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 № 5

Название: Конвейерные алгоритмы

Дисциплина: Анализ алгоритмов

Студент ИУ7-51Б О.А. Тюрин
(Группа) (Подпись, дата) (И.О. Фамилия)

Преподаватель 1 Л.Л. Волкова
(Подпись, дата) (И.О. Фамилия)

Москва, 2020

Содержание

Введение

Выполнение каждой команды складывается из ряда последовательных шагов суть которых не меняется от команды к команде. С целью увеличения быстродействия процессора и максимального использования всех его возможностей в современных микропроцессорах используется конвейерный принцип обработки информации. Этот принцип подразумевает, что в каждый момент времени процессор работает над различными стадиями выполнения нескольких команд, причем на выполнение каждой стадии выделяются отдельные аппаратные ресурсы. По очередному тактовому импульсу каждая команда в конвейере продвигается на следующую стадию обработки, выполненная команда покидает конвейер, а новая поступает в него.

Целью лабораторной работы является создание системы конвейерной обработки.
Задачами данной лабораторной работы являются:

- спроектировать ПО, реализующего конвейерную обработку;
- описать реализацию ПО;
- провести тестирование ПО.

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

1 Аналитическая часть

В данном разделе будут рассмотрены общие сведения о конвейерной обработке и произведена оценка производительности конвейера.

1.1 Конвейерная обработка

Конвейер — способ организации вычислений, используемый в современных процессорах и контроллерах с целью повышения их производительности (увеличения числа инструкций, выполняемых в единицу времени). Один из самых простых и наиболее распространенных способов повышения быстро- действия процессоров — конвейеризация процесса вычислений. Конвейеризация – это техника, в результате которой задача или команда разбивается на некоторое число подзадач, которые выполняются последовательно. Каждая подкоман- да выполняется на своем логическом устройстве. Все логические устройства (ступени) соединяются последовательно таким образом, что выход i-ой ступени связан с входом (i+1)-ой ступени, все ступени работают одновременно. Множество ступеней называется конвейером. Выигрыш во времени достигается при выполнении нескольких задач за счет параллельной работы ступеней, вовлекая на каждом такте новую задачу или команду.

1.2 Оценка производительности конвейера

Пусть задана операция, выполнение которой разбито на n последовательных этапов. При последовательном их выполнении операция выполняется за время

$$\tau_e = \sum_{i=1}^{n} \tau_i \tag{1.1}$$

где

n — количество последовательных этапов;

τ_i — время выполнения i-го этапа;

Быстродействие одного процессора, выполняющего только эту операцию, составит

$$S_e = \frac{1}{\tau_e} = \frac{1}{\sum\limits_{i=1}^{n} \tau_i} \tag{1.2}$$

где

τ_e — время выполнения одной операции;

n — количество последовательных этапов;

τ_i — время выполнения i-го этапа;

Выберем время такта — величину $t_T = max\sum_{i=1}^{n}(\tau_i)$ и потребуем при разбиении на этапы, чтобы для любого i = 1, ... , n выполнялось условие $(\tau_i + \tau_{i+1})modn = \tau_T$. То есть чтобы никакие два последовательных этапа (включая конец и новое начало операции) не могли быть выполнены за время одного такта.

Максимальное быстродействие процессора при полной загрузке конвейера составляет

$$S_{max} = \frac{1}{\tau_T} \tag{1.3}$$

где

τ_T — выбранное нами время такта;

Число n — количество уровней конвейера, или глубина перекрытия, так как каждый такт на конвейере параллельно выполняются n операций. Чем больше число уровней (станций), тем больший выигрыш в быстродействии может быть получен.

Известна оценка

$$\frac{n}{n/2} \leq \frac{S_{max}}{S_e} \leq n \tag{1.4}$$

где

S_{max} — максимальное быстродействие процессора при полной загрузке конвейера;

S_e — стандартное быстродействие процессора;

n — количество этапов.

