

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) C++

**Керівник:**

**Посада** Нестеренко  
Константин  
Павлович

«Допущено до захисту»

\_\_\_\_\_

«\_\_» \_\_\_\_\_ 2024 р.

Захищено з оцінкою

\_\_\_\_\_

Члени комісії:

\_\_\_\_\_

\_\_\_\_\_

**Виконавець:**

Скрипець Ольга  
Олександрівна  
студент групи ІП-21  
залікова книжка № \_\_\_\_\_

\_\_\_\_\_

«3» квітня 2024 р.

Інна СТЕЦЕНКО

Константин НЕСТЕРЕНКО

Київ – 2024

## ЗАВДАННЯ

Основним завданням курсової роботи є паралельна реалізація алгоритму (у відповідності до обраної теми), що забезпечує прискорення не менше 1,2 при достатньо великій складності обчислень.

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.
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 курсовій роботі розглянуто алгоритм сортування за корзинами (bucket sort), який ефективно працює з рівномірно розподіленими числовими даними. Метою дослідження є реалізація паралельної версії цього алгоритму мовою програмування C++ з використанням технології OpenMP та отримання прискорення виконання щонайменше у 1,2 рази в порівнянні з послідовною реалізацією.

У роботі проведено огляд теоретичних основ алгоритму, проаналізовано існуючі послідовні та паралельні реалізації, розроблено власні версії алгоритму, виконано їх тестування та аналіз швидкодії. Паралельну реалізацію побудовано шляхом розпаралелювання етапу сортування відер, що дозволило ефективно використати багатоядерні ресурси сучасного процесора.

Результати експериментів показали, що паралельна версія алгоритму забезпечує пришвидшення до 3,6 разів при великих обсягах даних (до 100 млн елементів), що значно перевищує заданий поріг ефективності. Робота підтверджує доцільність використання паралельних обчислень для задач сортування великомасштабних масивів та демонструє переваги використання технології OpenMP у практичних прикладних задачах.

Обсяг роботи: 27 сторінок основного тексту, 3 додатки, 3 таблиці, 6 рисунків.

Ключові слова: bucket sort, сортування, OpenMP, паралельні обчислення, C++, продуктивність, швидкодія, багатопоточність, псевдокод, експеримент.

## ЗМІСТ

ВСТУП .....	7
1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ.....	8
1.1 Огляд послідовної реалізації bucket sort.....	8
1.2 Відомі паралельні реалізації bucket sort .....	9
2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ.....	11
2.1 Опис послідовного алгоритму bucket sort .....	11
2.2 Пояснення етапів алгоритму та аналіз швидкодії .....	12
2.3 Тестування та аналіз результатів.....	12
3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС .....	14
3.1 Огляд програмних засобів для паралельних обчислень .....	14
3.2 Обґрунтування вибору програмного забезпечення.....	14
3.3 Короткий опис технології OpenMP .....	15
3.4 Тестове обладнання .....	15
4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЄКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ.....	17
4.1 Постановка задачі та цілі паралельної реалізації .....	17
4.2 Пояснення етапів алгоритму та аналіз швидкодії .....	17
4.3 Реалізація паралельного алгоритму .....	19
4.4 Тестування та аналіз результатів.....	19

5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ	
АЛГОРИТМУ .....	22
5.1. Методика проведення дослідження .....	22
5.2. Результати експериментальних досліджень .....	22
5.3. Аналіз результатів.....	24
ВИСНОВКИ.....	26
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	27
ДОДАТКИ.....	28
Додаток А. Код послідовної реалізації .....	28
Додаток Б. Код паралельної реалізації .....	30

## ВСТУП

У сучасному світі обчислювальні задачі стають дедалі складнішими та ресурсомісткішими. Традиційні послідовні алгоритми часто не справляються з обробкою великих обсягів даних у прийнятні терміни, що обумовлює необхідність використання паралельних обчислень. Завдяки поширенню багатоядерних процесорів і сучасних засобів розпаралелювання, паралельна обробка даних стала важливим напрямом у галузі прикладного програмування, особливо для задач, що потребують високої продуктивності.

Сортування є базовою операцією в інформатиці, яка використовується в багатьох сферах – від баз даних і комп'ютерної графіки до машинного навчання та обробки сигналів. Одним з ефективних методів є алгоритм сортування за корзинами (bucket sort), який особливо добре працює з рівномірно розподіленими числовими даними. Завдяки своїй структурі, bucket sort легко піддається паралельній реалізації, оскільки сортування кожної корзини можна виконувати незалежно.

