САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

ПЕТРА ВЕЛИКОГО

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Отчет по лабораторной работе

Дисциплина: **Параллельные вычисления**

Тема: **«Разработать программу для вычисления вероятностей появления триграмм с использованием модели OpenMP»**

Работу выполнил студент

группы 13541/2

Я.А. Селиверстов

Преподаватель

И.В. Стручков

Санкт-Петербург

2019

Оглавление

1. ТЕХНИЧЕСКОЕ ЗАДАНИЕ (ТЗ)

Требуется разработать программу для определения вероятностей появления 3-грамм в тексте на русском языке.

Программа должна быть реализована с использованием следующих моделей программирования:

1. Параллельная реализация при помощи POSIX threads;
2. Параллельная реализация при помощи технологии OpenMP.

Оценить скорости работы программы в зависимости от моделей программирования 1- реализация при помощи POSIX threads и 2 - реализация при помощи технологии OpenMP, а так же от количества потоков (процессов).

2. ОПИСАНИЕ ТЕХНИЧЕСКИХ ХАРАКТЕРИСТИК ОБОРУДОВАНИЯ

Процессор Intel (R) Core i5 - 4670K (рис.2.1.)

– 4 физических ядра (рис.2.2.);
– 8 логических ядер;
– 10 потоков;
– Базовая тактовая частота 3.4 GHz

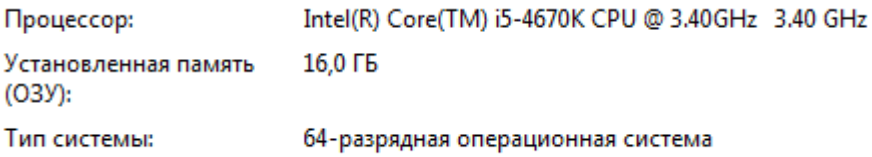

Рис.2.1. Тип процессора

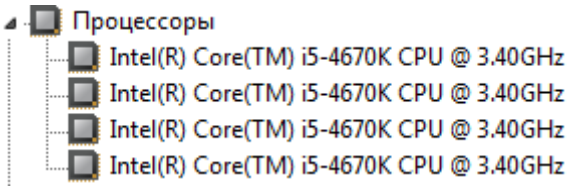

Рис.2.2. Логические ядра

3. РАСЧЕТ ВЕРОЯТНОСТИ ПОЯВЛЕНИЯ ТРИГРАММ

Текст состоит из слов, слова из букв. Количество различных букв в каждом языке ограничено и буквы могут быть просто перечислены. Важными характеристиками текста являются повторяемость букв, пар букв (биграмм), троек букв (триграмм) и вообще m-ок (m-грамм), сочетаемость букв друг с другом, чередование гласных и согласных и некоторые другие.

Таким образом, триграмма - это сочетание из трёх букв алфавита русского языка без пробелов (рис.3).

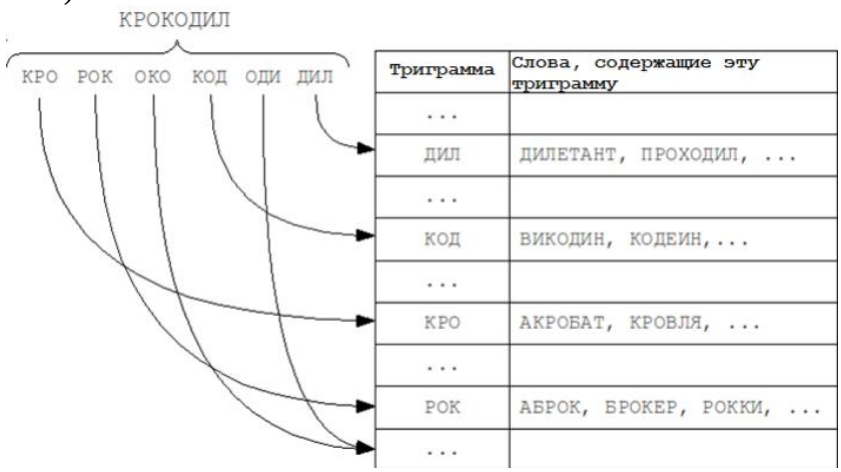

Рис.3.1. Триграммы

Идея состоит в подсчете чисел вхождений каждой n^m возможных m-грамм в достаточно длинных открытых текстах $T=t_1t_2...t_l$, составленных из букв алфавита $\{a_1, a_2, ..., a_n\}$. При этом просматриваются подряд идущие m-граммы текста:

$t_1t_2...t_m$, $t_2t_3...t_{m+1}$, ..., $t_{i-m+1}t_{l-m+2}...t_l$.

