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 Организация асинхронного взаимодействия потоков вычисления на примере конвейерных вычислений

Студент Пронина Л.Ю.

Группа ИУ7-54Б

Оценка (баллы) _____

Преподаватель Волкова Л. Л.

Москва — 2023 г.

Содержание

Введение

Конвейерная обработка является одним из примеров, где использование принципов параллельности помогает ускорить обработку данных. Суть та же, что и при работе реальных конвейерных лент — материал поступает на обработку, после окончания обработки материал передается на место следующего обработчика, при этом предыдущий обработчик не ждет полного цикла обработки материала, а получает новый материал и работает с ним.

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

— описать основы конвейерной обработки данных и алгоритма построения дерева синтаксических зависимостей, которые будут использоваться в текущей лабораторной работе;

— привести схемы конвейерной и линейной обработок;

— реализовать разработанные алгоритмы;

— провести сравнительный анализ по времени для реализованных алгоритмов;

— подготовить отчет о выполненной лабораторной работе.

1 Аналитическая часть

В этом разделе будет представлена информация по поводу сути конвейерной обработки данных.

1.1 Конвейерная обработка данных

Конвейер [1] – организация вычислений, при которой увеличивается количество выполняемых инструкций за единицу времени за счет использования принципов параллельности.

Конвейеризация в общем случае основана на разделении подлежащей исполнению функции на более мелкие части, называемые ступенями, и выделении для каждой из них отдельного блока аппаратуры. Так, обработку любой машинной команды можно разделить на несколько этапов (несколько ступеней), организовав передачу данных от одного этапа к следующему. При этом конвейерную обработку можно использовать для совмещения этапов выполнения разных команд. Производительность при этом возрастает, благодаря тому, что одновременно на различных ступенях конвейера выполняется несколько команд. Конвейерная обработка такого рода широко применяется во всех современных быстродействующих процессорах.

Конвейеризация увеличивает пропускную способность процессора (количество команд, завершающихся в единицу времени), но она не сокращает время выполнения отдельной команды. В действительности она даже несколько увеличивает время выполнения каждой команды из-за накладных расходов, связанных с хранением промежуточных результатов. Однако увеличение пропускной способности означает, что программа будет выполняться быстрее по сравнению с простой, неконвейерной схемой.

1.2 Деревья синтаксических зависимостей текста

Дерево синтаксических зависимостей представляет собой структуру, которая отображает отношения между словами в предложении или тексте. Оно

является графическим представлением синтаксической структуры предложения и позволяет анализировать связи между словами.

В дереве синтаксических зависимостей каждое слово представлено узлом, а связи между словами представлены направленными ребрами. Узлы могут иметь различные свойства, такие как лемма (нормальная форма слова), часть речи, морфологические характеристики и другие.

Каждое ребро в дереве синтаксических зависимостей указывает на отношение между словами. Например, ребро может указывать на то, что слово является зависимым от другого слова в качестве подлежащего, сказуемого, прямого или косвенного дополнения и т.д. Все эти отношения образуют иерархическую структуру, которая отображает семантику и синтаксис в предложении.

Построение дерева синтаксических зависимостей основано на лексическом и синтаксическом анализе текста. При анализе предложения или текста, алгоритм разбивает его на токены (слова), определяет их часть речи, а затем выстраивает связи между словами, чтобы создать дерево синтаксических зависимостей.

1.3 Описание алгоритмов

В качестве примера для конвейерной обработки будет обрабатываться текст. Всего будет использовано три ленты, которые делают следующее.

1) Чтение текста из файла.

2) Разбиение текста на предложения.

3) Построение деревьев синтаксических зависимостей для предложений.

1.4 Вывод

В данном разделе было рассмотрено понятие конвейерной обработки, а также выбраны этапы для обработки текста, которые будут обрабатывать ленты конвейера.

2 Конструкторская часть

В этом разделе будут представлены схемы алгоритмов конвейрной и линейной обработки текста.

2.1 Разработка алгоритмов

На рисунке 2.1 представлена схема алгоритма линейной обработки текста. На рисунках 2.2 схема алгоритма конвейерной обработки текста, а на рисунках 2.3-2.5 — схемы потоков обработки текста (ленты конвейера).

