Київський національний університет

імені Т.Шевченка

**Звіт**

до лабораторних робіт

з дисципліни «*Розподілене та паралельне програмування*»

***Студентки 4-го курсу***

***групи ТТП-41***

***Беденко Аліни***

***Київ-2024***

# Постановка задачі.

Реалізувати алгоритм розв’язання задачі сортування методом odd-even на великому наборі даних:

* Послідовно
* Паралельно за допомогою OpenMP
* Паралельно за допомогою MPI
* Проаналізувати отримані результати.

# Теоретичні відомості

## Сортування odd-even

Алгоритм сортування Odd-Even є варіацією бульбашкового сортування, призначений для паралельного використання, що забезпечує його ефективність на мультипроцесорних системах. Цей алгоритм працює шляхом послідовного порівняння пар сусідніх елементів у списку, розділяючи процес сортування на два типи кроків: "odd" кроки та "even" кроки.

На "odd" кроках, алгоритм порівнює пари елементів з непарними індексами з їх сусідами з парними індексами (наприклад, елементи 1 і 2, 3 і 4, і так далі). На "even" кроках, порівнюються пари з парними індексами з їх непарними сусідами (наприклад, елементи 2 і 3, 4 і 5, і так далі). Кожен крок виконується до тих пір, поки список не буде повністю відсортовано.

## OpenMP

OpenMP, що розшифровується як "Open Multi-Processing", є відкритим стандартом для паралельного програмування, який широко використовується для створення многопоточних додатків. Він дозволяє ефективно розподіляти виконання коду на кілька процесорів у системі зі спільною пам'яттю. Це особливо корисно для великих обчислювальних задач, які можна розбити на менші, паралельно виконувані фрагменти.

Основною перевагою OpenMP є його портативність і масштабованість. Стандарт підтримується багатьма компіляторами, включаючи ті, що працюють на основі C, C++ та Fortran, дозволяючи розробникам писати портативний код, який може виконуватися на різних платформах і архітектурах.

Паралелізм у OpenMP зазвичай реалізується за допомогою директив препроцесора, які вбудовуються в код програми. Ці директиви визначають області коду, які повинні виконуватися паралельно. Наприклад, найпростіша директива #pragma omp parallel ініціює блок коду, який виконується одночасно кількома потоками. Після цього можуть використовуватися додаткові директиви для управління розподілом даних та синхронізації між потоками.

OpenMP також включає можливості для розподілу ітерацій циклів між потоками (#pragma omp for), управління доступом до критичних ділянок коду, що вимагають взаємного виключення (#pragma omp critical), та створення бар'єрів для синхронізації потоків (#pragma omp barrier).

## MPI

MPI (Message Passing Interface) - це стандарт для паралельного програмування, який дозволяє взаємодіяти між процесами, які виконуються на різних вузлах обчислювального кластера чи комп'ютерної мережі. Основним принципом MPI є передача повідомлень між процесами, що дозволяє їм координувати свою діяльність та обмінюватися даними.

Одна з основних переваг MPI полягає в тому, що вона може бути використана на різних архітектурах, від простих кластерів до великих суперкомп'ютерів. Кожен процес може мати свій власний набір даних та інструкцій, і MPI дозволяє їм обмінюватися цими даними через ефективні механізми передачі повідомлень.

MPI надає широкий спектр функцій для керування процесами, включаючи створення, завершення, взаємодію та синхронізацію. Можна контролювати, як процеси обмінюються даними та як вони співпрацюють під час виконання обчислень. Також, подібно до OpenMP, MPI має багато бібліотек та інструментів, які полегшують розробку та налагодження паралельних програм.

