

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 бульбашкою

|  |  |
| --- | --- |
| **Керівник**:  асист. О.Ю.Дифучина  «Допущено до захисту»  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2024 р.  Захищено з оцінкою  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Члени комісії:  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Виконавець**:  Радзівіло Валерія Артемівна  студентка групи ІП-14  залікова книжка № \_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «25» квітня 2024 р.  Інна СТЕЦЕНКО  Олександра ДИФУЧИНА |

**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**

У ході виконання даної курсової роботи було виконане дослідження швидкодії алгоритму сортування бульбашкою, зроблено паралельну реалізацію. Були проведені експерименти, за якими визначено рівень прискорення від паралелізації алгоритму. У ході експерименту оцінено вплив основних параметрів на ефективність алгоритму.

При виконанні була використана мова Java.

Ключові слова: ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, ПАРАЛЕЛЬНІ АЛГОРИТМИ, СОРТУВАННЯ МАСИВІВ, АЛГОРИТМ БУЛЬБАШКИ, JAVA, ПАРНЕ-НЕПАРНЕ, РОДІЛЯЙ І ВОЛОДАРЮЙ.

ЗМІСТ

[ВСТУП 6](#_Toc164016771)

[1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ 7](#_Toc164016772)

[1.1 Опис та реалізація послідовного алгоритму 7](#_Toc164016773)

[1.2 Опис та реалізація паралельного алгоритму 9](#_Toc164016774)

[2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ 11](#_Toc164016775)

[2.1 Реалізація послідовного алгоритму 11](#_Toc164016776)

[2.2 Тестування обчислень послідовного алгоритму 12](#_Toc164016777)

[3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС 17](#_Toc164016778)

[4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЄКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ 18](#_Toc164016779)

[4.1 Проєктування паралельного алгоритму 18](#_Toc164016780)

[4.2 Реалізація паралельного алгоритму 19](#_Toc164016781)

[4.3. Тестування паралельного алгоритму 20](#_Toc164016782)

[4.4 Аналіз швидкодії паралельного алгоритму 20](#_Toc164016783)

[5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ 25](#_Toc164016784)

[5.1 Опис обладнання для проведення експериментів 25](#_Toc164016785)

[5.2 Дослідження ефективності паралельних обчислень алгоритму 25](#_Toc164016786)

[ВИСНОВКИ 30](#_Toc164016787)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 31](#_Toc164016788)

[ДОДАТКИ 32](#_Toc164016789)

[Додаток А. Назва додатку 32](#_Toc164016790)

[Додаток Б. Назва додатку 33](#_Toc164016791)

[Додаток В. Назва додатку 34](#_Toc164016792)

# ВСТУП

Сортування вважається фундаментальною операцією в інформатиці, і для покращення процесу сортування запропоновано різні алгоритми. Усі алгоритми сортування прагнуть покращити час роботи або простір, який використовує алгоритм сортування, щоб швидко й ефективно сортувати певний список.[1] Існує багато алгоритмів сортування. Серед них сортування бульбашкою, сортування вставкою, сортування виділенням, сортування злиттям, швидке сортування, сортування купи та інші.

Алгоритм сортування бульбашкою є найпростішим алгоритмом сортування, який працює шляхом багаторазового обміну сусідніх елементів, якщо вони розташовані не в тому порядку. Алгоритм є легким для розуміння, реалізації та перевірки. Проте, основним обмеженням даного сортування є його продуктивність на великих наборах даних. Через складність алгоритму за часом він стає неефективним для сортування довгих списків, оскільки потребує багатьох порівнянь та перестановок елементів. Сортування бульбашкою найкраще підходить для малих наборів даних або ситуацій, коли простота та легкість реалізації є більш пріоритетним за швидкість.

З метою оптимізації часу виконання створюється паралельний алгоритм. Використання паралельних обчислень дозволяє розподілити завдання між декількома потоками, що працюють одночасно, що може значно прискорити процес сортування.

У рамках курсової роботи будуть представлені різні варіанти реалізації алгоритму сортування бульбашкою, їх модифікації, а також проведені відповідні експерименти. Ефективність паралельного алгоритму сортування буде виміряна в порівнянні з відповідним послідовним варіантом. Додатково буде проведено дослідження впливу параметрів паралельного алгоритму на його продуктивність.

# 1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

Сортування бульбашкою є простим алгоритмом сортування, який працює на основі порівнянь. У сортуванні бульбашкою найбільше число спочатку переміщується на самий кінець списку за допомогою ряду операцій порівняння та обміну, починаючи з протилежного кінця. Процедура повторюється, зупиняючись перед попередньо розташованим найбільшим числом, щоб отримати наступне найбільше число. Таким чином, більші числа переміщуються до кінця списку. Дане переміщення схоже на рух бульбашки, звідки і взялась назва сортування.

## 1.1 Опис та реалізація послідовного алгоритму

Алгоритм сортування бульбашкою є найпростішим алгоритмом сортування, який працює шляхом багаторазового обміну сусідніх елементів, якщо вони розташовані не в тому порядку. Цей алгоритм не підходить для великих наборів даних, оскільки його середня та найгірша часова складність є досить високою.

Є велика кількість варіантів реалізації алгоритму сортування бульбашкою. Одним з найбільш розповсюджених є використання двох циклів: внутрішній цикл служить для порівняння та обміну елементів, а зовнішній цикл повторює цей процес до моменту, коли масив стає повністю відсортованим. Використовується функція обміну, яка змінює позиції об'єктів у масиві, фактично переміщаючи значення в пам'яті. Часова складність сортування бульбашкою O(n2). Псевдокод цього алгоритму представлено на рисунку 1.1.

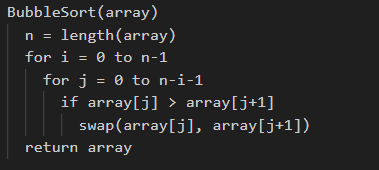


Рисунок 1.1 – Псевдокод послідовного сортування бульбашкою

Парне-непарне сортування - це варіант сортування бульбашкою, який працює в двох послідовних фазах: Парна фаза: парні індекси обмінюються значеннями з правими сусідами (0 ↔ 1, 2 ↔ 3, ...) Непарна фаза: непарні індекси обмінюються значеннями з правими сусідами (1 ↔ 2, 3 ↔ 4, ...). Також використовується прапорець, який визначає чи за останню ітерацію були перестановки. Якщо перестановок не було – сортування закінчене. Псевдокод парного-непарного сортування можна побачити на рисунку 1.2.