Если $\vartheta(a_{i1}, a_{i2}, \ldots, a_{im})$ - число появлений m-граммы $a_{i1}a_{i2}...a_{im}$ в тексте T, а L – общее число подсчитанных m-грамм, то относительную частоту появления данной m-граммы в случайно выбранном месте текста рассчитывают по формуле:

$$P(a_{i1}, a_{i2}, \ldots, a_{im}) = \frac{\vartheta(a_{i1}, a_{i2}, \ldots, a_{im})}{L} \quad (1)$$

В силу этого, относительную частоту (1) считают приближением вероятности $P(a_{i1}a_{i2}...a_{im})$ появления данной m-граммы в случайно выбранном месте текста (такой подход принят при статистическом определении вероятности).

Таким образом, для вычисления вероятностей появления их в тексте, будем подсчитывать количество встреченных триграмм каждого вида, после чего делить на общее количество встреченных триграмм.

Перед подсчетом вероятности необходимо провести предобработку текста – привести слова к нижнему регистру, удалить лишние символы, далее будем добавлять по ключу (триграмме) в словарь(unordered_map) новую запись, или инкрементировать существующую.

Параллелиться будет только время подсчета триграмм. Предобработка текста (фильтрация символов, приведение слов к нижнему регистру, удаление лишних символов), объединение результатов не участвуют в параллельном вычислении.

При расчете триграмм производится выборка двадцати наибольших значений из полученного словаря.

Далее эти значения делятся на общее количество найденных триграмм и выводятся в процентах с соответствующей триграммой.

Результат запуска программы представлен на рис.2.

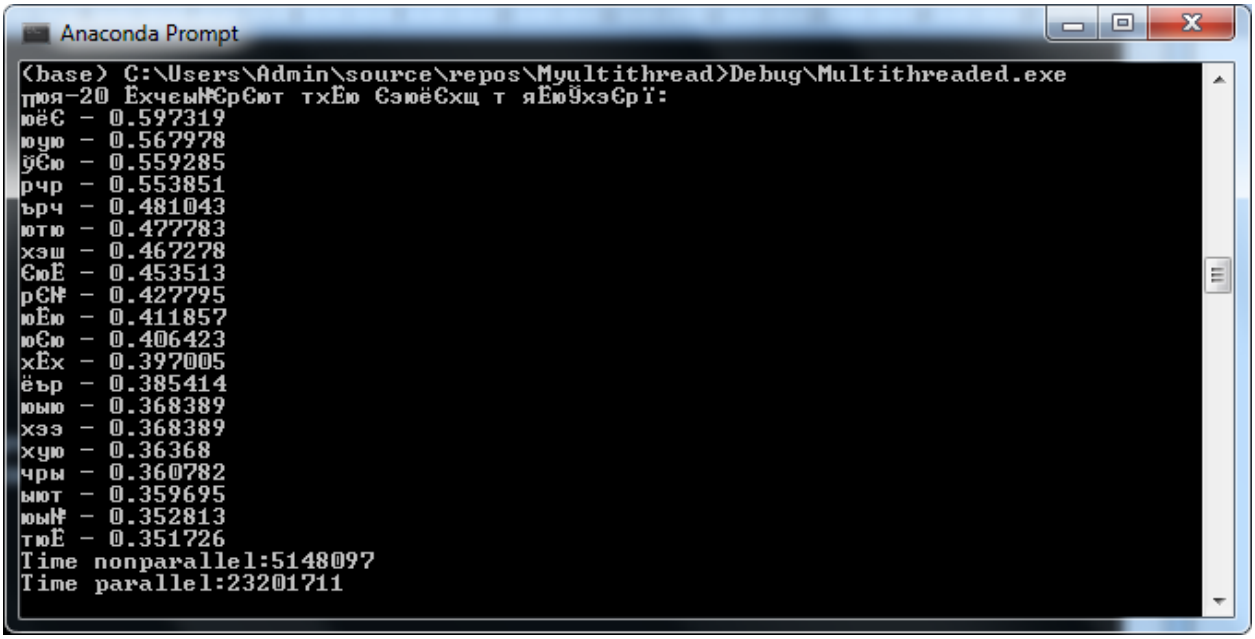

Рис.3.2. Расчет 20 наиболее часто встречающихся триграмм.

4.РЕАЛИЗАЦИЯ С ПОМОЩЬЮ БИБЛИОТЕКИ THREAD

Для распараллеливания рассмотренного алгоритма подсчёта вероятностей воспользуемся библиотекой thread (#include <thread>). Каждый поток будет использовать собственный участок текста и собственный словарь для результатов.

Так как используемый ПК имеет процессор с четырмя ядрами и возможностью исполнения 8 потоков (intel core i5 – 4670K) разделим выполняемую работу между потоками.

