



МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»  
КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту  
«Технології паралельних обчислень. Курсова робота»  
Тема: Алгоритм сортування злиттям

**Керівник:**

проф. Стеценко І. В.

«Допущено до захисту»

---

«\_\_\_» \_\_\_\_\_ 2023 р.

Захищено з оцінкою

---

Члени комісії:

---

---

**Виконавець:**

Чабан А. Є.

студент групи ІТ-03

залікова книжка № ІТ-0325

---

«б» червня 2023 р.

Інна СТЕЦЕНКО

Київ – 2023

## ЗАВДАННЯ

1. Виконати огляд існуючих реалізацій алгоритму, послідовних та паралельних, додати посилання на використані джерела інформації. Зробити висновок щодо актуальності виконаного дослідження.
2. Виконати розробку послідовного алгоритму у відповідності до варіанту завдання та обраного програмного забезпечення для реалізації. Дослідити швидкодію алгоритму при зростанні складності обчислень та зробити висновки про необхідність паралельної реалізації алгоритму.
3. Виконати розробку паралельного алгоритму у відповідності до варіанту завдання та обраного програмного забезпечення для реалізації. Забезпечити зручне введення даних для початку обчислень.
4. Виконати тестування алгоритму, що доводить коректність результатів обчислень.
5. Виконати дослідження швидкодії алгоритму при зростанні кількості даних для обчислень.
6. Виконати експериментальне дослідження прискорення розробленого алгоритму при зростанні кількості даних для обчислень.
7. Зробити висновки про переваги паралельної реалізації обчислень для алгоритму, що розглядається у курсовій роботі, та програмних засобів, які використовувались.

## РЕФЕРАТ

У рамках виконання курсової було спроектовано та реалізовано послідовний та паралельний алгоритм сортування злиттям, розкрито їх актуальність та фактори що вплинули на швидкодію паралельних обчислень.

Алгоритми сортування масивів даних є базовими задачами в програмуванні та проектуванні алгоритмів, а сортування злиттям, в свою чергу, є одним із ефективних рішень цієї задачі. Паралельна реалізація ж допоможе покращити ефективність цього алгоритму на великих обсягах даних.

У якості мови програмування для реалізації застосунку було обрано Java, так як мова добре розвинута, підтримується та має засоби для виконання паралельних обчислень, а також фреймворки які допомагають досягнути кращої швидкодії.

Було написано псевдокод алгоритму, виконано послідовну реалізацію та описану часову складність алгоритму. Далі алгоритм було протестовано на різних обсягах даних для виконання подальшої порівняльної характеристики між реалізаціями.

Для реалізації паралельного алгоритму було спроектовано логіку для розпаралелення алгоритму, виконано розробку з використанням Fork/Join Framework, протестовано роботу паралельної реалізації змінюючи кількість потоків та обсяг вхідних даних.

Згідно з результатами досліджень були зроблені висновки щодо успішності та ефективності паралельної реалізації алгоритму сортування злиттям засобами мови програмування Java.

Експерименти було виконано на машині з відповідними характеристиками:

- Процесор: Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 2.50 GHz
- Оперативна пам'ять: 16 GB

Ключові слова: ПАРАЛЕЛЬНИЙ АЛГОРИТМ, СОРТУВАННЯ ЗЛИТТЯМ, JAVA, FORK/JOIN FRAMEWORK, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ

## ЗМІСТ

<b>ВСТУП.....</b>	<b>6</b>
<b>1 ОПИС ПОСЛІДОВНОГО АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ.....</b>	<b>8</b>
1.1 Послідовний алгоритм.....	8
1.2 Паралельні реалізації .....	9
<b>2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ .....</b>	<b>10</b>
2.1 Псевдокод послідовного алгоритму.....	10
2.2 Реалізація послідовного алгоритму.....	10
2.3 Тестування роботи алгоритму .....	12
2.4 Аналіз швидкодії алгоритму .....	13
<b>3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС.....</b>	<b>15</b>
<b>4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ .....</b>	<b>17</b>
4.1 Проектування паралельної реалізації алгоритму.....	17
4.2 Реалізація паралельного алгоритму сортування злиттям з використанням Fork/Join Framework .....	17
4.3 Тестування паралельного алгоритму .....	19
4.4 Аналіз швидкодії паралельного алгоритму.....	20
<b>5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ.....</b>	<b>22</b>
<b>ВИСНОВКИ.....</b>	<b>26</b>

<b>СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ .....</b>	<b>27</b>
<b>ДОДАТКИ.....</b>	<b>28</b>
Додаток А. Код програми.....	28
Додаток Б. Тестування на правильність роботи .....	34

## ВСТУП

Працюючи з різноманітними задачами у сучасному світі, в якому швидкість обробки великого обсягу даних є критичним фактором для досягнення успіху в різних галузях, актуальність паралельних обчислень постійно зростає. Сучасні комп'ютери за своїми характеристиками дозволяють роботу програм у декількох потоках завдяки своїй архітектурі та підтримці багатопоточності цими програмами.

Паралельні обчислення дозволяють розподіляти обчислювальні завдання між багатьма обчислювальними ресурсами, що дозволяє значно прискорити час виконання завдань.

На сьогоднішній день, паралельні та розподілені обчислення стали поширеними і продовжують набирати популярності. Однією із багатьох задач які можна виконати завдяки паралельним обчисленням – задачі про сортування. Ці задачі дуже поширені у людському світі, навіть звичайний каталог магазину сортується за цінами, а якщо його обсяг великий, то відповідно і час очікування клієнту зростатиме. Тут і стають в нагоді паралельні обчислення, які можуть прискорити час роботи алгоритму з великими обсягами даних.

Ефективна обробка і сортування великого обсягу інформації є важливим завданням для багатьох застосувань. Алгоритм сортування злиттям (Merge Sort) є одним з найефективніших алгоритмів сортування, і його багатопоточна реалізація виникає як відповідь на вимогу до ефективної реалізації.