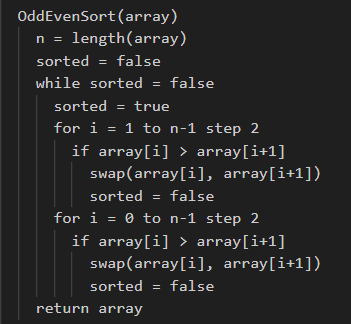


Рисунок. 1.2. – Псевдокод алгоритму парного-непарного сортування

Для послідовного програмування сортування за методом парних та непарних перестановок не має особливих переваг над звичайним сортуванням бульбашкою. Тому краще додатково використати модифікацію, яка розбиває масив на кілька підмасивів і виконує сортування кожного підмасиву за допомогою парного-непарного сортування. Після сортування підмасивів їх об'єднується, щоб отримати остаточно відсортований масив. Псевдокод парного-непарного сортування з модифікацією можна побачити на рисунку 1.3.

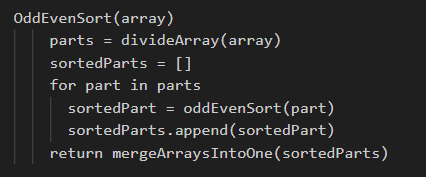


Рисунок 1.3 – Псевдокод алгоритму модифікованого парного-непарного сортування

## 1.2 Опис та реалізація паралельного алгоритму

Беручи до уваги невисоку продуктивність алгоритму сортування бульбашкою, його паралельна версія може суттєво підвищити цей показник.

Один з варіантів, яким ми можемо реалізувати імплементувати паралельний алгоритм сортування бульбашкою, - за допомоги розбиття масиву. Масив даних розбивається на n підмасивів, де n - це кількість доступних процесорів. Кожен підмасив розподіляється одному процесору для паралельного сортування. Це дозволяє використовувати більше обчислювальних ресурсів та прискорює процес сортування. Однак, паралельна реалізація вимагає додаткового коду для розподілу та об’єднання даних між потоками чи процесами. Саме тому, така реалізація не надає значного пришвидшення.

Як було зазначено раніше, послідовні алгоритми парного-непарного сортування та сортування бульбашкою можуть бути однаково неефективними при сортуванні великих обсягів даних. Проте, коли парне-непарне сортування виконують паралельно, воно працює швидше за звичайну послідовну реалізацію. Паралельний алгоритм працює на декількох процесорах або ядрах одночасно, а потім об'єднує часткові виходи, щоб виробити кінцевий результат.

Розділення паралельного сортування на масиви це найважливіший момент у алгоритмі. Якщо розділити на рівні частини першочерговий масив, необхідно буде додати постійні перевірки на те чи масив є повністю відсортованим, що суттєво збільшує час виконання. Тому, одним з рішень алгоритм розділяй і володарюй, який нерідко використовується в паралельних алгоритмах.

Алгоритми розділяй та володарюй вирішують проблеми шляхом розбивання на менші підпроблеми, рекурсивно вирішуючи підпроблеми, потім об’єднавши результати для створення такої рішення початкової проблеми. Простий алгоритм, який добре працює для проблем невеликого розміру, завершує рекурсію. Хороші алгоритми розділяй та володарюй існують для багатьох безліч проблем, включаючи сортування, маніпуляції з матрицею і багато проблем динамічного програмування [1].

Використовуючи даний алгоритм можна досягти ще більшого прискорення, не очікуючи на результати виконання. Візуалізація алгоритму можна побачити на рисунку 1.3.

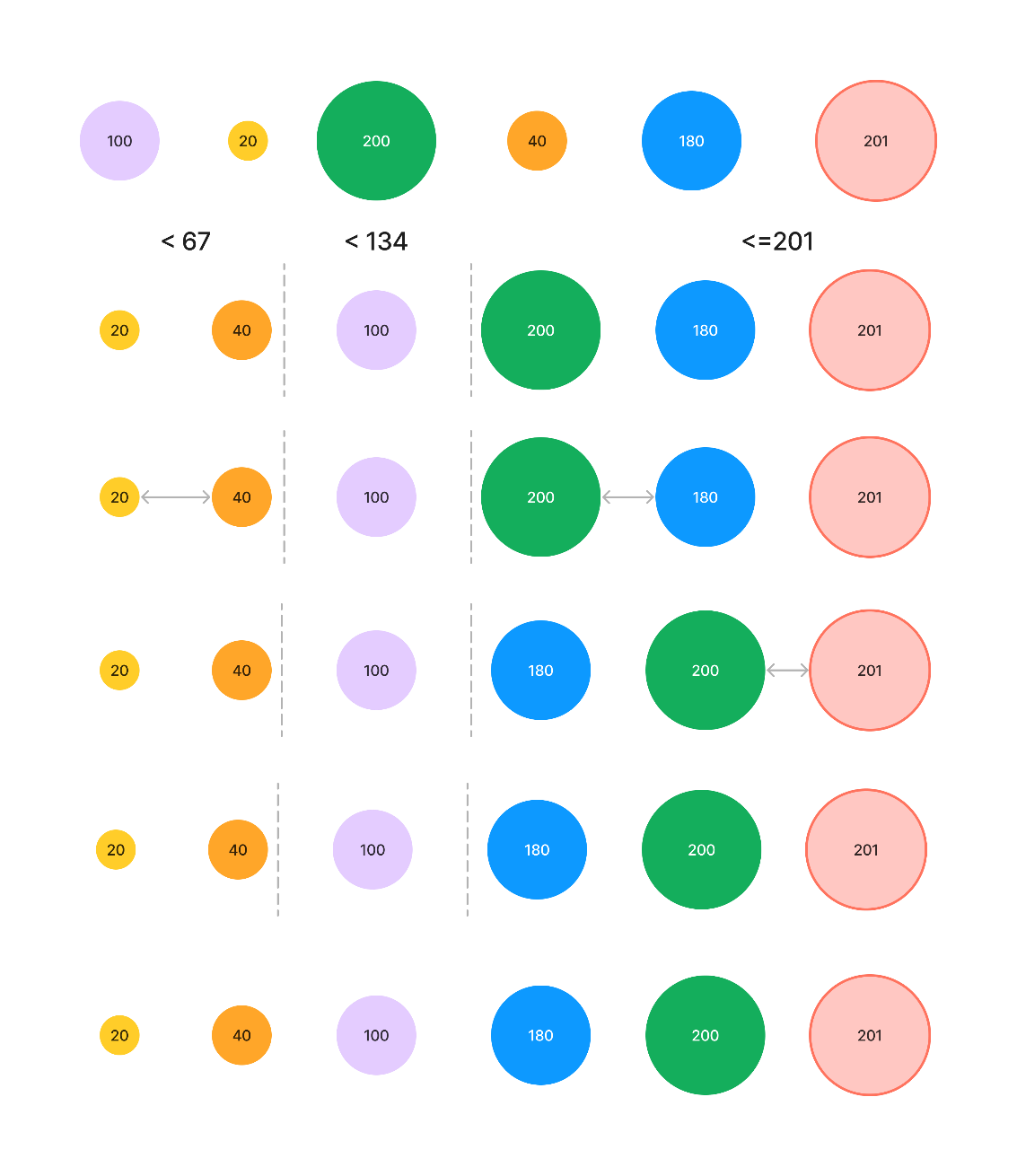


Рисунок 1.3 – Візуалізація паралельного алгоритму парного-непарного сортування

# 2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

## 2.1 Реалізація послідовного алгоритму

Використовуючи модифікацію алгоритму бульбашкового сортування, відомого як парне-непарне сортування, а також алгоритм розділяй та володарюй, будемо сортувати масив об'єктів BookCharacter. Порівняння елементів буде відбуватись за зростом персонажа. Детальний псевдокод можна побачити на рисунку 1.4.