Код создания потоков (принимает функцию обработки потока и номер потока) и завершения их работы (метод Join, блокирующий основной поток до окончания завершения созданных) приведен в Приложении.

После исполнения, потребуется объединить словари каждого из потоков в один, для получения общего результата.

Результат выполнения программы совпадает с последовательной программой.

Для сравнения разработанных реализаций, была выполнена оценка производительности созданных программ на основе времён их исполнения.

В качестве тестовых данных использовалась обработка восьми романов Л.Н. Толстого «Война и мир».

5.РЕАЛИЗАЦИЯ С ПОМОЩЬЮ БИБЛИОТЕКИ OpenMP

OpenMP – стандарт, ориентированный на написание многопоточных приложений, включает в себя набор директив компилятора, библиотечных функций и переменных окружения. Директивы OpenMP применяются к структурированным блокам, в нашем случае это цикл for, для которого будет применена директива #pragma omp parallel for num_threads(NumOfTreads). При достижении директивы parallel порождается группа потоков, родительский поток также входит в эту группу как master thread, после завершения работы всех потоков продолжает исполнение master thread. Параметром num_threads можно установить необходимое количество потоков.

Для использования OpenMP необходимо подключить соответствующую библиотеку(#include <omp.h>), а также установить ключ компилятора /openmp. Таким образом, изменения в программе коснулись только момента создания потоков.

Код и результаты исполнения программы представлены в Приложении и Программа эксперимента соответственно.

Для сравнения разработанных реализаций, была выполнена оценка производительности созданных программ на основе времён их исполнения.

В качестве тестовых данных использовалась обработка восьми романов Л.Н. Толстого «Война и мир».

6. ПРОГРАММА ЭКСПЕРИМЕНТА

Результат исполнения для трёх реализаций по 5 запусков:

1 потока (SingleThread)

```
(base) C:\Users\Святослав\Desktop\патраллельные
вычисления\Myultithread>python t
ime.py Debug\SingleThread.exe –runsCount 5
Performing 5 runs of “Debug\SingleThread.exe”
 Outer elapased mean time = 5.76 +- 0.02 s (sigma = 16848
us)
```

2 потока

```
(base) C:\Users\Святослав\Desktop\параллельные
вычисления\Myultithread>test
(base) C:\Users\Святослав\Desktop\параллельные
вычисления\Myultithread>python t
ime.py Debug\SingleThread.exe –runsCount 5
Performing 5 runs of “Debug\SingleThread.exe”
 Outer elapased mean time = 5.96 +- 0.04 s (sigma = 35405
us)

(base) C:\Users\Святослав\Desktop\патраллельные
вычисления\Myultithread>python t
ime.py Debug\Multithreaded.exe –runsCount 5
Performing 5 runs of “Debug\Multithreaded.exe”
 Outer elapased mean time = 3,40+- 0.07 s (sigma = 54435 us)

(base) C:\Users\Святослав\Desktop\патраллельные
вычисления\Myultithread>python t
ime.py Debug\omp.exe –runsCount 5
Performing 5 runs of “Debug\omp.exe”
 Outer elapased mean time = 3.00 +- 0.04 s (sigma = 30645
us)
```

3 потока

```
(base) C:\Users\Святослав\Desktop\патраллельные
вычисления\Myultithread>python t
ime.py Debug\SingleThread.exe –runsCount 5
Performing 5 runs of “Debug\SingleThread.exe”
 Outer elapased mean time = 5.887 +- 0.013 s (sigma = 10475
us)

(base) C:\Users\Святослав\Desktop\патраллельные
вычисления\Myultithread>python t
ime.py Debug\Multithreaded.exe –runsCount 5
Performing 5 runs of “Debug\Multithreaded.exe”
 Outer elapased mean time = 2.88 +- 0.04 s (sigma = 35831
us)
```

```
(base) C:\Users\Святослав\Desktop\патраллельные
вычисления\Myultithread>python t
ime.py Debug\omp.exe –runsCount 5
Performing 5 runs of “Debug\omp.exe”
 Outer elapased mean time = 2.67 +- 0.11 s (sigma = 87880
us)
```

