

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 Шелла та його паралельна реалізація
мовою Java

Керівник:

ст. вик. А. Ю. Дифучин

«Допущено до захисту»

«___» _____ 2024 р.

Захищено з оцінкою

Члени комісії:

Виконавець:

Сергієнко Юрій Вікторович

студент групи ІП-14

залікова книжка № ІП-1427

«11» квітня 2024 р.

Інна СТЕЦЕНКО

Антон ДИФУЧИН

Київ – 2024

ЗАВДАННЯ

1. Виконати огляд існуючих реалізацій алгоритму сортування Шелла, послідовних та паралельних, з відповідними посиланнями на джерела інформації (статті, книги, електронні ресурси). Зробити висновок про актуальність дослідження.
2. Виконати розробку послідовного алгоритму сортування мовою Java. 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 сортування мовою Java. 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 сортування вважається успішною, якщо прискорення не менше 2.
7. Дослідити вплив параметрів паралельного алгоритму на отримуване прискорення.
8. Зробити висновки про переваги паралельної реалізації обчислень для алгоритму сортування Шелла, та програмних засобів, які використовувались.

АНОТАЦІЯ

Пояснювальна записка до курсової роботи складається з: 5 розділів, містить 7 рисунків, 8 таблиць, 8 джерел.

Метою роботи є дослідження послідовної та паралельної реалізацій алгоритму сортування Шелла, проектування, реалізація, тестування та аналіз швидкодії послідовної та паралельної версій алгоритму, дослідження та оцінка ефективності паралельного алгоритму шляхом експериментів із заміру часу зі зміною вхідних даних та кількості потоків.

КЛЮЧОВІ СЛОВА: ПАРАЛЕЛЬНИЙ АЛГОРИТМ, АЛГОРИТМ СОРТУВАННЯ ШЕЛЛА, ПОТОКИ, JAVA, SORT, ПІДЗАДАЧА

ЗМІСТ

ВСТУП	6
1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ.....	7
1.1 Відомі реалізації послідовного алгоритму	7
1.2 Відомі реалізації паралельного алгоритму.....	8
2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ.....	9
2.1 Проектування алгоритму.....	9
2.2 Реалізація алгоритму	9
2.3 Тестування алгоритму	10
2.4 Аналіз швидкодії алгоритму.....	10
3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС	12
4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЄКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ.....	13
4.1 Проектування алгоритму.....	13
4.2 Реалізація алгоритму	13
4.3 Тестування алгоритму	15
4.4 Аналіз швидкодії алгоритму.....	16
5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ	17
5.1 Опис обладнання.....	17
5.2 Дослідження ефективності паралельних обчислень	17
ВИСНОВКИ.....	21

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	22
ДОДАТКИ.....	23
Додаток А. Код алгоритму та допоміжних класів.....	23
Додаток Б. Код тестування алгоритмів.....	29

ВСТУП

Алгоритм сортування Шелла – це розширення принципу алгоритму сортування вставками, яке збільшує швидкість виконання за рахунок обміну елементів, що знаходяться далеко один від одного. Основною ідеєю алгоритму є розбиття початкового масиву на підмасиви, які в свою чергу впорядковуються сортуванням вставками [1].

Хоча даний алгоритм є найбільш ефективним серед алгоритмів сортування $O(n^2)$ складності, він втрачає свою швидкість на великих об'ємах даних. Саме тому виникає необхідність у його оптимізації. Одним з найбільш практичних методів є застосування паралельних обчислень, що, в свою чергу, зменшить час роботи алгоритму [2, ст. 1].

Основна мета паралельної розробки - виконувати обчислення швидше, ніж це можна зробити з одним процесором, використовуючи декілька обчислювальних ресурсів одночасно. Досягнення цієї мети мало величезний вплив майже на всі види діяльності, пов'язані з обчисленнями. До них відносяться гідродинаміка, прогнозування погоди, моделювання та імітація великих систем, обробка та видобування інформації, обробка зображень, штучний інтелект та автоматизоване виробництво [3, ст. 4].

У ході дослідження будуть описані відомі реалізації алгоритму сортування, а також буде виконано паралелізацію даного алгоритму. Буде здійснено дослідження ефективності паралельного алгоритму відносно звичайного шляхом зміни кількості початкових даних та підзадач. Результати, отримані під час аналізу, будуть описані та відображені у вигляді таблиць та графіків.

1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

Сортування Шелла – це алгоритм, що розбиває метод сортування вставками на декілька етапів. Алгоритм був запропонований в роботі Шелла (1959) [4]. Метод отримав значну увагу за останні чверть століття після того, як книга Кнута 1973 року популяризувала його [5].

З практичної точки зору, інтерес до сортування Шелла обумовлений тим, що це досить простий метод сортування з невеликими накладними витратами і може бути легко реалізований. З теоретичної точки зору, інтерес полягає в тому, що сортування вставками має в середньому $O(n^2)$ часу виконання для сортування n -випадкових чисел, тоді як правильний вибір параметр поділу масиву на підмасиви може зменшити час виконання на порядок [6, ст.1].

