

Министерство науки и высшего образования Российской Федерации  
Федеральное государственное автономное образовательное учреждение  
высшего образования  
«Нижегородский государственный университет им. Н.И. Лобачевского»  
**Институт информационных технологий, математики и механики**  
Направление подготовки: Фундаментальная информатика и  
информационные технологии

## **Отчет**

по лабораторной работе

# **Сортировка Хоара со слиянием «Разделяй и властвуй»**

**Выполнила:**

студентка группы 381606-1

Батова Д.А.

**Проверил:**

доцент каф. МОСТ, ИИТММ,

кандидат технических наук

Сысоев А.В.

Нижний Новгород

2019

# Содержание

Введение.....	3
Постановка задачи.....	6
Метод решения.....	7
Описание схемы распараллеливания .....	11
Описание OpenMP-версии .....	15
Описание TBV-версии .....	18
Подтверждение корректности .....	20
Результаты экспериментов.....	22
Выводы из результатов.....	24
Заключение .....	28
Литература .....	29

## Введение

Первые прототипы современных методов сортировки появились уже в XIX веке. К 1890 году для ускорения обработки данных переписи населения в США американец Герман Холлерит создал первый статистический табулятор — электромеханическую машину, предназначенную для автоматической обработки информации, записанной на перфокартах. У машины Холлерита имелся специальный «сортировальный ящик» из 26 внутренних отделений. При работе с машиной от оператора требовалось вставить перфокарту и опустить рукоятку. Благодаря пробитым на перфокарте отверстиям замыкалась определённая электрическая цепь, и на единицу увеличивалось показание связанного с ней циферблата. Одновременно с этим открывалась одна из 26 крышек сортировального ящика, и в соответствующее отделение перемещалась перфокарта, после чего крышка закрывалась. Данная машина позволила обрабатывать около 50 карт в минуту, что ускорило обработку данных в 3 раза. К переписи населения 1900 года Холлерит усовершенствовал машину, автоматизировав подачу карт. Работа сортировальной машины Холлерита основывалась на методах поразрядной сортировки. В патенте на машину обозначена сортировка «по отдельности для каждого столбца», но не определён порядок. В другой аналогичной машине, запатентованной в 1894 году Джоном Гором, упоминается сортировка со столбца десятков. Метод сортировки, начиная со столбца единиц, впервые появляется в литературе в конце 1930-х годов. К этому времени сортировальные машины уже позволяли обрабатывать до 400 карт в минуту.

В дальнейшем история алгоритмов оказалась связана с развитием электронно-вычислительных машин. По некоторым источникам, именно программа сортировки стала первой программой для вычислительных машин. Некоторые конструкторы ЭВМ, в частности разработчики EDVAC, называли задачу сортировки данных наиболее характерной нечисловой задачей для

вычислительных машин. В 1945 году Джон фон Нейман для тестирования ряда команд для EDVAC разработал программы сортировки методом слияния. В том же году немецкий инженер Конрад Цузе разработал программу для сортировки методом простой вставки. К этому времени уже появились быстрые специализированные сортировальные машины, в сопоставлении с которыми и оценивалась эффективность разрабатываемых ЭВМ. Первым опубликованным обсуждением сортировки с помощью вычислительных машин стала лекция Джона Мокли, прочитанная им в 1946 году. Мокли показал, что сортировка может быть полезной также и для численных расчетов, описал методы сортировки простой вставки и бинарных вставок, а также поразрядную сортировку с частичными проходами. Позже организованная им совместно с инженером Джоном Эккертом компания «Eckert–Mauchly Computer Corporation» выпустила некоторые из самых ранних электронных вычислительных машин BINAC и UNIVAC. Наряду с отмеченными алгоритмами внутренней сортировки, появлялись алгоритмы внешней сортировки, развитию которых способствовал ограниченный объём памяти первых вычислительных машин. В частности, были предложены методы сбалансированной двухпутевой поразрядной сортировки и сбалансированного двухпутевого слияния.

К 1952 году на практике уже применялись многие методы внутренней сортировки, но теория была развита сравнительно слабо. В октябре 1952 года Даниэль Гольденберг привёл пять методов сортировки с анализом наилучшего и наихудшего случаев для каждого из них. В 1954 году Гарольд Сьюворд развил идеи Гольденберга, а также проанализировал методы внешней сортировки. Говард Демут в 1956 году рассмотрел три абстрактные модели задачи сортировки: с использованием циклической памяти, линейной памяти и памяти с произвольным доступом. Для каждой из этих задач автор предложил оптимальные или почти оптимальные методы сортировки, что помогло связать теорию с практикой. Из-за малого числа людей, связанных с вычислительной техникой, эти доклады не появлялись в «открытой литературе». Первой большой

обзорной статьёй о сортировке, появившейся в печати в 1955 году, стала работа Дж. Хоскена, в которой он описал всё имевшееся на тот момент оборудование специального назначения и методы сортировки для ЭВМ, основываясь на брошюрах фирм-изготовителей. В 1956 году Э. Френд в своей работе проанализировал математические свойства большого числа алгоритмов внутренней и внешней сортировки, предложив некоторые новые методы.

После этого было предложено множество различных алгоритмов сортировки: например, вычисление адреса в 1956 году; слияние с вставкой, обменная поразрядная сортировка, каскадное слияние и метод Шелла в 1959 году, многофазное слияние и вставки в дерево в 1960 году, осциллирующая сортировка и быстрая сортировка Хоара в 1962 году, пирамидальная сортировка Уильямса и обменная сортировка со слиянием Бэтчера в 1964 году. В конце 60-х годов произошло и интенсивное развитие теории сортировки. Появившиеся позже алгоритмы во многом являлись вариациями уже известных методов. Получили распространение адаптивные методы сортировки, ориентированные на более быстрое выполнение в случаях, когда входная последовательность удовлетворяет заранее установленным критериям.

## **Постановка задачи**

Основной целью данной лабораторной работы является реализация сортировки Хоара с использованием основных технологий параллельного программирования для систем с общей памятью (OpenMP, TBB).

Процесс выполнения лабораторной работы можно разделить на несколько основных этапов:

1. Реализация последовательной версии алгоритма сортировки Хоара с имитацией параллельной работы алгоритма.
2. Реализация параллельной версии алгоритма сортировки Хоара со слиянием «Разделяй и властвуй» с помощью технологии OpenMP.
3. Реализация параллельной версии алгоритма сортировки Хоара со слиянием «Разделяй и властвуй» с помощью технологии TBB.
4. Сравнение времени работы и ускорения полученных реализаций на сгенерированном случайном массиве данных.
5. Анализ полученных в четвёртом пункте результатов и подведение итогов.