4 потока

```
(base) C:\Users\Святослав\Desktop\параллельные
вычисления\Myultithread>python t
ime.py Debug\SingleThread.exe –runsCount 5
Performing 5 runs of “Debug\SingleThread.exe”
 Outer elapased mean time = 5.765 +- 0.020 s (sigma = 16207
us)

(base) C:\Users\Святослав\Desktop\параллельные
вычисления\Myultithread>python t
ime.py Debug\Multithreaded.exe –runsCount 5
Performing 5 runs of “Debug\Multithreaded.exe”
 Outer elapased mean time = 2.462 +- 0.031 s (sigma = 25030
us)

(base) C:\Users\Святослав\Desktop\параллельные
вычисления\Myultithread>python t
ime.py Debug\omp.exe –runsCount 5
Performing 5 runs of “Debug\omp.exe”
 Outer elapased mean time = 2.29 +- 0.04 s (sigma = 34873
us)

(base) C:\Users\Святослав\Desktop\параллельные
вычисления\Myultithread>
```

5 потоков

```
(base) C:\Users\Святослав\Desktop\патраллельные
вычисления\Myultithread>python t
ime.py Debug\SingleThread.exe –runsCount 5
Performing 5 runs of “Debug\SingleThread.exe”
 Outer elapased mean time = 5.764 +- 0.020 s (sigma = 16340
us)

(base) C:\Users\Святослав\Desktop\патраллельные
вычисления\Myultithread>python t
ime.py Debug\Multithreaded.exe –runsCount 5
Performing 5 runs of “Debug\Multithreaded.exe”
 Outer elapased mean time = 2.374 +- 0.019 s (sigma = 15628
us)

(base) C:\Users\Святослав\Desktop\патраллельные
вычисления\Myultithread>python t
ime.py Debug\omp.exe –runsCount 5
Performing 5 runs of “Debug\omp.exe”
```

```
 Outer elapased mean time = 2.182 +- 0.015 s (sigma = 12084
us)

(base) C:\Users\Святослав\Desktop\патраллельные
вычисления\Myultithread>
```

6 потоков

```
(base) C:\Users\Святослав\Desktop\параллельные
вычисления\Myultithread>test

(base) C:\Users\Святослав\Desktop\параллельные
вычисления\Myultithread>python t
ime.py Debug\SingleThread.exe –runsCount 5
Performing 5 runs of “Debug\SingleThread.exe”
 Outer elapased mean time = 5.98 +- 0.05 s (sigma = 37159
us)

(base) C:\Users\Святослав\Desktop\параллельные
вычисления\Myultithread>python t
ime.py Debug\Multithreaded.exe –runsCount 5
Performing 5 runs of “Debug\Multithreaded.exe”
 Outer elapased mean time = 2.376 +- 0.021 s (sigma = 17294
us)

(base) C:\Users\Святослав\Desktop\параллельные
вычисления\Myultithread>python t
ime.py Debug\omp.exe –runsCount 5
Performing 5 runs of “Debug\omp.exe”
 Outer elapased mean time = 2.13 +- 0.03 s (sigma = 25566
us)
```

7 потоков

```
(base) C:\Users\Святослав\Desktop\параллельные
вычисления\Myultithread>python t
ime.py Debug\SingleThread.exe –runsCount 5
Performing 5 runs of “Debug\SingleThread.exe”
 Outer elapased mean time = 6.15 +- 0.24 s (sigma = 190024
us)

(base) C:\Users\Святослав\Desktop\параллельные
вычисления\Myultithread>python t
ime.py Debug\Multithreaded.exe –runsCount 5
Performing 5 runs of “Debug\Multithreaded.exe”
 Outer elapased mean time = 2.48 +- 0.15 s (sigma = 120840
us)

(base) C:\Users\Святослав\Desktop\параллельные
вычисления\Myultithread>python t
ime.py Debug\omp.exe –runsCount 5
Performing 5 runs of “Debug\omp.exe”
```

```
 Outer elapased mean time = 2.55 +- 0.12 s (sigma = 98362
us)
```

8 потоков

```
(base) C:\Users\Святослав\Desktop\параллельные
вычисления\Myultithread>t

(base) C:\Users\Святослав\Desktop\патраллельные
вычисления\Myultithread>p
ime.py Debug\SingleThread.exe –runsCount 5
Performing 5 runs of “Debug\SingleThread.exe”
 Outer elapased mean time = 5.939 +- 0.031 s (sigma = 24684
us)

(base) C:\Users\Святослав\Desktop\патраллельные
вычисления\Myultithread>p
ime.py Debug\Multithreaded.exe –runsCount 5
Performing 5 runs of “Debug\Multithreaded.exe”
 Outer elapased mean time = 2.621 +- 0.021 s (sigma = 16799
us)

(base) C:\Users\Святослав\Desktop\патраллельные
вычисления\Myultithread>p
ime.py Debug\omp.exe –runsCount 5
Performing 5 runs of “Debug\omp.exe”
 Outer elapased mean time = 2.49 +- 0.04 s (sigma = 29692
us)

(base) C:\Users\Святослав\Desktop\патраллельные
вычисления\Myultithread>
```

7.РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТА

Результаты для однопоточной реализации, реализации с использованием модели thread и openMP представлены ниже.