1.1 Відомі реалізації послідовного алгоритму

Існує багато способів реалізації алгоритму, які відрізняються використанням різних послідовностей проміжків для сортування. Їхня спільна риса – усі повинні закінчуватись одиницею.

У оригінальному варіанті алгоритму використовується ряд чисел, де поточний проміжок дорівнює половині минулого проміжку (формула 1.1). Це найбільш поширена реалізація, за якої найгірший час виконання алгоритму досягає $O(n^2)$ (рисунок 1.1).

$$\left\lfloor \frac{N}{2^i} \right\rfloor \quad (1.1)$$

	5	8	0	2	4	6	1	3
gap=4	4	6	0	2	5	8	1	3
gap=2	0	2	1	3	4	6	5	8
gap=1	0	1	2	3	4	5	6	8

Рисунок 1.1 – Графічне представлення роботи алгоритму з проміжками Шелла

Одним з тих, хто створив свій варіант проміжку був Гібард. Так, у 1963 році він запропонував ряд чисел (формула 1.2), при використанні якого найгірший час виконання алгоритму становить $O(n^{3/2})$ [5].

$$2^i - 1 \tag{1.2}$$

1.2 Відомі реалізації паралельного алгоритму

Основною ідеєю паралельного алгоритму сортування Шелла є розділення масиву на частини, їх впорядкування та злиття. Найпопулярнішою є реалізація з розподіленням початкового масиву на кількість частин, рівних кількості процесорів системи та злиттям в кінці за допомогою алгоритму сортування Merge sort [7]. Також зустрічається варіант із класифікацією за певним параметром, в даному випадку за кількістю цифр в числі [8].

2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

2.1 Проектування алгоритму

Для кращого уявлення послідовного алгоритму, представимо його у вигляді псевдокоду:

START

PROCEDURE shellSort(list):

FOR int gap = list.size(); gap > 0; gap /= 2 **DO**

FOR int i = gap; i < list.size(); i++ **DO**

FOR int j = i - gap; j >= 0 **AND** list[j] > list[j + gap]; j -= gap **DO**

CALL swapList(list, j, j + gap)

END FOR

END FOR

END FOR

END PROCEDURE

PROCEDURE swapList(list, p1, p2):

temp = list[p1]

list[p1] = list[p2]

list[p2] = temp

END PROCEDURE

END

Можемо бачити, що зовнішній цикл функції shellSort() – проходження ряду чисел Шелла, останнім числом якого буде 1. Внутрішній цикл відтворює алгоритм сортування вставками: якщо поточне число більше за те, що попереду – відбувається їх перестановка.

2.2 Реалізація алгоритму

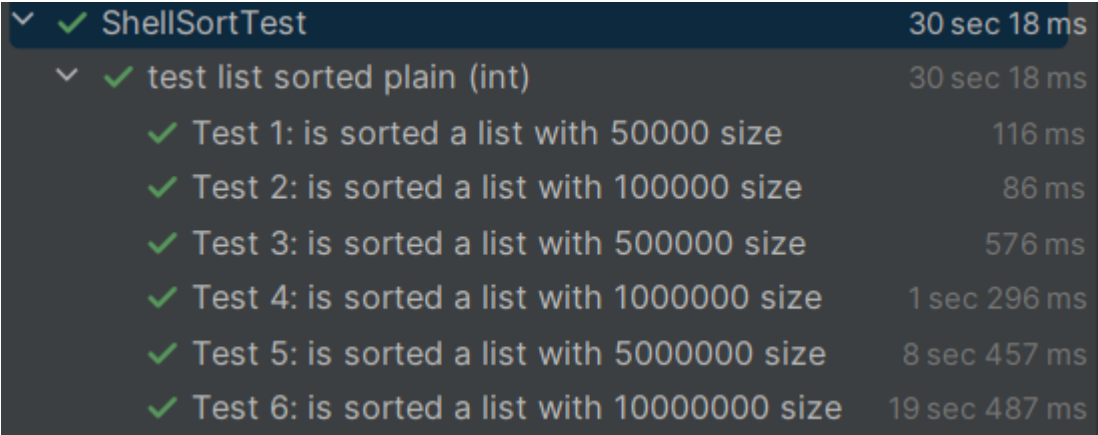
Для реалізації алгоритму необхідно обрати ряд проміжків. На мою думку, найкращим варіантом є використання оригінального ряду чисел Шелла.

Щодо складності алгоритму, на перший погляд можна подумати, що три цикли можуть продукувати велику часову складність, але це не так. Як було доведено, час роботи алгоритму залежить від проміжку, і найгіршим варіантом може бути $O(n^2)$. Весь код алгоритму можна переглянути у додатку А.

2.3 Тестування алгоритму

Для перевірки коректності роботи алгоритму створимо тести з використанням бібліотеки JUnit. Проведемо тестування на вхідних даних наступних розмірів: 50000, 100000, 500000, 1000000, 5000000, 10000000.

Після запуску тесту, згенеруємо список випадкових чисел вказаного розміру та проведемо його сортування. Для перевірки правильності сортування, використаємо функцію `assertTrue()` та функцію `isSortedAscending()`, її лістинг можна побачити у додатку А.



