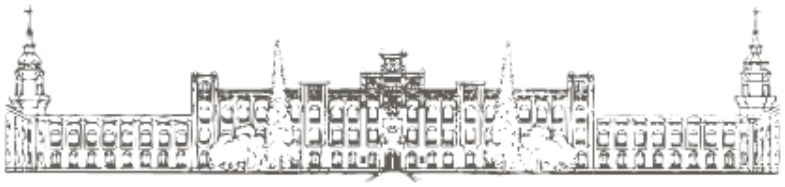

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту
«Технології паралельних обчислень. Курсова робота»
Тема: Алгоритм сортування (bucket sort) та його паралельна реалізація на Node.js

**Керівник**:
ст. викладач
Дифучин Антон Юрійович

«Допущено до захисту»
_____

«___» _____ 2024 р.

Захищено з оцінкою
_____

Члени комісії:

_____

_____

**Виконавець**:
Петров Ігор Ярославович
студент групи ІП-13
залікова книжка № ІП-1327
_____

«26» травня 2024 р.

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 38 сторінок, 5 розділів, містить 19 рисунків, 4 таблиці, 1 додаток, 5 джерел.

**Мета.** Реалізувати та проаналізувати паралельну реалізацію алгоритму сортування корзинами на Node.js з використанням базового функціоналу JavaScript і модулю Worker Threads, що допоможе оптимізувати роботу алгоритму у виконанні задачі сортування.

Робота включає п'ять розділів, кожен з яких описує алгоритм та його відомі паралельні реалізації. Також проводиться розробка послідовного алгоритму та аналіз його продуктивності. Вибирається та описується програмне забезпечення для розробки паралельних обчислень. Розробляється паралельний алгоритм з використанням обраного програмного забезпечення, та проводиться дослідження ефективності паралельних обчислень алгоритму.

КЛЮЧОВІ СЛОВА: СОРТУВАННЯ BUCKET SORT, СОРТУВАННЯ КОРЗИНАМИ, NODE.JS, WORKER_THREADS, JAVASCRIPT

# ЗМІСТ

# ВСТУП

У сучасному світі, де обробка великих обсягів даних стає все більш актуальною, ефективні алгоритми сортування набувають великого значення. Одним із таких алгоритмів є "алгоритм сортування корзинами" або "bucket sort". Цей алгоритм відноситься до категорії "розподільних" сортувань, які використовуються для сортування даних, розподілення їх у певні групи (корзини) та подальшого сортування кожної групи окремо.

Однак, існують випадки, коли потрібно обробляти великі обсяги даних на багатоядерних або розподілених системах. У таких випадках паралельна реалізація алгоритмів стає ключовою для забезпечення швидкодії та ефективності обробки даних.

Ця курсова робота спрямована на дослідження алгоритму сортування корзинами та розробку його паралельної реалізації на платформі Node.js. Реалізація на Node.js може бути нетривіальною задачею через однопоточність самої мови, однак для покращення ефективності використання ресурсів системи можна використовувати мультипроцесинг.

В цілому, мета цієї роботи полягає в дослідженні ефективності паралельної реалізації алгоритму сортування корзинами на платформі Node.js порівняно з його послідовною реалізацією.

# 1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

## 1.1 Опис основного алгоритму сортування

Алгоритм сортування корзинами (Bucket Sort)[1] - це алгоритм сортування, який використовує ідею розподілу елементів масиву у "корзини" на основі їх значень, а потім сортує кожну корзину окремо, використовуючи інший алгоритм сортування.

Основні кроки алгоритму сортування корзинами:

- Розподіл у відповідні корзини: Спочатку створюються певна кількість корзин, а потім кожен елемент вихідного масиву розподіляється до відповідної корзини згідно зі значенням.
- Сортування кожної корзини: Після розподілу всі елементи, що потрапили в одну корзину, сортуються за допомогою будь-якого іншого алгоритму.
- Об'єднання корзин: Нарешті, всі відсортовані елементи кожної корзини об'єднуються в один великий відсортований масив.

Псевдокод алгоритму:

```
function bucketSort(array, num_buckets):
    minValue = array[0]
    maxValue = array[0]

    for i from 1 to length(array) - 1:
        if array[i] < minValue:
            minValue = array[i]
        else if array[i] > maxValue:
            maxValue = array[i]
    interval = (maxValue - minValue + 1) / num_buckets
    buckets = Array(num_buckets, [])
```

```
for i from 0 to length(array) - 1:
    bucket_index = floor((array[i] - minValue) / interval)
    index = bucket_index == num_buckets ? num_buckets - 1 : bucket_index
    push array[i] to buckets[index]
sortedArray = []
for each bucket in buckets:
    bubbleSort(bucket)
    append bucket to sortedArray
return sortedArray
```

### 1.2 Опис алгоритму сортування корзин

Основним алгоритмом для сортування вже розбитих корзин було обрано сортування бульбашкою[2]. Це простий метод сортування, що порівнює пари сусідніх елементів масиву та, якщо вони знаходяться в неправильному порядку, обмінює їх. Цей процес повторюється доти, доки масив не буде відсортований. Асимптотична складність алгоритму складає $O(n^2)$, де n - кількість елементів у масиві.

Основні кроки алгоритму сортування бульбашкою:

- Проходи по масиву: Починаючи з початку масиву, алгоритм порівнює кожну пару сусідніх елементів.
- Порівняння і обмін елементів: Якщо елементи знаходяться в неправильному порядку (наприклад, елемент і більший за елемент i+1 у випадку сортування за зростанням), то вони обмінюються місцями.
- Повторення ітерації: Ці кроки повторюються для всього масиву до тих пір, поки всі елементи не будуть розташовані в правильному порядку.
- Кінець сортування: Алгоритм завершує роботу, коли під час одного повного проходу по масиву не відбувається жодного обміну елементів.

Сортування бульбашкою було обрано для забезпечення більшого навантаження на процеси, що дозволить наочно побачити плюси паралельної реалізації для основного алгоритму на великих обсягах даних.

Псевдокод алгоритму:

```
function bubbleSort(arr):
    len = length(arr)
    for i from 0 to len - 2:
        for j from 0 to len - 2 - i:
            if arr[j] > arr[j + 1]:
                temp = arr[j]
                arr[j] = arr[j + 1]
                arr[j + 1] = temp
    return arr
```

### 1.3 Опис паралельних реалізацій алгоритму

Паралельна реалізація алгоритму сортування корзинами передбачає розділення завдання на декілька підзадач, які можна виконувати паралельно. У контексті сортування корзинами, це означає, що розділення елементів у вихідному масиві на корзини та сортування кожної корзини можуть відбуватися паралельно.

