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

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

Москва, 2020

Содержание

Введение **3**

1 Аналитическая часть **4**
1.1 Описание алгоритмов . . . 4
1.2 Стандартный алгоритм . . . 4
1.3 Первая реализация паралелльного алгоритма . . . 4
1.4 Вторая реализация паралелльного алгоритма . . . 5
1.5 Параллельное программирование . . . 5
1.6 Организация взаимодействия параллельных потоков . . . 5
Вывод . . . 6

2 Конструкторская часть **7**
2.1 Разработка алгоритмов . . . 8
Вывод . . . 13

3 Технологическая часть **14**
3.1 Требования к программному обеспечению . . . 14
3.2 Средства реализации . . . 14
3.3 Листинг кода . . . 14
3.4 Описание тестирования . . . 18
Вывод . . . 18

4 Экспериментальная часть **19**
4.1 Сравнительный анализ на основе замеров времени работы алгоритмов . . . 19

Заключение **21**

Введение

Цель данной работы - изучение возможности параллельных вычислений и использование такого подхода на практике.

В даной лабораторной работе рассматривается простой алгоритм нахождения суммы элементов подмассива и его параллельная версия, представленная в двух вариантах.
Задачами данной лабораторной работы являются:

- рассмотрение стандартного алгоритма подсчёта суммы элементов массива;
- проведение сравнительного анализа последовательного алгоритма подсчёта и двух его пареллельных реализаций;
- определение зависимости времени работы алгоритма от числа потоков исполнения и количества слов.

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 последовательность чисел, длина последовательности - n.
Для того, чтобы подсчитать сумму элементов каждого подмассива, необходимо создать дополнительную переменную, и массив, который имеет размерность n.

Далее необходимо проитерировать каждый подмассив, и подсчитать в нем сумму.
Пример:

$$\text{numbers} = \{\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8, 9\}\}$$

Проходимся по каждому подмассиву, ищем сумму и записываем в результирующий вектор res.

$$\text{sum}(\{1, 2, 3\}) = 6$$
$$\text{res} = \{6\}$$
$$\text{sum}(\{4, 5, 6\}) = 15$$
$$\text{res} = \{6, 15\}$$
$$\text{sum}(\{7, 8, 9\}) = 24$$
$$\text{res} = \{6, 15, 24\}$$

Сумма ищется следющим способом:
Пример:
Возьмем подмассив $\{1, 2, 3\}$
Объявлем переменную $\text{resuslt} = 0$, в которую будет записываться сумма подмассива.
Далее итерируемся по подмассиву, прибавляя к result текущий элемент.

Таким образом, в res хранится последовательность, означающая сумму чисел каждого подмассива исходного массива.

1.3 Первая реализация паралелльного алгоритма

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

$$\text{numbers} = \{\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8, 9\}\}$$

Первый поток берёт на обработку подмассив $\{1, 2, 3\}$ (обработка аналогична описанной в пункте выше). Второй поток в это время берёт на обработку подмассив $\{4, 5, 6\}$. Далее первый поток берёт на обработку подмассив $\{7, 8, 9\}$. Выполнение алгоритма завершено.
В данном алгоритме очень важно следить за корректным забором элементов из очереди, а также за корректном добавлении в результирующий массив.

1.4 Вторая реализация паралелльного алгоритма

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

$$\text{numbers} = \{\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8, 9\}, \{\{1, 2, 4\}, \{5, 7, 9\}, \{6, 8, 4\}\}$$

Колчиство подмассивов - 6.
Количество потоков - 3.
Слеовательно, количество подмассивов на обработку для каждого потока - 2.
Для упрошения каждому из потоков можно давать на обработку те подмассивы, номера которых лежат между номером потока, умноженного на количество подмассивов, поделённое на количество потоков и номером следующего потока, умноженного на количество подмассивов, поделённое на количество потоков:

$$start = \frac{threadID * arrQuantity}{threadsQuantity} \tag{1.1}$$

$$end = \frac{(threadID + 1) * arrQuantity}{threadsQuantity} \tag{1.2}$$

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 исполняются в общем адресном пространстве параллельной программы. Как результат, взаимодействие параллельных потоков можно организовать через использование общих данных, являющихся доступными для всех потоков. Наиболее простая ситуация состоит в использовании общих данных только для чтения. В случае же, когда общие

данные могут изменяться несколькими потоками, необходимо пользоваться примитивами синхронизации: атомарными переменными, мьютексами.

