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 Пронина Л.Ю.

Группа ИУ7-54Б

Оценка (баллы)

Преподаватель Волкова Л. Л.

Москва — 2023 г.

Содержание

Введение

Многопоточность является одной из важных концепций в разработке программного обеспечения. Она позволяет выполнять несколько задач параллельно, увеличивая эффективность работы приложений и улучшая отзывчивость пользовательского интерфейса. В современном мире, где все больше задач требует обработки больших объемов данных или выполнения сложных вычислений, эффективное использование многопоточности становится критически важным.

Целью данной работы является описание параллельных вычислений на основе построения деревьев синтаксических зависимостей текстов при помощи библиотеки синтаксического анализа (spacy).

Для достижения поставленной цели необходимо выполнить следующие задачи:

1) описать основы многопоточного программирования;

2) реализовать алгоритм построения деревьев синтаксических зависимостей в тексте с использованием многопоточности и без;

3) провести сравнительный анализ реализаций рассматриваемого алгоритма по времени при разном размере текста без использования многопоточности и с использованием разного количества потоков;

4) описать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

В этом разделе будет представлена информация по поводу многопоточности и деревьев синтаксических зависимостей в тексте, построение которых будет распараллелено в данной лабораторной работе.

1.1 Многопоточность

Многопоточность — это возможность процессора выполнять несколько потоков одновременно в рамках использования ресурсов одного процессора. Поток представляет собой последовательность инструкций, которые могут быть выполнены параллельно с другими потоками в рамках одного процесса [1].

Процесс — это программа в состоянии выполнения. Когда программа или приложение запускается, создается процесс. Процесс может состоять из одного или нескольких потоков. Каждый поток представляет собой сегмент процесса, который выполняет задачи, стоящие перед приложением. Процесс завершается, когда все его потоки завершают свою работу.

В многопоточных программах необходимо учитывать, что если потоки запускаются последовательно и передают управление друг другу, то не получится полностью использовать потенциал многопоточности и получить выигрыш от параллельной обработки задач. Чтобы достичь максимальной эффективности, потоки должны выполняться параллельно, особенно для независимых по данным задач.

Одной из проблем, возникающих при использовании многопоточности, является совместный доступ к данным. Возникает конфликт, когда два или более потоков пытаются записать в одну и ту же ячейку памяти одновременно. Для решения этой проблемы используется механизм синхронизации доступа к данным, такой как мьютекс (mutex). Мьютекс позволяет одному потоку работать с данными в монопольном режиме, пока другие потоки ожидают освобождения мьютекса.

Критическая секция — это набор инструкций, выполняемых между захватом и освобождением мьютекса. Во время захвата мьютекса другие потоки, которым требуется доступ к общим данным, должны ждать его осво-

бождения. Поэтому необходимо минимизировать объем кода в критической секции, чтобы другие потоки могли получить к ней доступ как можно быстрее.

Таким образом, использование многопоточности в программировании требует хорошего понимания концепций синхронизации и разработки эффективного кода для доступа к общим данным, чтобы достичь максимальной производительности и избежать состояний гонки.

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.

Дерево синтаксических зависимостей полезно для понимания семантики предложения, разрешения синтаксической структуры, а также для выполнения различных задач в области обработки естественного языка, таких как

извлечение информации, ответы на вопросы и машинный перевод.

Библиотека spacy предоставляет мощные инструменты для анализа синтаксических зависимостей и визуализации деревьев синтаксических зависимостей, которые помогают исследователям и разработчикам в понимании структуры текста и автоматическом анализе естественного языка.

Вывод

В данном разделе были рассмотрены понятия многопоточности и деревьев синтаксических зависимостей в тексте.

2 Конструкторская часть

В этом разделе будут представлены схемы алгоритмов построения деревьев синтаксических зависимостей в тексте с распараллеливанием и без него.

