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 **ПРОГРАММОЕ ОБЕСПЕЧЕНИЕ ЭВМ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.04 Программная инженерия**

ОТЧЕТ

по лабораторной работе № 4

Название: Параллельное программирование

Дисциплина: Анализ алгоритмов

Студент	ИУ7-52Б		[redacted]
	(Группа)	(Подпись, дата)	(И.О. Фамилия)
Преподаватель 1			Л.Л. Волкова
		(Подпись, дата)	(И.О. Фамилия)
Преподаватель 2			Ю.В. Строганов
		(Подпись, дата)	(И.О. Фамилия)

Москва, 2020

Содержание

Введение

В даной лабораторной работе рассматривается простой алгоритм нахождения количества вхождений каждого символа в наборе строк и его параллельная версия, представленная в двух вариантах.
Задачами данной лабораторной работы являются:

- рассмотрение стандартного алгоритма подсчёта количества вхождений каждого символа в наборе строк;
- проведение сравнительного анализа последовательного алгоритма подсчёта количества вхождений каждого символа в наборе строк и двух его пареллельных реализаций;
- опрделение зависимости времени работы алгоритма от числа потоков исполнения и количества слов.

В ходе данной лабораторной работы будут изучены возможности параллельных вычислений и использование подобного подхода на практике.

1 Аналитическая часть

В данном разделе будет расмотрена теоретическая часть алгоритма подсчёта количества вхождений каждого символа в наборе строк и теоретические основы параллельного программирования.

1.1 Описание алгоритмов

В данном разделе будут описан каждый исследуемый алгоритм.

1.2 Стандартный алгоритм

Пусть дана следующая последовательность строк, длина последовательности - n.
Для того, чтобы подсчитать количество вхождений каждого символа в каждой строке, необходимо создать n дополнительных массивов, каждый из которых имеет размерность m, где:

- n - колчиество строк в исходной последовательности;
- m - мощность некого алфавита, на основе которого строятся строки.

Далее необходимо проитерировать каждую из строк и инкрементировать соответствующую ячейку массива.
Пример:

$$\text{strings} = \{\text{"abc"}\ \ \text{"defd"}\ \ \text{"hijij"}\}$$

Условимся, что все строки составляются из английских строчных букв. Таким образом мощность алфавита - 26.
Дополнительные массивы:

$$\begin{gathered}\text{arr1} = [26];\\ \text{arr2} = [26];\\ \text{arr3} = [26];\end{gathered}$$

Далее итерируем каждую из исходных строк. Перед итерированием необходимо обнулить значение для каждой ячейки массив!

$$\begin{gathered}\text{"abc"}\\ \text{arr1['a']} = \text{arr1['a']} + 1\\ \text{arr1['b']} = \text{arr1['b']} + 1\\ \text{arr1['c']} = \text{arr1['c']} + 1\end{gathered}$$

$$\begin{gathered}\text{"defd"}\\ \text{arr2['d']} = \text{arr2['d']} + 1\\ \text{arr2['e']} = \text{arr2['e']} + 1\\ \text{arr2['f']} = \text{arr2['f']} + 1\\ \text{arr2['d']} = \text{arr2['d']} + 1\end{gathered}$$

$$\begin{gathered}\text{"hijij"}\\ \text{arr3['h']} = \text{arr3['h']} + 1\\ \text{arr3['i']} = \text{arr3['i']} + 1\\ \text{arr3['j']} = \text{arr3['j']} + 1\\ \text{arr3['i']} = \text{arr3['i']} + 1\\ \text{arr3['j']} = \text{arr3['j']} + 1\end{gathered}$$

Таким образом на выходе получаем следующие значения:

$$\begin{gathered}\text{arr1['a']} = 1 \\ \text{arr1['b']} = 1 \\ \text{arr1['c']} = 1 \\ \text{arr2['d']} = 2 \\ \text{arr2['e']} = 1 \\ \text{arr2['f']} = 1 \\ \text{arr3['h']} = 1 \\ \text{arr3['i']} = 2 \\ \text{arr3['j']} = 2\end{gathered}$$

Все остальные ячейки равны нулю.
Таким образом в каждом из соответствуюших массивов хранится колчиство вхождений каждого из символов алфавита в соответствующей строке.