## Метод решения

Алгоритм *быстрой сортировки* разработал в 1960 году английский ученый Чарльз Хоар, занимавшийся тогда в МГУ машинным переводом. Алгоритм, по принципу функционирования, входит в класс обменных сортировок (сортировка перемешиванием, пузырьковая сортировка и др.), выделяясь при этом высокой скоростью работы.

Отличительной особенностью быстрой сортировки является операция разбиения массива на две части относительно опорного элемента. Например, если последовательность требуется упорядочить по возрастанию, то в левую часть будут помещены все элементы, значения которых меньше значения опорного элемента, а в правую элементы, чьи значения больше или равны опорному.

Вне зависимости от того, какой элемент выбран в качестве опорного, массив будет отсортирован, но все же наиболее удачным считается ситуация, когда по обеим сторонам от опорного элемента оказывается примерно равное количество элементов. Если длина какой-то из получившихся в результате разбиения частей превышает один элемент, то для нее нужно рекурсивно выполнить упорядочивание, т. е. повторно запустить алгоритм на каждом из отрезков.

Таким образом, алгоритм быстрой сортировки включает в себя два основных этапа:

- разбиение массива относительно опорного элемента;
- рекурсивная сортировка каждой части массива.

Выбор опорного элемента не влияет на результат, и потому может пасть на произвольный элемент. Тем не менее, как было замечено выше, наибольшая эффективность алгоритма достигается при выборе опорного элемента, делящего последовательность на равные или примерно равные части. Но, как правило, из-

за нехватки информации не представляется возможности наверняка определить такой элемент, поэтому зачастую приходится выбирать опорный элемент случайным образом.

В следующих пяти пунктах описана общая схема разбиения массива (считаем, что сортировка элементов осуществляется по возрастанию):

1. вводятся указатели *left* и *right* для обозначения начального и конечного элементов последовательности, а также опорный элемент *pivot*;
2. вычисляется значение опорного элемента  $\frac{left+right}{2}$ , заносится в переменную *pivot*;
3. указатель *left* смещается с шагом в 1 элемент к концу массива до тех пор, пока  $arr[left] > pivot$ . А указатель *right* смещается от конца массива к его началу, пока  $arr[right] < pivot$ ;
4. каждые два найденных элемента меняются местами;
5. пункты 3 и 4 выполняются до тех пор, пока  $left < right$ .

После разбиения последовательности следует проверить условие на необходимость дальнейшего продолжения сортировки его частей. Если в какой-то из получившихся в результате разбиения массива частей находится больше одного элемента, то следует произвести рекурсивное упорядочивание этой части, то есть выполнить над ней операцию разбиения, описанную выше. Для проверки условия «количество элементов  $> 1$ », нужно действовать примерно по следующей схеме:

Имеется массив  $arr[L..R]$ , где  $L$  и  $R$  – индексы крайних элементов этого массива. По окончании разбиения, указатели *left* и *right* оказались примерно в середине последовательности, тем самым образуя два отрезка: левый от  $L$  до *right* и правый от *left* до  $R$ . Выполнить рекурсивное упорядочивание



левой части нужно в том случае, если выполняется условие  $L < right$ . Для правой части условие аналогично:  $left < R$ .

Теперь на конкретном примере выполним разбиение массива.

Имеется массив целых чисел  $arr$ , состоящий из 8 элементов (рис. 5.5):  $arr[1..8]$ . Начальным значением  $left$  будет 1, а  $right$  – 8. Пройденная часть закрашивается голубым цветом.

В качестве опорного элемента возьмем элемент со значением 5, и индексом 4. Его мы вычислили, используя выражение  $\frac{left+right}{2}$ , отбросив дробную часть. Теперь  $pivot = 5$ .

6	7	2	5	9	1	3	8
---	---	---	---	---	---	---	---

Первый элемент левой части сравнивается с  $pivot$ .  $arr[1] > pivot$ , следовательно,  $left$  остается равным 1. Далее, элементы правой части сравниваются с  $pivot$ . Проверяется элемент с индексом 8 и значением 8.  $arr[8] > pivot$ , следовательно,  $right$  смещается на одну позицию влево.  $arr[7] < pivot$ , следовательно,  $right$  остается равным 7. На данный момент  $left = 1$ , а  $right = 7$ . Первый и седьмой элементы меняются местами. Оба указателя смещаются на одну позицию каждый в своем направлении.

3	7	2	5	9	1	6	8
---	---	---	---	---	---	---	---

Алгоритм снова переходит к сравнению элементов. Второй элемент сравнивается с опорным:  $arr[2] > pivot$ , следовательно  $left$  остается равным 2. Далее, элементы правой части сравниваются с  $pivot$ . Проверяется элемент с индексом 6 и значением 1:  $arr[6] < pivot$ , следовательно,  $right$  не изменяет своей позиции. На данный момент  $left = 2$ , а  $right = 6$ . Второй и шестой элементы меняются местами. Оба указателя смещаются на одну позицию каждый в своем направлении.

3	1	2	5	9	7	6	8
---	---	---	---	---	---	---	---

Алгоритм снова переходит к сравнению элементов. Третий элемент сравнивается с опорным:  $arr[3] < pivot$ , следовательно  $left$  смещается на одну позицию вправо. Далее, элементы правой части сравниваются с  $pivot$ . Проверяется элемент с индексом 5 и значением 9:  $arr[5] > pivot$ , следовательно,  $right$  смещается на одну позицию влево. Теперь  $left = right = 4$ , а значит, условие  $left < right$  не выполняется, этап разбиения завершается.

3	1	2	5	9	7	6	8
---	---	---	---	---	---	---	---

На этом этап разбиения закончен. Массив разделен на две части относительно опорного элемента. Осталось произвести рекурсивное упорядочивание его частей.

## Описание схемы распараллеливания

Для распараллеливания сортировки используется слияние «Разделяй и властвуй». Идея параллельной реализации с использованием такого типа слияния заключается в выполнении следующих шагов:

- Каждый поток отсортирует свою часть исходного массива.
- После сортировки частей массива выполняется слияние этих частей.

В первом пункте в качестве сортировки частей используется быстрая сортировка. Идея алгоритма описана в предыдущем разделе отчёта.

Во втором пункте используется алгоритм слияния «Разделяй и властвуй». Идея слияния по алгоритму "Разделяй и властвуй" заключается в разбиении массивов на участки, которые можно слить независимо. В первом массиве выбирается центральный элемент  $x$  (он разбивает массив на две равные половины), а во втором массиве с помощью бинарного поиска находится позиция наибольшего элемента меньше  $x$  (позиция этого элемента разбивает второй массив на две части). После такого разбиения первые и вторые половины массивов могут сливаться независимо, т.к. в первых половинах находятся элементы меньше элемента  $x$ , а во второй – большие (рис. 1). Для слияния двух массивов несколькими потоками можно в первом массиве выбрать несколько ведущих элементов, разделив его на равные порции, а во втором массиве найти соответствующие подмассивы. Каждый поток получит свои порции на обработку.