2.1 Требования к программному обеспечению

К программе предъявлен ряд требований:

1) иметь интерфейс для выбора действий;
2) работать с «нативными» потоками;
3) замерять процессорное время работы реализаций алгоритмов.

2.2 Разработка алгоритмов

На рисунках 2.1, 2.2 и 2.3 представлены схемы алгоритмов построения дерева синтаксических зависимостей в тексте без многопоточности и с ней соответственно, а также схема алгоритма одного рабочего потока для варианта с многопоточностью.

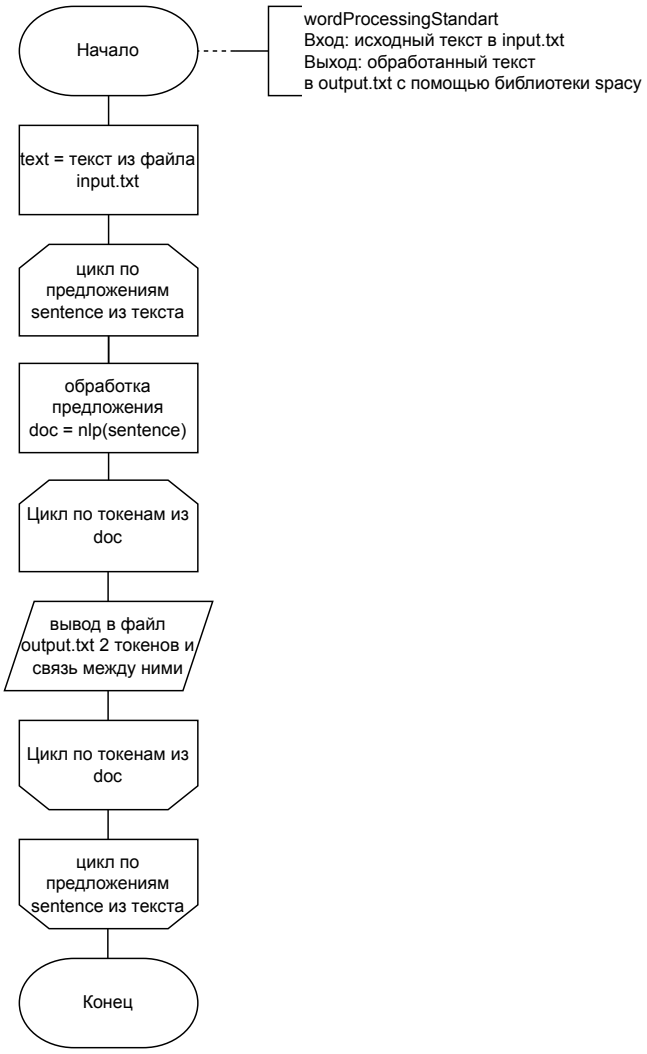

Рисунок 2.1 – Схема алгоритма построения деревьев синтаксических зависимостей в тексте (без многопоточности)