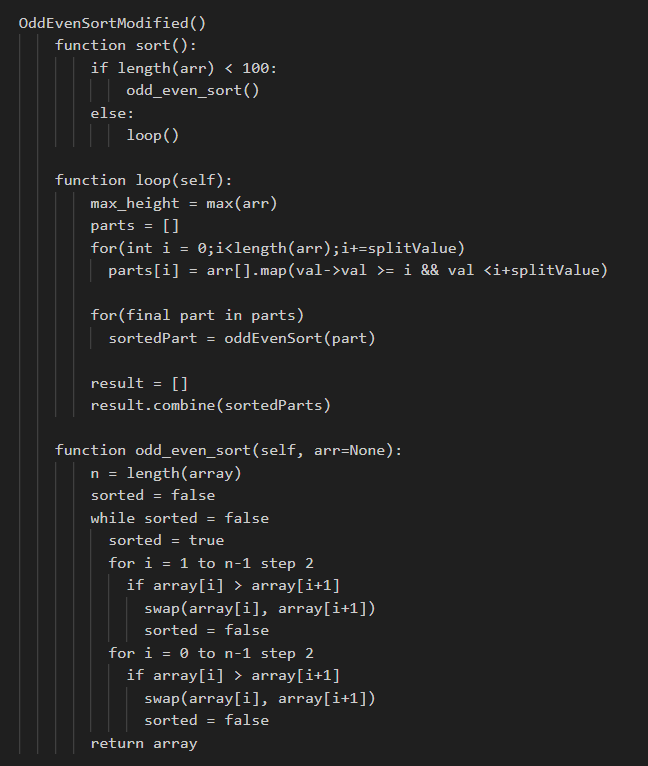


Рисунок 2.1 – Детальний псевдокод обраного послідовного алгоритму

Алгоритм, який ми розглядаємо, має часову складність O(n2). Це пов'язано з тим, що він використовує парно-непарний алгоритм сортування в середині розділених елементів. Парно-непарний алгоритм сортування, в свою чергу, також має часову складність O(n2). Отже, незалежно від того, як ми розбиваємо наші дані, основна операція сортування, яку ми виконуємо, має квадратичну часову складність.

Втім, важливо відзначити, що завдяки цим модифікаціям нам не потрібно виконувати повторні проходи по масиву після першого сортування за допомогою парно-непарного алгоритму. Таким чином, при роботі з великими масивами ми можемо спостерігати значне покращення часу виконання в порівнянні з традиційним парно-непарним алгоритмом.

Повний лістинг коду можна побачити у Додатку А.

## 2.2 Тестування обчислень послідовного алгоритму

Створимо OddEvenSortArraySizeTest - клас тестування, який використовується для перевірки коректності роботи алгоритму сортування на різних розмірах масиву.

Цей клас містить набір тестових методів, кожен з яких перевіряє роботу алгоритму сортування на масивах різного розміру - від 10000 до 100000 елементів.

У кожному тестовому методі генерується випадковий масив об'єктів BookCharacter, який потім сортується за допомогою створеного послідовного алгоритму. Після сортування перевіряється, чи відсортовані елементи масиву відповідно до їхньої висоти. Якщо всі елементи відсортовані вірно, тест проходить успішно. Якщо ж виявляється, що хоча б один елемент масиву розташований не на своєму місці, тест вважається невдалим, і це свідчить про помилку в алгоритмі сортування.

Результати тестування можна побачити на рисунку 2.2.

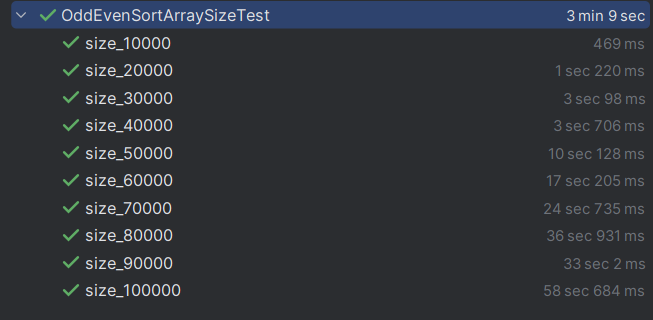


Рисунок 2.2 – Результати тестування послідовного алгоритму

Звідси можна зробити висновок, що код успішно проходить тестування та готовий для аналізу швидкодії та порівнянь з його паралельною імплементацією. Лістинг коду для даного тестування можна побачити у Додатку Б.

**2.3 Аналіз швидкодії послідовного алгоритму**

Аналіз швидкодії буде проводитись за середнім значенням 10 ітерацій сортувань випадково згенерованого масиву певного розміру. Лістинг коду для даного аналізу можна побачити у Додатку А.

Результати обчислень можна побачити на рисунках 2.3 – 2.7.