Эффективность такого слияния во многом зависит от того, насколько равномерно произошло "разделение" второго массива.

Слияние «Разделяй и властвуй» позволяет задействовать 2 потока при слиянии двух упорядоченных массивов. В этом случае слияние  $N$  массивов могут выполнять  $N$  параллельных потоков. На следующем шаге слияние  $N/2$  полученных массивов осуществляют  $N/2$  параллельных потоков. На последнем шаге два массива будут сливать 2 потока.



Рисунок 1. Слияние "Разделяй и властвуй"

Рассмотрим алгоритм слияния на основе последовательной версии, которая была направлена на создание имитации параллельной работы. Для наилучшего понимания изучим детально алгоритм слияния и выделим основные моменты и связанные с ними функции. Пусть на входе имеется неупорядоченный массив некоторой длины  $N$  (в последовательной версии симитирована работа двух потоков, поэтому дальнейшее описание будет так же основываться на факте, что рабочих потока у нас только 2):

1. Каждый поток осуществляет сортировку массива длины  $N/2$ . На этом шаге применяется алгоритм быстрой сортировки. Результатами этого шага являются два отсортированных разными потоками массива.
2. В одном из полученных массивов выбирается центральный элемент  $x$  (центральный по индексу), предполагая, что он является средним значением массива.
3. Во втором массиве отыскивается позиция наибольшего элемента меньшего  $x$ . Данную операцию можно было бы осуществить, применив всем известный алгоритм бинарного поиска. Однако реализация этого шага была немного видоизменена. Мы воспользовались двумя функциями: `std::lower_bound(int* first, int* last, int value)`, которая возвращает итератор, указывающий на первый элемент в диапазоне  $[first, last)$ , который не меньше (то есть больше или равен) значению  $value$  или последний элемент, если  $value$  не найден, и функцией `std::distance(int* first, int* last)`, которая возвращает расстояние между двумя указателями  $first$  и  $last$ . В результате этого шага во втором массиве находится позиция наибольшего элемента меньшего  $x$ .

4. Выполняется слияние первых половин массивов (в них расположены элементы меньше элемента  $x$ ) и вторых половин массивов (в них расположены элементы больше элемента  $x$ ). Получаем две отсортированные половинки, составляющие один результирующий массив.

```
void QuickSortMerge(int* array, int* res, int size) {
    int size1 = size / 2;
    int size2 = size - size1;
    quickSort(array, 0, size1-1);
    quickSort(array + size1, 0, size2 - 1);
    Merge_Two_Arrays_Into_Res(array, array + size1, size1, size2, res);
}
```

Рисунок 2. Функция QuickSortMerge. Первые четыре строчки иллюстрируют 1-й шаг, последняя - слияние массивов.

```
void Merge_Two_Arrays_Into_Res(int* arr1, int* arr2,
                               int size1, int size2, int* res) {
    int x = arr1[size1/2];
    auto it = std::lower_bound(arr2, arr2 + size2, x);
    int x1 = std::distance(arr2, it);
    int nSize1 = size1 / 2 + x1;
    Merge_Two_Arrays_Into_Tmp(arr1, arr2, res, size1 / 2, x1);
    Merge_Two_Arrays_Into_Tmp(arr1 + size1 / 2, arr2 + x1,
                               res + nSize1, size1 - size1 / 2, size2 - x1);
}
```

Рисунок 3. Функция Merge\_Two\_Arrays\_Into\_Res. Иллюстрирует 2-, 3-, 4-й шаги.

Стоит обратить внимание на функцию, которая осуществляет слияние двух отсортированных массивов (рис. 4). Мы проходимся по индексам массивов, сравнивая элементы, соответствующие этим индексам, и заполняем результирующий массив элементом, оказавшимся наименьшим, до тех пор, пока не дойдём до конца хотя бы одного массива. Если мы обошли первый массив, а второй массив ещё не закончился, то заполняем оставшуюся часть результирующего массива элементами второго массива простым циклом (это возможно в силу отсортированности массива)

```

void Merge_Two_Arrays_Into_Tmp(int* arr1, int* arr2,
    int* tmp, int size1, int size2) {
    int index1 = 0;
    int index2 = 0;
    int i = 0;

    for (; (index1 != size1) && (index2 != size2); i++) {
        if (arr1[index1] <= arr2[index2]) {
            tmp[i] = arr1[index1];
            index1++;
        } else {
            tmp[i] = arr2[index2];
            index2++;
        }
    }

    if (index1 == size1) {
        int j = index2;
        while (j < size2) {
            tmp[i++] = arr2[j++];
        }
    } else {
        int j = index1;
        while (j < size1) {
            tmp[i++] = arr1[j++];
        }
    }
}

```

Рисунок 4. Функция Merge\_Two\_Arrays\_Into\_Tmp.

## Описание OpenMP-версии

Для решения поставленной задачи необходимо выполнить разные процедуры обработки данных, при этом данные процедуры являются слабо связанными. В этом случае такие процедуры можно выполнить параллельно; такой подход обычно именуется *распараллеливанием по задачам*. Для поддержки такого способа организации параллельных вычислений в OpenMP для параллельного фрагмента программы, создаваемого при помощи директивы **parallel**, можно выделять параллельно выполняемые программные секции (директива **sections**).

Формат директивы **sections** имеет вид:

```
#pragma omp sections [<параметр> ...]
{
    #pragma omp section
    <блок_программы>
    #pragma omp section
    <блок_программы>
}
```

При помощи директивы **sections** выделяется программный код, который далее будет разделен на параллельно выполняемые секции. Директивы **section** определяют секции, которые могут быть выполнены параллельно (для первой по порядку секции директива **section** не является обязательной). В зависимости от взаимного сочетания количества потоков и количества определяемых секций, каждый поток может выполнить одну или несколько секций (вместе с тем, при малом количестве секций некоторые потоки могут оказаться и без секций и окажутся незагруженными). Как результат, можно отметить, что использование секций достаточно сложно поддается масштабированию (настройке на число имеющихся потоков).



Рисунок 5. Общая схема выполнения параллельных секций директивы sections

Отметим, что по умолчанию выполнение директивы **sections** синхронизировано, т.е. потоки, завершившие свои вычисления, ожидают окончания работы всех потоков для одновременного завершения директивы.