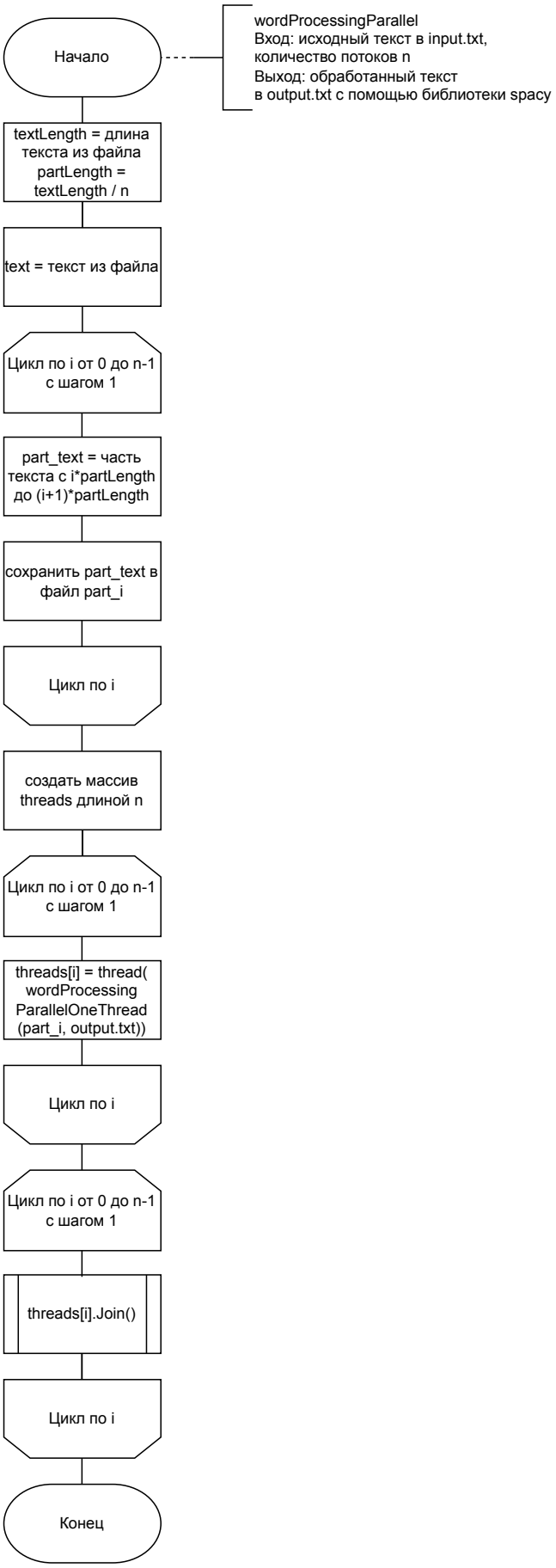

Рисунок 2.2 – Схема алгоритма построения деревьев синтаксических зависимостей в тексте (с многопоточностью)

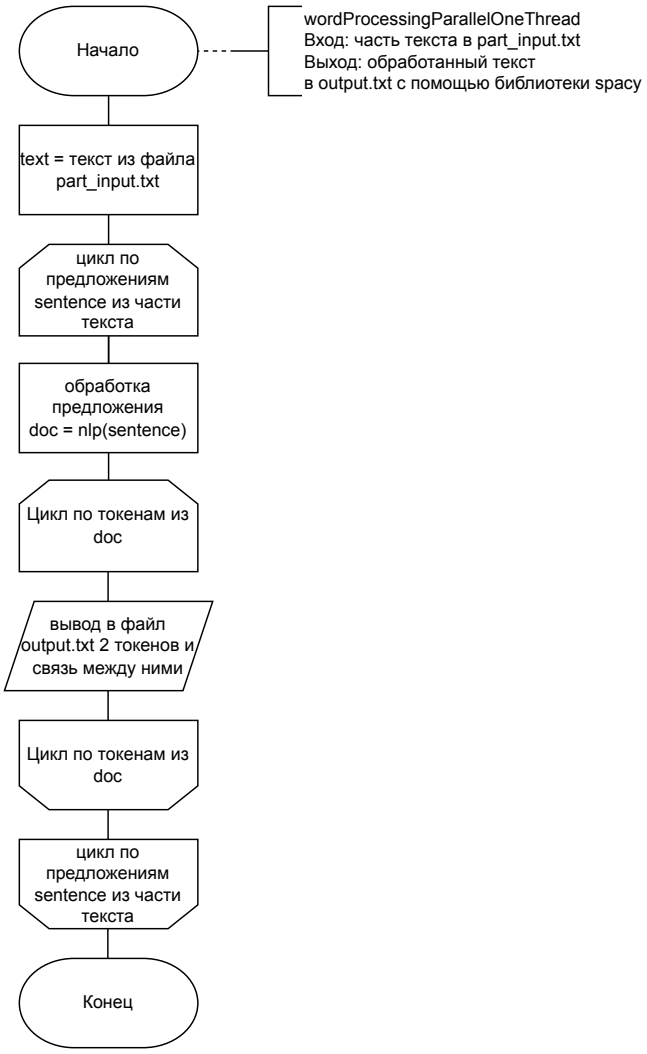

Рисунок 2.3 – Схема алгоритма одного рабочего потока для варианта с многопоточностью

