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 рубежному контролю № 1

по курсу «Анализ алгоритмов»

на тему: «Многопоточное исправление орфографических ошибок»

Студент ИУ7-54Б _____ Булдаков М.
(Группа) (Подпись, дата) (И. О. Фамилия)

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

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

С развитием вычислительных систем появилась потребность в параллельной обработке данных для повышения эффективности систем, ускорения вычислений и более рационального использования имеющихся ресурсов. Благодаря совершенствованию процессоров стало возможно использовать их для выполнения множества одновременных задач, что привело к появлению понятия «многопоточность» [1].

Цель данного рубежного контроля — описать принципы параллельных вычислений на основе нативных потоков для исправления орфографических ошибок в тексте. Для достижения поставленной цели необходимо выполнить следующие задачи:

— описать алгоритм исправления орфографических ошибок в тексте;

— спроектировать программное обеспечение, реализующее алгоритм и его параллельную версию;

— выбрать инструменты для реализации и замера процессорного времени выполнения реализаций алгоритмов;

— проанализировать затраты реализаций алгоритмов по времени.

1 Аналитический раздел

В данном разделе будет представлена информация о многопоточности и исследуемом алгоритме исправления орфографических ошибок в тексте.

1.1 Многопоточность

Многопоточность [2] — это способность центрального процессора одновременно выполнять несколько потоков, используя ресурсы одного процессора. Каждый поток представляет собой последовательность инструкций, которые могут выполняться параллельно с другими потоками, созданными одним и тем же процессом.

Процессом называют программу в стадии выполнения [3]. Один процесс может иметь один или несколько потоков. Поток — это часть процесса, которая выполняет задачи, необходимые для выполнения приложения. Процесс завершается, когда все его потоки полностью завершены.

Одной из сложностей, связанных с использованием потоков, является проблема доступа к данным. Основным ограничением является невозможность одновременной записи в одну и ту же ячейку памяти из разных потоков. Это означает, что нужен механизм синхронизации доступа к данным, так называемый “мьютекс” (от англ. mutex - mutual exclusion, взаимное исключение). Мьютекс может быть захвачен одним потоком для работы в режиме монопольного использования или освобожден. Если два потока попытаются захватить мьютекс одновременно, то успех будет у одного потока, а другой будет блокирован, пока мьютекс не освободится.

1.2 Исправления орфографических ошибок в тексте

Для распознавания слов, написанных с ошибками, используется расстояние Левенштейна — минимальное количество ошибок, исправление которых приводит одно слово к другому [4]. Т. о. для введенного слова осуществляется проверка по корпусу, если данное слово не найдено в корпусе, то ищется ближайшее слово к данному по расстоянию Левенштейна.

Кроме того, следует вводить ограничение на количество ошибок, которые позволяется допустить. Как говорит поговорка: «Если в слове хлеб допустить всего четыре ошибки, то получится слово пиво». Если фиксируется число

ошибок, то для коротких слов оно может оказаться избыточным. Верхнюю границу числа ошибок обычно ограничивают как процентным соотношением, так и фиксированным числом. Например, не более 30% букв входного слова, но не более 3. При этом все равно стараются найти слова с минимальным количеством ошибок [4].

1.3 Использование потоков для исправления орфографических ошибок

Поскольку задача сводится к поиску слова в корпусе, можно распараллелить поиск по этому корпусу. В таком случае каждый поток будет вычислять расстояние Левенштейна между заданным словом и некоторым словом из корпуса и в случае, если расстояние будет удовлетворять требованиям, то данное слово будет записано в массив. Для определения наилучших соответствий необходимо хранить минимальное количество ошибок на текущий момент, т. е. возможна ситуация, когда в одном потоке минимальное количество ошибок будет считано, а в другом в тот же момент изменено, следовательно возникает конфликт. То же касается и записи подходящих слов в массив, требуется отбирать лучшие k слов, поэтому возможна ситуация, когда значение длины массива считывается в одном потоке и в тот же момент изменяется в другом потоке, т. е. возникает конфликт. Для решения проблем синхронизации необходимо использовать мьютекс, чтобы обеспечить монопольный доступ к длине массива и текущему минимальному количеству ошибок.

Корпус можно представить с помощью сегментированного словаря. Каждый сегмент будет хранить слова, которые начинаются на одну и ту же букву алфавита. Поскольку ошибки чаще бывают в середине слова, а не в начале, то такой подход позволит минимизировать количество слов среди которых нужно осуществлять поиск.

Вывод

В данном разделе была представлена информация о многопоточности и исследуемом алгоритме.

2 Конструкторский раздел

В этом разделе будет представлено описание используемых типов данных, а также схемы алгоритмов исправления орфографических ошибок.

2.1 Требования к программному обеспечению

Программа должна поддерживать два режима работы: режим массового замера времени и режим исправления введенного слова.

Режим массового замера времени должен обладать следующей функциональностью:

— генерировать корпус слов;

— осуществлять массовый замер, используя сгенерированные данные;

— результаты массового замера должны быть представлены в виде таблицы.

К режиму исправления введенного слова выдвигается следующий ряд требований:

— возможность вводить слова, которые отсутствуют в корпусе;

— слова вводятся на кириллице;

— наличие интерфейса для выбора действий;

— на выходе программы, набор из самых близких слов к введенному.

2.2 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры и типы данных:

— слово — массив букв;

— корпус — массив слов, отсортированный в лексикографическом порядке;

— сегментированный словарь — словарь, где ключом является буква, а значением корпус;

— мьютекс — примитив синхронизации.