Паралельне розділення і заповнення корзин не є вигідним по часу і ресурсам, тому найкращим рішенням буде зосередити увагу на обчисленнях, які вимагають найбільших обчислювальних потужностей, а саме сортування корзин.

Для реалізації паралельних обчисленнь у Node.js існує три відомих варіанти: Child Processes, Worker Threads, Clustering. Детальніше розглянемо кожен з них.

Child Processes дозволяють запускати окремі процеси для виконання важких обчислень, але вони можуть мати значні затримки через міжпроцесовий зв'язок і вимагають додаткової пам'яті та ресурсів.

Worker Threads забезпечують можливість створювати багатопоточні обчислення без міжпроцесового зв'язку, що дозволяє ефективніше використовувати ресурси обчислювальної системи.

Clustering дозволяє використовувати всю потужність багатоядерних систем для обробки багатьох запитів одночасно, але потребує ретельного керування станом додатку та обміном станом між процесами через міжпроцесовий зв'язок.

З усіх цих методів, Worker Threads є найбільш швидким і ефективним для Bucket Sort. Вони надають зручний інтерфейс для створення багатопоточних обчислень без зайвого для цієї задачі міжпроцесового зв'язку. Крім того, вони можуть ефективно використовувати всі ресурси багатоядерних систем, що робить їх ідеальним вибором для паралельного виконання великих обчислень або обробки великих обсягів даних у Node.js додатках. Також для досягнення синхронізації обчислень будуть використані базові інструменти JavaScript: асинхронні функції і новий синтаксис async/await та Promises.

# 2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

## 2.1 Розробка послідовного алгоритму

Перш ніж приступати до паралельної реалізації алгоритму Bucket Sort, важливо спочатку створити його послідовну версію. Послідовна реалізація дозволяє глибше зрозуміти основні принципи алгоритму, включаючи розподіл елементів по корзинам, сортування кожної корзини та злиття відсортованих корзин у фінальний результат. Вона є простішою для відлагодження та тестування, що допомагає виявити та виправити помилки в логіці алгоритму, забезпечуючи стабільну основу для подальших оптимізацій. Також послідовний алгоритм є більш ефективним для невеликих обсягів даних, оскільки розділення на підзадачі і передача даних між процесами також займає час і ресурси.

На рисунку 2.1. наведено паралельну реалізацію алгоритму сортування.

```
import bubbleSort from "./bubbleSort.js";

export default function bucketSort(array, num_buckets) {
    if (array.length === 0) {
        return array;
    }

    let minValue = array[0];
    let maxValue = array[0];

    for (let i = 1; i < array.length; i++) {
        if (array[i].accountBalance < minValue.accountBalance) {
            minValue = array[i];
        } else if (array[i].accountBalance > maxValue.accountBalance) {
            maxValue = array[i];
        }
    }

    const interval = (maxValue.accountBalance - minValue.accountBalance + 1) / num_buckets;
    const buckets = Array.from({ length: num_buckets }, () => []);

    for (let i = 0; i < array.length; i++) {
        const bucket_index = Math.floor((array[i].accountBalance - minValue.accountBalance) / interval);
        const index = bucket_index === num_buckets ? num_buckets - 1 : bucket_index;
        buckets[index].push(array[i]);
    }

    const sortedArray = [];

    buckets.forEach(bucket => {
        bubbleSort(bucket);
        sortedArray.push(...bucket);
    });

    return sortedArray;
}
```

Рисунок 2.1. – код послідовного алгоритму сортування корзинами.

Також для сортування корзин використовується послідовний алгоритм сортування бульбашкою, наведений на рисунку 2.2.

```
export default function bubbleSort(arr) {
    const len = arr.length;
    for (let i = 0; i < len - 1; i++) {
        for (let j = 0; j < len - 1 - i; j++) {
            if (arr[j].accountBalance > arr[j + 1].accountBalance) {
                let temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    return arr;
}
```

Рисунок 2.2. – код послідовного алгоритму сортування бульбашкою.

Перейдемо до покрокового опису роботи основного алгоритму.

Першим кроком є перевірка, чи не є вхідний масив array порожнім; якщо так, функція негайно повертає цей масив, завершуючи виконання. Далі, алгоритм визначає мінімальне (minValue) і максимальне (maxValue) значення властивості accountBalance серед елементів масиву, ініціалізуючи обидва значення першим елементом масиву і проходячи через масив для оновлення цих значень.

Після визначення мінімального і максимального значень, алгоритм розраховує інтервал, який визначає діапазон значень, що потрапляють в кожну корзину. Створюється масив buckets, що складається з переданого у функцію num_buckets порожніх підмасивів. Кожен елемент вхідного масиву розподіляється у відповідну корзину на основі його значення accountBalance.

Після розподілу елементів по корзинам, алгоритм створює порожній масив sortedArray для збирання відсортованих елементів. Кожна корзина сортується окремо за допомогою функції bubbleSort, а відсортовані елементи додаються до sortedArray. Нарешті, відсортований масив sortedArray, що містить елементи всіх корзин, повертається як результат роботи функції. Таким чином, алгоритм розподіляє елементи масиву по корзинам на основі їх значень, сортує кожну корзину окремо і об'єднує відсортовані елементи в кінцевий масив.

Перейдемо до опису вспоміжного алгоритму сортування корзин.

Першим кроком є визначення довжини масиву len, щоб використовувати її у подальших циклах. Алгоритм використовує вкладені цикли для проходження через масив та порівняння сусідніх елементів.

Зовнішній цикл проходить через масив len - 1 разів, що забезпечує необхідну кількість проходів для гарантованого сортування. Внутрішній цикл проходить через масив від початку до останнього невідсортованого елемента, визначеного як len - 1 - i. Це означає, що з кожним проходом зовнішнього циклу порівнюється все менша кількість елементів, оскільки найбільші елементи "спливають" на свої правильні позиції.

Всередині внутрішнього циклу відбувається порівняння поточного елемента arr[j] з наступним arr[j + 1] за значенням accountBalance. Якщо поточний елемент більший за наступний, вони обмінюються місцями, використовуючи тимчасову змінну temp. Це забезпечує поступове "спливання" найбільших елементів масиву до кінця масиву з кожним проходом циклу. Після завершення всіх проходів циклів, відсортований масив повертається.

### 2.2 Аналіз швидкодії послідовного алгоритму

Для цього проведемо декілька замірів на різних розмірностях масиву та проаналізуємо отримані результати.