Потоки	SingleThread	Multithreaded	omp
1,00	5,76		
2,00	5,96	3,40	3,00
3,00	5,87	2,88	2,67
4,00	5,76	2,46	2,29
5,00	5,76	2,37	2,18
6,00	5,98	2,37	2,13
7,00	6,15	2,48	2,55
8,00	5,93	2,62	2,49

График времени исполнения каждой из реализаций в зависимости от потоков представлен на рис. 7.1.

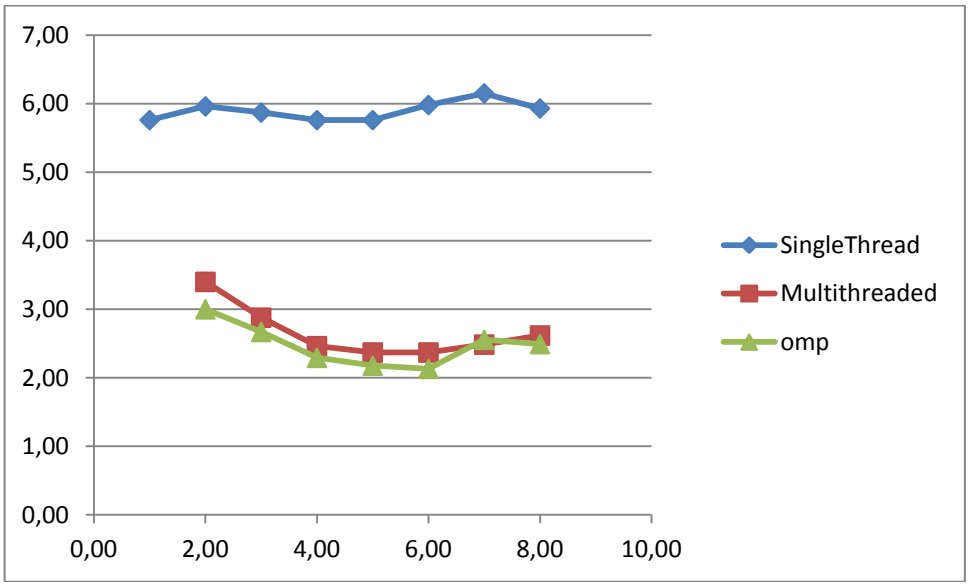

Рис.7.1

ЗАКЛЮЧЕНИЕ

В зоде выполнения работы были рассмотрены технологии параллельной реализация при помощи POSIX threads и Open MP . Рассмотренные способы распараллеливания алгоритма вычисления позволяет ускорить вычисления триграмм примерно в три раза. Влияние распараллеливания возрастает с ростом объёма данных

Тестирование производилось на операционной системе общего назначения, и на результаты вычислений указывала ощутимое влияние работа других процессов. В результате чего OpenMP не смог полностью реализовать преимущества привязке к одному ядру. Кроме того, были отдельно измерены времена работы параллельной части и непараллельной. В результате было выявлено, что причиной снижения производительности многопоточной программы был контейнер map.

Времена запуска Multihreaded на 2-х ядерной машине.

	1 поток	2 потока	1 поток без map	2 потока без map
Время непараллельной части (такты)	4 585 413	4 771 582	5 215 803	4 580 089
Время параллельной части (такты)	26 964 931	21 075 302	7 406 621	5 071 546

При удалении обращений к нему асимптотика работы многопоточных программ выравнивалась.

Данный вопрос требует более глубокого тестирования различных структур данных на разного рода данных.