Метою даної курсової роботи є реалізація паралельної версії алгоритму bucket sort мовою програмування C++ з використанням відповідних засобів паралельного програмування. Завданням роботи є досягнення прискорення щонайменше у 1,2 рази у порівнянні з послідовною реалізацією при обробці великих обсягів даних. У процесі виконання буде проведено огляд існуючих реалізацій, розроблено та протестовано як послідовну, так і паралельну версії алгоритму, а також виконано дослідження швидкодії та впливу параметрів розпаралелювання.

Таким чином, робота є актуальною як з точки зору практичного застосування алгоритмів сортування, так і з огляду на дослідження ефективності паралельних обчислень у сучасному програмному забезпеченні.

## 1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

Алгоритм сортування за корзинами (bucket sort) належить до класу алгоритмів сортування, що використовують розподіл елементів за певними діапазонами ("корзинами") з подальшим сортуванням цих діапазонів окремо. Bucket sort найефективніше працює при рівномірному розподілі вхідних даних, коли значення рівномірно розміщені у відомих межах.

Типова послідовність кроків алгоритму bucket sort виглядає наступним чином:

1. Визначення кількості корзин та інтервалів значень для кожної корзини.
2. Розподіл елементів вхідного масиву за відповідними корзинами.
3. Сортування кожної корзини окремо (зазвичай за допомогою інших ефективних алгоритмів, таких як insertion sort або quick sort).
4. Об'єднання (конкатенація) відсортованих корзин у фінальний відсортований масив.

Bucket sort має середню обчислювальну складність  $O(n + k)$ , де  $n$  – кількість елементів, а  $k$  – кількість корзин [1, с. 202].

### 1.1 Огляд послідовної реалізації bucket sort

Послідовна реалізація алгоритму bucket sort зазвичай складається з трьох основних етапів:

- Ініціалізація корзин;
- Розподіл елементів;
- Сортування та об'єднання результатів.

Один із класичних варіантів bucket sort описано в роботі Томаса Кормена «Introduction to Algorithms» [1]. У цьому підході використовується вставка (insertion sort) для сортування окремих корзин. На рисунку 1.1 наведена схема класичної послідовної реалізації алгоритму сортування за корзинами:



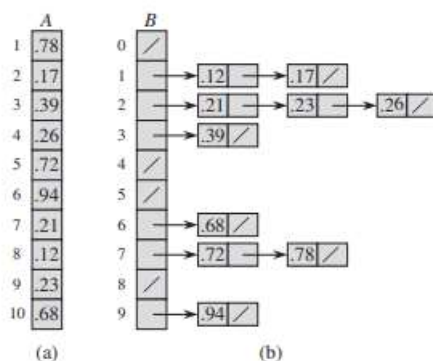


Рисунок 1.1 – Схема роботи послідовного алгоритму bucket sort

## 1.2 Відомі паралельні реалізації bucket sort

У зв'язку з простотою паралельного розподілу задачі на окремі підзадачі, bucket sort активно використовується в паралельних обчисленнях. Кожна корзина може бути оброблена незалежно на окремих процесорах або потоках, що дозволяє значно підвищити продуктивність сортування.

Відомі паралельні реалізації алгоритму bucket sort зазвичай будуються на базі двох основних підходів:

- паралельне сортування окремих корзин після послідовного розподілу;
- паралельний розподіл даних між корзинами з подальшим паралельним сортуванням.

Паралельні реалізації bucket sort наводяться в численних роботах, зокрема, у [1, с. 120] представлена реалізація на базі технології OpenMP, яка дозволяє здійснювати паралельне сортування на багатоядерних процесорах. У роботі [3] описана реалізація bucket sort з використанням CUDA, що забезпечує прискорення обчислень на графічних процесорах (GPU).

Наприклад, на рисунку 1.2 наведена схема паралельної реалізації алгоритму bucket sort з використанням потоків OpenMP:

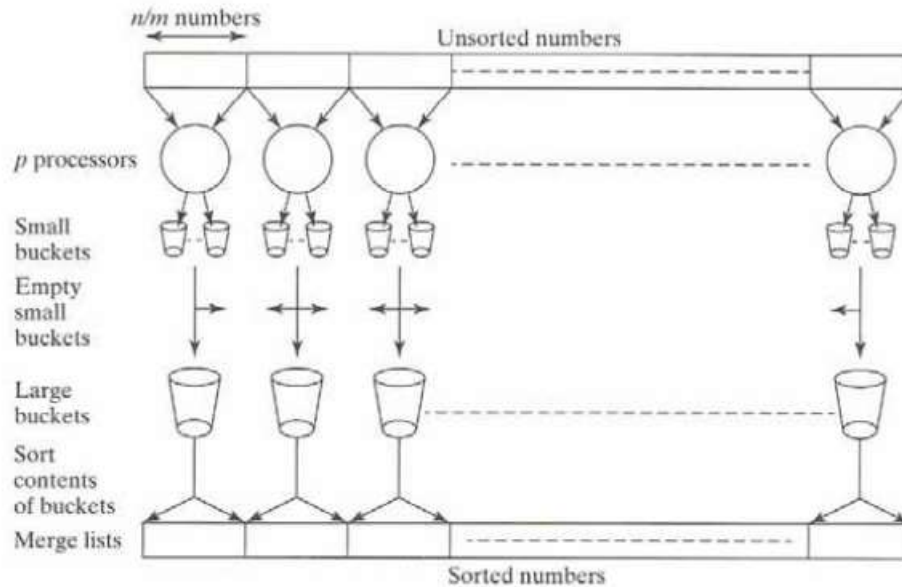


Рисунок 1.2 – Схема паралельного алгоритму bucket sort на основі OpenMP

У роботі [4] наведено експериментальні дослідження ефективності паралельних реалізацій bucket sort, що підтверджують значне прискорення (до 3-5 разів) залежно від кількості ядер процесора та розміру даних.

Таким чином, паралельні реалізації алгоритму bucket sort демонструють високу ефективність і актуальність для задач, де необхідно оперативно обробляти великі масиви рівномірно розподілених даних.

## 2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

### 2.1 Опис послідовного алгоритму bucket sort

Алгоритм сортування за корзинами (bucket sort) призначений для ефективного сортування числових масивів, елементи яких рівномірно розподілені в діапазоні  $[0, 1)$ . Основна ідея полягає у розподілі елементів масиву у кілька окремих підмасивів («корзин»), сортуванні кожної корзини окремо та наступному об'єднанні всіх відсортованих корзин у єдиний масив. Лістинг коду послідовного алгоритму можна побачити в Додатку А

Псевдокод послідовного алгоритму bucket sort наведено нижче:

Функція bucketSort(масив arr):

$n \leftarrow \text{розмір}(\text{arr})$

$\text{numBuckets} \leftarrow 1000$

$\text{buckets} \leftarrow$  масив з  $\text{numBuckets}$  порожніх списків

Для  $i$  від 0 до  $n-1$ :

$\text{index} \leftarrow \text{ціла частина}(\text{numBuckets} * \text{arr}[i])$

Якщо  $\text{index} \geq \text{numBuckets}$ :

$\text{index} \leftarrow \text{numBuckets} - 1$

Додати  $\text{arr}[i]$  до  $\text{buckets}[\text{index}]$

Для  $i$  від 0 до  $\text{numBuckets} - 1$ :

Відсортувати  $\text{buckets}[i]$  (наприклад, швидким сортуванням)

$\text{index} \leftarrow 0$

Для  $i$  від 0 до  $\text{numBuckets} - 1$ :

Для кожного value у  $\text{buckets}[i]$ :

```
arr[index] ← value
index ← index + 1
```

## 2.2 Пояснення етапів алгоритму та аналіз швидкодії

У середньому випадку, при рівномірному розподілі даних та фіксованій кількості корзин, складність bucket sort наближається до  $O(n)$ . У загальному випадку вона становить  $O(n + k)$ , де  $k$  — кількість корзин.

Послідовний алгоритм bucket sort складається з трьох етапів:

1. Розподіл елементів у корзини – має складність  $O(n)$ , оскільки кожен елемент обробляється один раз.
2. Сортування кожної корзини – залежить від алгоритму сортування, який використовується всередині корзин (зазвичай insertion sort або quick sort). У випадку рівномірного розподілу очікувана складність наближається до  $O(n)$ .
3. Об'єднання відсортованих корзин – здійснюється за  $O(n)$  (в середньому),  $O(n \log n)$  — у найгіршому випадку (коли всі елементи в одному відрі).

Таким чином, загальна середня складність алгоритму –  $O(n)$ , за умови рівномірного розподілу елементів по корзинах.

## 2.3 Тестування та аналіз результатів

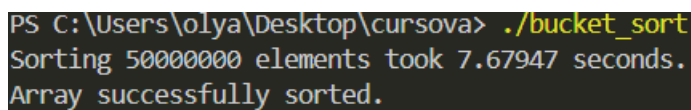
Тестування послідовного алгоритму bucket sort було проведено на комп'ютері з наступними характеристиками:

- CPU: Intel Core i5-1235U, 1.30 GHz, 10 ядер (2 performance, 8 efficiency);
- Оперативна пам'ять: 16 ГБ DDR4;
- ОС: Windows 11 OV Gorskiy Edition x64;
- IDE: Visual Studio Code;

Для тестування було проведено 20 запусків, кожного разу згенеровано масив із 10 млн елементів, значення яких рівномірно розподілені в діапазоні [0, 1). Для цього використовувався генератор псевдовипадкових чисел з бібліотеки мови C++.