1.3 Первая реализация паралелльного алгоритма

Реализуем некую очередь, в которой каждый из потоков будет брать следующую строку. Таким образом, как только поток освобождается - он берёт следующую строку из очереди.
Пример: Пусть есть 2 потока и следующий список строк:

$$\text{strings} = \{\text{"abc"} \ \ \text{"def"} \ \ \text{"hij"}\}$$

Первый поток берёт на обработку строку "abc"(обработка аналогична описанной в пункте выше). Второй поток в это время берёт на обработку строку "def". Далее первый поток берёт на обработку строку "hij". Выполнение алгоритма завершено. В данном алгоритме очень важно следить за корректным взятием элементов из очереди.

1.4 Вторая реализация паралелльного алгоритма

Так как количество строк заранее известно, то каждому из потоков можно выдать одинаковое количество строк на обработку, поделив количство строк на число потоков.
Пример: Пусть есть 3 потока и следующий список строк:

$$\text{strings} = \{\text{"abc"} \ \ \text{"def"} \ \ \text{"hij"} \ \ \text{"klm"} \ \ \text{"nop"} \ \ \text{"qrs"}\}$$

Колчиство строк - 6. Количство потоков - 3. Слеовательно, количество строк на обработку для каждого потока - 2.
Для упрошения каждому из потоков можно давать на обработку те строки, номера которых лежат между номром потока, умноженного на количество строк, поделённое на количество потоков и номром следующего потока, умноженного на количество строк, поделённое на количество потоков:

$$startIndex = \frac{threadID * stringsAmount}{threadsAmount} \tag{1.1}$$

$$endIndex = \frac{(threadID + 1) * stringsAmount}{threadsAmount} \tag{1.2}$$

1.5 Параллельное программирование

При использовании многопроцессорных вычислительных систем с общей памятью обычно предполагается, что имеющиеся в составе системы процессоры обладают равной производительностью, являются равноправными при доступе к общей памяти, и время доступа к памяти является одинаковым (при одновременном доступе нескольких процессоров к одному и тому же элементу памяти очередность и синхронизация доступа обеспечивается на аппаратном уровне). Многопроцессорные системы подобного типа обычно именуются симметричными мультипроцессорами (symmetric multiprocessors, SMP) [3].

Обычный подход при организации вычислений для многопроцессорных вычислительных систем с общей памятью – создание новых параллельных методов на основе обычных последовательных программ, в которых или автоматически компилятором, или непосредственно программистом выделяются участки независимых друг от друга вычислений. Возможности автоматического анализа программ для порождения параллельных вычислений достаточно ограничены, и второй подход является преобладающим.

Широко используемый подход состоит и в применении тех или иных библиотек, обеспечивающих определенный программный интерфейс (application programming interface, API) для разработки параллельных программ. В рамках такого подхода наиболее известны Windows Thread API [1]. Однако первый способ применим только для ОС семейства Microsoft Windows, а второй вариант API является достаточно трудоемким для использования и имеет низкоуровневый характер

1.6 Организация взаимодействия параллельных потоков

Потоки, в отличии от процессов, не имеют собственоого адресного пространства. Как результат, взаимодействи потоков можно реализовать через использование общих данных. В случае, когда общие данны необходимо только читать - проблем возникнуть не может. В случае, если необходимо общие данные изменять - необходимо пользоваться средствами синхронизации: mutex, семафор.

Вывод

В данном разделе была рассмотрена теоретичсекая часть алгоритма поиска вхождения символа в каждой из строк из набора строк, а также описаны две его параллельные версии. Была рассмотрена технология параллельного программирования и организация взаимодействия параллельных потоков.

2 Конструкторская часть

Требования к вводу: на вход подаётся количество строк и сами строки.
Требования к программе:: Подсчёт количество вхождений каждого символа в каждой строке.