Одним з головних викликів при сортуванні великих наборів даних є час, потрібний для обробки і сортування. Багатопоточна реалізація алгоритму сортування злиттям створює можливість паралельної обробки даних за допомогою використання багатьох потоків виконання. Кожен потік може обробляти свою власну частину набору даних, а після цього результати зливаються для отримання відсортованого масиву.

Багатопоточна реалізація алгоритму сортування злиттям забезпечує приріст продуктивності і швидкості сортування. Це досягається завдяки

використанню паралельної обробки даних, що дозволяє ефективно використовувати ресурси багатоядерних та багатопроцесорних систем. Завдяки розділенню завдання сортування між різними потоками, час виконання може бути значно зменшений, що робить багатопоточну реалізацію алгоритму сортування злиттям особливо актуальною для обробки великих об'ємів даних в сучасних завданнях.

## **1 ОПИС ПОСЛІДОВНОГО АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ**

Алгоритм сортування злиттям (MergeSort) є ефективним алгоритмом сортування, який базується на принципі "розділяй та володарюй". Він використовує рекурсивний підхід до розбиття списку на менші підсписки, сортує кожен з них окремо, а потім злиттям об'єднує відсортовані підсписки, поки не отримаємо повністю відсортований список.

Проблема: сортування списку елементів з довільним порядком у висхідному або низхідному порядку.

Для того, щоб розпаралелити алгоритм, потрібно виділити задачу, що може обчислюватися незалежно від інших. Для цього спочатку необхідно дослідити класично, послідовну реалізацію алгоритму сортування злиттям.

Вхідні дані: масив даних заданої величини які необхідно відсортувати.

Вихідні дані: відсортований за зростанням масив даних заданої величини.

### **1.1 Послідовний алгоритм**

Порядок дій послідовного алгоритму сортування злиттям:

1. Перевірка базового випадку: Якщо масив містить лише один елемент або ж є порожнім, то він вже відсортований. В цьому випадку повертається масив без змін.
2. Рекурсивне розбиття масиву: Вихідний масив розбивається на дві половини. Це можна зробити, наприклад, шляхом знаходження серединного елемента і розділення масиву навпіл.
3. Рекурсивне сортування: Застосовується сортування злиттям для кожної половини масиву, викликаючи алгоритм рекурсивно. Тобто, ліва та права половини масиву сортуються окремо.
4. Злиття: Після того як ліва та права половини масиву відсортовані, необхідно злити їх у впорядкований вихідний масив. Це робиться шляхом порівняння найменших елементів лівої та правої частини і



додавання їх до нового масиву. Якщо одна половина масиву містить більше елементів, ніж інша, то залишок цієї половини також додається до вихідного масиву.

#### 5. Повернення результату: Повертаємо відсортований масив.

Весь алгоритм виконується рекурсивно, допоки не досягається стартова точка рекурсивного виклику.

## 1.2 Паралельні реалізації

На сьогоднішній день вже існують деякі паралельні реалізації алгоритму сортування злиттям, але ніяка з них не є загальноприйнятим стандартом, так як існує багато варіантів сортування даних, які використовуються по ситуації.

Однією з публічних існуючих реалізацій є робота Christopher Zelenka, написана за стандартом MPI [\[1\]](#).

Проаналізувавши хід роботи послідовного алгоритму, помітно що алгоритм можна розпаралелити розділивши вхідний масив на менші частини, які будуть сортуватися окремо, а потім об'єднати відсортовані частини [\[6\]](#).

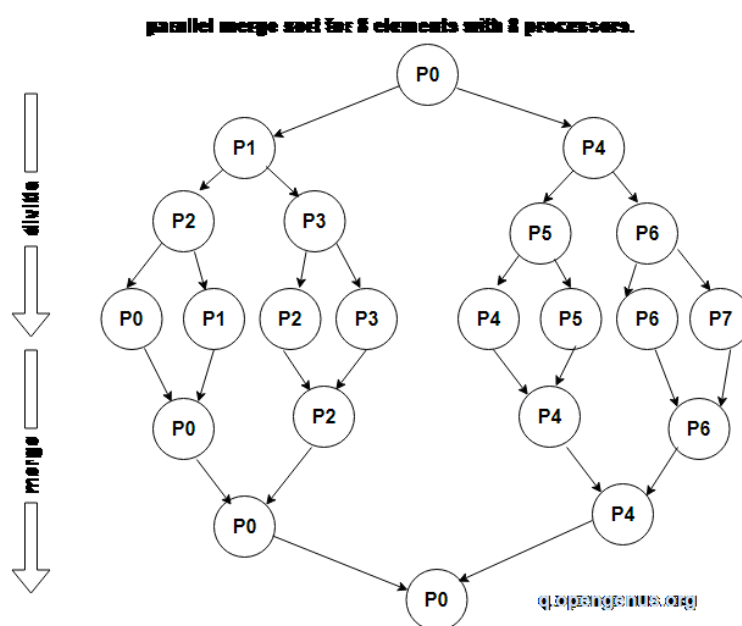


Рисунок 1.1. – Візуалізація паралельного алгоритму [\[6\]](#).

## 2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

### 2.1 Псевдокод послідовного алгоритму

Порядок виконання алгоритму згідно псевдокоду алгоритму [\[7\]](#):

`mergeSort(arr):`

`If arr.size > 1`

Крок 1: Визначаємо розмір `leftSubArray` та `rightSubArray` , для того щоб розділити вхідний масив на 2 частини

`leftSize = arr.size / 2;`

`rightSize = arr.size - leftSize;`

Крок 2: Виклик `mergesort` для `leftSubArray`

`mergeSort(leftSubArray);`

Крок 3: Виклик `mergesort` для `rightSubArray`

`mergeSort(rightSubArray);`

Step 4: Виклик методу `merge` для злиття двох частин

`merge (arr, leftSubArray, rightSubArray)`

### 2.2 Реалізація послідовного алгоритму

Для реалізації алгоритму сортування злиттям, згідно описаного вище псевдокоду, необхідно написати 2 методи `mergeSort()` та `merge()` [\[1\]](#), [\[2\]](#).