Вывод

В данном разделе были представлены схемы алгоритмов построения деревьев синтаксических зависимостей в тексте с распараллеливанием и без него.

3 Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги алгоритмов построения деревьев синтаксических зависимостей в тексте с распараллеливанием и без него.

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.

Построение деревьев синтаксических зависимостей с помощью библиотеки *spacy* и графиков для анализа времени были реализованы с использованием языка программирования *Python* [3].

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

3.2 Листинги кода

В листинге 3.2 представлена реализация алгоритма построения деревьев синтаксических зависимостей с чтением исходного текста из файла и сохранением результата в файл. Реализация алгоритма построения деревьев синтаксических зависимостей без многопоточности представлена в листинге 3.1, а с многопоточностью в приложении.

Листинг 3.1 – Алгоритм построения деревьев синтаксических зависимостей без многопоточности

```
void exec_no_parallel(QString size)
{
    std::ifstream inputFile("data/input_" + size.toStdString() +
       "kb.txt");
    std::string text;
    std::string line;
    while (std::getline(inputFile, line))
        text += line + " ";
    std::vector<std::string> sentences;
    std::string delimiter = ".!?";
    size_t pos = 0;
    std::string token;
    while (true) {
        pos = text.find_first_of(delimiter);
        if (pos == std::string::npos)
            break;
        token = text.substr(0, pos+1);
        sentences.push_back(token);
        text.erase(0, pos+1);
    }
    if (!text.empty())
        sentences.push_back(text);
    std::string filename_prep = "./data/prep/no_" +
       size.toStdString() + ".txt";
    std::ofstream outputFile(filename_prep);
    if (!outputFile.is_open()) {
        std::cout << "Failed to open output file" <<  std::endl;
        return;
    }
    for (const auto& sentence : sentences)
        outputFile << sentence << std::endl;
    outputFile.close();
    std::string command = "python3 main.py " + filename_prep + "
       ./result/no_" + size.toStdString() + "output.txt";
    system(command.c_str());
}
```

Листинг 3.2 – Алгоритм построения деревьев синтаксических зависимостей с помощью библиотеки spacy

```
import spacy
import sys
nlp = spacy.load("ru_core_news_sm")
with open(sys.argv[1], "r") as file:
    while (1):
        sentence = file.readline()
        if (len(sentence) == 0):
            break
        doc = nlp(sentence)
        dependency_tree = []
        for token in doc:
            dependency_tree.append(token.text + "␣" + token.dep_ +
               "␣" + token.head.text)
        with open(sys.argv[2], "a") as file_output:
            file_output.write("\n".join(dependency_tree))
            file_output.write("\n\n")
```

3.3 Сведения о модулях программы

Программа состоит из следующих модулей:

1) *main.cpp* — файл, содержащий функцию, вызывающую интерфейс программы;

2) *mainwindow.h* и *mainwindow.cpp* — файлы, содержащие код всех методов, реализующих интерфейс программы и взаимодействующие с ним;

3) *building.h* и *building.cpp* — файлы, содержащие код реализаций алгоритмов построения деревьев синтаксических зависимостей с многопоточностью и без нее, а также функции замера времени;

4) *graph.py* — файл, содержащий функции построения графиков для замеров по времени;

5) *main.py* — файл, содержащий само использование библиотеки *spacy* для построения деревьев синтаксических зависимостей.