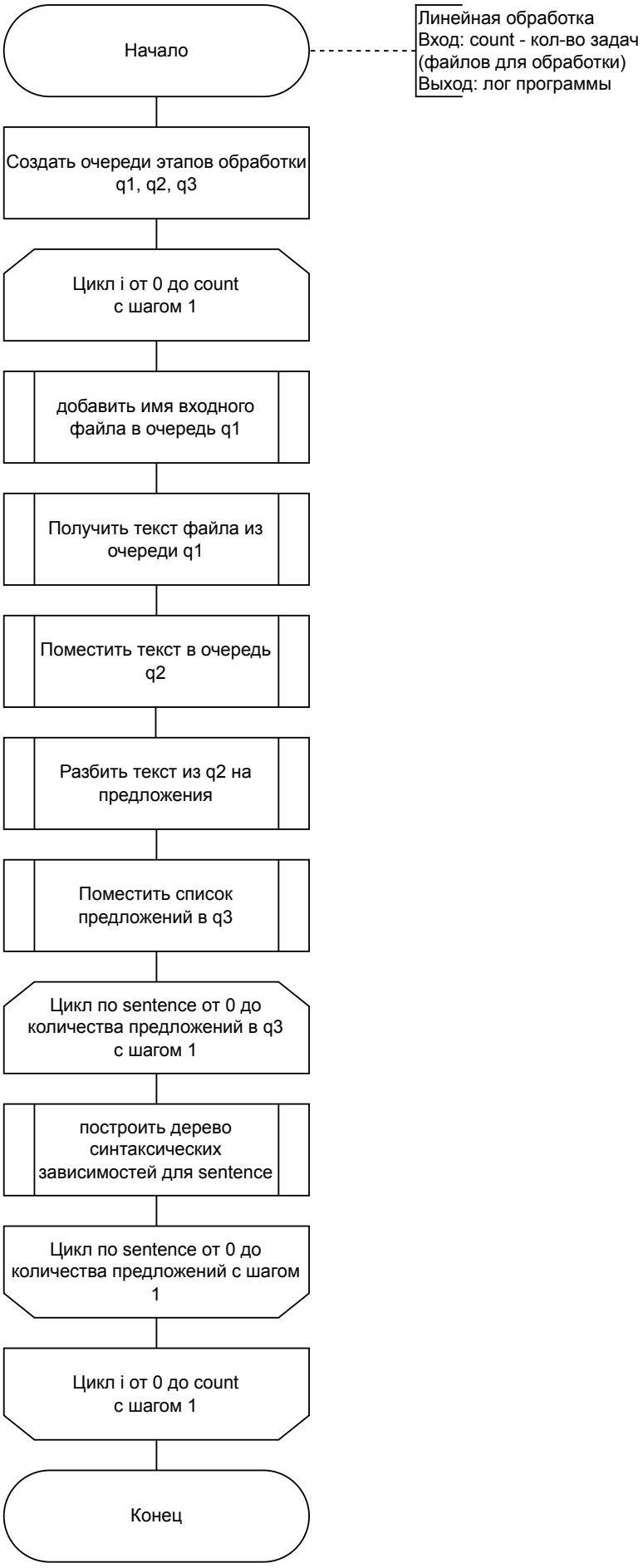

Рисунок 2.1 – Схема алгоритма линейной обработки матрицы

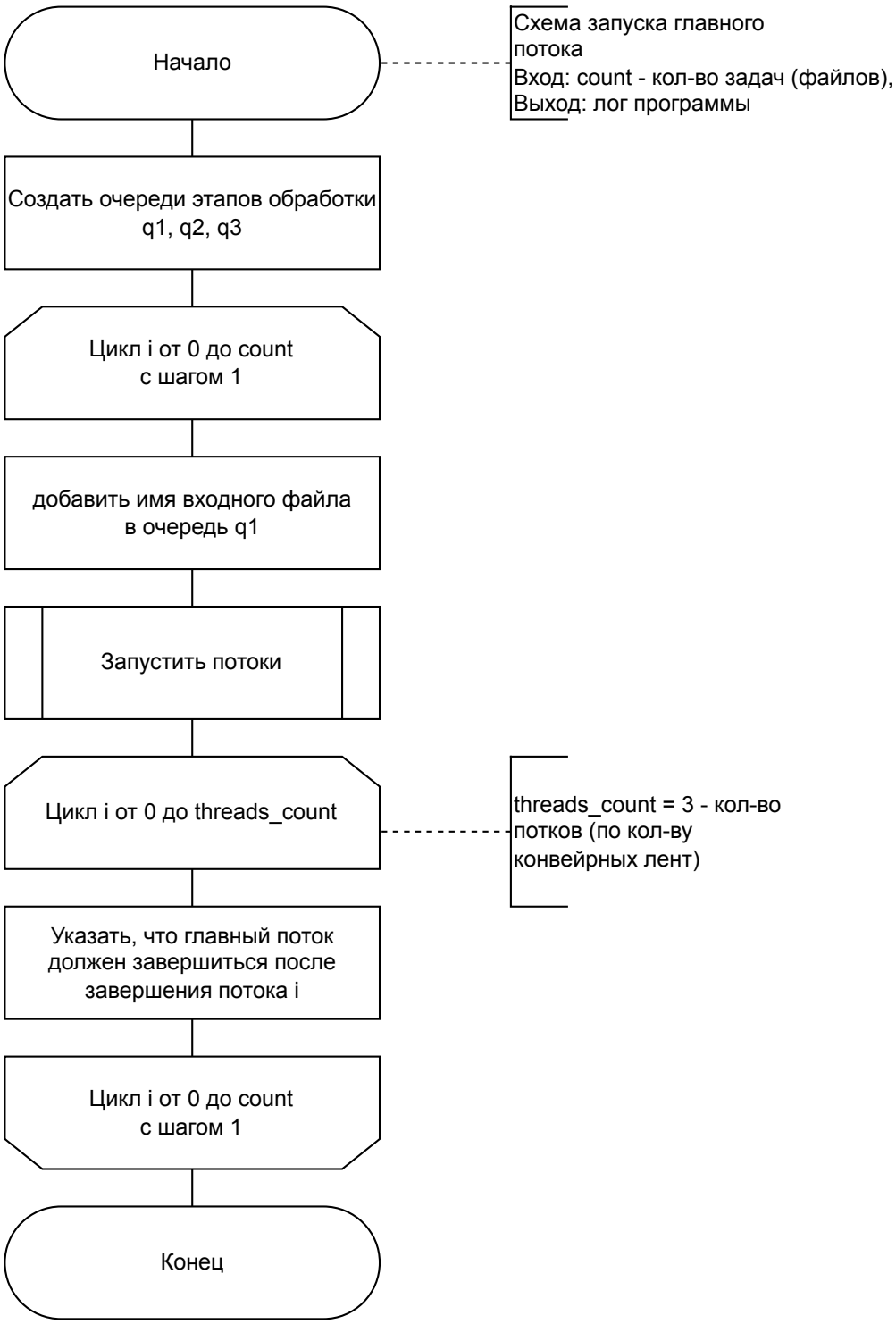

Рисунок 2.2 – Схема конвейерной обработки матрицы

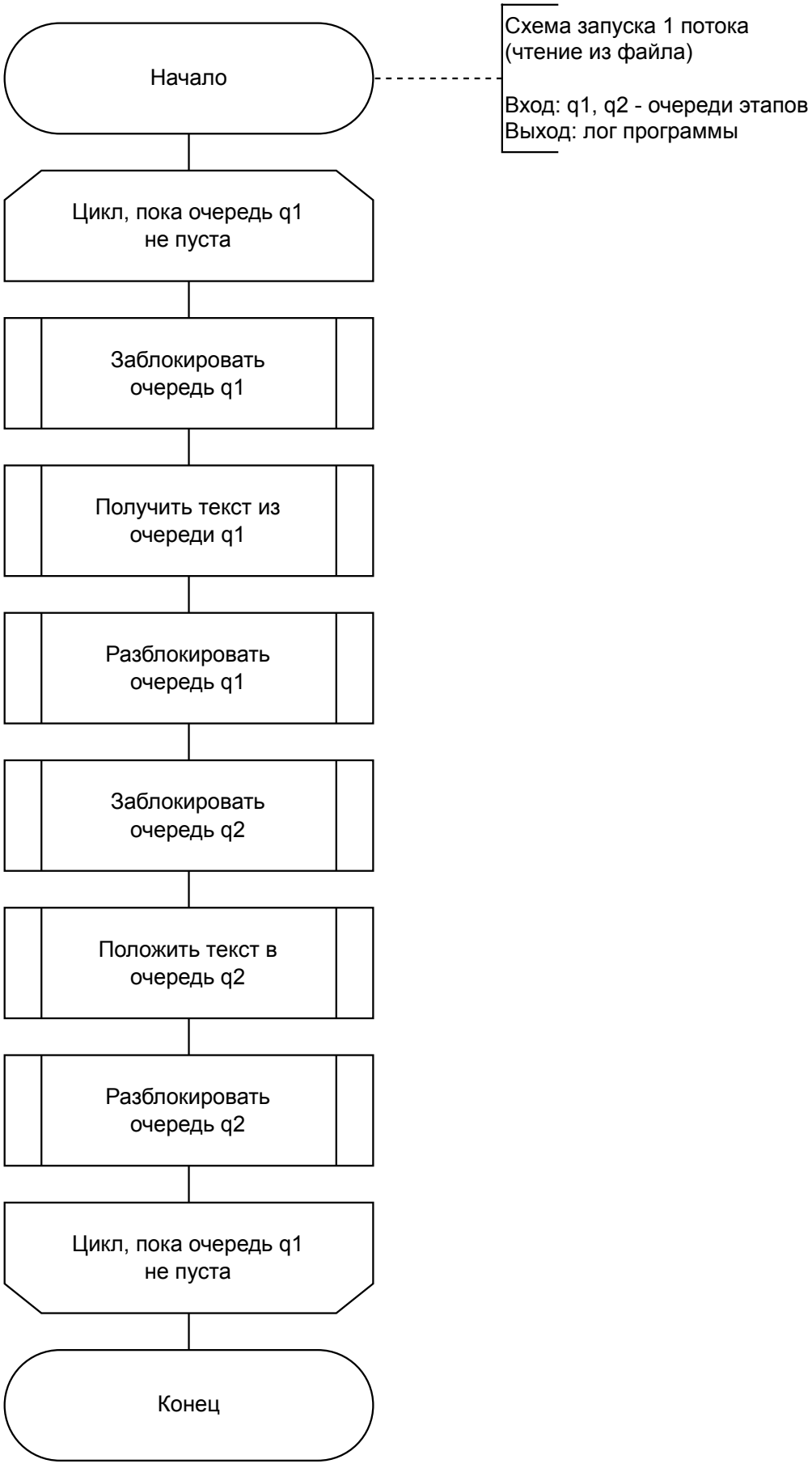

Рисунок 2.3 – Схема 1 потока обработки матрицы — чтение текста из файла

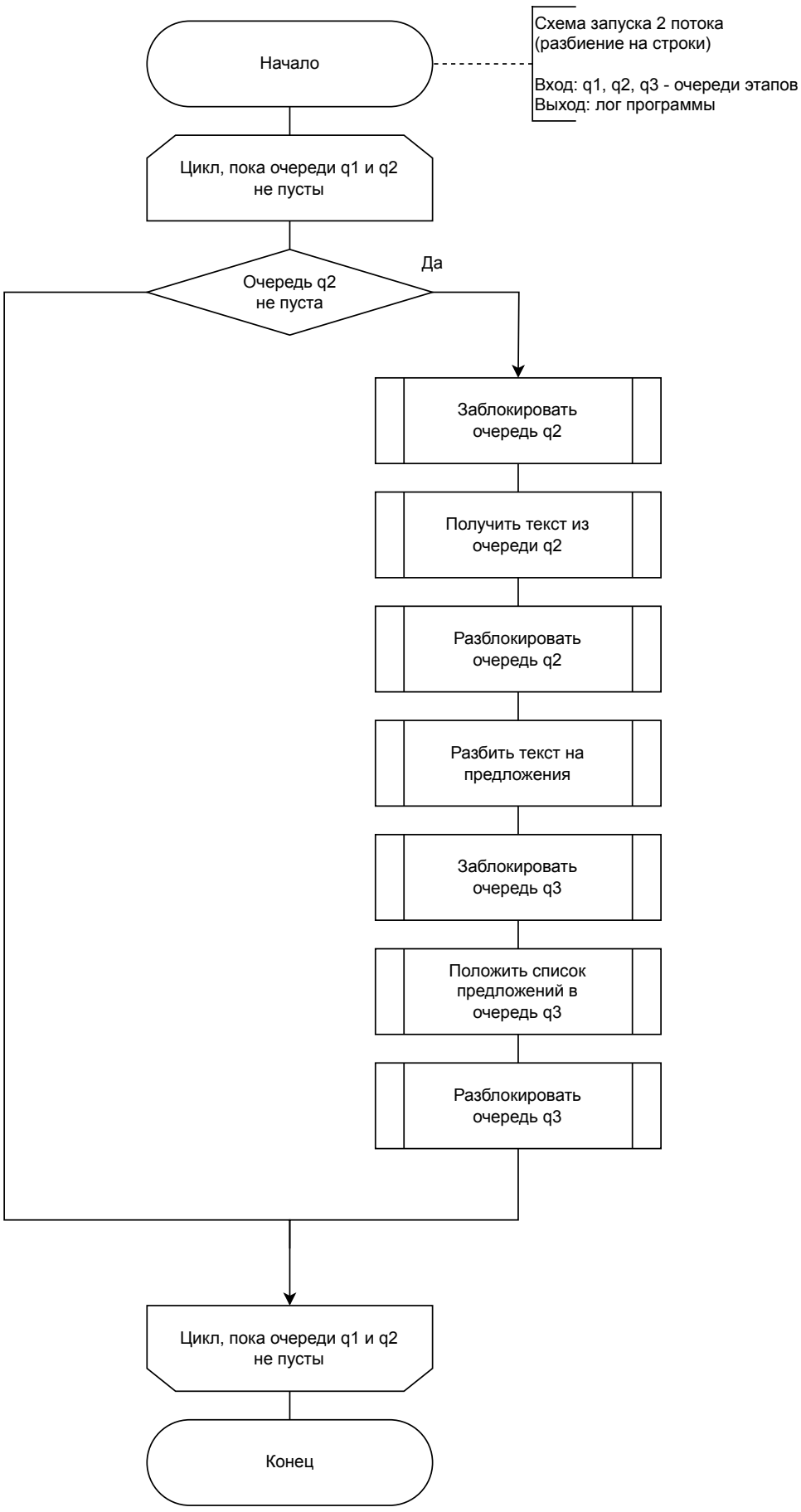

Рисунок 2.4 – Схема 2 потока обработки матрицы — разбиение текста на предложения

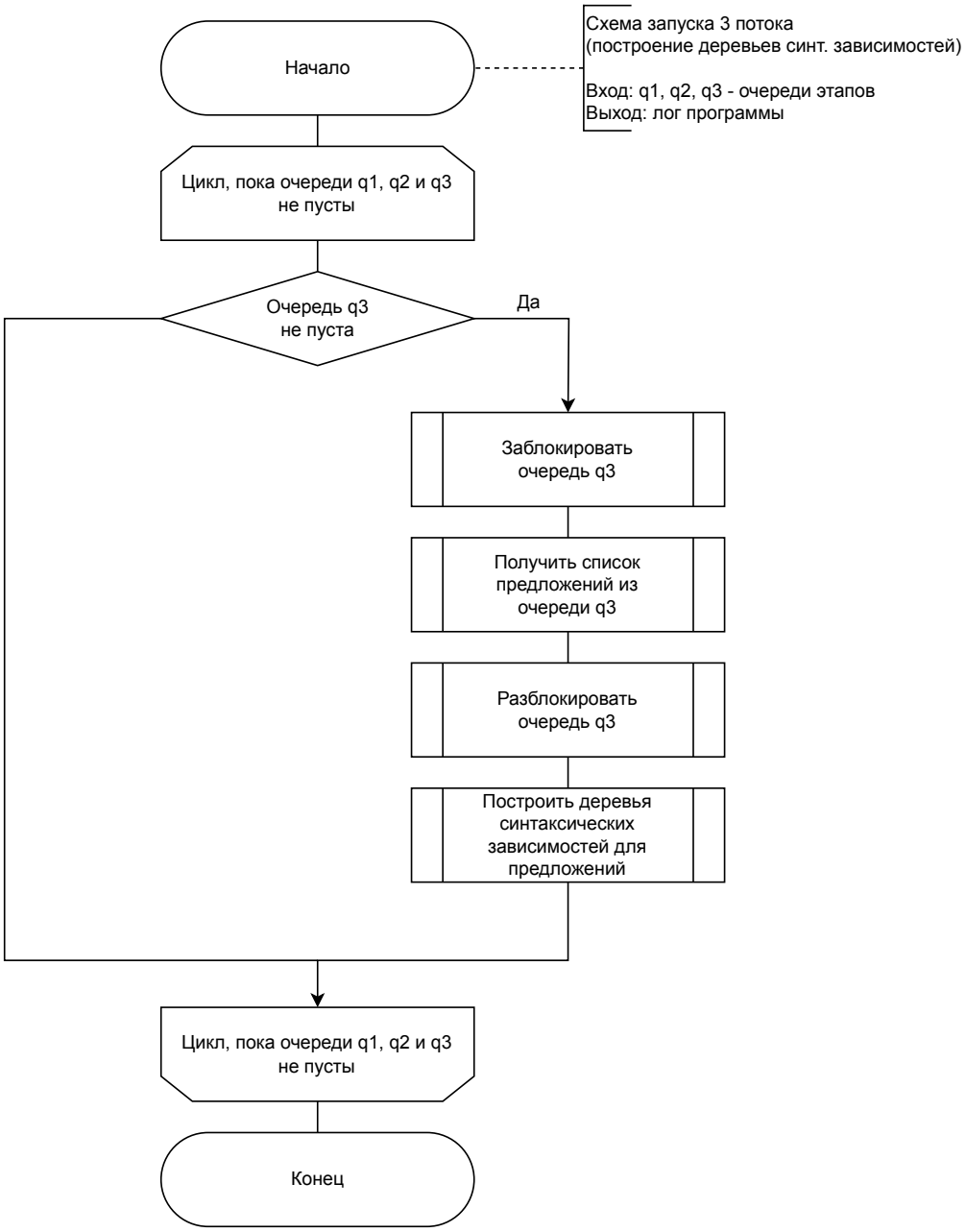

Рисунок 2.5 – Схема 3 потока обработки матрицы — построение деревьев синтаксических зависимостей

2.2 Вывод

В данном разделе были построены схемы алгоритмов, рассматриваемых в лабораторной работе.

3 Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги линейной и конвейерной орбаботок текста.

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования $C++$ [2]. В текущей лабораторной работе требуется замерить процессорное время для выполняемой программы, а также реализовать принципы многопоточного алгоритма. Все эти инструменты присутствуют в выбранном языке программирования.

