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»

Лабораторная работа № 5

Дисциплина Анализ алгоритмов

Тема Конвейерные алгоритмы

Студент Искакова К.М.

Группа ИУ7-52Б

Оценка (баллы) _____

Преподаватель Волкова Л. Л.

Москва — 2021 г.

Содержание

Введение

Термин «конвейер» пришёл из промышленности, где используется подобный принцип работы — материал автоматически подтягивается по ленте конвейера к рабочему, который осуществляет с ним необходимые действия, следующий за ним рабочий выполняет свои функции над получившейся заготовкой, следующий делает ещё что-то. Таким образом, к концу конвейера цепочка рабочих полностью выполняет все поставленные задачи, сохраняя высокий темп производства. Например, если на самую медленную операцию затрачивается одна минута, то каждая деталь будет сходить с конвейера через одну минуту. В процессорах роль рабочих исполняют функциональные модули, входящие в состав процессора.

В данной лабораторной работе рассматривается конвейерный алгоритм нахождения количества вхождений каждого символа в наборе строк, представленный в двух вариантах: с использованием многопоточности и без.

Задачи работы:

1) рассмотрение стандартного и многопоточного алгоритма нахождения количества вхождений каждого символа в наборе строк;

2) проведение сравнительного анализа двух реализаций алгоритма;

3) описание и обосноване полученных результатов.

В ходе лабораторной работы будут изучены возможности конвейерных 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 конвейерного алгоритма без многопоточности

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 – количество 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:

strings{"abcd "defd "hiji "kl"}

Условимся, что все строки составляются из английских строчных букв, таким образом мощностб алфавита = 26.

Дополнительные массивы:

arr1 = [26];
arr2 = [26];
arr3 = [26];
arr4 = [26];

Допустим, что количество лент = 2. Каждая из лент должна обработать свою часть.

Так как ленты 2, то каждьая должна обработать по половине каждой строки.

Лента 1:

"abcd"
arr1['a'] = arr1['a'] + 1
arr1['b'] = arr1['b'] + 1

"defd"
arr2['a'] = arr2['d'] + 1
arr2['e'] = arr2['e'] + 1

"hiji"

arr3['h'] = arr3['h'] + 1
arr3['i'] = arr3['i'] + 1

"kl"
arr4['k'] = arr4['k'] + 1

Лента 2:

"abcd"
arr1['c'] = arr1['c'] + 1
arr1['d'] = arr1['d'] + 1

"defd"
arr2['f'] = arr2['f'] + 1
arr2['d'] = arr2['d'] + 1

"hiji"
arr3['j'] = arr3['j'] + 1
arr3['i'] = arr3['i'] + 1

"kl"
arr4['l'] = arr4['l'] + 1

Таким образом на выходе получаем следующие значения:

arr1['a'] = 1
arr1['b'] = 1
arr1['c'] = 1
arr1['d'] = 1
arr2['d'] = 2
arr2['e'] = 1
arr2['f'] = 1
arr3['h'] = 1
arr3['i'] = 1
arr3['j'] = 2
arr4['k'] = 1
arr4['l'] = 1

Все остальные ячейки равны нулю.

Таким образом, в каждом из соответствующих массивов хранится количество вхождений каждого из символов алфавита в соответствующей строке.

1.2. Описание конвейерного алгоритма с использованием многопоточности

В целом алгоритм, в основе которого лежит использование множества потоков является схожим с последовательным алгоритмом с той лишь разницей, что потоки делят между собой адресное пространство. Таким образом, в отличие от последовательного алгоритма можно не дожидаться пока каждый из конвейеров обработает **все** строки, а передавать **уже обработанные** строки на обработку следующему конвейеру.

1.3. 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) [1].

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 [2]. Однако первый способ применим только для ОС семейства Microsoft Windows, а второй вариант API является достаточно трудоемким для использования и имеет низкоуровневый характер.

1.4. Организация взаимодействия параллельных потоков

Как уже было сказано, потоки, в отличии от процессов, не имеют собственного адресного пространства. Как результат, взаимодействи потоков можно реализовать через использование общих данных. В случае, когда общие данные необходимо только читать — проблем возникнуть не может. В случае, если необходимо общие данные изменять — необходимо пользоваться средствами синхронизации: mutex, семафор.