Вывод

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

4 Исследовательская часть

В данном разделе будет приведен пример 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 — четырехъядерный 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 интерфейс программы.

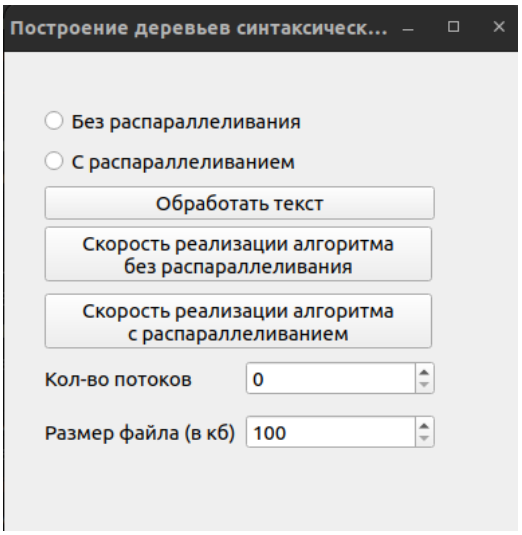

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения реализаций 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 файлов с текстом размером от 100 до 1000 кб, без дополнительных потоков и с потоками, количество которых от 1 до 32.

Результаты замеров приведены в таблицах 4.1 и 4.2 (время в с). А на рисунках 4.2 и 4.3 приведены графические результаты замеров.

