

Санкт-Петербургский политехнический университет Петра Великого  
Институт компьютерных наук и технологий  
Кафедра компьютерных систем и программных технологий

# Параллельные вычисление

Отчёт по теме :

Создание программы сортировки последовательности чисел при помощи  
MPI

**Работу**

**выполнил:**

Ковальков А.В.

Группа:

3540901/91501

**Преподаватель:**

Стручков И.В.

Санкт-Петербург  
2020

# Содержание

<b>1. Цель работы</b>	<b>3</b>
<b>2. Программа работы</b>	<b>3</b>
<b>3. Сведения о системе</b>	<b>3</b>
<b>4. Теоретическая информация</b>	<b>4</b>
<b>5. Ход выполнения работы</b>	<b>4</b>
5.1. Алгоритм решения . . . . .	4
5.1.1. Общие сведения и выбор алгоритма . . . . .	4
5.1.2. Unit-тестирование . . . . .	6
5.1.3. Вспомогательные методы . . . . .	7
5.1.4. Последовательная реализация . . . . .	8
5.1.5. Параллельная реализация . . . . .	9
5.2. Эксперименты . . . . .	10
5.2.1. Вариативность скорости работы от величины исходных данных . . .	11
5.3. Оценка производительности работы параллельной программы в зависимо- сти от количества потоков . . . . .	15
5.3.1. Оценка результатов прогонов при помощи статистических показателей	16
<b>6. Заключение</b>	<b>17</b>

## 1. Цель работы

Задание. Вариант 3. MPI: Выполнить сортировку последовательности чисел.

В данное задание входит разбор алгоритма выполнения работы, проектирование распределенного и последовательного приложения и проведение тестирования скорости их работы различными методами.

## 2. Программа работы

- Для алгоритма из полученного задания написать последовательную программу на языке C или C++, реализующую этот алгоритм.
- Для созданной последовательной программы необходимо написать 3-5 тестов, которые покрывают основные варианты функционирования программы. Для создания тестов можно воспользоваться механизмом Unit-тестов среды NetBeans, или описать входные тестовые данные в файлах. При использовании NetBeans необходимо в свойствах проекта установить ключ компилятора -pthread.
- Проанализировать полученный алгоритм, выделить части, которые могут быть распараллелены, разработать структуру параллельной программы. Определить количество используемых потоков, а также правила и используемые объекты синхронизации.
- Согласовать разработанную структуру и детали реализации параллельной программы с преподавателем.
- Написать код параллельной программы и проверить ее корректность на созданном ранее наборе тестов. При необходимости найти и исправить ошибки.
- Провести эксперименты для оценки времени выполнения последовательной и параллельной программ. Проанализировать полученные результаты.
- Сделать общие выводы по результатам проделанной работы: Различия между способами проектирования последовательной и параллельной реализаций алгоритма, Возможные способы выделения параллельно выполняющихся частей, Возможные правила синхронизации потоков, Сравнение времени выполнения последовательной и параллельной программ, Принципиальные ограничения повышения эффективности параллельной реализации по сравнению с последовательной.

## 3. Сведения о системе

- intel core i7-4700HQ 2.4 GHz, 4 ядра, 8 потоков
- RAM 16gb DDR3
- Java 1.8\_255 Adopted Open JDK

## 4. Теоретическая информация

Message Passing Interface (MPI, интерфейс передачи сообщений) — программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. Разработан Уильямом Гроуппом, Эвином Ласком (англ.) и другими.

MPI является наиболее распространённым стандартом интерфейса обмена данными в параллельном программировании, существуют его реализации для большого числа компьютерных платформ. Используется при разработке программ для кластеров и суперкомпьютеров. Основным средством коммуникации между процессами в MPI является передача сообщений друг другу.

Стандартизацией MPI занимается MPI Forum. В стандарте MPI описан интерфейс передачи сообщений, который должен поддерживаться как на платформе, так и в приложениях пользователя. В настоящее время существует большое количество бесплатных и коммерческих реализаций MPI. Существуют реализации для языков Фортран 77/90, Java, Си и C++.

В первую очередь MPI ориентирован на системы с распределенной памятью, то есть когда затраты на передачу данных велики, в то время как OpenMP ориентирован на системы с общей памятью (многоядерные с общим кешем). Обе технологии могут использоваться совместно, чтобы оптимально использовать в кластере многоядерные системы.

## 5. Ход выполнения работы

### 5.1. Алгоритм решения

#### 5.1.1. Общие сведения и выбор алгоритма

Алгоритмов сортировки много, и изобретать свой собственный не имеет смысла, поэтому, в первую очередь необходимо было выбрать наиболее оптимальный алгоритм сортировки.