В нашей задаче мы воспользуемся параллельными секциями для возможности реализации парадигмы «Разделяй и властвуй», заключающейся в рекурсивном разбиении решаемой задачи на две или более подзадачи того же типа, но меньшего размера, и комбинировании их решений для получения ответа к исходной задаче; разбиения выполняются до тех пор, пока все подзадачи не окажутся элементарными. Таким образом, мы создадим рекурсивно две подзадачи с уменьшенным вдвое размером исходного массива и количеством потоком (задаётся пользователем, либо по умолчанию). Такое разбиение мы будем совершать до тех пор, пока количество потоков не станет равно 1 (в этом и заключается элементарность задачи). Как только это произошло, поток сортирует ту часть массива, которая ему выделена подзадачей, и принимает участие в дальнейшем слиянии своего отсортированного массива с массивами оставшихся потоков. Всё, что связано с непосредственной реализацией слияния массивов, было описано в предыдущем разделе. На рис. 6 представлена программная реализация OpenMP-версии.



```

void QuickSortMerge(int* array, int* res, int size, int pNum) {
    if (pNum == 1) {
        quickSort(array, 0, size - 1);
#pragma omp critical
        {
            ShowArray(array, size);
        }
    } else {
#pragma omp parallel sections
        {
#pragma omp section
            QuickSortMerge(array, res, size / 2, pNum / 2);
#pragma omp section
            QuickSortMerge(array + size / 2, res + size / 2,
                size - size / 2, pNum - pNum / 2);
        }
        Merge_Two_Arrays_Into_Res(array, array + size / 2,
            size / 2, size - size / 2, res);
        CopyArray(res, array, size);
#pragma omp critical
        {
            ShowArray(array, size);
        }
    }
}

```

Рисунок 6. Реализация OpenMP-версии.

## Описание ТВВ-версии

Для параллельной реализации слияния «Разделяй и властвуй» мы воспользуемся логическими задачами. Логическая задача в библиотеке ТВВ представлена в виде класса *tbb::task*. Этот класс является базовым при реализации задач, т.е. должен быть унаследован всеми пользовательскими логическими задачами. Класс *tbb::task* содержит виртуальный метод *task::execute*, в котором выполняются вычисления. В этом методе производятся необходимые вычисления, после чего возвращается указатель на следующую задачу, которую необходимо выполнить. Если возвращается NULL, то из пула готовых к выполнению задач выбирается новая.

Создание задачи должно осуществляться только с помощью оператора *new*, перегруженного в библиотеке ТВВ. Мы создадим главную задачу типа *QuickSortMergeTask* с помощью метода *new(task::allocate\_root())*, указав при этом начальные размер массива и количество потоков. Затем запустим на выполнение данную задачу, используя метод *static void task::spawn\_root\_and\_wait(task& root)*. В качестве подзадач меньшего размера у нас будут выступать подчиненные задачи по отношению к главной задаче. Для их создания мы используем метод *new(this.allocate\_child())* (эти задачи будут такого же типа, что и главная задача, но с уменьшенными вдвое размером массива и количеством потоков). Одну из подчиненных задач мы отправляем в пул готовых к выполнению, в то время как вторая подчиненная задача так же отправляется в пул готовых к выполнению, но ожидает завершения выполнения всех подчиненных задач. Всё, что описано в данном абзаце, происходит в том случае, когда количество потоков отлично от 1, в противном случае, поток сортирует часть массива, определенную логической задачей, используя быструю сортировку. Результатом работы алгоритма будет служить полностью отсортированный массив. На рис. 7 представлена программная реализация ТВВ-версии.

```

class QuickSortMergeTask : public tbb::task {
private:
    int* array;
    int* res;
    int size;
    int pNum;

public:
    QuickSortMergeTask(int* _array, int* _res, int _size, int _pNum) :
        array(_array), res(_res), size(_size), pNum(_pNum) { }
    tbb::task* execute() {
        if (pNum == 1) {
            quickSort(array, 0, size - 1);
        } else if (pNum > 1) {
            QuickSortMergeTask& a = *new(allocate_child())
                QuickSortMergeTask(array, res, size / 2, pNum / 2);
            QuickSortMergeTask& b = *new(allocate_child())
                QuickSortMergeTask(array + size / 2, res + size / 2, size - size / 2, pNum - pNum / 2);
            set_ref_count(3);
            spawn(b);
            spawn_and_wait_for_all(a);
            Merge_Two_Arrays_Into_Res(array, array + size / 2, size / 2, size - size / 2, res);
            std::copy(res, res + size, array);
        }
        return nullptr;
    }
};

void ParallelQuickSort(int* array, int* res, int size, int pNum) {
    QuickSortMergeTask& a = *new(tbb::task::allocate_root())
        QuickSortMergeTask(array, res, size, pNum);
    tbb::task::spawn_root_and_wait(a);
}

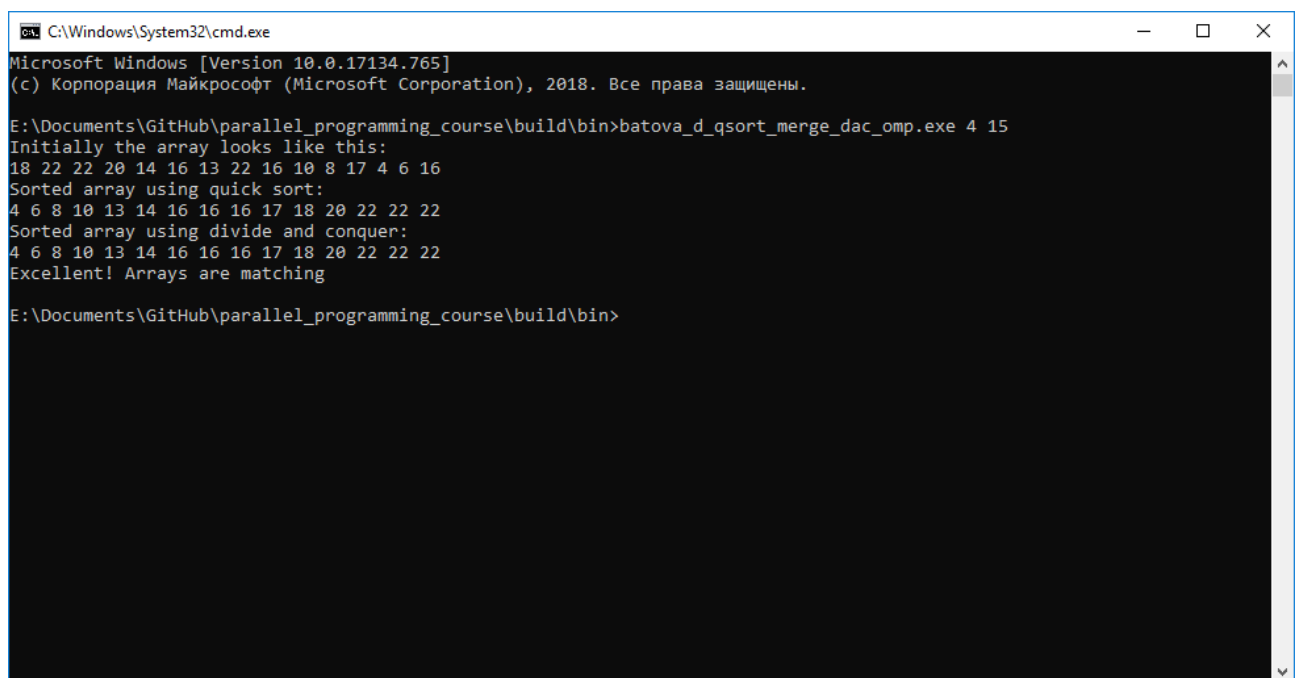
```

Рисунок 7. Реализация ТВВ-версии.