Було створено клас `SeqMergeSorter`, в якому реалізовано відповідні методи.

Метод `mergeSort()` приймає масив цілих чисел. Умовою виходу з рекурсивного виклику є розмір масиву менше 2. Далі визначається центральний елемент та задаються розміри масивів для розділення основного, після чого

відповідні елементи копіюються до відповідних масивів залежно від їх розміщення. Виконується рекурсивний виклик методу `mergeSort()` для лівого та правого масивів відповідно.

```

3 usages Anton Chaban
5 @ public void mergeSort(int[] array) {
6     var size = array.length;
7
8     if (size < 2) {
9         return;
10    }
11
12    var mid = size / 2;
13    var left = new int[mid];
14    var right = new int[size - mid];
15
16    for (int i = 0; i < mid; i++) {
17        left[i] = array[i];
18    }
19    for (int i = mid; i < size; i++) {
20        right[i - mid] = array[i];
21    }
22
23    mergeSort(left);
24    mergeSort(right);
25
26    merge(array, left, right);
27 }

```

Рисунок 2.1. – Реалізація послідовного алгоритму

Далі викликається метод `merge()`, який приймає в собі 3 масиви, основний, ліву та праву частини [2].

Перша частина коду виконує злиття двох масивів `left` та `right` за допомогою циклу `while`. В цьому циклі порівнюються елементи з двох масивів, і менший з них копіюється в результуючий масив. Цикл `while` продовжується до тих пір, поки не буде досягнута кінцева позиція одного з масивів. Після завершення першого циклу один з масивів може мати залишкові елементи, які ще не були включені в результуючий масив. Щоб включити ці залишкові елементи, використовуються ще два цикли `while`. Які переносять залишкові елементи. Ці дії реалізовано в методі `merge()`, який поданий на рисунку нижче:

```

1 usage  Anton Chaban
29 @
30 public static void merge(int[] arr, int[] left, int[] right) {
31     int i = 0, j = 0, k = 0;
32     while (i < left.length && j < right.length) {
33         if (left[i] < right[j])
34             arr[k++] = left[i++];
35         else
36             arr[k++] = right[j++];
37     }
38     while (i < left.length) {
39         arr[k++] = left[i++];
40     }
41     while (j < right.length) {
42         arr[k++] = right[j++];
43     }
44 }

```

Рисунок 2.2. – Реалізація методу злиття

### 2.3 Тестування роботи алгоритму

Для тестування коректності роботи алгоритму було створено допоміжний клас ArrayTools, в якому є 3 методи для генерації, перевірки коректності сортування масиву та друку. Для тестування коректності сортування використовується метод isSorted(), який проходиться по масиву і перевіряє чи елементи розміщені за зростанням, реалізація на рисунку 2.3:

```

4 usages  Anton Chaban
public static boolean isSorted(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        if (arr[i] > arr[i + 1])
            return false;
    }
    return true;
}

```

Рисунок 2.3. – Метод перевірки коректності сортування

Для початку необхідно перевірити коректність роботи послідовного алгоритму на невеликому обсязі даних, результат на рисунку 2.4:

```

C:\Users\anton\.jdk\corretto-17.0
Array to sort:
52 1 36 7 59 31 65 56 43 7
#####
1 7 7 31 36 43 52 56 59 65
Array is sorted

Process finished with exit code 0

```

## Рисунок 2.4. – Результат тестування

### 2.4 Аналіз швидкодії алгоритму

Результати роботи послідовного алгоритму представлено в таблиці 2.1:

Таблиця 2.1. – Час виконання послідовного алгоритму сортування злиттям в залежності від розміру масиву.

Розмір	Час виконання (у мілісек)
25000	11
50000	15
100000	36
500000	117
1000000	191
5000000	870
10000000	1431
15000000	1983
20000000	2587
30000000	4024
50000000	6796
100000000	13783
150000000	23608

Графік залежності росту часу виконання від розміру масиву представлено на рисунку 2.5.



Рисунок 2.5. – Графік результату замірів часу.

Дивлячись на результати тестування, можна побачити, що час виконання обчислень зростає разом з розміром масиву. Отже, чим більший об'єм вхідних даних необхідно обробити – тим більше часу витрачається на обчислення.

### **З ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС**

Для виконання розробки паралельної реалізації алгоритму сортування злиттям було обрано Java, вона є однією з найпопулярніших мов програмування, яка забезпечує широкий спектр функціональних можливостей для розробки алгоритмів та ефективної обробки даних. Java з ранніх версій підтримувала багатопоточність та має багатий набір бібліотек, що сприяє швидкому розробленню та впровадженню алгоритмів. Саме тому Java є чудовим вибором для розв'язання даної задачі.

Класи, що надають необхідний функціонал для роботи з багатопоточністю, знаходяться у пакеті `java.util.concurrent`. Fork/Join Framework є важливою частиною пакету `java.util.concurrent`, що надає зручні та потужні засоби для паралельного програмування. Його основною концепцією є розбиття завдання на менші підзадачі, їх незалежне виконання та об'єднання результатів. Це дозволяє розподілити обчислення між доступними процесорними ядрами і забезпечити більшу продуктивність [\[4\]](#).

Основними компонентами Fork-Join Framework є:

**ForkJoinPool:** Є основною складовою фреймворка. Це реалізація служби `ExecutorService`, яка керує робочими потоками і надає інструменти для отримання інформації про стан і продуктивність пулу потоків.

**ForkJoinTask:** Це базовий клас для всіх завдань, які можна виконувати в Fork-Join Framework. Він є абстрактним класом, який можна розширювати для створення власних завдань.

**RecursiveAction:** Цей клас представляє завдання, яке не повертає значення (`void`). Воно просто виконує певну роботу або обчислення.

**RecursiveTask:** Цей клас представляє завдання, яке повертає результат. Він виконує певні обчислення та повертає значення.