2.3 Разработка алгоритмов

На рисунке 2.1 представлена схема поиска ближайших слов в словаре, алгоритм одинаков в случае если используются потоки и если не используются, поскольку потоки применяются только для поиска слов в корпусе. На рисунке 2.2 представлена схема поиска ближайших слов в корпусе без использования потоков. На рисунке 2.3 представлена схема поиска ближайших слов в корпусе с использованием потоков. На рисунке 2.4 представлена схема алгоритма программы, выполняющейся в потоке.

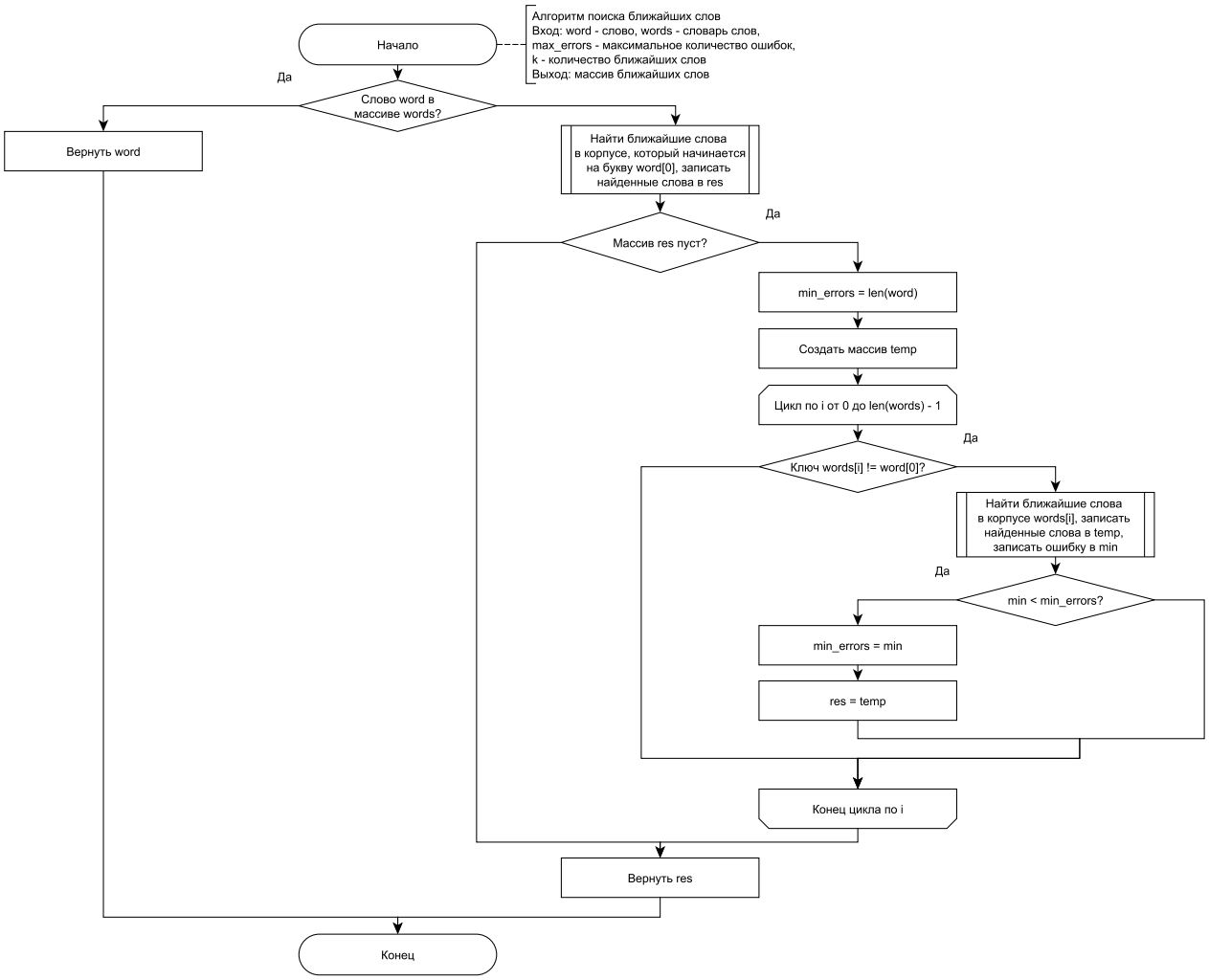

Рисунок 2.1 – Схема алгоритма поиска ближайших слов в сегментированном словаре

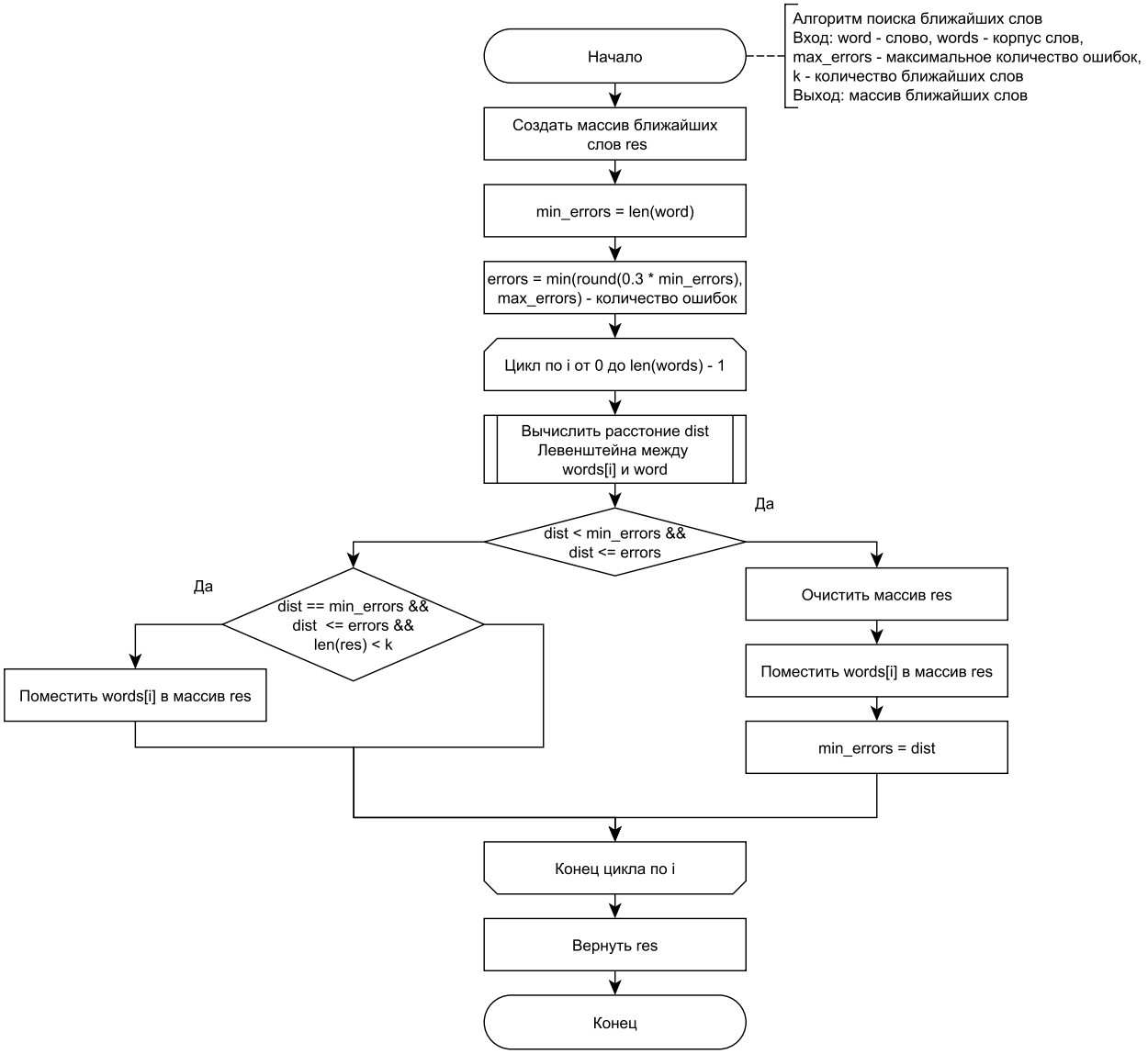

Рисунок 2.2 – Схема алгоритма поиска ближайших слов в корпусе

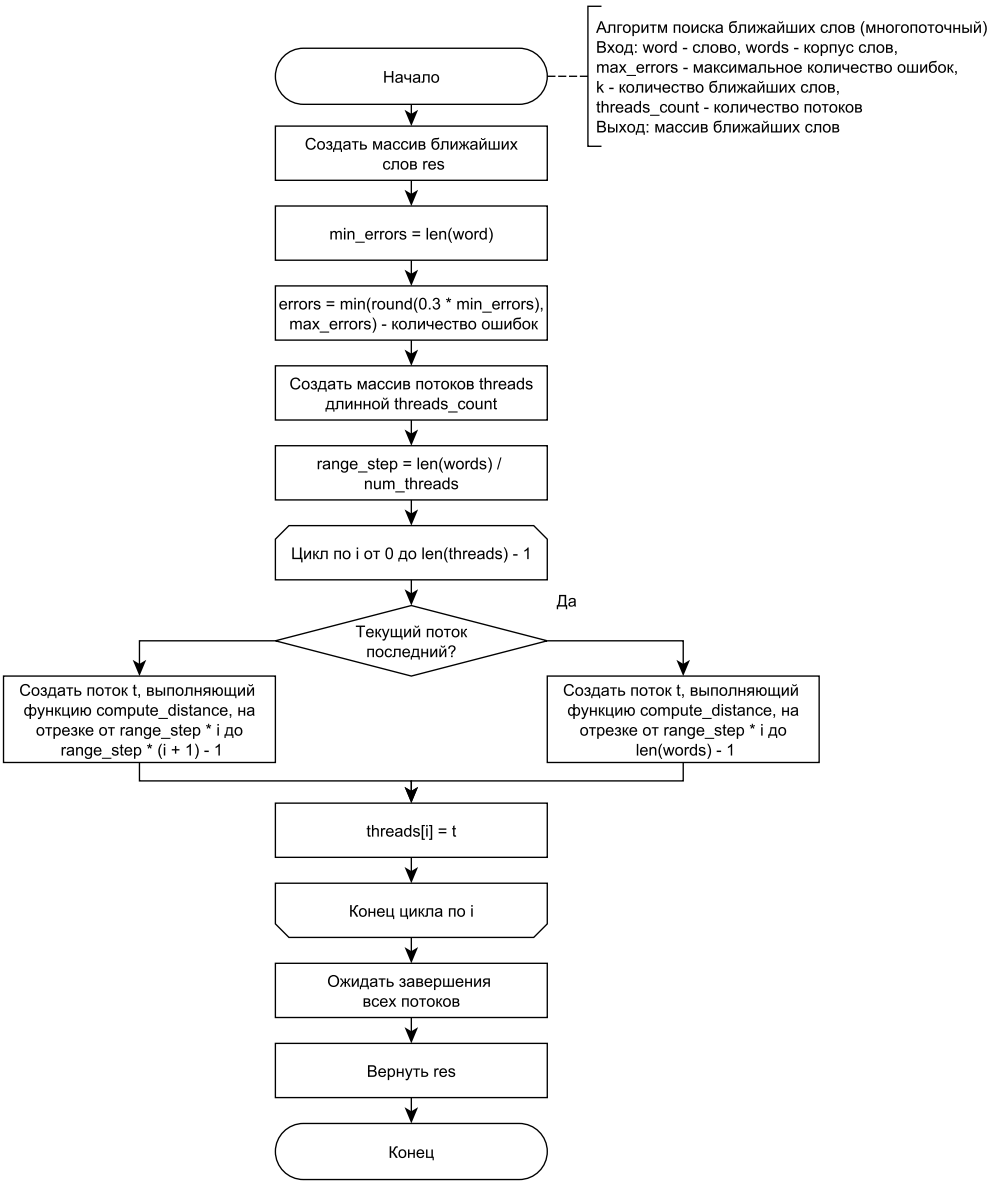

Рисунок 2.3 – Схема многопоточного алгоритма поиска ближайших слов в корпусе

