

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**

**АНОТАЦІЯ**

ЗМІСТ

[**ВСТУП** 5](#_Toc158713962)

[**1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ** 6](#_Toc158713963)

[**1.1 Назва підрозділу** 6](#_Toc158713964)

[**1.2 Назва підрозділу** 6](#_Toc158713965)

[**2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ** 7](#_Toc158713966)

[**3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС** 8](#_Toc158713967)

[**4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ** 9](#_Toc158713968)

[**5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ** 10](#_Toc158713969)

[**ВИСНОВКИ** 11](#_Toc158713970)

[**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ** 12](#_Toc158713971)

[**ДОДАТКИ** 13](#_Toc158713972)

[**Додаток А. Нзава додатку** 13](#_Toc158713973)

[**Додаток Б. Назва додатку** 14](#_Toc158713974)

[**Додаток В. Назва додатку** 15](#_Toc158713975)

# 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) - 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. 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(n2), 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 i 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 цїєї 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.

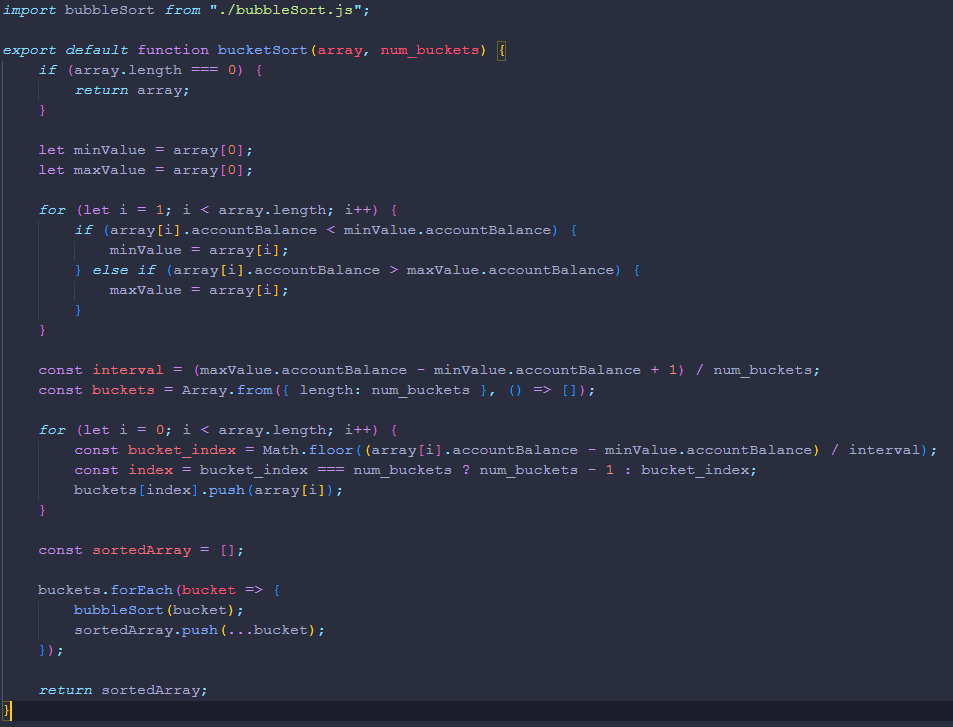


Рисунок 2.1. – код послідовного алгоритму сортування корзинами.

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

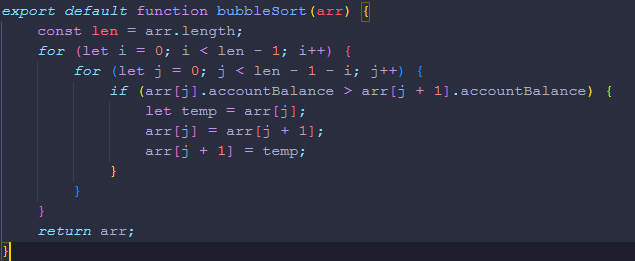


Рисунок 2.2. – код послідовного алгоритму сортування бульбашкою.

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.

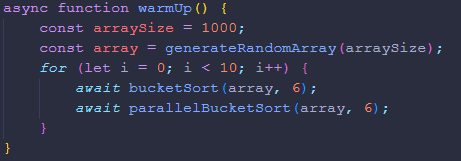


Рисунок 2.3. – код для прогрівання програми

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.

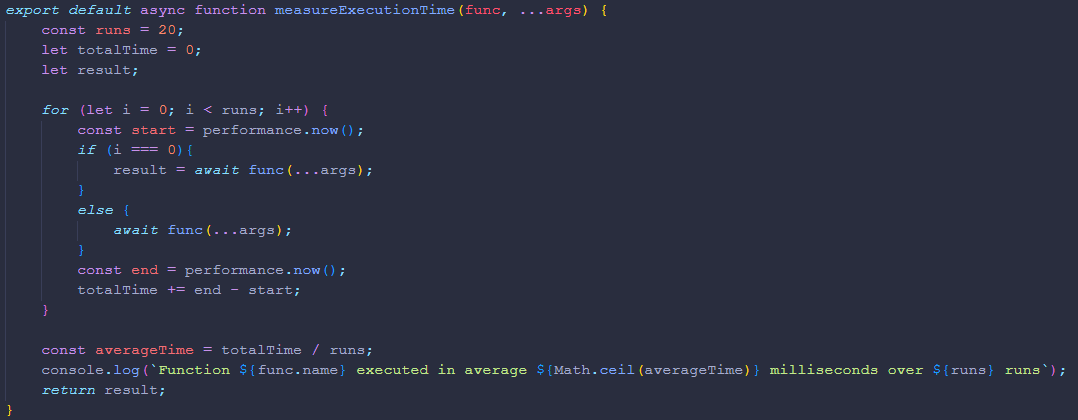


Рисунок 2.4. – функція заміру часу роботи алгоритму

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

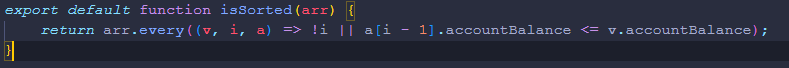


Рисунок 2.5. – функція перевірки впорядкованості масиву

Приклад використання у головному файлі наведений на рисунку 2.6.



Рисунок 2.6. – головний файл index.js

Результати пробного експерименту можна побачити на рисунку 2.7.



Рисунок 2.6. – вивід програми для 100000 елементів

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

|  |  |
| --- | --- |
| Кількість елементів | Час послідовного алгоритму, мілісекунд |
| 10000 |  |
| 50000 |  |
| 100000 |  |
| 200000 |  |
| 300000 |  |
| 500000 |  |
| 1000000 |  |

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

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

# 5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

…представлені у таблиці 5.1…..

Таблиця 5.1. – Назва таблиці

|  |  |  |
| --- | --- | --- |
| Кількість елементів | Час послідовного алгоритму, мікросекунд | Час паралельного алгоритму, мікросекунд |
| 1000 | 2477 | 475 |
| 3000 | 2338 | 837 |

На рисунку 5.1….

…..(рис.5.3).

Рисунок 5.1. – Назва рисунку

Рисунок 5.3. – Результат виконання програми у консолі

# ВИСНОВКИ

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. …...
2. …..
3. …..
4. ….
5. …..

# ДОДАТКИ

## Додаток А. Назва додатку

## Додаток Б. Назва додатку

## Додаток В. Назва додатку