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»

на тему: «Параллельные вычисления на основе нативных потоков»

Студент ИУ7-53Б _____ Лысцев Н. Д.
(Группа) (Подпись, дата) (И. О. Фамилия)

Преподаватель _____ Волкова Л. Л.
(Подпись, дата) (И. О. Фамилия)

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Многопоточность — свойство кода программы выполняться параллельно (одновременно) на нескольких ядрах процессора или псевдопараллельно на одном ядре (каждый поток получает в свое распоряжение некоторое время, за которое он успевает исполнить часть своего кода на процессоре) [1].

Поток (thread) представляет собой независимую последовательность инструкций в программе. В приложениях, которые имеют пользовательский интерфейс, всегда есть как минимум один главный поток, который отвечает за состояние компонентов интерфейса. Кроме него в программе может создаваться множество независимых дочерних потоков, которые будут выполняться независимо [1].

Целью данной лабораторной работы является изучение принципов и получение навыков организации параллельного выполнения операций.

Для достижения поставленной цели необходимо решить следующие задачи:

1) изучить и описать алгоритм составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке;

2) разработать последовательную и параллельную версии данного алгоритма;

3) реализовать каждую версию алгоритма;

4) провести сравнительный анализ алгоритмов по времени работы реализаций;

5) обосновать полученные результаты в отчете к выполненной лабораторной работе.

1 Аналитический раздел

В данном разделе будет приведено теоретическое описание последовательной и параллельной версии алгоритма составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке.

1.1 N-граммы

Пусть задан некоторый конечный алфавит

$$V = w_i \tag{1.1}$$

где w_i — символ.

Языком $L(V)$ называют множество цепочек конечной длины из символов w_i. N-граммой на алфавите V (1.1) называют произвольную цепочку из $L(V)$ длиной N, например последовательность из N букв русского языка одного слова, одной фразы, одного текста или, в более интересном случае, последовательность из грамматически допустимых описаний N подряд стоящих слов [2].

1.2 Последовательная версия алгоритма

Алгоритм составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке состоит из следующих шагов:

1) считывание текста в массив строк;
2) преобразование считанного текста (перевод букв в нижний регистр, удаление знаков препинания);
3) обработка каждой строки текста.

Обработка строки текста включает следующие шаги:

1) обработка каждого слова из строки текста;
2) выделение существующих в этом слове N-грамм;
3) увеличение количества выделенных N-грамм в словаре.

1.3 Параллельная версия алгоритма

В алгоритме составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке обработка строк текста происходит независимо, поэтому есть возможность произвести распараллеливание данных вычислений.

Для этого строки текста поровну распределяются между потоками. Каждый поток получает локальную копию словаря для N-грамм, производит вычисления над своим набором строк и после завершения работы всех потоков словари с количеством употреблений N-грамм каждого потока объединяются в один. Так как каждая строка массива передается в монопольное использование каждому потоку, не возникает конфликтов доступа к разделяемым ячейки памяти,следовательно, в использовании средства синхронизации в виде мьютекса нет необходимости.

Вывод

В данном разделе было приведено теоретическое описание последовательной и параллельной версии алгоритма составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке.

2 Конструкторский раздел

2.1 Разработка алгоритмов

2.1.1 Последовательная версия алгоритма

На рисунках 2.1 и 2.2 представлена схема последовательной версии алгоритма составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке.

