



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»
(ИУ7)

НАПРАВЛЕНИЕ ПОДГОТОВКИ 09.03.01 Информатика и вычислительная техника

О Т Ч Е Т

по лабораторной работе № 4

Название: Параллельное программирование

Дисциплина: Анализ алгоритмов

Студент

ИУ7-54Б

(Группа)

Елгин И.Ю.

(Подпись, дата)

(И.О. Фамилия)

Преподаватель

Волкова Л.Л.

(Подпись, дата)

(И.О. Фамилия)

Москва, 2021

Оглавление

Введение	3
1 Аналитическая часть	4
1.1 Описание необходимых операций	4
1.2 Используемые алгоритмы	5
1.3 Вывод	5
2 Конструкторская часть	6
2.1 Алгоритм Дамерау-Левенштейна	6
2.2 Стандартный алгоритм поиска слова	7
2.3 Паралельный алгоритм	9
2.4 Вывод	10
3 Технологическая часть	11
3.1 Выбор языка программирования	11
3.2 Реализация потоков	11
3.3 Листинг кода	12
3.4 Результаты тестирования	17
3.5 Оценка времени	18
3.6 Вывод	18
4 Исследовательская часть	19
4.1 Результаты экспериментов	19
4.2 Вывод	20
Заключение	22
Список литературы	23

Введение

В данной лабораторной работе реализуется и оценивается параллельный алгоритм поиска наиболее похожего слова с использованием алгоритма Дамерау-Левенштейна.

Параллелизм – выполнение нескольких вычислений в различных потоках. При параллельном программировании в процессоре с многоядерной архитектурой несколько процессов могут выполняться одновременно на разных ядрах. Это приводит к тому, что время выполнения параллельного алгоритма может быть ощутимо меньше, чем у его однопоточного аналога. Однако, на создание потоков тратится время, в связи с чем на малых данных простой алгоритм может показывать лучшие результаты по времени.

Стандартный алгоритм поиска наиболее похожего слова в словаре сводится к проходу по всему словарю и применению алгоритма Дамерау-Левенштейна для введённого слова и очередного слова из словаря. Наименьший результат указывает на наиболее похожее слово в словаре.

Сравнение очередного слова из словаря с введённым является отдельной процедурой и не зависит от других сравнений. Поэтому мы можем разбить словарь на участки и запустить для них потоки.

1. Аналитическая часть

Целью лабораторной работы является разработка и исследование параллельных алгоритмов поиска наиболее похожего слова в словаре.

Можно выделить следующие задачи лабораторной работы:

- описание алгоритма поиска похожего слова в словаре;
- распаралеливание данного алгоритма;
- проведение замеров процессорного времени работы алгоритмов при разном количестве потоков;
- анализ полученных результатов.

1.1. Описание необходимых операций

Наиболее подходящим словом будем называть слово из словаря с наименьшим результатом выполнения алгоритма Дамерау-Левенштейна.

Алгоритм Дамерау-Левенштейна вычисляет минимальное количество операций необходимых для получения одного слова из другого.

Операции возможные над словом :

Действия обозначаются так:

1. D (англ. delete) — удалить;
2. I (англ. insert) — вставить;
3. R (replace) — заменить;
4. M (match) - совпадение;
5. T (transposition) - транспозиция в алгоритме Дамерау - Левенштейна.

В качестве алгоритма Дамерау-Левенштейна выберем алгоритм с тремя строками в виду его быстродействия и экономии памяти [3].

1.2. Используемые алгоритмы

Алгоритм подразумевает проход по словарю и выполнение алгоритма Дамерау-Левенштейна для каждой записи.

Параллелизм может быть достигнут за счёт выделения процессов, которые могут выполняться независимо друг от друга. В данном случае для каждой записи мы можем независимо вычислить результат Дамерау-Левенштейна.

1.3. Вывод

Результатом аналитического раздела стало определение цели и задач работы, описано понятие операций необходимых для реализации алгоритма.

2. Конструкторская часть

В данном разделе рассмотрим описанные алгоритмы нахождения наиболее похожего слова в словаре с.

2.1. Алгоритм Дамерау-Левенштейна

Схема алгоритма Дамерау-Левенштейна приведена на рисунке 2.1, 2.2.

