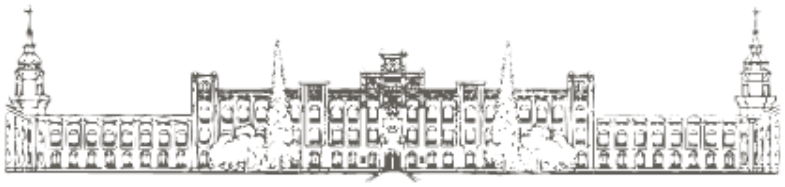

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту
«Технології паралельних обчислень. Курсова робота»
Тема: Алгоритм сортування злиттям та його паралельна реалізація мовою програмування Java

Керівник:
ст. викладач
Дифучин Антон Юрійович

Виконавець:
Бондаренко Максим Вікторович
студент групи ІП-13
залікова книжка № ІП-1304

«Допущено до захисту»
_____

«___» _____ 2024 р.

Захищено з оцінкою
_____

Члени комісії:

_____ Інна СТЕЦЕНКО

_____ Антон ДИФУЧИН

Київ – 2024

ЗАВДАННЯ

1. Виконати огляд існуючих реалізацій алгоритму, послідовних та паралельних, з відповідними посиланнями на джерела інформації (статті, книги, електронні ресурси). Зробити висновок про актуальність дослідження.
2. Виконати розробку послідовного алгоритму у відповідності до варіанту завдання та обраного програмного забезпечення для реалізації. Опис алгоритму забезпечити у вигляді псевдокоду. Провести тестування алгоритму та зробити висновок про коректність розробленого алгоритму. Дослідити швидкодію алгоритму при зростанні складності обчислень та зробити висновки про необхідність паралельної реалізації алгоритму.
3. Виконати розробку паралельного алгоритму у відповідності до обраного завдання та обраного програмного забезпечення для реалізації. Опис алгоритму забезпечити у вигляді псевдокоду. Забезпечити ініціалізацію даних при будь-якому великому заданому параметрі кількості даних.
4. Виконати тестування алгоритму, що доводить коректність результатів обчислень. Тестування алгоритму обов’язково проводити на великій кількості даних. Коректність перевіряти порівнянням з результатами послідовного алгоритму.
5. Виконати дослідження швидкодії алгоритму при зростанні кількості даних для обчислень.
6. Виконати експериментальне дослідження прискорення розробленого алгоритму при зростанні кількості даних для обчислень. Реалізація алгоритму вважається успішною, якщо прискорення не менше 1,2.
7. Дослідити вплив параметрів паралельного алгоритму на отримуване прискорення. Один з таких параметрів – це кількість підзадач, на які поділена задача при розпаралелюванні її виконання.
8. Зробити висновки про переваги паралельної реалізації обчислень для алгоритму, що розглядається у курсовій роботі, та програмних засобів, які використовувались.

АНОТАЦІЯ

Структура та обсяг роботи. Пояснювальна записка курсової роботи складається з 34 сторінок, 5 розділів, містить 11 рисунків, 4 таблиці, 1 додаток, 5 джерел.

Мета. Реалізувати та проаналізувати паралельну реалізацію алгоритму сортування злиттям на мові Java з використанням ForkJoin Framework, що допоможе оптимізувати роботу алгоритму у виконанні задачі сортування.

Робота складається з п’яти розділів, в яких описується алгоритм та його відомі паралельні реалізації, розробляється послідовний алгоритм та аналізується його швидкодія, обирається та описується програмне забезпечення для розробки паралельних обчислень, розробляється паралельний алгоритм з використанням обраного ПЗ, досліджується ефективність паралельних обчислень алгоритму.

КЛЮЧОВІ СЛОВА: СОРТУВАННЯ ЗЛИТТЯМ, FORKJOIN FRAMEWORK, JAVA, ПАРАЛЕЛЬНИЙ АЛГОРИТМ

ЗМІСТ

ВСТУП

Алгоритми сортування лежать в основі багатьох операцій з обробки даних. Одним із найпоширеніших алгоритмів є сортування злиттям, який характеризується своєю стабільністю та ефективністю. Даний алгоритм базується на концепції «розділяй і володарюй», бо він розділяє масиви на менші підмасиви, сортує їх окремо, а потім зливає в один відсортований масив. Стабільним та ефективним його робить часова складність $O(n\log n)$, яка зберігається у найгіршому, середньому та найкращому випадках.

Однак, навіть ефективні алгоритми потребують оптимізації, коли мова йде про серйозне збільшення розмірів даних та суворі вимоги до продуктивності. Ефективним рішенням даної проблеми є розпаралелення алгоритмів. Паралельні обчислення дозволяють розподіляти навантаження між кількома потоками, які виконують задачу одночасно, що сильно зменшує загальний час виконання задачі. У контексті нашої задачі розпаралелення може значно прискорити процес виконання, оскільки різні частини масиву можуть сортуватись одночасно.

У сучасному світі все більше комп’ютерів мають багатоядерні процесори, які здатні виконувати багато задач одночасно. Також великі обсяги даних стають дедалі частішою реальністю в сучасних додатках. Усім цим обумовлена доцільність розпаралелення алгоритму сортування злиттям, оскільки це допоможе використовувати ресурси більш ефективно, зменшуючи при цьому час виконання.

1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

1.1 Опис алгоритму

Алгоритм сортування злиттям (merge sort) є класичним алгоритм, що працює за принципом «розділяй і володарюй». Основна ідея полягає в рекурсивному поділі масиву на менші підмасиви до тих пір, поки кожен з них не буде уявляти з себе масив довжиною 1, після ці підмасиви зливаються назад в один відсортований масив. [1]

Алгоритм сортування складається з трьох основних етапів:

1. Розділення. Розбиття масиву на дві частини.
2. Рекурсивне сортування. Рекурсивно застосовуємо алгоритм до розділених частин.
3. Злиття. Зливаємо дві частини в один відсортований масив.