Оптимальность заключается в двух параметрах, первый - скорость работы. Оптимальным в данном случае будет выбор алгоритма работающего за  $O(n \log n)$ , к таким относятся быстрая сортировка, сортировка слиянием и др. Во вторых, данный алгоритм должен быть способен к работе с несколькими потоками. Например алгоритм Шелла и некоторые другие алгоритмы реализовать значительно трудней при условии работы нескольких потоков, а некоторые и вовсе работают непроизводительно.

Таким образом, выбор пал на сортировку слиянием. Так как смысл данного алгоритма разбивать массивы на подмассивы, сортировать их и соединять. То если мы изначально разделим наш массив на подмассивы соизмеримо количеству потоков, а потом соединим их. Это удобно и кажется в контексте данной задачи неплохим решением.

Сама сортировка, как уже было сказано состоит из двух ключевых этапов (методов), этап рекурсивного разбиения на подмассивы, и этап их перестановки(сортировки) и слияния. Наиболее оптимальным является рекурсивная реализация, по этому и будет использована она. Но немного адаптированная под условия задания.

Рекурсивные методы наиболее удобно создавать при помощи возврата непосредственно рекурсивного вызова, но учитывая то, что мы должны вернуть массив чисел, а не просто отсортировав вывести, то нам нужно реализовать рекурсию немного по другому, например в полях переменных ( подмассивов левой и правой части базового массива).

Для реализации был выбран язык Java. Версии 1.8, и библиотека соблюдающая спецификацию MPI - MPJexpress.

Общий код алгоритма продемонстрирован на листинге ниже:

Листинг 1: MergeSort.java

```
1 package Sorting;
2
3 public class MergeSort {
4
5
6     // merge sorting
7     public static int [] sortArray(int [] arrayA){
8         if (arrayA == null || arrayA.length < 2) {
9             return arrayA;
10        }
11
12        int [] arrayB = new int[arrayA.length / 2];
13        System.arraycopy(arrayA, 0, arrayB, 0, arrayA.length / 2);
14
15        int [] arrayC = new int[arrayA.length - arrayA.length / 2];
16        System.arraycopy(arrayA, arrayA.length / 2, arrayC, 0, arrayA.length -
17        ↪ arrayA.length / 2);
18
19        arrayB = sortArray(arrayB);
20        arrayC = sortArray(arrayC);
21
22        return mergeArray(arrayB, arrayC);
23    }
24
25    public static int [] mergeArray(int [] a1, int [] a2) {
26        int [] b = new int[a1.length + a2.length];
27        int positionA1 = 0;
28        int positionA2 = 0;
29
30        for(int i = 0; i < b.length; i++) {
31            if(positionA1 == a1.length){
32                b[i] = a2[positionA2];
33                positionA2++;
34            } else if(positionA2 == a2.length){
35                b[i] = a1[positionA1];
36                positionA1++;
37            } else if(a1[positionA1] < a2[positionA2]){
38                b[i] = a1[positionA1];
39                positionA1++;
40            } else {
41                b[i] = a2[positionA2];
42                positionA2++;
43            }
44        }
45        return b;
46    }
47
48    // for final merge
49    public static int [][] splitting(int [][] subArr){
50        if (subArr.length == 1) {
51            return subArr;
52        }
53
54        int [][] arr = new int[subArr.length/2][subArr[0].length * 2];
55
56        for (int i = 0, j = 0; i < subArr.length; i+=2, j++) {
57            arr[j] = mergeArray(subArr[i], subArr[i + 1]);
58        }
59    }
60 }
```

```

57     }
58
59     return splitting(arr);
60 }
61 }

```

Как мы можем увидеть из листинга 1. В данном классе всего 3 метода. Первый метод - `sortArray`. Принимает простой массив целых чисел, разбивает его на равные части и проверяет массив на пустоту (`null`) и на длину в 1 элемент, так как оба варианта таких массивов являются отсортированными. Также происходит разделение на два подмассива - левого и правого, и рекурсивно вызывается данный метод для каждой из половин. При этом возвращает метод второй метод `mergeArray`, который как раз таки, поочередно сравнивает элементы поступивших к нему массивов, укладывает это в один массив по возрастанию значений и возвращает обратно в метод `sortArray`.