Окрім бібліотек які стосуються безпосередньо паралельної реалізації алгоритму були використані допоміжні засоби для зменшення об'єму та підвищення читабельності коду. Так для скорочення кількості шаблонного коду використовується бібліотека Lombok. Для підключення цієї бібліотеки також використовувався інструмент для керування залежностями який називається Maven, підключення прописано у файлі pom.xml.



## **4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ**

### **4.1 Проектування паралельної реалізації алгоритму**

Дивлячись послідовну реалізацію алгоритму, можна помітити що сама суть алгоритму включає в себе розділення вхідного масиву на два підмасиви. Отже, в цьому напрямку і варто будувати логіку розпаралелення алгоритму.

Для реалізації багатопоточної версії використаємо підхід поділу відповідальності за частини масиву між потоками. Масив буде рекурсивно поділятися на 2 підмасиви, сортування підмасивів буде відбуватися паралельно. Далі поділені масиви будуть зливатися звичайним методом `merge()` з послідовної реалізації.

Отже, для реалізації алгоритму потрібно розділити масив на дві частини, очікувати завершення сортування обох підмасивів і після цього виконувати їх зливання.

Алгоритм дій з використанням Fork/Join Framework буде наступним:

- Розбиття вхідного масиву на два підмасиви.
- Паралельне виконання сортування для кожного підмасиву, використовуючи рекурсивний підхід.
- Очікуємо завершення паралельних задач та об'єднуємо впорядковані підмасиви в один відсортований масив.

### **4.2 Реалізація паралельного алгоритму сортування злиттям з використанням Fork/Join Framework**

Для реалізації паралельного алгоритму сортування злиттям було створено клас `ParallelMergeSorter` який успадковує абстрактний клас `RecursiveAction`, з пакету `java.util.concurrent`. Клас за допомогою конструктора (створено з використанням Lombok анотації) приймає в себе масив який необхідно відсортувати [\[5\]](#).

```

7
4 usages  Anton Chaban *
8 @AllArgsConstructor
9 public class ParallelMergeSorter extends RecursiveAction {
10     private int[] arr;
11

```

Рисунок 4.1. – Клас для паралельної реалізації

Наслідування від класу `RecursiveAction` потребує перевизначення методу `compute()`, в цьому методі виконується основна багатопоточна логіка алгоритму. До змінної `size` записується розмір масиву, умовою виходу з рекурсивного виклику є розмір масиву менший двох – це означає що масив не потрібно сортувати. В усіх інших випадках алгоритм визначає середину масиву та розділяє його на дві частини, записуючи відповідну частину даних. За допомогою методу `invokeAll()` створюється два нових завдання для лівої та правої частини масиву відповідно. Після завершення обробки лівої та правої частин масиву викликається послідовний метод `merge()` який поєднує два відсортовані підмасиви. Використання методу `invokeAll()` гарантує, що головний потік чекатиме завершення виконання обох підзавдань перед тим як продовжити свою роботу [4].

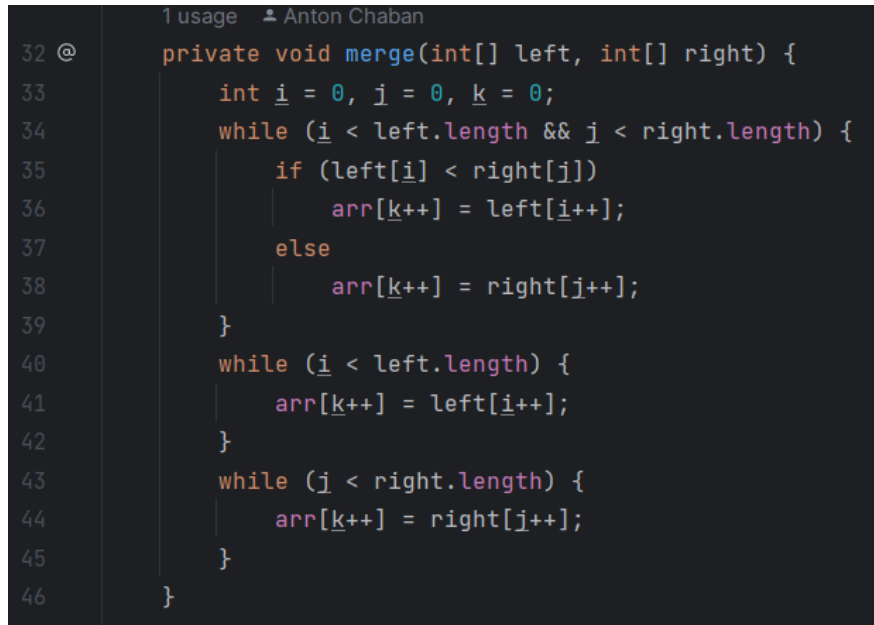
```

12 @Override
13 protected void compute() {
14     var size = arr.length;
15
16     if (size < 2) {
17         return;
18     }
19     var mid = size / 2;
20
21     var left = new int[mid];
22     System.arraycopy(arr, srcPos: 0, left, destPos: 0, mid);
23
24     var right = new int[arr.length - mid];
25     System.arraycopy(arr, mid, right, destPos: 0, length: arr.length - mid);
26
27     invokeAll(new ParallelMergeSorter(left), new ParallelMergeSorter(right));
28     merge(left, right);
29 }
30