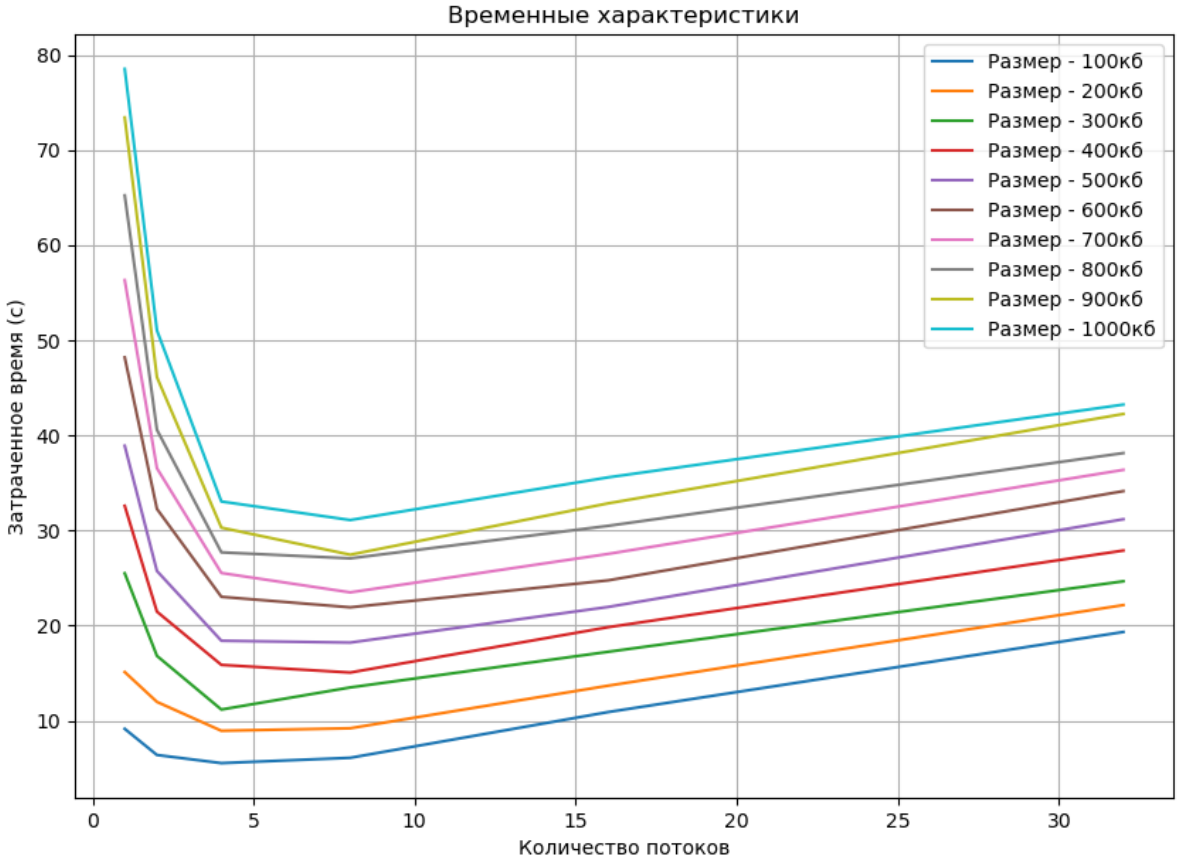

Рисунок 4.2 – Использование разного количества потоков

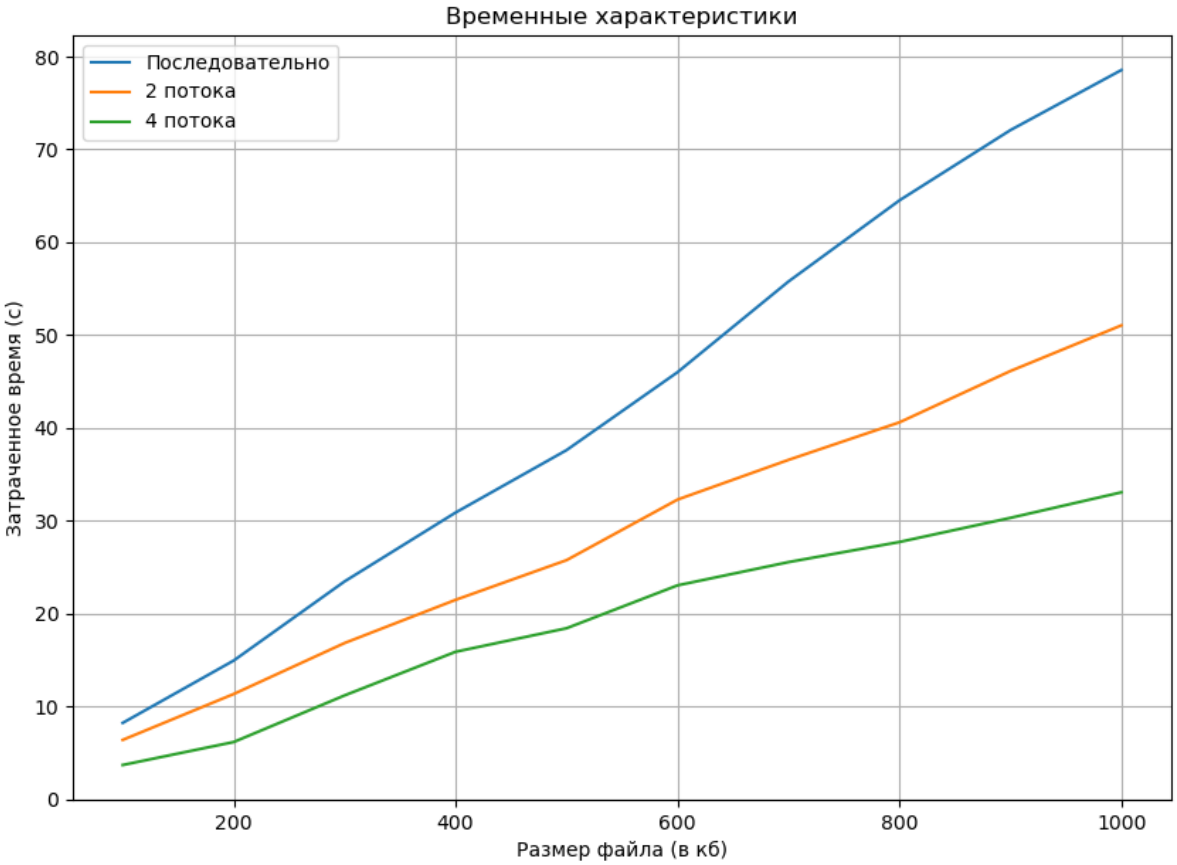

Рисунок 4.3 – Без использования многопоточности и с 2 и 4 потоками

Таблица 4.1 – Результаты замеров времени (в с) для разного размера файла и разного количества потоков