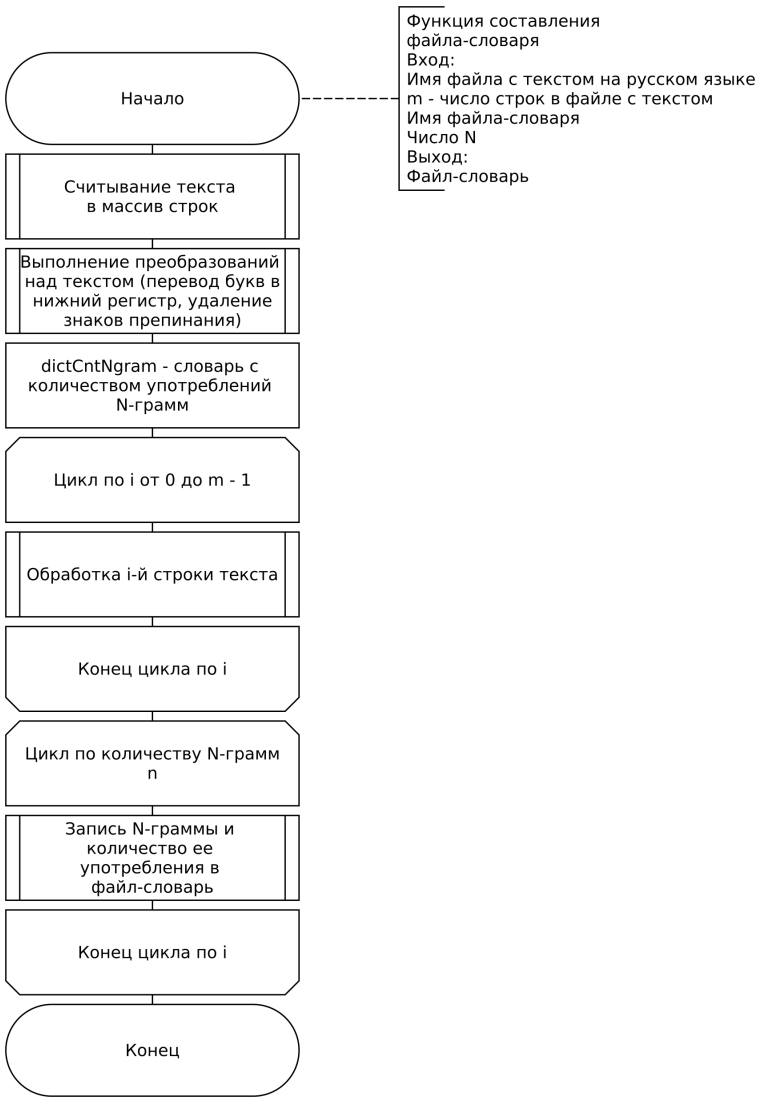

Рисунок 2.1 – Схема алгоритма для обработки целого текста

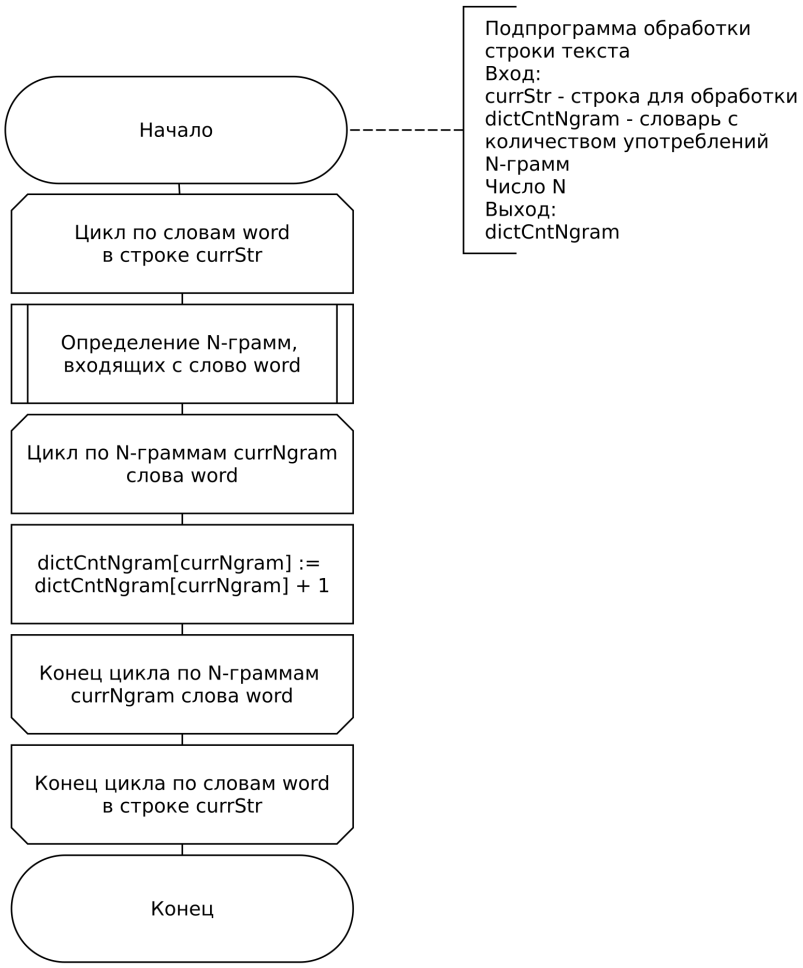

Рисунок 2.2 – Схема алгоритма для обработки строки текста

2.1.2 Параллельная версия алгоритма

На рисунках 2.3 и 2.4 представлена схема параллельной версии алгоритма составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке.