```

Рисунок 4.2. – Перевизначення методу `compute()` для розпаралелення алгоритму

Метод `merge()` працює за такою самою логікою як і в послідовній реалізації в циклі порівнюються елементи з двох масивів, і менший з них копіюється в результуючий масив, а залишкові елементи, які ще не були включені в результуючий масив, переносяться в результуючий масив. Реалізацію метода наведено на рисунку 4.3.



```

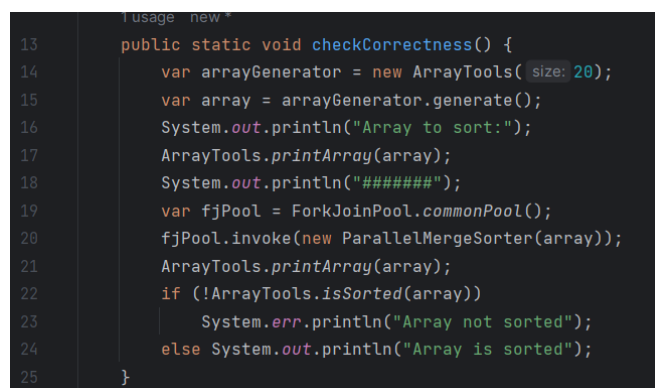
32 @ private void merge(int[] left, int[] right) {
33     int i = 0, j = 0, k = 0;
34     while (i < left.length && j < right.length) {
35         if (left[i] < right[j])
36             arr[k++] = left[i++];
37         else
38             arr[k++] = right[j++];
39     }
40     while (i < left.length) {
41         arr[k++] = left[i++];
42     }
43     while (j < right.length) {
44         arr[k++] = right[j++];
45     }
46 }

```

Рисунок 4.3. – Метод для злиття двох відсортованих масивів

### 4.3 Тестування паралельного алгоритму

Для тестування використовується вже раніше створений клас `ArrayTools`, в якому є методи для перевірки коректності роботи. Спершу було виконано перевірку коректності роботи алгоритму на невеликому масиві, метод для тестування представлено на рисунку 4.4.



```

13 public static void checkCorrectness() {
14     var arrayGenerator = new ArrayTools( size: 20);
15     var array = arrayGenerator.generate();
16     System.out.println("Array to sort:");
17     ArrayTools.printArray(array);
18     System.out.println("#####");
19     var fjPool = ForkJoinPool.commonPool();
20     fjPool.invoke(new ParallelMergeSorter(array));
21     ArrayTools.printArray(array);
22     if (!ArrayTools.isSorted(array))
23         System.err.println("Array not sorted");
24     else System.out.println("Array is sorted");
25 }

```

Рисунок 4.4. – Метод для тестування алгоритму

Результат тесту наведений на рисунку 4.5. Видно що алгоритм відпрацьовує коректно на сортує масив за зростанням.

```
C:\Users\anton\.jdk\corretto-17.0.6\bin\java.exe -javaagent:0
Array to sort:
57 38 30 65 62 71 22 14 46 42 44 7 10 13 86 70 11 6 63 0
#####
0 6 7 10 11 13 14 22 30 38 42 44 46 57 62 63 65 70 71 86
Array is sorted

Process finished with exit code 0
```

Рисунок 4.5. – Результат тестування алгоритму

#### 4.4 Аналіз швидкодії паралельного алгоритму

Для перевірки швидкодії алгоритму було створено методи для тестування з замірами часу на масивах різного обсягу.

```
1 usage new *
29 public static void measureTime() {
30     var SIZES = new int[]{25_000, 50_000, 100_000, 500_000, 1_000_000, 5_000_000, 10_000_000, 15_000_000, 20_000_000
31         , 30_000_000, 50_000_000, 100_000_000, 150_000_000};
32     for (int size : SIZES) {
33         checkTime(size);
34     }
35 }
36
37 1 usage new *
38 private static void checkTime(int size) {
39     var arrayGenerator = new ArrayTools(size);
40     var array = arrayGenerator.generate();
41     var fjPool = ForkJoinPool.commonPool();
42     var start = System.currentTimeMillis();
43     fjPool.invoke(new ParallelMergeSorter(array));
44     System.out.println("Time spent: " + (System.currentTimeMillis() - start) + " ms, for size " + size
45         + " with " + ForkJoinPool.getCommonPoolParallelism() + " threads");
46     if (!ArrayTools.isSorted(array))
47         System.err.println("Array not sorted");
48     else System.out.println("Array is sorted");
49 }
```

Рисунок 4.6. – Метод для тестування з замірами часу з використанням commonPool()

Результати швидкодії представлено на графіку в таблиці 4.1:

Таблиця 4.1. – Час виконання паралельного алгоритму сортування злиттям в залежності від розміру масиву

Розмір	Час виконання
25000	33
50000	26
100000	40
500000	41
1000000	89
5000000	339
10000000	632
15000000	843
20000000	1253
30000000	1699
50000000	2967
100000000	6864
150000000	9685

## 5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

Дослідження послідовного та паралельного алгоритмів виконувались на апаратному забезпеченні з наступними характеристиками:

- Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 2.50 GHz (4 фізичних та 8 логічних ядер)
- Обсяг оперативної пам'яті: 16 Gb

Для встановлення ефективності послідовного та паралельного алгоритмів було виконано декілька експериментів, результати яких представлено в таблиці 5.1:

Таблиця 5.1. – Виміри часу для послідовного та паралельного алгоритму з різною кількістю потоків.

Кількість елементів, тис.	Послідовний алгоритм, мілісекунд	Паралельний алгоритм (p=2), мілісекунд	Паралельний алгоритм (p=4), мілісекунд	Паралельний алгоритм (p=6), мілісекунд	Паралельний алгоритм (p=7), мілісекунд	Паралельний алгоритм (p=8), мілісекунд	Паралельний алгоритм (p=16), мілісекунд
25	8	40	34	35	43	41	34
50	14	39	37	21	40	45	36
100	14	42	49	25	39	23	26
500	81	93	52	39	35	57	38
1000	170	168	78	73	75	64	57
5000	670	522	355	337	313	302	312
10000	1139	920	628	671	528	563	654
15000	1704	1433	943	858	886	777	768
20000	2239	1795	1358	1152	1228	1075	1213
30000	3431	2799	2097	1846	1676	1800	1685
50000	5816	5496	3411	3240	3133	2960	2902
100000	12017	10185	7202	6664	6124	5724	5834

Відповідно до таблиці 5.1. був створений графік, для візуалізації часу витраченого на виконання алгоритму, в залежності від кількості елементів та потоків.