Нижче наведено псевдокод алгоритму сортування злиттям. Функція mergeSort(arr) – головна функція алгоритму, яка виконує всю його логіку. В якості вхідного параметру вона приймає масив arr. Якщо довжина масиву менша або дорівнює 1, то він повертається без змін, оскільки це є базовим випадком рекурсії. В іншому випадку масив розбивається на дві частини за індексом, який визначає середину масиву: $mid = \frac{len}{2}$, де mid – індекс середнього елементу, len – довжина масиву. Обидві частини сортуються рекурсивно за допомогою викликів відповідних методів mergeSort(arr[0:середина]) та mergeSort(arr[середина:довжина arr]). Після сортування частин вони зливаються в один відсортований масив за допомогою функції merge(left, right), яка приймає обидві частини в якості параметрів. Функція merge(left, right) – є допоміжною функцією для злиття двох відсортованих масивів в один масив. Вона почергово порівнює елементи обох масивів і додає до результуючого масиву менший елемент. Коли елементи одного масиву закінчуються, до результуючого масиву додаються всі елементи іншого, які залишились нерозподіленими.

Псевдокод алгоритму:

```
функція mergeSort(arr):
    якщо довжина arr ≤ 1:
        повернути arr
    mid = довжина arr / 2
    left = mergeSort(arr[0:середина])
    right = mergeSort(arr[середина:довжина arr])
    повернути merge(left, right)

функція merge(left, right):
    result = []
    i = 0, j = 0
    поки i < довжина left і j < довжина right:
        якщо left[i] ≤ right[j]:
            додати left[i] до result
            i = i + 1
        інакше:
            додати right[j] до result
            j = j + 1
    поки i < довжина left:
        додати left [i] до result
        i = i + 1
    поки j < довжина right:
        додати right[j] до result
        j = j + 1
    повернути result
```

Часова складність алгоритму становить $O(n\log n)$, що робить його достатньо ефективним для великих обсягів даних. Просторова складність становить $O(n)$, оскільки потрібен простір для злиття підмасивів. [1]

1.2 Опис відомих паралельних реалізацій алгоритму

Паралельні реалізації алгоритму сортування злиттям дозволяють прискорити сортування за рахунок розподілу навантаження між кількома процесорами або ядрами. Для виконання задач розпаралелювання даного алгоритму існує кілька підходів. Розглянемо основні відомі підходи до вирішення цієї задачі.

Бібліотека Fork/Join в Java дозволяє реалізовувати паралельні обчислення шляхом рекурсивного розподілу завдань на підзадачі. Кожна задача розбивається на дві підзадачі, які виконуються паралельно, після чого результати зливаються. [2]

Інший підхід до паралельного сортування полягає у використанні звичайних потоків. У цьому випадку для кожної рекурсивної задачі створюється новий потік, який виконує сортування підмасиву. [3]

Також існує підхід, який застосовується до розподілених систем з великою кількістю процесорів, з використанням MPI. У даному випадку різні процесори виконують сортування окремих частин масиву та обмінюються даними для злиття результатів.

2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

2.1 Розробка послідовного алгоритму

При розробці паралельного алгоритму за допомогою Fork/Join наявність окремої реалізації послідовного алгоритму є обов'язковою, оскільки при розділенні на підзадачі ми маємо порогове значення нижче якого розділення не може відбуватись, а підмасиви розміром менше цього значення будуть сортуватись послідовним сортуванням, оскільки розділення маленьких задач на підзадачі тільки сповільнить виконання роботи.

Нижче наведено код послідовного алгоритму сортування злиттям (рис. 2.1)

```
public static <T> void sequentialSort(T[] array, int left, int right, Comparator<T> comparator) {
    if (left < right) {
        int mid = (left + right) / 2;
        sequentialSort(array, left, mid, comparator);
        sequentialSort(array, left: mid + 1, right, comparator);
        merge(array, left, mid, right, comparator);
    }
}

2 usages  maksb
private static <T> void merge(T[] array, int left, int mid, int right, Comparator<T> comparator) {
    T[] temp = (T[]) new Object[right - left + 1];
    int i = left, j = mid + 1, k = 0;

    while (i <= mid && j <= right) {
        if (comparator.compare(array[i], array[j]) <= 0) {
            temp[k++] = array[i++];
        } else {
            temp[k++] = array[j++];
        }
    }

    while (i <= mid) {
        temp[k++] = array[i++];
    }

    while (j <= right) {
        temp[k++] = array[j++];
    }

    System.arraycopy(temp, srcPos: 0, array, left, temp.length);
}
```

Рисунок 2.1.1 – код послідовного алгоритму сортування злиттям.

Алгоритм починається з перевірки, щоб індекс початку масиву був меншим за індекс кінця, інакше це масив з одного елементу, який і так є відсортованим. Якщо індекс початку все-таки менше, то рекурсивно викликається сортування для двох підмасивів, на які було поділено початковий масив за індексом mid. Після сортування обидва підмасива зливаються у відсортований масив.

Злиття в свою чергу починається з виділення тимчасового масиву для збереження даних при злитті. Потім визначаємо індекси початку обох підмасивів та індекс для ітерації по тимчасовому масиву. Маємо цикл, який проходиться одночасно по двом підмасивам та порівнює між собою елементи цих підмасивів, менший з яких записуючи в результуючий масив, після чого здвигає індекси належним чином. Коли один з підмасивів повністю розподілиться, запишемо в тимчасовий масив усі елементи іншого підмасиву, які залишились. Вміст тимчасового масиву копіюється назад у вихідний масив.

2.2 Аналіз швидкодії послідовного алгоритму

Проведемо аналіз швидкодії послідовного алгоритму сортування злиттям. Для цього проведемо декілька замірів на різних розмірностях масиву та проаналізуємо отримані результати. У якості масиву для сортування буде виступати масив об’єктів будівель, а в якості компаратора буде використовуватись ціна кожної будівлі, що міститься в кожному окремому об’єкті.