✓ ShellSortTest	30 sec 18 ms
✓ test list sorted plain (int)	30 sec 18 ms
✓ Test 1: is sorted a list with 50000 size	116 ms
✓ Test 2: is sorted a list with 100000 size	86 ms
✓ Test 3: is sorted a list with 500000 size	576 ms
✓ Test 4: is sorted a list with 1000000 size	1 sec 296 ms
✓ Test 5: is sorted a list with 5000000 size	8 sec 457 ms
✓ Test 6: is sorted a list with 10000000 size	19 sec 487 ms

Рисунок 2.1 – Тестування послідовного алгоритму

У результаті можемо бачити, що алгоритм успішно пройшов тестування. Лістинг тестів можна побачити у додатку Б.

2.4 Аналіз швидкодії алгоритму

Проведемо перевірку швидкодії алгоритму з використанням списків таких самих розмірів. Для коректного заміру часу роботи алгоритму необхідно пройти декілька ітерацій без заміру, тому перші 20 ітерацій пропустимо. За наступні 20 ітерацій порахуємо середній час сортування та занесемо до таблиці 2.1.

Таблиця 2.1 – Середній час сортування послідовним алгоритмом

Розмір списку	Час виконання, мс
50000	22
100000	61
500000	529
1000000	1234
5000000	7965
10000000	17050

Представимо отримані результати у вигляді графіку на рисунку 2.2.

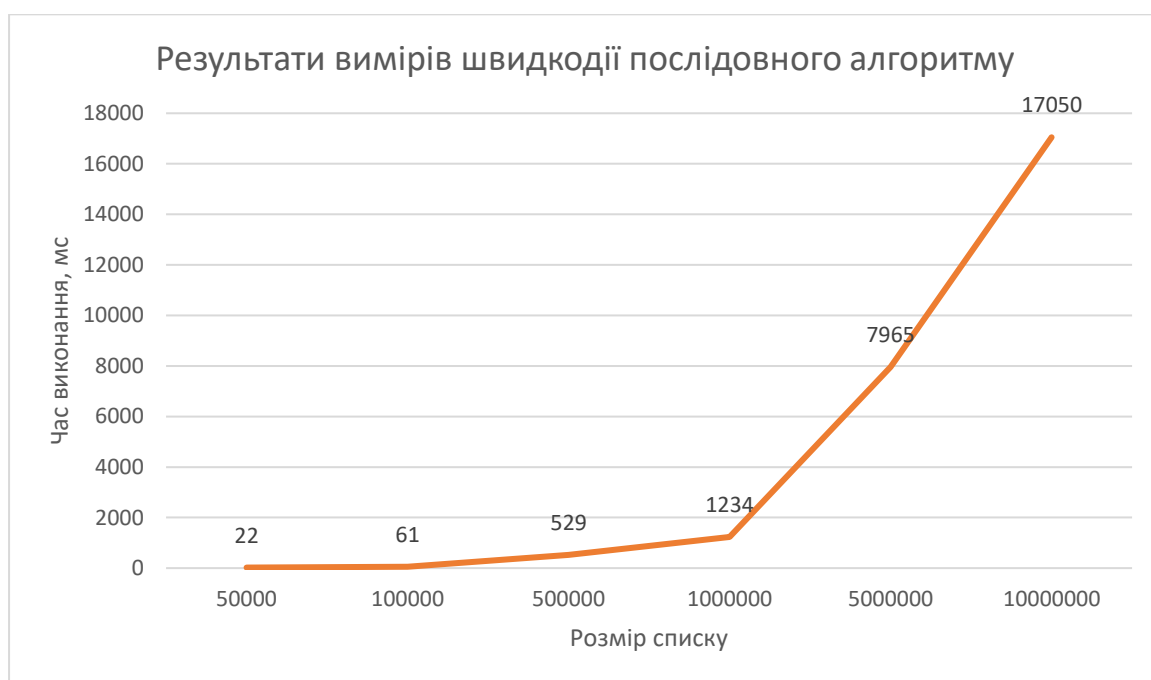


Рисунок 2.2 – Результати вимірів швидкодії послідовного алгоритму

Можемо бачити, що через середню складність алгоритму $O(n \log(n))$, час виконання зростає зі збільшенням елементів, але не надзвичайно стрімко. Також можна зробити висновок, що алгоритм досить ефективний для малих об'ємів даних, але потребує оптимізації на великих об'ємах.

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

Для реалізації паралельного алгоритму та його тестування було обрано мову програмування Java, адже у ній наявні бібліотеки, методи та інструменти для роботи з багатопоточністю.

Java надає різноманітний, потужний набір вбудованих пакетів для розробки паралельного програмного забезпечення, такі як `java.util.concurrent` та `java.lang`. Говорячи про клас `Thread` та інтерфейс `Runnable` пакету `java.lang`, можна сказати, що дані інструменти дають змогу програмісту контролювати процес паралельних обчислень, дозволяючи встановлювати пріоритети потоків, керувати їх життєвим циклом та взаємодіяти між ними у безпечний спосіб.