РЕАЛИЗАЦИЯ Singlethread

==========

```
#include <iostream>
#include <fstream>
#include <vector>
#include <map>
#include <string>
#include <algorithm>
#include <windows.h>
using namespace std;

int main() {
	ifstream t("textfile.txt");
	if (!t.is_open()) {
		cout << "Error\n";
		return 0;
	}
	setlocale(LC_ALL, "Russian");
	SetConsoleOutputCP(866);
	t.seekg(0, std::ios::end);
	size_t size = t.tellg();
	std::string text(size, ' ');
	t.seekg(0);
	t.read(&text[0], size);
	//выкидываем знаки препинания, подправляем регистр
	for (int i = 0; i < text.size(); i++)
	{
		if (text[i] >= 'А' && text[i] <= 'П')
		{
			text[i] = (text[i] - 'А' + 'а');
		}
		else if (text[i] >= 'Р' && text[i] <= 'Я')
		{
			text[i] = (text[i] - 'Р' + 'р');
		}
		else if (text[i] >= 'а' && text[i] <= 'п' || text[i] >= 'р' && text[i] <=
'я')
		{
		}
		else
		{
			text[i] = ' ';
		}
	}
	int help = text.size() / 4;
	int delim[5] = { 0, help,2 * help,3 * help,4 * help - 1 };
	//приступим к подсчёту частоты встреч
	map <string, int> MapOfFreq;
	for (int a = 0; a < 4; a++)
		for (int b = delim[a]; b < delim[a + 1] - 2; b++)
		{
			string str = text.substr(b, 3);
			int prob = 0;
			for (int i = 1; i < 4 && prob == 0; i++)
			{
				if (str[i - 1] == ' ')
					prob = i;
			}
			if (prob == 0)
			{
				map<string, int>::iterator it = MapOfFreq.find(str);
```

```
                        if (it != MapOfFreq.end())
                        {
                                it->second++;
                        }
                        else
                                MapOfFreq.insert(pair<string, int>(str, 1));
                }
                else
                        b += prob;
        }
    //выводим результат
    printf("Топ-20 результатов вероятностей в процентах:\n");
    int summ = 0;
    for (map<string, int>::iterator it = MapOfFreq.begin(); it != MapOfFreq.end();
            ++it)
    {
            if (it->second > 2)
                    summ += it->second;
    }
    //берём топ 20
    std::vector<std::pair<std::string, int>> top_ten(20);
    std::partial_sort_copy(MapOfFreq.begin(),
            MapOfFreq.end(),
            top_ten.begin(),
            top_ten.end(),
            [](std::pair<const std::string, int> const& l,
                    std::pair<const std::string, int> const& r)
    {
            return l.second > r.second;
    });
    for (vector<pair<string, int>>::iterator it = top_ten.begin(); it !=
top_ten.end();
            it++)
            cout << it->first << " - " << (float)it->second / summ * 100 << endl;
}
```

// РЕАЛИЗАЦИЯ Multithreaded

==========

```
#include <iostream>
#include <fstream>
#include <vector>
#include <map>
#include <string>
#include <algorithm>
#include <thread>
#include <windows.h>
using namespace std;
// Число потоков PROCCOUNT=2
#define PROCCOUNT 2

int part_text[PROCCOUNT];
int part_text_size[PROCCOUNT];
std::string text;
map <string, int> MapOfFreqs[PROCCOUNT];
void my_thread_proc(int procnum)
{
	int idx = part_text[procnum];
	int sz = part_text_size[procnum];
	for (int b = idx; b < idx+sz - 2; b++)
	{
		string str = text.substr(b, 3);
		int prob = 0;
		for (int i = 1; i < 4 && prob == 0; i++)
		{
			if (str[i - 1] == ' ')
				prob = i;
		}
		if (prob == 0)
		{
			map<string, int>::iterator it
				= MapOfFreqs[procnum].find(str);
			if (it != MapOfFreqs[procnum].end())
			{
				it->second++;
			}
			else
				MapOfFreqs[procnum].insert(pair<string, int>(str, 1));
		}
		else
			b += prob;
	}
}
int main() {
	ifstream t("textfile.txt");
	if (!t.is_open()) {
		cout << "Error\n";
		return 0;
	}
	setlocale(LC_ALL, "Russian");
	SetConsoleOutputCP(866);
	t.seekg(0, std::ios::end);
	size_t size = t.tellg();
	text.assign(size, ' ');
	t.seekg(0);
	t.read(&text[0], size);
	//выкидываем знаки препинания, подправляем регистр
	for (int i = 0; i < text.size(); i++)
	{
```

```
		if (text[i] >= 'А' && text[i] <= 'П')
		{
			text[i] = (text[i] - 'А' + 'а');
		}
		else if (text[i] >= 'Р' && text[i] <= 'Я')
		{
			text[i] = (text[i] - 'Р' + 'р');
		}
		else if (text[i] >= 'а' && text[i] <= 'п' || text[i] >= 'р' && text[i] <=
'я')
		{
		}
		else
		{
			text[i] = ' ';
		}
	}
	int help = text.size() / PROCCOUNT;

	for (int ii = 0; ii < PROCCOUNT; ii++) {
		part_text[ii] = help * ii;
		part_text_size[ii] = help;
	}


	//приступим к подсчёту частоты встреч
	std::vector<std::thread> threads;
	for (int a = 0; a < PROCCOUNT; a++)
	{
		std::thread thr(my_thread_proc, a);
		threads.emplace_back(std::move(thr));
	}
	for (auto& thr : threads) {
		thr.join();
	}
	//Сводим полученные ответы в один словарь
	map<string, int> MapOfFreq = MapOfFreqs[0];
	for (int i = 1; i < PROCCOUNT; i++)
	{
		for (map<string, int>::iterator it = MapOfFreqs[i].begin();
			it != MapOfFreqs[i].end(); ++it)
		{
			map<string, int>::iterator it2 = MapOfFreq.find(it->first);
			if (it2 != MapOfFreq.end())
			{
				it2->second += it->second;
			}
			else
				MapOfFreqs[i].insert(pair<string, int>(it->first, it-
>second));
		}
	}

	/*
	map <string, int> MapOfFreq;
	for (int a = 0; a < 4; a++)
		for (int b = delim[a]; b < delim[a + 1] - 2; b++)
		{
			string str = text.substr(b, 3);
			int prob = 0;
			for (int i = 1; i < 4 && prob == 0; i++)
			{
				if (str[i - 1] == ' ')
					prob = i;
			}
```

```
                        if (prob == 0)
                        {
                                map<string, int>::iterator it = MapOfFreq.find(str);
                                if (it != MapOfFreq.end())
                                {
                                        it->second++;
                                }
                                else
                                        MapOfFreq.insert(pair<string, int>(str, 1));
                        }
                        else
                                b += prob;
                }*/
        //выводим результат
        printf("Топ-20 результатов вероятностей в процентах:\n");
        int summ = 0;
        for (map<string, int>::iterator it = MapOfFreq.begin(); it != MapOfFreq.end();
                ++it)
        {
                if (it->second > 2)
                        summ += it->second;
        }
        //берём топ 20
        std::vector<std::pair<std::string, int>> top_ten(20);
        std::partial_sort_copy(MapOfFreq.begin(),
                MapOfFreq.end(),
                top_ten.begin(),
                top_ten.end(),
                [](std::pair<const std::string, int> const& l,
                        std::pair<const std::string, int> const& r)
        {
                return l.second > r.second;
        });
        for (vector<pair<string, int>>::iterator it = top_ten.begin(); it !=
top_ten.end();
                it++)
                cout << it->first << " - " << (float)it->second / summ * 100 << endl;
}
```