Важливим етапом перед тестуванням є прогрівання, це допоможе отримати більш точні та стабільні результати вимірювань. Прогрівання дозволяє уникнути впливу різних факторів, які можуть спотворювати перші запуски програми, тому в нашому випадку перед проведенням замірів будемо обов’язково проводити прогрівання п’ятьма ітераціями запуску алгоритму.

Наступним етапом є проведення основного тестування, де ми маємо запустити наш алгоритм 20 разів на копіях одного й того самого масиву задля об’єктивності. На кожній ітерації відпрацювання програми отримуємо час виконання нашого алгоритму та інформацію про правильність сортування. У

підсумку час виконання усіх ітерацій складається та ділиться на кількість цих ітерація, завдяки чому ми отримуємо середній час виконання сортування одного масиву, що робить наші дані для експерименту більш об’єктивними. Нижче наведено приклад виконання подібного експерименту (рис. 2.2.1) та таблиця з результатами експериментів для різної кількості елементів масиву (табл. 2.2.1).

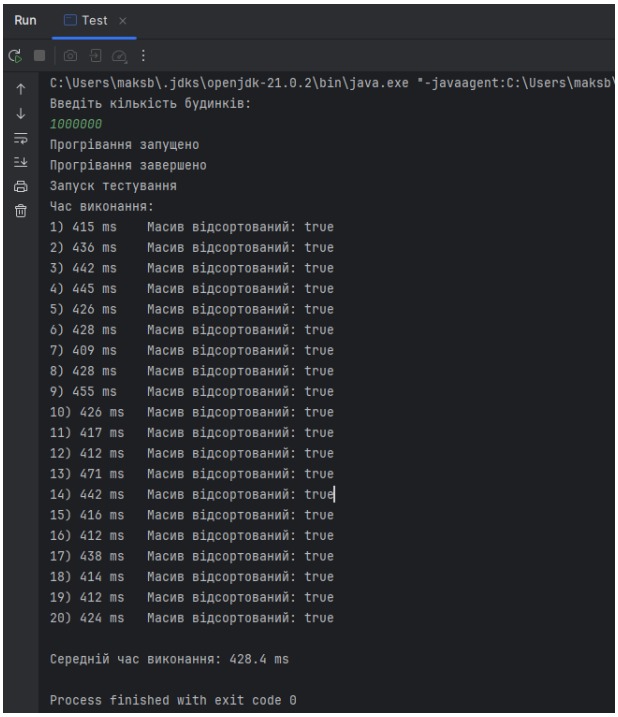

Рисунок 2.2.1 – Приклад обчислення середнього часу виконання.

Таблиця 2.2.1 – Результати експериментів для різної розмірності масивів.

Кількість елементів	Середній час послідовного алгоритму, мілісекунд
100000	28.05
250000	81.65
500000	191.15
1000000	428.4
1500000	713.65
2000000	966.7
3000000	1595.2
5000000	3094.5
10000000	6286.25

Судячи з результатів у таблиці бачимо очікуваний результат: час виконання зростає зі збільшенням розмірності масивів, але таблиці недостатньо для більш глибокого аналізу, тому побудуємо графіки, на яких буде більш наглядно видно, як змінюється швидкість виконання в залежності від розміру.

Для початку побудуємо графік залежності середнього часу виконання від кількості елементів (рис. 2.2.2).

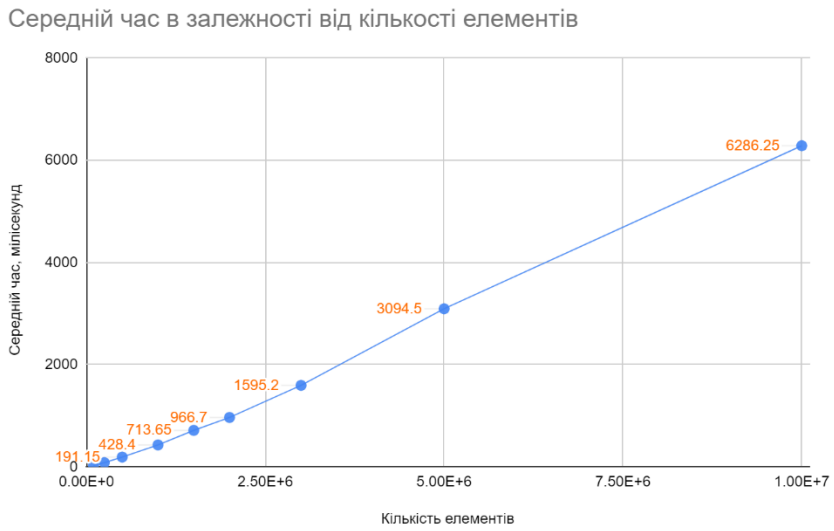

Рисунок 2.2.2 – Графік залежності середнього часу від розмірності масиву.

Графік демонструє, як змінюється час виконання послідовного алгоритму при збільшенні розміру вхідних даних. Одразу бачимо, що графік має лінійно-логарифмічну залежність, що відповідає зазначеній часовій складності алгоритма: $O(n\log n)$. З ростом кількості елементів час виконання зростає більше, ніж лінійно, але менше, ніж квадратично. На прикладі 1000000 та 2000000 елементів, розмір збільшився вдвічі, а час збільшився в 2.257 рази. Також на графіку не спостерігається значних відхилень чи аномалій, що свідчить про стабільну роботу алгоритму на різних обсягах даних та відсутність суттєвих затримок чи надмірних витрат на конкретних етапах сортування. На великих обсягах даних (10 мільйонів) час виконання алгоритму становить більше 6 секунд, що свідчить про потребу в розпаралеленні алгоритму для оптимізації роботи на великих розмірностях.

3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС

Для реалізації паралельних обчислень у рамках даної курсової роботи було обрано мову програмування Java та її технологію Fork/Join, яка надає інструменти для пришвидшення паралельних обчислень шляхом спроби використовувати всі доступні ядра процесора, що досягається за допомогою принципу «розділяй та володарюй», що ідеально підходить до нашого алгоритму.