Пакет `java.util.concurrent` містить набір класів та інтерфейсів, які допомагають у створенні ефективних і надійних паралельних програм. Починаючи від атомарних змінних та закінчуючи пулом потоків, усі інструменти даного пакету дають змогу безпечно використовувати усі ресурси системи у зручний спосіб.

У моєму випадку, Fork-Join фреймворк був найкращим вибором для розробки паралельного алгоритму, адже цей механізм дозволяє розбивати великі задачі на менші частини, які можна виконувати паралельно, а потім об'єднувати їх результати виконання. Особливістю даного фреймворку є «work-stealing» алгоритм, за якого завдання автоматично розподіляються між доступними потоками. Коли один потік завершує свою роботу, він може «вкрасти» завдання з черги іншого потоку.

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

4.1 Проектування алгоритму

Для проектування алгоритму необхідно обрати стратегію розбиття початкового списку на частини та їх подальше злиття. Для цього я обрав алгоритм сортування Samplesort – алгоритм групи «divide and conquer», основною ідеєю якого є розбиття масиву на частини за певним критерієм (найчастіше це числа, що входять у певний проміжок), їх сортування та конкатенація. Відображення його роботи можна побачити на рисунку 4.1.

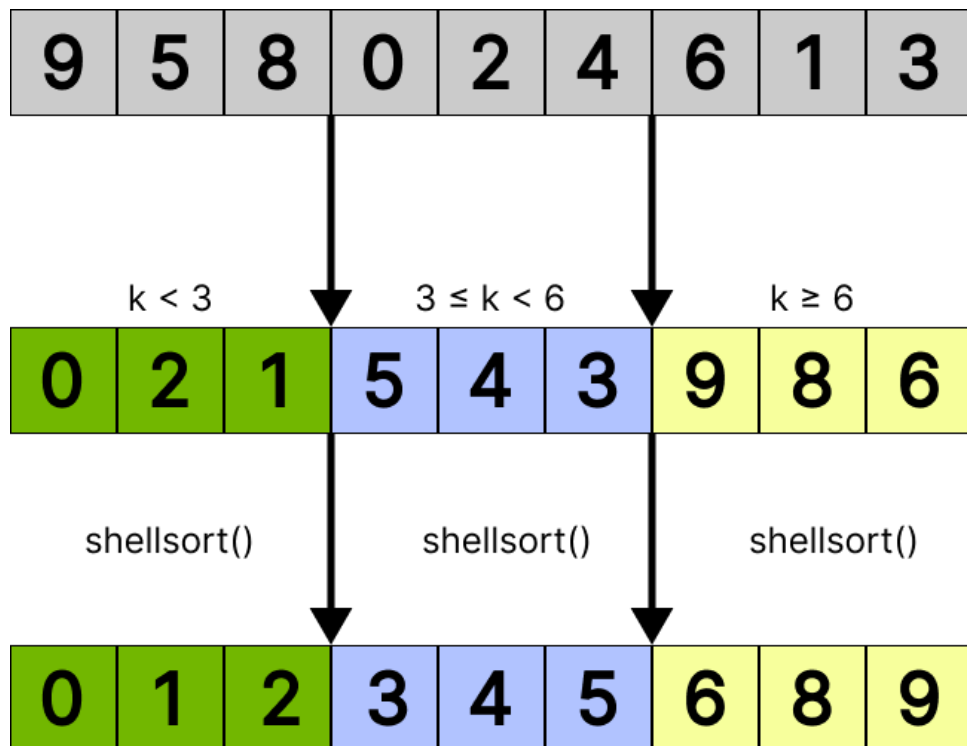


Рисунок 4.1 – Принцип роботи алгоритму Samplesort

4.2 Реалізація алгоритму

Для реалізації паралельного алгоритму необхідно обрати технологію, якою найлегше буде розбити вхідний список на частини і потім синхронно зібрати. На мою думку, найкращим варіантом є Fork-Join фреймворк, адже він дозволяє ефективно провести як розбиття, так і злиття.

Фреймворк містить клас RecursiveTask, за допомогою якого можна легко розбити список на підсписки. Для цього, у методі compute() треба створювати та

викликати підзадачі до того часу, поки не буде досягнуто бажаного розміру списку. Я обрав соту частину кількості елементів початкових даних як оптимальний розмір підсписку. В такому випадку не треба розділяти дані по потоках, а пул потоків сам визначає собі роботу.

