КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА КІБЕРНЕТИКИ

Звіт до лабораторної роботи з курсу

«Розподілене та паралельне програмування»

Студента групи ТТП-41

Маркова Максима Юрійовича

викладач: Дерев’янченко Олександр Валерійович

Київ-2025

**Анотація**

В цій лабораторній роботі було реалізовано алгоритм **MergeSort** із використанням **OpenMP** та **MPI** для ефективного сортування великих масивів даних у паралельному середовищі.

**Тема роботи**

Паралельна та розподілена реалізація алгоритму MergeSort з використанням OpenMP та MPI

**Мета роботи**

Метою даної роботи є розробка та аналіз ефективності паралельного та розподіленого алгоритму MergeSort із використанням OpenMP та MPI для прискорення сортування великих масивів даних.

Для досягнення цієї мети необхідно:

* Реалізувати багатопотокову версію MergeSort з OpenMP.
* Реалізувати розподілену версію MergeSort з MPI.
* Дослідити вплив кількості потоків/процесів на швидкодію алгоритму.
* Порівняти продуктивність OpenMP та MPI на різних розмірах вхідних даних.
* Оцінити масштабованість та ефективність паралельного сортування.

Отримані результати дозволять визначити оптимальний підхід для сортування великих обсягів даних у багатоядерних та кластерних системах.

1. **Теоретичні відомості**

**1.1 MergeSort з використанням MPI**

Основні етапи роботи:

1. Ініціалізація середовища MPI
   1. Визначається кількість процесів.
   2. Кожен процес отримує свій унікальний ідентифікатор.
2. Завантаження даних (виконує лише один процес)
   1. Зчитується набір чисел із файлу.
   2. Визначається загальна кількість елементів.
3. Розподіл даних між процесами
   1. Дані поділяються на частини для кожного процесу.
   2. Якщо кількість процесів не ділить рівно об’єм даних, деякі процеси отримують на один елемент більше.
   3. Всі процеси отримують свої частини даних.
4. Локальне сортування
   1. Кожен процес виконує сортування власного набору даних за допомогою рекурсивного методу MergeSort.
5. Паралельне злиття відсортованих підмасивів
   1. Після локального сортування відбувається поступове об'єднання відсортованих частин.
   2. Процеси попарно передають свої відсортовані частини сусіднім процесам, які виконують злиття.
   3. На кожному етапі кількість активних процесів зменшується вдвічі, поки всі дані не об’єднаються в одному процесі.
6. Вимірювання часу виконання
   1. Визначається загальний час сортування.
7. Виведення результатів та завершення роботи
   1. Відсортовані дані можуть бути збережені або виведені на екран.
   2. Всі виділені ресурси звільняються.
   3. Завершується виконання програми MPI.

**1.2 MergeSort з використанням OpenMP**

Основні етапи роботи:

1. Зчитування вхідних даних
   1. Числа зчитуються з файлу та зберігаються у контейнері.
   2. Перевіряється коректність зчитування (наприклад, чи файл не порожній).
2. Підготовка до паралельного сортування
   1. Визначається кількість потоків, які будуть використовуватися.
   2. Створюється допоміжний масив для тимчасового зберігання елементів при злитті.
3. Рекурсивне сортування методом MergeSort
   1. Масив рекурсивно ділиться на дві частини.
   2. Для обох частин виконується сортування в окремих потоках за допомогою OpenMP-завдань.
   3. Глибина рекурсії, на якій використовується OpenMP, обмежена для запобігання надмірному створенню потоків.
4. Злиття відсортованих частин
   1. Після сортування двох частин виконується їх злиття в один відсортований підмасив.
   2. Використовується додатковий масив, щоб тимчасово зберегти проміжні значення.
5. Вимірювання часу виконання
   1. Перед запуском сортування фіксується початковий час.
   2. Після завершення алгоритму визначається загальний час роботи.
6. Виведення результату та завершення роботи
   1. Виводиться час виконання сортування.
   2. Виділені ресурси звільняються.