Перед виконанням контрольних замірів буде проведено прогрівання, його алгоритм представлений на рисунку 2.3.

```
async function warmUp() {
    const arraySize = 1000;
    const array = generateRandomArray(arraySize);
    for (let i = 0; i < 10; i++) {
        await bucketSort(array, 6);
        await parallelBucketSort(array, 6);
    }
}
```

Рисунок 2.3. – код для прогрівання програми

Функція прогрівання запускає послідовний і паралельний алгоритм по 10 разів на невеликих обсягах даних. Це допоможе налаштувати середовище

виконання, зокрема, зменшити вплив таких факторів, як оптимізації віртуальної машини JavaScript (V8), завантаження кешу та інші фактори, які можуть впливати на перші виконання алгоритму.

Основне тестування буде проводитись на 20 ітераціях на копіях одного й того самого масиву. Також у масиві будуть використані об’єкти зі значенням поля для порівняння від 1 до 1000, масив буде розбито на 10 корзин. Після проведення експерименту функція для заміру часу (рисунок 2.4.) виведе середній час і поверне результат виконання сортування.

```
export default async function measureExecutionTime(func, ...args) {
    const runs = 20;
    let totalTime = 0;
    let result;

    for (let i = 0; i < runs; i++) {
        const start = performance.now();
        if (i === 0){
            result = await func(...args);
        }
        else {
            await func(...args);
        }
        const end = performance.now();
        totalTime += end - start;
    }

    const averageTime = totalTime / runs;
    console.log(`Function ${func.name} executed in average ${Math.ceil(averageTime)} milliseconds over ${runs} runs`);
    return result;
}
```

Рисунок 2.4. – функція заміру часу роботи алгоритму

Далі масив буде перевірено на правильність сортування за допомогою іншої функції (рисунок 2.5).

```
export default function isSorted(arr) {
    return arr.every((v, i, a) => !i || a[i - 1].accountBalance <= v.accountBalance);
}
```

Рисунок 2.5. – функція перевірки впорядкованості масиву

Приклад використання у головному файлі наведений на рисунку 2.6.

```
import measureExecutionTime from "./utils/measureExecutionTime.js";
import generateRandomArray from "./utils/generateArray.js";
import isSorted from "./utils/isSorted.js";
import "dotenv/config"

async function warmUp() {
    const arraySize = 1000;
    const array = generateRandomArray(arraySize);
    for (let i = 0; i < 10; i++) {
        await bucketSort(array, 5);
        // await parallelBucketSort(array, 5);
    }
}

async function main() {
    const arraySize = parseInt(process.env.ARRAY_SIZE, 10);
    const array = generateRandomArray(arraySize);

    await warmUp();

    const sortedSequential = await measureExecutionTime(bucketSort, array, 10);
    // const sortedParallel = await measureExecutionTime(parallelBucketSort, array, 10);

    console.log(`Is sequential sorted: ${isSorted(sortedSequential)}`);
    // console.log(`Is parallel sorted: ${isSorted(sortedParallel)}`);
    // console.log(`Are equal: ${JSON.stringify(sortedSequential) === JSON.stringify(sortedParallel)}`)
}

main().catch(console.error);
```

Рисунок 2.6. – головний файл index.js

Результати пробного експерименту можна побачити на рисунку 2.7.

```
PS C:\Users\igorp\OneDrive\Рабочий стол\ParallelBucketSort> npm run dev

> coursework@1.0.0 dev
> node index.js

Function bucketSort executed in average 3687 milliseconds over 20 runs
Is sequential sorted: true
```

Рисунок 2.7. – Приклад результату роботи алгоритму на 100000 елементів

Далі будуть проведені експерименти з різною кількістю елементів масиву і відображені у таблиці (таблиця 2.1.).

Таблиця 2.1. – Результати тестування

| Кількість елементів | Час послідовного алгоритму, мілісекунд |
|---|---|
| 10000 | 50 |
| 50000 | 1274 |
| 100000 | 6105 |
| 125000 | 10167 |
| 175000 | 24497 |
| 200000 | 37142 |
| 300000 | 117529 |

Зі збільшенням кількості елементів час сортування значно зростає. Це відображається у збільшенні часу виконання програми при збільшенні розміру вхідних даних. Подивившись на величини часу, можна помітити квадратичну залежність часу від кількості елементів. Це відбувається за рахунок використання Bubble Sort для сортування кожної корзини.

Побудуємо графік залежності (рисунок 2.7.).

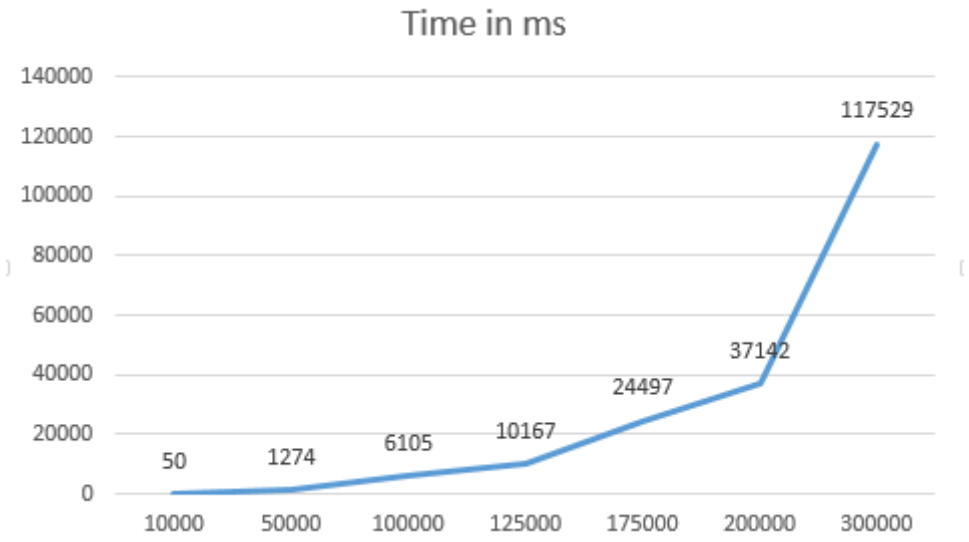


Рисунок 2.7 – Графік залежності часу сортування від розміру масива

Графік показує, що час (у мілісекундах) зростає не лінійно, а з прискоренням, що характерно для квадратичних залежностей. Зокрема, значення часу зростають набагато швидше при більших розмірностях масиву, що вказує на квадратичний характер залежності між змінними. Це обумовлено використанням бульбашкового сортування для корзин.