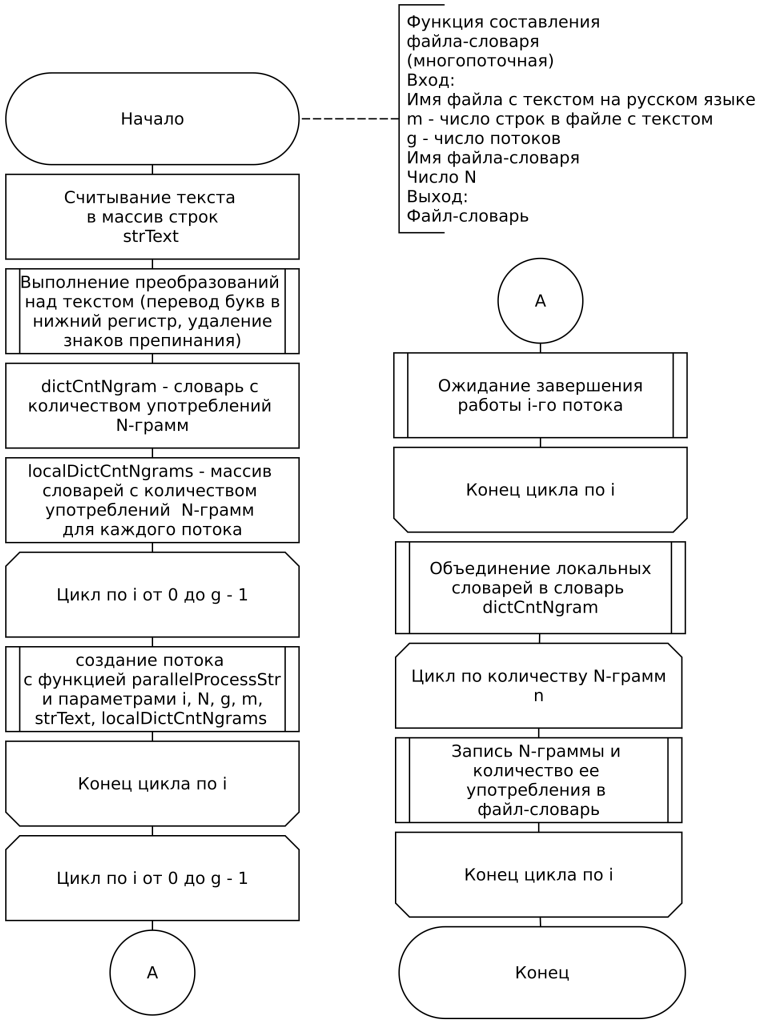

Рисунок 2.3 – Схема алгоритма для обработки целого текста

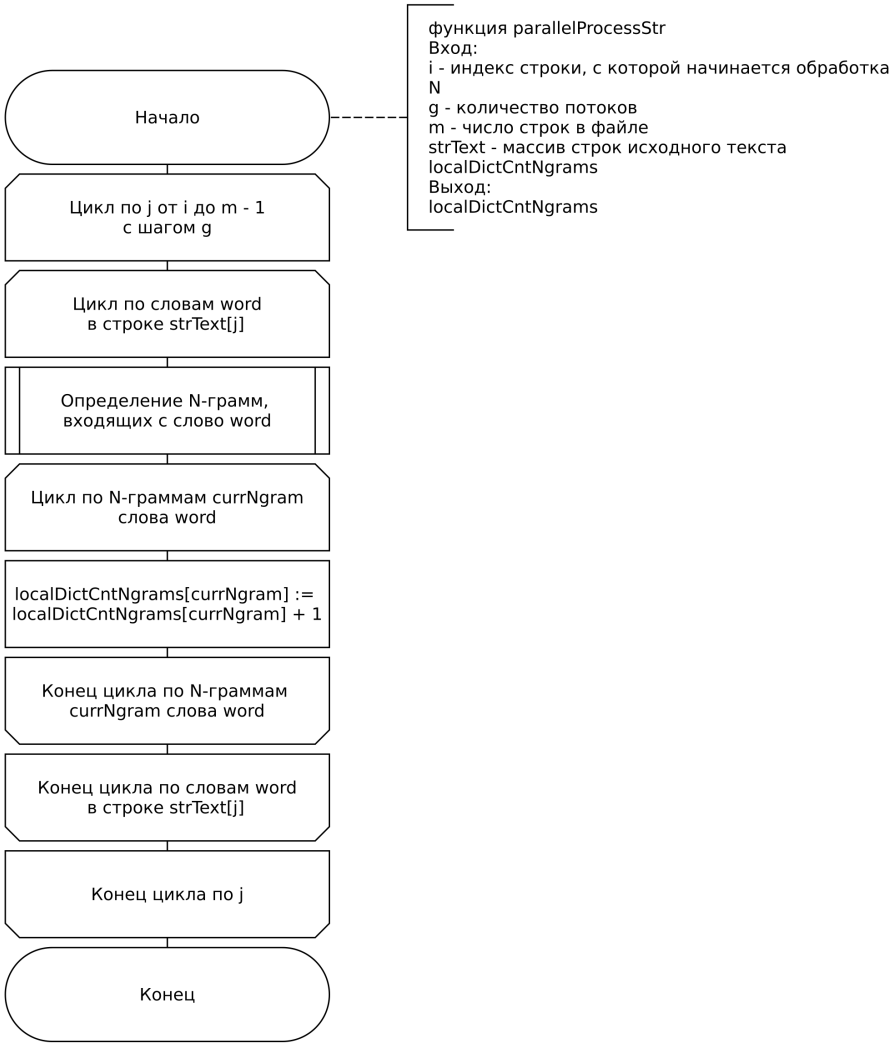

Рисунок 2.4 – Схема алгоритма функции для одного потока

Вывод

В данном разделе были построены схемы последовательной и параллельной версии алгоритма составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке.

3 Технологический раздел

В данном разделе будут перечислены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к программному обеспечению

К программе предъявляется ряд требований:

- на вход имя файла с текстом на русском языке, имя файла-словаря, число N — N-грамма;
- в программе для распараллеливания вычислений используются только нативные потоки;
- в результате работы программы получаем файл-словарь с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке.

3.2 Средства реализации

В качестве языка программирования для этой лабораторной работы был выбран $C++$ [3] по следующим причинам:

- в $C++$ есть встроенный модуль *ctime*, предоставляющий необходимый функционал для замеров процессорного времени;
- в $C++$ есть встроенный модуль *thread* [4], предоставляющий необходимый интерфейс для работы с нативными потоками.

В качестве функции, которая будет осуществлять замеры процессорного времени, будет использована функция *clock_gettime* из встроенного модуля *ctime* [5].

3.3 Сведения о модулях программы

Программа состоит из пяти модулей:

1) `algorithms.cpp` — модуль, хранящий реализации последовательной и параллельной версии алгоритма составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке;

2) `processTime.cpp` — модуль, содержащий функцию для замера процессорного времени;

3) `timeMeasurements.cpp` — модуль, содержащий функции, позволяющие провести сравнительный анализ использования времени;

4) `main.cpp` — файл, содержащий точку входа в программу;