// РЕАЛИЗАЦИЯ ОМР

==========

```
#include <iostream>
#include <fstream>
#include <vector>
#include <map>
#include <string>
#include <algorithm>
#include <thread>
#include <windows.h>
#include <omp.h>
using namespace std;
// Число потоков PROCCOUNT=2
#define PROCCOUNT 2

int part_text[PROCCOUNT];
int part_text_size[PROCCOUNT];
std::string text;
map <string, int> MapOfFreqs[PROCCOUNT];
void my_thread_proc(int procnum)
{
	int idx = part_text[procnum];
	int sz = part_text_size[procnum];
	for (int b = idx; b < idx + sz - 2; b++)
	{
		string str = text.substr(b, 3);
		int prob = 0;
		for (int i = 1; i < 4 && prob == 0; i++)
		{
			if (str[i - 1] == ' ')
				prob = i;
		}
		if (prob == 0)
		{
			map<string, int>::iterator it
				= MapOfFreqs[procnum].find(str);
			if (it != MapOfFreqs[procnum].end())
			{
				it->second++;
			}
			else
				MapOfFreqs[procnum].insert(pair<string, int>(str, 1));
		}
		else
			b += prob;
	}
}
int main() {
	ifstream t("textfile.txt");
	if (!t.is_open()) {
		cout << "Error\n";
		return 0;
	}
	setlocale(LC_ALL, "Russian");
	SetConsoleOutputCP(866);
	t.seekg(0, std::ios::end);
	size_t size = t.tellg();
	text.assign(size, ' ');
	t.seekg(0);
	t.read(&text[0], size);
	//выкидываем знаки препинания, подправляем регистр
	for (int i = 0; i < text.size(); i++)
	{
```

```
			if (text[i] >= 'А' && text[i] <= 'П')
			{
				text[i] = (text[i] - 'А' + 'а');
			}
			else if (text[i] >= 'Р' && text[i] <= 'Я')
			{
				text[i] = (text[i] - 'Р' + 'р');
			}
			else if (text[i] >= 'а' && text[i] <= 'п' || text[i] >= 'р' && text[i] <=
'я')
			{
			}
			else
			{
				text[i] = ' ';
			}
		}
		int help = text.size() / PROCCOUNT;

		for (int ii = 0; ii < PROCCOUNT; ii++) {
			part_text[ii] = help * ii;
			part_text_size[ii] = help;
		}

		//разделяем вычисления директивой
		#pragma omp parallel for num_threads(PROCCOUNT)
		for (int a = 0; a < PROCCOUNT; a++)
		{
			my_thread_proc(a);
		}
		/*//приступим к подсчёту частоты встреч
		std::vector<std::thread> threads;
		for (int a = 0; a < PROCCOUNT; a++)
		{
			std::thread thr(my_thread_proc, a);
			threads.emplace_back(std::move(thr));
		}
		for (auto& thr : threads) {
			thr.join();
		}*/
		//Сводим полученные ответы в один словарь
		map<string, int> MapOfFreq = MapOfFreqs[0];
		for (int i = 1; i < PROCCOUNT; i++)
		{
			for (map<string, int>::iterator it = MapOfFreqs[i].begin();
				it != MapOfFreqs[i].end(); ++it)
			{
				map<string, int>::iterator it2 = MapOfFreq.find(it->first);
				if (it2 != MapOfFreq.end())
				{
					it2->second += it->second;
				}
				else
					MapOfFreqs[i].insert(pair<string, int>(it->first, it-
>second));
			}
		}

		/*
		map <string, int> MapOfFreq;
		for (int a = 0; a < 4; a++)
			for (int b = delim[a]; b < delim[a + 1] - 2; b++)
			{
				string str = text.substr(b, 3);
				int prob = 0;
```

```
                        for (int i = 1; i < 4 && prob == 0; i++)
                        {
                                if (str[i - 1] == ' ')
                                        prob = i;
                        }
                        if (prob == 0)
                        {
                                map<string, int>::iterator it = MapOfFreq.find(str);
                                if (it != MapOfFreq.end())
                                {
                                        it->second++;
                                }
                                else
                                        MapOfFreq.insert(pair<string, int>(str, 1));
                        }
                        else
                                b += prob;
                }*/
                //выводим результат
        printf("Топ-20 результатов вероятностей в процентах:\n");
        int summ = 0;
        for (map<string, int>::iterator it = MapOfFreq.begin(); it != MapOfFreq.end();
                ++it)
        {
                if (it->second > 2)
                        summ += it->second;
        }
        //берём топ 20
        std::vector<std::pair<std::string, int>> top_ten(20);
        std::partial_sort_copy(MapOfFreq.begin(),
                MapOfFreq.end(),
                top_ten.begin(),
                top_ten.end(),
                [](std::pair<const std::string, int> const& l,
                        std::pair<const std::string, int> const& r)
        {
                return l.second > r.second;
        });
        for (vector<pair<string, int>>::iterator it = top_ten.begin(); it !=
top_ten.end();
                it++)
                cout << it->first << " - " << (float)it->second / summ * 100 << endl;
}
```