1.5. Вывод

В данном разделе была рассмотрена теоритическая часть конвейерного алгоритма поиска вхождения символа в каждой из строк из набора строк, а такие описаны две его реализации: с использованием многопоточности и без. Была рассмотрена технология параллельного программирования и организация взаимодействия параллелных потоков.

Также предоставим к ПО следующие требования:

Требования к вводу: на вход подаётся количество строк и сами строки.

Требования к программе: Подсчёт количества вхождений каждого символа в каждой строке.

2 | Конструкторская часть

В этом разделе содержатся схемы реализуемых алгоритмов.
На вход алгоритм принимает количество строк и сами строки.

2.1. Схемы алгоритмов

На Рис. 2.1–2.2 представлена схема последовательного конвейерного алгоритма подсчета.

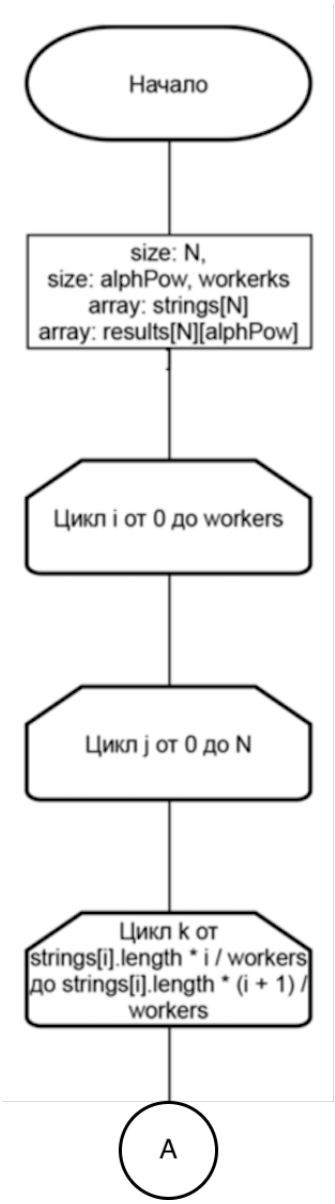

Рис. 2.1. Схема последовательного конвейерного алгоритма подсчета. Часть 1.

Рис. 2.2. Схема последовательного конвейерного алгоритма подсчета. Часть 2.

На Рис. 2.3 представлена схема главного потока параллельного конвейерного алгоритма.