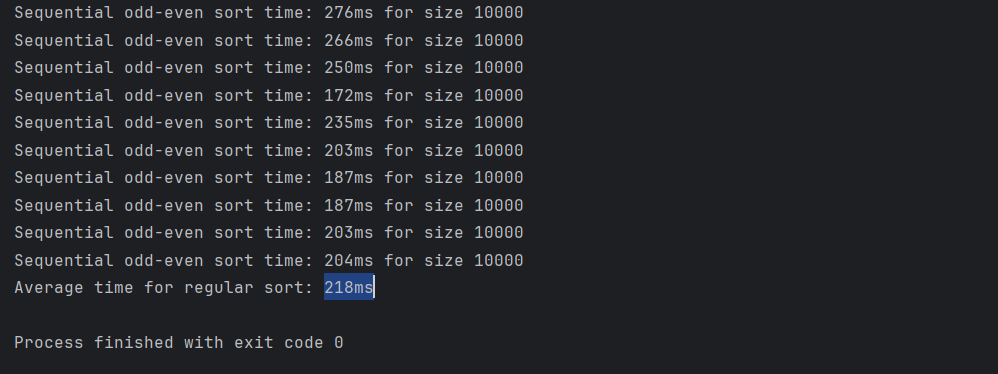


Рисунок 2.3 – Аналіз швидкодії послідовного алгоритму при 10000 елементах масиву

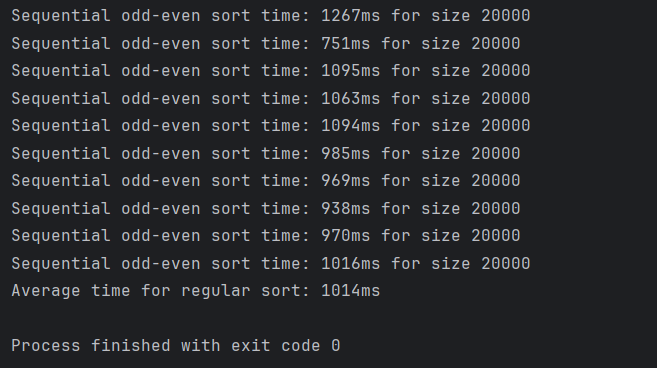


Рисунок 2.4 – Аналіз швидкодії послідовного алгоритму при 20000 елементах масиву

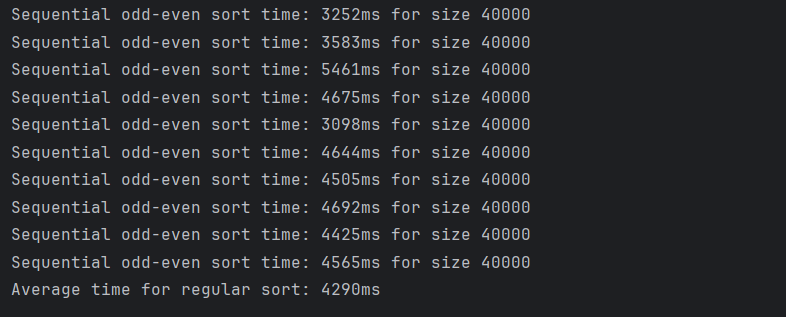


Рисунок 2.5 – Аналіз швидкодії послідовного алгоритму при 40000 елементах масиву

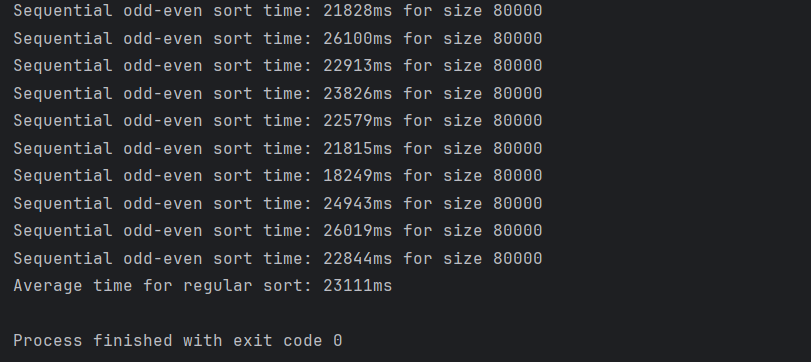


Рисунок 2.6 – Аналіз швидкодії послідовного алгоритму при 80000 елементах масиву

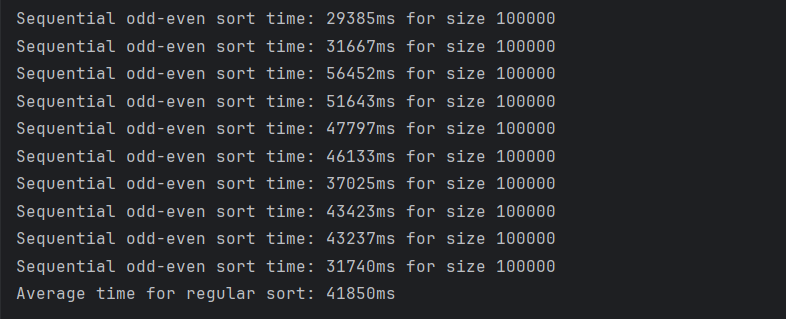


Рисунок 2.7 – Аналіз швидкодії послідовного алгоритму при 100000 елементах масиву

Для порівнянь отриманих результатів використаємо таблицю 2.1.

Таблиця 2.1 – Аналіз швидкодії послідовного алгоритму

|  |  |
| --- | --- |
| Кількість елементів в масиві | Середній час алгоритму, мілісекунди |
| 10000 | 218 |
| 20000 | 1014 |
| 40000 | 4290 |
| 80000 | 23111 |
| 10000 | 41850 |

Створимо графік для додаткового аналізу результатів. Графік можна побачити на рисунку 2.8.

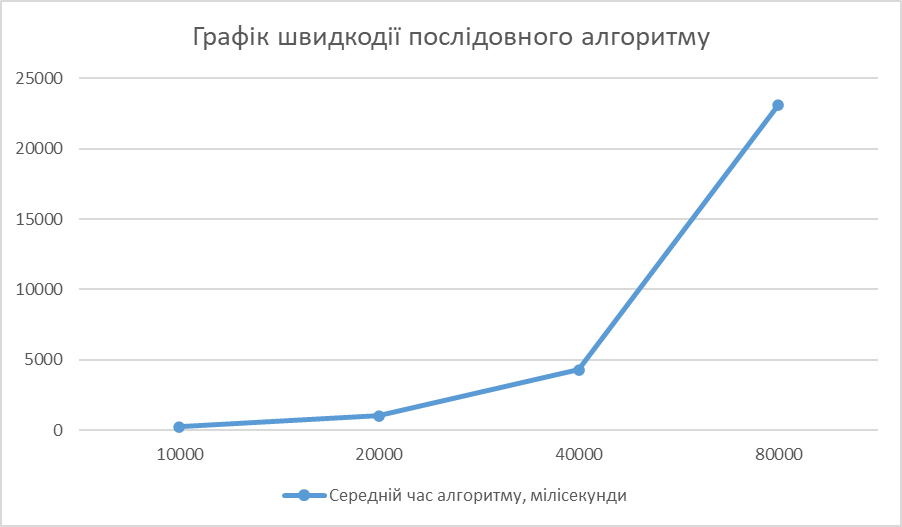


Рисунок 2.8 – Графік швидкодії послідовного алгоритму

На основі усіх проаналізованих даних, можна зробити висновок, що час виконання алгоритму зростає при збільшенні розміру масиву. Це очікувано, оскільки алгоритм сортування має квадратичну складність O(n2), де n - розмір масиву. Це також підтверджує графік, який є близьким до параболи.