Java є однією з найпопулярніших мов програмування, яка вирізняється стабільністю та активною підтримкою розробників з боку спільноти. До основних переваг можна віднести те, що програми можуть виконуватись на будь-якій платформі з встановленою JVM, Java забезпечує високий рівень безпеки і стабільності програми, має вбудовану підтримку багатопотоковості, що сильно допомагає в паралельних обчислення, має набір бібліотек, які допомагають у розробці додатків з паралельним обчисленням. [4]

Бібліотека Fork/Join є інструментом для реалізації паралельних обчислень за принципом «розділяй і володарюй». Дана бібліотека дозволяє розбивати задачі на менші підзадачі, які виконуються паралельно, що сприяє пришвидшенню виконання алгоритму на багатоядерних процесорах. З основних переваг бібліотеки можна виділити автоматичне управління потоками, оскільки Fork/Join сам керує пулом потоків, ефективне використання ресурсів, що забезпечує високу продуктивність. Алгоритм сортування злиттям ідеально підходить до паралельної реалізації за допомогою Fork/Join завдяки рекурсивній природі та принципу «розділяй і володарюй». [2]

В якості середовища розробки було обрано IntelliJ IDEA. Воно широко використовується для розробки програм на Java завдяки великій кількості інструментів, зручному інтерфейсу та підтримці великої кількості бібліотек. [5]

4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЄКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ

4.1 Проєктування

Оскільки ми використовуємо бібліотеку ForkJoin для паралельного виконання завдань, основною ідеєю є розділення задачі на менші підзадачі, які можна виконувати паралельно. Основними компонентами будуть масиви об’єктів, що будуть розділятись та зливатись, а також індекси, що вказують на границі підмасивів. При розробці паралельного алгоритму з використанням ForkJoin, стратегія паралельного виконання грає ключову роль в подальшій ефективності алгоритму.

Основні кроки для визначення стратегії виконання:

- Початковий масив даних буде розділятись на менші підмасиви, кожен з яких буде оброблятись як окрема задача. Для цього використаємо рекурсивний підхід, де кожна задача розділяється на дві половини до досягнення порогового розміру.
- Після розділення задач на менші підзадачі, підзадачі мають виконуватись паралельно. За паралельне виконання підзадач відповідає ForkJoinPool.
- Після завершення виконання підзадач необхідно злити їх результати в результуючий масив.

Також необхідно розглянути обробку базового випадку, коли розмір підмасиву стає досить малим, а його подальше ділення зменшує ефективність. У такому випадку до цього підмасиву застосовується послідовний алгоритм сортування.

4.2 Реалізація

Реалізація паралельного алгоритму є достатньо схожою на послідовний алгоритм. Тільки тепер ми маємо критичну зону, яка визначається не зменшенням підмасиву до одного елементу, а зменшенням до певного порогового значення, після якого масив сортується звичайним послідовним

сортуванням. Визначення центрального елементу, за яким ділиться масив, також не зазнало змін. Сам алгоритм виконується в методі compute(), який викликається при спробі виконати задачу. Тобто тепер при рекурсивному виклику сортування підмасивів, у нас кожен підмасив і є новою задачею для ForkJoin. Цими задачами в подальшому оперує ForkJoinPool, виконуючи їх паралельно в заданій кількості потоків. Після виконання підзадач викликається метод merge(), який об’єднує відсортовані масиви.

Отже, основною відмінністю паралельної реалізації є обгортка у вигляді ForkJoin, яка не просто розділяє масив на підмасиви, а надає ці підмасиви окремим підзадачам, які так само рекурсивно сортують їх, а виконанням цим задач керує ForkJoinPool.

Код паралельної реалізації алгоритму сортування злиттям наведено нижче (рис. 4.2.1).

```
@Override
protected void compute() {
    if (right - left < threshold) {
        sequentialSort(array, left, right, comparator);
    } else {
        int mid = (left + right) / 2;
        invokeAll(new ParallelMergeSort<>(array, left, mid, comparator),
                new ParallelMergeSort<>(array, left: mid + 1, right, comparator));

        merge(array, left, mid, right, comparator);
    }
}
```

Рисунок 4.2.1 – Паралельна реалізація алгоритму сортування злиттям.

Код створення пулу потоків та запуску виконання задачі наведено нижче (рис. 4.2.2).

```
ForkJoinPool pool = new ForkJoinPool(numberOfThreads);
ParallelMergeSort<Building> task = new ParallelMergeSort<>(buildings, left: 0, right: buildings.length - 1, priceComparator);

pool.invoke(task);
```

Рисунок 4.2.2 – Створення пулу потоків та запуск виконання задачі.

4.3 Тестування

Для тестування правильної роботи паралельної реалізації алгоритму сортування злиттям проведемо два експерименти, які схожий з тим, що ми робили з послідовним алгоритмом. Для першого експерименту ми будемо

генерувати різні набори даних для кожної ітерації виконання сортування. Це допоможе нам перевірити правильність самого сортування на різних масивах. При кожній ітерації будемо перевіряти порядок елементів у відсортованому масиві. Для даного експерименту маємо наступний код (рис. 4.3.1). На даному етапі розмірність та кількість потоків не важлива, оскільки ми просто перевіряємо правильність виконання алгоритму. Протестуємо правильність виконання на 30 ітераціях (рис. 4.3.2).

```
System.out.println("\n\nЗапуск тестування паралельного алгоритму");
long totalTimeParallel = 0;
System.out.println("Час виконання: ");
for (int i = 0; i < iterations; i++) {

    Building[] buildingsLocal = new Building[size];
    if (GENERATE_NEW_ARRAY) {
        for (int j = 0; j < size; j++) {
            buildingsLocal[j] = new Building( name: "Building" + j,  price: 3000 + Math.random() * 997000);
        }
        for (int j = 0; j < 10; j++) {
            System.out.println(buildingsLocal[j].getPrice());
        }
    } else {
        buildingsLocal = buildings.clone();
    }

    long startTime = System.currentTimeMillis();
    ForkJoinPool pool = new ForkJoinPool(numberOfThreads);
    ParallelMergeSort<Building> task = new ParallelMergeSort<>(buildingsLocal,  left: 0,  right: buildingsLocal.length - 1, priceComparator);
    pool.invoke(task);
    long endTime = System.currentTimeMillis();
    System.out.print(i + 1 + ") " + (endTime - startTime) + " ms");
    System.out.println("\t Масив відсортований: " + isSorted(buildingsLocal));
    totalTimeParallel += (endTime - startTime);
}
long averageTimeParallel = totalTimeParallel / iterations;
System.out.println("\nСередній час виконання паралельного алгоритму: " + averageTimeParallel + " ms");
```