Функции построения графиков были реализованы с использованием языка программирования *Python* [3].

Время замерено с помощью *std::chrono::system_clock::now(...)* — функции из библиотеки *chrono* [4].

3.2 Реализация алгоритмов

В листинге 3.1 представлена реализация алгоритма линейной обработки текста, а в листингах 3.2–3.4 — реализация алгоритма конвейерной обработки текста.

Листинг 3.1 – Алгоритм линейной обработки текста

```
void parse_linear(int count, std::vector<std::string> filenames,
    bool is_print)
{
    time_now = 0;
    std::queue<std::string> q1;
    std::queue<std::string> q2;
    std::queue<std::vector<std::string>> q3;
    queues_t queues = {.q1 = q1, .q2 = q2, .q3 = q3};
    for (int i = 0; i < count; i++)
    {
        queues.q1.push(filenames[i]);
        std::string filename = queues.q1.front();
        stage1_linear(filename, i + 1, is_print);
        queues.q1.pop();
        queues.q2.push(filename);
        std::string text = queues.q2.front();
        stage2_linear(filename, i + 1, is_print); // Stage 2
        queues.q2.pop();
        queues.q3.push(filename);
        std::vector<std::string> sentences = queues.q3.front();
        stage3_linear(filename, i + 1, is_print); // Stage 3
        queues.q3.pop();
    }
}
```

Листинг 3.2 – Алгоритм конвейерной обработки текста

```
1  void parse_parallel(int count, std::vector<std::string> filenames,
       bool is_print)
2  {
3      time_now = 0;
4      std::queue<matrix_t> q1;
5      std::queue<matrix_t> q2;
6      std::queue<matrix_t> q3;
7      queues_t queues = {.q1 = q1, .q2 = q2, .q3 = q3};
8      for (int i = 0; i < count; i++)
9      {
10         q1.push(filenames[i]);
11     }
12     std::thread threads[THREADS];
13     threads[0] = std::thread(stage1_parallel, std::ref(q1),
           std::ref(q2), std::ref(q3), is_print);
14     threads[1] = std::thread(stage2_parallel, std::ref(q1),
           std::ref(q2), std::ref(q3), is_print);
15     threads[2] = std::thread(stage3_parallel, std::ref(q1),
           std::ref(q2), std::ref(q3), is_print);
16     for (int i = 0; i < THREADS; i++)
17     {
18         threads[i].join();
19     }
20 }
```

Листинг 3.3 – Алгоритм запуска 1 потока для чтения текста из файла

```
void stage1_parallel(std::queue<std::string> &q1,
    std::queue<std__string> &q2,
    std::queue<std::vector<std::string>> &q3, bool is_print)
{
     int task_num = 1;
     std::mutex m;
     while(!q1.empty())
     {
          m.lock();
          std::string filename = q1.front();
          m.unlock();
          log(filename, task_num++, 1, read_file, is_print);
          m.lock();
          q2.push(filename);
          q1.pop();
          m.unlock();
     }
}
```