Вывод

В данном разделе была рассмотрена теоретичсекая часть алгоритма поиска суммы в каждом подмассиве массива, 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 подмассивов и сами подмассива.
Требования к программе:: Подсчёт суммы элементов каждого подмассива.

2.1 Разработка алгоритмов

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

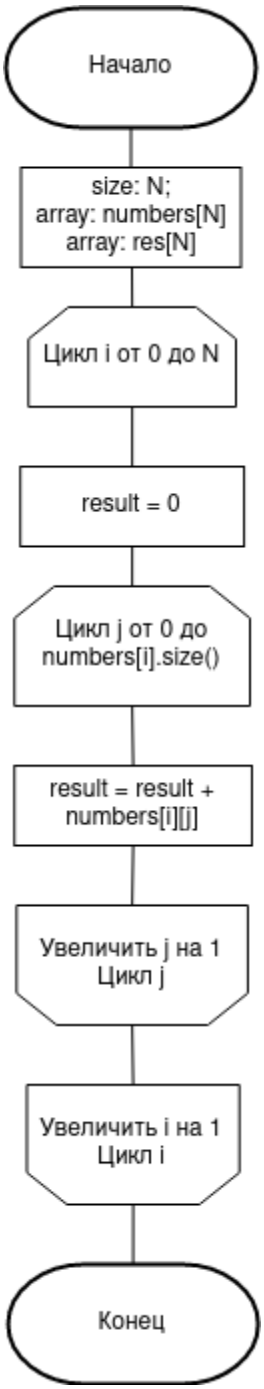

Рис. 2.1: Схема последовательного алгоритма подсчёта

На рисунке 2.2.1 представлена схема главного потока первого парллельного алгоритма.

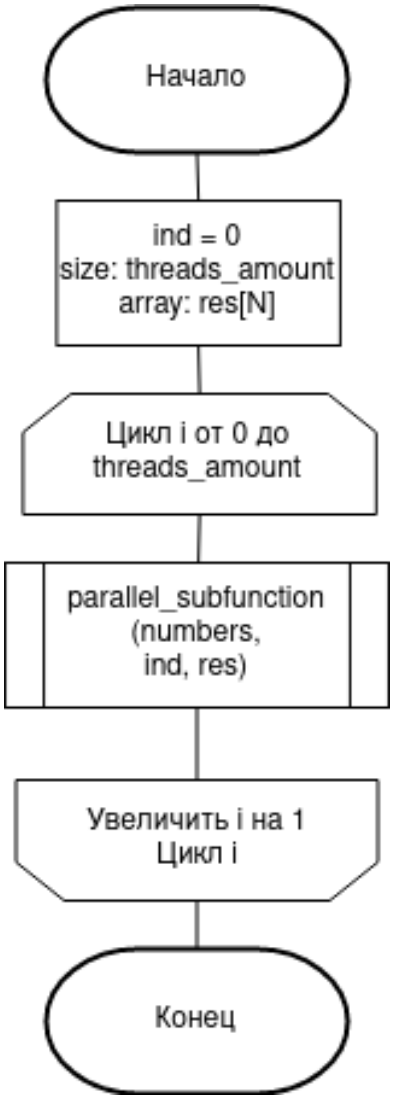

Рис. 2.2.1: Схема главного потока первого парллельного алгоритма

На рисунке 2.2.2 представлена схема дочернего потока первого парллельного алгоритма.