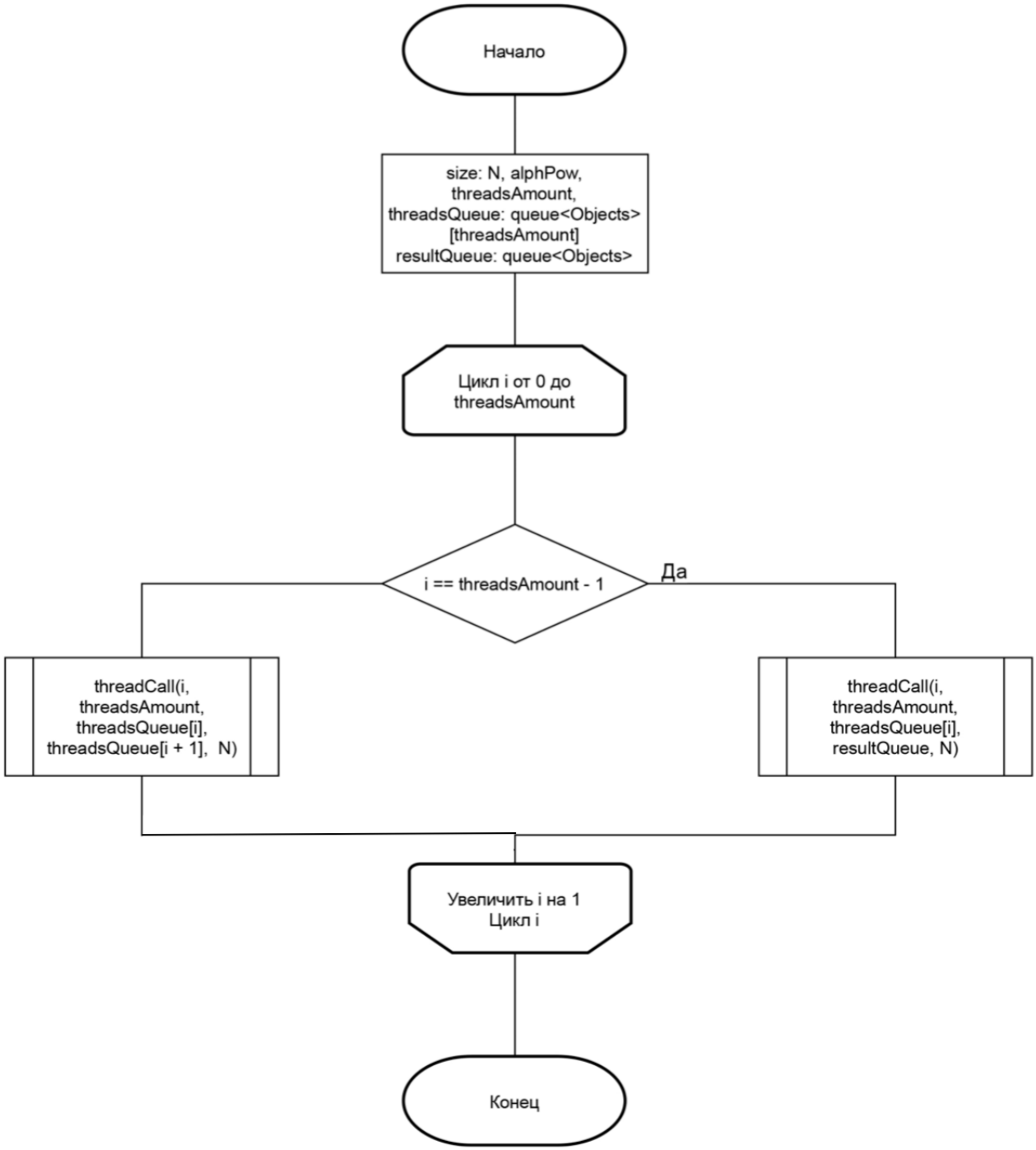

Рис. 2.3. Схема главного потока параллельного конвейерного алгоритма.

На Рис. 2.4-2.5 представлена схема дочернего потока параллельного конвейерного алгоритма.