Для кращого розуміння алгоритму, представимо його у вигляді псевдокоду:

START

INPUT: main, from, to, count

// de main – список, from – початок, to – кінець, count – початкова к-сть елементів

PROCEDURE compute():

IF to – from > count / 100 **THEN**

barrier = (from – to) / 2

startTask = new ShellSortTask(

getChunk(main, from, barrier), from, barrier, count)

endTask = new ShellSortTask(

getChunk(main, from, barrier), from, barrier, count)

startTask.fork()

endTask.fork()

main.clear()

main.addAll(startTask.join())

main.addAll(endTask.join())

ELSE

shellSort(main)

END IF

RETURN main

END PROCEDURE

```

PROCEDURE getChunk(main, start, end):
    RETURN main.filter(el >= start AND el < end)
END PROCEDURE

```

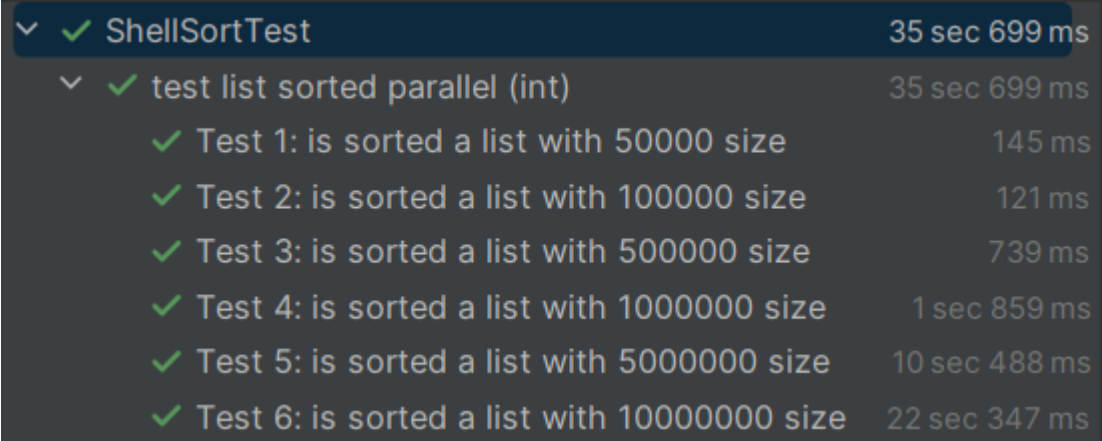
END

Із псевдокоду можна побачити, що створені підзадачі будуть виконуватись паралельно за допомогою методу `fork()` і, врешті-решт, збиратись у один список за допомогою методу `join()`. Треба зазначити, що після виклику методу `join()` задача очікує виконання обох підзадач, і після цього завершує свою роботу. Коли усі підзадачі будуть завершені, `ForkJoinPool` поверне повний результат.

4.3 Тестування алгоритму

Для перевірки коректності роботи алгоритму проведемо тестування на згенерованих вхідних даних наступних розмірів: 50000, 100000, 500000, 1000000, 5000000, 10000000.

Після запуску тесту, згенеруємо список випадкових чисел вказаного розміру та продублюємо його. Далі, посортуємо обидва списки, один з них – послідовним алгоритмом, а інший – паралельним. Для перевірки правильності сортування, використаємо функції `assertTrue()`, `isSortedAscending()` та `compareLists()`, їх лістинг відображений у додатку А.



✓ ShellSortTest	35 sec 699 ms
✓ test list sorted parallel (int)	35 sec 699 ms
✓ Test 1: is sorted a list with 50000 size	145 ms
✓ Test 2: is sorted a list with 100000 size	121 ms
✓ Test 3: is sorted a list with 500000 size	739 ms
✓ Test 4: is sorted a list with 1000000 size	1 sec 859 ms
✓ Test 5: is sorted a list with 5000000 size	10 sec 488 ms
✓ Test 6: is sorted a list with 10000000 size	22 sec 347 ms

Рисунок 4.2 – Результати тестувань паралельного алгоритму

Усі тести пройшли успішно, їх код можна побачити у додатку Б.

4.4 Аналіз швидкодії алгоритму

Проведемо перевірку швидкодії паралельного алгоритму з використанням списків таких самих розмірів. Для прикладу візьмемо найбільшу можливу кількість потоків - 12. Так як і для послідовного, необхідно пройти декілька ітерацій без заміру, тому перші 20 ітерацій пропустимо. За наступні 20 ітерацій порахуємо середній час сортування та занесемо до таблиці 4.1.

Таблиця 4.1 – Середній час сортування паралельним алгоритмом

Розмір списку	Час виконання, мс
50000	5
100000	10
500000	66
1000000	145
5000000	1811
10000000	4068

Представимо отримані результати у вигляді графіку на рисунку 4.3.



Рисунок 4.3 – Результати вимірів швидкодії паралельного алгоритму

Можемо бачити, що алгоритм виявився ефективним для великих та малих об'ємів даних.

5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

Для дослідження ефективності алгоритму необхідно провести експерименти по заміру часу роботи алгоритмів. Для кожного заміру проводиться по 40 ітерацій, перші 20 з яких пропускаються.

5.1 Опис обладнання

Дослідження алгоритму було проведено на персональному комп'ютері з операційною системою Windows. Процесор AMD Ryzen 5 4600h 6-ядерний, 12 логічних потоків, 16 ГБ ОЗУ.

5.2 Дослідження ефективності паралельних обчислень

Порівнюємо час виконання послідовного та паралельного алгоритмів. Для паралельного використаємо 2 потоки. Результати порівнянь відображені у таблиці 5.1.