то есть выигрыш в быстродействии получается от n/2 до n раз [2].

Реальный выигрыш в быстродействии оказывается всегда меньше, чем указанный выше, поскольку:

1) некоторые операции, например, над целыми, могут выполняться за меньшее количество этапов, чем другие арифметические операции. Тогда отдельные станции конвейера будут простаивать;

2) при выполнении некоторых операций на определённых этапах могут требоваться результаты более поздних, ещё не выполненных этапов предыдущих операций. Приходится приостанавливать конвейер;

3) поток команд(первая ступень) порождает недостаточное количество операций для полной загрузки конвейера.

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) [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

1.4 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 Конвейерный алгоритм без многопоточности

Есть последовательность строк, длиной n.
Чтобы подсчитать количество вхождений каждого символа в каждой строке, необходимо создать n дополнительных массивов, каждый из которых имеет размерность m (m - мощность некого алфавита, на основе которого строятся строки). Далее необходимо про-

итерировать каждую из строк и инкрементировать соответствующую ячейку массива.
Пример:

$$\text{strings} = \{\text{"qwer"}, \text{"asdf"}, \text{"zxcv"}\}$$

Все строки составляются из английских строчных букв. Следжовательно мощность алфавита - 26.
Создаем дополнительные массивы размерностью 26:

$$\text{arr1} = \text{arr2} = \text{arr3} = [26]$$

Возьмем 2 конвейера. Тогда каждый из конвейеров должен обработать свою часть.
Так как конвейеров 2, то каждый из конвейеров должен обработать по половине каждой строки.

Конвейер 1:	**Конвейер 2:**
	"qwer"
arr1['q'] = arr1['q'] + 1	arr1['e'] = arr1['c'] + 1
arr1['w'] = arr1['w'] + 1	arr1['r'] = arr1['d'] + 1
	"asdf"
arr2['a'] = arr2['a'] + 1	arr2['d'] = arr2['f'] + 1
arr2['s'] = arr2['s'] + 1	arr2['f'] = arr2['d'] + 1
	"zxcv"
arr3['z'] = arr3['z'] + 1	arr3['c'] = arr3['j'] + 1
arr3['x'] = arr3['x'] + 1	arr3['v'] = arr3['i'] + 1

Таким образом в каждом из соответствуюших массивов хранится колчиство вхождений каждого из символов алфавита в соответствующей строке.

1.6 Конвейерный алгоритм 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** стркои, а передавать **уже обработанные** строки на обработку следующему конвейеру.

1.7 Вывод

В данном разделе были рассмотрены основы конвейерной обработки и произведена оценка производительности конвейера.

2 Конструкторская часть

2.1 Описание алгоритмов

В данном разделе будут описан каждый исследуемый алгоритм.

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

2.2 Разработка алгоритмов

В данном разделе представлены схемы реализуемых алгоритмов.

На рисунке 2.1 представлена схема последовательного конвейерного алгоритма подсчёта.

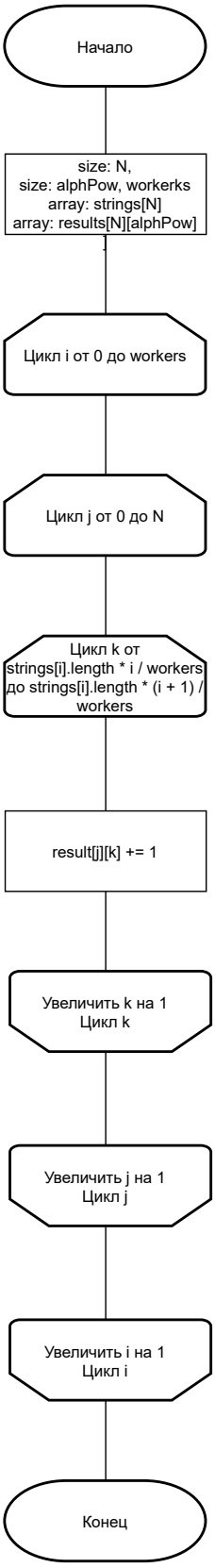

Рис. 2.1: Схема последовательного конвейерного алгоритма подсчёта

На рисунке 2.2.1 представлена схема главного потока параллельного конвейерного алгоритма.