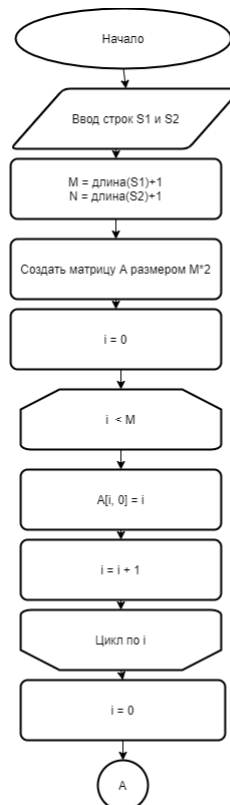


Рис. 2.1: Схема алгоритма Дамерау-Левенштейна часть 1

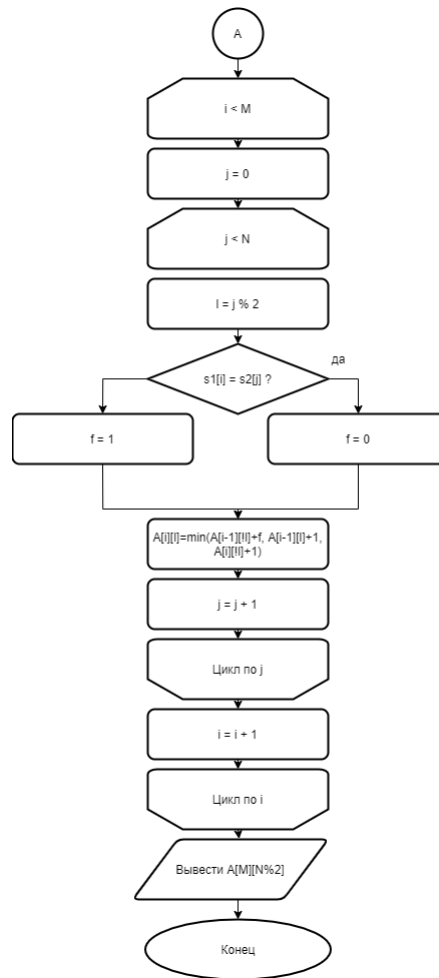


Рис. 2.2: Схема алгоритма Дамерау-Левенштейна часть 2

2.2. Стандартный алгоритм поиска слова

Схема простого алгоритма приведена на рисунке 2.3.

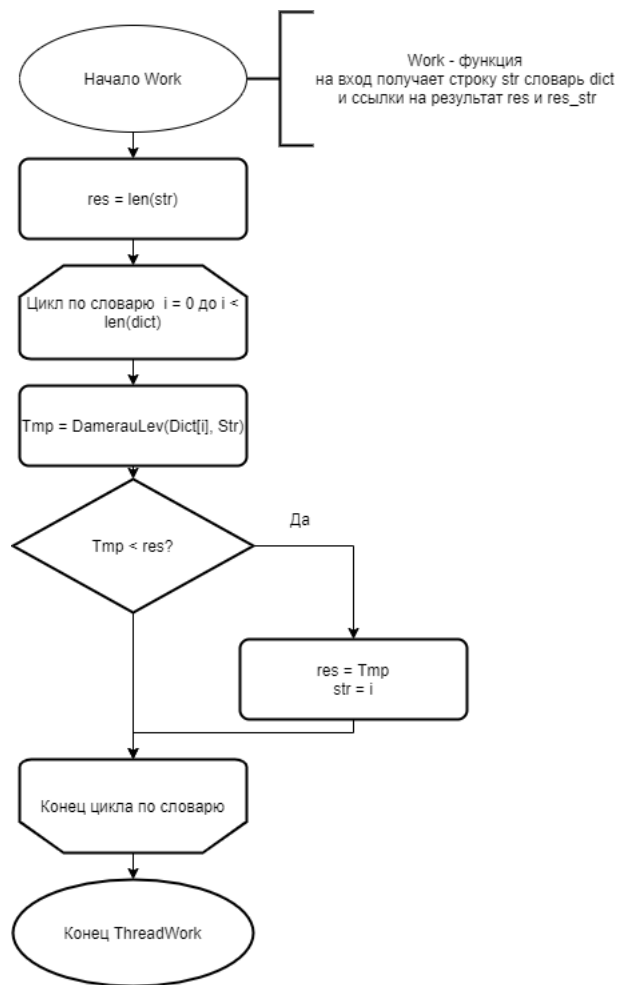


Рис. 2.3: Схема простого алгоритма

2.3. Паралельный алгоритм

Вычисление значений алгоритма Дамерау-Левенштейна для введённой строки и строк из словаря не зависимо для каждой строки словаря, следовательно может быть распаралелено.