2.1 Разработка алгоритмов

В данном разделе представлены схемы реализуемых алгоритмов.
На рисунке 2.1 представлена схема последовательного алгоритма подсчёта.

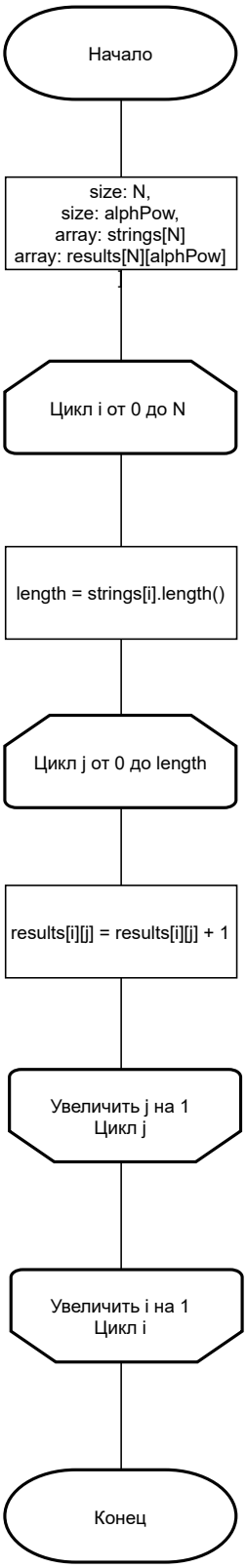

Рис. 2.1: Схема последовательного алгоритма подсчёта

На рисунке 2.2.1 представлена схема главного потока первого парллельного алгоритма.

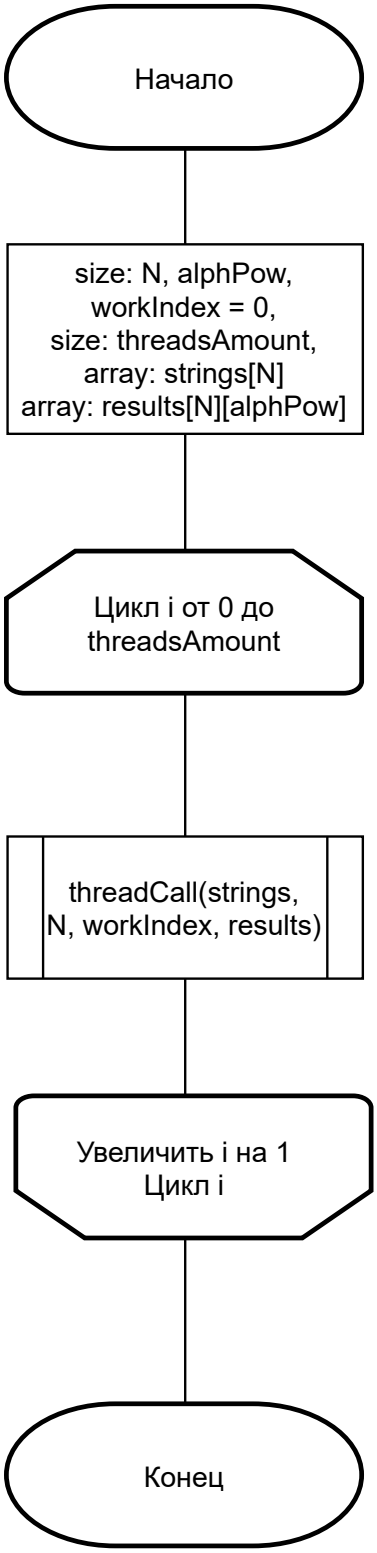

Рис. 2.2.1: Схема главного потока первого парллельного алгоритма

На рисунке 2.2.2 представлена схема дочернего потока первого парллельного алгоритма.