Рисунок 4.3.1 – Код для тестування паралельної реалізації.

```
Введіть кількість будинків:
1000000


Запуск тестування паралельного алгоритму
Час виконання:
1) 273 ms    Масив відсортований: true
2) 199 ms    Масив відсортований: true
3) 184 ms    Масив відсортований: true
4) 233 ms    Масив відсортований: true
5) 199 ms    Масив відсортований: true
6) 175 ms    Масив відсортований: true
7) 174 ms    Масив відсортований: true
8) 184 ms    Масив відсортований: true
9) 220 ms    Масив відсортований: true
10) 166 ms   Масив відсортований: true
11) 194 ms   Масив відсортований: true
12) 177 ms   Масив відсортований: true
13) 291 ms   Масив відсортований: true
14) 197 ms   Масив відсортований: true
15) 180 ms   Масив відсортований: true
16) 173 ms   Масив відсортований: true
17) 170 ms   Масив відсортований: true
18) 195 ms   Масив відсортований: true
19) 228 ms   Масив відсортований: true
20) 166 ms   Масив відсортований: true
21) 242 ms   Масив відсортований: true
22) 183 ms   Масив відсортований: true
23) 249 ms   Масив відсортований: true
24) 170 ms   Масив відсортований: true
25) 225 ms   Масив відсортований: true
26) 184 ms   Масив відсортований: true
27) 203 ms   Масив відсортований: true
28) 175 ms   Масив відсортований: true
29) 185 ms   Масив відсортований: true
30) 217 ms   Масив відсортований: true

Середній час виконання паралельного алгоритму: 200 ms
```

Рисунок 4.3.2 – Результати тестування правильності роботи алгоритму.

З результатів бачимо, що в усіх 30 випадках масиви було успішно відсортовано, що свідчить про правильність роботи паралельної реалізації.

У другому експерименті протестуємо роботу алгоритму на однаковому масиві, тут нас вже цікавить середній час виконання на різних розмірностях. Для цього експерименту будемо використовувати той же код. Для об’єктивності результатів будемо обраховувати середній час з 20 ітерацій виконання програми та з попереднім прогріванням. Тестування будемо проводити на 6 потоках. Нижче наведено таблицю з результатами тестування (табл. 4.3.1).

Таблиця 4.3.1 – Результати тестування паралельного алгоритму.

Кількість елементів	Середній час паралельного алгоритму, мілісекунд
100000	10.05
250000	35.5
500000	81.05
1000000	181.6
1500000	302.6
2000000	420.35
3000000	650.1
5000000	1269.3
10000000	2623.75

З результатів бачимо ситуацію схожу на послідовний алгоритм, але швидкість виконання значно зросла в усіх випадках. Бачимо, що при збільшенні кількості елементів удвічі, середній час збільшується трохи більше, ніж удвічі. Це також свідчить про лінійно-логарифмічну залежність. Для кращого розуміння побудуємо графік залежності часу виконання від кількості елементів (рис. 4.3.3).

Рисунок 4.3.3 – Графік залежності часу виконання від кількості елементів масиву.

Цей графік є дуже схожим на графік залежності послідовного алгоритму, окрім швидкості виконання, яка тут значно вище. На самому графіку видно лінійно-логарифмічну функцію, яка відповідає часовій складності нашого алгоритму.

5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

5.1 Порівняння результатів паралельного та послідовного алгоритмів

Дослідження ефективності почнемо з порівняння результатів паралельного алгоритму з результатами послідовного. Для цього використаємо випробування, які було проведено раніше. Всі випробування проводились на ОС Windows 11 (x64), процесорі Intel Core i5-9400f з 6 ядрами та 6 потоками. Перед кожним випробуванням проводилось прогрівання, а самі випробування є результатом циклічного виконання алгоритму на одному й тому ж масиві 20 разів, що робить результати більш об’єктивними. У випадку з маленькими маленькими розмірностями можливі випробування з більшою кількістю ітерацій для збільшення точності результатів. Результати випробувань для визначення середнього часу виконання паралельного та послідовного алгоритмів наведено в таблиці нижче (табл. 5.1). Випробування для паралельного алгоритму проводились при використанні 6 потоків. Порогове значення для використання послідовного алгоритму становило 1000.

Формула обрахунку прискорення (5.1):

$$speedup = \frac{T_{seq}}{T_{parallel}} \quad (5.1)$$

Таблиця 5.1 – Порівняння часу виконання послідовного та паралельного алгоритмів.