## Подтверждение корректности

Для подтверждения корректности в программе реализована функция сравнения массивов, полученных в результате последовательной и параллельной версий алгоритма. В случае, если мы не получаем одинаковые результаты, выводится сообщение: “Error! Arrays are not matching”. В случае, если массивы в обоих случаях совпадают, выводится сообщение: “Excellent! Arrays are matching”.

Чтобы продемонстрировать корректность, запустим параллельную программу, рассматривая случай, когда количество потоков равно 4, размер массива равен 15 (специально берём маленький размер, чтобы можно было вывести результаты). Результат работы программы представлен на рис. 8. В данном случае в консоли выводятся массивы, полученные в параллельной версии и в последовательной версии. Поскольку размер массива мал, можно сделать вывод о правильности работы программы, ведь массивы совпали.



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.17134.765]
(c) Корпорация Майкрософт (Microsoft Corporation), 2018. Все права защищены.

E:\Documents\GitHub\parallel_programming_course\build\bin>batova_d_qsort_merge_dac_omp.exe 4 15
Initially the array looks like this:
18 22 22 20 14 16 13 22 16 10 8 17 4 6 16
Sorted array using quick sort:
4 6 8 10 13 14 16 16 16 17 18 20 22 22 22
Sorted array using divide and conquer:
4 6 8 10 13 14 16 16 16 17 18 20 22 22 22
Excellent! Arrays are matching