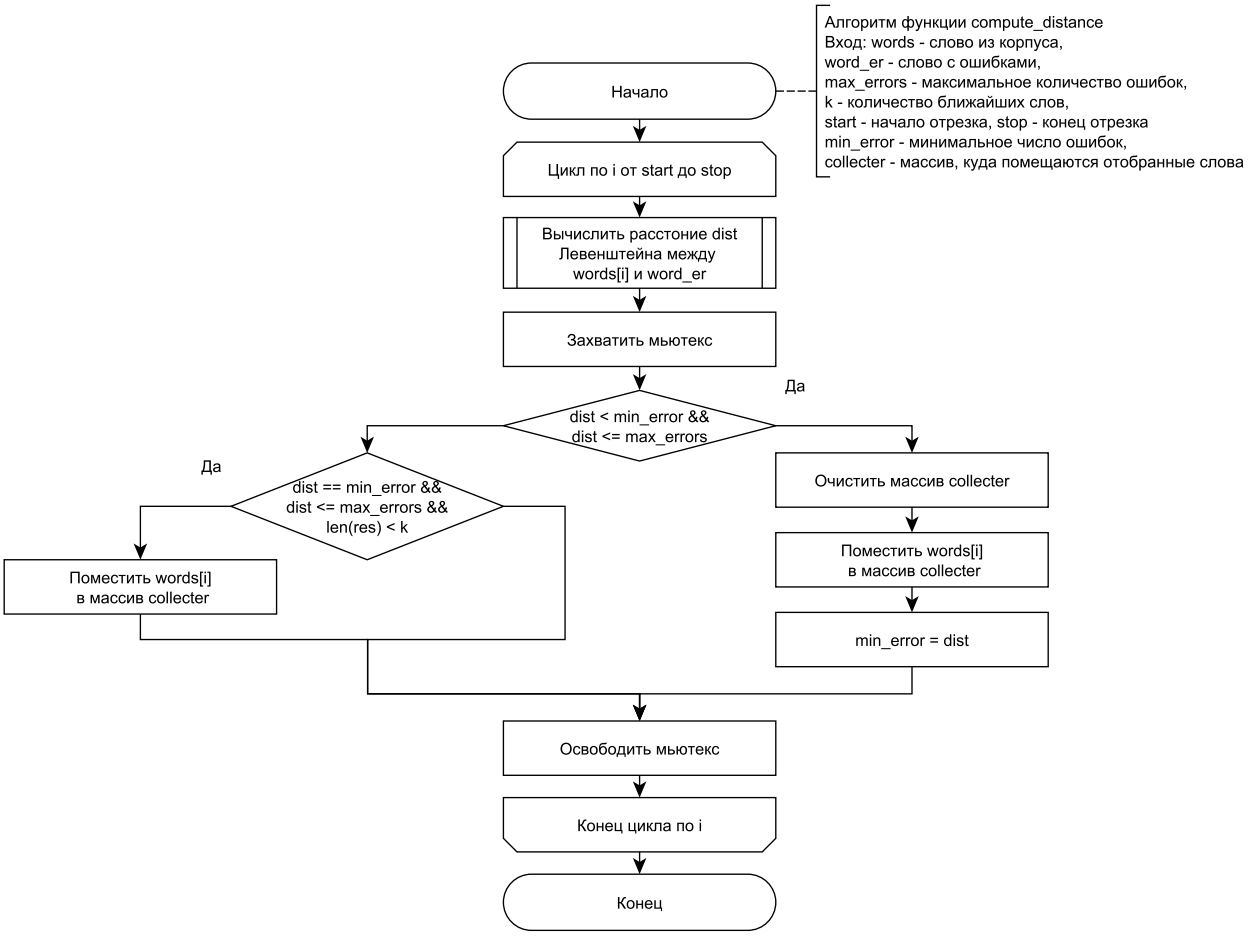

Рисунок 2.4 – Схема алгоритма потока

Поскольку в массиве в каждый момент времени содержатся слова, которые находятся на одинаковом расстоянии от введенного слова, то нет необходимости в какой-либо сортировке результатов, т. к. любое из найденных слов с одинаковой вероятностью может оказаться искомым. Поэтому шаг особого слияния результатов не требуется и достаточно помещать подходящее слово в массив.