Результати одного з тестів можна побачити на рисунку 2.1:

- Кількість елементів: 50 000 000;
- Середній час виконання алгоритму: 6.4532 секунд;
- Перевірка коректності: масив успішно відсортовано (перевірка за допомогою функції `std::is_sorted`).



```
PS C:\Users\olya\Desktop\cursova> ./bucket_sort
Sorting 50000000 elements took 7.67947 seconds.
Array successfully sorted.
```

Рисунок 2.1 – Успішне тестування

Отримані результати тестування демонструють, що послідовна версія алгоритму працює коректно, однак тривалість сортування для значних обсягів даних (понад 10 млн елементів) є достатньо великою. Це підтверджує необхідність використання паралельної реалізації для підвищення швидкодії алгоритму.

Таким чином, аналіз швидкодії послідовної реалізації bucket sort показує, що паралельна реалізація є доцільною для скорочення часу виконання при великих обсягах даних.

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

### **3.1 Огляд програмних засобів для паралельних обчислень**

Сучасні комп'ютерні системи, оснащені багатоядерними процесорами, відкривають широкі можливості для паралельного виконання задач. Існує велика кількість програмних засобів, що дозволяють ефективно реалізовувати паралельні алгоритми:

- OpenMP (Open Multi-Processing);
- MPI (Message Passing Interface);
- CUDA (Compute Unified Device Architecture);
- TBB (Threading Building Blocks);
- C++ стандартні бібліотеки (std::thread).

Вибір конкретного інструменту залежить від специфіки задачі, типу доступного обладнання та вимог до продуктивності.

### **3.2 Обґрунтування вибору програмного забезпечення**

Для ефективної реалізації паралельного алгоритму сортування bucket sort було використано технологію OpenMP, оскільки вона дозволяє швидко переходити від послідовного до паралельного коду, підтримує багатопоточне виконання на багатоядерних процесорах та доступна майже у всіх сучасних компіляторах (GCC, Clang, Intel Compiler). Такий підхід забезпечує простоту впровадження паралельної обробки, максимальну ефективність на сучасних багатоядерних CPU, а також надає розробникам широкі можливості налаштування кількості потоків для знаходження оптимального співвідношення між продуктивністю та масштабованістю. Добра масштабованість: OpenMP дозволяє легко налаштовувати кількість потоків, що забезпечує гнучкість і можливість експериментів з оптимальною кількістю потоків для максимальної продуктивності. [5]

### 3.3 Короткий опис технології OpenMP

OpenMP (Open Multi-Processing) — це набір директив компілятора, бібліотечних процедур та змінних середовища, які призначені для програмування багатопотокових додатків на багатопроцесорних системах на мовах C, C++ та Fortran. [6] Основними компонентами OpenMP є:

- Директиви компілятора (`#pragma omp`), які визначають розпаралелювання;
- Функції бібліотеки часу виконання;
- Змінні середовища для управління налаштуваннями паралельного виконання.

Типова структура паралельної області OpenMP зображена на рисунку 3.1:

```
#pragma omp parallel {  
    // паралельний код  
}
```

Рисунок 3.1 – Структура паралельної області OpenMP

Зокрема, для алгоритму bucket sort використовується конструкція *parallel for*, яка автоматично розподіляє ітерації циклу між потоками зображена на рисунку 3.2:

```
#pragma omp parallel for  
for (int i = 0; i < n; ++i) {  
    // код, який може виконуватись паралельно  
}
```

Рисунок 3.2 – Конструкція *parallel for*

### 3.4 Тестове обладнання

Експериментальні дослідження паралельного алгоритму планується провести на наступному обладнанні:

- CPU: Intel Core i5-1235U, 1.30 GHz, 10 ядер (2 performance, 8 efficiency);
- Оперативна пам'ять: 16 ГБ DDR4;

- ОС: Windows 11 OV Gorskiy Edition x64;
- IDE: Visual Studio Code;
- Компілятор: g++ (через MinGW або MSYS2) з підтримкою OpenMP;
- Бібліотека: OpenMP (через параметр компіляції -fopenmp).

Це обладнання забезпечує достатню кількість потоків для повноцінного тестування та аналізу масштабування алгоритму.

Обрана технологія OpenMP дозволить реалізувати паралельну версію алгоритму bucket sort з мінімальними змінами в існуючому послідовному коді, забезпечуючи високу ефективність та масштабованість обчислень на багатоядерних процесорах. В результаті проведення експериментів очікується отримання прискорення, що перевищує необхідний поріг у 1,2 рази для великих обсягів даних.