Пусть производится работа с T потоками а всего записей в словаре N . В таком случае, i -й поток будет производить вычисление с $(i - 1) * (N/T + (N - i) \% (N \% T))$ по $i * N/T + (N - i) \% (N \% T)$ Схема паралельного алгоритма приведена на рисунке 2.4.

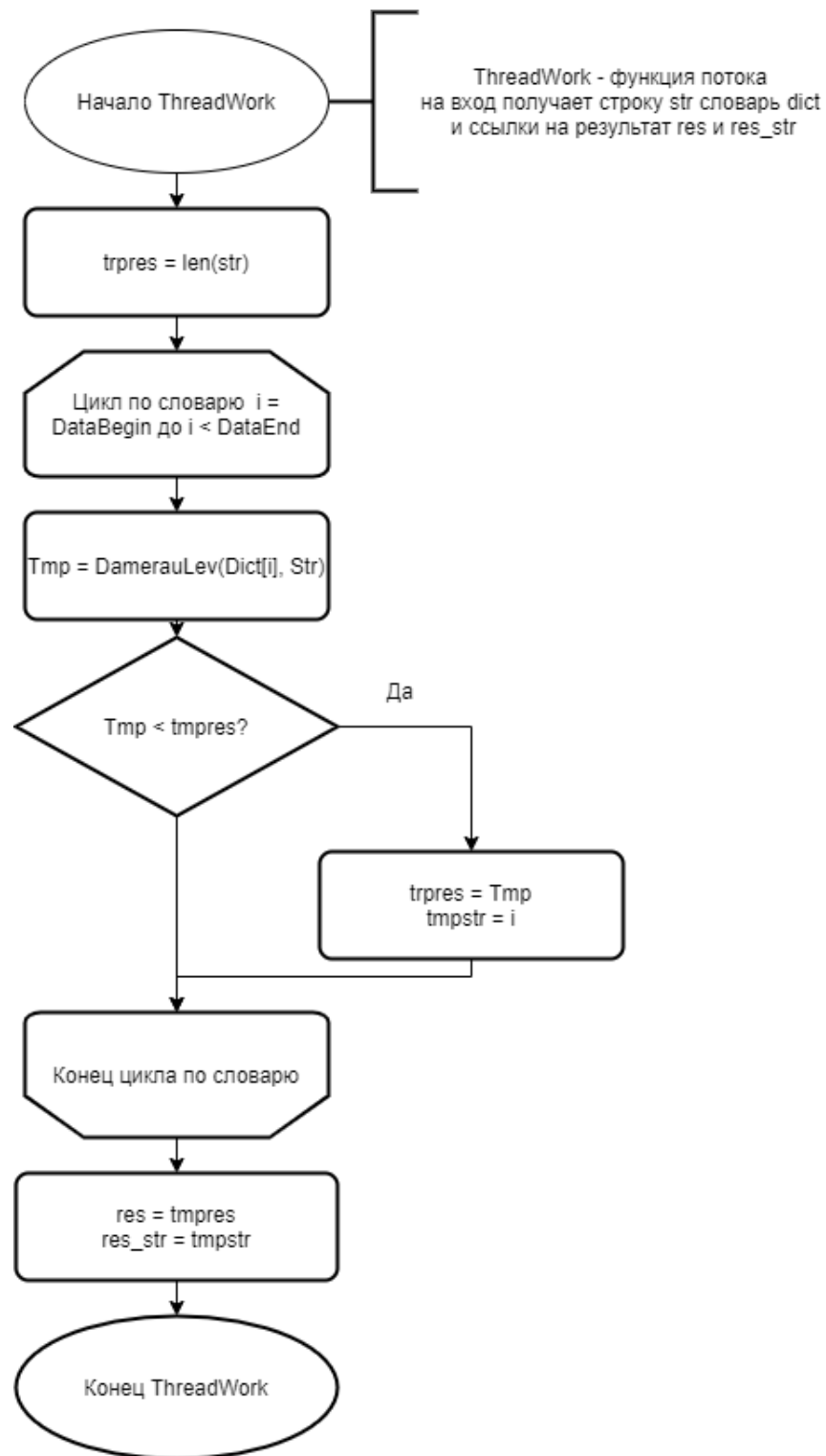


Рис. 2.4: Схема паралельного алгоритма

2.4. Вывод

Результатом конструкторской части стало схематическое описание алгоритмов поиска наиболее похожего слова в словаре, сформулированы тесты и требования к программному обеспечению.

3. Технологическая часть

В данной главе происходит выбор языка программирования и реализация алгоритмов на выбранном языке.