Последний метод - `splitting`, изначально задумывался для финальной сборки и упорядочивания массивов. При помощи метода `mergeArray`. Но так как он также рекурсивный и мало чем отличен от метода `sorting`. К тому же в `java` не лучший язык для хвостовых рекурсий на данный момент, и данный метод был бы оптимальным в случае отсутствия рекурсии, по этому данный метод на момент написания отчёта не используется, но является альтернативным вариантом финальной сборки и сортировки данных для параллельного выполнения.

### 5.1.2. Unit-тестирование

Все основные случаи для сортировки были протестированы при помощи фреймворка `JUnit` версии 4.12.

Как видно из листинга ниже. Были протестированы отсортированный и обратно отсортированный массивы, пустой, с одним элементом, с дублированием, с соседними элементами и при этом пограничным количеством элементов с нижней стороны.

Всего 7 тестов, тестирующих 9 различных ситуаций при сортировке. Листинг ниже демонстрирует тесты для заданного алгоритма.

Листинг 2: `MergeSortTests.java`

```

1 package tests;
2
3 import Sorting.MergeSort;
4 import org.junit.Assert;
5 import org.junit.Test;
6
7 public class MergeSortTests {
8
9     private static final int[] ORDERED =
10     ↪ {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20}; //
        упорядоченный
11     private static final int[] REVERS_ORDERED =
12     ↪ {20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1}; //
        обратно
13     private static final int[] ALL_THE_SAME =
14     ↪ {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}; // Все числа
        одинаковы
15     private static final int[] TWO_NEAR =
16     ↪ {2,2,2,2,2,2,2,2,2,2,2,2,2,1,1,1,1,1,1,1}; // Два близких
        числа
17     private static final int[] TWO_NEAR_2 = {1,2};
18     ↪ // Два близких
        числа
19     private static final int[] TWO_NEAR_3 = {2,1};
20     ↪ // Два близких
        числа

```

```

15     private static final int[] ONE_ELEM = {1};
    ↪                                     // Один
    элемент
16     private static final int[] EMPTY_ELEM = {};
    ↪                                     // Пустой
    элемент
17     private static final int[] NULL_ELEM = null;
    ↪                                     // Пустой
    элемент
18
19
20     @Test
21     public void empty() {
22         Assert.assertEquals(new int[] {}, EMPTY_ELEM);
23     }
24
25     @Test
26     public void nullArr() {
27         Assert.assertNull(MergeSort.sortArray(NULL_ELEM));
28     }
29     @Test
30     public void oneElement() {
31         Assert.assertEquals(new int[] {1}, MergeSort.sortArray(ONE_ELEM));
32     }
33     @Test
34     public void nears() {
35         Assert.assertEquals(new int
    ↪ [|{1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,2}, MergeSort.sortArray(
    ↪ TWO_NEAR));
36         Assert.assertEquals(new int[] {1,2}, MergeSort.sortArray(TWO_NEAR_2)
    ↪ );
37         Assert.assertEquals(new int[] {1,2}, MergeSort.sortArray(TWO_NEAR_3)
    ↪ );
38     }
39
40     @Test
41     public void same() {
42         Assert.assertEquals(new int
    ↪ [|{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}, MergeSort.sortArray(
    ↪ ALL_THE_SAME));
43     }
44
45     @Test
46     public void ordered() {
47         MergeSort.sortArray(ORDERED);
48         Assert.assertEquals(ORDERED, MergeSort.sortArray(ORDERED));
49     }
50     @Test
51     public void reversOrdered() {
52         Assert.assertEquals(ORDERED, MergeSort.sortArray(REVERS_ORDERED));
53     }
54 }

```

### 5.1.3. Вспомогательные методы

Для того, чтобы можно было адекватно оценить производительность данного алгоритма, было также принято решение написать класс, способный генерировать массив заданной длины.

Массив собирается на основе случайных чисел, в методе generate. Также существует несколько методов для работы с данным массивом. Во первых метод toSimpleArray, кото-

рый переводит массив чисел из листа (коллекции) в массив фиксированного размера.

А также методы предоставляющий массив чисел и метод разбивающий массив на подмассивы равно длинны. Подобный бинарный массив в начале использовался в паре с методом `splitting` листинга 1. Но так как тот метод сейчас не используется, то данный метод оставлен для просто для демонстрации, альтернативного пути исполнения.

