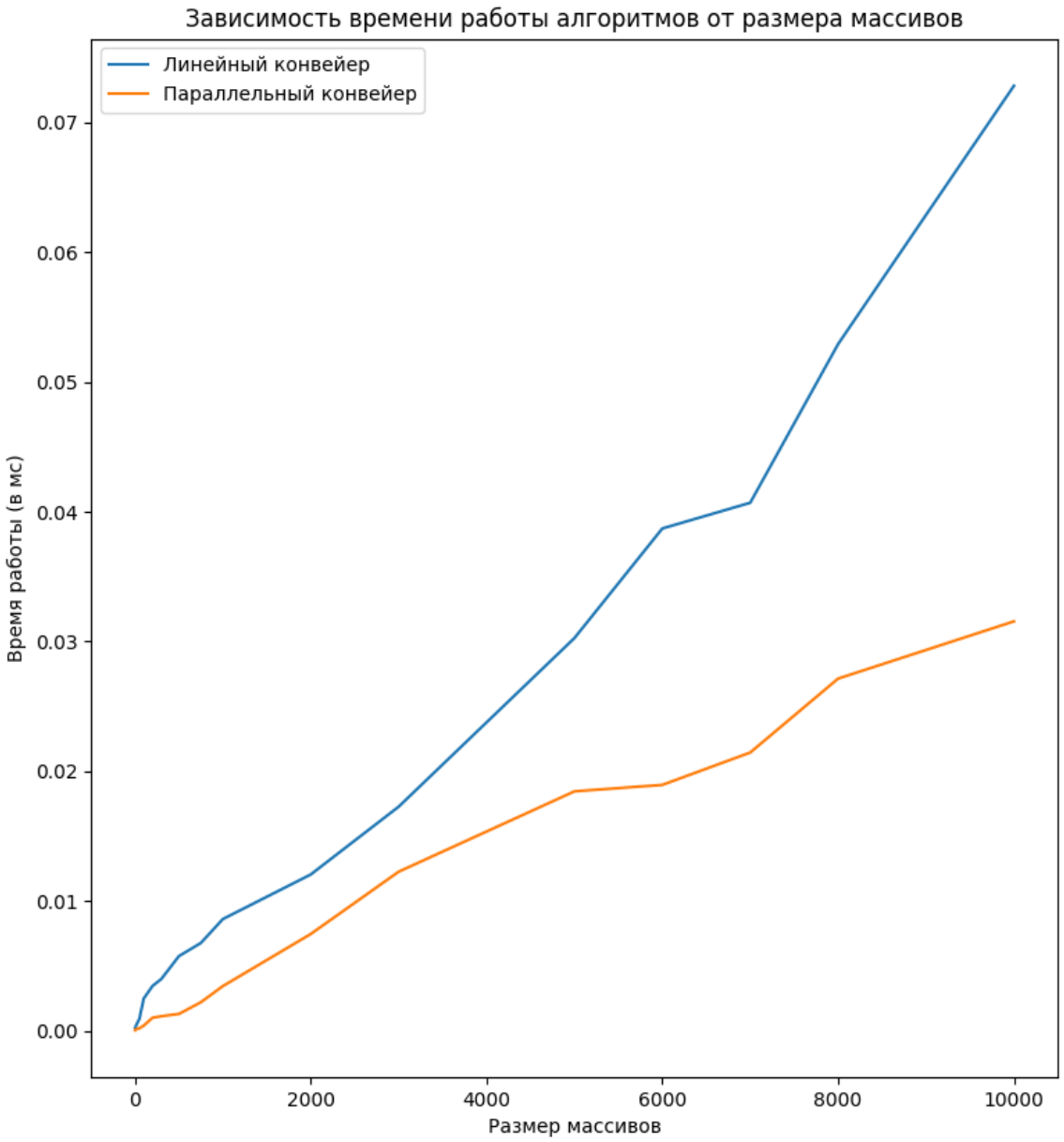

Рисунок. 4.4: График зависимости времени работы алгоритмов от размера массивов

4.4 Вывод

Параллельный конвейер оказался более быстрым по всем измерениям. При количестве массивов $N = 10$ параллельный алгоритм работает в 2.06 раз быстрее, чем линейная реализация. При количестве массивов $N = 700$ параллельная реализация опередила линейную в 2.12 раз, а на $N = 1000$ разница достигла 2.3 раз. Уже при размере очереди в два элемента реализация вычислительного конвейера оправдывает себя.

При фиксированном количестве массивов (500) параллельный конвейер также быстрее при любом размере массива. При размере массива $N = 50$ параллельная реализация опередила линейную в 5.34 раза. При $N = 5000$ конвейер был быстрее в 1.6 раз, при $N = 10000$ – в 2.3 раза.

Таким образом, конвейер позволил существенно увеличить скорость выполнения задачи, разделив одну задачу на несколько простых.

5 Заключение

В ходе работы была рассмотрена методология конвейерных вычислений на примере задачи стандартизации данных. Были составлены схемы алгоритмов. Было реализовано работоспособное ПО, удалось провести анализ зависимости затрат по времени от размера массива и количества массивов.

Вычислительный конвейер оказался быстрее на всех тестах. При количестве массивов $N = 10$ параллельный алгоритм работает в 2.06 раз быстрее, чем линейная реализация. При количестве массивов $N = 1000$ разница достигла 2.3 раз. Уже при размере очереди в два элемента реализация вычислительного конвейера оправдывает себя.

При фиксированном количестве массивов (500) параллельный конвейер также быстрее при любом размере массива. При размере массива $N = 50$ параллельная реализация опередила линейную в 5.34 раза. При $N = 5000$ конвейер был быстрее в 1.6 раз, при $N = 10000$ – в 2.3 раза.

Конвейерные вычисления позволили существенно увеличить скорость выполнения задачи.

Литература

[1] В. П. Меднов Е. П. Бондарев. Транспортные, распределительные и рабочие конвейеры. – М, 1970.

[2] Центральный процессор «Эльбрус-4С» [Электронный ресурс]. Режим доступа: http://www.mcst.ru/mikroprocessor-elbrus4s (дата обращения: 18.12.2020).

[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).