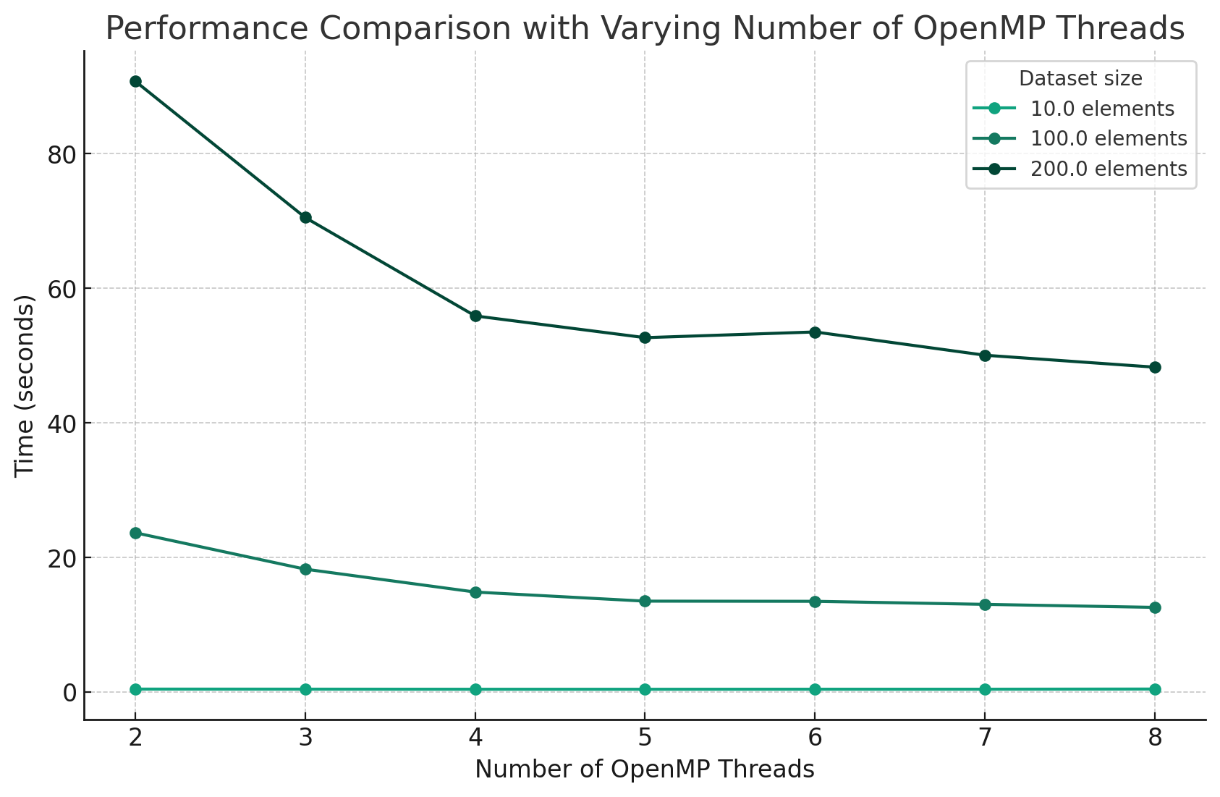
# Вирішення задачі

Код вирішених задач можна знайти за **посиланням**.

## OpenMP

*Виконання коду з використанням open mp у різній кількості потоків:*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| open\_mp | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 10.000 | 0.436 | 0.426 | 0.413 | 0.41 | 0.417 | 0.418 | 0.444 |
| 100.000 | 23.672 | 18.262 | 14.859 | 13.508 | 13.483 | 13.03 | 12.58 |
| 200.000 | 90.785 | 70.524 | 55.902 | 52.669 | 53.517 | 50.059 | 48.282 |



|  |  |
| --- | --- |
| one\_thread | 1 |
| 10.000 | 0.708 |
| 100.000 | 40.281 |
| 200.000 | 164 |

Також було додатково змінено структуру однопоточного коду для оптимізованої взаємодії з потоками.

В оптимізованому коді я використала OpenMP reduction, щоб уникнути зайвої синхронізації між потоками під час сортування odd-even. Кожен потік тепер виконує свою частину роботи незалежно, із локальним статусом сортування, який потім комбінується в кінці кожного проходу. Це знижує накладні витрати на управління потоками та покращує загальну продуктивність. Крім того, я задала автоматичне використання максимальної кількості доступних потоків, щоб забезпечити оптимальне використання системних ресурсів.