Листинг 3.4 – Алгоритм запуска 3 потока для построения деревьев синтаксических зависимостей

```
void stage3_parallel(std::queue<std::string> &q1,
    std::queue<std::string> &q2,
    std::queue<std::vector<std::string>> &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();
               std::vector<std::string> sentences = q3.front();
               q3.pop();
               m.unlock();
               log(filename, task_num++, 3, tree, is_print);
          }
     } while (!q1.empty() || !q2.empty() || !q3.empty());
}
```

Листинг 3.5 – Алгоритм запуска 2 потока для разделения текста на предложения

```
void stage2_parallel(std::queue<std::string> &q1,
    std::queue<std::string> &q2,
    std::queue<std::vector<std::string>> &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();
               std::string filename = q2.front();
               m.unlock();
               log(filename, task_num++, 2, get_sentences, is_print);
               m.lock();
               q3.push(filename);
               q2.pop();
               m.unlock();
          }
     } while (!q1.empty() || !q2.empty());
}
```

3.3 Сведения о модулях программы

Программа состоит из следующих модулей:

- *main.cpp* — файл, содержащий меню программы;
- *conveyor.h* и *conveyor.cpp* — файлы, содержащие код реализации линейной и конвейерной обработок, а также функции замера времени;
- *graph_build.py* — файл, содержащий функции построения графиков для замеров по времени.

3.4 Вывод

Были представлены листинги всех алгоритмов линейной и конвейерной обработки текста. Также в данном разделе была приведена информация о выбранных средствах для разработки алгоритмов и сведения о модулях программы.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программы, а также проведен сравнительный анализ алгоритмов при различных ситуациях на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялся эксперимент представлены далее:

1) операционная система — Ubuntu 22.04.3 [5] Linux x86_64;

2) память — 16 Гб;

3) процессор — Intel® Core™ i5-1135G7 @ 2.40 ГГц.

При эксперименте ноутбук не был включен в сеть электропитания.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы при линейной обработке, а на рисунке 4.2 – при конвейерной.

```
Количество: 9
Task:   1, Tape:   1, Start: 0.000000, End: 0.000010
Task:   1, Tape:   2, Start: 0.000010, End: 0.000011
Task:   1, Tape:   3, Start: 0.000011, End: 1.599800
Task:   2, Tape:   1, Start: 1.599800, End: 1.599813
Task:   2, Tape:   2, Start: 1.599813, End: 1.599815
Task:   2, Tape:   3, Start: 1.599815, End: 3.164765
Task:   3, Tape:   1, Start: 3.164765, End: 3.164778
Task:   3, Tape:   2, Start: 3.164778, End: 3.164779
Task:   3, Tape:   3, Start: 3.164779, End: 4.724940
Task:   4, Tape:   1, Start: 4.724940, End: 4.724951
Task:   4, Tape:   2, Start: 4.724951, End: 4.724953
Task:   4, Tape:   3, Start: 4.724953, End: 6.258220
Task:   5, Tape:   1, Start: 6.258220, End: 6.258233
Task:   5, Tape:   2, Start: 6.258233, End: 6.258234
Task:   5, Tape:   3, Start: 6.258234, End: 7.762281
Task:   6, Tape:   1, Start: 7.762281, End: 7.762293
Task:   6, Tape:   2, Start: 7.762293, End: 7.762295
Task:   6, Tape:   3, Start: 7.762295, End: 9.252547
Task:   7, Tape:   1, Start: 9.252547, End: 9.252559
Task:   7, Tape:   2, Start: 9.252559, End: 9.252560
Task:   7, Tape:   3, Start: 9.252560, End: 10.769866
Task:   8, Tape:   1, Start: 10.769866, End: 10.769877
Task:   8, Tape:   2, Start: 10.769877, End: 10.769879
Task:   8, Tape:   3, Start: 10.769879, End: 12.281115
Task:   9, Tape:   1, Start: 12.281115, End: 12.281128
Task:   9, Tape:   2, Start: 12.281128, End: 12.281130
Task:   9, Tape:   3, Start: 12.281130, End: 13.772820
```