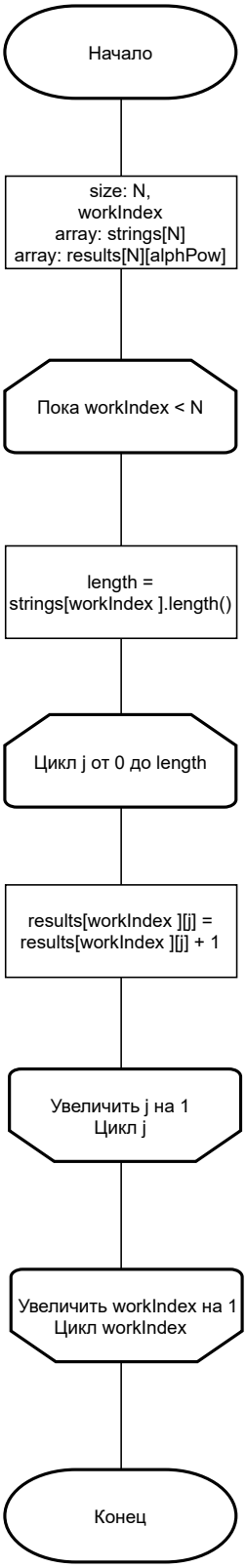

Рис. 2.2.2: Схема дочернего потока первого парллельного алгоритма

На рисунке 2.3.1 представлена схема главного потока второго парллельного алгоритма.

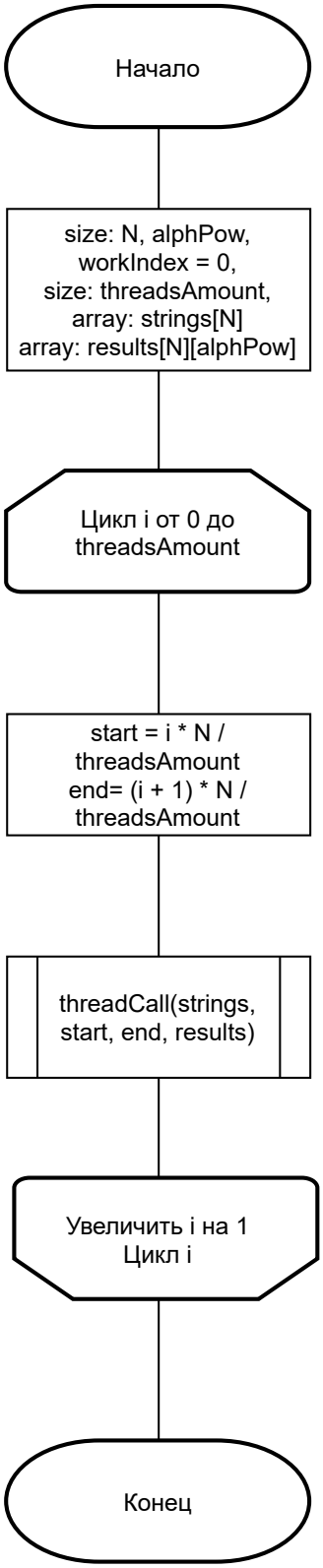

Рис. 2.3.1: Схема главного потока второго парллельного алгоритма

На рисунке 2.3.2 представлена схема дочернего потока второго парллельного алгоритма.

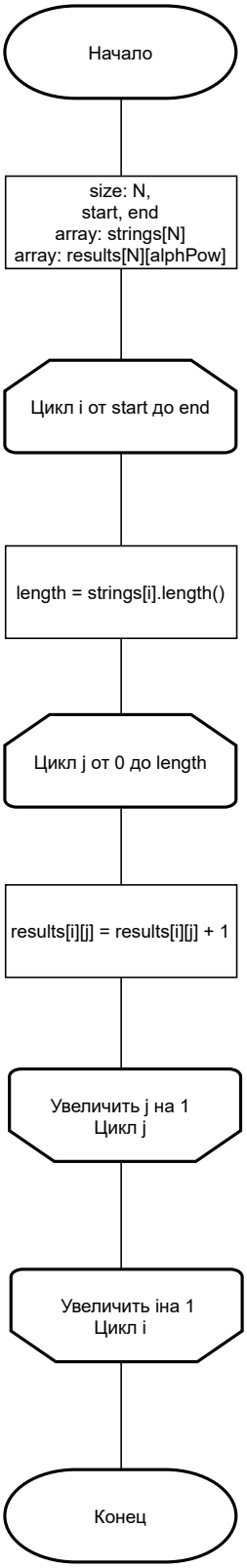

Рис. 2.3.2: Схема дочернего потока второго парллельного алгоритма

Вывод

В данном разделе были рассмотрены схемы реализуемых алгоритмов.