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

При розробці алгоритму було використано Node.js з базовою мовою програмування JavaScript. Ця мова є однопоточною, тому паралелізація була виконана за допомогою мультипроцесингу через Worker Threads, що перекладається, як “Робочі Потоки”, однак фактично вони використовують мультипроцесинг, а не мультипоточність.

Node.js - це платформа з відкритим вихідним кодом для створення швидких і масштабованих мережевих застосунків на JavaScript. Вона базується на двигуні V8, розробленому Google для браузера Chrome. Однією з ключових особливостей Node.js є можливість виконувати JavaScript на серверному боці, що дозволяє запустити код на комп’ютері замість браузера. Node.js має неблокуючий, подійно-орієнтований архітектурний стиль, що дозволяє ефективно обробляти велику кількість одночасних запитів і забезпечувати високу продуктивність застосунків. Він також підтримує широкий спектр модулів і бібліотек, що спрощує розробку і розширення застосунків.

Модуль worker_threads в Node.js фактично реалізує мультипроцесинг для нашої задачі. За допомогою цього модуля можна створювати нові робочі процеси, передавати їм дані та отримувати результати їх виконання. Це особливо корисно для виконання важких обчислень, які можуть заблокувати основний потік виконання, таких як обробка великих обсягів даних або інтенсивні операції обробки. У порівнянні з мультипоточністю, де потоки виконуються в межах одного процесу та ділять загальну пам'ять, мультипроцесинг, який реалізує бібліотека worker_threads, може забезпечити кращу ізоляцію між різними частинами програми, але вимагає більшого обсягу ресурсів для кожного процесу.

У якості середовища розробки було обрано Visual Studio Code[3]. Він надає ряд корисних можливостей, включаючи інтегровану систему керування версіями, підтримку розширень (extensions) для роботи з різними мовами програмування та фреймворками, також він є дуже легким і практично не потребує ресурсів комп’ютеру у порівнянні з аналогами JetBrains[4].

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

## 4.1 Проєктування

Паралельна реалізація алгоритму сортування відбувається за допомогою розподілення роботи між кількома робітниками (workers). Алгоритм сортування розділяє вхідний масив на кілька корзин. Кількість корзин визначається параметром у функції.

Кожен робітник має отримати свій набір корзин, які він сортує незалежно від інших. Після завершення сортування кожного робітника, вони повертають відсортовані підмасиви (chunks) даних назад до основного потоку виконання за допомогою обміну повідомленнями.

Необхідно також забазпечити синхронізацію, щоб основний потік чекав, поки усі воркери завершать свою роботу, а потім об'єднає всі відсортовані підмасиви в один відсортований масив, який і повертається, як результат роботи.

Цей підхід дозволяє використовувати кілька процесорів чи ядер для паралельного оброблення різних частин вхідних даних, що зменшує час виконання сортування порівняно зі стандартним однопоточним підходом.

Детальніше опишемо кроки виконання алгоритму:

- Початковий масив даних розділяється на менші підмасиви у певних діапазонах чисел, розмір діапазонів залежить від переданої кількості бакетів, кожен підмасив буде сортуватись окремо за допомогою Bubble Sort.
- Після розділення на корзини головний процес рівномірно розподіляє корзини за процесами.
- Корзини сортуються паралельно і повертаються назад у головний процес.
- Головний процес очікує виконання задач усіх воркерів і збирає відсортовані корзини у результуючий масив.

### 4.2 Програмна реалізація

Початкове розбиття на корзини відбувається так само, як і у послідовному алгоритмі (рисунок 4.1.).

```
export default function parallelBucketSort(array, num_buckets) {
    return new Promise((resolve, reject) => {
        if (array.length === 0) {
            resolve(array);
            return;
        }

        let minValue = array[0];
        let maxValue = array[0];

        for (let i = 1; i < array.length; i++) {
            if (array[i].accountBalance < minValue.accountBalance) {
                minValue = array[i];
            } else if (array[i].accountBalance > maxValue.accountBalance) {
                maxValue = array[i];
            }
        }

        const interval = (maxValue.accountBalance - minValue.accountBalance + 1) / num_buckets;
        const buckets = Array.from({ length: num_buckets }, () => []);

        for (let i = 0; i < array.length; i++) {
            const bucket_index = Math.floor((array[i].accountBalance - minValue.accountBalance) / interval);
            const index = bucket_index === num_buckets ? num_buckets - 1 : bucket_index;
            buckets[index].push(array[i]);
        }
```

Рисунок 4.1 – Початкове розбиття на корзини

У паралельному алгоритмі сортування відбувається паралельно за допомогою розділення роботи між кількома робітниками (workers). Замість послідовного сортування кожної корзини, кожна корзина передається окремому робітнику, який відповідає за сортування цієї конкретної картки. Після того, як всі робітники завершують сортування своїх корзин, відсортовані корзини об'єднуються разом у відсортований вихідний масив.

Для справедливого і ефективного розподілення корзин між процесами ми отримаємо кількість ядер процесора і створимо таку саму кількість воркерів або, якщо кількість ядер перевищує кількість корзин, то створимо по процесу на кожну корзину. Також на основі цього числа розподілимо корзини за процесами і також врахуємо випадок, коли перередана кількість корзин не ділиться на кількість ядер (рисунок 4.2.).

```
const cpuCount = os.cpus().length;
const workers = Math.min(cpuCount, num_buckets);
const promises = [];

const bucketsPerWorker = Math.floor(num_buckets / workers);
const extraBuckets = num_buckets % workers;
```

Рисунок 4.2 – Визначення числа корзин на процес

Далі головний процес розподіляє корзини за воркерами і приймає від них повідомлення з відсортованими корзинами (рисунок 4.3.).

```
let offset = 0;
for (let i = 0; i < workers; i++) {

    const end = offset + bucketsPerWorker + (i < extraBuckets ? 1 : 0);
    const workerBuckets = buckets.slice(offset, end);
    offset = end;

    promises.push(
        new Promise((resolve, reject) => {

            const worker = new Worker(new URL("./worker.js", import.meta.url), {
                workerData: workerBuckets,
            });

            worker.on("message", resolve);
            worker.on("error", reject);
            worker.on("exit", (code) => {
                if (code !== 0) {
                    reject(new Error(`Worker stopped with exit code ${code}`));
                }
            });
        })
    );
}
```

Рисунок 4.3 – Розподілення корзин і отримання відсортованих чанків головним процесом

