

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

1. Цель работы	3
2. Программа работы	3
3. Сведения о системе	3
4. Теоретическая информация	4
5. Ход выполнения работы	4
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
5.2.2. Оценка результатов прогонов при помощи статистических показателей	12
6. Заключение	13

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    }
```

```

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.

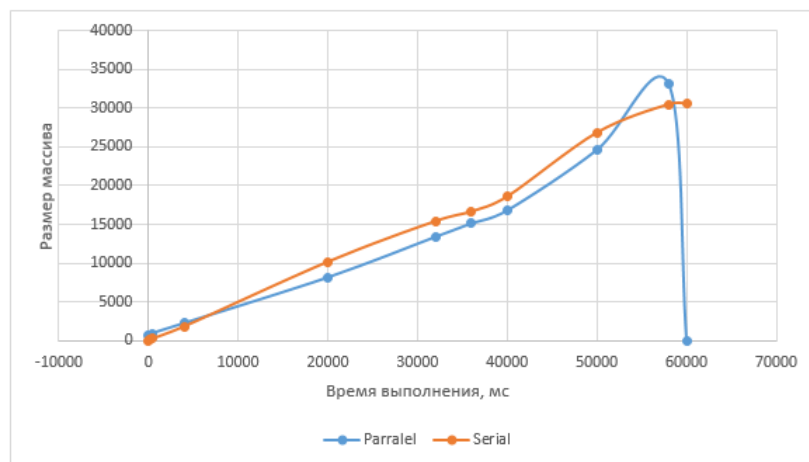


Рисунок 5.1. Зависимость времени выполнения от длины массива

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

	Parralel	Serial
Среднее значение	81159	10175
Дисперсия	53	38
Ср.квaдр отклонение	230	195

По результатам расчётов, мы можем считать, что наша выборка является достоверной, так как дисперсия имеет крайне малое значение, а значит результаты каждого замера достаточно не значительно отличаются друг от друга.

Далее необходимо посчитать доверительный интервал, чтобы определить насколько вероятно при любой случайно попытке получить значение близкое значение к среднему с погрешностью в среднеквадратическое отклонение.

Доверительный интервал будет определяться по следующей формуле:

$$(\bar{x} - t_{kp} \frac{s}{\sqrt{n}}; \bar{x} + t_{kp} \frac{s}{\sqrt{n}})$$

Найдём интервал для последовательной реализации.

Поскольку $n > 30$, то определяем значение t_{kp} по таблицам функции Лапласа. В этом случае

$$2\Phi(t_{kp}) = Y$$

$$\Phi(t_{kp}) = Y/2 = 0.95/2 = 0.475$$

По таблице функции Лапласа найдем, при каком t_{kp} значение

$$\Phi(t_{kp}) = 0.475$$

$$t_{kp}(Y) = (0.475) = 1.96$$

Теперь подставляем полученные значения в формулу ниже:

1) Для последовательной реализации

$$\epsilon = t_{kp} \frac{s}{\sqrt{n}} = 1.96 \frac{0.23}{\sqrt{49}} = 0.0644$$

$$(10.175 - 0.0644; 10.175 + 0.0644) = (10.11; 10.24)$$

С вероятностью 0.95 можно утверждать, что среднее значение при выборке большего объема не выйдет за пределы найденного интервала.

2) Для параллельной реализации.

$$\epsilon = t_{kp} \frac{s}{\sqrt{n}} = 1.96 \frac{0.195}{\sqrt{50}} = 0.0541$$

$$(8.1 - 0.0541; 8.1 + 0.0541) = (8.05; 8.15)$$

С вероятностью 0.95 можно утверждать, что среднее значение при выборке большего объема не выйдет за пределы найденного интервала.

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 объемы данных, но о данном факте необходимо было знать заранее, чтобы проектировать всю логику с учётом возможности переполнения кучи памяти.