5) `task7` — модуль, содержащий набор скриптов для проведения замеров программы по времени и памяти и построения графиков по полученным данным.

3.4 Реализации алгоритмов

В листингах 3.1 – 3.3 представлена последовательная версия алгоритма составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке.

Листинг 3.1 – Реализация функции вычисления N-грамм в слове и построчного считывания текста из файла с преобразованиями

```
static std::vector<std::wstring> getNgramsByWord(const
   std::wstring &word, int ngram)
{
    std::vector<std::wstring> ngrams;

    for (size_t i = 0; i <= word.length() - ngram; ++i)
        ngrams.push_back(word.substr(i, ngram));

    return ngrams;
}

static std::vector<std::wstring> getVectorText(std::wifstream
   &inputFile)
{
    std::vector<std::wstring> vecStrText;
    std::wstring currStr;

    while (std::getline(inputFile, currStr))
    {
        currStr.erase(std::remove_if(currStr.begin(),
           currStr.end(), ::iswpunct), currStr.end());
        std::transform(currStr.begin(), currStr.end(),
           currStr.begin(), ::towlower);

        vecStrText.push_back(currStr);
    }

    return vecStrText;
}
```

Листинг 3.2 – Реализация функции обработки строки

```
static void processStr(std::wstring &currStr, const int ngram,
   std::map<std::wstring, int> &ngramCounts)
{
    size_t startPos = 0;
    size_t endPos = 0;

    while (endPos != std::wstring::npos)
    {
        endPos = currStr.find(L' ', startPos);

        std::wstring word = currStr.substr(startPos, endPos -
           startPos);

        if (static_cast<int>(word.size()) < ngram)
        {
            startPos = endPos + 1;
            continue;
        }

        std::vector<std::wstring> ngrams = getNgramsByWord(word,
           ngram);

        for (const auto &ngram : ngrams)
        {
            ngramCounts[ngram]++;
        }

        startPos = endPos + 1;
    }
}
```