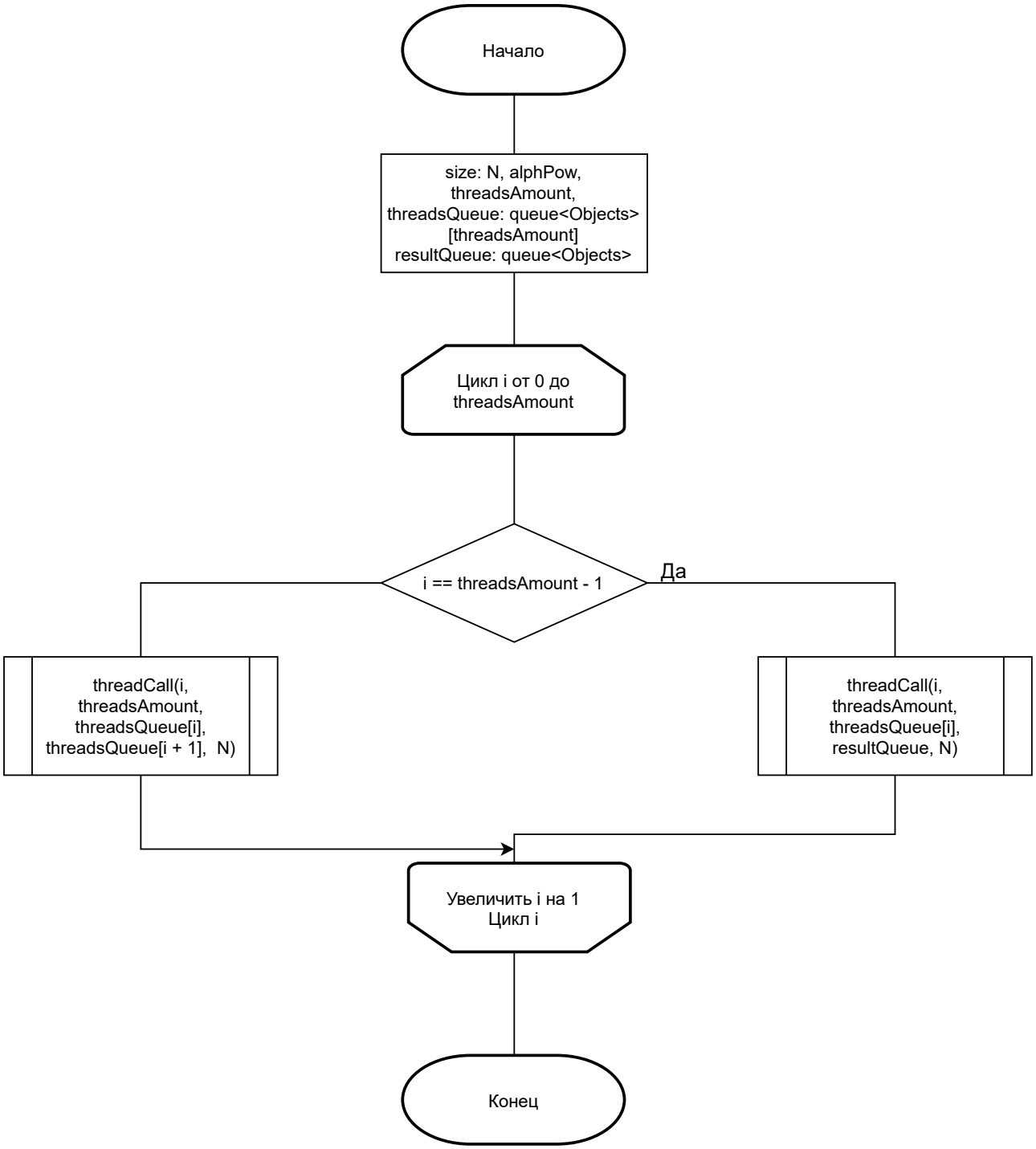

Рис. 2.2.1: Схема главного потока параллельного конвейерного алгоритма

На рисунке 2.2.2 представлена схема дочернего потока параллельного конвейерного алгоритма.