Листинг 3: ListGenerator.java

```
1 package helpers;
2
3 import java.util.List;
4 import java.util.Random;
5 import java.util.stream.Collectors;
6 import java.util.stream.Stream;
7
8 public class ListGenerator {
9     private int[] ints;
10
11     public ListGenerator(int c) {
12         ints = toSimpleArray(generate(c));
13     }
14
15     // generate list of random unsigned int from 0 to 100
16     private List<Integer> generate(int c){
17         return Stream.generate(() -> new Random(47).nextInt(100)).limit(c).
18         ↪ collect(Collectors.toList());
19     }
20
21     // transform list to int[]
22     private int[] toSimpleArray(List<Integer> list){
23         ints = new int[list.size()];
24         for (int i = 0; i < list.size(); i++) {
25             ints[i] = list.get(i);
26         }
27         return ints;
28     }
29
30     public int[] getInts() {
31         return ints;
32     }
33
34     // divide one array into equal parts
35     public static int[][] getRankedInts(int worldSize, int[] ints){
36         int rankSize = ints.length/worldSize;
37         int[][] ints1 = new int[worldSize][rankSize];
38         for (int i = 0, x = 0; i < worldSize; i++, x+=rankSize) {
39             if (rankSize >= 0) System.arraycopy(ints, x, ints1[i], 0, rankSize);
40         }
41         return ints1;
42     }
```

#### 5.1.4. Последовательная реализация

Последовательная генерация выполняется крайне просто. В начале и в конце метода снимаются показатели о текущем времени(8,15), и в середине создаётся массив заданной длины (10) и вызывается метод его сортировки (11).

Листинг 4: Serial.java



```

1 import Sorting.MergeSort;
2 import helpers.ListGenerator;
3
4 import java.util.Arrays;
5
6 public class Serial {
7     public static void main(String[] args) {
8         long startTime = System.currentTimeMillis();
9
10        int[] shuf = new ListGenerator(40_000_000).getInts();
11        int[] sort = MergeSort.sortArray(shuf);
12        System.out.printf("Shuffled: %s\n", Arrays.toString(shuf));
13        System.out.printf("Sorted: %s\n", Arrays.toString(sort));
14
15        long finishTime = System.currentTimeMillis();
16        System.out.printf("Time is: %d ms", finishTime - startTime);
17    }
18 }

```

### 5.1.5. Параллельная реализация

Параллельная реализация намного более многословна. Так как помимо описания и реализации специфичных для MPI методов, необходимо было позаботиться о финальной сортировке данных.

Код параллельной реализации продемонстрирован ниже:

Листинг 5: Parallel.java

```

1 import Sorting.MergeSort;
2 import helpers.ListGenerator;
3 import mpi.MPI;
4 import java.util.Arrays;
5 public class Parallel {
6     public static void main(String[] args) {
7         long timeStart = System.currentTimeMillis();
8
9         // Init MPI
10        MPI.Init(args);
11        // rank - number of process
12        // size - quantities of process
13        int rank = MPI.COMM_WORLD.Rank(), size = MPI.COMM_WORLD.Size(), root =
14        ↪ 0;
15        // length - length of array which we wanna sort
16        // unitSize - length of one process
17        int length = 40_000_000, unitSize = length / size;
18        // send - what we send from root to all
19        // recv - what we receive from root
20        // newrecv - what we receive in root from all
21        int[] sendBuffer = new int[length], recvBuffer = new int[unitSize],
22        ↪ newRecvBuffer = new int[length];
23
24        if(rank == root){
25            // generate random array
26            sendBuffer = new ListGenerator(length).getInts();
27            System.out.printf("Shuffled: %s\n", Arrays.toString(sendBuffer));
28        }
29        // sending parts of sendBuffer to all process

```

```

30      MPI.COMM_WORLD.Scatter(sendBuffer, 0, unitSize, MPI.INT, recvBuffer, 0,
    ↪ unitSize, MPI.INT, root);
31
32      // sort in each process
33      recvBuffer = MergeSort.sortArray(recvBuffer);
34
35      // assembly all parts to one array
36      MPI.COMM_WORLD.Gather(recvBuffer, 0, unitSize, MPI.INT, newRecvBuffer, 0,
    ↪ unitSize, MPI.INT, root);
37
38      if (rank == root) {
39          // finally merge
40      // int[][] result = Sorting.MergeSort.splitting(helpers.ListGenerator.getRankedInts(size, newRecvBuffer));
41      // System.out.printf("Sorted :System.out.printf("Sorted :
42          // check time
43          long timeStop = System.currentTimeMillis();
44          System.out.printf("Time_is:_%d_ms", timeStop - timeStart);
45      }
46      // finalization for garbage collector
47      MPI.Finalize();
48  }
49 }

```

Данная программа также как и последовательная начинается и заканчивается показанием текущего времени, для оценки скорости работы программы.