На рисунку 4.4 відображено отримання воркером частини бакетів і їх сортування за допомогою бульбашки. Після сортування кожен воркер поверне свою частину відсортованих бакетів.

```
import { parentPort, workerData } from "worker_threads";
import bubbleSort from "../bubbleSort.js";

const sortedBuckets = workerData.map(bucket => {
    bubbleSort(bucket);
    return bucket;
});

parentPort.postMessage(sortedBuckets);
```

Рисунок 4.4 – Сортування бакетів у воркері

У кінці функція очікує за допомогою JavaScript Promise.all()[5] на закінчення роботи усіх процісів, збирає відсортовані бакети у результуючий масив і повертає його (рисунок 4.5.).

```
        Promise.all(promises)
            .then((sortedBuckets) => {
                const sortedArray = [];
                sortedBuckets.forEach((sortedChunk) => {
                    sortedChunk.forEach((bucket) => {
                        sortedArray.push(...bucket);
                    });
                });
                resolve(sortedArray);
            })
            .catch(reject);
    });
```

Рисунок 4.5 – Збірка фінального масиву і повернення результату

### 4.3 Тестування алгоритму

Для цього виконаємо ті ж самі кроки, що були описані під час тестування послідовного алгоритму. Виконаємо розігрів, проведемо експеримент на 20 повтореннях з копіями масиву для різних розмірностей масиву і отримаємо середній час виконання. Наприкінці перевіримо коректність сортування.

Приклад використання у головному файлі наведений на рисунку 4.6.

```
import measureExecutionTime from "./utils/measureExecutionTime.js";
import generateRandomArray from "./utils/generateArray.js";
import isSorted from "./utils/isSorted.js";
import "dotenv/config"

async function warmUp() {
    const arraySize = 1000;
    const array = generateRandomArray(arraySize);
    for (let i = 0; i < 10; i++) {
        // await bucketSort(array, 5);
        await parallelBucketSort(array, 5);
    }
}

async function main() {
    const arraySize = parseInt(process.env.ARRAY_SIZE, 10);
    const array = generateRandomArray(arraySize);

    await warmUp();

    // const sortedSequential = await measureExecutionTime(bucketSort, array, 10);
    const sortedParallel = await measureExecutionTime(parallelBucketSort, array, 10);

    // console.log(`Is sequential sorted: ${isSorted(sortedSequential)}`);
    console.log(`Is parallel sorted: ${isSorted(sortedParallel)}`);
    // console.log(`Are equal: ${JSON.stringify(sortedSequential) === JSON.stringify(sortedParallel)}`)
}

main().catch(console.error);
```

Рисунок 4.6. – головний файл index.js

Приклад виконання коду наведено на рисунку 4.7.

```
PS C:\Users\igorp\OneDrive\Рабочий стол\ParallelBucketSort> npm run dev

> coursework@1.0.0 dev
> node index.js

Function parallelBucketSort executed in average 1384 milliseconds over 20 runs
Is parallel sorted: true
```

Рисунок 4.7. – Вивід програми для 100000 елементів

Далі будуть проведені експерименти з різною кількістю елементів масиву і відображені у таблиці (таблиця 4.1.). Тестування буде так само проходити на 10 корзинах з використанням 8-х процесів-воркерів. Тестування проводиться на процесорі AMD Ryzen 7 4600U, 8 ядер, 8 потоків.

Таблиця 4.1. – Результати тестування

| Кількість елементів | Час паралельного алгоритму, мілісекунд |
|---|---|
| 10000 | 79 |
| 50000 | 383 |
| 100000 | 1137 |
| 125000 | 1909 |
| 175000 | 4576 |
| 200000 | 7022 |
| 300000 | 17581 |

Функція демонструє ту ж саму асимптотику, що й послідовний алгоритм, однак тепер сортування виконується набагато швидше на великих обсягах даних. Невелике сповільнення відбувається лише на 10000, це відбувається через те, що виділення ресурсів на процеси займає більше часу ніж саме сортування.

Побудуємо графік залежності (рисунок 4.8.).

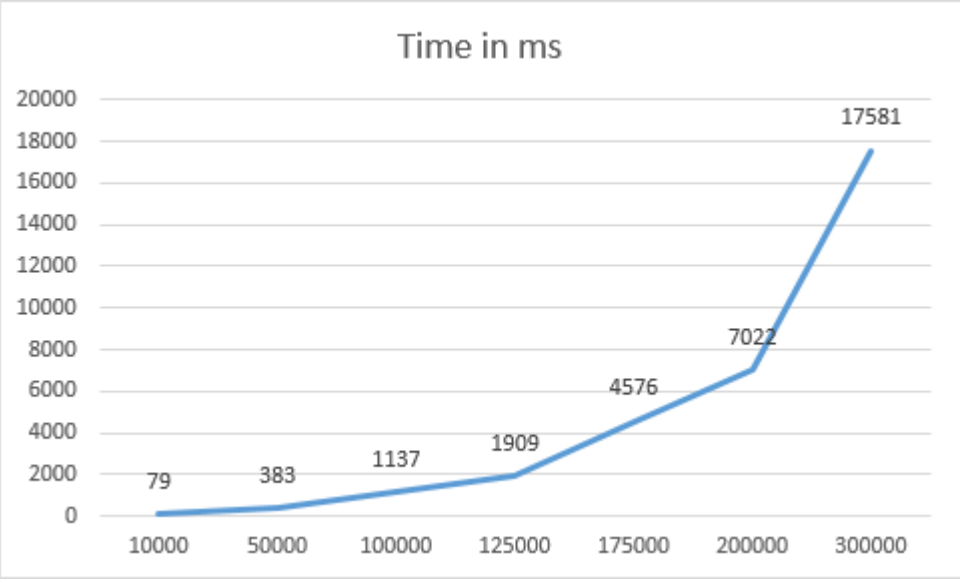


Рисунок 4.8 – Графік залежності часу сортування від розміру масива

Графік підтверджує заміри, можна також помітити, що зріст часу на розмірах від 50000 до 200000 відбувається більш плавно ніж у послідовному алгоритмі.

# 5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

## 5.1 Порівняння результатів паралельного та послідовного алгоритмів

Цей розділ присвячений дослідженню ефективності паралельної реалізації алгоритму сортування корзинами порівняно з його послідовним аналогом. Ми прагнемо визначити, наскільки розпаралелювання може покращити час виконання цього алгоритму та наскільки ефективним є використання паралельних ресурсів у цьому контексті. Для цього, перш за все, ми порівняємо середні результати послідовного алгоритму і паралельного на 8 процесах. У обох випадках масив буде розбито на 10 корзин. Для кожного експерименту (однієї розмірності масиву) буде виконано по 20 ітерацій алгоритму, а перед цим розігрів на невеликих обсягах даних.