При подальшому збільшенні розміру масиву, зростання часу виконання стає ще більш вираженим. Наприклад, коли розмір масиву збільшується в 2 рази з 40000 до 80000, час виконання збільшується більш ніж в 5 разів.

Ці спостереження підтверджують, що алгоритм сортування має квадратичну складність, і що він може стати непридатним для використання на великих масивах через високий час виконання. Саме цю проблему має вирішити паралельний алгоритм, зменшивши кількість витраченого часу на сортування.

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

Для розробки програмного забезпечення була обрана мова програмування Java. Головною причиною стало те, що Java має вбудовану підтримку багатопоточності, що робить її ідеальною для розробки алгоритмів, які можуть виконуватися паралельно. Це особливо важливо для задач сортування, де масиви даних можна розбивати на менші підмасиви та сортувати їх одночасно на різних потоках.

Наприклад, ForkJoinPool є одним з ключових компонентів в Java для реалізації паралельного програмування. Це введено в Java для реалізації алгоритмів, які можна розбити на менші завдання, а потім об'єднати їх для отримання кінцевого результату.

Важливим аспектом написання програми є можливість написання чистого та читабельного коду. Java є об'єктно-орієнтованою мовою програмування, що дозволяє створювати модульні програми та повторно використовувати код. В даному проєкті, це дозволяє легко створювати різні версії алгоритму сортування, використовуючи спадкування та поліморфізм.

Також, Java добре підходить для експериментальних досліджень, оскільки має зручні засоби для вимірювання часу виконання коду та аналізу його продуктивності.

Отже, використання Java дозволить легко виконати всі етапи завдання: від розробки послідовного та паралельного алгоритмів до тестування їх продуктивності та експериментального дослідження прискорення при зростанні обчислювальної складності.

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

## 4.1 Проєктування паралельного алгоритму

Створюючи паралельну реалізацію алгоритму, починаємо з розбиття головної задачі на менші, які можна виконувати паралельно. Обираємо додаткову модифікацію розділяй і володарюй, розділивши масив на підмасиви відносно ключових чисел. Без даної модифікації масив буде складатись з частин відсортованого масиву, але повний масив буде залишатись невідсортованим.

Ключові числа визначаються довжиною масиву та кількістю потоків. Кількість потоків є досліджуваним параметром, який буде проаналізований згодом. При застосуванні до діапазону, що складається принаймні з двох елементів, розбиття створює поділ на декілька послідовних непорожніх підмасиви таким чином, що жоден елемент першого підмасиву не є більшим за будь-який елемент другого підмасиву і так для кожного наступного підмасиву. Після застосування цього сортуємо піддіапазони. створимо потоки для паралельного виконання усіх наявних задач. Кожен окремий запущений потік виконує лінійне сортування парним-непарним сортуванням на своєму масиві. Після закінчення виконання усіх потоків отримуємо відсортований масив.

Паралельний алгоритм є покращенням основного, тому існують значення, для яких даний алгоритм не принесе користі. Відсічемо розмір таких масивів на початку реалізації.

Розбивши потік на підзадачі, створимо потоки для паралельного виконання усіх наявних задач. Кожен окремий запущений потік виконує лінійне сортування парним-непарним сортуванням на своєму масиві. Після закінчення виконання усіх потоків отримуємо відсортований масив.

З псевдокодом даного алгоритму можна ознайомитись на рисунку 4.1.

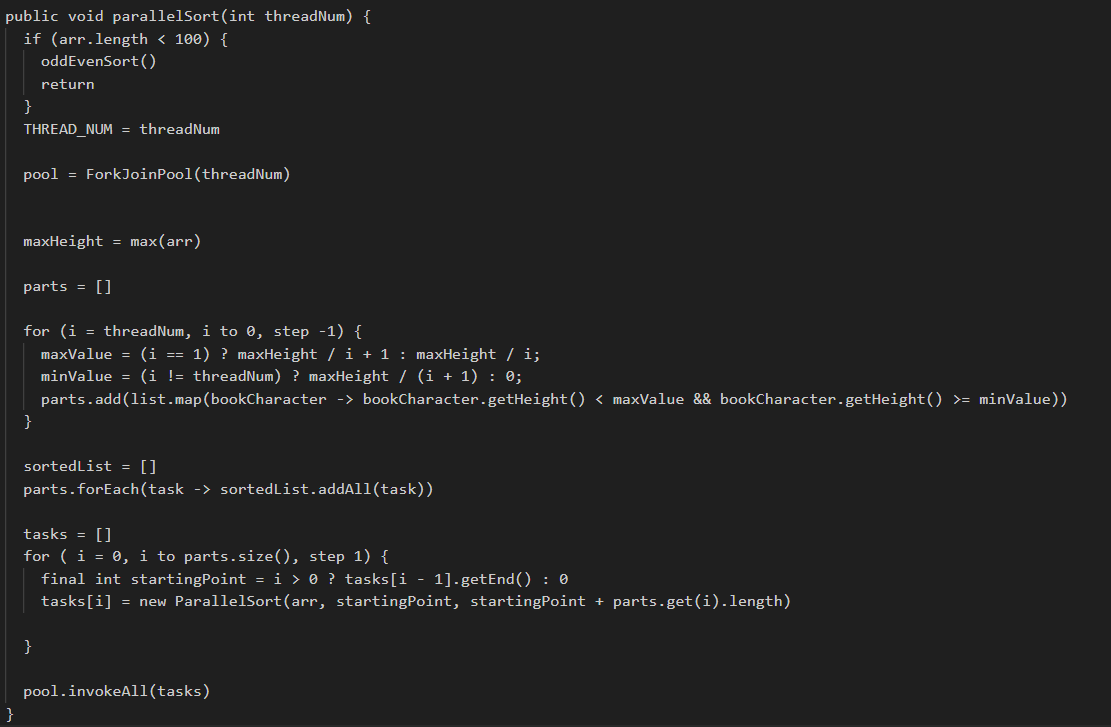


Рисунок 4.1 – Детальний псевдокод паралельного алгоритму

## 4.2 Реалізація паралельного алгоритму

Алгоритм реалізовується за допомогою Java Concurrency API. Він реалізує інтерфейс Callable<Void>, що означає, що його можна використовувати з ExecutorService для виконання в окремому потоці.

Також, використовується ForkJoinPool - один з ключових компонентів Java Concurrency API, який використовується для ефективного виконання задач, які можна розбити на менші задачі, що і відбувається в наявному паралельному алгоритмі. ForkJoinPool автоматично припинить роботу пулу потоків, як тільки всі завдання будуть виконані.

Повний лістинг коду можна побачити в Додатку А.