Листинг 3.3 – Реализация функции обработки всего текста

```
int solution(const std::string &filename, const std::string
   &outputFilename, const int ngram)
{
    std::wifstream inputFile(filename);

    if (!inputFile.is_open())
    {
        std::wcerr << L"Ошибка открытия файла" << std::endl;
        return 1;
    }

    std::vector<std::wstring> vecStrText =
       getVectorText(inputFile);

    inputFile.close();

    std::map<std::wstring, int> ngramCounts;

    for (int i = 0; i < (int)vecStrText.size(); ++i)
        processStr(vecStrText[i], ngram, ngramCounts);

    std::wofstream outputFile(outputFilename);

    if (!outputFile.is_open())
    {
        std::wcerr << L"Ошибка открытия файла" << std::endl;
        return 2;
    }

    for (const auto &entry : ngramCounts)
        outputFile << entry.first << ": " << entry.second <<
           std::endl;

    outputFile.close();

    return 0;
}
```

В листингах 3.4 – 3.3 представлена последовательная версия алгоритма составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке.

Листинг 3.4 – Реализация функции вычисления N-грамм в слове и построчного считывания текста из файла с преобразованиями

```
static std::vector<std::wstring> getNgramsByWord(const
   std::wstring &word, int ngram)
{
    std::vector<std::wstring> ngrams;

    for (size_t i = 0; i <= word.length() - ngram; ++i)
        ngrams.push_back(word.substr(i, ngram));

    return ngrams;
}

static std::vector<std::wstring> getVectorText(std::wifstream
   &inputFile)
{
    std::vector<std::wstring> vecStrText;
    std::wstring currStr;

    while (std::getline(inputFile, currStr))
    {
        currStr.erase(std::remove_if(currStr.begin(),
           currStr.end(), ::iswpunct), currStr.end());
        std::transform(currStr.begin(), currStr.end(),
           currStr.begin(), ::towlower);

        vecStrText.push_back(currStr);
    }

    return vecStrText;
}
```

Листинг 3.5 – Реализация функции обработки строк для каждого потока

```
void parallelProcessStr(int i, std::vector<std::wstring>
   &vecStrText, int ngram, int numThreads, std::map<std::wstring,
   int> &localNgramCounts)
{
    for (int j = i; j < (int)vecStrText.size(); j += numThreads)
    {
        size_t startPos = 0;
        size_t endPos = 0;

        while (endPos != std::wstring::npos)
        {
            endPos = vecStrText[j].find(L' ', startPos);

            std::wstring word = vecStrText[j].substr(startPos,
               endPos - startPos);

            if (static_cast<int>(word.size()) < ngram)
            {
                startPos = endPos + 1;
                continue;
            }

            std::vector<std::wstring> ngrams =
               getNgramsByWord(word, ngram);

            for (const auto &ngram : ngrams)
                localNgramCounts[ngram]++;

            startPos = endPos + 1;
        }
    }
}
```

Листинг 3.6 – Реализация функции многопоточной обработки всего текста (часть 1)

```
int solution(const std::string &filename, const std::string
   &outputFilename, const int ngram, const int numThreads)
{
    std::wifstream inputFile(filename);
    std::wofstream outputFile(outputFilename);

    if (!inputFile.is_open())
    {
        std::wcerr << L"Ошибка открытия файла" << std::endl;
        return 1;
    }

    if (!outputFile.is_open())
    {
        std::wcerr << L"Ошибка открытия файла" << std::endl;
        return 2;
    }

    std::vector<std::wstring> vecStrText =
       getVectorText(inputFile);
    std::map<std::wstring, int> ngramCounts;
    std::vector<std::thread> threads(numThreads);
    std::vector<std::map<std::wstring, int>>
       localNgramCounts(numThreads);

    for (int i = 0; i < numThreads; ++i)
        threads[i] = std::thread(parallelProcessStr, i,
           std::ref(vecStrText), ngram, numThreads,
           std::ref(localNgramCounts[i]));

    for (auto &thread : threads)
        thread.join();

    for (const auto &localCount : localNgramCounts)
        for (const auto &entry : localCount)
            ngramCounts[entry.first] += entry.second;
```

Листинг 3.7 – Реализация функции многопоточной обработки всего текста (часть 2)

```
    for (const auto &entry : ngramCounts)
        outputFile << entry.first << ": " << entry.second <<
           std::endl;

    inputFile.close();
    outputFile.close();

    return 0;
}
```

Вывод

В данном разделе были реализованы последовательная и параллельная версии алгоритма составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке.

4 Исследовательский раздел

В данном разделе будут проведены сравнения реализаций алгоритмов сортировки по времени работы и по затрачиваемой памяти.

