МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования

**«Нижегородский государственный университет им. Н.И. Лобачевского»**

**Национальный исследовательский университет**

**Отчет по лабораторной работе**

**«Поразрядная сортировка для вещественных чисел (тип double) с четно-нечетным слиянием Бэтчера»**

**Выполнил:** студент группы 381506-1

Шутина Светлана Юрьевна

\_\_\_\_\_\_\_\_\_\_\_ Подпись

**Руководитель:**

Кустикова Валентина Дмитриевна

\_\_\_\_\_\_\_\_\_\_\_ Подпись

Нижний Новгород

2018

**Содержание**

1. Постановка задачи 3

2. Последовательная реализация 4

3. Схема распараллеливания 5

4. Описание программной реализации 6

5. Подтверждение корректности 7

6. Результаты экспериментов по оценке масштабируемости 8

7. Выводы 12

**Постановка задачи**

Требуется разработать программу, которая производят сортировку массива посредством заданного алгоритма. Для этого разработать последовательный и параллельный алгоритмы. Затем сравнить время выполнения на различном числе процессов (потоков) и определить ускорение.

Для распараллеливания использовать следующие инструменты:

* стандарт OpenMP;
* библиотеку Threading Building Blocks.

**Последовательная реализация**

В соответствии со стандартом IEEE754 для хранения числа типа double отводится три поля: знак, смещенный порядок и мантисса.

Знак числа кодируется одним битом: если знак числа равен нулю, то число положительное, иначе - отрицательное. Мантисса представляет собой последовательность бит, которая задает значащие цифры числа. Порядок задает положение десятичного разделителя в числе.

1. Создать count[] - массив счетчиков для i-го разряда. Source[] – исходный масив чисел.
2. counters[] - массив для счетчиков (i-й массив count[] расположен, начиная с адреса counters+256\*i).
3. Пройти по массиву и заполнить counters[]: увеличить количество байт с одинаковым значением.
4. Процедура сортировки заключается в осуществлении sizeof(double) проходов по направлению от младшего байта к старшему байту.
   1. В переменной count[0] хранится общее количество нулевых байтов на текущей сортируемой позиции, и если оно равно общему количеству чисел, то сортировка по этому разряду не нужна.
   2. Присвоить count[i] значение, равное сумме всех элементов до данного элемента.
   3. Произвести окончательную расстановку.

Для каждого числа source[i] мы знаем, сколько чисел меньше него - это значение хранится в count[source[i]]. Таким образом, нам известно окончательное место числа в упорядоченном массиве: если есть K чисел меньше данного, то оно должно стоять на позиции K+1.   
Осуществляем проход по массиву source слева направо, одновременно заполняя выходной массив dest[].

* 1. Необходимо откорректировать расположение чисел, т. к. отрицательные числа находятся в конце массива. Для этого модифицируем последний проход сортировки, работающий со старшими байтами и производящий окончательную расстановку.  
     Все, что нам необходимо - узнать номер первого отрицательного числа numNeg, и заполнять массив dest[] сначала числами после numNeg (отрицательными), а потом – остальными (положительными).

**Схема распараллеливания**

Четно-нечетное слияние Бэтчера заключается в том, что два упорядоченных массива, которые необходимо слить, разделяются на четные и нечетные элементы. Далее выполняется слияние четных и нечетных элементов массивов. Такое слияние может быть выполнено параллельно. Чтобы массив стал окончательно отсортированным, достаточно сравнить пары элементов, стоящие на нечетной и четной позициях. Первый и последний элементы массива проверять не надо, т.к. они являются минимальным и максимальным элементов массивов.

Четно-нечетное слияние Бэтчера 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 будут выполнять *n*/2 потоков и т.д. На последнем шаге два массива будут сливать 2 потока (рис. 1).

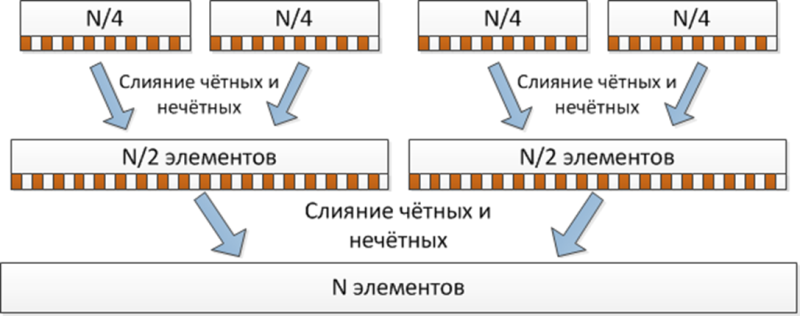


Рис.1.Четно-нечетное слияние Бэтчера.