Далее происходит инициализация MPI при помощи метода Init, затем создание переменных необходимых для управления потоками, size - общее количество потоков, rank - номер текущего процесса от 0 до n-1, а также мы сразу объявляем о нашем корневом процессе (root), по умолчанию корневым процессом мы считаем процесс с rank = 0.

Затем мы инициализируем переменные, говорящие о объемах массивов данных. length - говорит об общей длине сгенерированных данных, а unitSize - говорит о том, сколько элементов будет включать в себя подмассив у каждого потока. Очевидно, что данный подход не защищает от нечётного числа потоков, но в пользу упрощения логики было принято решение оставить данное решение, так как планировалось использовать четное количество потоков.

Последним этапом создания переменных является создание буферов приема и передачи данных. sendBuffer - буфер, который отправит корневой процесс всем остальным, recvBuffer - то что примут процессы, newRecvBuffer - то что примет от потоков обратно корневой процесс.

Следующим этапом, на строках 23-27 мы создаем массив нужно нам длинны в корневом процессе.

Затем при помощи Scatter (основной метод коллективного обмена данными в mpi), мы делим исходный массив на равные части и раздаём его части нашим потокам.

Далее производим сортировку во всех потоках сразу. А затем используем метод Gather, который обратен методу Scatter (передает данные от всех процессов к корневому соединяя их в один массив).

Затем, мы производим финальную сортировку массива в корневом процессе. И закрываем MPI при помощи Finalize, вызывая сборщик мусора.

## 5.2. Эксперименты

Все эксперименты проводились с участием 4 потоков.

### 5.2.1. Вариативность скорости работы от величины исходных данных

В ходе данного эксперимента, необходимо было выяснить, с какой скоростью работает последовательный и параллельный алгоритмы. Как сильно отличается скорость работы между собой при различных размерах входных данных.

Для замеров скорости работы использовался метод `System.currentTimeMillis`, запускаемый при старте и финише программы. Разница старта и финиша и есть - скорость работы в миллисекундах (далее мс).

Для сравнений скорости работы был определен следующие размеры массивов: 4 тысячи, 40 тысяч, 400 тысяч, 4 миллиона, 20 миллионов, 32 миллиона, 36 миллиона, 40 миллионов, 50 миллионов, 58 и 60. Для каждого размера массива замеры проводились 50 раз с подсчётом среднего значения (средне арифметическое). Размеры массива были выбраны случайно и не на что не опирались вплоть до 50 миллионов (далее, шел постепенный поиск предела по памяти для JVM).

Несмотря на то, что массив у нас генерируется случайно, любое случайное значение находится в пределах от 1 до 99, а также большое количество замеров и высокая длина массивов, все эти факторы стабилизируют время, которое тратит программа при запуске на генерацию массивов.

Таб.1. Время выполнение(мс) в зависимости от размера массива (тысячи)

Размер	Параллельная	Последовательная
4	645	35
40	687	77
400	933	258
4000	2257	1858
20000	8159	10175
32000	13394	15410
36000	15120	16649
40000	16814	18582
50000	24636	26821
58000	33186	30477
60000	0	30598

Как можно увидеть из таблицы выше, вплоть до 400 тысяч элементов первенство было у последовательной реализации. А на ранних этапах существенное. Связано это в первую очередь с тем, что для выделения процессов, выделения под них память и манипуляция ей - всё это ресурсоёмкие задачи, которые превосходят затраты на сортировку массива малого размера.

Далее, при увеличении размеров, параллельная реализация становится быстрее, но как только она доходит до 60 миллионов, несмотря на то что при всех тех же условиях последовательная реализация справляется с задачей, а параллельная нет.

Появляется ошибка следующего содержания:

Листинг 6: err

```
1 # There is insufficient memory for the Java Runtime Environment to continue.
2 # Native memory allocation (malloc) failed to allocate 2097152 bytes for
  ↪ AllocateHeap
```

Таким образом, куча была переполнена. При оптимизации, например если убрать вывод в консоль результатов, максимально допустимый размер который получилось отсортировать 78 миллионов, при 80 миллионах снова выводится ошибка отсутствия свободного места.

При этом мы опять мы можем заметить, что на подходе к ошибке, программа теряет разницу между последовательной работой, а затем уступает ей. В результате изучения вопроса, было выяснено, что данные задержки были связаны с уборщиком мусора, который работает уже по факту заполнения, а не когда захочет пользователь или же заранее. Таким образом, данный подход, возможно менее оптимален для параллельных вычислений больших объёмов данных нежели C++.

Далее можно построить график средних значений скорости, в зависимости от объёма массива.