4.1 Технические характеристики

Технические характеристики устройства, на котором проводились исследования:

- операционная система: Ubuntu 22.04.3 LTS x86_64 [6];
- оперативная память: 16 Гб;
- процессор: 11th Gen Intel® Core™ i7-1185G7 @ 3.00 ГГц × 8, 4 физических ядра, 8 логических ядер.

4.2 Проведение первого исследования

Цель первого исследования

Целью первого исследования является проведение сравнительного анализа времени работы последовательной версии алгоритма и параллельной версии алгоритма с одним вспомогательным потоком

Наборы варьируемых и фиксированных параметров

Замеры времени проводились для случайно сгенерированных файлом с числом строк, равным 10, 20, 30, 40, 50, 60, 70, 80, 90, 100.

В качестве фиксированного параметра было выбрано число N, равное 3.

Замеры времени для каждого размера 1000 раз. Время работы алгоритмов измерялось с использованием функции *clock_gettime* из встроенного модуля *ctime*.

Результаты первого исследования

Таблица 4.1 – Замер времени для файлов с числом строк от 10 до 100

Число строк в файле, единицы	**Время, мкс**	
	Последовательная версия	**Параллельная версия с 1-м впомогат. потоком**
10	435.759	882.219
20	923.062	1 532.931
30	1 435.287	2 343.806
40	2 000.813	3 070.613
50	2 383.545	3 892.355
60	3 051.124	4 674.078
70	3 511.962	5 709.621
80	4 054.163	6 309.628
90	4 709.397	7 285.999
100	5 569.732	8 170.710

