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](https://github.com/aniilab/RPP)**.

## OpenMP

Для вирішення задачі була написана програма на C++ з використанням OpenMP - бібліотеки для паралельного програмування.

Програма зчитує масив чисел з заданого input файлу:

1. vector<int> numbers;

2. int number;

3. while (inputFile >> number) {

4. numbers.push\_back(number);

5. }

6. inputFile.close();

Також потрібно задати кількість потоків проміжком, на якому ми хочемо перевірити роботу паралельного програмування.

1.     int startThreads, endThreads;

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

3.     cin >> startThreads >> endThreads;

Далі ми виконуємо сортування для кожної кількості потоків, вимірюючи і виводячи час виконання.

Тепер перейдемо до функції із сортуванням, де відбувається розпаралелення задачі:

1. void oddEvenSort(vector<int>& numbers, int numThreads) {

2.     int n = numbers.size();

3.     bool globallySorted = false;

4.

5.     while (!globallySorted) {

6.         bool locallySorted = true;

7.

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

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

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

11.                 swap(numbers[i], numbers[i + 1]);

12.                 locallySorted = false;

13.             }

14.         }

15.

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

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

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

19.                 swap(numbers[i], numbers[i + 1]);

20.                 locallySorted = false;

21.             }

22.         }

23.

24.         globallySorted = locallySorted;

25.     }

26. }

27.

Маємо numbers – це посилання на вектор цілих чисел, які потрібно відсортувати та numThreads: кількість потоків, які будуть використовуватися для паралельних обчислень.

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

Ця директива OpenMP встановлює, що цикл буде виконаний паралельно з вказаною кількістю потоків.

shared(numbers) вказує, що масив numbers доступний усім потокам.

reduction(&&:locallySorted): використовується для комбінування значень змінної locallySorted з усіх потоків за допомогою логічного І (AND).

Після кожного повного проходу (парний + непарний) значення locallySorted (що є результатом редукції) використовується для оновлення globallySorted. Якщо locallySorted виявляється true, це означає, що жодних обмінів не відбулося під час останнього повного проходу, і масив вважається впорядкованим.

Цикли продовжуються, поки масив не стане повністю впорядкованим. Завдяки паралельному виконанню на кожному кроці алгоритм може бути значно швидшим на багатоядерних процесорах порівняно з послідовною версією сортування.

Виконання коду з використанням OpenMP у різній кількості потоків:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 10.000 | 0.708 | 0.436 | 0.426 | 0.413 | 0.41 | 0.417 | 0.418 | 0.444 |
| 100.000 | 40.281 | 23.672 | 18.262 | 14.859 | 13.508 | 13.483 | 13.03 | 12.58 |
| 200.000 | 164 | 90.785 | 70.524 | 55.902 | 52.669 | 53.517 | 50.059 | 48.282 |

## MPI

Цей код реалізує паралельний алгоритм сортування за методом парного обміну (odd-even sort) з використанням бібліотеки MPI для забезпечення реалізації моделі обміну повідомленнями між процесами.

Головна функція main ініціалізує MPI, зчитує дані з файлу "randoms.txt", розподіляє їх між процесами, виконує паралельне сортування та збирає відсортовані дані.

1. int main(int argc, char \*\*argv) {

2.     MPI\_Init(&argc, &argv);

3.     int rank, size;

4.     MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

5.     MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

6.

7.     ifstream inFile("randoms.txt");

8.     vector<int> array;

9.     int value;

10.     while (inFile >> value) {

11.         array.push\_back(value);

12.     }

13.     inFile.close();

14.

15.     if (rank == 0) {

16.         double startTimeSeq = MPI\_Wtime();

17.         sequentialOddEvenSort(array);

18.         double endTimeSeq = MPI\_Wtime();

19.         cout << "Sequential sort time: " << endTimeSeq - startTimeSeq << " seconds." << endl;

20.         ofstream outFile("SequentialSortedArray.txt");

21.         for (int num : array) {

22.             outFile << num << "\n";

23.         }

24.         outFile.close();

25.     }

1.  int N = array.size();

2.     int localN = N / size;

3.     int remainder = N % size; // Залишок елементів, які не розподілені рівномірно

4.

5.     // Кількість елементів, які потрібно додати до локального масиву для кожного процесу

6.     int extraElements = (rank < remainder) ? 1 : 0;

7.

8.     // Розмір локального масиву для кожного процесу з додатковими елементами, якщо потрібно

9.     localN += extraElements;

10.

11.     vector<int> localArr(localN);

12.     MPI\_Scatter(array.data(), localN, MPI\_INT, localArr.data(), localN, MPI\_INT, 0, MPI\_COMM\_WORLD);

13.

14.     double startTimePar = MPI\_Wtime();

15.     oddEvenSort(localArr, rank, size);

16.     double endTimePar = MPI\_Wtime();

17.

18.     vector<int> sortedArray(N);

19.     MPI\_Gather(localArr.data(), localN, MPI\_INT, sortedArray.data(), localN, MPI\_INT, 0, MPI\_COMM\_WORLD);

20.