E:\Documents\GitHub\parallel_programming_course\build\bin>
```

Рисунок 8. Вывод результатов

Однако, опыты проводятся, в основном, на достаточно больших размерах массива, что делает вывод результатов на консоль неразумным. Поэтому в таких случаях в программе поэлементно сравниваются массивы параллельной и

последовательной версии с дальнейшим выводом на консоли сообщения о совпадении или же несовпадении.

## Результаты экспериментов

Одной из задач данной лабораторной работы было сравнение времени работы параллельных версий сортировки одного и того же массива. В случае последовательной версии для сортировки массива используется алгоритм сортировки Хоара, сложность которого составляет  $O(n \log n)$  на случайных данных.

Были проведены следующие эксперименты, на основании которых нужно сделать вывод об эффективности параллельной программы. Все результаты занесены в таблицы, также построены графики на основании результатов. Время вычисляется в секундах.

Количество элементов	50000	10000	1000	500	1000
		00	000	0000	0000
Последовательная версия алгоритма	0.004 96127	0.010 0289	0.120 925	0.65 3711	1.35 916
OpenMP-версия алгоритма	0.003 42005	0.006 2487	0.071 5497	0.38 5798	0.81 6369
Ускорение	1.450 64	1.606 96	1.690 08	1.69 444	1.66 488

Таблица 1. Результаты для OpenMP-версии на 2-х потоках.

Количество элементов	50000	10000	1000	500	1000
		0	000	0000	0000
Последовательная версия алгоритма	0.004 92023	0.011 8051	0.122 435	0.69 6102	1.43 673
ТБВ-версия алгоритма	0.004 32426	0.007 46965	0.073 1044	0.40 9874	0.86 1435
Ускорение	1.137 82	1.580 41	1.647 39	1.69 833	1.66 783

Таблица 2. Результаты для ТБВ-версии на 2-х потоках.

<b>Количество элементов</b>	<b>5000 0</b>	<b>1000 00</b>	<b>1000 000</b>	<b>5000 000</b>	<b>1000 0000</b>
<b>Последовательная версия алгоритма</b>	0.004 8171	0.012 7196	0.116 709	0.65 1105	1.36 57
<b>OpenMP-версия алгоритма</b>	0.003 9526	0.007 3176	0.055 7183	0.28 6091	0.56 4163
<b>Ускорение</b>	1.218 74	1.738 21	2.094 63	2.27 586	2.42 076

Таблица 3. Результаты для OpenMP-версии на 4-х потоках.

<b>Количество элементов</b>	<b>50000</b>	<b>1000 00</b>	<b>1000 000</b>	<b>500 0000</b>	<b>1000 0000</b>
<b>Последовательная версия алгоритма</b>	0.005 2466	0.010 6509	0.119 859	0.67 5126	1.40 128
<b>ТБВ-версия алгоритма</b>	0.004 30981	0.006 6172	0.061 8095	0.30 8832	0.58 894