## 4.3. Тестування паралельного алгоритму

Тестування проходить аналогічно з тестуванням для послідовного алгоритму. Тому, протестуємо алгоритм для масивів від 10000 до 100000. Лістинг коду тестування можна побачити у Додатку Б. Результати тестування можна побачити на рисунку 4.2.

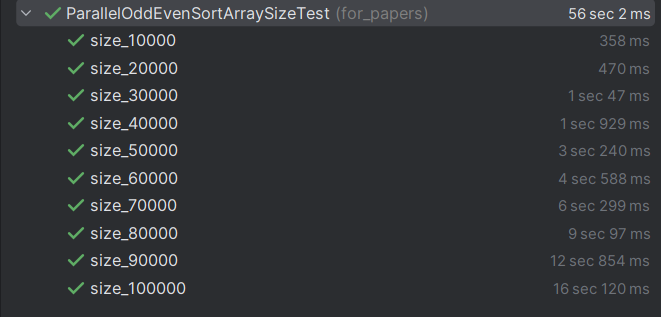


Рисунок 4.2 – Тестування паралельного алгоритму для різних розмірів масивів

Вже з цього тестування помітно, що виконання аналогічних тестів для паралельного алгоритму витрачає менше часу. Це означає, що ми успішно впорались зі створенням паралельного алгоритму.

## 4.4 Аналіз швидкодії паралельного алгоритму

Переконавшись у правильності роботи паралельного алгоритму, виконаємо більш чітку оцінку швидкодії, замірюючи середнє значення 10 ітерацій сортування випадкових масивів конкретного розміру. Лістинг коду можна знайти в Додатку Б.

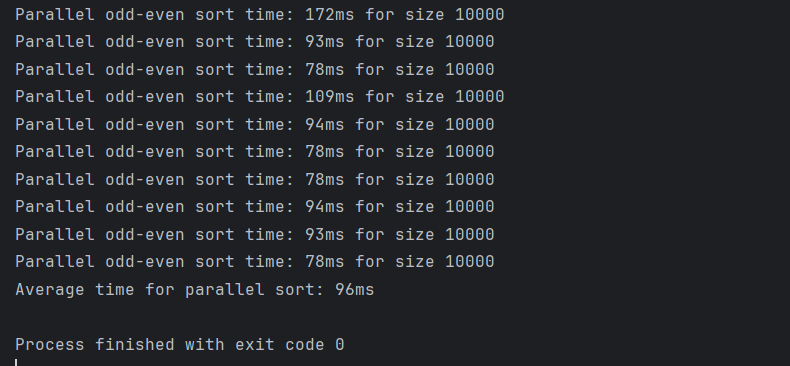


Рисунок 4.3 – Аналіз швидкодії паралельного алгоритму для 10000 елементів

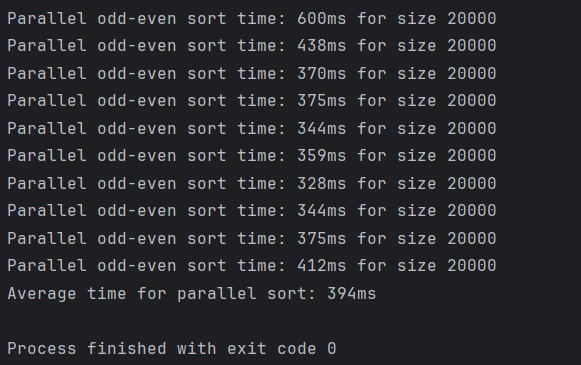


Рисунок 4.4 – Аналіз швидкодії паралельного алгоритму для 20000 елементів

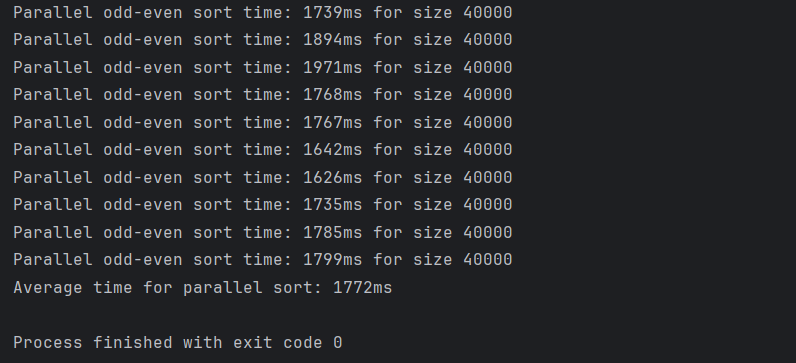


Рисунок 4.5 – Аналіз швидкодії паралельного алгоритму для 40000 елементів

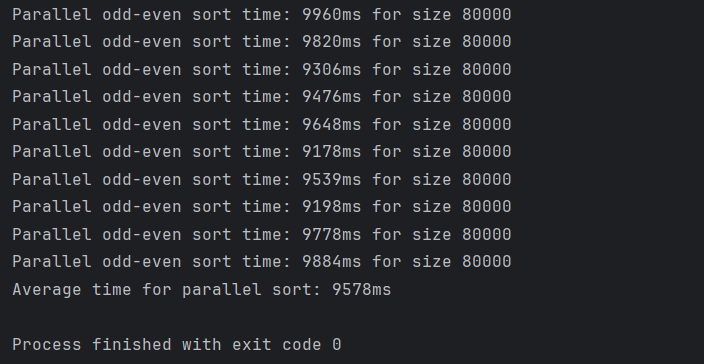


Рисунок 4.6 – Аналіз швидкодії паралельного алгоритму для 80000 елементів

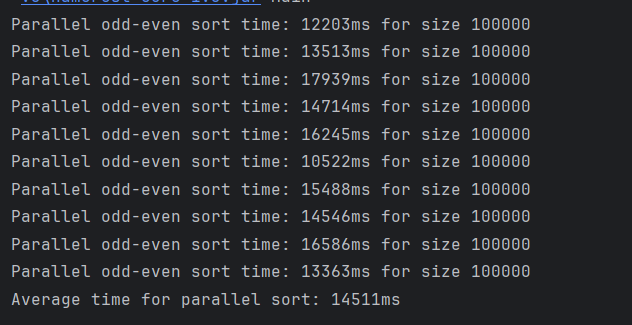


Рисунок 4.7 – Аналіз швидкодії паралельного алгоритму для 100000 елементів

Результати аналізу можна побачити на таблиці 4.1.

Таблиця 4.1 – Аналіз швидкодії паралельного алгоритму

|  |  |
| --- | --- |
| Кількість елементів в масиві | Середній час алгоритму, мілісекунди |
| 10000 | 96 |
| 20000 | 394 |
| 40000 | 1772 |
| 80000 | 9578 |
| 10000 | 14511 |

Для більшої наглядності результатів зобразимо отримані дані у вигляді графіку. Графік можна побачитти на рисунку 4.8.