3.1. Выбор языка программирования

В качестве языка программирования был выбран C++, так как имеется опыт работы с ним. Так же для данного языка существуют библиотеки, с помощью которых можно удобно создавать потоки.

Разработка проводилась в среде QtCreator.

3.2. Реализация потоков

Потоки реализуются с помощью библиотеки *Threads* [1].

Для избежания неправильных ответов в результате гонок потоков был использован мьютекс из библиотеки *Mutex* [4].

Для передачи данных в функцию потока была создана структура включающая ссылки на результат, данные обрабатываемые потоком и мьютекс листинг 3.1.

Листинг 3.1: Структура данных

```
1 struct args{
2     args(int &result0 ,int &res_string0 ,int data_start0 ,int data_end0 ,
3         result(result0) ,
4         res_string(res_string0) ,
5         data_start(data_start0) ,
6         data_end(data_end0) ,
7         data(data0) ,
8         serch_str(serch_str0) ,
9         mtx(mtx0)
10    };
11    int &result ;
12    int &res_string ;
13    int data_start ;
14    int data_end ;
15    std::vector <std::string> &data ;
16    std::string &serch_str ;
17    std::mutex &mtx ;
18 };
```

3.3. Листинг кода

В листингах 3.2 - 3.6 приведены реализации поиска наиболее похожего слова в словаре.

Листинг 3.2: функция алгоритма Дамерау-Левенштейна

```

1  int DemLevAlg(std::string &str1 , std::string &str2)
2  {
3      int f;
4      int l, l1 , l2;
5
6      int n = str1.length() + 1, m = str2.length() + 1;
7
8      int matrix[n + 1][3];
9      for (int i = 0; i < n; i++)
10         matrix[i][0] = i;
11
12     for (int j = 1; j < m; j++)
13     {
14         l = j % 3;
15         l1 = (l + 2) % 3;
16         l2 = (l + 1) % 3;
17         matrix[0][l] = j;
18         for (int i = 1; i < n; i++)
19         {
20             if (str1[i] == str2[j])
21                 f = 0;
22             else
23                 f = 1;
24             if (i != 1 and j != 1 and str1[i] == str1[j-1] and
25                 matrix[i][l] = min(matrix[i-2][l2] + 1, matrix
26             else
27                 matrix[i][l] = min(matrix[i-1][l1] + f, matrix
28         }
29     }
30     return matrix[n - 1][(m - 1)%3];
31 }

```

Листинг 3.3: функция потоков и их создания

```

1 void thread_work( args arg )
2 {
3     int temp_res = DemLevAlg( arg.data.at( arg.data_start ),
4     int tmp;
5     for (int i = arg.data_start; i < arg.data_end; i++)
6     {
7         tmp = DemLevAlg( arg.data.at( i ), arg.serch_str );
8         if (tmp < temp_res)
9         {
10             temp_res = tmp;
11             temp_str = i;
12         }
13     }
14     arg.mtx.lock();
15     if (arg.result > temp_res)
16     {
17         arg.result = temp_res;
18         arg.res_string = temp_str;
19     }
20     arg.mtx.unlock();};
21
22 void run_theared( int th_count, int &res_str, int &res_int, int
23 {
24     std::thread thr[th_count];
25     int ud_data = all_data / th_count;
26     std::mutex mtx;
27     for (int i = 0; i < th_count - 1; i++)
28     {
29         args arg( res_int, res_str, i * ud_data, ud_data * (i +
30         thr[i] = std::thread( thread_work, arg );
31     }
32     args arg( res_int, res_str, (th_count - 1) * ud_data, all_da
33     thr[th_count - 1] = std::thread( thread_work, arg );
34     for (int i = 0; i < th_count; i++)
35         thr[i].join();}