**Описание программной реализации**

* Solver - решение задачи последовательным алгоритмом.
* Generator - программа для генерации набора тестовых данных и их сохранение в бинарные файлы.

Через командную строку пользователь вводит номер теста, который соответствует индексу в массиве. Значения массива – числа, равные количеству элементов массива. Далее этот массив заполняется случайными числами вещественными числами и записывается в бинарный файл.

* Checker - программа для проверки корректности параллельных версий.
* Solver\_OpenMP - решение задачи параллельным алгоритмом с помощью OpenMP.
* Solver\_TBB - решение задачи параллельным алгоритмом с помощью TBB.
* Deserializer - считывание данных из бинарного файла в текстовый.
* Serializer - считывание данных из текстового файла в бинарный.
* Набор тестов (папка tests) для проверки корректности параллельных версий.

**Подтверждение корректности**

Для подтверждения корректности нужно запустить программу Checker, в которой с помощью стандартного метода сортировки sort() получаем эталонный результат. Этот результат сравнивается с результатом, полученным с помощью программы Solver, Solver\_OpenMP или Solver\_TBB. Результат сравнения записывается в файл result.txt.

**Результаты экспериментов по оценке масштабируемости**

Тесты по оценке масштабируемости выполнялись на следующей инфраструктуре:

* Процессор Core i5-4210U CPU @ 1.70GHz 2.40 GHz
* ОЗУ 4GB
* 64-разрядная операционная система, процессор x64
* OS Windows 8.1

Ниже приведены значения времени исполнения (Таблица 1) и ускорения (Таблица 2) с использованием стандарта OpenMP, и времени исполнения (Таблица 3) и ускорения (Таблица 4) с использованием библиотеки TBB.

Таблица 1. Время выполнения (нс)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Количество**  **процессов**  **Размер массива** | **1** | **2** | **4** | **16** |
| **100000** | 312352,18 | 388501,84 | 245950,81 | 248770,76 |
| **1000000** | 1254759,79 | 1587539,54 | 997368,48 | 1004399,12 |
| **5000000** | 5481464,39 | 3622641,64 | 3447830,65 | 1015989,91 |
| **10000000** | 10699929,9 | 5421297,27 | 4838240,16 | 2616756,26 |