Таблиця 5.1. – Порівняння послідовного та паралельного алгоритмів (2 потоки)

Кількість елементів	Час послідовного алгоритму, мікросекунд	Час паралельного алгоритму, мікросекунд
50000	22	8
100000	61	18
500000	529	168
1000000	1234	496
5000000	7965	4108
10000000	17050	9697

Прискорення алгоритму відображене у таблиці 5.2.

Таблиця 5.2. – Прискорення паралельного алгоритму (2 потоки)

Кількість елементів	Прискорення алгоритму, разів
50000	2,75
100000	3,39
500000	3,15

Продовження таблиці 5.2:

Кількість елементів	Прискорення алгоритму, разів
1000000	2,49
5000000	1,93
10000000	1,75

Як бачимо, прискорення є хорошим, але недостатнім. Видно, що алгоритм достатньо легко справляється з малими об'ємами даних, але досі витрачає багато часу для великих об'ємів. Спробуємо збільшити кількість потоків до 6.

Таблиця 5.3. – Порівняння послідовного та паралельного алгоритмів (6 потоків)

Кількість елементів	Час послідовного алгоритму, мікросекунд	Час паралельного алгоритму, мікросекунд
50000	22	6
100000	61	12
500000	529	79
1000000	1234	267
5000000	7965	2150
10000000	17050	5142

Прискорення алгоритму з використанням 6 потоків відображене у таблиці 5.4.

Таблиця 5.4. – Прискорення паралельного алгоритму (6 потоків)

Кількість елементів	Прискорення алгоритму, разів
50000	3,67
100000	5,08
500000	6,69
1000000	4,62
5000000	3,70
10000000	3,31

З використанням половини ресурсів системи, було досягнуто бажаного прискорення на усіх ітераціях. Можемо бачити, що найкраще прискорення спостерігається на 6-цифрових наборах даних, а на 10 мільйонах елементів прискорення покращилось майже вдвічі від минулих замірів.

Оскільки час виконання алгоритму з використанням 12 потоків було заміряно, прорахуємо його прискорення. Результат відображений у таблиці 5.5.

Таблиця 5.5 – Прискорення паралельного алгоритму (12 потоків)

Кількість елементів	Прискорення алгоритму, разів
50000	4,4
100000	6,1
500000	8,01
1000000	8,51
5000000	4,39
10000000	4,19

Як можемо бачити, було досягнуто максимального прискорення у 8,5 разів з кількістю елементів 1000000.

Порівняємо отримані результати у таблиці 5.6.

Таблиця 5.6 – Порівняння результатів прискорення

	2 потоки	6 потоків	12 потоків
50000	2,75	3,67	4,4
100000	3,39	5,08	6,1
500000	3,15	6,69	8,01
1000000	2,49	4,62	8,51
5000000	1,93	3,70	4,39
10000000	1,75	3,31	4,19

Отже, зі збільшенням кількості потоків для алгоритму, його ефективність збільшується. Візуалізацію прискорення можемо бачити на рисунку 5.1.

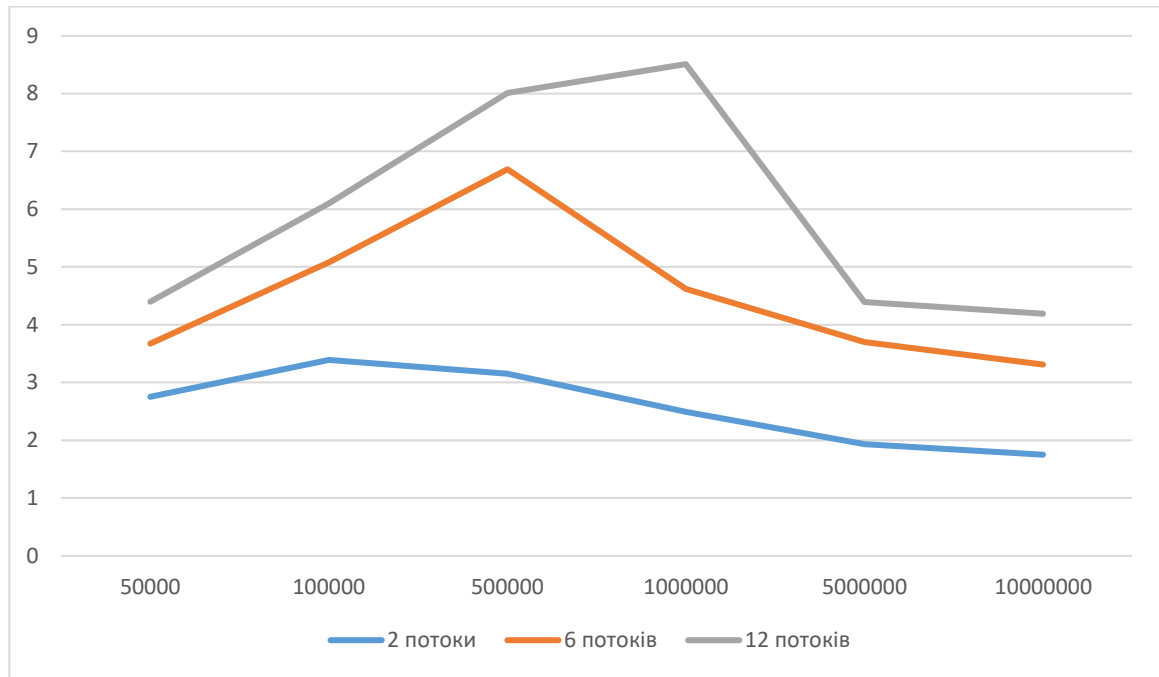


Рисунок 5.1 – Візуалізація прискорення паралельного алгоритму

Результати показують, що найбільш ефективним паралельний алгоритм виявився на середніх об'ємах даних.