Результати порівняння з прискоренням наведено у таблиці 5.1. Прискорення визначається за формулою (формула 5.1). Також прискорення буде округлено до 2 цифр після коми задля зручності.

Формула прискорення (5.1.)

$$a = \frac{seqTime}{parallelTime} \tag{5.1}$$

Таблиця 5.1. – Порівняння швидкодії алгоритму

| Кількість елементів | Час послідовного алгоритму, мілісекунд | Час паралельного алгоритму, мілісекунд | Коефіцієнт прискорення |
|---|---|---|---|
| 10000 | 50 | 79 | 0,63 |
| 50000 | 1274 | 383 | 3,32 |
| 100000 | 6105 | 1137 | 5,36 |
| 125000 | 10167 | 1909 | 5,32 |
| 175000 | 24497 | 4576 | 5,35 |
| 200000 | 37142 | 7022 | 5,28 |
| 300000 | 117529 | 17581 | 6,68 |

На основі проведених експериментів та отриманих результатів можна зробити кілька важливих висновків щодо ефективності паралельної реалізації алгоритму:

- Малий розмір масиву (до 10,000 елементів): Для невеликих розмірів масивів паралельна реалізація показала гірші результати, ніж послідовна. Це можна пояснити значними накладними витратами на управління потоками, які перевищують виграш від паралельного виконання.
- Середні та великі розміри масивів: Для масивів від 50,000 до 300,000 елементів паралельний алгоритм показав значне прискорення, з коефіцієнтом прискорення, що коливається від 3.32 до 5.35. Це вказує на те, що для таких розмірів даних паралельне виконання стає ефективним, оскільки виграш у часі обчислень перевищує накладні витрати на управління потоками.
- Загальна ефективність паралельної реалізації: Паралельна реалізація алгоритму сортування корзинами показує значне покращення продуктивності порівняно з послідовною реалізацією, особливо для середніх та великих обсягів даних.

Таке велике прискорення може бути пов’язане зі слабкістю кожного окремого логічного підпроцесора комп’ютера і їх великою кількістю, тому послідовний алгоритм виконується достатньо довго.

Загалом, можна вважати реалізацію успішною. Отримані результати вказують на значне покращення паралельної реалізації алгоритму сортування корзинами відносно послідовної та підтверджують важливість врахування розміру вхідних даних при виборі методу сортування.

Для більш наочної демонстрації результатів зробимо графік часу виконання обох алгоритмів (рисунок 5.1.).

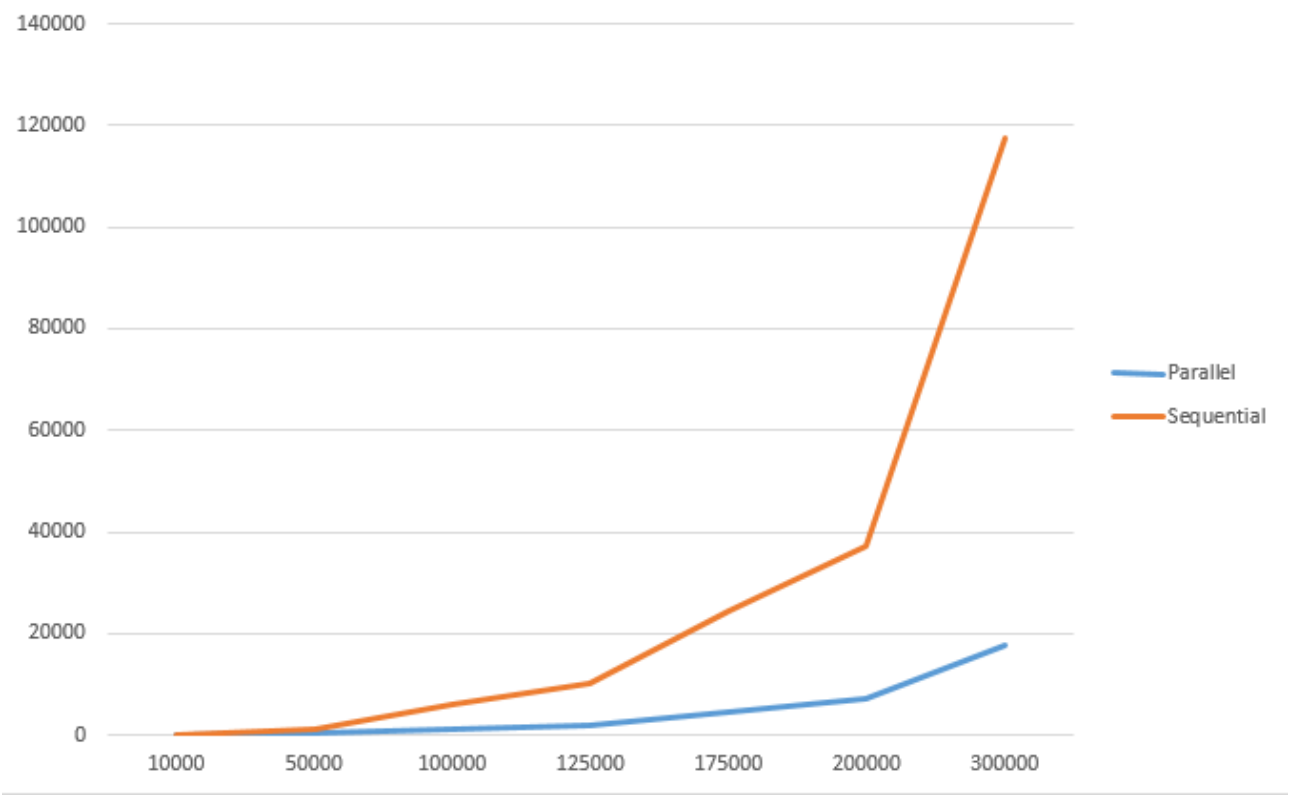


Рисунок 5.1. – Порівняння швидкодії обох алгоритмів

Аналогічно, з графіку бачимо, що паралельна реалізація сортування від 50000 елементів має набагато кращий час сортування та значно перевершує по цьому показнику послідовну реалізацію алгоритму.

**5.2 Вплив кількості процесів на ефективність**

Після перевірки коректності роботи і прискорення на рівному кількості ядер кількості процесів, можемо провести додаткові експерименти варіюючи

кількість процесів. Експерименти будуть проведені на 10 корзинах і кількістю процесів від 2 до 10 (кожному процесу буде виділено один бакет). Тестування проводиться на процесорі AMD Ryzen 7 4600U, 8 ядер, 8 потоків.