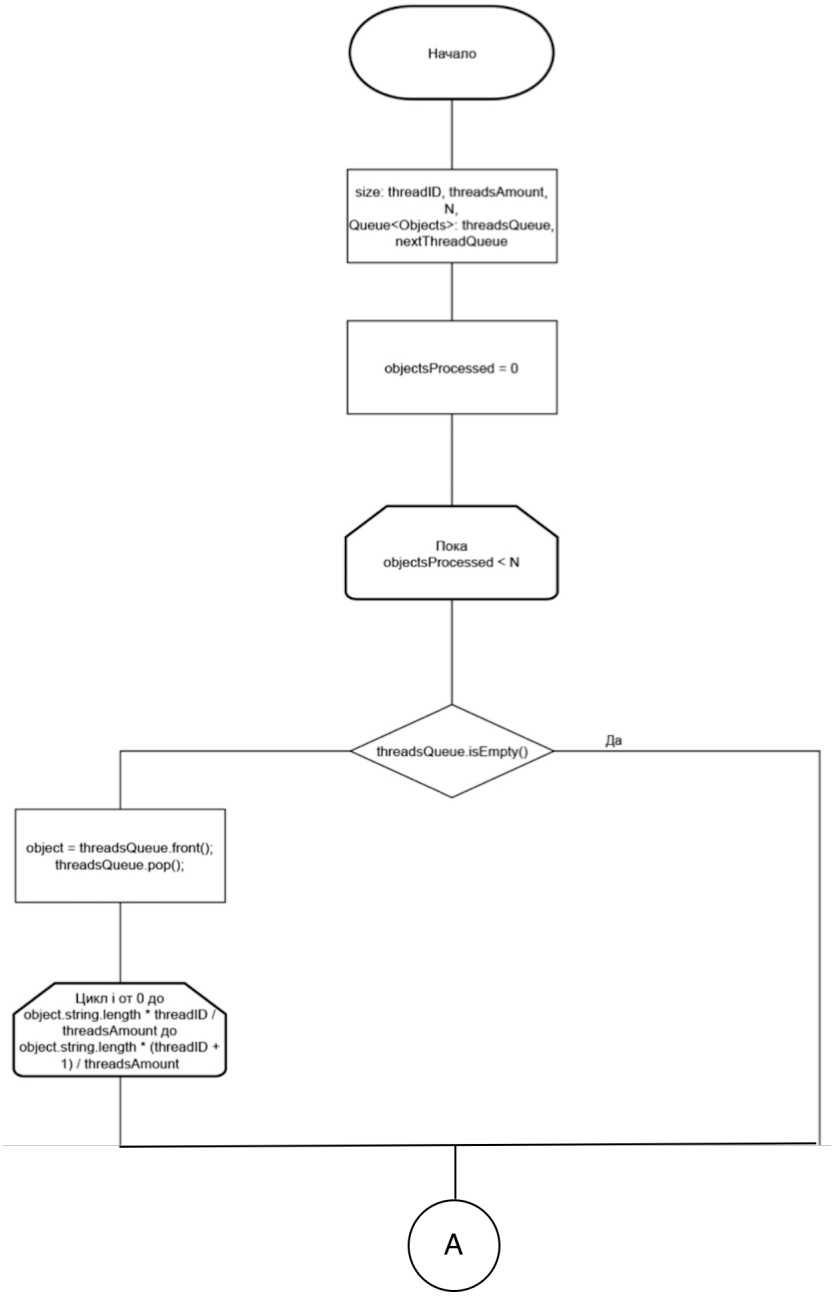

Рис. 2.4. Схема дочернего потока параллельного конвейерного алгоритма. Часть 1.

Рис. 2.5. Схема дочернего потока параллельного конвейерного алгоритма. Часть 2.

3 | Технологическая часть

В данном разделе будут рассмотрены требования к программному обеспечению, средства реализации, представлен листинг кода и описание тестирования.

3.1. Средства реализации

В качестве языка программирования был выбран C++ т.к. я знакома с данным языком, у него есть уникальный баланс между возможностями объектно-ориентированного программирования и производительностью. Он одновременно позволяет писать высокоуровневый абстрактный код, который при этом работает со скоростью близкой к машинному коду.

Среда разработки — CLion, которая предоставляет умную проверку кода, быстрое выявление ошибок и оперативное исправление, вкупе с автоматическим рефакторингом кода, и богатыми возможностями в навигации.

Время работы алгоритмов было замерено с помощью функции steady_clock() из библиотеки chrono [3].

Для тестирования использовался компьютер на базе процессора 2,2 GHz Intel Core i7 [5], 6 ядер, 4 логических процессора

3.2. Листинг кода

В Листинге 3.1 показана реализация последовательного алгоритма подсчета.

Листинг 3.1. Последовательный алгоритм подсчета.

```
1  static void countLettersInObject (WorkObject& object, size_t startLetter,
       size_t endLetter)
2  {
3      for (size_t i = startLetter; i < endLetter; i++)
4      {
5          object.result[object.string[i] - 'a']++;
6      }
7  }
8
9  void linearWorker(std::queue<WorkObject>& objects,
10                 const size_t workerNumber,
11                 const size_t workersAmount,
12                 size_t elementsToProcess)
13 {
14     for (size_t i = 0; i < elementsToProcess; i++)
15     {
16         WorkObject object = objects.front();
17         objects.pop();
18         countLettersInObject(object,
19                              object.string.length() * (double)workerNumber /
                                   workersAmount,
20                              object.string.length() * (double)(workerNumber +
                                    1) / workersAmount);
21         objects.push(object);
22     }
23 }
24
25 std::vector<WorkObject> initLinearWork(std::vector<std::string>& strings,
26                const int workersAmount)
27 {
28     std::queue<WorkObject> objects;
29     for (auto& string : strings)
30     {
31       objects.emplace(WorkObject(string));
32     }
33     MyTimer Timer;
34     for (int i = 0; i < workersAmount; i++)
35     {
36       linearWorker(objects, i, workersAmount, strings.size());
37     }
38     std::cout << "Linear alg works for: " <<
39                  Timer.elapsed() <<
```

```
40                  " seconds" << std::endl;
41      std::vector<WorkObject> result;
42      while (!objects.empty())
43      {
44        result.push_back(objects.front());
45        objects.pop();
46      }
47      return result;
48  }
```