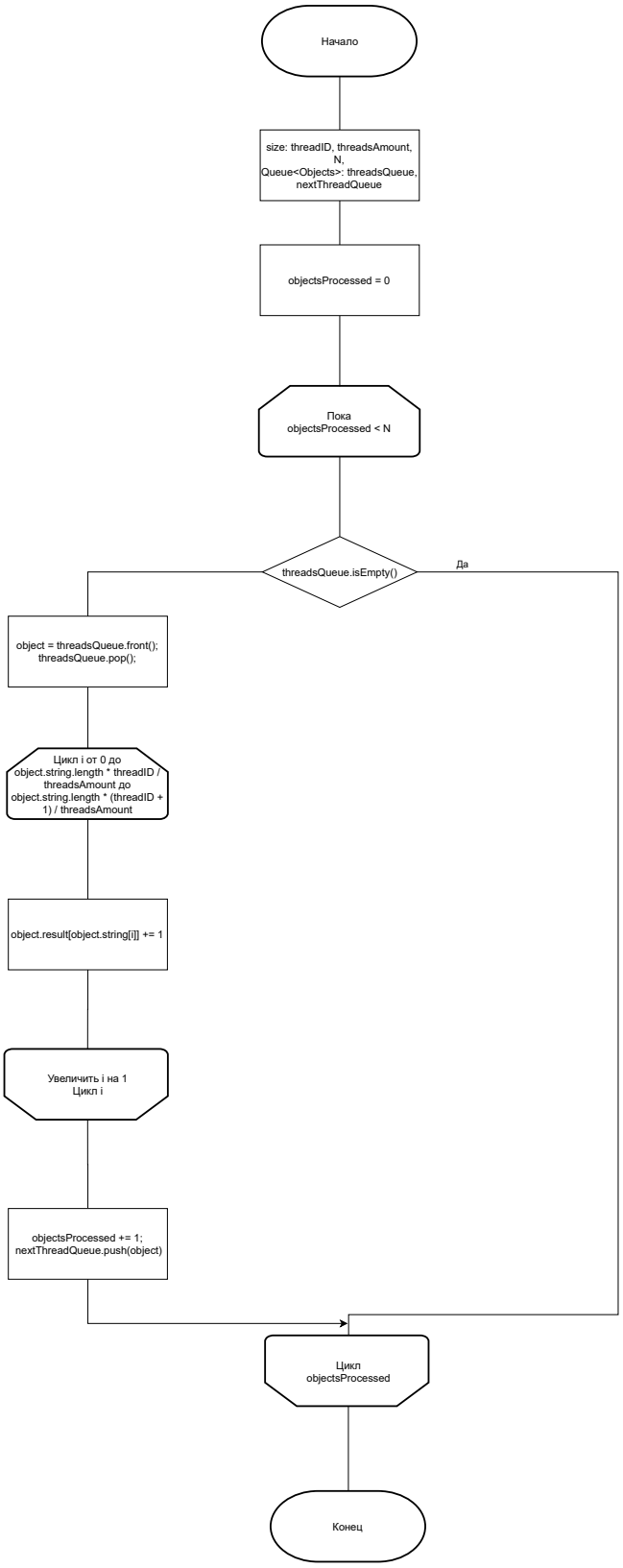

Рис. 2.2.2: Схема дочернего потока параллельного конвейерного алгоритма

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 строк, сами строки
Выходные данные: массивы, содержащие количество вхождений каждого из символов в каждой строке
Среда выполнения: Windows 10 x64 CPU: Core i7-8550U