Кількість елементів	Середній час паралельного алгоритму, мілісекунд	Середній час послідовного алгоритму, мілісекунд	Прискорення
100000	10.05	28.05	2.791044776
250000	35.5	81.65	2.3
500000	81.05	191.15	2.358420728
1000000	181.6	428.4	2.359030837
1500000	302.6	713.65	2.358393919
2000000	420.35	966.7	2.299750208
3000000	650.1	1595.2	2.453776342
5000000	1269.3	3094.5	2.43795793
10000000	2623.75	6286.25	2.395902811

З таблиці бачимо, що прискорення залишається однаково великим при всіх розмірностях. Більш наглядно можемо побачити це на графіку нижче (рис. 5.1). Це обумовлено тим, що ForkJoinPool ефективно балансує навантаження між потоками незалежно від розміру масиву. Кожен потік отримує приблизно рівну кількість роботи і час простою мінімізується. Навіть на невеликих масивах алгоритм забезпечує достатній рівень паралельності. Також ForkJoinPool мінімізує накладні витрати на управління потоками. Тому незалежно від розміру масиву, ці витрати залишаються малими відносно тієї вигоди, яку ми отримуємо від розпраралелення. Дуже важливим фактором є правильно підібране порогове значення, при якому починає застосовуватись послідовне сортування. Звісно, при зовсім маленьких розмірах паралельний алгоритм програє у швидкості. Так, наприклад, при розмірності 5000 прискорення становить 0.64.

Рисунок 5.1 – Графік залежності прискорення від розміру масиву.

Також маємо графік порівняння середнього часу виконання двох варіацій алгоритму (рис. 5.2). На даному графіку видно, що зі збільшенням кількості елментів в абсолютній величині час на виконання зростає менш стрімко. Так, наприклад, абсолютна різниця в часі між масивами розмірностями 5 та 10 мільйонів становить 1354.45 мс для паралельного алгоритму та 3191.75 мс для послідовного, це обумовленно загальним прискоренням роботи алгоритму.

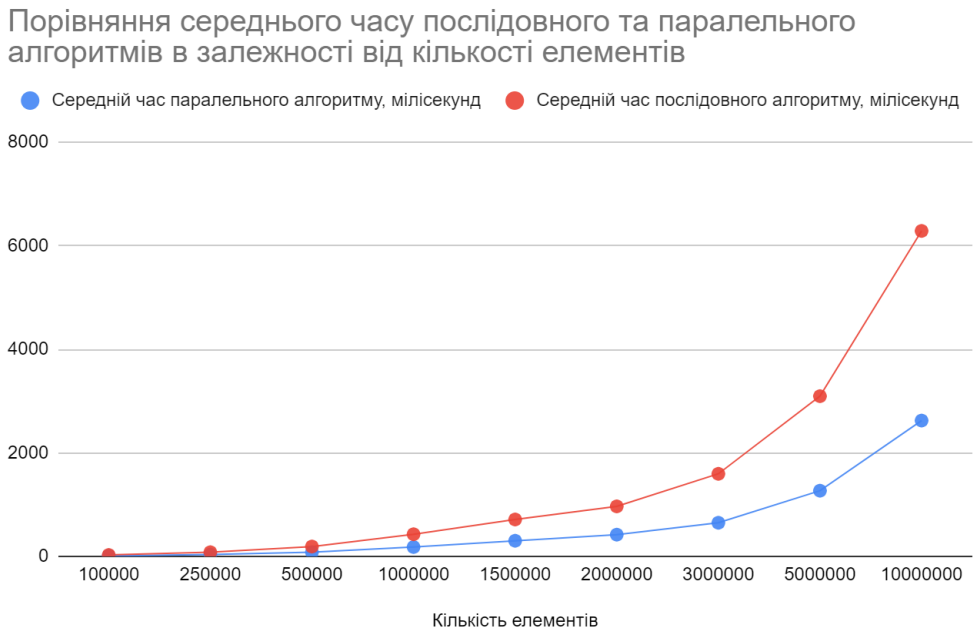

Рисунок 5.2 – Графік порівняння середнього часу виконання двох варіацій алгоритмів в залежності від кількості елементів.

5.2 Вплив кількості потоків на ефективність

Важливим параметром для дослідження є кількість потоків. Дослідимо вплив цього параметру на швидкість роботи паралельного алгоритму. Випробування буде проводитись в тих же умовах, що і раніше.

Таблиця 5.2.1 – Таблиця з результатами виконання паралельного алгоритму на різній кількості потоків.

Кількість елементів	Кількість потоків			
	2	4	8	16
100000	16.8	11.65	9.75	10.2
250000	49.6	37.1	36.7	34.15
500000	116.9	87.6	81.65	78.9
1000000	262	199.65	180.7	180.85
1500000	424.15	319.2	294.8	286.55
2000000	588.85	440.2	406.6	396.95
3000000	954.3	703.1	644.05	627.85
5000000	1702.1	1248.2	1184.8	1157.15
10000000	3737.2	2752.7	2617.35	2459.05

Майже в усіх випадках, при збільшенні кількості потоків, час виконання зменшується, що свідчить про ефективність паралельної обробки даних. Збільшення потоків з 2 до 4 дає значне пришвидшення алгоритму, подальше збільшення потоків також пришвидшує виконання, але вже не так сильно, це вказує на досягнення межі ефективності паралелізації. Для малих масивів різниця в часі між різною кількістю потоків значно менша, ніж для великих, що вказує на те, що для великих обсягів даних паралелізація дає значно більший приріст продуктивності. Для наглядності нижче наведено графік залежності часу виконання від кількості елементів масиву для різних кількостей потоків (рис. 5.2.1).