Вывод

На основе теоретических данных, полученных из аналитического раздела были построены схемы требуемых алгоритмов.

3 Технологический раздел

В данном разделе будут приведены требования к программному обеспечению, средства реализации, листинг кода и функциональные тесты.

3.1 Средства реализации

Для реализации данной работы был выбран язык *C++* [5]. Данный выбор обусловлен следующим:

— язык поддерживает все структуры данных, которые выбраны в результате проектирования;

— язык позволяет реализовать все алгоритмы, выбранные в результате проектирования;

— язык позволяет работать с нативными потоками [6].

Время выполнения реализаций было замерено с помощью функции *clock* [7]. Для хранения слов использовалась структура данных *wstring* [8], в качестве массивов использовалась структура данных *vector* [9], в качестве словаря структура данных *map* [10]. В качестве примитива синхронизации использовался *mutex* [11].

Для создания потоков и работы с ними был использован класс *thread* из стандартной библиотеки выбранного языка [6]. В листинге 3.1, приведен пример работы с описанным классом, каждый объект класса представляет собой поток операционной системы, что позволяет нескольким функциям выполняться параллельно [6].

Листинг 3.1 – Пример работы с классом thread

```
#include <iostream>
#include <thread>

void foo(int a)
{
    std::cout << a << '\n';
}

int main()
{
    std::thread thread(foo, 10);
    thread.join();

    return 0;
}
```

3.2 Сведения о модулях программы

Данная программа разбита на следующие модули:

— *main.cpp* — файл, содержащий функцию *main*;

— *correcter.cpp* — файл, содержащий код реализаций всех алгоритмов исправления ошибок;

— *measure_time.cpp* — файл, в котором содержатся функции для замера и вывода времени выполнения реализаций алгоритмов;

— *utils.cpp* — файл, в котором содержатся вспомогательные функции;

— *levenstein.cpp* — файл, в котором содержится реализация алгоритма поиска расстояния Левенштейна.

3.3 Реализация алгоритмов

На листингах 3.2 – 3.5 приведены реализации функций, который осуществляют работу с сегментированным словарем. В листинге 3.6 приведена реализация алгоритма поиска слов в корпусе. В листинге 3.7 приведена реализация алгоритма поиска слов в корпусе с использованием дополнительных

потоков. В листинге 3.8 приведена реализация функции, которая выполняется потоком.

Листинг 3.2 – Функция поиска слов в словаре с дополнительными потоками

```
std::vector<std::wstring> get_closest_words_with_segmented_mt(
    std::map<wchar_t, std::vector<std::wstring>> &words,
    const std::wstring &word, size_t k, size_t max_errors,
       size_t num_threads){

    if (is_word_in_vec(words[word[0]], word))
        return {word};

    auto res = get_closest_words_mt(words[word[0]], word, k,
       max_errors, num_threads);

    std::vector<std::wstring> ans = res.first;

    if (ans.empty()){
        ans = full_search_mt(words, word, k, max_errors,
           num_threads);
    }

    return ans;
}
```

Листинг 3.3 – Функция поиска слов в словаре

```
std::vector<std::wstring> get_closest_words_with_segmented(
    std::map<wchar_t, std::vector<std::wstring>> &words,
    const std::wstring &word, size_t k, size_t max_errors){

    if (is_word_in_vec(words[word[0]], word))
        return {word};

    auto res = get_closest_words(words[word[0]], word, k,
       max_errors);

    std::vector<std::wstring> ans = res.first;

    if (ans.empty()){
        ans = full_search(words, word, k, max_errors);
    }

    return ans;
}
```

Листинг 3.4 – Функция полного перебора словаря

```
std::vector<std::wstring> full_search(
    std::map<wchar_t, std::vector<std::wstring>> &words,
    const std::wstring &word, size_t k, size_t max_errors){

    std::vector<std::wstring> res;
    size_t min = word.size();

    for (const auto &p: words){
        if (p.first != word[0]){
            auto temp = get_closest_words(p.second, word, k,
               max_errors);

            if (temp.second < min){
                res = temp.first;
                min = temp.second;
            }
        }
    }

    return res;
}
```

Листинг 3.5 – Функция полного перебора словаря с дополнительными потоками

```
std::vector<std::wstring> full_search_mt(
    std::map<wchar_t, std::vector<std::wstring>> &words,
    const std::wstring &word, size_t k, size_t max_errors,
       size_t num_threads){

    std::vector<std::wstring> res;
    size_t min = word.size();

    for (const auto &p: words){
        if (p.first != word[0]){
            auto temp = get_closest_words_mt(p.second, word, k,
               max_errors, num_threads);

            if (temp.second < min){
                res = temp.first;
                min = temp.second;
            }
        }
    }

    return res;
}
```