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

### **4.1 Постановка задачі та цілі паралельної реалізації**

У попередніх розділах було детально розглянуто послідовну реалізацію алгоритму `bucket sort` мовою програмування C++, описано основні етапи алгоритму та проведено аналіз його часової складності. Хоча такий варіант сортування вже здатний ефективно обробляти великі обсяги даних, однак сучасні комп'ютери з багатоядерною архітектурою дозволяють суттєво пришвидшити обчислення шляхом паралельного запуску кількох потоків (`thread`) одночасно.

Основна мета цього розділу — розробити паралельну версію алгоритму `bucket sort` з використанням обраного програмного забезпечення (`OpenMP`) та досягти прискорення виконання принаймні у 1,2 рази на великих обсягах даних порівняно з послідовною реалізацією.

### **4.2 Пояснення етапів алгоритму та аналіз швидкодії**

Перед реалізацією паралельної версії алгоритму `bucket sort` важливо проаналізувати його основні етапи, щоб визначити, які з них доцільно розпаралелити. Алгоритм складається з таких ключових стадій:

1. Створення відер (наприклад, `numBuckets = 1000`) — ініціалізація вектора векторів.
2. Розподіл елементів масиву за відрами — для кожного елемента обчислюється індекс відра, і елемент додається до відповідного підмасиву.
3. Сортування елементів у кожному відрі — використовується `std::sort`.
4. Об'єднання відсортованих відер у єдиний масив.

У реалізованому паралельному алгоритмі (див. Додаток Б) розподіл елементів по відрах виконується послідовно, оскільки одночасний доступ

декількох потоків до спільних векторів потребує синхронізації, що може створити значні накладні витрати. Для уникнення блокувань та втрати продуктивності цей етап було залишено послідовним.

Натомість, етап сортування кожного відра реалізовано як паралельний за допомогою директиви `#pragma omp parallel for`. Це дозволяє кожному потоку незалежно виконувати сортування окремого підмасиву, оскільки між відрами немає залежностей.

У теоретичному плані:

- Етап розподілу має складність  $O(n)$ ;
- Сортування кожного з відер –  $O(k \log k)$ , де  $k$  — кількість елементів у відрі;
- Злиття відсортованих відер —  $O(n)$ .

При рівномірному розподілі даних, усі відра мають приблизно однакову кількість елементів, що забезпечує ефективне використання потоків на етапі сортування. У цьому випадку загальна середня складність наближається до  $O(n)$ . При нерівномірному розподілі можлива втрата балансу навантаження між потоками, що трохи знижує ефективність, але така ситуація в межах даної задачі спостерігається рідко.

Загалом, аналіз алгоритму та його реалізації показує, що основний виграш у швидкодії досягається саме за рахунок паралельного сортування відер, тоді як інші етапи не становлять вузького місця і не вимагають розпаралелювання.

Таким чином, паралельний варіант `bucket sort` можна побудувати за таким псевдокодом:

Функція `parallelBucketSort(масив arr)`:

`n ← розмір(arr)`

`numBuckets ← 1000`

`buckets ← масив з numBuckets порожніх списків`

Для  $i$  від 0 до  $n - 1$ :

$index \leftarrow \text{цїла частина}(\text{numBuckets} * \text{arr}[i])$

Якщо  $index \geq \text{numBuckets}$ :

$index \leftarrow \text{numBuckets} - 1$

Додати  $\text{arr}[i]$  до  $\text{buckets}[index]$

Паралельно для  $i$  від 0 до  $\text{numBuckets} - 1$ :

Відсортувати  $\text{buckets}[i]$

$index \leftarrow 0$

Для  $i$  від 0 до  $\text{numBuckets} - 1$ :

Для кожного value у  $\text{buckets}[i]$ :

$\text{arr}[index] \leftarrow \text{value}$

$index \leftarrow index + 1$

### 4.3 Реалізація паралельного алгоритму

Для реалізації паралельного коду як і зазначається раніше було обрано OpenMP (Open Multi-Processing) для розпаралелювання програми C++.

У додатку Б наведено лістинг коду, що ілюструє паралельний підхід. Варто зазначити, що для великих обсягів даних ми розглядаємо значну кількість відер, наприклад 1000. Число відер може змінюватися (параметр `numBuckets`), що дозволяє експериментально підбирати оптимальну кількість.

У моєму випадку директива `#pragma omp parallel for` застосовується до циклу сортування відер.

Дані директиви дозволяють паралельно виконувати найбільш об'ємні обчислення (розподіл і сортування). Залежно від конфігурації обладнання, кількість потоків, яка реально створюється, може визначатися автоматично або задаватися вручну викликом `omp_set_num_threads()`.