**Код для одного потоку:**

#include <iostream>

#include <fstream>

#include <vector>

#include <algorithm>

#include <ctime>

int main()

{

    std::ifstream inputFile("randoms10.txt");

    std::ofstream outputFile("output.txt");

    if (!inputFile)

    {

        std::cerr << "Error: Could not open input file." << std::endl;

        return 1;

    }

    if (!outputFile)

    {

        std::cerr << "Error: Could not create output file." << std::endl;

        return 1;

    }

    std::vector<int> numbers;

    int number;

    while (inputFile >> number)

    {

        numbers.push\_back(number);

    }

    clock\_t startTime = clock();

    int n = numbers.size();

    bool isSorted = false;

    while (!isSorted)

    {

        isSorted = true;

        for (int i = 1; i < n - 1; i += 2)

        {

            if (numbers[i] > numbers[i + 1])

            {

                std::swap(numbers[i], numbers[i + 1]);

                isSorted = false;

            }

        }

        for (int i = 0; i < n - 1; i += 2)

        {

            if (numbers[i] > numbers[i + 1])

            {

                std::swap(numbers[i], numbers[i + 1]);

                isSorted = false;

            }

        }

    }

    clock\_t endTime = clock();

    double executionTime = static\_cast<double>(endTime - startTime) / CLOCKS\_PER\_SEC;

    for (size\_t i = 0; i < numbers.size(); ++i)

    {

        outputFile << numbers[i] << std::endl;

    }

    std::cout << "Numbers sorted and written to output.txt." << std::endl;

    std::cout << "Execution time: " << executionTime << " seconds." << std::endl;

    std::cin >> executionTime;

    return 0;

}

**Код для багатьох потоків:**

#include <iostream>

#include <fstream>

#include <vector>

#include <algorithm>

#include <omp.h>

void oddEvenSort(std::vector<int>& numbers, int numThreads) {

    int n = numbers.size();

    bool globallySorted = false;

    while (!globallySorted) {

        bool locallySorted = true;

        #pragma omp parallel for num\_threads(numThreads) shared(numbers) reduction(&&:locallySorted)

        for (int i = 1; i < n - 1; i += 2) {

            if (numbers[i] > numbers[i + 1]) {

                std::swap(numbers[i], numbers[i + 1]);

                locallySorted = false;

            }

        }

        #pragma omp parallel for num\_threads(numThreads) shared(numbers) reduction(&&:locallySorted)

        for (int i = 0; i < n - 1; i += 2) {

            if (numbers[i] > numbers[i + 1]) {

                std::swap(numbers[i], numbers[i + 1]);

                locallySorted = false;

            }

        }

        globallySorted = locallySorted;

    }

}

int main() {

    std::ifstream inputFile("randoms10.txt");

    std::ofstream outputFile("output.txt");

    if (!inputFile) {

        std::cerr << "Error: Could not open input file." << std::endl;

        return 1;

    }

    std::vector<int> numbers;

    int number;

    while (inputFile >> number) {

        numbers.push\_back(number);

    }

    inputFile.close();

    int startThreads, endThreads;

    std::cout << "Enter the start and end number of threads: ";

    std::cin >> startThreads >> endThreads;

    for (int numThreads = startThreads; numThreads <= endThreads; ++numThreads) {

        std::vector<int> tempNumbers = numbers;

        double startTime = omp\_get\_wtime();

        oddEvenSort(tempNumbers, numThreads);

        double endTime = omp\_get\_wtime();

        std::cout << "Execution time with " << numThreads << " threads: "

                  << (endTime - startTime) << " seconds." << std::endl;

    }

    for (int num : numbers) {

        outputFile << num << std::endl;

    }

    outputFile.close();

    std::cin >> endThreads;

    return 0;

}

**MPI**

**10.000**

Elapsed parallel time: 0.048611

Elapsed nonparallel time: 0.008562

**50.000**

Elapsed parallel time: 16.308590

Elapsed nonparallel time: 3.298579

**Код**