Листинг 3.6 – Функция исправления ошибок

```
std::pair<std::vector<std::wstring>, size_t> get_closest_words(
    const std::vector<std::wstring> &words, const std::wstring
       &word,
    size_t k, size_t max_errors) {

    std::vector<std::wstring> temp;
    size_t min = word.size();

    size_t errors = std::min(static_cast<size_t>(std::ceil(0.3 *
       word.size())), max_errors);

    for (const auto &cur_word: words) {
        int dist = lev_mtr(cur_word, word);
        if (dist < min && dist <= errors) {
            temp.clear();
            temp.push_back(cur_word);
            min = dist;
        } else if (dist == min && dist <= errors && temp.size()
           < k)
            temp.push_back(cur_word);
    }

    return {temp, min};
}
```

Листинг 3.7 – Функция многопоточного исправления ошибок

```
std::pair<std::vector<std::wstring>, size_t>
    get_closest_words_mt(
      const std::vector<std::wstring> &words, const std::wstring
        &word,
      size_t k, size_t max_errors, size_t num_threads) {

      size_t min = word.size();
      size_t errors = std::min(static_cast<size_t>(std::ceil(0.3 *
        word.size())), max_errors);
      std::vector<std::wstring> collector;
      std::thread threads[num_threads];

      size_t range_step = words.size() / num_threads;

      for (size_t i = 0; i < num_threads; ++i) {
          if (i != num_threads - 1) {
              threads[i] = std::thread(
                  compute_distance, words,
                  word, errors, k,
                  range_step * i, range_step * (i + 1),
                  std::ref(min), std::ref(collector));
          } else
              threads[i] = std::thread(
                  compute_distance, words,
                  word, errors, k,
                  range_step * i, words.size(),
                  std::ref(min), std::ref(collector));
      }

      for (size_t i = 0; i < num_threads; ++i) {
          threads[i].join();
      }

      return {collector, min};
}
```

Листинг 3.8 – Функция, выполняющаяся в потоке

```
std::mutex mutex;

void compute_distance(const std::vector<std::string> &words,
                      const std::string &word_er,
                      size_t errors,
                      size_t k,
                      size_t start,
                      size_t stop,
                      size_t &min,
                      std::vector<std::string> &collector) {
    for (size_t i = start; i < stop; ++i) {
        const auto word_cor = words[i];
        int dist = lev_mtr(word_cor, word_er);

        mutex.lock();
        if (dist < min && dist <= errors) {
            collector.clear();
            collector.push_back(word_cor);
            min = dist;
        } else if (dist == min && dist <= errors &&
           collector.size() < k)
            collector.push_back(word_cor);
        mutex.unlock();
    }
}
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для разработанных алгоритмов исправления ошибок. Для данных тестов максимальное количество ошибок равно двум и из массива выбираются 3 слова. Все тесты пройдены успешно. В таблице 3.1 пустое слово обозначается с помощью λ.

Таблица 3.1 – Функциональные тесты

Корпус	**Слово**	**Ожидаемый результат**
[мама, мыла, раму]	мама	[мама]
[мама, мыла, раму]	мамы	[мама]
[мама, мыла, раму]	мыма	[мама, мыла]
[мама, мыла, раму]	ахтунг	[ ]
[ ]	ахтунг	[ ]
[ ]	λ	[ ]
[Мама, Мыла, Раму]	λ	[ ]

Вывод

Были разработаны и протестированы спроектированные алгоритмы исправления ошибок.

4 Исследовательский раздел

В данном разделе будут приведены: пример работы программы, постановка исследования и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Демонстрация работы программы

На рисунках 4.1 и 4.2 представлена демонстрация работы разработанного программного обеспечения, а именно показаны результаты исправления ошибок в слове *policu*.

```
Меню:
1 - Прочитать корпус
2 - Проверить слово (без потоков)
3 - Проверить слово (с потоками)
4 - Функциональное тестирование
5 - Замеры времени
0 - Выйти

Выберите пункт меню:1
 1
Корпус считан
Меню:
1 - Прочитать корпус
2 - Проверить слово (без потоков)
3 - Проверить слово (с потоками)
4 - Функциональное тестирование
5 - Замеры времени
0 - Выйти

Выберите пункт меню:2
 2
Введите слово:policu
 policu
Максимальное число ошибок:2
 2
Сколько слов вывести:5
 5
Результат:
police
policy
```

Рисунок 4.1 – Демонстрация работы программы при исправлении слова без потоков

```
Меню:
1 - Прочитать корпус
2 - Проверить слово (без потоков)
3 - Проверить слово (с потоками)
4 - Функциональное тестирование
5 - Замеры времени
0 - Выйти