3.2 Средства реализации

Для выполнения данной лабораторной работы использовался язык программирования C++ стандарта 2020 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  static void count_letters_in_object(work_object& object,
2                                      size_t start_letter,
3                                      size_t end_letter) {
4      for (size_t i = start_letter; i < end_letter; i++) {
5          object.result[object.string[i] - 'a']++;
6      }
7  }
8
9  void linear_worker(std::queue<work_object>& objects,
10                    const size_t worker_number,
11                    const size_t worker_amount,
12                    size_t elements_to_process) {
13     for (size_t i = 0; i < elements_to_process; i++) {
14         work_object object = objects.front();
15         objects.pop();
16         count_letters_in_object(object,
```

```
17                                  object.string.length() *
18                                  (double)worker_number /
19                                  worker_amount,
20                                  object.string.length() *
21                                  (double)(worker_number + 1) /
22                                  worker_amount);
23          objects.push(object);
24      }
25  }
26
27  std::vector<work_object> init_linear_work(
28                              std::vector<std::string>& strings,
29                              const int worker_amount) {
30      std::queue<work_object> objects;
31      for (auto& string : strings) {
32          objects.emplace(work_object(string));
33      }
34      MyTimer Timer;
35      for (int i = 0; i < worker_amount; i++) {
36          linear_worker(objects, i, worker_amount, strings.size());
37      }
38      std::cout << "Linear algorithm works for: " <<
39                   Timer.elapsed() <<
40                   " seconds" << std::endl;
41      std::vector<work_object> result;
42      while (!objects.empty()) {
43          result.push_back(objects.front());
44          objects.pop();
45      }
46      return result;
47  }
```

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

Листинг 3.2.1: Главный поток параллелльного конвейерного алгоритма подсчёта

```
std::vector<WorkObject> init_conveyor_work(
                        std::vector<std::string>& strings,
                        const int workers_amount) {
    std::queue<WorkObject> complete_objects;
    std::vector<std::queue<WorkObject>> workers_queues(workers_amount);
    std::vector<std::thread> threads;
    for (auto& string : strings) {
        workers_queues[0].push(WorkObject(string));
    }

    std::vector<std::mutex> mutexes(workers_amount + 1);
    MyTimer timerAllWork;
    for (size_t i = 0; i < workers_amount; i++) {
        threads.emplace_back(std::thread(threadWork, i,
                             workers_amount,
                             std::ref(workers_queues[i]),
                             i == workers_amount - 1 ?
                             std::ref(complete_objects) :
                             std::ref(workers_queues[i + 1]),
                             strings.size(),
                             std::ref(mutexes[i]),
                             std::ref(mutexes[i + 1])));
    }

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

    std::cout << "All Conveyor worked for: " <<
                 timerAllWork.elapsed() <<
                 " seconds" << std::endl;
    std::vector<WorkObject> result;
    while (!complete_objects.empty()) {
        result.push_back(complete_objects.front());
        complete_objects.pop();
    }
    return result;
```

```
38 }
```

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

Листинг 3.2.2: Дочерний поток параллельного конвейерного алгоритма подсчёта