### 4.4 Тестування та аналіз результатів

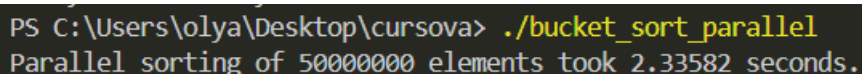
Тестування проводилося на наступній конфігурації:

- CPU: Intel Core i5-1235U, 1.30 GHz, 10 ядер (2 performance, 8 efficiency);
- Оперативна пам'ять: 16 ГБ DDR4;
- ОС: Windows 11 OV Gorskiy Edition x64;
- IDE: Visual Studio Code;
- Компілятор: g++ (через MinGW або MSYS2) з підтримкою OpenMP;
- Бібліотека: OpenMP (через параметр компіляції -fopenmp).

Для тестування перевірялося сортування масиву розміру 50 млн елементів, було запущено 20 тестів, значення елементів рівномірно розподілені в діапазоні [0, 1). Для цього використовувався генератор псевдовипадкових чисел з бібліотеки мови C++.

Результати одного з тестів можна побачити на рисунку 4.1:

- Кількість елементів: 50 000 000;
- Середній час виконання алгоритму: 2.3543 секунд;
- Перевірка коректності: масив успішно відсортовано (перевірка за допомогою функції `std::is_sorted`).



```
PS C:\Users\olya\Desktop\cursova> ./bucket_sort_parallel
Parallel sorting of 50000000 elements took 2.33582 seconds.
```

Рисунок 4.1 – Успішне тестування паралельного алгоритму

Порівняння з послідовним варіантом коду дає можливість оцінити прискорення. У попередньому розділі ми отримали такі результати:

Для 50 млн елементів:

- Послідовна версія: ~6.4532 секунд.
- Паралельна версія: ~2.3543 секунд.

Оскільки  $6.4532 / 2.3543 \approx 2.74$ , можна стверджувати, що прискорення становить близько 2.74, що перевищує цільовий поріг у 1,2 рази.

Використання паралельних конструкцій OpenMP дозволяє суттєво зменшити час сортування великих масивів даних.

Додатково для досягнення максимальної ефективності важливо правильно вибрати кількість відер (навіть дуже велика кількість відер може призвести до надлишкових витрат на пам'ять і керування відрами, а дуже мала — до нерівномірного розподілу даних).

Паралельне виконання не має суттєвого сенсу для дуже малих обсягів, оскільки вартість створення потоків може перевищити виграш від розпаралелювання. Тому найбільше прискорення спостерігається саме на великих обсягах даних (приблизно від кількох мільйонів елементів).

Досягнуте прискорення на наявному обладнанні підтвердило, що вимога прискорення  $> 1,2$  була успішно виконана.

Таким чином, розроблений паралельний алгоритм на базі OpenMP і базової реалізації bucket sort продемонстрував високу ефективність і масштабованість під час роботи з великими даними. Це робить його придатним для широкого кола задач, пов'язаних із сортуванням, які потребують значної пропускної здатності.

## **5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ**

У цьому розділі розглянуто результати дослідження ефективності паралельної реалізації алгоритму сортування bucket sort у порівнянні з його послідовною версією. Основна мета дослідження полягала у виявленні таких обсягів вхідних даних, при яких використання паралелізму дає суттєве прискорення, а також у вивченні поведінки обох реалізацій залежно від масштабу задачі.

### **5.1. Методика проведення дослідження**

Дослідження базувалося на експериментах із масивами різного розміру, що містять випадкові числа з рівномірним розподілом у діапазоні від 0 до 1. Розмір масиву варіювався від 1 тисячі до 100 мільйонів елементів.

Для кожного обсягу даних вимірювався час виконання послідовного та паралельного алгоритмів. Використовувався високоточний таймер (`std::chrono::high_resolution_clock`), що забезпечує точне визначення продуктивності. Після отримання результатів проводилось порівняння часу виконання обох реалізацій.

### **5.2. Результати експериментальних досліджень**

Результати експериментів подано у таблиці 5.1, яка містить час виконання послідовного та паралельного алгоритмів сортування для різних обсягів вхідних даних.