Выберите пункт меню:3
 3
Введите слово:policu
 policu
Максимальное число ошибок:2
 2
Сколько слов вывести:5
 5
Количество потоков:4
 4
Результат:
police
policy
```

Рисунок 4.2 – Демонстрация работы программы при исправлении слова с дополнительными потоками

Технические характеристики устройства, на котором выполнялись замеры по времени, следующие:

— процессор: AMD Ryzen 5 4600H 3 ГГц, 6 физических ядер, 12 логических процессоров [12];

— оперативная память: 16 ГБайт;

— операционная система: Windows 10 Pro 64-разрядная система версии 22H2 [13].

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Время выполнения реализаций алгоритмов

Результаты замеров времени выполнения реализации алгоритма исправления ошибок в зависимости от числа потоков приведены в таблице 4.1. Замеры времени проводились на корпусе длины 10000, состоящего из слов длины 10. Замеры времени проводились на корпусах одного размера и усреднялись для каждого набора одинаковых экспериментов.

Таблица 4.1 – Время работы реализации алгоритма исправления ошибок (в мс)

Количество потоков	Время исправления ошибки (мс)
1	3.996
2	3.208
3	4.192
4	5.388
5	6.712
6	7.061
7	8.252
9	8.382
11	8.500
13	8.368
15	8.615
17	9.571
19	9.775
21	10.24
23	11.45
25	12.20
27	12.51
29	13.28
31	14.10
33	14.81
35	15.16
37	15.47
39	16.83
41	17.32
43	17.99
45	18.84
47	19.69
49	20.84

Результаты замеров времени выполнения однопоточной и многопоточной реализаций алгоритма исправления ошибок в зависимости от размеров корпуса приведены в таблице 4.2. Замеры времени проводились на корпусах одного размера и усреднялись для каждого набора одинаковых экспериментов.

Таблица 4.2 – Время работы реализаций алгоритма исправления ошибок (в мс)

Размер корпуса	С доп. потоком (мс)	Без доп. потоков (мс)
1000	$1.536 \cdot 10^{-1}$	$7.337 \cdot 10^{-1}$
2000	$3.722 \cdot 10^{-1}$	1.472
3000	$6.929 \cdot 10^{-1}$	2.162
4000	1.015	2.934
5000	1.173	3.675
6000	1.831	4.418
7000	2.147	5.187
8000	2.715	5.985
9000	3.285	6.743
10000	4.135	7.384

На рисунке 4.3 приведен график зависимости времени выполнения реализации от числа потоков. На рисунке 4.4 приведен график зависимости времени выполнения реализаций от размеров корпуса.

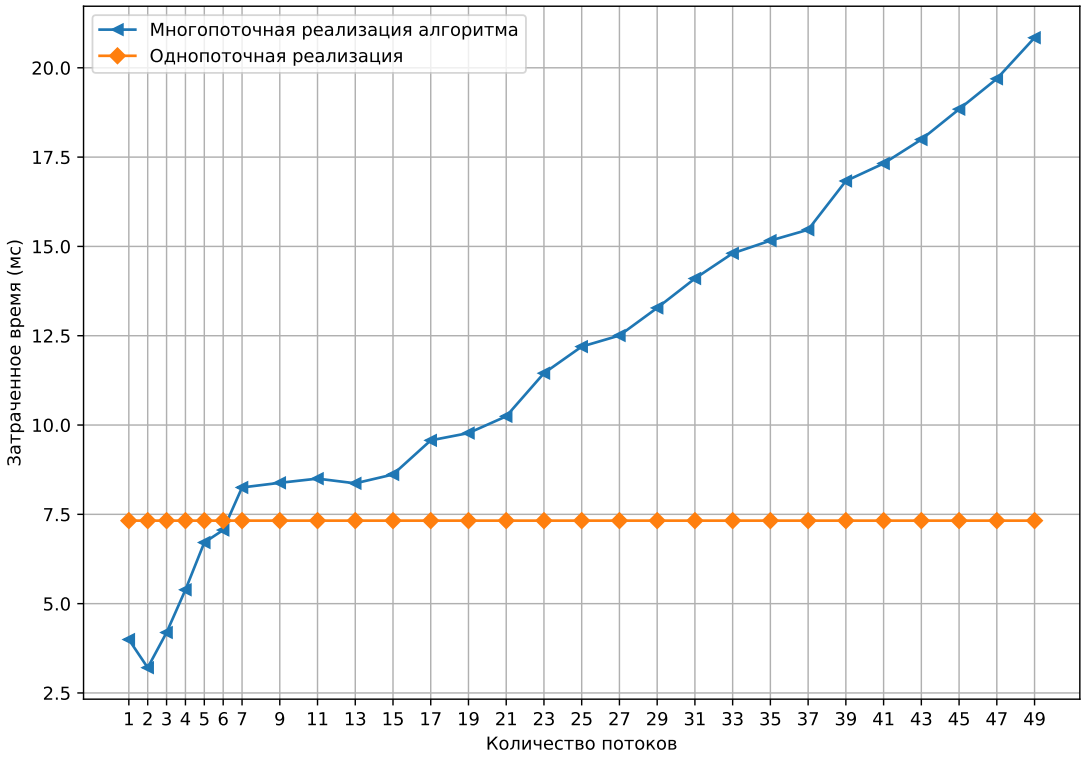

Рисунок 4.3 – График зависимости времени выполнения реализации от числа потоков

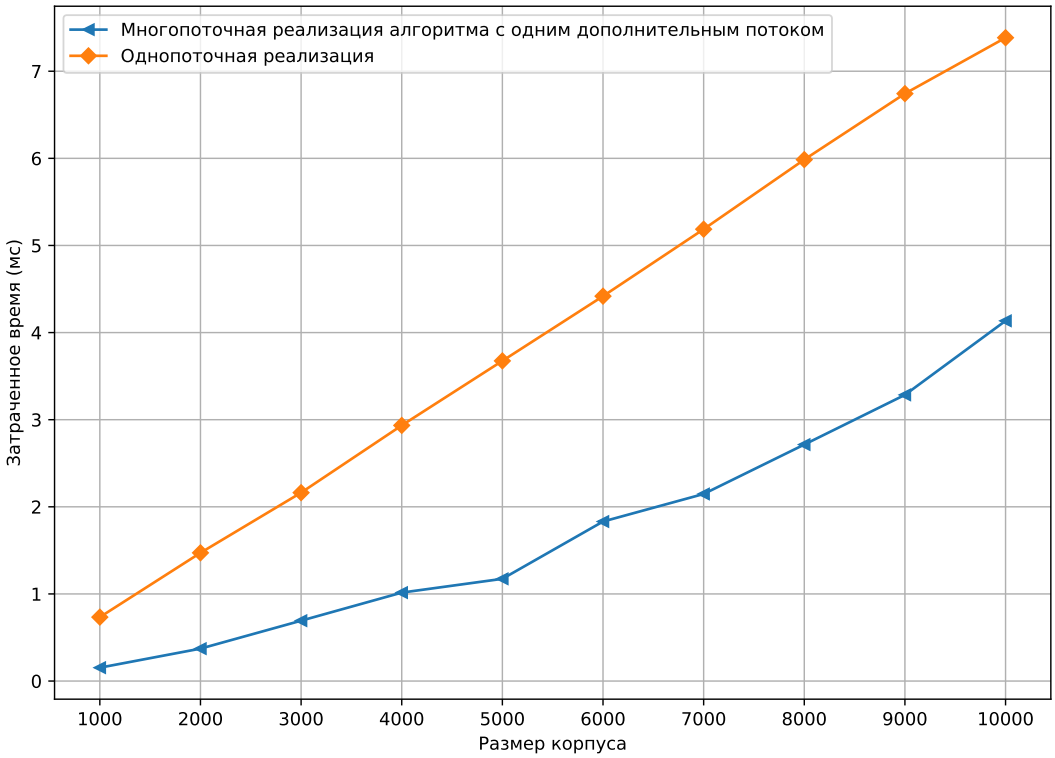

Рисунок 4.4 – График зависимости времени выполнения реализаций от размеров корпуса

Вывод

В результате анализа таблицы 4.1, было получено, что время выполнения многопоточной реализации алгоритма меньше времени выполнения однопоточной реализации при количестве потоков не больше 6, т. е. при количестве потоков не больше количества физических ядер. При двух дополнительных потоках многопоточная реализации быстрее однопоточной в 2.28 раза.

Из таблицы 4.2, можно сделать выводы, что при увеличении числа слов в корпусе время получения результата с использованием нескольких потоков также увеличивается. При увеличении размеров корпуса с 1000 до 10000, время получения результата без использования дополнительных потоков увеличилось в 10 раз, а время получения результата с использованием одного потока увеличилось в 27 раз. При этом при 10000 слов в корпусе реализация с одним дополнительным потоком быстрее реализации без дополнительных потоков в 1.8 раза. Таким образом, рекомендуется использование двух вспомогательных потоков, т. к. это позволит осуществлять поиск по корпусу с минимальными