Результати будуть наведені у вигляді таблиці (таблиця 5.2.).

Таблиця 5.2. – Результати тестування для різної кількості процесів.

| Кількість елементів | Кількість процесів, час у мілісекундах | | | |
|---|---|---|---|---|
| | 2 | 4 | 7 | 10 |
| 50000 | 1042 | 722 | 565 | 403 |
| 100000 | 3243 | 2475 | 2056 | 1910 |
| 125000 | 6282 | 3604 | 2175 | 2934 |
| 175000 | 11889 | 9292 | 6208 | 4015 |
| 200000 | 15649 | 12784 | 8814 | 5633 |
| 300000 | 39763 | 27545 | 20937 | 14308 |

Результати експериментів показують значне зменшення часу виконання при збільшенні кількості процесів для кожної кількості елементів. Наприклад, для 50000 елементів час сортування зменшується з 1042 мс при 2 процесах до 403 мс при 10 процесах. Однак ефективність паралелізації зменшується на не надвеликих обсягах зі збільшенням кількості процесів через накладні витрати на координацію і недостатнє навантаження на процеси.

Таким чином, паралельне сортування значно покращує продуктивність, особливо при переході від меншої до середньої кількості процесів. Найбільше прискорення спостерігається при збільшенні кількості процесів з 2 до 4, після чого приріст продуктивності зменшується. Ефективність паралелізації залежить від розміру задачі та обчислювальних ресурсів, що підкреслює важливість оптимального вибору кількості процесів для досягнення найкращих результатів.

Для наглядності також побудуємо графік ефективнонсті(рисунок 5.2.).

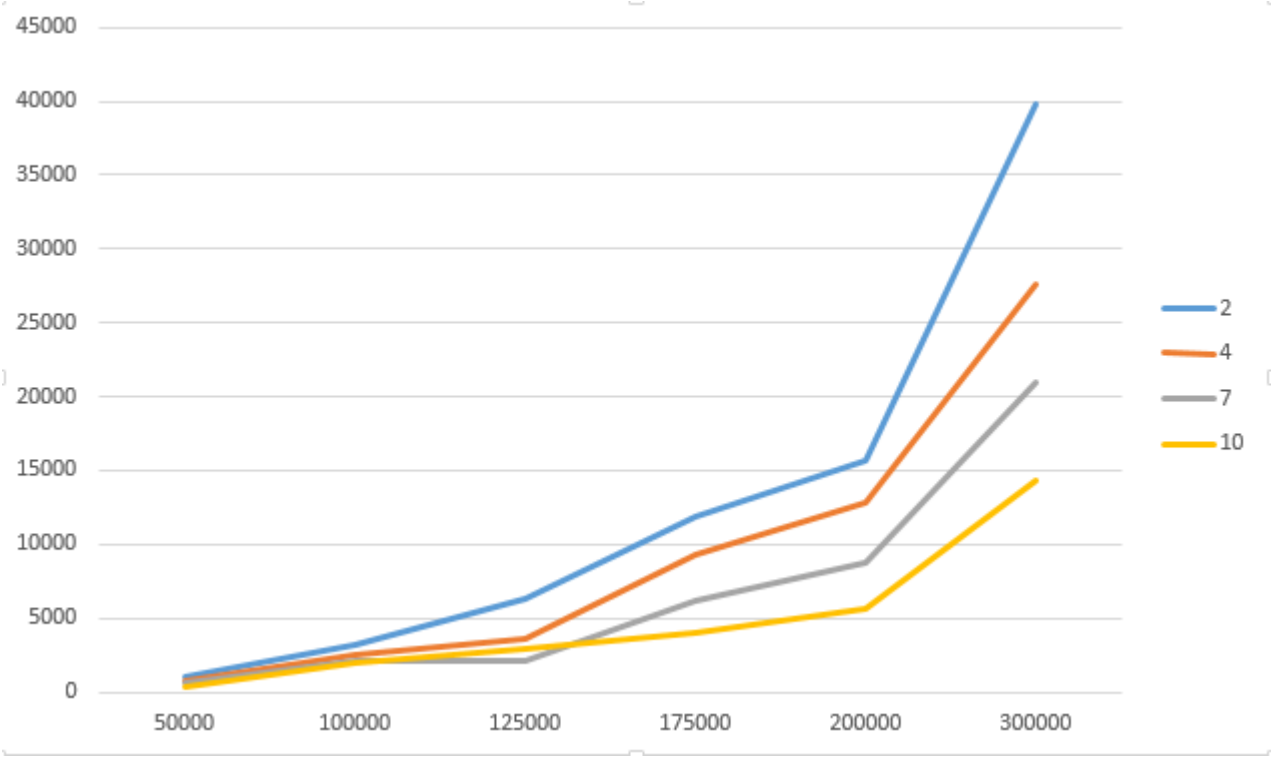


Рисунок 5.1. – Порівняння шкидкодії для різної кількості процесів

# ВИСНОВКИ

У цій роботі було розроблено та проаналізовано паралельну реалізацію алгоритму сортування методом bucket sort на платформі Node.js (JavaScript).

Перш за все було описано основний алгоритм і алгоритм для сортування корзин з псевдокодом і покроковим описом принципу роботи. Потім було проаналізовано усі найбільш відому способи паралелізації і обрано найефективніші технології для забезпечення найбільшої швидкодії. Ці дослідження дозволили розробити максимально ефективний алгоритм вже у програмному коді.

Реалізація алгоритму за допомогою Worker Threads дозволила ефективно розпаралелити сортування з використанням мультипроцесингу. Алгоритм пройшов тестування, і результати тестів дозволили провести аналіз його ефективності. За результатами тестів на різних розмірностях масиву можна побачити, що паралельний алгоритм працює значно швидше за послідовний.

Також було вивчено вплив кількості процесів на продуктивність алгоритму. Для великих масивів найкращі результати показало використання більшої кількості процесів, однак найкращий приріст ефективності показав перехід від малої до середньої кількості процесів(від кількість ядер/2 до кількості ядер процесора).