Рисунок 4.1 – Пример работы программы (линейная)

```
Количество: 9
Task:   1, Tape:   1, Start: 0.000000, End: 0.000016
Task:   1, Tape:   2, Start: 0.000016, End: 0.000018
Task:   2, Tape:   1, Start: 0.000016, End: 0.000043
Task:   1, Tape:   3, Start: 0.000018, End: 1.523812
Task:   2, Tape:   2, Start: 1.523824, End: 1.523826
Task:   3, Tape:   1, Start: 1.523812, End: 1.524312
Task:   2, Tape:   3, Start: 1.523826, End: 3.032673
Task:   3, Tape:   2, Start: 2.032685, End: 2.032687
Task:   4, Tape:   1, Start: 2.032687, End: 2.035466
Task:   4, Tape:   2, Start: 2.035466, End: 2.035765
Task:   3, Tape:   3, Start: 3.032687, End: 4.524449
Task:   4, Tape:   3, Start: 4.524463, End: 5.996318
Task:   5, Tape:   1, Start: 5.996318, End: 5.996330
Task:   5, Tape:   2, Start: 5.996330, End: 5.996333
Task:   6, Tape:   1, Start: 6.108119, End: 6.108131
Task:   5, Tape:   3, Start: 5.996333, End: 7.508119
Task:   7, Tape:   1, Start: 6.055649, End: 6.055660
Task:   6, Tape:   2, Start: 6.508131, End: 6.508133
Task:   6, Tape:   3, Start: 6.508133, End: 7.955649
Task:   7, Tape:   2, Start: 9.055660, End: 9.055662
Task:   8, Tape:   1, Start: 8.894045, End: 8.894060
Task:   7, Tape:   3, Start: 9.055662, End: 10.594045
Task:   8, Tape:   2, Start: 9.594060, End: 9.594062
Task:   9, Tape:   1, Start: 10.133392, End: 10.133403
Task:   9, Tape:   2, Start: 10.133403, End: 10.133405
Task:   8, Tape:   3, Start: 9.594062, End: 11.133392
Task:   9, Tape:   3, Start: 11.133405, End: 12.544291
```

Рисунок 4.2 – Пример работы программы (конвейерная)

4.3 Время выполнения алгоритмов

Как было сказано выше, используется функция замера процессорного времени *std::chrono::system_clock::now(...)* из библиотеки *chrono* на C++. Функция возвращает процессорное время типа float в секундах.

Функция используется дважды: перед началом выполнения алгоритма и после завершения, затем из конечного времени вычитается начальное, чтобы получить результат.

Замеры проводились для разного размера файлов, а также для разного количества файлов, чтобы определить, когда наиболее эффективно использовать конвейерную обработку.

Результаты замеров приведены в таблицах 4.1-4.4 (время в мс).

Таблица 4.1 – Результаты замеров времени (линейная, разное количество файлов)

Кол-во матриц	Время
1	1.5238
2	3.0126
3	4.4944
4	5.9263
5	7.4181
6	8.9156
7	10.2940
8	11.9333
9	13.2111

Таблица 4.2 – Результаты замеров времени (конвейерная, разное количество файлов)

Кол-во матриц	Время
1	1.1238
2	2.2126
3	3.4932
4	4.9264
5	6.1183
6	7.2151
7	8.4947
8	9.4323
9	10.1131

Таблица 4.3 – Результаты замеров времени (линейная, разные размеры файлов)

Размер матриц	Время
100	1.5238
200	2.2589
300	3.3409
400	4.3925
500	5.5193
600	6.7832
700	7.6820
800	8.9341
900	9.2402

Таблица 4.4 – Результаты замеров времени (конвейерная, разные размеры файлов)

Размер	Время
100	1.1238
200	1.5573
300	1.9623
400	2.8435
500	3.6133
600	4.6842
700	5.3834
800	6.4346
900	7.1013

Также на рисунках 4.3–4.4 приведены графические результаты замеров.

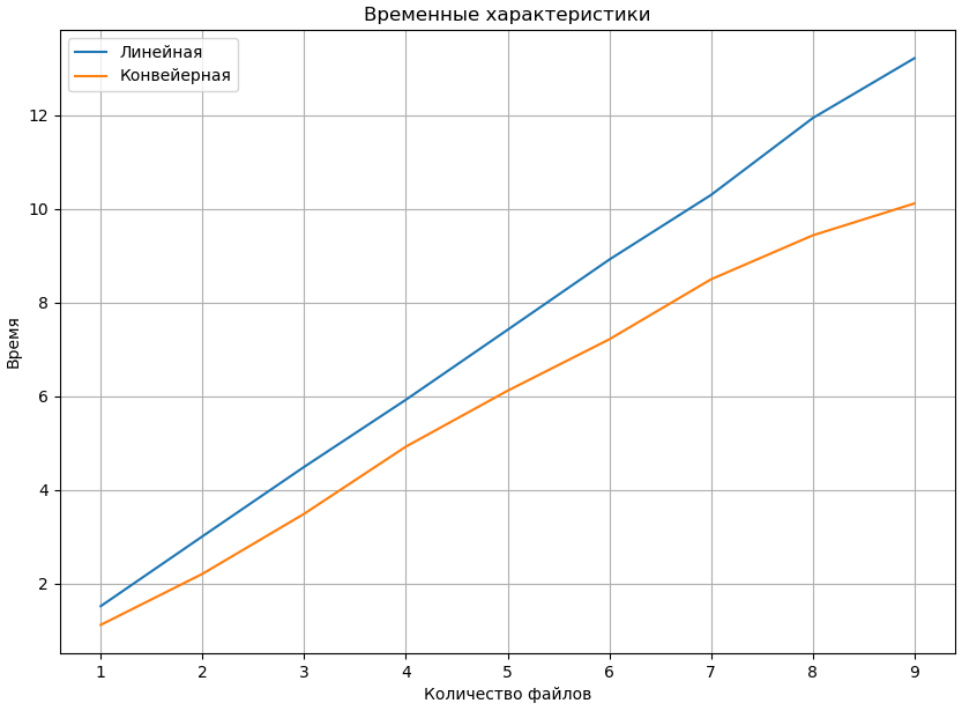

Рисунок 4.3 – Сравнение по времени линейной и конвейерной обработок для разного количества файлов