тратами на переключения контекста.

ЗАКЛЮЧЕНИЕ

В результате исследования реализаций было получено, что время выполнения многопоточной реализации алгоритма меньше времени выполнения однопоточной реализации при количестве потоков не больше 6, т. е. когда количество потоков не больше количества физических ядер. При двух дополнительных потока многопоточная реализации быстрее однопоточной в 2.28 раза.

При увеличении числа слов в корпусе время получения результата с использованием нескольких потоков также увеличивается. При увеличении размеров корпуса с 1000 до 10000, время получения результата без использования дополнительных потоков увеличилось в 10 раз, а время получения результата с использованием одного потока увеличилось в 27 раз. При этом при 10000 слов в корпусе реализация с одним дополнительным потоком быстрее реализации без дополнительных потоков в 1.8 раза. Таким образом, рекомендуется использование двух вспомогательных потоков, т. к. это позволит осуществлять поиск по корпусу с минимальными тратами на переключения контекста.

Цель данной лабораторной работы была достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Гладышев Е.*, *Мурыгин А.* Многопоточность в приложениях // Актуальные проблемы авиации и космонавтики. — 2012. — № 8.
2. *Stoltzfus J.* Multithreading [Электронный ресурс]. — Режим доступа: `https://www.techopedia.com/definition/24297/multithreading-computer-architecture` (дата обращения: 07.12.2023).
3. *Ричард Стивенс У.*, *Стивен Раго А.* UNIX. Профессиональное программирование. 3-е издание. — СПб.: Питер, 2018. — С. 994.
4. *Большакова Е.*, *Клышинский Э.* Автоматическая обработка текстов на естественном языке и компьютерная лингвистика //. — М.: МИЭМ, 2011. — С. 122—124.
5. C++ reference [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/` (дата обращения: 20.12.2023).
6. Concurrency support library [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/cpp/thread` (дата обращения: 20.12.2023).
7. std::clock [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/cpp/chrono/c/clock` (дата обращения: 20.12.2023).
8. Strings library [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/cpp/string` (дата обращения: 20.12.2023).
9. std::vector [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/cpp/string` (дата обращения: 20.12.2023).
10. std::map [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/cpp/container/map` (дата обращения: 22.12.2023).
11. std::mutex [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/cpp/thread/mutex` (дата обращения: 20.12.2023).
12. Amd [Электронный ресурс]. — Режим доступа: `https://www.amd.com/en.html` (дата обращения: 20.12.2023).

13. Windows 10 Pro 22h2 64-bit [Электронный ресурс]. — Режим доступа: https://www.microsoft.com/ru-ru/software-download/windows10 (дата обращения: 20.12.2023).