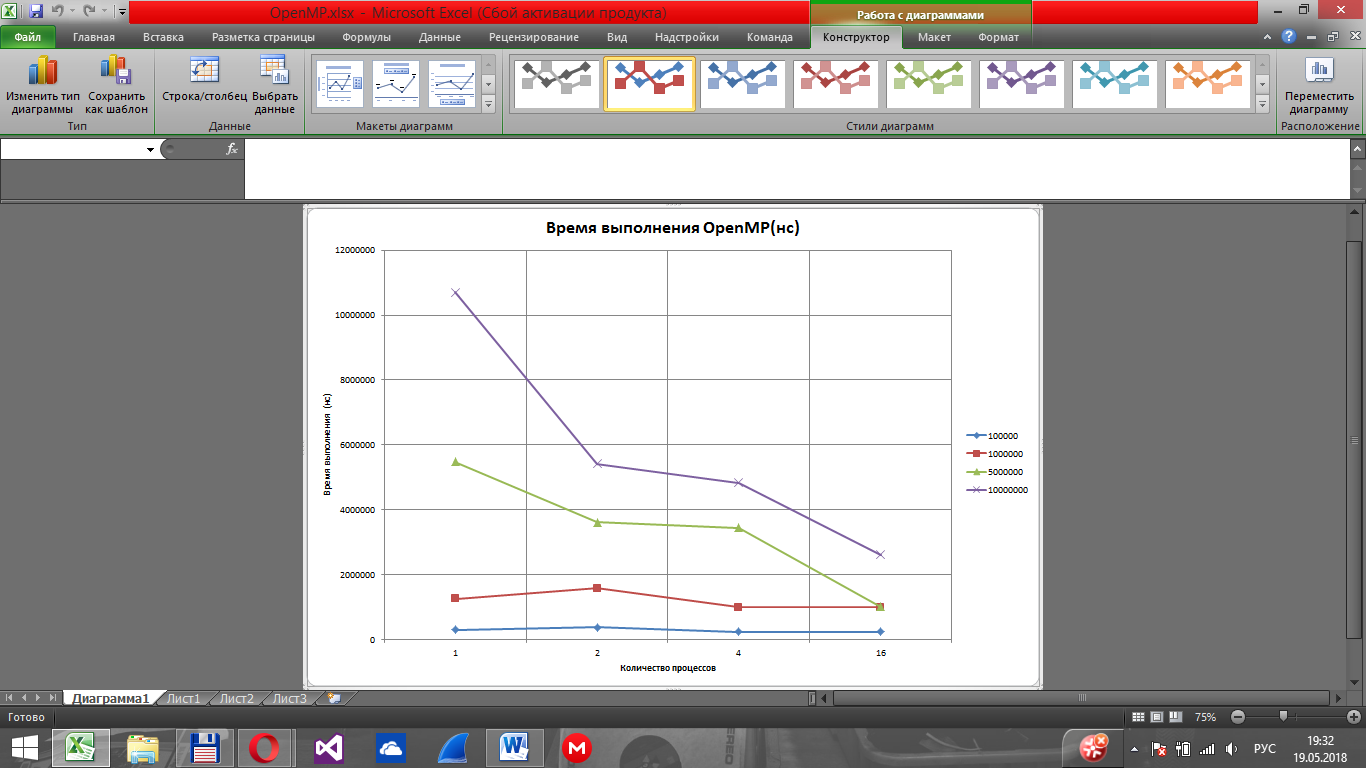


Рис.2. График зависимости времени выполнения от количества процессов

Таблица 2. Ускорение параллельной реализации по сравнению с однопроцессной

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Количество**  **процессов**  **Размер массива** | **1** | **2** | **4** | **16** |
| **100000** | 1 | 0,703 | 1,269 | 1,255 |
| **1000000** | 1 | 0,79 | 1,258 | 1,249 |
| **5000000** | 1 | 1,5131 | 1,5898 | 5,3951 |
| **10000000** | 1 | 1,9736 | 2,2115 | 4,089 |