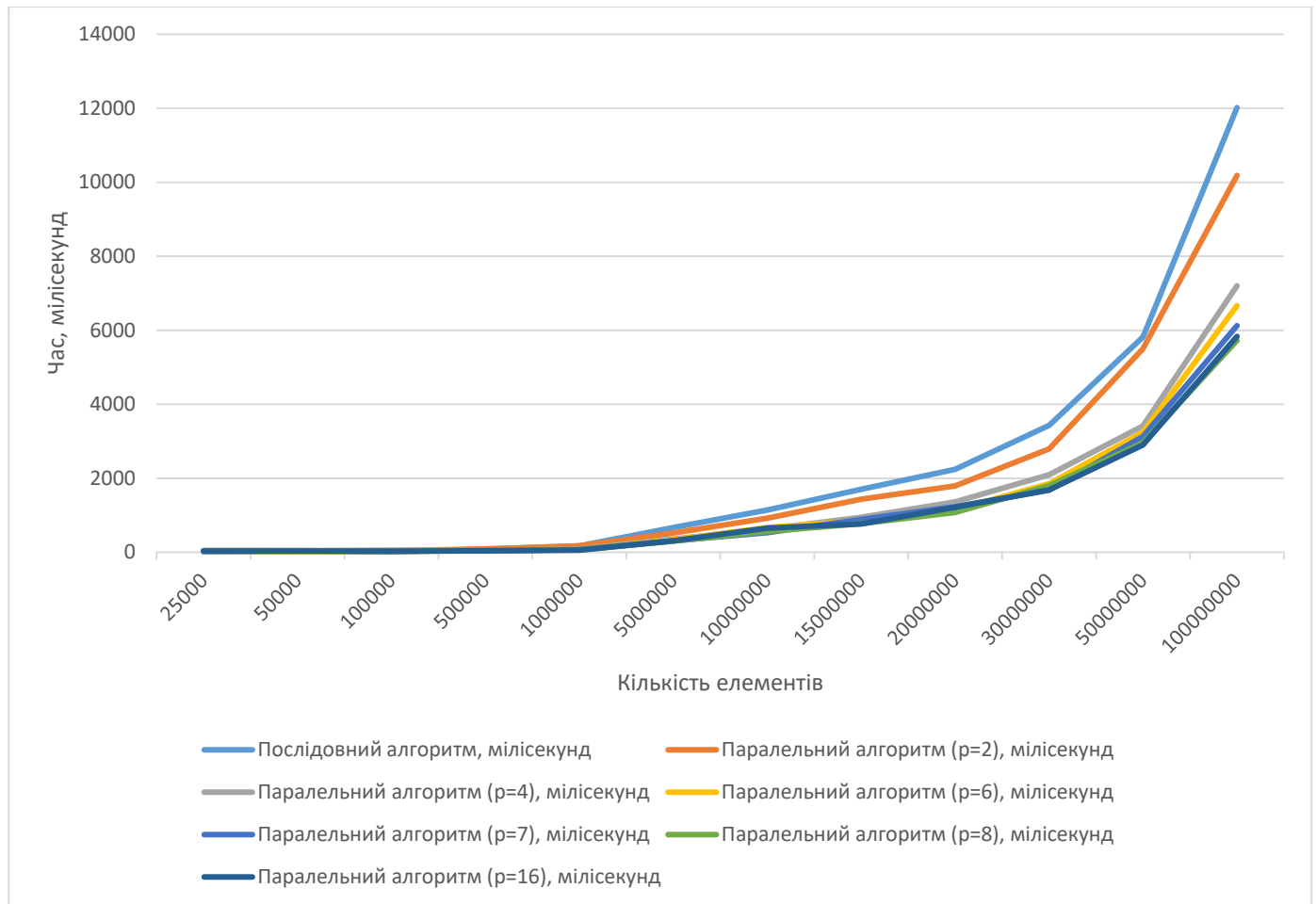


Рисунок 5.1. – Графік залежності часу виконання від розмірів масиву.

З графіку можна помітити, що зі збільшенням кількості елементів паралельний алгоритм має часову перевагу над послідовним алгоритмом при будь-якій кількості потоків.

Проаналізуємо прискорення роботи алгоритму відносно його послідовної реалізації, дані подано в таблиці 5.2.

Таблиця 5.2. – Прискорення паралельного алгоритму.

Кількість елементів, тис.	Кількість елементів	Прискорення паралельного алгоритму (p=2)	Прискорення паралельного алгоритму (p=4)	Прискорення паралельного алгоритму (p=6)	Прискорення паралельного алгоритму (p=7)	Прискорення паралельного алгоритму (p=8)	Прискорення паралельного алгоритму (p=16)
25	25000	0,2	0,2353	0,2286	0,186	0,1951	0,2353
50	50000	0,3589	0,3784	0,6667	0,35	0,3111	0,3889
100	100000	0,3333	0,2857	0,56	0,359	0,6087	0,5385
500	500000	0,8709	1,5577	2,0769	2,3143	1,4211	2,1316
1000	1000000	1,0119	2,1795	2,3288	2,2667	2,6563	2,9825
5000	5000000	1,2835	1,8873	1,9881	2,1406	2,2185	2,1474
10000	10000000	1,2380	1,8137	1,6975	2,1572	2,0231	1,7416
15000	15000000	1,1891	1,807	1,986	1,9233	2,1931	2,2188
20000	20000000	1,2473	1,6487	1,9436	1,8233	2,0828	1,8458
30000	30000000	1,2257	1,6361	1,8586	2,0471	1,9061	2,0362
50000	50000000	1,0582	1,7051	1,7951	1,8564	1,9649	2,0041
100000	100000000	1,1798	1,6686	1,8033	1,9623	2,0994	2,0598

Згідно з визначенням, прискорення визначається з формулою  $S_p = \frac{T_s}{T_p}$

(відношення часу послідовного алгоритму до часу паралельного). Аналізуючи таблицю помітно що послідовний алгоритм виконувався швидше для відносно невеликих за обсягом масивів (менше 100 тис.). На масивах більших за обсягом вже явно помітне прискорення.