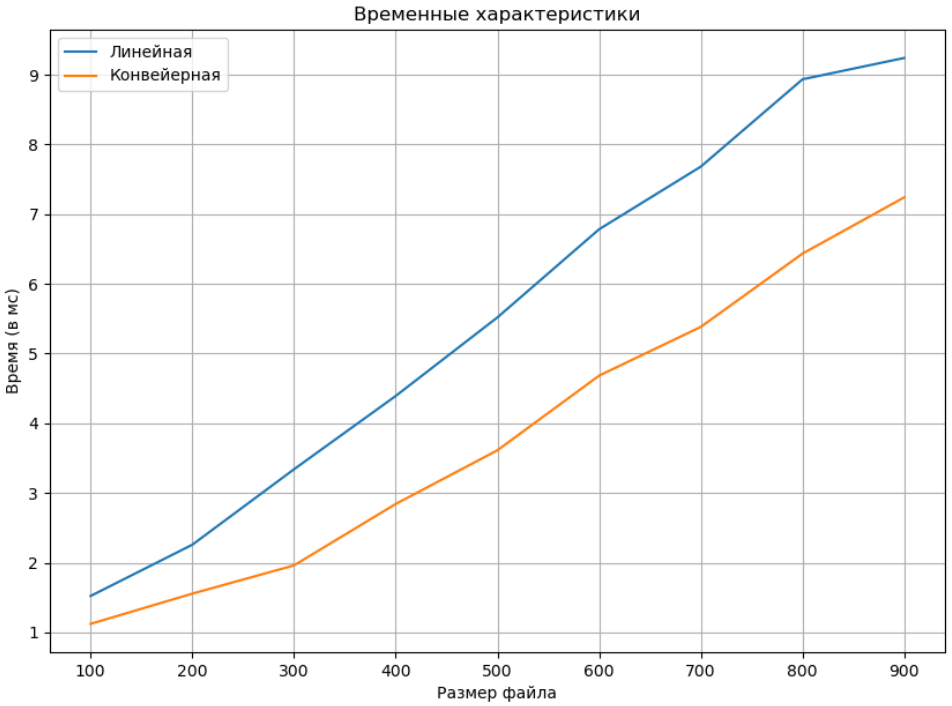

Рисунок 4.4 – Сравнение по времени линейной и конвейерной обработок для разных размеров файлов

4.4 Вывод

В результате эксперимента было получено, что при использовании конвейерной обработки время выполенения меньше, чем при линейной реализации при количестве файлов, равном 4, в 1.2 раза, а при количестве файлов, равном 9, уже в 1.3 раза. Следовательно, конвейерная реализация лучше линейной при увеличении количества задач (файлов).

Также при проведении эксперимента было выявлено, что при увеличении размера файла конвейерная реализация выдает лучшие результаты. Так, при размере файла в 100 кб конвейерная реализация быстрее в 1.2 раза, чем линейная, а при размере файла, равном 900 кб, в 1.3 раза.

Заключение

Результаты эксперимента показали, что при конвейерной обработке время выполнения показывает лучшие результаты по сравнению с линейной реализацией. При количестве файлов, равном 4, время выполнения конвейерной реализации оказалось в 1.2 раза меньше, а при количестве файлов, равном 9, — в 1.3 раза меньше. Это свидетельствует о преимуществе конвейерной реализации при увеличении количества задач (файлов).

Также при увеличении размера файла обнаружено, что конвейерная реализация продемонстрировала более высокую производительность. Например, при размере файла 100 кб время выполнения конвейерной реализации оказалось в 1.2 раза меньше, чем линейной реализации, а при размере файла 900 кб — уже в 1.3 раза меньше.

Цель, которая была поставлена в начале лабораторной работы была достигнута, а также в ходе выполнения лабораторной работы были решены следующие задачи:

- описаны основы конвейерной обработки данных и алгоритма построения дерева синтаксических зависимостей;
- приведены схемы конвейерной и линейной обработок;
- реализованы разработанные алгоритмы;
- проведен сравнительный анализ по времени для реализованных алгоритмов;
- подготовлен отчет о выполненной лабораторной работе.

Список используемых источников

[1] Конвейерная обработка данных [Электронный ресурс]. Режим доступа: https://studref.com/636041/ekonomika/konveyernaya_obrabotka_dannyh (дата обращения: 23.10.2021).

[2] Программирование на C/C++ [Электронный ресурс]. Режим доступа: http://www.c-cpp.ru/ (дата обращения: 23.10.2021).

[3] Welcome to Python [Электронный ресурс]. Режим доступа: https://www.python.org (дата обращения: 23.10.2021).

[4] Date and time utilities [Электронный ресурс]. Режим доступа: https://en.cppreference.com/w/cpp/chrono (дата обращения: 23.10.2021).

[5] Ubuntu 20.04.3 LTS (Focal Fossa) [Электронный ресурс]. Режим доступа: https://releases.ubuntu.com/20.04/ (дата обращения: 15.10.2021).

[6] Linux [Электронный ресурс]. Режим доступа: https://www.linux.org/forums/#linux-tutorials.122 (дата обращения: 15.10.2021).

[7] Процессор Intel® Core™ i5-7300HQ [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/97456/intel-core-i5-7300hq-processor-6m-cache-up-to-3-50-ghz.html (дата обращения: 04.10.2021).