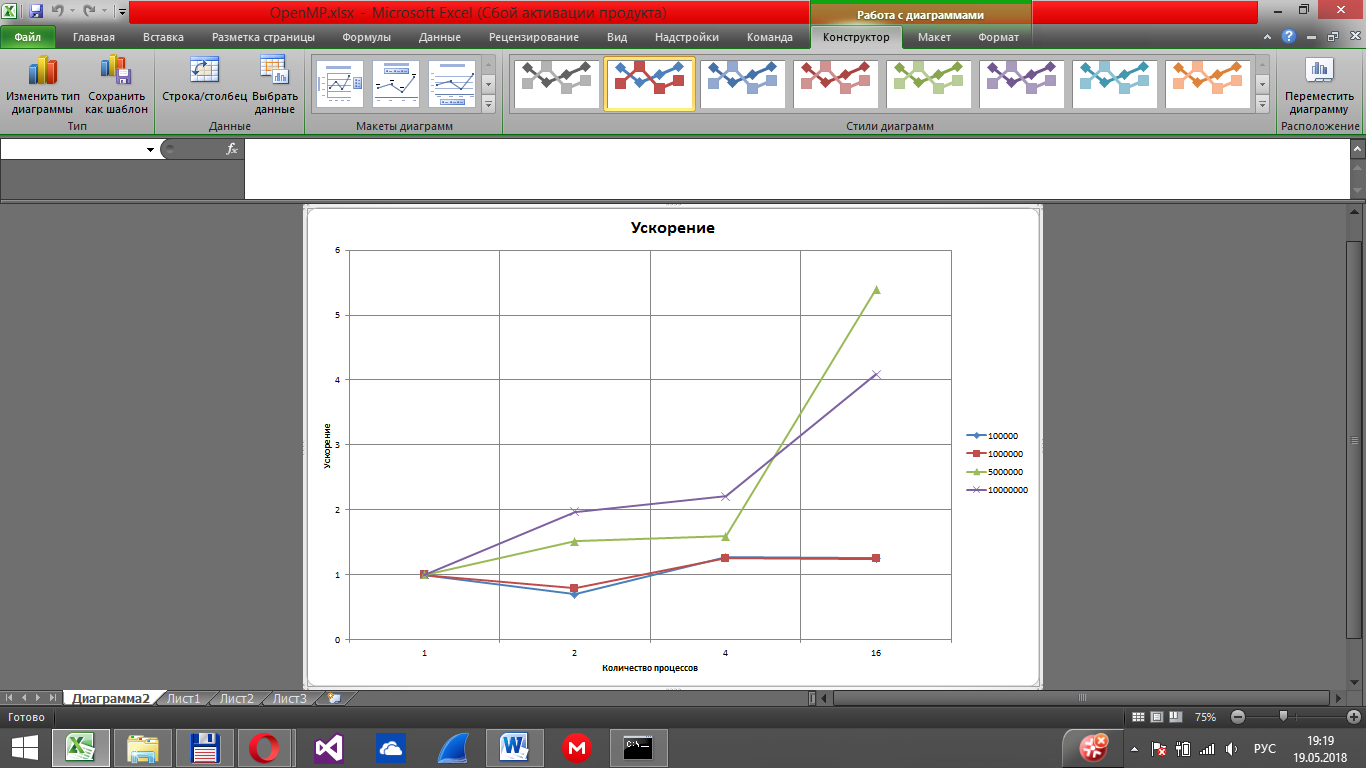


Рис.3. График зависимости ускорения от количества процессов

Таблица 3. Время выполнения (нс)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Количество**  **процессов**  **Размер массива** | **1** | **2** | **4** | **16** |
| **100000** | 312352,18 | 284222,51 | 325895,93 | 298285,8 |
| **1000000** | 1254759,79 | 1547709,18 | 1846898,06 | 1023049,15 |
| **5000000** | 5481464,39 | 1812206,7 | 1261389,18 | 1125567,44 |
| **10000000** | 10699929,9 | 3791224,27 | 3383632,47 | 2263019,4 |

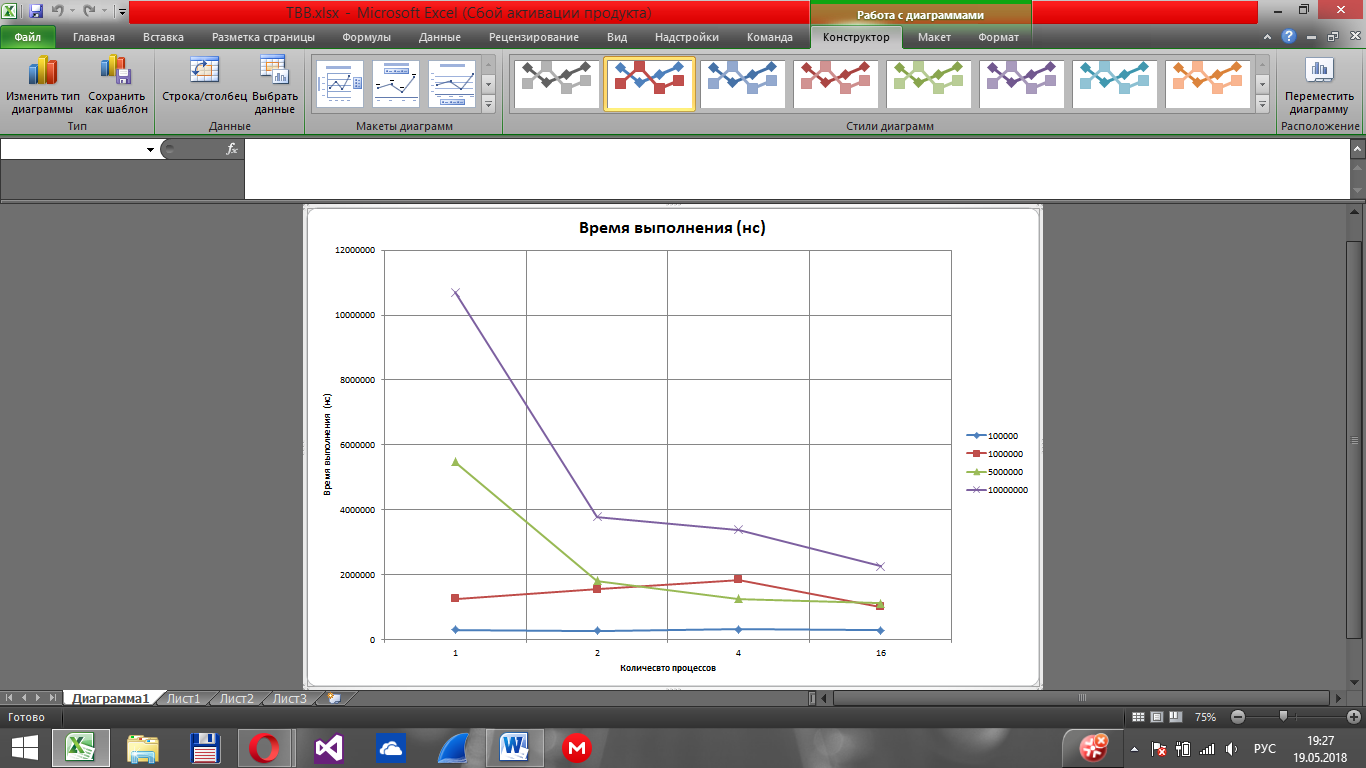


Рис.4. График зависимости времени выполнения от количества процессов

Таблица 4. Ускорение параллельной реализации по сравнению с однопроцессной

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Количество**  **процессов**  **Размер массива** | **1** | **2** | **4** | **16** |
| **100000** | 1 | 1,09 | 0,958 | 1,047 |
| **1000000** | 1 | 0,81 | 0,67 | 1,22 |
| **5000000** | 1 | 3,024 | 4,345 | 4,869 |
| **10000000** | 1 | 2,822 | 3,162 | 4,72 |