Размер файла (в кб)	Количество потоков					
	1	2	4	8	16	32
100	9.15	6.40	5.54	6.10	10.91	19.33
200	15.12	11.98	8.94	9.21	13.68	22.16
300	25.50	16.82	11.18	13.50	17.25	24.66
400	32.59	21.47	15.87	15.07	19.83	27.90
500	38.91	25.75	18.41	18.22	21.96	31.19
600	48.21	32.27	23.03	21.93	24.76	34.15
700	56.33	36.55	25.54	23.50	27.54	36.38
800	65.24	40.59	27.71	27.07	30.49	38.15
900	73.43	46.12	30.30	27.46	32.85	42.26
1000	78.54	51.04	33.05	31.10	35.59	43.24

Таблица 4.2 – Результаты замеров времени без многопоточности

Размер файла (в кб)	Время (в с)
100	8.22
200	14.93
300	23.47
400	30.88
500	37.61
600	46.00
700	55.77
800	64.51
900	72.08
1000	78.55

Вывод

В результате эксперимента было получено, что при использовании 4 потоков, время работы многопоточной реализации алгоритма построения деревьев синтаксических зависимостей в тексте меньше, чем время работы реализации без многопоточности в 2.04 раз на размере файла, равном 500 кб. Данное количество потоков обусловлено тем, что на ноутбуке, на котором проводилось тестирование ПО, имеется всего 4 логических ядра. Также важную роль играет то, что работа распределяется примерно равно между всеми потоками, остатки работы отдаются крайнему потоку. При 4 потоках остатка не будет. Именно поэтому лучшие результаты достигаются именно на 4 потоках, даже не смотря на ресурсы, которые дополнительно затрачиваются на содержание потоков. В итоге, можно сказать, что при таких данных следует использовать многопоточную реализацию алгоритма.

Также при проведении эксперимента было выявлено, что при увеличении размера файла, многопоточная реализация выдает все более заметное преимущество по времени перед реализацией без многопоточностью. Так, при размере файла, равном 300 кб многопоточная реализация быстрее реализации без многопоточности в 2.09 раз, а на размере файла, равном 1000 кб — в 2.37 раз.

Кроме этого, из эксперимента следует, что реализация с использованием 8 дополнительных потоков работает быстрее, чем с 4 дополнительными потоками при размере файла более 500 кб.

Заключение

В результате эксперимента было обнаружено, что при использовании 4 потоков скорость многопоточной реализации алгоритма построения деревьев синтаксических зависимостей в тексте превосходит скорость однопоточной реализации на размере файла, равном 500 кб, примерно в 2.04 раз. Количество использованных потоков соответствует количеству логических ядер на испытуемом ноутбуке (4 ядра). Распределение работы между всеми потоками примерно одинаково, что позволяет достичь лучших результатов на 4 потоках и устранить недоработки, которые возникают при использовании меньшего числа потоков. Это означает, что многопоточная реализация алгоритма представляет собой предпочтительный вариант при обработке таких данных.

Кроме того, в ходе эксперимента было выявлено, что при увеличении размера файла многопоточная реализация демонстрирует еще более заметные преимущества. Например, при размере файла в 300 кб реализация с 4 потоками работает быстрее однопоточной реализации примерно в 2.09 раз, а при размере файла в 1000 кб — в 2.37 раз.