<b>Ускорение</b>	1.217 38	1.609 58	1.939 17	2.18 606	2.37 932

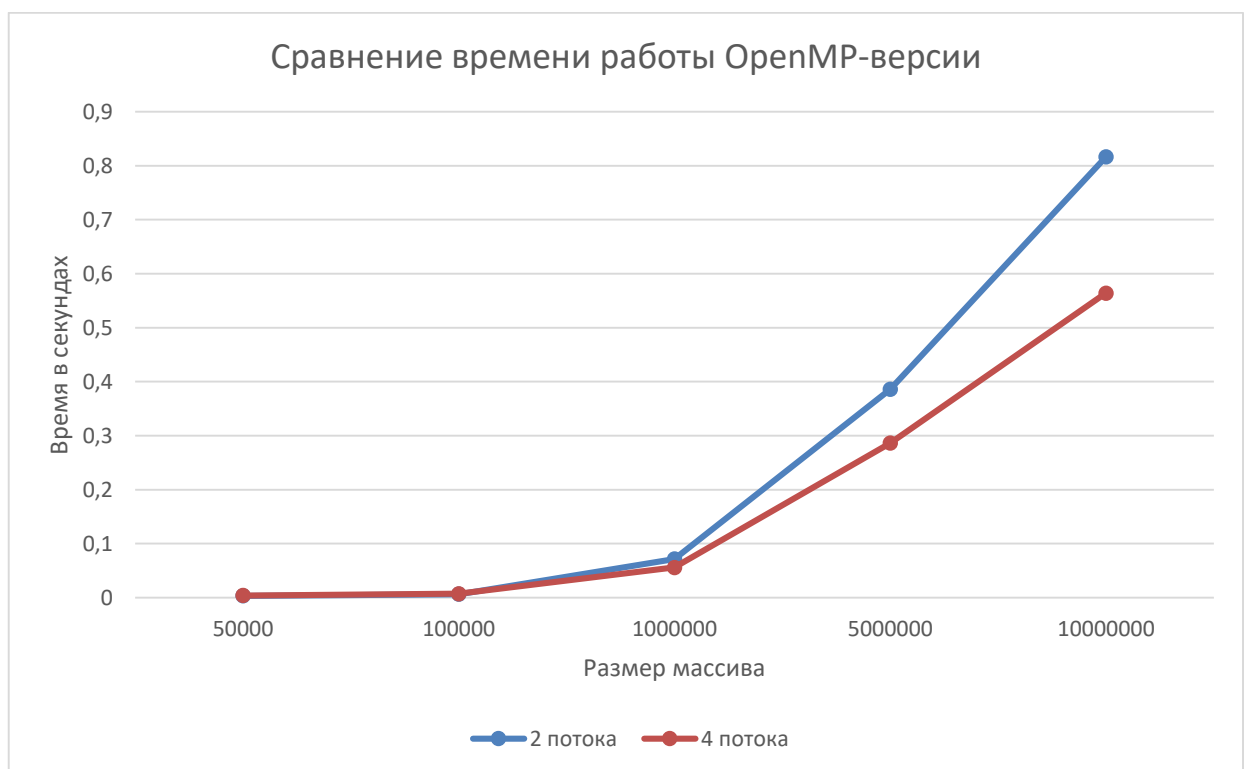
Таблица 4. Результаты для ТБВ-версии на 4-х потоках.

Характеристики персонального компьютера:

1. Процессор AMD A8-7410 APU with AMD Radeon R5 Graphics
2. Оперативная память 6 Гб 1600 МГц DDR3

## Выводы из результатов

На первом графике рассмотрим наглядно разницу во времени выполнения параллельной OpenMP-версии на 2-х и 4-х потоках соответственно. Мы не рассматриваем работу больше 4-х потоков, поскольку машина 4-х ядерная, а значит, при больших значениях программа не будет по-настоящему параллельно выполняться, т.е. будет наблюдаться простаивание потоков, что отрицательно скажется на эффективности программы. График строится на основе данных, занесенных в таблицы выше (см. таблицы 1 и 3).

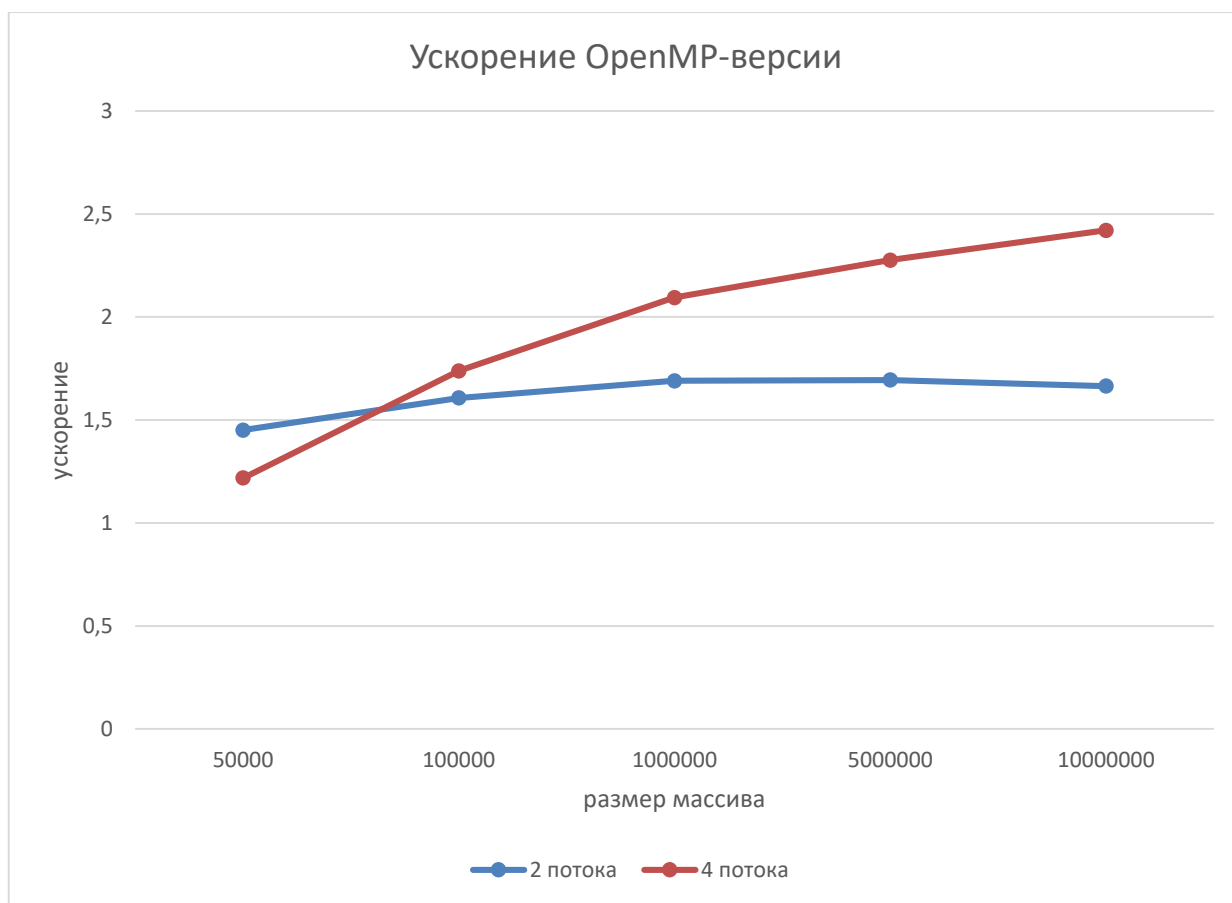


Как видно из диаграммы, на небольшом размере входных данных (размер массива) время работы OpenMP-версии на 2-х и 4-х потоках почти не отличается. Однако при увеличении размера массива задействование большего количества потоков уменьшает время выполнения программы. Этого и стоило ожидать, ведь увеличение размера массива приводит к повышению сложности алгоритма быстрой сортировки.

Чтобы сделать вывод об эффективности параллельной программы будем использовать такое понятие, как ускорение. Ускорение – отношение времени

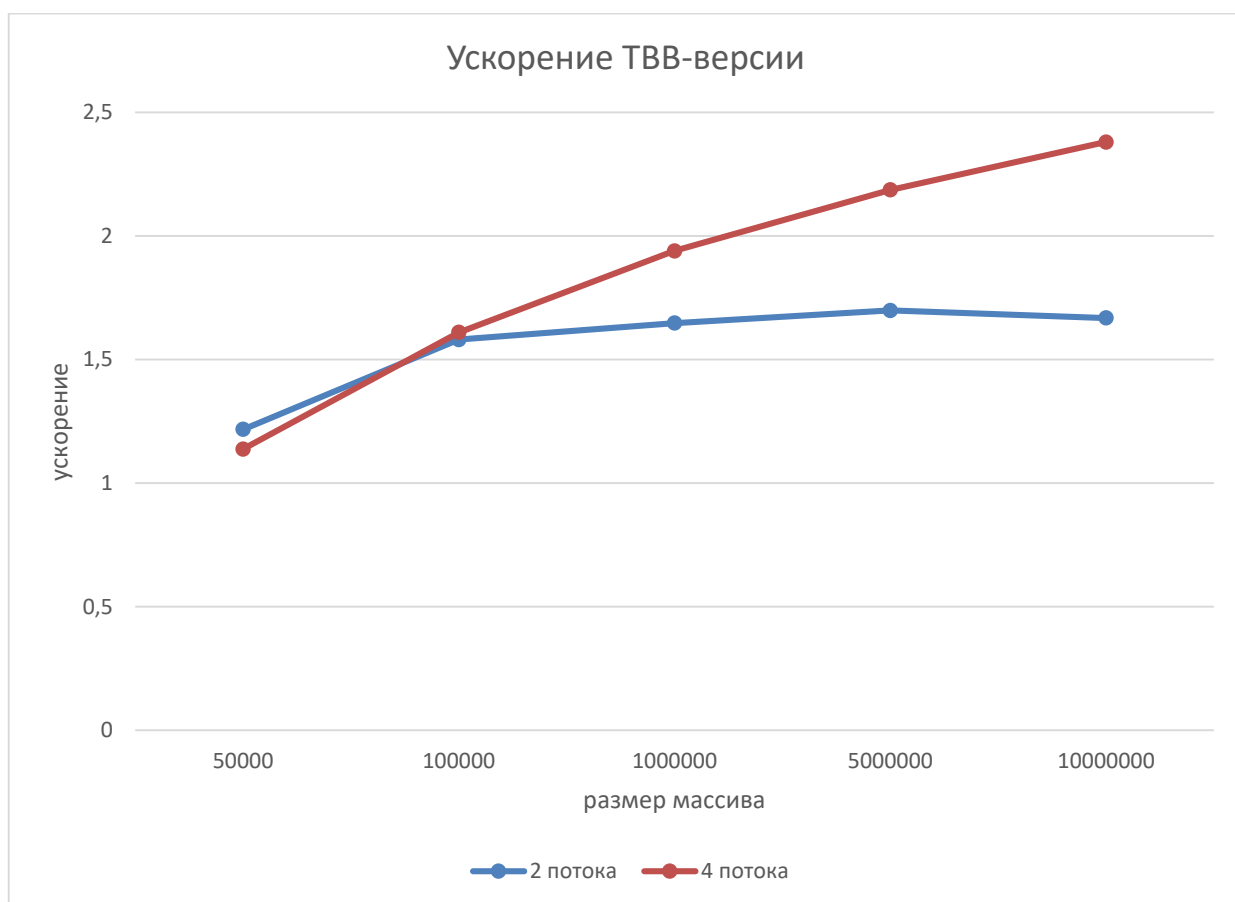
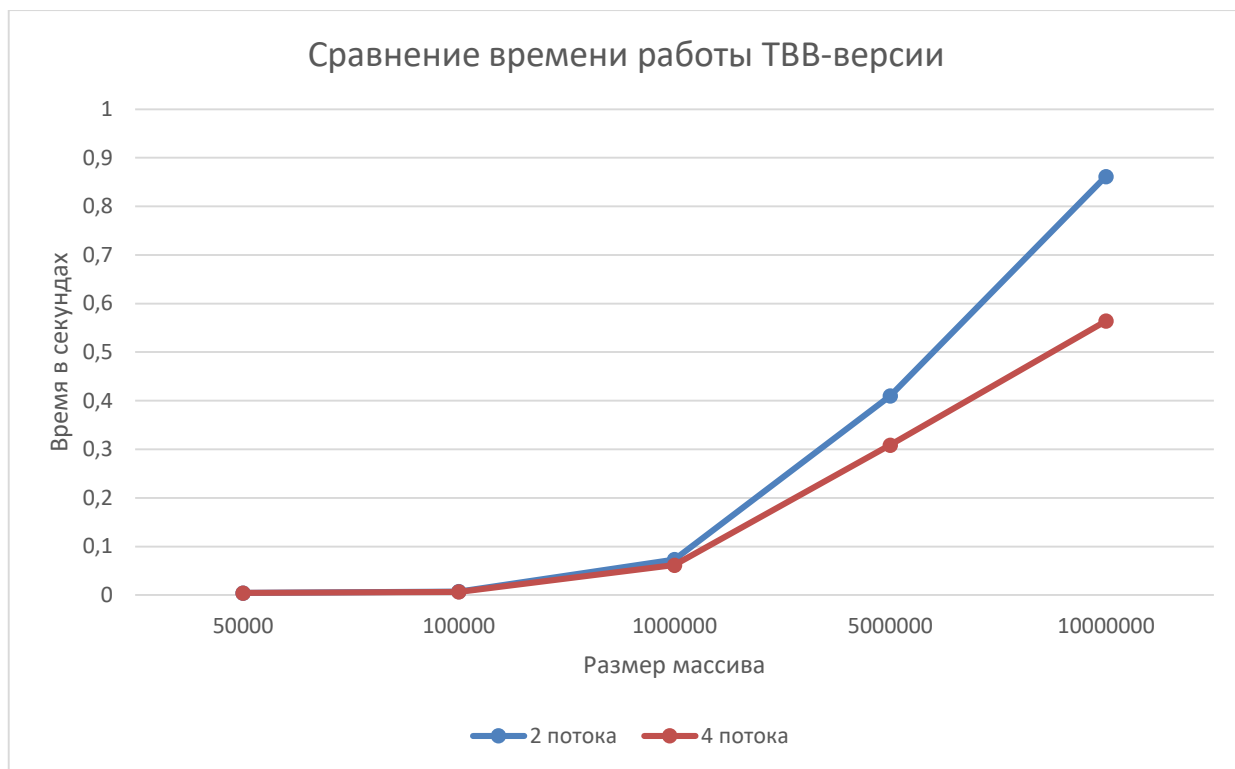


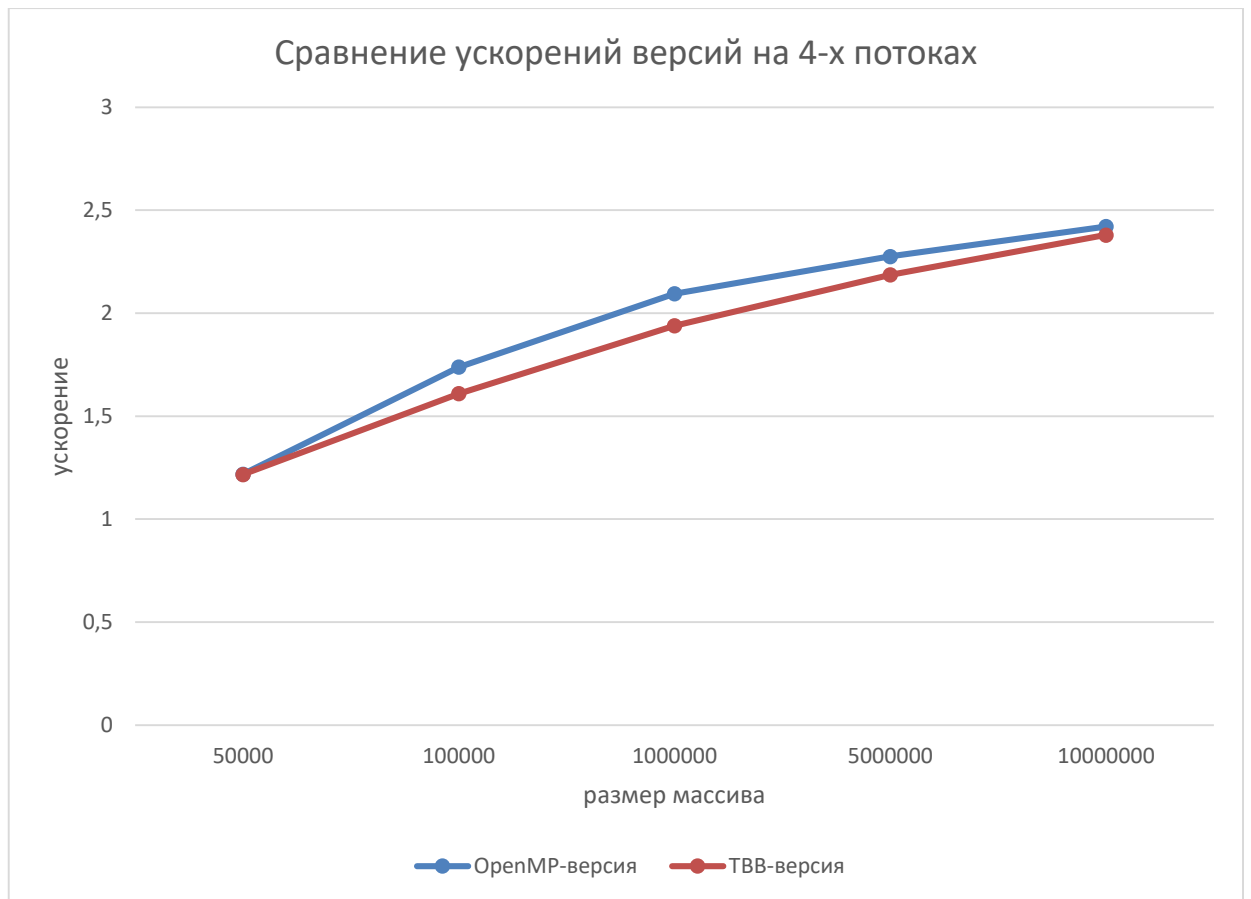
выполнения последовательного алгоритма к времени выполнения параллельного алгоритма. Построим график ускорения.



Как видно из графиков, на исходных данных небольшого размера параллельная версия алгоритма на 2-х потоках работает с большим ускорением, чем аналогичная версия на 4-х потоках. Это связано с тем, что больше времени уходит на взаимодействие между потоками, то есть на операции слияния, нежели на сортировку массивов. С ростом размера массива ситуация изменяется, поскольку теперь больше времени уходит на сортировку массивов, чем на взаимодействие потоков, таким образом, большее ускорение получается на 4-х потоках по сравнению с версией, отработанной на 2-х потоках.

Аналогичную ситуацию мы можем наблюдать, анализируя результаты экспериментов для ТВВ-версии параллельной сортировки.





