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.

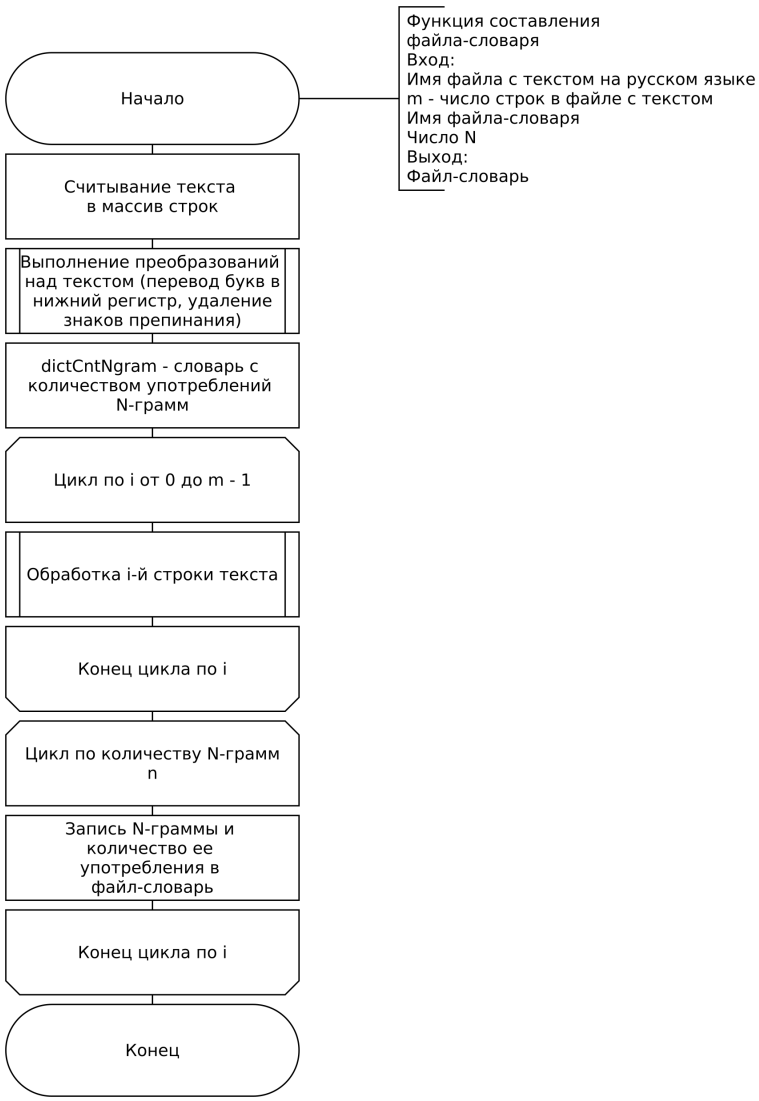

Рисунок 2.1 – Схема алгоритма для обработки целого текста

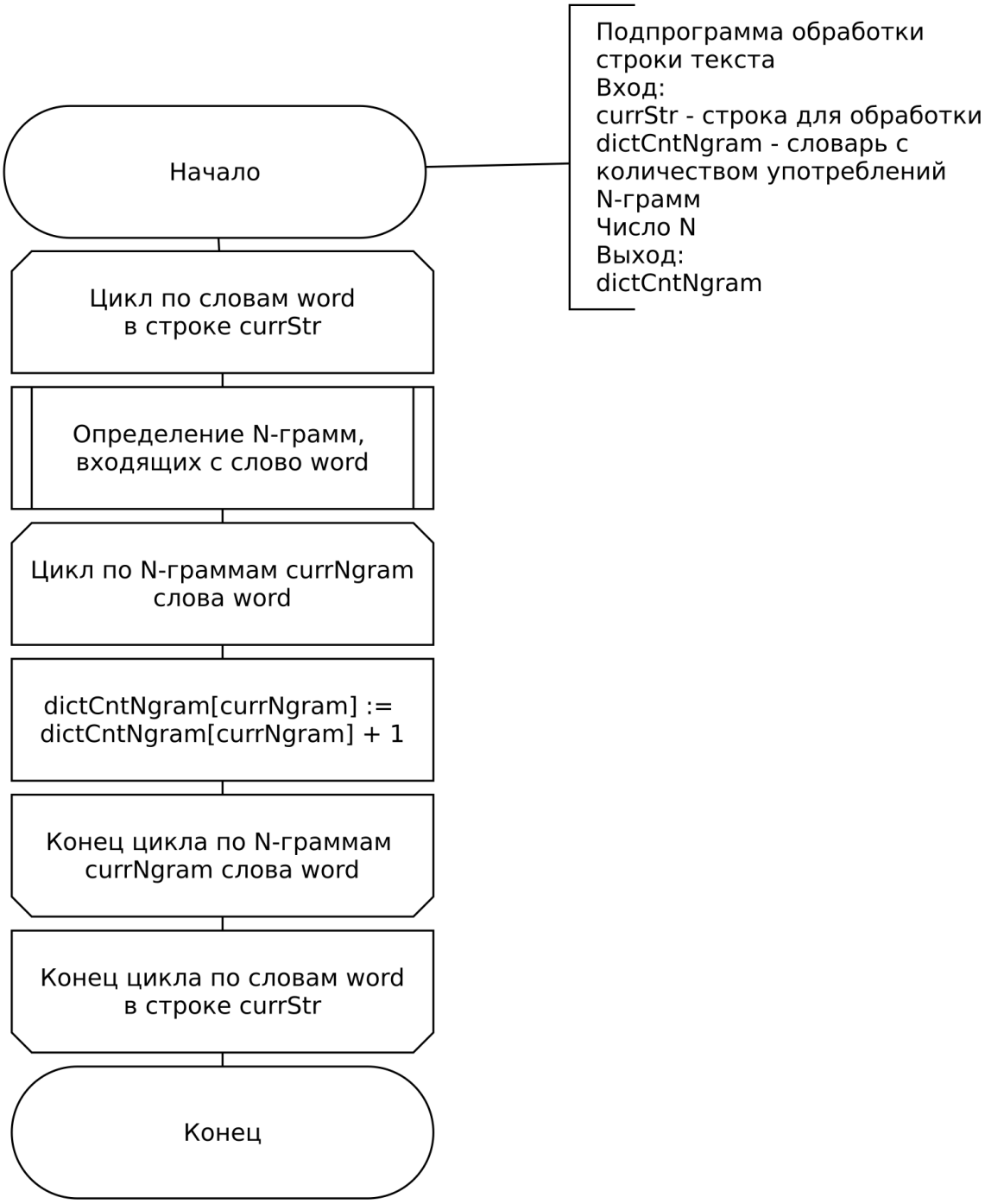

Рисунок 2.2 – Схема алгоритма для обработки строки текста

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.