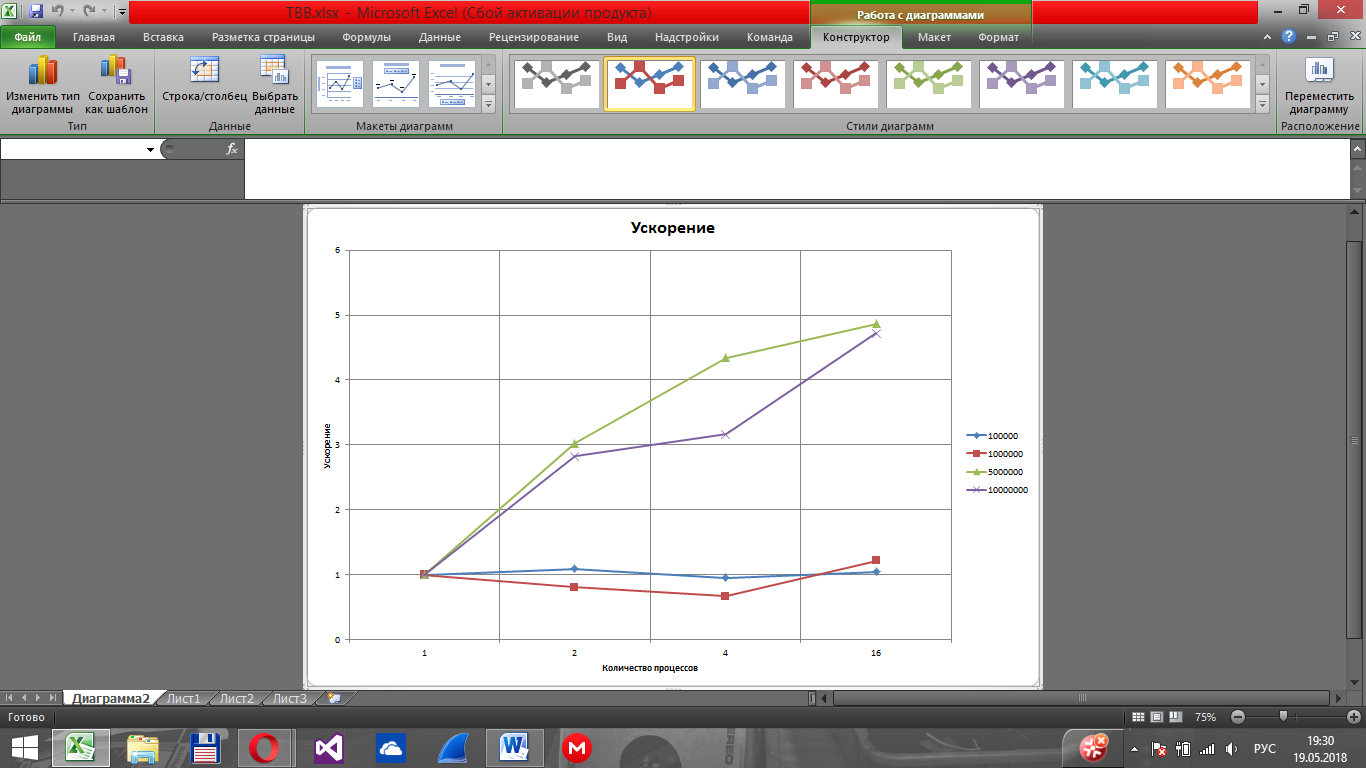


Рис.5. График зависимости ускорения от количества процессов

**Выводы**

В результате выполнения лабораторной работы написали последовательную версию программы и две параллельные версии (OpenMP, TBB).

При проведении экспериментов по оценке масштабируемости увидели, что при малых значениях размера массива последовательная версия работает быстрее, чем параллельная. Но при увеличении размера массива появляется ускорение. При увеличении количества процессов время выполнения также уменьшается. Также видно, что время выполнения программы, написанной с помощью OpenMP, примерно совпадает со временем работы программы, написанной с помощью TBB.