ВИСНОВКИ

В рамках виконання даної роботи було досліджено ефективність алгоритму сортування Шелла шляхом порівняння послідовної та паралельної реалізацій.

Було розглянуто та описано відомі версії алгоритму сортування, як послідовні, так і паралельні, в результаті чого було обрано оригінальний варіант розподілу проміжків. Також, було спроектовано послідовний алгоритм шляхом створення псевдокоду, далі реалізовано та протестовано.

Був обґрунтований вибір мови програмування Java, фреймворку та технологій, що були використані при розробці паралельного алгоритму. В свою чергу, було описано роботу паралельної версії сортування з використанням алгоритму Samplesort. В результаті, реалізований алгоритм вирішив проблему послідовного алгоритму: за менші проміжки часу він здатний сортувати більші об'єми даних.

Дослідження ефективності паралельного алгоритму довело, що потрібне прискорення було досягнуто. Результати показали, що алгоритм пропорційно залежить від кількості потоків, виділених для роботи над даними. Це означає, що поставлене завдання було виконано успішно.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Algorithms, 4th edition. Elementary Sorts [Електронний ресурс] // Robert Sedgewick. URL: <https://algs4.cs.princeton.edu/21elementary/index.php#2.3> (дата звернення: 29.03.2024).
2. Enhanced Shell Sorting Algorithm [Електронний ресурс] // Basit Shahzad. URL: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=fdd359a2698c2e45dcc739c964a6d98315b9c9f3> (дата звернення: 29.03.2024).
3. An Introduction to Parallel Algorithms [Електронний ресурс] // Joseph JaJa. URL: <https://users.cs.utah.edu/~hari/teaching/bigdata/book92-JaJa-parallel.algorithms.intro.pdf> (дата звернення: 29.03.2024).
4. A High-Speed Sorting Procedure [Електронний ресурс] // Donald Shell. URL: <https://dl.acm.org/doi/pdf/10.1145/368370.368387> (дата звернення: 29.03.2024).
5. Knuth D. E. The art of computer programming. Vol.3: Sorting and searching: USA, 1973. 792 с. (дата звернення: 29.03.2024).
6. Stochastic Analysis of Shell Sort [Електронний ресурс] // R. T. Smythe and J. Wellner. URL: <https://sites.stat.washington.edu/people/jaw/JAW-papers/jaw-smythe-Algorithm01.pdf> (дата звернення: 29.03.2024).
7. Implementation Of Parallel Shell Sort Using MPI [Електронний ресурс] // Dr. Russ Miller. URL: <https://cse.buffalo.edu/faculty/miller/Courses/CSE633/prasad-salvi-Spring-2017-CSE633.pdf> (дата звернення: 29.03.2024).
8. Parallel Implementation of Sorting Algorithms [Електронний ресурс] // Malika Dawra. URL: <https://www.ijcsi.org/papers/IJCSI-9-4-3-164-169.pdf> (дата звернення: 29.03.2024).

ДОДАТКИ

Додаток А. Код алгоритму та допоміжних класів

Код алгоритму: <https://github.com/sergienkoyura/shellsort>

ShellSortUtil.java

```
package com.kpi.parallel;

import java.util.List;
import java.util.concurrent.ForkJoinPool;

public class ShellSortUtil {

    public static void shellSortParallel(List<Integer> list, int threads){
        ForkJoinPool pool = new ForkJoinPool(threads);
        pool.invoke(new ShellSortTask(list, 0, list.size(), list.size()));
    }

    public static void shellSort(List<Integer> list) {
        for (int gap = list.size(); gap > 0; gap /= 2)
            for (int i = gap; i < list.size(); i++)
                for (int j = i - gap; j >= 0 && list.get(j) > list.get(j + gap); j -= gap)
                    swapList(list, j, j + gap);
    }

    public static void swapList(List<Integer> list, int p1, int p2) {
        int temp = list.get(p1);
        list.set(p1, list.get(p2));
        list.set(p2, temp);
    }
}
```

```

public static boolean isSortedAscending(List<Integer> array) {
    int count = 0;
    for (int i = 0; i < array.size() - 1; i++) {
        if (array.get(i) > array.get(i + 1)) {
            count++;
        }
    }
    return !(count > 0);
}

```

```

public static boolean compareLists(List<Integer> list1, List<Integer> list2) {
    if (list1.size() != list2.size()) {
        return false;
    }

    for (int i = 0; i < list1.size(); i++) {
        if (!list1.get(i).equals(list2.get(i))) {
            return false;
        }
    }

    return true;
}