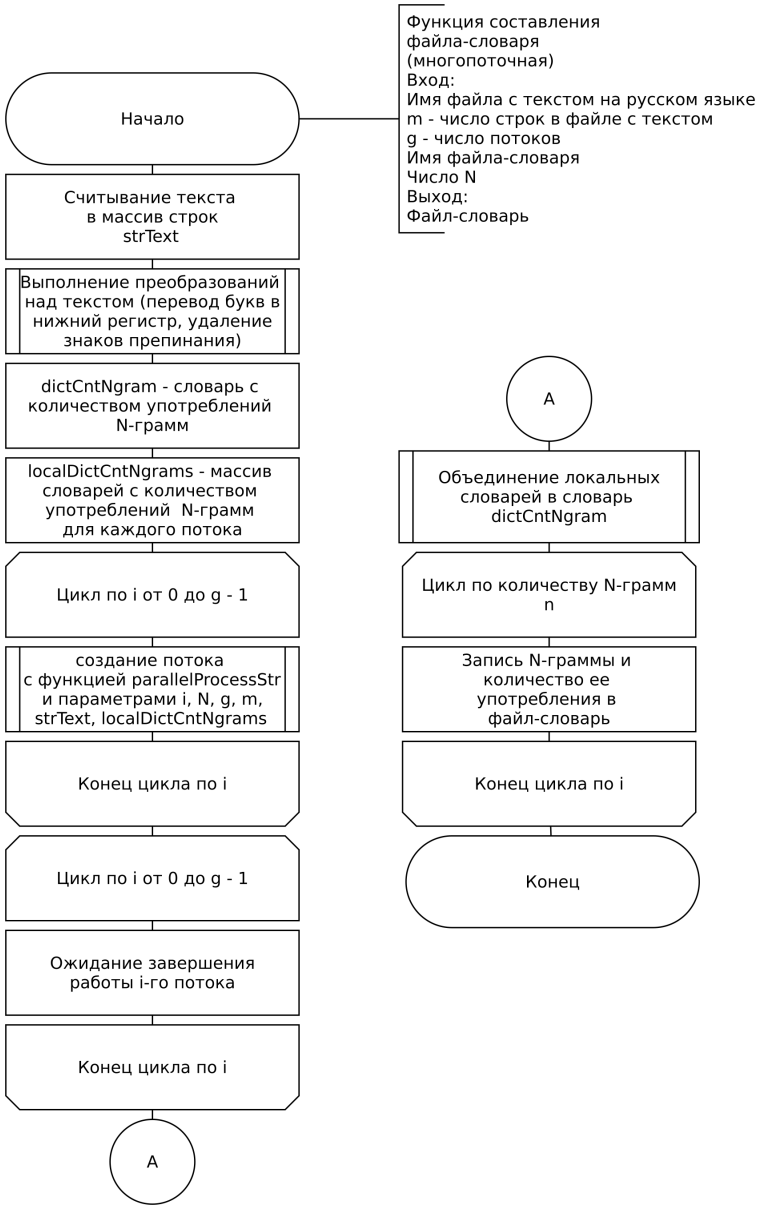

Рисунок 2.3 – Схема алгоритма для обработки целого текста

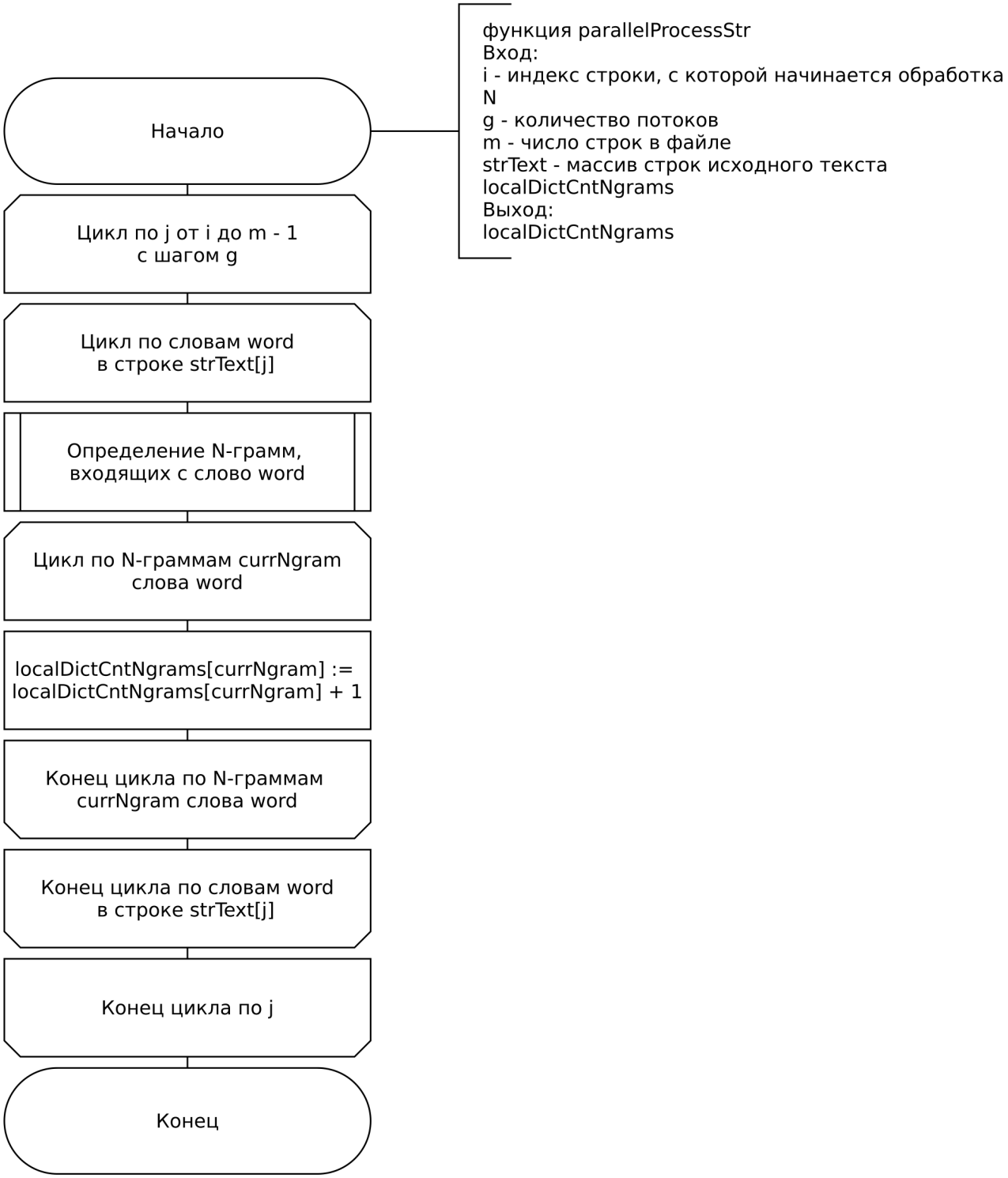

Рисунок 2.4 – Схема алгоритма функции для одного потока

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.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были решены следующие задачи:

1) Изучены и описаны три алгоритма сортировки: блочной, слиянием и поразрядной.

2) Создано программное обеспечение, реализующее следующие алгоритмы:

   – алгоритм блочной сортировки;

   – алгоритм сортировки слиянием;

   – алгоритм поразрядной сортировки.

3) Проведен анализ эффективности реализаций алгоритмов по памяти и по времени.

4) Проведена оценка трудоемкости алгоритмов сортировки.

5) Подготовлен отчет по лабораторной работе.

Цель данной лабораторной работы, а именно исследование трех алгоритмов сортировки: блочной сортировки, сортировки слиянием и поразрядной сортировки, также была достигнута.

Согласно теоретическим расчетам трудоемкости алгоритмов сортировки наименее трудоемким на равномерно распределенных данных оказался алгоритм блочной сортировки, наиболее трудоемким – алгоритм поразрядной сортировки.

Результаты проведенного исследования практически подтвердили теоретические расчеты трудоемкости: наиболее эффективным по времени работы и по используемой памяти является алгоритм блочной сортировки, но наиболее трудоемким оказался алгоритм сортировки слиянием.

На равномерно распределенных данных лучше всего использовать алгоритм блочной сортировки, а для сортировки любых элементов, которые можно поделить на разряды, подошел бы алгоритм поразрядной сортировки.

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).