```
1  static void count_letters_in_object (WorkObject& object,
2                                       size_t start_letter,
3                                       size_t end_letter) {
4      for (size_t i = start_letter; i < end_letter; i++) {
5          object.result[object.string[i] - 'a']++;
6      }
7  }
8
9  std::mutex allThreadsLock;
10
11
12 void threadWork(const int thread_number,
13                  const int threads_amount,
14                  std::queue<WorkObject>& threads_queue,
15                  std::queue<WorkObject>& next_thread_queue,
16                  size_t objects_to_process,
17                  std::mutex& threads_mutex,
18                  std::mutex& next_threads_mutex) {
19      MyTimer timerThreadWork;
20      size_t sleepTime;
21      size_t processed = 0;
22      while(processed != objects_to_process) {
23          if (threads_queue.size()) {
24              WorkObject object = threads_queue.front();
25
26              threads_mutex.lock();
27              threads_queue.pop();
28              threads_mutex.unlock();
29
30              count_letters_in_object(object,
31                                      object.string.length() *
32                                      (double)thread_number /
33                                      threads_amount,
34                                      object.string.length() *
35                                      (double)(thread_number + 1) /
```

```
36                                      threads_amount);
37
38              next_threads_mutex.lock();
39              next_thread_queue.push(object);
40              next_threads_mutex.unlock();
41
42              processed++;
43          } else {
44              usleep(1000);
45              sleepTime += 1;
46          }
47      }
48
49      allThreadsLock.lock();
50      std::cout << "Thread #" << thread_number <<
51                   " worked for " << timerThreadWork.elapsed() <<
52                   " seconds" << std::endl;
53
54      std::cout << "Thread #" << thread_number <<
55                   " sleeped for " << sleepTime <<
56                   " milliseconds" << std::endl;
57      allThreadsLock.unlock();
58  }
```

3.4 Описание тестирования

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

Ниже, на листинге 3.3, представлен фрагмент кода тестирования корректной работы реализации алгоритмов

Листинг 3.3: Тестирование корректной работы алгоритмов

```
int tests(size_t words_amount, size_t letters_in_word) {
    std::vector<std::string> strings;
    for (size_t i = 0; i < words_amount; i++) {
        std::string string;
        for (size_t j = 0; j < letters_in_word; j++) {
            string.push_back(rand() % LETTERS_IN_ENG_ALPHABET + 'a');
        }
        strings.emplace_back(string);
    }

    std::vector<WorkObject> result_consistent =
                            initLinearWork(strings, 3);

    std::vector<WorkObject> result_conveyor =
                            initConveyorWork(strings, 3);
    if (result_consistent != result_conveyor) {
        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.
Для всех замеров количество конвейеров = 3.

Ниже, на графике 4.1, показана графическая интерпретация замеров времени работы

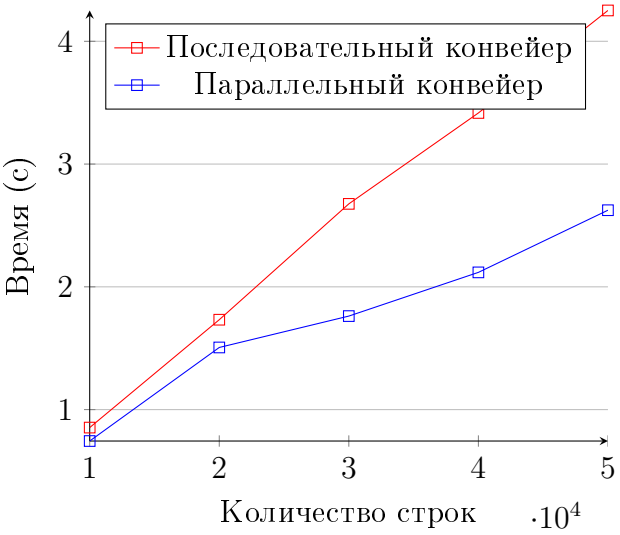

График 4.1: Замеры вермени работы первого параллельного алгоритма

Вывод

По результатам исследования выяснилось, что при малых размерностях входных данных разница по времени между алгоритмами не существенна, но с увелечением размерности разница становится всё больше и больше.

Заключение

В ходе работы были изучен алгоритм нахождения количества вхождений каждого символа в наборе строк, а также разработаны 2 конвейреные версии этого алгоритма: последовательный и параллельный. Экспериментально было установлено, что параллельная версии быстрее последовательной алгоритма.

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).