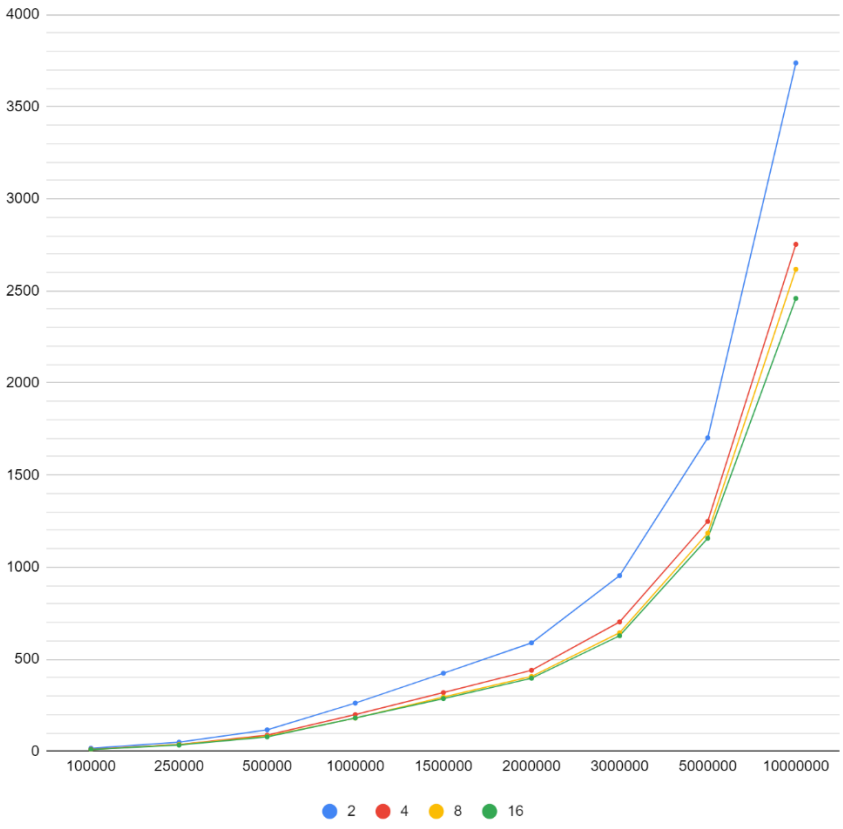

Рисунок 5.2.1 – Графік залежності швидкості виконання від кількості елементів по кількості потоків.

ВИСНОВКИ

У даній роботі було розглянуто та проаналізовано паралельну реалізацію алгоритму сортування злиттям мовою Java. Для початку було описано сам алгоритм та його відомі паралельні реалізації. Це дозволило нам обрати найкращу реалізацію у вигляді Fork/Join бібліотеки, які ідеально підходить для вирішення подібних задач.

Було розроблено та проаналізовано послідовний алгоритм, за результатами роботи якого було виявлено потребу в його оптимізації шляхом розпаралелення.

Також було проаналізовано інструменти для розробки алгоритму. Мова Java виявилась достатньо ефективною для виконання подібних задач, також вона містить бібліотеку ForkJoin, яка і допомогла нам легко реалізувати ефективний паралельний алгоритм.

Процес розробки паралельного алгоритму не був складним через легкість використання ForkJoin, який дозволяє гарно розпаралелити рекурсивний алгоритм. Алгоритм було протестовано та результати виконання допомогли нам в аналізі ефективності алгоритму. Алгоритм навіть без належного аналізу явно показав значно кращі результати.

Для аналізу ефективності було порівняно час виконання паралельного та послідовного алгоритмів, також обраховано прискорення, яке алгоритм дає. Прискорення залишалось однаково високим для всіх розмірностей масивів, окрім прям сильно маленьких масивів, які розбивались всього на декілька підзадач, там паралелізація була недоречною. Також було проаналізовано вплив кількості потоків на швидкодію алгоритму. Для масивів великих розмірів найкращим варіантом є використання більшої кількості потоків. Так, наприклад, 16 потоків показали найкращий результат на найбільших масивах. На найменшому масиві розміром 100000 найбільш ефективним виявилось 8 потоків.

Отже, розпаралелювання значно оптимізувало роботу алгоритму сортування злиттям, видаючи середнє прискорення в районі 2.3, що є дуже гарним результатом.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Merge Sort Algorithm. URL: https://www.javatpoint.com/merge-sort (дата звернення: 19.05.2024)
2. Guide to the Fork/Join Framework in Java. URL: https://www.baeldung.com/java-fork-join. (дата звернення: 21.05.2024)
3. Parallel Merge Sort Algorithm. URL: https://rachitvasudeva.medium.com/parallel-merge-sort-algorithm-e8175ab60e7 (дата звернення: 21.05.2024)
4. Java Documentation. URL: https://docs.oracle.com/en/java/ (дата звернення: 20.05.2024)
5. IntelliJ IDEA The Leading Java and Kotlin IDE. URL: https://www.jetbrains.com/idea/ (дата звернення: 20.05.2024)

ДОДАТКИ

Додаток А. Лістинг програмного коду

Main.java

```
import java.util.Comparator;
import java.util.concurrent.ForkJoinPool;

public class Main {
    public static void main(String[] args) {
        int size = 1000000;
        int numberOfThreads = 8;



        Building[] buildings = new Building[size];

        for (int i = 0; i < size; i++) {
            buildings[i] = new Building("Building" + i, 3000 + Math.random() * 997000);
        }

        Building[] buildings2 = buildings.clone();

        Comparator<Building> priceComparator =
Comparator.comparingDouble(Building::getPrice);



        long startTime2 = System.currentTimeMillis();
        ParallelMergeSort.sequentialSort(buildings2, 0, buildings2.length - 1,
priceComparator);
        long endTime2 = System.currentTimeMillis();
```

```
        System.out.println("Час виконання послідовного сортування: " + (endTime2 - startTime2) + " ms");



        long startTime = System.currentTimeMillis();
        ForkJoinPool pool = new ForkJoinPool(numberOfThreads);
        ParallelMergeSort<Building> task = new ParallelMergeSort<>(buildings, 0, buildings.length - 1, priceComparator);

        pool.invoke(task);
        long endTime = System.currentTimeMillis();

        System.out.println("Час виконання паралельного сортування: " + (endTime - startTime) + " ms");

        System.out.println("Масив відсортований: " + Test.isSorted(buildings));

        System.out.println("Масиви рівні: " + Test.areEqual(buildings, buildings2));

        System.out.println("Speedup: " + (double)(endTime2 - startTime2) / (endTime - startTime));
    }
}
```

Building.java

```
public class Building {
    private String name;
    private double price;

    public Building(String name, double price) {
```

```
        this.name = name;
        this.price = price;
    }

    public double getPrice() {
        return price;
    }
}
```