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bucket Sort Algorithm. Посилання: https://www.scaler.com/topics/data-structures/bucket-sort/
2. Bubble Sort. Посилання: https://en.wikipedia.org/wiki/Bubble_sort
3. VS Code. Посилання: https://code.visualstudio.com/
4. JetBrains. Посилання: https://www.jetbrains.com/
5. Promises. Посилання: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

# ДОДАТКИ

## Додаток А. Лістинг програмного коду

**index.js**

```
import bucketSort from "./algorithms/bucketSort.js";
import parallelBucketSort from "./algorithms/parallelBucketSort/parallelBucketSort.js";
import measureExecutionTime from "./utils/measureExecutionTime.js";
import generateRandomArray from "./utils/generateArray.js";
import isSorted from "./utils/isSorted.js";
import "dotenv/config"

async function warmUp() {
    const arraySize = 5000;
    const array = generateRandomArray(arraySize);
    for (let i = 0; i < 10; i++) {
        await bucketSort(array, 5);
        await parallelBucketSort(array, 5);
    }
}

async function main() {
    const arraySize = parseInt(process.env.ARRAY_SIZE, 10);
    const array = generateRandomArray(arraySize);

    await warmUp();

    const sortedSequential = await measureExecutionTime(bucketSort, array, 10);
    const sortedParallel = await measureExecutionTime(parallelBucketSort, array, 10);

    console.log(`Is sequential sorted: ${isSorted(sortedSequential)}`);
```

```
    console.log(`Is parallel sorted: ${isSorted(sortedParallel)}`);
    console.log(`Are equal: ${JSON.stringify(sortedSequential) === JSON.stringify(sortedParallel)}`)
}

main().catch(console.error);
```

**algorithms/bucketSort.js**

```
import bubbleSort from "./bubbleSort.js";

export default function bucketSort(array, num_buckets) {
    if (array.length === 0) {
        return array;
    }

    let minValue = array[0];
    let maxValue = array[0];

    for (let i = 1; i < array.length; i++) {
        if (array[i].accountBalance < minValue.accountBalance) {
            minValue = array[i];
        } else if (array[i].accountBalance > maxValue.accountBalance) {
            maxValue = array[i];
        }
    }

    const interval = (maxValue.accountBalance - minValue.accountBalance + 1) / num_buckets;
    const buckets = Array.from({ length: num_buckets }, () => []);
```

```
    for (let i = 0; i < array.length; i++) {
        const bucket_index = Math.floor((array[i].accountBalance - minValue.accountBalance) / interval);
        const index = bucket_index === num_buckets ? num_buckets - 1 : bucket_index;
        buckets[index].push(array[i]);
    }

    const sortedArray = [];

    buckets.forEach(bucket => {
        bubbleSort(bucket);
        sortedArray.push(...bucket);
    });

    return sortedArray;
}
```

**algorithms/bubbleSort.js**

```
export default function bubbleSort(arr) {
    const len = arr.length;
    for (let i = 0; i < len - 1; i++) {
        for (let j = 0; j < len - 1 - i; j++) {
            if (arr[j].accountBalance > arr[j + 1].accountBalance) {
                let temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    return arr;
```

```
}
```

**algorithms/parallelBucketSort/parallelBucketSort.js**

```
import { Worker } from "worker_threads";
import os from "os";

export default function parallelBucketSort(array, num_buckets) {
    return new Promise((resolve, reject) => {
        if (array.length === 0) {
            resolve(array);
            return;
        }

        let minValue = array[0];
        let maxValue = array[0];

        for (let i = 1; i < array.length; i++) {
            if (array[i].accountBalance < minValue.accountBalance) {
                minValue = array[i];
            } else if (array[i].accountBalance > maxValue.accountBalance) {
                maxValue = array[i];
            }
        }

        const interval = (maxValue.accountBalance - minValue.accountBalance + 1) /
num_buckets;
        const buckets = Array.from({ length: num_buckets }, () => []);

        for (let i = 0; i < array.length; i++) {
```

```
        const bucket_index = Math.floor((array[i].accountBalance - minValue.accountBalance) / interval);
        const index = bucket_index === num_buckets ? num_buckets - 1 : bucket_index;
        buckets[index].push(array[i]);
    }

    const cpuCount = os.cpus().length;
    const workers = Math.min(cpuCount, num_buckets);
    const promises = [];

    const bucketsPerWorker = Math.floor(num_buckets / workers);
    const extraBuckets = num_buckets % workers;

    let offset = 0;
    for (let i = 0; i < workers; i++) {

        const end = offset + bucketsPerWorker + (i < extraBuckets ? 1 : 0);
        const workerBuckets = buckets.slice(offset, end);
        offset = end;

        promises.push(
            new Promise((resolve, reject) => {

                const worker = new Worker(new URL("./worker.js", import.meta.url), {
                    workerData: workerBuckets,
                });

                worker.on("message", resolve);
                worker.on("error", reject);
```

```
            worker.on("exit", (code) => {
                if (code !== 0) {
                    reject(new Error(`Worker stopped with exit code ${code}`));
                }
            });
        })
    );
}

Promise.all(promises)
    .then((sortedBuckets) => {
        const sortedArray = [];
        sortedBuckets.forEach((sortedChunk) => {
            sortedChunk.forEach((bucket) => {
                sortedArray.push(...bucket);
            });
        });
        resolve(sortedArray);
    })
    .catch(reject);
});
}
```

**algorithms/parallelBucketSort/worker.js**

```
import { parentPort, workerData } from "worker_threads";
import bubbleSort from "../bubbleSort.js";

const sortedBuckets = workerData.map(bucket => {
    bubbleSort(bucket);
    return bucket;
});
```

```
parentPort.postMessage(sortedBuckets);
```

**utils/generateArray.js**

```
function getRandomNumber(min, max) {
   return Math.floor(Math.random() * (max - min + 1)) + min;
}

export default function generateRandomArray(size, minNumber = 1, maxNumber = 1000) {
   const randomArray = [];
   for (let i = 0; i < size; i++) {
      randomArray.push({accountBalance: getRandomNumber(minNumber, maxNumber)});
   }
   return randomArray;
}
```

**utils/isSorted.js**

```
export default function isSorted(arr) {
   return arr.every((v, i, a) => !i || a[i - 1].accountBalance <= v.accountBalance);
}
```

**utils/measureExecutionTime.js**

```
import { performance } from 'perf_hooks';

export default async function measureExecutionTime(func, ...args) {
   const runs = 3;
   let totalTime = 0;
   let result;
```

```
    for (let i = 0; i < runs; i++) {
        const start = performance.now();
        if (i === 0){
            result = await func(...args);
        }
        else {
            await func(...args);
        }
        const end = performance.now();
        totalTime += end - start;
    }

    const averageTime = totalTime / runs;
    console.log(`Function ${func.name} executed in average ${Math.ceil(averageTime)} milliseconds over ${runs} runs`);
    return result;
}
```