3 Технологическая часть

В данном разделе будет описана технологическая часть лабораторной работы: требования к ПО, листинг кода, сравнительный анализ всех алгоритмов.

3.1 Требования к программному обеспечению

Входные данные: размерность массива, сам массив
Выходные данные: отсортированный массив
Среда выполнения: Windows 10 x64 CPU: Core i7-8550U

3.2 Средства реализации

Для выполнения данной лабораторной работы использовался язык программирования C++ стандарта 2017 года, а также среда разработки CLion 2020.2. Для замены времени было использовано средство **std::chrono** [4]

3.3 Листинг кода

В данном разделе будет представлен листинг кода разработанных алгоритмов.

Ниже, на листинге 3.1, представлена реализация последовательного алгоритма подсчёта:

Листинг 3.1: Последовательный алгоритм подсчёта

```
1  std::vector<std::vector<int>> countWordsInStringConsistent(
2      const std::vector<std::string>& strings) {
3      std::vector<std::vector<int>> arr;
4      for (int i = 0; i < strings.size(); i++) {
5          std::vector<int> subArr(DICT_SIZE);
6          arr.push_back(subArr);
7      }
8
9      for (int i = 0; i < strings.size(); i++) {
10         for (const auto& word: strings[i]) {
11             arr[i][word]++;
12         }
13     }
14
15     return arr;
16 }
```

Ниже, на листинге 3.2.1, представлена реализация главного потока первого параллельного алгоритма подсчёта:

Листинг 3.2.1: Главный поток первого алгоритма подсчёта

```
std::vector<std::vector<int>> countWordsInStringParallel(
    const std::vector<std::string>& strings,
    int threadsAmount) {
    std::vector<std::vector<int>> arr;
    for (int i = 0; i < strings.size(); i++) {
    std::vector<int> subArr(DICT_SIZE);
    arr.push_back(subArr);
    }

    std::vector<std::thread> threadVector;
    int* wordPosition = new int;
    *wordPosition = 0;
    for (int i = 0; i < threadsAmount; i++) {
        threadVector.emplace_back(std::thread(
                                    parallelBadThreadFunction,
                                    std::ref(strings),
                                    wordPosition,
                                    std::ref(arr)));
    }

    for (int i = 0; i < threadsAmount; i++) {
        threadVector[i].join();
    }
    delete wordPosition;


    return arr;
}
```

Ниже, на листинге 3.2.2, представлена реализация дочернего потока первого параллельного алгоритма подсчёта:

Листинг 3.2.1: Дочерний поток первого алгоритма подсчёта

```
static void parallelThreadFunction(
            const std::vector<std::string>& strings,
            int *wordIndex,
            std::vector<std::vector<int>>& arr) {
    const int size = strings.size();
    while (true) {
        thread_lock.lock();
        if (*wordIndex >= size) {
            thread_lock.unlock();
            break;
        }
        int currPosition = (*wordIndex);
        *wordIndex += 1;
```

```
14            thread_lock.unlock();
15            for (const auto& chr : strings[currPosition]) {
16                ++arr[currPosition][chr];
17            }
18        }
19    }
```

Ниже, на листинге 3.3.1, представлена реализация главного потока второго параллельного алгоритма подсчёта:

Листинг 3.3.1: Главный поток второго алгоритма подсчёта

```
std::vector<std::vector<int>> countWordsInStringParallel(
    const std::vector<std::string>& strings,
    int threadsAmount) {
    std::vector<std::vector<int>> arr;
    for (int i = 0; i < strings.size(); i++) {
        std::vector<int> subArr(DICT_SIZE);
        arr.push_back(subArr);
    }

    std::vector<std::thread> threadVector;
    for (int i = 0; i < threadsAmount; i++) {
        int start = ((double)strings.size() / threadsAmount * i);
        int end = ((double)strings.size() / threadsAmount * (i + 1));
        threadVector.emplace_back(
            std::thread(parallelGoodThreadFunction,
            std::ref(strings), start, end, std::ref(arr)));
    }

    for (int i = 0; i < threadsAmount; i++) {
        threadVector[i].join();
    }

    return arr;
}
```

Ниже, на листинге 3.3.2, представлена реализация дочернего потока второго параллельного алгоритма подсчёта:

Листинг 3.3.2: Дочерний поток второго алгоритма подсчёта

```
static void parallel1ThreadFunction(
                const std::vector<std::string>& strings,
            int start, int end,
            std::vector<std::vector<int>>& arr) {
    for (int i = start; i < end; i++) {
        for (const auto& chr : strings[i]) {
            arr[i][chr]++;
        }
    }
}
```

3.4 Описание тестирования

Были проведены тесты на больших размерностях со случайными строками в качестве элементов. Ниже, на листинге 3.4, представлен фрагмент кода тестирования корректной работы реализации алгоритмов

Листинг 3.4: Тестирование корректной работы алгоритмов

```
int startTests () {
    for (int stringsAmount = 100;
        stringsAmount < 1e8;
        stringsAmount *= 10) {
        std::cout << "Strings amount is " << stringsAmount;
        std::cout << std::endl;
        std::vector<std::vector<int>> dict1;
        std::vector<std::vector<int>> dict2;
        std::vector<std::vector<int>> dict3;

        std::vector<std::string> strings;
        for (int i = 0; i < stringsAmount; i++) {
            std::string string;
            for (int j = 0; j < rand() % 500; j++) {
                string.push_back((rand() % 26) + 'a');
            }
            strings.push_back(string);
        }

        dict1 = countWordsInStringConsistent(strings);
        dict2 = countWordsInStringParallelQueue(strings, 8);
        dict3 = countWordsInStringParallelGood(strings, 8);

        if (dict1 != dict2 ||
            dict1 != dict3 ||
            dict2 != dict3) {
            return EXIT_FAILURE;
        }
    }

    return EXIT_SUCCESS;
}
```

Все тесты пройдены успешно.

Вывод

В данном разделе был представлен листинг реализованных алгоритмов, а также описание тестирование корректности их работы.

4 Экспериментальная часть

В данной части работы будут приведен анализ алгоритмов на основе эксперементальных данных.

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведён замер времени работы каждого из параллельных алгоритмов. Длина каждой строки 10000 символов. Каждый эксперимент на каждой размерности массива строк был произведён 5 раз, затем бралось среднее арифметическое полученного результата.
Все эксперименты были проведены для 400000 строк.
Ниже во всех таблицах время выполнения указано в секундах.

Ниже, в таблице 4.1 показаны результаты замеров выполнения программы для первого пареллельного алгоритма
Ниже, на графиках 4.1, показана зависимость времени выполнения программы от количества потоков для первой параллельной версии алгоритма:

Таблица 4.1: Замеры времени работы первого параллельного алгоритма

Количество потоков	Время выполнения(с)
1	95.8329
2	28.7719
4	20.4898
8	16.1282
16	15.973
32	16.0602