// РЕАЛИЗАЦИЯ TIME

```
from os import system, access, remove, W_OK
import numpy as np
import scipy.stats as st
import math
import argparse
import time
def chooseAppropriateTimeUnit(timeInMicros):
	if (timeInMicros / 1000000 > 1):
		return "s"
	elif (timeInMicros/ 1000 > 1):
		return "ms"
	else:
		return "us"
def adjustToTimeUnit(timeInMicros, timeunit):
	if timeunit == "s":
		return timeInMicros / 1000000
	elif timeunit == "ms":
		return timeInMicros / 1000
	else:
		return timeInMicros
def formatProperly(mean, r, timeunit):
	mean = adjustToTimeUnit(mean, timeunit)
	r = adjustToTimeUnit(r, timeunit)
	precision = -round(math.log10(r)) + 1
	if precision < 0:
		roundedMean = int(round(mean, precision))
		roundedRadius = int(round(r, precision))
		return "{:d} +- {:d} {}".format(roundedMean, roundedRadius, timeunit)
	else:
		return "{:.{prec}f} +- {:.{prec}f} {}".format(mean, r, timeunit,
prec=precision)
def findMeanEstimate(vals, p=0.95):
	N = len(vals)
	mean = np.mean(vals)
	tinv = st.t.ppf((1 + p)/2, N - 1)
	r = tinv*st.sem(vals)/math.sqrt(N)
	return (mean, r)
if __name__ == "__main__":
	parser = argparse.ArgumentParser(description="Statistics")
	parser.add_argument("executable")
	parser.add_argument("--runsCount", type=int, required=True)
	parser.add_argument("--cleanup", action="store_true")
	args = vars(parser.parse_args())

	runsCount = args["runsCount"]
	executable = args["executable"]
	print("Performing %d runs of \"%s\"" % (runsCount, executable))
	outerTimesList = list()
	for i in range(runsCount):
		begin = time.perf_counter()
		system("%s > NUL" % (executable))
		end = time.perf_counter()
```

```
       outerTimesList.append((end - begin) * 1000000)
(mean, r)=findMeanEstimate(outerTimesList)
print(" Outer elapased mean time = %s (sigma = %.0f us)" % (\
       formatProperly(mean, r, "s"),\st.sem(oute rTimesList)))
```