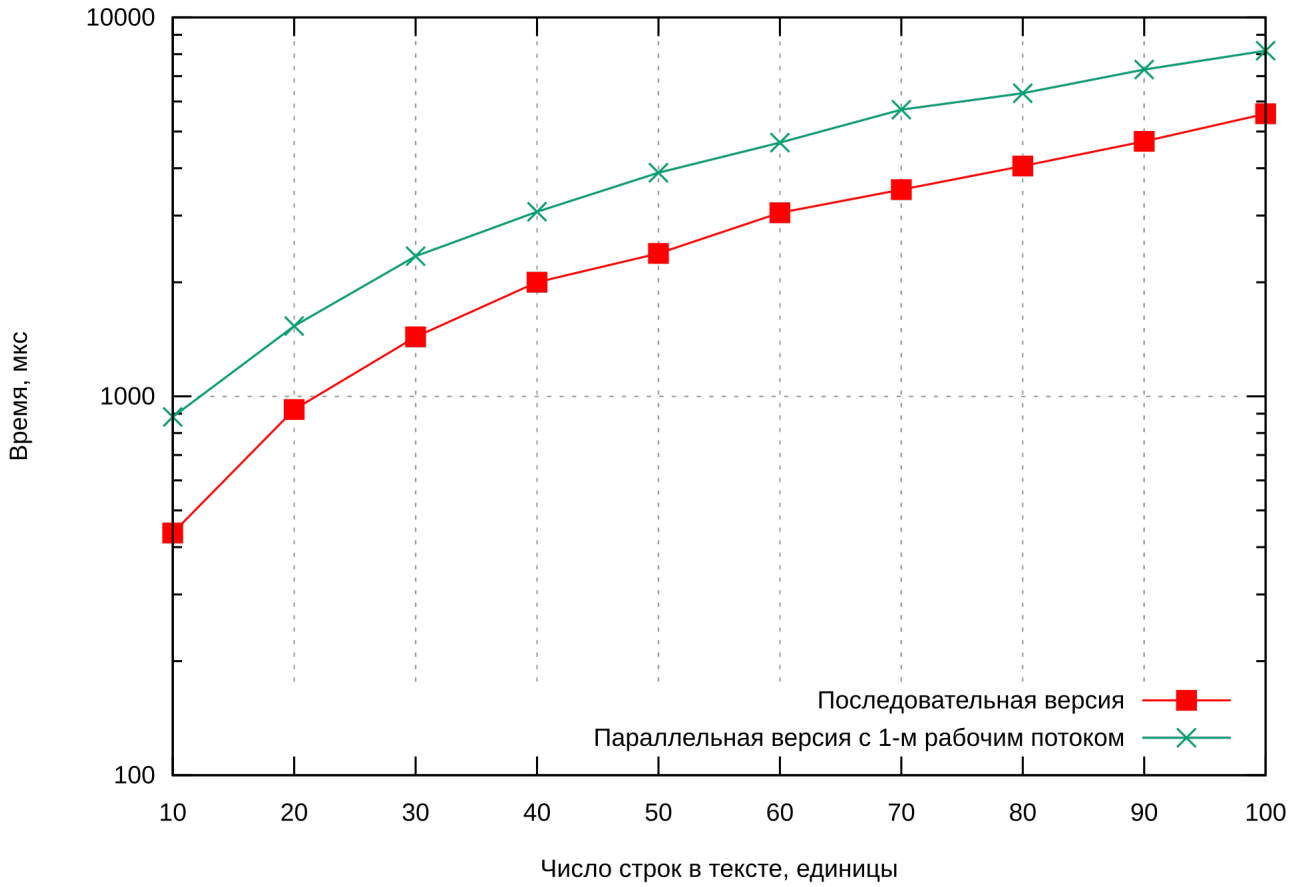

Рисунок 4.1 – Результаты замеров времени работы алгоритмов для файлов с числом строк от 10 до 100

Из полученных данных следует, что однопоточный процесс демонстрирует более высокую эффективность по сравнению с процессом, включающим в себя создание вспомогательного потока для обработки всех строк файла. Это объясняется дополнительными временными затратами, связанными с созданием потока и передачей ему необходимых аргументов.

4.3 Проведение второго исследования

Цель второго исследования

Целью второго исследования является проведение сравнительного анализа времени работы параллельной версии алгоритма с разным числом потоков.

Наборы варьируемых и фиксированных параметров

Замеры времени проводились для числа потоков, равного 1, 2, 4, 8, 16, 34, 64.

В качестве фиксированного параметра было выбрано число N, равное 3 и число строк в файле с текстом, равное 100.

Замеры времени для каждого размера 1000 раз. Время работы алгоритмов измерялось с использованием функции *clock_gettime* из встроенного модуля *ctime*.

Результаты второго исследования

Таблица 4.2 – Замер времени для числа потоков от 1 до 64

Число потоков, единицы	**Время, мкс**
	Параллельная версия
1	7 592.842
2	7 397.898
4	8 306.083
8	11 318.340
16	11 416.410
32	12 302.330
64	12 837.810