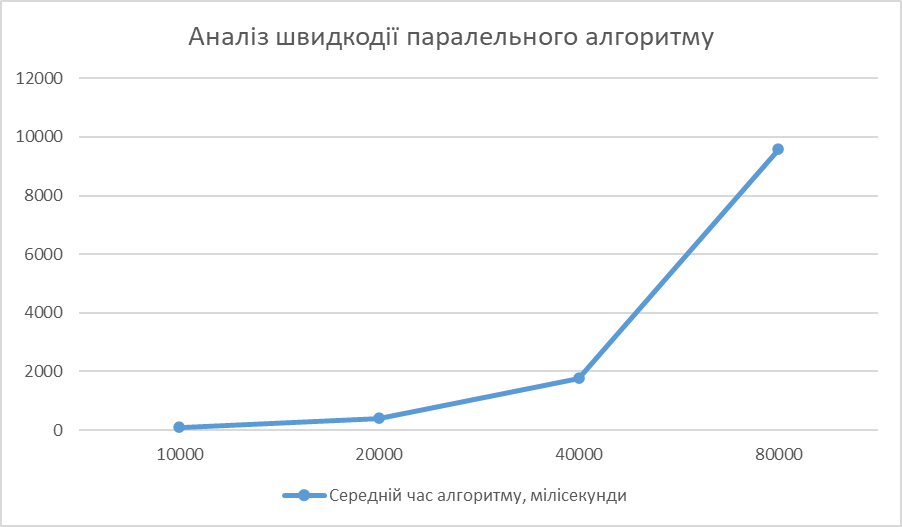


Рисунок 4.8 – Графік аналізу швидкодії алгоритму

Проаналізувавши усі наявні результати, можемо прийти до висновку, що паралельний алгоритм є значно швидшим за послідовний, що і було завданням для реалізації. Враховуючи наявний графік, можемо підтвердити, раніше написану часову складність алгоритму O(n2), адже графік знов ближчий до параболічного.

# 5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

## 5.1 Опис обладнання для проведення експериментів

Усі тестування було проведено на портативному персональному комп'ютері, який має процесор Intel(R) Core(TM) i7-9750H, 6 ядер, 16 гб ОЗУ, ліцензійну операційну систему Windows 11.

Для наступних експериментів збільшимо кількість сортувань з 10 до 20, та залишимо вирахування середнього арифметичного для порівнянь.

## 5.2 Дослідження ефективності паралельних обчислень алгоритму

Для дослідження ефективності треба порівняти час послідовного та паралельного алгоритму з 5 потоками та роздільним значенням – 4.

Таблиця 5.1 – Дослідження часу виконання паралельного та послідовного алгоритмів (5 потоків, роздільне значення – 4)

|  |  |  |
| --- | --- | --- |
| Кількість елементів | Час послідовного алгоритму, мілісекунд | Час паралельного алгоритму, мілісекунд |
| 10000 | 203 | 89 |
| 20000 | 833 | 332 |
| 40000 | 5033 | 1589 |
| 80000 | 36416 | 9255 |
| 100000 | 62082 | 14808 |

Далі обраховуємо показники ефективності. Прискорення можна порахувати за формулою (1.1)

(1.1)

Таблиця 5.2 - Порівняння прискорення алгоритму при різній кількості потоків та роздільного значення.

|  |  |
| --- | --- |
| Кількість елементів | Прискорення |
| 10000 | 2,28 |
| 20000 | 2,5 |
| 40000 | 3,17 |
| 80000 | 3,93 |
| 100000 | 4,19 |

Використовуючи дані, отримані раніше, можна подумати, що якщо роздільне значення менше за кількість потоків, то прискорення буде високим. Перевіримо цю теорію, обравши показник роздільного значення вищий за кількість потоків.

Таблиця 5.3 – Дослідження часу виконання паралельного та послідовного алгоритмів (5 потоків, роздільне значення – 6)

|  |  |  |
| --- | --- | --- |
| Кількість елементів | Час послідовного алгоритму, мілісекунд | Час паралельного алгоритму, мілісекунд |
| 10000 | 194 | 89 |
| 20000 | 468 | 225 |
| 40000 | 1684 | 747 |
| 80000 | 7579 | 2745 |
| 100000 | 17345 | 5502 |

Далі обраховуємо прискорення за формулою (1.1)

Таблиця 5.4 - Порівняння прискорення виконання паралельного та послідовного алгоритмів (5 потоків, роздільне значення – 6)

|  |  |
| --- | --- |
| Кількість елементів | Прискорення |
| 10000 | 2,18 |
| 20000 | 2,08 |
| 40000 | 2,25 |
| 80000 | 2,76 |
| 100000 | 3,15 |

Для того, щоб мати можливість зробити висновки, порівняємо час виконання алгоритмів з більшою різницею у значеннях.

Таблиця 5.5 – Дослідження часу виконання паралельного та послідовного алгоритмів (5 потоків, роздільне значення – 10)

|  |  |  |
| --- | --- | --- |
| Кількість елементів | Час послідовного алгоритму, мілісекунд | Час паралельного алгоритму, мілісекунд |
| 10000 | 162 | 84 |
| 20000 | 383 | 223 |
| 40000 | 1320 | 726 |
| 80000 | 6182 | 2799 |
| 100000 | 12437 | 5185 |

Далі обраховуємо прискорення за формулою (1.1)

Таблиця 5.6 - Порівняння прискорення виконання паралельного та послідовного алгоритмів (5 потоків, роздільне значення – 10)

|  |  |
| --- | --- |
| Кількість елементів | Прискорення |
| 10000 | 1,93 |
| 20000 | 1,71 |
| 40000 | 1,81 |
| 80000 | 2,21 |
| 100000 | 2,39 |

Отже, показник роздільного значення впливає на прискорення. Якщо роздільне значення менше за кількість потоків, прискорення вище. Однак, якщо роздільне значення більше за кількість потоків, прискорення знижується для середніх значень. З порівнянь можна побачити позитивну тенденцію збільшення пришвидшення зі збільшенням кількості елементів. Це означає, що прискорення зростає зі збільшенням кількості елементів. Звідси, паралельний алгоритм є більш ефективним ніж послідовний для обробки великих наборів даних не залежно від роздільного значення, хоча більше роздільне значення надає перевагу послідовному алгоритму.

Доказавши те, що різниця показника роздільного значення та кількості потоків не суттєво впливає на прискорення, можемо її не додавати до порівнянь. Надалі роздільне значення буде рівне кількості потоків для всіх досліджень.