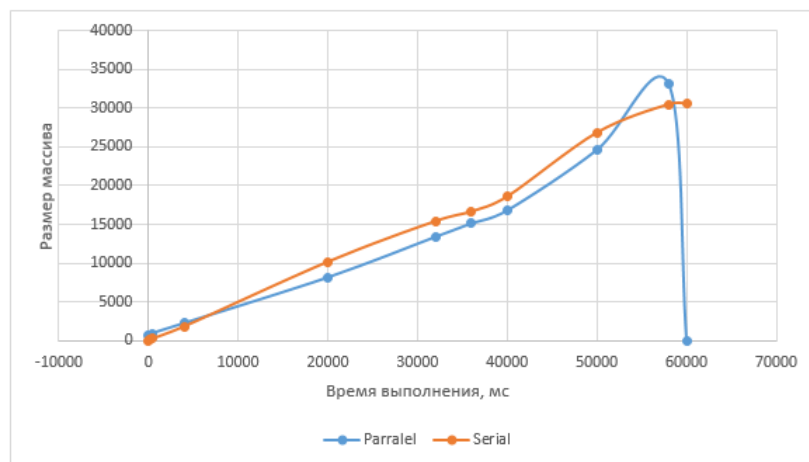


Рисунок 5.1. Зависимость времени выполнения от длины массива

Исходя из полученных результатов, мы можем сделать вывод, что разница между последовательным и параллельным выполнением в моей программе не существенно, следовательно ключевые проблемы такой производительности кроются в программе, по этому было принято решение произвести оптимизацию программы.

#### Листинг 7: MergeSort.java

```

1 package Sorting;
2
3 public class MergeSort {
4
5
6     // merge sorting
7     public static int [] sortArray(int [] arrayA){
8         if (arrayA == null || arrayA.length < 2) {
9             return arrayA;
10        }
11
12        int [] arrayB = new int[arrayA.length / 2];
13        System.arraycopy(arrayA, 0, arrayB, 0, arrayA.length / 2);
14
15        int [] arrayC = new int[arrayA.length - arrayA.length / 2];
16        System.arraycopy(arrayA, arrayA.length / 2, arrayC, 0, arrayA.length -
17        ↪ arrayA.length / 2);
18
19        arrayB = sortArray(arrayB);

```

```

19         arrayC = sortArray(arrayC);
20
21         return mergeArray(arrayB, arrayC);
22     }
23
24     public static int[] mergeArray(int[] a1, int[] a2) {
25         int[] b = new int[a1.length + a2.length];
26         int positionA1 = 0;
27         int positionA2 = 0;
28
29         for(int i = 0; i < b.length; i++) {
30             if(positionA1 == a1.length){
31                 b[i] = a2[positionA2];
32                 positionA2++;
33             } else if(positionA2 == a2.length){
34                 b[i] = a1[positionA1];
35                 positionA1++;
36             } else if(a1[positionA1] < a2[positionA2]){
37                 b[i] = a1[positionA1];
38                 positionA1++;
39             } else {
40                 b[i] = a2[positionA2];
41                 positionA2++;
42             }
43         }
44         return b;
45     }
46
47     // for final merge
48     public static int[][] splitting(int[][] subArr){
49         if (subArr.length == 1) {
50             return subArr;
51         }
52
53         int[][] arr = new int[subArr.length/2][subArr[0].length * 2];
54
55         for (int i = 0, j = 0; i < subArr.length; i+=2, j++) {
56             arr[j] = mergeArray(subArr[i], subArr[i + 1]);
57         }
58
59         return splitting(arr);
60     }
61 }

```

Как мы видим появился метод для финального слияния оптимизированный под любое количество потоков. Который избегает многократного рекурсивного копирования массивов полной длины как в самом классе сортировки, наши массивы больше не разделяются на ещё более мелкие массивы, а соединяются сразу. Таким образом улучшения были ощутимыми.