В Листинге 3.2 показана реализация главного потока параллельного конвейерного алгоритма подсчета.

Листинг 3.2. Главный поток параллельного конвейерного алгоритма подсчета.

```
1  std::vector<WorkObject> initConveyorWork(std::vector<std::string>& strings,
       const int workersAmount) {
2    std::queue<WorkObject> completedObjects;
3    std::vector<std::queue<WorkObject>> workersQueues(workersAmount);
4    std::vector<std::thread> threads;
5    for (auto& string : strings) {
6      workersQueues[0].push(WorkObject(string));
7    }
8
9    std::vector<std::mutex> mutexes(workersAmount + 1);
10
11   MyTimer timerAllWork;
12   for (size_t i = 0; i < workersAmount; i++) {
13     threads.emplace_back(std::thread(threadWork, i, workersAmount, std::ref(
         workersQueues[i]),
14     i == workersAmount - 1 ? std::ref(completedObjects) : std::ref(
15     workersQueues[i + 1]), strings.size(), std::ref(mutexes[i]),
16     std::ref(mutexes[i + 1])));
17   }
18
19   for (size_t i = 0; i < workersAmount; i++) {
20     threads[i].join();
21   }
22
23   std::cout << "All Conveyor worked for: " << timerAllWork.elapsed() << "
       seconds" << std::endl;
24   std::vector<WorkObject> result;
25   while (!completedObjects.empty()) {
26     result.push_back(completedObjects.front());
27     completedObjects.pop();
28   }
29   return result;
```

```
}
```

В Листинге 3.3 представлена реализация дочернего потока параллельного конвейерного алгоритма подсчета. .

Листинг 3.3. Дочерний поток параллельного конвейерного алгоритма подсчета.

```
static void countLettersInObject (WorkObject& object, size_t startLetter,
    size_t endLetter) {
  for (size_t i = startLetter; i < endLetter; i++) {
    object.result[object.string[i] - 'a']++;
  }
}

std::mutex allThreadsLock;

void threadWork(const int threadNumber, const int threadsAmount, std::queue<
    WorkObject>& threadsQueue, std::queue<WorkObject>& nextThreadQueue,
    size_t objectsToProcess,
std::mutex& threadsMutex, std::mutex& nextThreadsMutex) {
  MyTimer timerThreadWork;
  size_t sleepTime;
  size_t processed = 0;
  while(processed != objectsToProcess) {
    if (threadsQueue.size()) {
      WorkObject object = threadsQueue.front();

      threadsMutex.lock();
      threadsQueue.pop();
      threadsMutex.unlock();

      countLettersInObject(object, object.string.length() * (double)
          threadNumber / threadsAmount, object.string.length() * (double)(
          threadNumber + 1) / threadsAmount);

      nextThreadsMutex.lock();
      nextThreadQueue.push(object);
      nextThreadsMutex.unlock();

      processed++;
    } else {
      usleep(1000);
      sleepTime += 1;
    }
  }

  allThreadsLock.lock();
  std::cout << "Thread # " << threadNumber << " worked for " <<
      timerThreadWork.elapsed() << " seconds" << std::endl;
```

```
37    std::cout << "Thread # " << threadNumber << " sleeped for " << sleepTime
          << " milliseconds" << std::endl;
38    allThreadsLock.unlock();
39  }
```

3.3. Тестирование

Были проведены тесты на больших размерностях со случайными строками в качестве элементов.