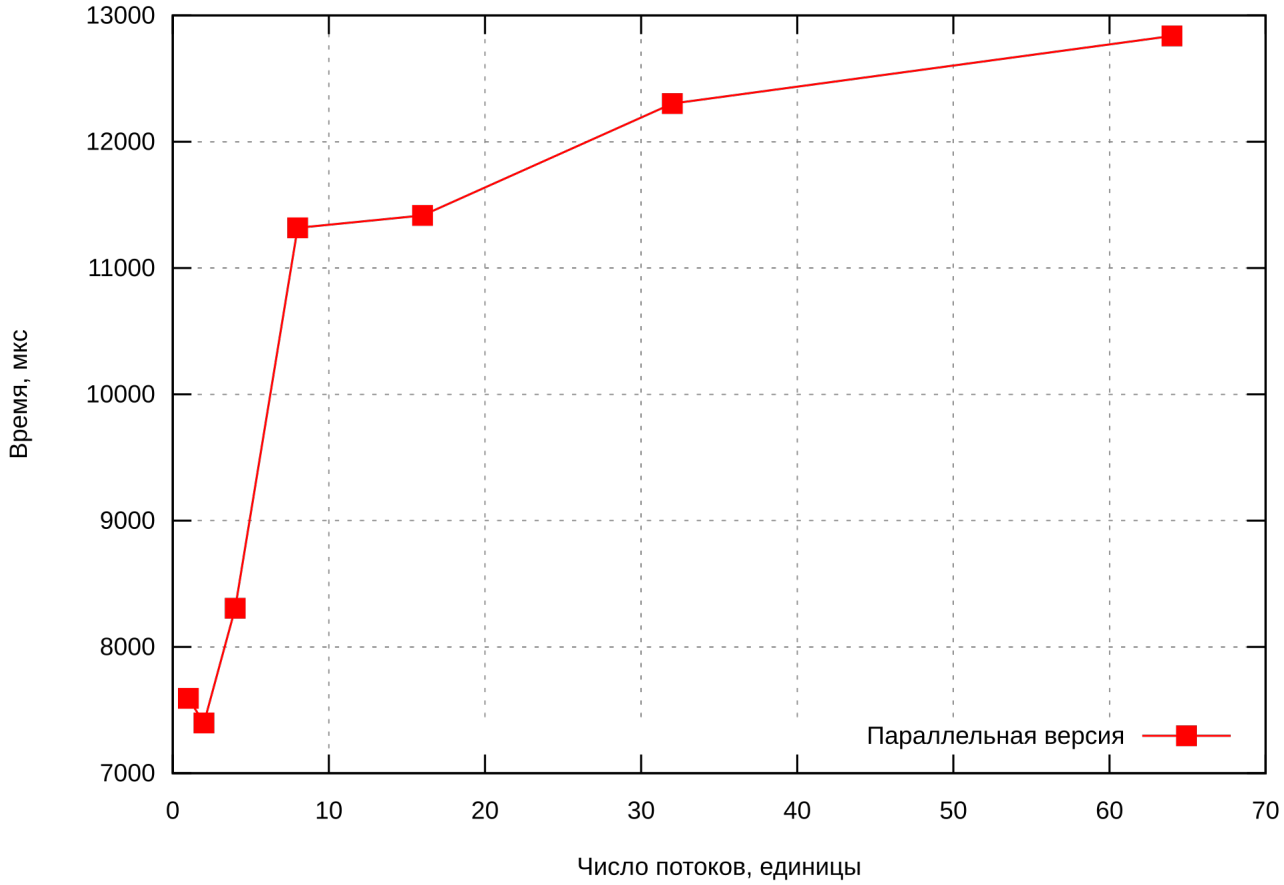

Рисунок 4.2 – Результаты замеров времени работы алгоритмов для числа потоков от 1 до 64

Оптимальные результаты по времени достигаются при использовании процесса с двумя дополнительными потоками, ответственными за вычисления. Для данной архитектуры вычислительной машины рекомендуется удерживаться при числе дополнительных потоков, равном 2. При увеличении числа потоков выше этого значения, расходы на поддержание потоков превышают выгоду от использования многопоточности, и функция времени в зависимости от количества потоков начинает увеличиваться.

Вывод

Для файла с текстом на русском языке размером в 100 строк:

– однопоточный процесс — 5569.732 мкс;

– параллельная версия с 1-м вспомогательным потоком — 8170.710;

– параллельная версия с 2-мя вспомогательными потоками – 7397.898;

– параллельная версия с 64-мя вспомогательными потоками – 12837.810;

Таким образом, для данного алгоритма использование дополнительных потоков приводит лишь к увеличению времени работы программы (даже с 2-мя потоками время работы программы в 1.33 раза больше, чем у однопоточного процесса).Для данной архитектуры вычислительной машины рекомендуется удерживаться при числе дополнительных потоков, равном 2.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были решены следующие задачи:

1) изучен и описан алгоритм составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке;

2) разработана последовательная и параллельная версии данного алгоритма;

3) реализована каждая версия алгоритма;

4) проведен сравнительный анализ алгоритмов по времени работы реализаций;

5) обоснованы полученные результаты в отчете к выполненной лабораторной работе.

Цель данной лабораторной работы, изучение принципов и получение навыков организации параллельного выполнения операций, также была достигнута.

В ходе выполнения лабораторной работы было выявлено, что в результате использования многопоточной реализации время выполнения процесса может как улучшиться, так и ухудшиться в зависимости от количества используемых потоков.

Для рассматриваемого алгоритма использование многопоточности приводит лишь к увеличению времени работы программы (даже с 2-мя потоками время работы программы в 1.33 раза больше, чем у однопоточного процесса). Для данной архитектуры вычислительной машины рекомендуется удерживаться при числе дополнительных потоков, равном 2.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Многопоточность в языке программирования C# [Электронный ресурс]. — Режим доступа: https://cyberleninka.ru/article/n/mnogopotochnost-v-yazyke-programmirovaniya-c (дата обращения: 27.01.2024).
2. N-граммы в лингвистике [Электронный ресурс]. — Режим доступа: https://cyberleninka.ru/article/n/n-grammy-v-lingvistike (дата обращения: 27.01.2024).
3. Справочник по языку C++ [Электронный ресурс]. — Режим доступа: https://learn.microsoft.com/ru-ru/cpp/cpp/cpp-language-reference?view=msvc-170 (дата обращения: 28.09.2022).
4. std::thread [Электронный ресурс]. — Режим доступа: https://en.cppreference.com/w/cpp/thread/thread (дата обращения: 27.01.2024).
5. clock_getres [Электронный ресурс]. — Режим доступа: https://pubs.opengroup.org/onlinepubs/9699919799/functions/clock_getres.html (дата обращения: 28.09.2022).
6. Ubuntu 22.04.3 LTS (Jammy Jellyfish) [Электронный ресурс]. — Режим доступа: https://releases.ubuntu.com/22.04/ (дата обращения: 28.09.2022).