#### Листинг 8: Parallel.java оптимизированный

```

1 import Sorting.MergeSort;
2 import helpers.ListGenerator;
3 import mpi.MPI;
4 import java.util.Arrays;
5
6 import static Sorting.MergeSort.finMerge;
7 import static Sorting.MergeSort.mergeArray;
8
9 public class Parallel {

```

```

10  public static void main(String[] args) {
11      // Init MPI
12      MPI.Init(args);
13      // rank - number of process
14      // size - quantities of process
15      int rank = MPI.COMM_WORLD.Rank(), size = MPI.COMM_WORLD.Size(), root =
16      ↪ 0;
17      // length - length of array which we wanna sort
18      // unitSize - length of one process
19      int length = ListGenerator.getLength(), unitSize = length / size;
20      // send - what we send from root to all
21      // recv - what we receive from root
22      // newrecv - what we receive in root from all
23      int[] sendBuffer = new int[length], recvBuffer = new int[unitSize],
24      ↪ newRecvBuffer = new int[length];
25
26      if(rank == root){
27          // generate random array
28          sendBuffer = new ListGenerator(length).getInts();
29      }
30
31      long timeStart = System.currentTimeMillis();
32      MPI.COMM_WORLD.Scatter(sendBuffer, 0, unitSize, MPI.INT, recvBuffer, 0,
33      ↪ unitSize, MPI.INT, root);
34
35      recvBuffer = MergeSort.sortArray(recvBuffer);
36
37      MPI.COMM_WORLD.Gather(recvBuffer, 0, unitSize, MPI.INT, newRecvBuffer,
38      ↪ 0, unitSize, MPI.INT, root);
39
40      if (rank == root) {
41          // finall merge
42          finMerge(ListGenerator.getRankedInts(size, newRecvBuffer));
43          System.out.println("Sorted_" + newRecvBuffer.toString());
44          long timeStop = System.currentTimeMillis();
45          System.out.printf("Time_is:_%d_ms", timeStop - timeStart);
46      }
47      MPI.Finalize();
48  }

```

Также, было переставлено положение таймеров, которые отсчитывают время, так как изначально время начала фиксировалось в момент генерации массива, что не корректно, так как генерация массива может быть отличной по времени от раза к разу, за счёт случайности данных, но, главное, что генерация массива является однопоточным в обоих вариантах, при этом не относящимся непосредственно к алгоритму выполняемой задачи, а служат лишь исходными данными.

Для выполнения программы было выбрано количество потоков, равное 4-ем.

Таб.2. Время выполнения (мс) в зависимости от размера массива (тысячи)

Размер	Параллельно	Последовательно
0,4	14,4	10,6
4	16,2	15,6
40	27,2	24,6
400	74,6	129,4
4000	496	1048
20000	1557	4817
40000	3284	8955
60000	5921	13347

Как мы можем заметить, на маленьких значениях наблюдается большая скорость у последовательной обработки, так как на выделение потоков тратятся ресурсы, но очень быстро, уже после 40 тысяч, пальму первенства отбирает параллельная программа, при том разница практически везде более чем трехкратная.

В идеале при использовании 4-х потоков вместо одного, наша программа должна выполняться в 4 раза быстрее. Но в нашем случае мы знаем, что у нас присутствует финальное слияние, которое по сути своей линейно, и выполняется одним потоком, что в целом, конечно, делает время выполнения больше, но учитывая что каждый поток сортирует свою часть массива и её надо слить в единый отсортированный массив, то у нас нет другого выбора, кроме как пожертвовать частью производительности.

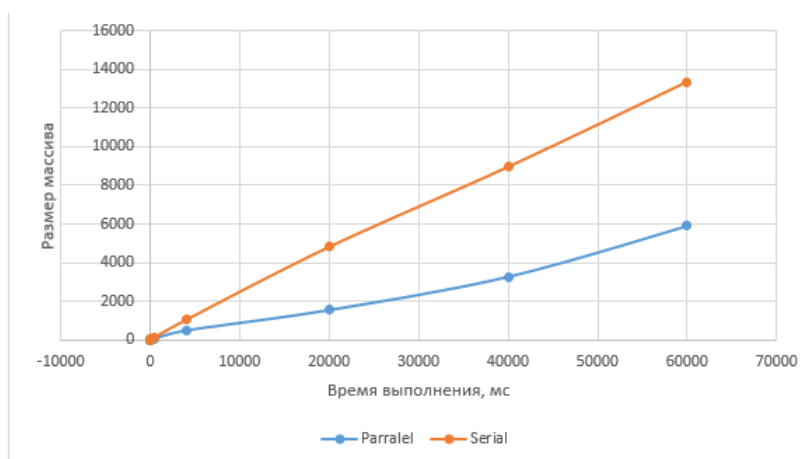


Рисунок 5.2. Зависимость времени выполнения от длины массива

Как видно из графика разница достигается достаточно значительно, при этом там, где предыдущий вариант завершался с переполнением памяти, данный вариант оптимизирован и на этот счёт, так как в коде теперь отсутствует бесполезный вывод массивов до и после сортировки, что было крайне ресурсоёмкой задачей, особенно при больших значениях массива.