На листинге 3.4 представлен фрагмент кода тестирования корректной работы реализации алгоритмов.

Листинг 3.4. Тестирование корректной работы алгоритмов.

```
1  int tests(size_t wordsAmount, size_t lettersInWord)
2  {
3    std::vector<std::string> strings;
4    for (size_t i = 0; i < wordsAmount; i++)
5    {
6      std::string string;
7      for (size_t j = 0; j < lettersInWord; j++)
8      {
9        string.push_back(rand() % LETTERS_IN_ENG_ALPHABET + 'a');
10     }
11     strings.emplace_back(string);
12   }
13
14   std::vector<WorkObject>  resultConsistent = initLinearWork(strings, 3);
15
16   std::vector<WorkObject>  resultConveyor = initConveyorWork(strings, 3);
17
18   if (resultConsistent != resultConveyor)
19   {
20     return EXIT_FAILURE;
21   }
22
23   return EXIT_SUCCESS;
24 }
```

Все тесты пройдены успешно.

4 Экспериментальная часть

В данном разделе 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.

Для всех замеров количество лент = 3. Во всех таблицах время выполнения указано в секундах.

В таблице 4.1 показаны результаты замеров выполнения программы для последовательной и параллельной реализаций соответственно.

Таблица 4.1: Замеры времени работы последовательного конвейерного алгоритма

Количество строк	Время послед. алгоритма (с)	Время параллел. алгоритма (с)
10000	0.8535188	0.744018
20000	1.73213	1.506136
30000	2.675778	1.761744
40000	3.416202	2.11773
50000	4.251266	2.624158

Ниже, на рисунке 4.1, показана графическая интерпретация таблицы 4.1

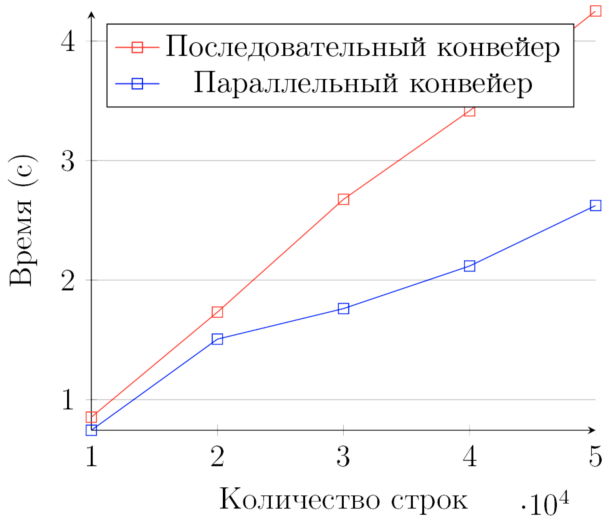

Рисунок 4.1: Замеры вермени работы первого параллельного алгоритма

Вывод

По результатам исследования получилось, что при малых размерностях входных данных разница по времени между алгоритмами не существенна, но с увеличением размерности эффективнее использовать параллельный алгоритм подсчета.

Заключение

В ходе лабораторной работы был изучен алгоритм нахождения количества вхождений каждого символа в наборе строк, а также разработаны 2 конвейерные версии этого алгоритма: последовательный и параллельный. Эксперементально было установлено, что параллельные версии быстрее последовательного алгоритма.

Список литературы

1. Богачев К.Ю. Основы параллельного программирования. – М.: БИНОМ. Лаборатория знаний 2003. - 237 с.

2. Справка по потокам в ОС Windows [электронный ресурс]. Режим доступа: https://docs.microsoft.com/en-us/windows/win32/procthread/process-and-thread-functions, свободный (дата обращения: 09.12.2021)

3. Официальный сайт Microsoft, документация [электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/cpp/standard-library/chrono?view=vs-2017, свободный (дата обращения: 09.12.21)

4. Кнут Д. Э. Искусство программирования. Том 1. Сортировка и поиск. – М.: Вильямс, 2007. - 832 с.

5. Официальный сайт Intel [электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/134906/intel-core-i78750h-processor-9m-cache-up-to-4-10-ghz.html, свободный (дата обращения: 09.12.21)