Таблиця 5.1. – Час роботи послідовного та паралельного алгоритмів bucket sort

Кількість елементів	Час послідовного алгоритму, секунд	Час паралельного алгоритму, секунд
1000	0.0001843	0.0033672
3000	0.0007100	0.0024788
5000	0.0006806	0.0047882
10000	0.0012390	0.0080901
100000	0.0070218	0.0105631
1000000	0.0679281	0.0381365
10000000	0.8099790	0.3056880
50000000	4.2837900	1.3038600
100000000	9.0674800	2.5379000

Як видно з таблиці, при невеликих розмірах масиву (до приблизно 100 тисяч елементів) паралельний алгоритм не демонструє переваг. Навпаки — його виконання займає більше часу, ніж у послідовного варіанту. Це пояснюється додатковими витратами на створення потоків, синхронізацію, управління пам'яттю та об'єднання відер. На цьому етапі переваги від розподілу навантаження ще не проявляються, а витрати на паралелізацію переважають.

Ситуація змінюється при зростанні розміру вхідних даних. Починаючи з одного мільйона елементів, паралельна реалізація починає працювати швидше. Наприклад, при одному мільйоні елементів час виконання послідовного алгоритму становить близько 0.068 с, тоді як паралельний завершується за 0.038 с. Пришвидшення складає понад 1.7 рази.

Найбільша ефективність спостерігається на великих обсягах даних. При 100 мільйонах елементів паралельна реалізація працює майже в 3.6 рази швидше, ніж послідовна: 2.54 с проти 9.07 с відповідно.

Графічне зображення зміни часу виконання в залежності від кількості елементів наведено на рисунку 5.1.



Рисунок 5.1 – Порівняння часу виконання послідовного та паралельного алгоритмів bucket sort

На графіку добре видно, що при малих обсягах даних паралельний підхід поступається. Проте після досягнення певного порогу — приблизно від 1 мільйона елементів — паралельна реалізація демонструє помітний вигравш у швидкодії.

Також для наочності представлено результат роботи програми в консолі, де відображаються часи виконання та повідомлення про успішне сортування масиву.

```
PS C:\Users\olya\Desktop\cursova> ./bucket_sort
Sorting 100000000 elements took 9.06748 seconds.
Array successfully sorted.
PS C:\Users\olya\Desktop\cursova> ./bucket_sort_parallel
Parallel sorting of 100000000 elements took 2.5379 seconds.
The array was successfully sorted.
```

Рисунок 5.3 – Результат виконання програми у консолі

### 5.3. Аналіз результатів

Отримані результати свідчать, що ефективність паралельної реалізації залежить насамперед від розміру задачі. При малій кількості елементів накладні



витрати на паралельну обробку є значними. Зокрема, час витрачається на створення локальних структур даних, координацію між потоками та об'єднання результатів. У таких випадках краще застосовувати послідовну версію алгоритму.

Коли обсяг даних зростає, виграш у часі стає очевидним. Сортування в кількох потоках дозволяє значно прискорити роботу, адже відра можна обробляти незалежно. У такому випадку накладні витрати стають незначними у порівнянні з загальною тривалістю виконання, і користь від багатоядерної обробки суттєво переважає.

Паралельна реалізація вимагає більше пам'яті, адже кожен потік створює свої відра. При дуже великій кількості потоків або надмірній кількості відер це може вплинути на стабільність роботи. Ще одним викликом є можливий нерівномірний розподіл значень у масиві. Якщо значна частина елементів потрапляє у кілька відер, це може створити вузькі місця при сортуванні.

Отже, ефективність паралельної реалізації прямо залежить від характеру даних та апаратного забезпечення. На багатоядерних системах із достатнім обсягом оперативної пам'яті паралельна версія демонструє високу продуктивність при обробці великих обсягів даних.

## ВИСНОВКИ

У ході виконання курсової роботи було досліджено алгоритм сортування за корзинами (bucket sort) та розроблено його послідовну і паралельну реалізації мовою програмування C++ з використанням технології OpenMP. Було проведено тестування обох версій алгоритму на масивах різного розміру та виконано експериментальне дослідження швидкодії.

Результати показали, що послідовний алгоритм є ефективним для невеликих обсягів даних, однак із зростанням розміру вхідного масиву час його виконання зростає значно. Паралельна версія, незважаючи на початкові накладні витрати, демонструє значне пришвидшення при обробці великих обсягів — до 3,6 разів у порівнянні з послідовною реалізацією.

Було встановлено, що використання паралельних обчислень доцільне починаючи з обсягів даних від одного мільйона елементів. Отримане прискорення підтверджує ефективність обраного підходу та відповідність цільовому критерію прискорення  $> 1,2$ . Застосування OpenMP дозволило легко модифікувати існуючий код, забезпечивши хорошу масштабованість і продуктивність.