### 5.3. Оценка производительности работы параллельной программы в зависимости от количества потоков

В данном эксперименте мы взяли длину массива в 20 миллионов, и сделали прогоны на различном количестве потоков. Всего, имеющейся процессор обладает 4 ядрами и 8 потоками, и больше чем 8 потоков сравнивать попросту не имеет смысла.

В результате выполнения, что наихудшее время работы, ожидаемо находится у 2 потоков, а наилучшее у 8. Да, разница между временем выполнения от 4 до 8 потоков не столь значительна. Обусловлено это тем, что при финальном слиянии выполняемом в корневом потоке, сливаются по парно по два массива при том рекурсивно. И если при 2 потоках мы сливаем наши массивы 1 раз, то при 8 потоках финальное слияние происходит уже 4 раза. Ввиду этого, данная скорость работы вполне оправдана. Стоит найти баланс между количеством потоков и скоростью работы. Таким образом, наиболее адекватным выбором, на мой взгляд будет 4 потока. Так как мы имеем всего 2 финальных слияния, при этом время работы не сильно отличается при средней длине массива. При условиях полной загрузки машины, естественно мы отдаём предпочтение максимальному выделению потоков, но при частичной загрузке, вполне достаточно и 4-6 потоков.

Таб.3. Скорость выполнения (мс) от количества потоков

Количество потоков	Время работы, мс
2	2662
3	2022
4	1557
5	1555
6	1554
7	1344
8	1298

### 5.3.1. Оценка результатов прогонов при помощи статистических показателей

В следующем эксперименте, нам было необходимо выбрать какой-либо средний размер массива ( 20 миллионов) и провести на нём 50 замеров для каждой реализации и вывести таким образом основные статистические показатели ( среднее, доверительный интервал, дисперсия и отклонение).

Таб.4. Основные статистические показатели

Кол-во, тыс. ед	0,4		4		40		400	
Реализация	Parralel	Serial	Parralel	Serial	Parralel	Serial	Parralel	Serial
Среднее значение	26,72	24,58	32,34	27,82	44,28	41,12	175,88	109,1
Дисперсия	23,5216	27,7236	36,9444	43,5476	44,2016	47,1456	127,2256	95,85
Ср.кв. откл.	4,849907	5,265321	6,078191	6,599061	6,648428	6,866265	11,27943	9,790301
Дов.инт. 95	0,04323	0,046933	0,054179	0,058821	0,059262	0,061203	0,100541	0,087267

Кол-во, тыс. ед	4м		20м		40м		60м	
Реализация	Parralel	Serial	Parralel	Serial	Parralel	Serial	Parralel	Serial
Среднее значение	1274,24	644,4	1557,3	4817	9837,42	4077,3	15009,2	7588,96
Дисперсия	2236,342	774,08	491	687,1	2000,124	2136,89	3170,32	12435,56
Ср.кв. откл.	47,28998	27,82229	22,16	26,21	44,72274	46,22651	56,30559	111,5148
Дов.инт. 95	0,421525	0,247997	0,303	0,302	0,398641	0,412045	0,71357	1,413246



По результатам расчётов, мы можем считать, что наша выборка является достоверной так как дисперсия, как мера рассеянности выборки показывает адекватные показатели в большинстве случаев. Среднеквадратическое отклонение напрямую зависит от дисперсии, а также от размеров выборки.

Доверительный интервал, от длины массива 400 до 60 000 000 постепенно растёт. Следовательно, мы можем сказать что до 40 миллионов вероятность попасть в наш доверительный интервал достаточно велика, но уже для 60 миллионов, достаточно не стабильна, так как данный показатель не должен быть настолько большим. Это может говорить о каком-то пороге, с которым работает программа, например сборщик мусора, который так или иначе замедляет работу.

## 6. Заключение

В данном занятии была произведена разработка алгоритма параллельной сортировки слиянием, со сравнением его работы с последовательной реализацией, а также с оценкой достоверности предоставляемых данных.

В ходе данной работы мы поняли, что для небольших исходных данных никакого смысла делать параллельную реализацию нет, это дольше, при том как в исполнении так и в реализации. Для средних и крупных размеров данных параллельная реализация показывает себя лучше.

Но стоит сделать оговорку, о том, что у каждого языка есть своя специфика, и автоматическая сборка мусора является специфичной для Java, и является одним из узких мест спроектированной системы. Безусловно при помощи Java можно обрабатывать намного большие объёмы данных, но о данном факте необходимо было знать заранее, чтобы проектировать всю логику с учётом возможности переполнения кучи памяти.