```

Листинг 3.4: функции загрузки словаря

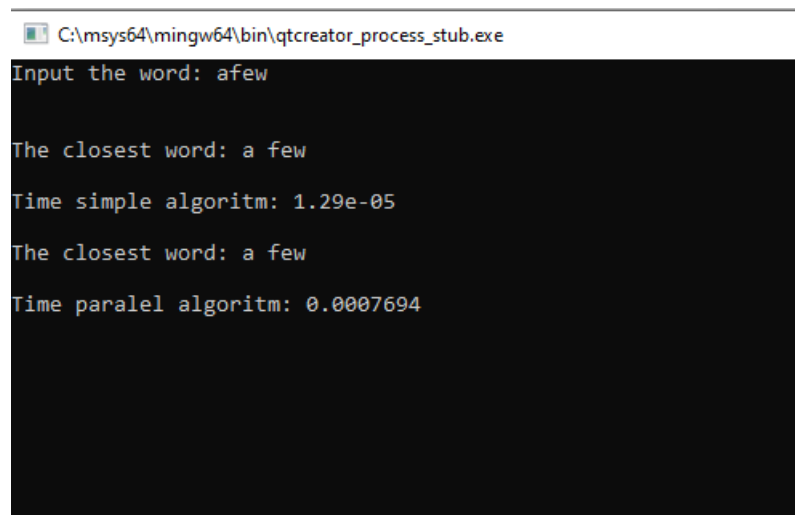
```
1 void LoadDictionary(std::vector<std::string> &dictionary, std
2 {
3     int n = min(string_count(name), dic_len);
4     char str[M_STR];
5     dictionary.resize(n);
6     std::ifstream f(name);
7     for (int i = 0; i < n; i++)
8     {
9         if (f.getline(str, M_STR, '\n'))
10             dictionary.at(i) = str;
11     }
12 }
```

Листинг 3.5: функция main

```
1 int main(int argc , char *argv [])
2 {
3     std::vector <std::string> dictionary;
4     std::string file_name = "ENRUS.txt", serch_str;
5     std::cout << "Input the word: ";
6     std::cin >> serch_str;
7     LoadDictionary(dictionary , file_name);
8     int res = serch_str.size(), res_str = 0, tmp = 0;
9     StartCounter();
10    for (int i = 0; i < dictionary.size(); i++)
11    {
12        tmp = DemLevAlg(dictionary.at(i), serch_str);
13        //std::cout << tmp << '\n';
14        if (tmp < res)
15        {
16            res = tmp;
17            res_str = i;
18        }
19    }
20    double time = GetCounter();
21    std::cout << "\n\nThe closest word: " << dictionary[res_str];
22    std::cout << "\n\nTime simple algoritm: " << time;
23    res = serch_str.size();
24    StartCounter();
25    run_theared(thread_count, res_str, res, dictionary.size(),
26    time = GetCounter());
27    std::cout << "\n\nThe closest word: " << dictionary[res_str];
28    std::cout << "\n\nTime paralel algoritm: " << time;
29    return 0;
30
31 }
```


3.4. Результаты тестирования

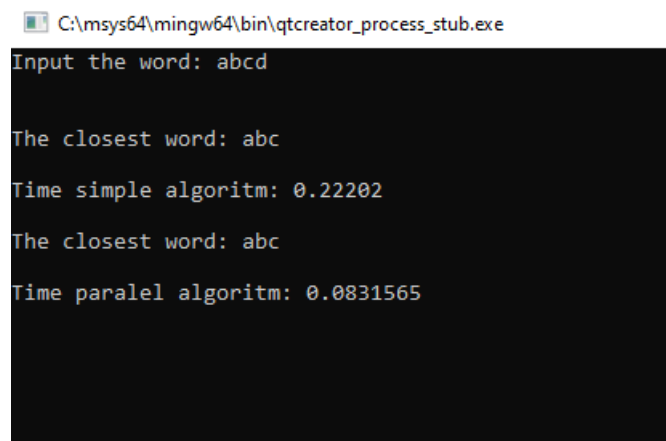
На рисунках 3.1 - 3.2 приведены скриншоты интерфейса программы и тестирования, которые проводились в ручную.



```
C:\msys64\mingw64\bin\qtcreeator_process_stub.exe
Input the word: afew

The closest word: a few
Time simple algoritm: 1.29e-05
The closest word: a few
Time paralel algoritm: 0.0007694
```

Рис. 3.1: Пример 1



```
C:\msys64\mingw64\bin\qtcreeator_process_stub.exe
Input the word: abcd

The closest word: abc
Time simple algoritm: 0.22202
The closest word: abc
Time paralel algoritm: 0.0831565
```

Рис. 3.2: Пример 2

Программа работает корректно.

3.5. Оценка времени

Для замера процессорного времени исполнения функции используется функция `QueryPerfomanceCounter` библиотеки `windows.h`. []

Измерение производится в функциях, которые приведены в листинге 3.6.