ParallelMergeSort.java

```
import java.util.Comparator;
import java.util.concurrent.RecursiveAction;

public class ParallelMergeSort<T> extends RecursiveAction {
    private T[] array;
    private int left;
    private int right;
    private Comparator<T> comparator;
    private int threshold = 1000;

    public ParallelMergeSort(T[] array, int left, int right, Comparator<T> comparator)
{
        this.array = array;
        this.left = left;
        this.right = right;
        this.comparator = comparator;
    }

    @Override
    protected void compute() {
```

```
        if (right - left < threshold) {
            sequentialSort(array, left, right, comparator);
        } else {
            int mid = (left + right) / 2;
            invokeAll(new ParallelMergeSort<>(array, left, mid, comparator),
                    new ParallelMergeSort<>(array, mid + 1, right, comparator));

            merge(array, left, mid, right, comparator);
        }
    }

    public static <T> void sequentialSort(T[] array, int left, int right, Comparator<T>
comparator) {
        if (left < right) {
            int mid = (left + right) / 2;
            sequentialSort(array, left, mid, comparator);
            sequentialSort(array, mid + 1, right, comparator);
            merge(array, left, mid, right, comparator);
        }
    }

    private static <T> void merge(T[] array, int left, int mid, int right, Comparator<T>
comparator) {
        T[] temp = (T[]) new Object[right - left + 1];
        int i = left, j = mid + 1, k = 0;

        while (i <= mid && j <= right) {
            if (comparator.compare(array[i], array[j]) <= 0) {
                temp[k++] = array[i++];
            } else {
```

```
            temp[k++] = array[j++];
        }
    }

    while (i <= mid) {
        temp[k++] = array[i++];
    }

    while (j <= right) {
        temp[k++] = array[j++];
    }

    System.arraycopy(temp, 0, array, left, temp.length);
  }
}
```

Test.java

```
import java.util.Comparator;
import java.util.Random;
import java.util.Scanner;
import java.util.concurrent.ForkJoinPool;

public class Test {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    System.out.println("Введіть кількість будинків: ");
    int size = scanner.nextInt();

    int iterations = 20;
```

```
        int warmupIterations = 5;
        int numberOfThreads = 8;
        boolean GENERATE_NEW_ARRAY = false;

        Building[] buildings = new Building[size];
        Random random = new Random();

        for (int i = 0; i < size; i++) {
            buildings[i] = new Building("Building" + i, 3000 + Math.random() *
997000);
        }

        Comparator<Building> priceComparator =
Comparator.comparingDouble(Building::getPrice);

        System.out.println("Прогрівання запущено");
        for (int i = 0; i < warmupIterations; i++) {
            Building[] buildingsCopy = buildings.clone();
            Building[] buildingsCopyParallel = buildings.clone();

            ParallelMergeSort.sequentialSort(buildingsCopy, 0, buildingsCopy.length - 1,
priceComparator);

            ForkJoinPool pool = new ForkJoinPool(numberOfThreads);
            ParallelMergeSort<Building> task = new
ParallelMergeSort<>(buildingsCopyParallel, 0, buildingsCopyParallel.length - 1,
priceComparator);
            pool.invoke(task);
        }
        System.out.println("Прогрівання завершено");
```

```
        System.out.println("\n\nЗапуск тестування послідовного алгоритму");
        long totalTime = 0;
        System.out.println("Час виконання: ");
        for (int i = 0; i < iterations; i++) {
            Building[] buildingsLocal = buildings.clone();
            long startTime = System.currentTimeMillis();
                ParallelMergeSort.sequentialSort(buildingsLocal, 0, buildingsLocal.length
- 1, priceComparator);
            long endTime = System.currentTimeMillis();
            System.out.print(i + 1 + ") " + (endTime - startTime) + " ms");
            System.out.println("\t Масив відсортований: " + isSorted(buildingsLocal));
            totalTime += (endTime - startTime);
        }
        double averageTime = (double) totalTime / iterations;
        System.out.println("\nСередній час виконання послідовного алгоритму: " +
averageTime + " ms");



        System.out.println("\n\nЗапуск тестування паралельного алгоритму");
        long totalTimeParallel = 0;
        System.out.println("Час виконання: ");
        for (int i = 0; i < iterations; i++) {

            Building[] buildingsLocal = new Building[size];
            if (GENERATE_NEW_ARRAY) {
                for (int j = 0; j < size; j++) {
                    buildingsLocal[j] = new Building("Building" + j, 3000 + Math.random()
* 997000);
```

```
            }
        } else {
            buildingsLocal = buildings.clone();
        }

        long startTime = System.currentTimeMillis();
        ForkJoinPool pool = new ForkJoinPool(numberOfThreads);
        ParallelMergeSort<Building> task = new
ParallelMergeSort<>(buildingsLocal, 0, buildingsLocal.length - 1, priceComparator);
        pool.invoke(task);
        long endTime = System.currentTimeMillis();

        System.out.print(i + 1 + ") " + (endTime - startTime) + " ms");
        System.out.println("\t Масив відсортований: " + isSorted(buildingsLocal));
        totalTimeParallel += (endTime - startTime);
    }
    double averageTimeParallel = (double) totalTimeParallel / iterations;
    System.out.println("\nСередній час виконання паралельного алгоритму: " +
averageTimeParallel + " ms");

    System.out.println("\n\nSpeedup: " + (double) totalTime / totalTimeParallel);
  }
  public static boolean isSorted(Building[] buildings) {
    for (int i = 1; i < buildings.length; i++) {
      if (buildings[i - 1].getPrice() > buildings[i].getPrice()) {
        return false;
      }
    }
    return true;
  }
```

```
    public static boolean areEqual(Building[] buildings1, Building[] buildings2) {
        if (buildings1.length != buildings2.length) {
            return false;
        }
        for (int i = 0; i < buildings1.length; i++) {
            if (buildings1[i].getPrice() != buildings2[i].getPrice()) {
                return false;
            }
        }
        return true;
    }

}
```