В паралельній реалізації помітно що алгоритм має найвище прискорення при 4,6,7 та 8 потоках, а при 16 потоках прискорення вже знижується. Це



пов'язано з тим що машина, на якій виконувались експерименти не має необхідну кількість ядр.

Отже, можна дійти висновку, що при розмірі вхідних даних більше ста тисяч, паралельна реалізація має швидкісну перевагу над його послідовною версією.

## ВИСНОВКИ

У ході виконання даної курсової роботи було розкрито тему актуальності та переваг паралельних обчислень у сучасному світі, де більшість систем підтримують багатопоточність. Використанням багатопоточності дозволяє повноцінно використовувати ресурси сучасних комп'ютерів, розподіляючи обчислювання між багатьма обчислювальними ресурсами системи.

У якості інструменту для реалізації алгоритму була використана мова Java та Fork/Join Framework, який є потужним інструментом для роботи з багатопоточністю. Він надає зручні засоби для розподілу роботи між потоками, автоматичного збору результатів та синхронізації потоків в паралельному середовищі.

У роботі була розроблена ефективна паралельна реалізація алгоритму сортування злиттям за допомогою Fork/Join Framework. Результати експериментів показали, що паралельна версія алгоритму суттєво прискорює сортування для великих масивів даних. Продуктивність залежить від кількості доступних процесорних ядер і розміру вхідних даних.

Паралельний алгоритм обчислює задачу швидше при обсязі вхідних даних більше ста тисяч. Найкращих часових результатів було досягнути при використанні від 4 до 8 потоків. Це пов'язано з тим що машина, на якій виконувались обчислення має саме 4 фізичних ядра та 8 логічних.

Отже, в ході експериментів було показано переваги використання потенціалу багатопоточних реалізацій.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Katajainen, Jyrki; Pasanen, Tomi; Teuhola, Jukka (1996). "Practical in-place mergesort".
2. GeekForGeeks Merge Sort Algorithm  
URL: <https://www.geeksforgeeks.org/merge-sort/>
3. Parallel Merge Sort Cristopher Zelenka  
URL: <https://www.sjsu.edu/people/robert.chun/courses/cs159/s3/T.pdf>
4. Guide to the Fork/Join Framework in Java on Baeldung  
URL: <https://www.baeldung.com/java-fork-join>
5. Fork/Join (The Java™ Tutorials > Essential Java Classes > Concurrency) (oracle.com)  
URL:  
<https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html>
6. Parallel Merge Sort Algorithm on OpenGenus  
URL: <https://iq.opengenus.org/parallel-merge-sort/>
7. Merge Sort pseudo code and its implementation in C++  
URL: <https://www.codingninjas.com/codestudio/library/merge-sort-pseudocode-in-cc>

## ДОДАТКИ

### Додаток А. Код програми

GitHub репозиторій: <https://github.com/antonchaban/MultiMergeCW>

Клас SeqMergeSorter для послідовної реалізації алгоритму:

```
package kpi.fict.chaban.sequential;

public class SeqMergeSorter {

    public void mergeSort(int[] array) {
        var size = array.length;

        if (size < 2) {
            return;
        }

        var mid = size / 2;
        var left = new int[mid];
        var right = new int[size - mid];

        for (int i = 0; i < mid; i++) {
            left[i] = array[i];
        }
        for (int i = mid; i < size; i++) {
            right[i - mid] = array[i];
        }

        mergeSort(left);
        mergeSort(right);

        merge(array, left, right);
    }

    public static void merge(int[] arr, int[] left, int[] right) {
        int i = 0, j = 0, k = 0;
        while (i < left.length && j < right.length) {
            if (left[i] < right[j])
                arr[k++] = left[i++];
            else
                arr[k++] = right[j++];
        }
        while (i < left.length) {
            arr[k++] = left[i++];
        }
        while (j < right.length) {
            arr[k++] = right[j++];
        }
    }
}
```

Клас ParallelMergeSorter – паралельна реалізація алгоритму:

```

package kpi.fict.chaban.parallel;

import lombok.AllArgsConstructor;

import java.util.Arrays;
import java.util.concurrent.RecursiveAction;

@AllArgsConstructor
public class ParallelMergeSorter extends RecursiveAction {
    private int[] arr;

    @Override
    protected void compute() {
        var size = arr.length;

        if (size < 2) {
            return;
        }
        var mid = size / 2;

        var left = new int[mid];
        System.arraycopy(arr, 0, left, 0, mid);

        var right = new int[arr.length - mid];
        System.arraycopy(arr, mid, right, 0, arr.length - mid);

        invokeAll(new ParallelMergeSorter(left), new
ParallelMergeSorter(right));
        merge(left, right);
    }

    private void merge(int[] left, int[] right) {
        int i = 0, j = 0, k = 0;
        while (i < left.length && j < right.length) {
            if (left[i] < right[j])
                arr[k++] = left[i++];
            else
                arr[k++] = right[j++];
        }
        while (i < left.length) {
            arr[k++] = left[i++];
        }
        while (j < right.length) {
            arr[k++] = right[j++];
        }
    }
}

```

Класи для тестування алгоритмів:

```

package kpi.fict.chaban.tools;

import lombok.Getter;
import lombok.RequiredArgsConstructor;
import lombok.Setter;

```

```

@Getter
@Setter
@RequiredArgsConstructor
public class ArrayTools {
    private final int size;
    private int[] array;

    public int[] generate() {
        array = new int[size];
        for (int i = 0; i < size; i++) {
            array[i] = (int) (Math.random() * 100);
        }
        return array;
    }

    public static boolean isSorted(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            if (arr[i] > arr[i + 1])
                return false;
        }
        return true;
    }

    public static void printArray(int[] array) {
        for (int j : array) {
            System.out.printf("%d ", j);
        }
        System.out.println();
    }
}

package kpi.fict.chaban.tests.parallel;

import kpi.fict.chaban.parallel.ParallelMergeSorter;
import kpi.fict.chaban.sequential.SeqMergeSorter;
import kpi.fict.chaban.tools.ArrayTools;

import java.util.concurrent.ForkJoinPool;

public class ParallelTest {
    public static void main(String[] args) {
        // checkCorrectness();
        measureTime();
    }

    public static void checkCorrectness() {
        var arrayGenerator = new ArrayTools(20);
        var array = arrayGenerator.generate();
        System.out.println("Array to sort:");
        ArrayTools.printArray(array);
        System.out.println("#####");
        var fjPool = ForkJoinPool.commonPool();
        fjPool.invoke(new ParallelMergeSorter(array));
        ArrayTools.printArray(array);
        if (!ArrayTools.isSorted(array))
            System.err.println("Array not sorted");
    }
}