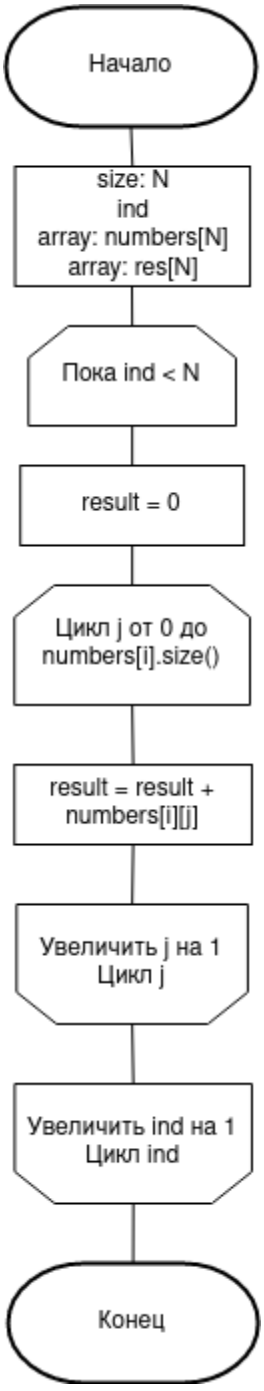

Рис. 2.2.2: Схема дочернего потока первого парллельного алгоритма

На рисунке 2.3.1 представлена схема главного потока второго парллельного алгоритма.

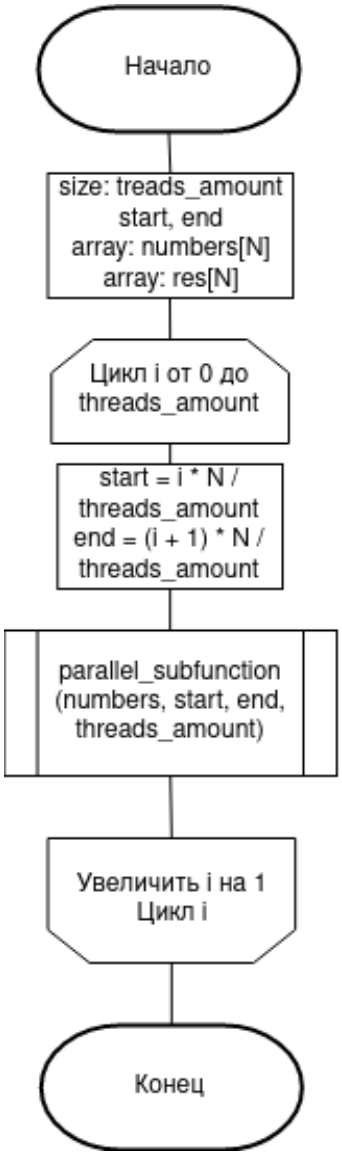

Рис. 2.3.1: Схема главного потока второго парллельного алгоритма

На рисунке 2.3.2 представлена схема дочернего потока второго парллельного алгоритма.

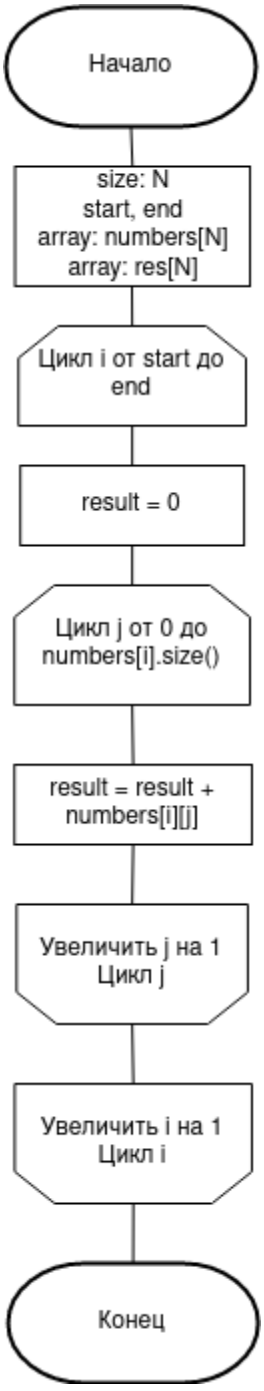

Рис. 2.3.2: Схема дочернего потока второго парллельного алгоритма

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: массив с суммами элементов подмассивов
Среда выполнения: Linux Ubuntu 20.04 x64
CPU: Intel 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  std::vector<int> count_sum_ordinary(
2          std::vector<std::vector<int>> numbers) {
3      std::vector<int> res;
4
5      for (int i = 0; i < numbers.size(); i++) {
6          int result = 0;
7          for (int j = 0; j < numbers[i].size(); j++) {
8              result = result + numbers[i][j];
9          }
10         res.push_back(result);
11     }
12
13     return res;
14 }
```

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<int> count_sum_parallel_queue
        (const std::vector<std::vector<int>> numbers,
                                        int threads_amount) {
    std::vector<int> res;
    std::vector<std::thread> thread_vector;

    int ind = 0;
    for (int i = 0; i < threads_amount; i++) {
        thread_vector.emplace_back(std::thread(parallel_subfunction,
                                               numbers,
                                               std::ref(ind),
                                               std::ref(res)));
    }

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

    return res;
}
```