Из последней диаграммы видно, что ускорения двух параллельных версий алгоритма приблизительно совпадают. Это можно объяснить одинаковыми алгоритмами распараллеливания.

## Заключение

Поставленная задача полностью выполнена. Были реализованы и протестированы последовательный и параллельные алгоритмы сортировки Хоара. В параллельном алгоритме использовалось слияние «Разделяй и властвуй». По данным экспериментов видно, что время работы сортировки в параллельной версии зависит от механизмов взаимодействия между потоками и от самой схемы распараллеливания т.е. имеет смысл применять параллельную версию данной сортировки при больших размерах массива. При маленьких же размерах массива эффективность последовательного алгоритма (сортировка Хоара) будет выше, поскольку накладные расходы на переключение потоков в таком случае не окупаются. Стоит отметить, что ускорение OpenMP-версии всё же немного больше по сравнению с ускорением TBV-версии. Этот факт можно объяснить тем, что использование объектно-ориентированного подхода в библиотеке TBV добавляет дополнительные расходы на создание классов, хранение экземпляров этих классов, выделение памяти под экземпляры, а это негативно сказывается на времени работы TBV-версии алгоритма.

## Литература

- [1]. Гегель, В.П. Параллельное программирование с использованием OpenMP // Нижегородский государственный университет им. Н.И. Лобачевского / Национальный исследовательский университет. – 2014. №1. – с. 44.
- [2]. Сысоев, А.В. Параллельное программирование // Параллельное программирование с использованием OpenMP // Нижегородский государственный университет им. Н.И. Лобачевского / Национальный исследовательский университет. – 2016. №2. – с. 27.
- [3]. Мееров И. Б. Инструменты параллельного программирования для систем с общей памятью. Библиотека Intel Threading Building Blocks – краткое описание / Сысоев А.В., Сиднев А.А. // Нижегородский государственный университет им. Н.И. Лобачевского. – 2009. – с. 172.
- [4]. Википедия - свободная энциклопедия. Алгоритм сортировки [Электронный ресурс] // [https://ru.wikipedia.org/wiki/Алгоритм\\_сортировки](https://ru.wikipedia.org/wiki/Алгоритм_сортировки)
- [5]. Быстрая сортировка [Электронный ресурс] // <http://kvodo.ru/quicksort.html>