```

```

        else System.out.println("Array is sorted");
    }

    public static void measureTime() {
        var SIZES = new int[]{25_000, 50_000, 100_000, 500_000, 1_000_000,
5_000_000, 10_000_000, 15_000_000, 20_000_000
        , 30_000_000, 50_000_000, 100_000_000, 150_000_000};
        for (int size : SIZES) {
            checkTime(size);
        }
    }

    private static void checkTime(int size) {
        var arrayGenerator = new ArrayTools(size);
        var array = arrayGenerator.generate();
        var fjPool = ForkJoinPool.commonPool();
        var start = System.currentTimeMillis();
        fjPool.invoke(new ParallelMergeSorter(array));
        System.out.println("Time spent: " + (System.currentTimeMillis() - start)
+ " ms, for size " + size
        + " with " + ForkJoinPool.getCommonPoolParallelism() + "
threads");
        if (!ArrayTools.isSorted(array))
            System.err.println("Array not sorted");
        else System.out.println("Array is sorted");
    }
}

package kpi.fict.chaban.tests.sequential;

import kpi.fict.chaban.sequential.SeqMergeSorter;
import kpi.fict.chaban.tools.ArrayTools;

public class SeqTest {
    public static void main(String[] args) {
        //      checkCorrectness();
        measureTime();
    }

    public static void checkCorrectness() {
        var arrayGenerator = new ArrayTools(100);
        var array = arrayGenerator.generate();
        System.out.println("Array to sort:");
        ArrayTools.printArray(array);
        System.out.println("#####");
        var seqCalc = new SeqMergeSorter();
        seqCalc.mergeSort(array);
        ArrayTools.printArray(array);
        if (!ArrayTools.isSorted(array))
            System.err.println("Array not sorted");
        else System.out.println("Array is sorted");
    }

    public static void measureTime() {

```

```

        var SIZES = new int[]{25_000, 50_000, 100_000, 500_000, 1_000_000,
5_000_000, 10_000_000, 15_000_000, 20_000_000
        , 30_000_000, 50_000_000, 100_000_000, 150_000_000};
        for (int size : SIZES) {
            checkTime(size);
        }

        private static void checkTime(int size) {
            var arrayGenerator = new ArrayTools(size);
            var array = arrayGenerator.generate();
            var seqCalc = new SeqMergeSorter();
            var start = System.currentTimeMillis();
            seqCalc.mergeSort(array);
            System.out.println("Time spent: " + (System.currentTimeMillis() - start)
+ " ms, for size " + size);
            if (!ArrayTools.isSorted(array))
                System.err.println("Array not sorted");
            else System.out.println("Array is sorted");
        }
    }
}

```

```

package kpi.fict.chaban.tests;

import kpi.fict.chaban.parallel.ParallelMergeSorter;
import kpi.fict.chaban.sequential.SeqMergeSorter;
import kpi.fict.chaban.tools.ArrayTools;

import java.util.concurrent.ForkJoinPool;

public class MultiTest {
    public static void main(String[] args) {
        compareSpeed();
    }

    public static void compareSpeed() {
        var SIZES = new int[]{25_000, 50_000, 100_000, 500_000, 1_000_000,
5_000_000, 10_000_000, 15_000_000, 20_000_000
        , 30_000_000, 50_000_000, 100_000_000};
        for (int size : SIZES) {
            compareSpeed(size);
        }
    }

    private static void compareSpeed(int size) {
        var arrayGenerator = new ArrayTools(size);
        var arraySeq = arrayGenerator.generate();
        var arrayPar = new int[arraySeq.length];
        System.arraycopy(arraySeq, 0, arrayPar, 0, arraySeq.length);

        System.out.println("#####");
        var start = System.currentTimeMillis();
        var seqCalc = new SeqMergeSorter();
        seqCalc.mergeSort(arraySeq);
    }
}

```



```
        System.out.println("Time taken seq: " + (System.currentTimeMillis() -
start) + " ms, for size " + size);
        start = System.currentTimeMillis();
        var fjPool = new ForkJoinPool(16);
        fjPool.invoke(new ParallelMergeSorter(arrayPar));
        System.out.println("Time taken par: " + (System.currentTimeMillis() -
start) + " ms, for size " + size
                        + " with " + fjPool.getParallelism() + " threads");
    }
}
```

## Додаток Б. Тестування на правильність роботи

Тестування на невеликих обсягах даних для перевірки коректності алгоритму.

Тестування послідовного алгоритму:

```
C:\Users\anton\.jdk\corretto-17.0.6\bin\java.exe -javaagent:C:\Users\anton\.jdk\corretto-17.0.6\bin\javaagent.jar
Array to sort:
44 86 29 3 90 69 42 49 16 65 54 82 4 93 15 86 76 93 58 98
#####
3 4 15 16 29 42 44 49 54 58 65 69 76 82 86 86 90 93 93 98
Array is sorted

Process finished with exit code 0
```

Тестування паралельного алгоритму:

```
C:\Users\anton\.jdk\corretto-17.0.6\bin\java.exe -javaagent:C:\Users\anton\.jdk\corretto-17.0.6\bin\javaagent.jar
Array to sort:
61 75 1 14 54 34 43 0 10 42 90 5 0 17 50 36 30 80 53 29
#####
0 0 1 5 10 14 17 29 30 34 36 42 43 50 53 54 61 75 80 90
Array is sorted

Process finished with exit code 0
```