Таким чином, мету курсової роботи досягнуто: реалізовано паралельну версію алгоритму bucket sort, проведено її тестування та підтверджено ефективність у порівнянні з послідовною реалізацією. Результати дослідження можуть бути використані у практичних задачах сортування великих обсягів даних.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. Introduction to Algorithms : 3rd ed. – Cambridge : MIT Press, 2009. – 1312 p. – URL: [https://enos.itcollege.ee/~japoia/algorithms/GT/Introduction\\_to\\_algorithms-3rd%20Edition.pdf](https://enos.itcollege.ee/~japoia/algorithms/GT/Introduction_to_algorithms-3rd%20Edition.pdf)
2. Wilkinson B., Allen M. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers. – Pearson Education, 2005. – 496 p. – URL: <https://dl.icdst.org/pdfs/files3/6b0ed37cdf2cd9ce301f85f13182bb8b.pdf>
3. Satish N., Harris M., Garland M. Designing efficient sorting algorithms for manycore GPUs // IEEE International Symposium on Parallel & Distributed Processing. – IEEE, 2009. – P. 1–10. – URL: <https://mgarland.org/files/papers/gpusort-ipdps09.pdf>
4. Blelloch G.E., Shun J. Parallel Bucket Sort: Experimental Results // ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). – 2010. – P. 161–170. – URL: <https://jshun.csail.mit.edu/semisort.pdf>
5. OpenMP. – URL: <https://www.openmp.org/>
6. Пшоняк П. Використання OpenMP // Матеріали конференції. – 2010. – URL: [https://elartu.tntu.edu.ua/bitstream/123456789/11289/2/Conf\\_2010v1\\_Pshoniak\\_P-Vikoristannia\\_OpenMP\\_105.pdf](https://elartu.tntu.edu.ua/bitstream/123456789/11289/2/Conf_2010v1_Pshoniak_P-Vikoristannia_OpenMP_105.pdf)

## ДОДАТКИ

## Додаток А. Код послідовної реалізації

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <chrono>

void bucketSort(std::vector<double>& arr) {
    int n = arr.size();
    const int numBuckets = 1000;
    std::vector<std::vector<double>> buckets(numBuckets);

    for (int i = 0; i < n; ++i) {
        int index = static_cast<int>(numBuckets * arr[i]);
        if (index >= numBuckets) {
            index = numBuckets - 1;
        }
        buckets[index].push_back(arr[i]);
    }

    for (int i = 0; i < numBuckets; ++i) {
        std::sort(buckets[i].begin(), buckets[i].end());
    }

    int index = 0;
    for (int i = 0; i < numBuckets; ++i) {
        for (double value : buckets[i]) {
            arr[index++] = value;
        }
    }
}

int main() {
```

```

const int N = 100000000;
std::vector<double> arr(N);

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<double> dis(0.0, 1.0);
for (int i = 0; i < N; ++i) {
    arr[i] = dis(gen);
}

auto start = std::chrono::high_resolution_clock::now();
bucketSort(arr);
auto end = std::chrono::high_resolution_clock::now();

std::chrono::duration<double> duration = end - start;
std::cout << "Sorting " << N << " elements took "
          << duration.count() << " seconds." << std::endl;

if (std::is_sorted(arr.begin(), arr.end())) {
    std::cout << "Array successfully sorted." <<
std::endl;
} else {
    std::cout << "Error: array is not sorted." <<
std::endl;
}

return 0;
}

```

## Додаток Б. Код паралельної реалізації

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <chrono>
#include <omp.h>

void parallelBucketSort(std::vector<double>& arr) {
    int n = arr.size();
    const int numBuckets = 1000;
    std::vector<std::vector<double>> buckets(numBuckets);

    for (int i = 0; i < n; ++i) {
        int index = static_cast<int>(numBuckets * arr[i]);
        if (index >= numBuckets) index = numBuckets - 1;
        buckets[index].push_back(arr[i]);
    }

    #pragma omp parallel for
    for (int i = 0; i < numBuckets; ++i) {
        std::sort(buckets[i].begin(), buckets[i].end());
    }

    int index = 0;
    for (int i = 0; i < numBuckets; ++i) {
        for (double value : buckets[i]) {
            arr[index++] = value;
        }
    }
}

int main() {
```

```

const int N = 100000000;
std::vector<double> arr(N);

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<double> dis(0.0, 1.0);
for (int i = 0; i < N; i++) {
    arr[i] = dis(gen);
}

auto start = std::chrono::high_resolution_clock::now();
parallelBucketSort(arr);
auto end = std::chrono::high_resolution_clock::now();

std::chrono::duration<double> duration = end - start;
std::cout << "Parallel sorting of " << N << " elements took
"
    << duration.count() << " seconds." << std::endl;

if (std::is_sorted(arr.begin(), arr.end())) {
    std::cout << "The array was successfully sorted." <<
std::endl;
} else {
    std::cout << "Error: the array is not sorted." <<
std::endl;
}

return 0;
}

```