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 parallel_subfunction(
    const std::vector<std::vector<int>> numbers,
                                        int &ind,
                            std::vector<int> &res) {

    while (true) {

        mtx.lock();
        if (ind >= numbers.size()) {
            mtx.unlock();
            break;
        }
        int cur_ind = ind++;
        mtx.unlock();
        int result = 0;
        for (int i = 0; i < numbers[cur_ind].size(); i++) {
            result = result + numbers[cur_ind][i];
        }
        mtx.lock();
        res.push_back(result);
        mtx.unlock();
```

```
22          }
23  }
```

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<int> count_sum_parallel_distribution(
            std::vector<std::vector<int>> numbers,
                                int threads_amount) {

    std::vector<int> res(numbers.size());

    std::vector<std::thread> thread_vector;

    for (int i = 0; i < threads_amount; i++) {
        int start = ((double)numbers.size() / threads_amount * i);
        int end = ((double)numbers.size() / threads_amount * (i + 1));
        thread_vector.emplace_back(std::thread(parallel_subfunction,
                                               ref(numbers),
                                               start,
                                               end,
                                               std::ref(res)));
    }

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


    return res;
}
```

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 parallel_subfunction(
        std::vector<std::vector<int>> &numbers,
                                    int start,
                                    int end,
                        std::vector<int> &res) {

    for (int i = start; i < end; i++) {
        int result = 0;
        for (int j = 0; j < numbers[i].size(); j++) {
            result = result + numbers[i][j];
        }

        res[i] = result;
    }
}
```

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 tests() {

    std::vector<int> res1;
    std::vector<int> res2;
    std::vector<int> res3;


    int arr_amount = 400;
    int arr_size = 100;

    std::vector<std::vector<int>> arr;

    for (int i = 0; i < arr_amount; i++) {
        std::vector<int> sub_arr;
        for (int j = 0; j < arr_size; j++) {
            sub_arr.push_back(rand() % 9 + 1);
        }
        arr.push_back(sub_arr);
    }

    res1 = count_sum_ordinary(arr);
    res2 = count_sum_parallel_queue(arr, 16);
    res3 = count_sum_parallel_distribution(arr, 16);

    if (arr1 == arr2 == arr3) return 0;
    else return -1;


    return 0;
}
```

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

Для сравнения замеров времени были взяты подмассивы размерами по 1000 элементов и длины массивов от 10000 до 50000 с шагом 10000. Каждый эксперимент на каждой размерности был произведён 10 раз, затем бралось среднее арифметическое полученного результата. Ниже, на графике 4.1, видно, что обе параллельные версии алгоритма выигрывают у последовательной:

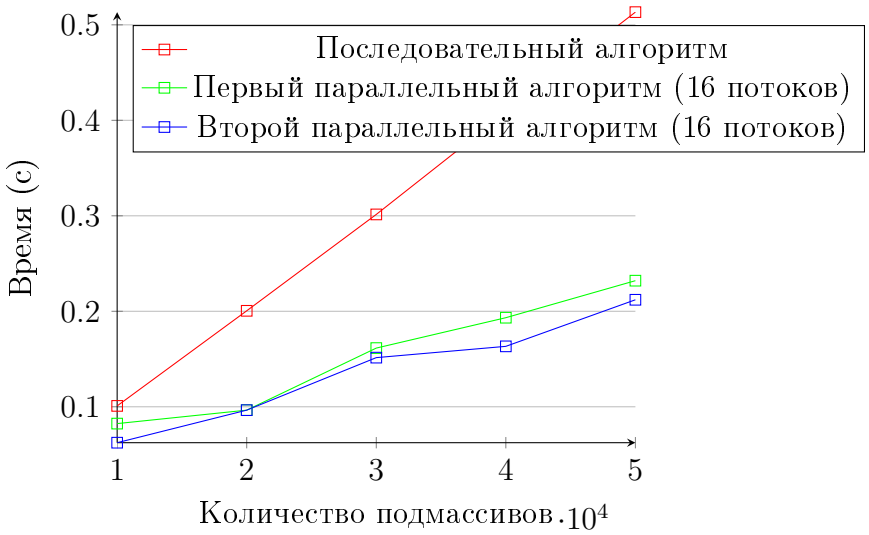

График 4.1: Замеры вермени работы алгоритмов

Был проведён замер времени работы каждого из параллельных алгоритмов. Длина каждого подмассива 10000 символов. Каждый эксперимент на каждой размерности был произведён 10 раз, затем бралось среднее арифметическое полученного результата.
Все эксперименты были проведены для 80000 подмассивов.
Ниже во всех таблицах время выполнения указано в секундах.

Ниже, на графике 4.2, показана зависимость времени выполнения программы от количества потоков для первой параллельной версии алгоритма:

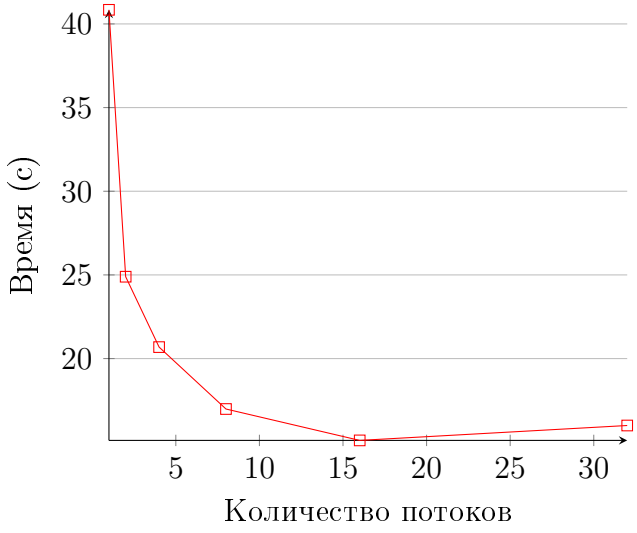

График 4.2: Замеры вермени работы первого параллельного алгоритма

Ниже, на графике 4.3, показана зависимость времени выполнения программы от количества строк и количества потоков для первой параллельной версии алгоритма:

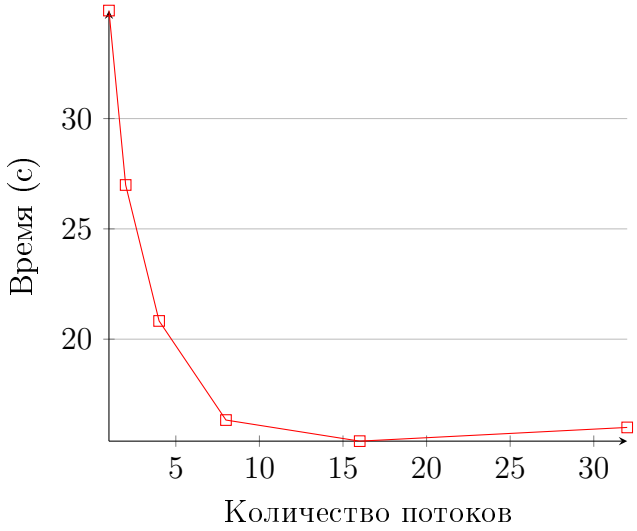

График 4.3: Замеры вермени работы второго параллельного алгоритма

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

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. Причем самой быстрой версией параллельного алгоритма(любой из его версий) является та, где число потоков равно числу логических процессоров. 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 подсчета суммы каждого подмассива в массиве, а также разработаны 3 версии этого алгоритма: 1 последовательная и 2 параллельных. Экспериментально было установлено, что параллельные версии быстрее классического алгоритма, причем чем больше количество подмассивов на вход - тем больше выигрыш. Было установлено, что максимальная скорость работы параллельных алгоритмов достигается при равенстве количества потоков числу логических процессоров в системе.

Список 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).