Листинг 3.6: функции замера процессорного времени

```
1 double PCFreq = 0.0;
2 __int64 CounterStart = 0;
3
4 void StartCounter ()
5 {
6     LARGE_INTEGER li ;
7     if (!QueryPerformanceFrequency(&li ))
8         std::cout << "QueryPerformanceFrequency failed!\n";
9
10    PCFreq = double(li.QuadPart)/1000.0;
11
12    QueryPerformanceCounter(&li );
13    CounterStart = li.QuadPart;
14 }
15
16 double GetCounter ()
17 {
18     LARGE_INTEGER li ;
19     QueryPerformanceCounter(&li );
20     return double(li.QuadPart-CounterStart)/PCFreq;
21 }
```

3.6. Вывод

Результатом технологической части стал выбор используемых технических средств реализации и последующая реализация алгоритмов и замера времени работы на языке C++.

4. Исследовательская часть

Измерения процессорного времени проводятся на словарях с размерами: 100, 1000, 10000, 100000. Средняя длина слова 7 букв. Изучается серия экспериментов с количеством потоков 1, 2, 3, 4, 8, 16.

Для повышения точности, каждый замер производится 5 раз, за конечный результат берётся среднее арифметическое.

4.1. Результаты экспериментов

Эксперименты проводились на компьютере со следующими характеристиками:

- ОС - Windows 10, 64bit;
- Процессор - Intel Core i3 7th Gen 2.3GHz, 2 Core 4 Logical Processor
- ОЗУ - 8Gb

По результатам измерений процессорного времени можно составить таблицы 4.1 - 4.4.

Таблица 4.1: Результаты замеров процессорного времени при размере 100 (в миллисекундах)

Потоки	1	2	4	8	16
Многопоточность	5.52	2.73	2.80	2.88	3.09
Однопоточно	0.96				

Таблица 4.2: Результаты замеров процессорного времени при размере 1000 (в секундах)

Потоки	1	2	4	8	16
Многопоточность	0.0064	0.0054	0.0050	0.058	0.060
Однопоточно	0.0051				

Таблица 4.3: Результаты замеров процессорного времени при размере 10000 (в секундах)

Потоки	1	2	4	8	16
Многопоточность	0.037	0.025	0.012	0.012	0.014
Однопоточно	0.032				

Таблица 4.4: Результаты замеров процессорного времени при размере 100000 (в секундах)

Потоки	1	2	4	8	16
Многопоточность	0.61	0.38	0.17	0.18	0.21
Однопоточно	0.55				

4.2. Вывод

По результатам экспериментов можно заключить следующее:

- при относительно небольшом размере словаря (менее 1000) использование потоков для уменьшения времени исполнения нецелесообразно, так как накладные расходы времени на управление потоками и mutex-ами больше, чем выигрыш от параллельного выполнения вычислений;
- использование по крайней мере двух потоков даёт ощутимый выигрыш по времени по сравнению с однопоточной версией алгоритма;
- использование одного потока в многопоточных версиях алгоритма проигрывает по времени по сравнению с однопоточной версией алгоритма, что объясняется накладными расходами времени на управление потоками и mutex-ами;
- использование 8 и 16 потоков показывает результат по времени несколько хуже, чем при 4 потоках, из чего следует, что увеличение потоков даёт выигрыш по времени лишь до достижения определённого количества, так как появляются большие накладные затраты по времени для управления большим количеством потоков и mutex-ов;

- наиболее быстродейственно алгоритм действует на 4 потоках, что равно количеству логических процессоров на испытуемом компьютере.

Заключение

В ходе лабораторной работы достигнута поставленная цель: разработка и исследование алгоритма поиска наиболее похожего слова в словаре. Решены все задачи.

Были описаны и реализованы непараллельная и параллельная реализации алгоритма. Проведены замеры процессорного времени работы алгоритмов при различном количестве потоков. На основании экспериментов проведён сравнительный анализ.

Из проведённых экспериментов было выявлено, что наиболее быстродейственным является использование количество потоков, которое совпадает с количеством логических процессоров процессора. Увеличение или уменьшение количества потоков ведёт к большему времени выполнения вычислений. Однако, использование потоков даёт выигрыш по времени работы только для размеров словаря более 1000 записей, иначе их использование лишь увеличит время вычислений за счёт накладных расходов.

Список литературы

1. Документация языка C++ 98 [Электронный ресурс], режим доступа: <http://www.open-std.org/JTC1/SC22/WG21/>, свободный (дата обращения: 14.10.2021).
2. R. A. Wagner, M. J. Fischer. The string-to-string correction problem. J. ACM 21 1 (1974). P. 168—173
3. В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. 163.4:845-848.
4. Mutex C++ [Электронный ресурс].-Режим доступа: <https://www.cplusplus.com/reference/mutex/mutex/> (Дата обращения 21.10.2021)