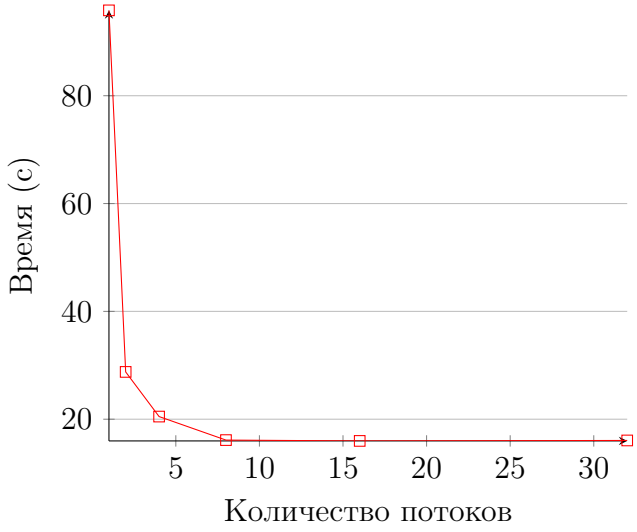

График 4.1: Замеры вермени работы первого параллельного алгоритма

Ниже, в таблице 4.2 показаны результаты замеров выполнения программы для второго пареллельного алгоритма
Ниже, на графиках 4.2, показана зависимость времени выполнения программы от количества строк и количества потоков для первой параллельной версии алгоритма:

Таблица 4.2: Замеры времени работы второго параллельного алгоритма

Количество потоков	Время выполнения(с)
1	48.6109
2	28.7293
4	18.8235
8	16.33
16	15.3795
32	16.4104

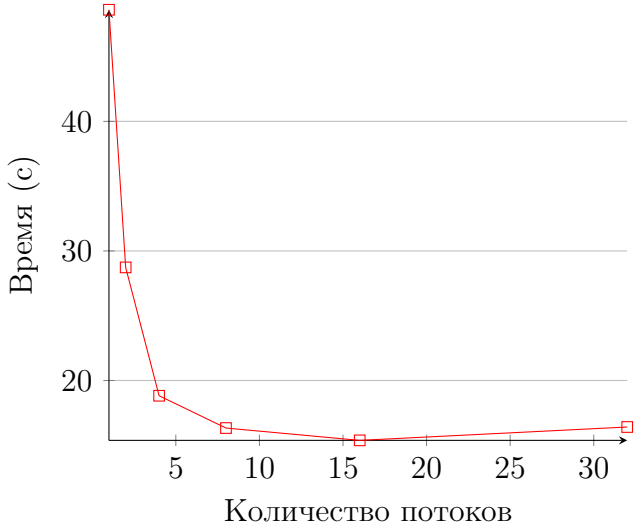

График 4.2: Замеры вермени работы второго параллельного алгоритма

В результате экспериментов было выяснено, что максимальной производительности в обоих реализация удалость достичь при использовании 16 потоков благодаря технология Intel® Hyper-Threading.

Для сравнения, время работы последовательного алгоритма на тех же значениях = 48.6889 секунд, что примерно в 3 раза медленнее, чем самая быстрая многопоточная релизация.

Вывод

По результатам исследования получилось, что обе параллельные версии алгоритма работают приблизительно за равное время, но каждая из них быстрее классического алгоритма. Также установлено, что увеличение количества потоков имеет смысл, пока не будет достигнуто число, равное количеству логических потоков в системе. Но если процессор поддерживает технологию Intel® Hyper-Threading, то программа наилучшим образом будет работать при количестве потоков, равному удвоенному количеству потоков в процессоре.

Заключение

В ходе работы были изучен алгоритм нахождения количества вхождений каждого символа в наборе строк, а также разработаны 3 версии этого алгоритма: 1 последовательная и 2 параллельных. Экспериментально было установлено, что параллельные версии быстрее классического алгоритма. Было установлено, что максимальная скорость работы параллельных алгоритмов достигается при равенстве количества потоков числу потоков в процессоре, умноженному на 2.

Список использованной литературы

[1] Справка по потокам в ОС Windows // https://docs.microsoft.com/en-us/windows/win32/procthread/process-and-thread-functions (дата обращения: 27.10.2020).

[2] Кнут Д. Э. Искусство программирования. Том 1. Сортировка и поиск . – М.: Вильямс, 2007. – 832 с.

[3] Богачев К.Ю Основы параллельного программирования. – М.: БИНОМ. Лаборатория знаний 2003. – 237 с.

[4] Справка по C++ // cppreference URL: https://en.cppreference.com/w/ (дата обращения: 27.10.2020).