Таблиця 5.7 - Порівняння прискорення виконання паралельного та послідовного алгоритмів (2 потоки)

|  |  |  |
| --- | --- | --- |
| Кількість елементів | Час послідовного алгоритму, мілісекунд | Час паралельного алгоритму, мілісекунд |
| 10000 | 331 | 85 |
| 20000 | 854 | 209 |
| 40000 | 3434 | 675 |
| 80000 | 17389 | 2652 |
| 100000 | 41004 | 5147 |

Таблиця 5.8 - Порівняння прискорення виконання паралельного та послідовного алгоритмів (3 потоки)

|  |  |  |
| --- | --- | --- |
| Кількість елементів | Час послідовного алгоритму, мілісекунд | Час паралельного алгоритму, мілісекунд |
| 10000 | 257 | 93 |
| 20000 | 648 | 218 |
| 40000 | 2461 | 685 |
| 80000 | 13620 | 2703 |
| 100000 | 31229 | 5436 |

Таблиця 5.9 - Порівняння прискорення виконання паралельного та послідовного алгоритмів (4 потоки)

|  |  |  |
| --- | --- | --- |
| Кількість елементів | Час послідовного алгоритму, мілісекунд | Час паралельного алгоритму, мілісекунд |
| 10000 | 226 | 94 |
| 20000 | 601 | 221 |
| 40000 | 2230 | 740 |
| 80000 | 10371 | 2871 |
| 100000 |  |  |

Таблиця 5.10 - Порівняння прискорення виконання паралельного та послідовного алгоритмів (5 потоків)

|  |  |  |
| --- | --- | --- |
| Кількість елементів | Час послідовного алгоритму, мілісекунд | Час паралельного алгоритму, мілісекунд |
| 10000 |  |  |
| 20000 |  |  |
| 40000 |  |  |
| 80000 |  |  |
| 100000 |  |  |

Таблиця 5.11 - Порівняння прискорення виконання паралельного та послідовного алгоритмів (6 потоків)

|  |  |  |
| --- | --- | --- |
| Кількість елементів | Час послідовного алгоритму, мілісекунд | Час паралельного алгоритму, мілісекунд |
| 10000 |  |  |
| 20000 |  |  |
| 40000 |  |  |
| 80000 |  |  |
| 100000 |  |  |

Таблиця 5.12 - Порівняння прискорення виконання паралельного та послідовного алгоритмів (7 потоків)

|  |  |  |
| --- | --- | --- |
| Кількість елементів | Час послідовного алгоритму, мілісекунд | Час паралельного алгоритму, мілісекунд |
| 10000 |  |  |
| 20000 |  |  |
| 40000 |  |  |
| 80000 |  |  |
| 100000 |  |  |

Таблиця 5.13 - Порівняння прискорення виконання паралельного та послідовного алгоритмів (8 потоків)

|  |  |  |
| --- | --- | --- |
| Кількість елементів | Час послідовного алгоритму, мілісекунд | Час паралельного алгоритму, мілісекунд |
| 10000 |  |  |
| 20000 |  |  |
| 40000 |  |  |
| 80000 |  |  |
| 100000 |  |  |

Таблиця 5.14 - Порівняння прискорення виконання паралельного та послідовного алгоритмів (9 потоків)

|  |  |  |
| --- | --- | --- |
| Кількість елементів | Час послідовного алгоритму, мілісекунд | Час паралельного алгоритму, мілісекунд |
| 10000 |  |  |
| 20000 |  |  |
| 40000 |  |  |
| 80000 |  |  |
| 100000 |  |  |

Таблиця 5.15 - Порівняння прискорення виконання паралельного та послідовного алгоритмів (10 потоків)

|  |  |  |
| --- | --- | --- |
| Кількість елементів | Час послідовного алгоритму, мілісекунд | Час паралельного алгоритму, мілісекунд |
| 10000 |  |  |
| 20000 |  |  |
| 40000 |  |  |
| 80000 |  |  |
| 100000 |  |  |

Таблиця 5.16 – Порівняння прискорення залежно від кількості потоків

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Кількість елементів | Потоки | | | | | | | | |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 10000 | 3,89 | 2,76 | 2,4 |  |  |  |  |  |  |
| 20000 | 4,09 | 2,97 | 2,72 |  |  |  |  |  |  |
| 40000 | 5,09 | 3,59 | 3,01 |  |  |  |  |  |  |
| 80000 | 6,55 | 5,04 | 3,61 |  |  |  |  |  |  |
| 100000 | 7,97 | 5,74 |  |  |  |  |  |  |  |

# ВИСНОВКИ

У ході курсової роботи проведено дослідження швидкодії алгоритму сортування бульбашкою та розроблено його паралельну реалізацію використовуючи модифікації парне-непарне сортування та розділяй та володарюй мовою програмування Java.

Було виконано огляд існуючих реалізацій, який підкреслив актуальність дослідження та необхідність використання модифікацій.

Послідовний та паралельний алгоритм було протестовано для доведення коректності та швидкості роботи алгоритмів. Швидкодії алгоритмів досліджено при зростанні складності обчислень та вплив параметрів алгоритмів на прискорення. Експериментально встановлено, що прискорення паралельної реалізації становить від 1,7 до 7,97, що свідчить про її успішність.

В даній роботі паралельний алгоритм сортування має декілька переваг. Одна з найважливіших є швидкість, адже паралельні алгоритми використовують багатопоточність для виконання декількох операцій одночасно, що може значно прискорити виконання завдань, особливо на багатоядерних процесорах. Також, паралельні алгоритми дозволяють ефективніше використовувати ресурси системи, розподіляючи навантаження між доступними ядрами процесора.

Отже, результати дослідження підтверджують, що паралельна реалізація алгоритму сортування бульбашкою може ефективно оптимізувати обчислення в умовах обробки великих обсягів даних використовуючи додаткові модифікації. Аналіз показав потенціал мови програмування Java для реалізації паралельних обчислень та підтвердив її популярність для таких завдань.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Magnetic Bubble Sort Algorithm. Obed Appiah and Ezekiel Mensah Martey. Режим доступу до ресурсу: <https://www.researchgate.net/profile/Obed-Appiah-4/publication/280114536_Magnetic_Bubble_Sort_Algorithm/links/55aac6ad08aea9946724161e/Magnetic-Bubble-Sort-Algorithm.pdf>
2. Automatic Parallelization of Divide and Conquer Algorithms. Radu Rugina and Martin Rinard. Режим доступу до ресурсу: <https://dl.acm.org/doi/pdf/10.1145/329366.301111>

# ДОДАТКИ

## Додаток А. Код алгоритмів

Код на GitHub: <https://github.com/valeriiaradzivilo/parallel_course_work/tree/master>

## Додаток Б. Назва додатку

## Додаток В. Назва додатку