21.     if (rank == 0)

22.     {

23.         cout << "Parallel sort time: " << endTimePar - startTimePar << " seconds." << endl;

24.         ofstream outFile("ParallelSortedArray.txt");

25.         for (int num : sortedArray)

26.         {

27.             outFile << num << "\n";

28.         }

29.         outFile.close();

30.     }

31.

32.     MPI\_Finalize();

33.     return 0;

Функція sequentialOddEvenSort реалізує послідовне сортування за методом парного обміну, яке використовується для порівняння з паралельним алгоритмом.

1. void sequentialOddEvenSort(vector<int>& arr) {

2.     bool isSorted = false;

3.     int n = arr.size();

4.     while (!isSorted) {

5.         isSorted = true;

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

7.             if (arr[i - 1] > arr[i]) {

8.                 swap(arr[i - 1], arr[i]);

9.                 isSorted = false;

10.             }

11.         }

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

13.             if (arr[i] > arr[i + 1]) {

14.                 swap(arr[i], arr[i + 1]);

15.                 isSorted = false;

16.             }

17.         }

18.     }

19. }

20.

Функція oddEvenSort виконує паралельне сортування за методом парного обміну, де кожен процес сортує свій власний підмасив та обмінюється даними з іншими процесами.

1. void oddEvenSort(vector<int> &localArr, int rank, int size)

2. {

3.     vector<int> partnerData(localArr.size());

4.     MPI\_Status status;

5.

6.     for (int phase = 0; phase < size; ++phase)

7.     {

8.         sort(localArr.begin(), localArr.end());

9.

10.         int partner;

11.         if (phase % 2 == 0)

12.         {

13.             if (rank % 2 == 0)

14.             {

15.                 partner = rank + 1;

16.             }

17.             else

18.             {

19.                 partner = rank - 1;

20.             }

21.         }

22.         else

23.         {

24.             if (rank % 2 == 0)

25.             {

26.                 partner = rank - 1;

27.             }

28.             else

29.             {

30.                 partner = rank + 1;

31.             }

32.         }

33.

34.         if (partner >= 0 && partner < size)

35.         {

36.             if (rank < partner)

37.             {

38.                 MPI\_Send(localArr.data(), localArr.size(), MPI\_INT, partner, 0, MPI\_COMM\_WORLD);

39.                 MPI\_Recv(partnerData.data(), localArr.size(), MPI\_INT, partner, 0, MPI\_COMM\_WORLD, &status);

40.             }

41.             else

42.             {

43.                 MPI\_Recv(partnerData.data(), localArr.size(), MPI\_INT, partner, 0, MPI\_COMM\_WORLD, &status);

44.                 MPI\_Send(localArr.data(), localArr.size(), MPI\_INT, partner, 0, MPI\_COMM\_WORLD);

45.             }

46.

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

48.             {

49.                 if (rank < partner)

50.                 {

51.                     localArr[i] = min(localArr[i], partnerData[i]);

52.                 }

53.                 else

54.                 {

55.                     localArr[i] = max(localArr[i], partnerData[i]);

56.                 }

57.             }

58.         }

59.     }

60. }

61.

Комунікація MPI здійснюється за допомогою функцій MPI\_Recv i MPI\_Send

Виконання коду з використанням MPI на різній кількості записів та вузлів:

|  |  |  |  |
| --- | --- | --- | --- |
|  | К-ть записів | Послідовне виконання | Паралельне виконанння |
| 7 | 10,000 | 0.392446 | 0.0026813 |
| 7 | 50,000 | 9.90673 | 0.0077384 |
| 6 | 10,000 | 9.90673 | 0.0077384 |
| 6 | 50,000 | 0.396841 | 0.0018031 |
| 5 | 10,000 | 0.36307 | 0.0012541 |
| 5 | 50,000 | 9.21052 | 0.0013537 |
| 4 | 10,000 | 0.36629 | 0.0006765 |
| 4 | 50,000 | 9.18197 | 0.0007686 |
| 3 | 10,000 | 0.383288 | 0.0020549 |
| 3 | 50,000 | 10.0929 | 0.0080615 |
| 2 | 10,000 | 0.363144 | 0.000461 |
| 2 | 50,000 | 9.38323 | 0.0003328 |

# Висновки

Отже, як можемо спостерігати з отриманих результатів, використання MPI для розпаралелення роботи алгоритму є досить ефективним та значно скорочує час виконання програми. Проте, на великих даних (50,000) може займати забагато часу - в такому випадку найкращим варіантом є використання MPI на 4 вузлах – це і не забирає забагато часу для створення вузлів і ефективно обробляє інформацію.

Щодо OpenMP, то ця технологія показала свою ефективність на різних даних та у різних кількостях потоків: чим більше потоків – тим швидше виконання. Проте, якщо порівнювати ці 2 технології на 10,000 записах, то MPI справляється з даною задачею набагато швидше.