Также интересным результатом эксперимента стало то, что реализация с использованием 8 дополнительных потоков оказалась эффективнее по времени в сравнении с использованием 4 дополнительных потоков при размере файла более 500 кб. Это может быть связано с некоторыми ограничениями системы и показывает, что оптимальное количество потоков может зависеть от конкретных условий.

В ходе лабораторной работы были выполнены следующие задачи:

1) описаны основы многопоточного программирования;

2) реализованы алгоритмы построения деревьев синтаксических зависимостей в тексте с использованием многопоточности и без;

3) проведен сравнительный анализ реализаций рассматриваемого алгоритма по времени при разном размере текста без использования многопоточности и с использованием разного количества потоков;

4) описаны и обоснованы полученные результаты в отчете о выполненной лабораторной работе.

Таким образом, все задачи лабораторной работы были выполнены и, следовательно, поставленная цель была достигнута.

Список используемых источников

1. Stoltzfus Justin. Multithreading. [Электронный ресурс]. Режим доступа: https://www.techopedia.com/definition/24297/multithreading-computer-architecture (дата обращения: 29.11.2023).

2. Программирование на C/C++ [Электронный ресурс]. Режим доступа: `http://www.c-cpp.ru/` (дата обращения: 29.11.2023).

3. Welcome to Python [Электронный ресурс]. Режим доступа: `https://www.python.org` (дата обращения: 29.11.2023).

4. Date and time utilities [Электронный ресурс]. Режим доступа: `https://en.cppreference.com/w/cpp/chrono` (дата обращения: 29.11.2023).

5. Ubuntu 20.04.3 LTS (Focal Fossa) [Электронный ресурс]. Режим доступа: `https://releases.ubuntu.com/20.04/` (дата обращения: 29.11.2023).

Приложение А

В листинге 1 представлена реализация алгоритма построения деревьев синтаксических зависимостей с многопоточностью.

Листинг 1 – Алгоритм построения деревьев синтаксических зависимостей с многопоточностью

```
void execute_one_thread(std::string filename_prep, std::string
    file_output) {
    std::string command = "python3 main.py " + filename_prep + " "
        + file_output;
    system(command.c_str());
}
void exec_parallel(QString size, int threads_count)
{
    std::ifstream inputFile("data/input_" + size.toStdString() +
        "kb.txt");
    std::string line;
    std::string text;
    while (std::getline(inputFile, line))
    text += line + " ";
    std::vector<std::string> sentences;
    std::string delimiter = ".!?";
    size_t pos = 0;
    std::string token;
    while (true) {
        pos = text.find_first_of(delimiter);
        if (pos == std::string::npos)
        break;
        token = text.substr(0, pos+1);
        sentences.push_back(token);
        text.erase(0, pos+1);
    }
    if (!text.empty())
    sentences.push_back(text);
    int sentences_count = sentences.size();
    int part_count = sentences_count / threads_count;
    for (int i = 0; i < threads_count; ++i) {
        std::string filename_prep = "./data/prep/" +
            size.toStdString() + "_" +
            QString::number(i).toStdString() + ".txt";
```

```
30          std::ofstream outputFile(filename_prep);
31          if (!outputFile.is_open()) {
32              std::cout << "Failed to open output file" << std::endl;
33              return;
34          }
35          for (int j = 0; j < part_count; ++j)
36          outputFile << sentences[i * part_count + j] << std::endl;
37          outputFile.close();
38      }
39      std::vector<std::thread> threads;
40      for (int i = 0; i < threads_count; ++i) {
41          std::string file_output = "./result/" + size.toStdString()
              + "output.txt";
42          std::string filename_prep = "./data/prep/" +
              size.toStdString() + "_" +
              QString::number(i).toStdString() + ".txt";
43          threads.push_back(std::thread(execute_one_thread,
              filename_prep, file_output));
44      }
45      for (std::thread& t : threads)
46      t.join();
47  }
```