```

ShellSortTask.java

```

package com.kpi.parallel;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.RecursiveTask;

```



```

public class ShellSortTask extends RecursiveTask<List<Integer>> {
    private final List<Integer> main;
    private final int from;
    private final int to;
    private final int count;

    public ShellSortTask(List<Integer> main, int from, int to, int count) {
        this.main = main;
        this.from = from;
        this.to = to;
        this.count = count;
    }

```

@Override

```

protected List<Integer> compute() {
    if(to - from > count / 100) {
        int barrier = (from + to) / 2;
        ShellSortTask startTask = new ShellSortTask(getChunk(main, from,
barrier), from, barrier, count);
        ShellSortTask endTask = new ShellSortTask(getChunk(main, barrier,
to), barrier, to, count);
        startTask.fork();
        endTask.fork();

        main.clear();
        main.addAll(startTask.join());
        main.addAll(endTask.join());
    } else {
        ShellSortUtil.shellSort(main);
    }
}

```

```

    }
    return main;
}
private List<Integer> getChunk(List<Integer> main, int start, int end){
    return new ArrayList<>(main.stream().filter(el -> el >= start && el <
end).toList());
}
}

```

Timer.java

```

package com.kpi.parallel;

import java.util.ArrayList;
import java.util.List;

public class Timer {
    public static void main(String[] args) {

        int count = 50000;
        int threads = 6;
        int repeats = 20;
        long plainAvgTime = 0;
        long parallelAvgTime = 0;

        System.out.printf("%d elements%n", count);
        warmup(count, threads);
        for (int i = 1; i <= repeats; i++){
            List<Integer> listPlain = Generator.generate(count);
            List<Integer> listParallel = new ArrayList<>(listPlain);

            long start = System.currentTimeMillis();

```

```

        ShellSortUtil.shellSort(listPlain);
        long end = System.currentTimeMillis();

        plainAvgTime = (plainAvgTime * (i - 1) + end - start) / i;

        System.out.printf("Plain algorithm -> process took %d milliseconds%n",
end - start);

        start = System.currentTimeMillis();
        ShellSortUtil.shellSortParallel(listParallel, threads);
        end = System.currentTimeMillis();

        parallelAvgTime = (parallelAvgTime * (i - 1) + end - start) / i;

        System.out.printf("Parallel algorithm -> %d threads -> process took %d
milliseconds%n", threads, end - start);

        System.out.printf("Sorted:                                %b%n",
ShellSortUtil.isSortedAscending(listParallel));
        System.out.printf("Same: %b%n", ShellSortUtil.compareLists(listPlain,
listParallel));

    }

    System.out.println("AVG PLAIN: " + plainAvgTime);
    System.out.println("AVG PARALLEL: " + parallelAvgTime);

    System.out.printf("better in %.3f times", (double) plainAvgTime /
parallelAvgTime);
}

```

```

static void warmup(int count, int threads){
    for (int i = 0; i < 20; i++){
        System.out.println("Warmup " + i);
        List<Integer> listPlain = Generator.generate(count);
        List<Integer> listParallel = new ArrayList<>(listPlain);
        ShellSortUtil.shellSort(listPlain);
        ShellSortUtil.shellSortParallel(listParallel, threads);
    }
}
}

```

Generator.java

```

package com.kpi.parallel;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class Generator {
    public static List<Integer> generate(int count){
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < count; i++){
            list.add(new Random().nextInt(count));
        }
        return list;
    }
}

```

Додаток Б. Код тестування алгоритмів

ShellSortTest.java

```

import com.kpi.parallel.Generator;
import com.kpi.parallel.ShellSortUtil;
import org.junit.jupiter.api.DisplayNameGeneration;
import org.junit.jupiter.api.DisplayNameGenerator;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

import java.util.ArrayList;
import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
public class ShellSortTest {

    @ParameterizedTest(name = "Test {index}: is sorted a list with {0} size")
    @ValueSource(ints = {50000, 100000, 500000, 1000000, 5000000,
10000000})
    void test_list_sorted_plain(int count) {
        List<Integer> listPlain = Generator.generate(count);

        long start = System.currentTimeMillis();
        ShellSortUtil.shellSort(listPlain);
        long end = System.currentTimeMillis();

        System.out.printf("Size: %d; process took %d milliseconds%n", count, end
- start);

```

```

        assertTrue(ShellSortUtil.isSortedAscending(listPlain));
    }

    @ParameterizedTest(name = "Test {index}: is sorted a list with {0} size")
    @ValueSource(ints = {50000, 100000, 500000, 1000000, 5000000,
10000000})
    void test_list_sorted_parallel(int count) {
        List<Integer> listPlain = Generator.generate(count);
        List<Integer> listParallel = new ArrayList<>(listPlain);
        int threads = 12;

        ShellSortUtil.shellSort(listPlain);
        ShellSortUtil.shellSortParallel(listParallel, threads);

        assertTrue(ShellSortUtil.isSortedAscending(listParallel));
        assertTrue(ShellSortUtil.compareLists(listPlain, listParallel));
    }
}

```