**2. Результати виконання програм**

**2.1 Послідовний MergeSort**

Компіляція: g++ Merge.cpp -o merge.exe

Запуск: ./merge.exe

При n = 1000000

1. 249 ms
2. 244 ms
3. 243 ms
4. 246 ms
5. 240 ms
6. 246 ms
7. 246 ms

Середній час виконання: 245 ms

При n = 10000000

1. 2837 ms
2. 2830 ms
3. 2830 ms
4. 2840 ms
5. 2825 ms
6. 2838 ms
7. 2836 ms

Середній час виконання: 2834.2 ms

**2.2 MergeSort з MPI**

Компіляція: g++ MPI.cpp -I"C:\Program Files (x86)\Microsoft SDKs\MPI\Include" -L"C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64" -lmsmpi -o MPI.exe

Запуск: mpiexec -n <к-ть процесів> MPI.exe <вхідний файл>

При n = 1000000 та к-ть процесів = 2

1. 100 ms
2. 97 ms
3. 96 ms
4. 99 ms
5. 97 ms
6. 97 ms
7. 97 ms

Середній час виконання: 97.4 ms

При n = 1000000 та к-ть процесів = 4

1. 56 ms
2. 56 ms
3. 56 ms
4. 56 ms
5. 66 ms
6. 56 ms
7. 56 ms

Середній час виконання: 56 ms

При n = 1000000 та к-ть процесів = 8

1. 47 ms
2. 46 ms
3. 46 ms
4. 47 ms
5. 49 ms
6. 47 ms
7. 48 ms

Середній час виконання: 47 ms

При n = 10000000 та к-ть процесів = 2

1. 1075 ms
2. 1074 ms
3. 1070 ms
4. 1067 ms
5. 1064 ms
6. 1060 ms
7. 1070 ms

Середній час виконання: 1069 ms

При n = 10000000 та к-ть процесів = 4

1. 637 ms
2. 615 ms
3. 656 ms
4. 616 ms
5. 609 ms
6. 640 ms
7. 626 ms

Середній час виконання: 626.8 ms

При n = 10000000 та к-ть процесів = 8

1. 453 ms
2. 452 ms
3. 441 ms
4. 448 ms
5. 463 ms
6. 469 ms
7. 464 ms

Середній час виконання: 456 ms

**2.3 MergeSort з OpenMP**

Компіляція: g++ -fopenmp OpenMP.cpp -o OpenMP

Запуск: ./OpenMP.exe <вхідний файл> <к-ть потоків>

При n = 1000000 та к-ть потоків = 2

1. 157 ms
2. 151 ms
3. 151 ms
4. 160 ms
5. 160 ms
6. 158 ms
7. 158 ms

Середній час виконання: 156.8 ms

При n = 1000000 та к-ть потоків = 4

1. 122 ms
2. 123 ms
3. 121 ms
4. 120 ms
5. 122 ms
6. 123 ms
7. 123 ms

Середній час виконання: 122.2 ms

При n = 1000000 та к-ть потоків = 8

1. 75 ms
2. 80 ms
3. 81 ms
4. 75 ms
5. 72 ms
6. 79 ms
7. 81 ms

Середній час виконання: 78 ms

При n = 10000000 та к-ть потоків = 2

1. 1775 ms
2. 1753 ms
3. 1785 ms
4. 1775 ms
5. 1793 ms
6. 1755 ms
7. 1742 ms

Середній час виконання: 1768.6 ms

При n = 10000000 та к-ть потоків = 4

1. 1378 ms
2. 1365 ms
3. 1365 ms
4. 1370 ms
5. 1345 ms
6. 1376 ms
7. 1375 ms

Середній час виконання: 1370.2 ms

При n = 10000000 та к-ть потоків = 8

1. 764 ms
2. 799 ms
3. 760 ms
4. 769 ms
5. 776 ms
6. 775 ms
7. 810 ms

Середній час виконання: 776.6 ms

**2.4 Порівняльна характеристика**

#### Для n = 1 000 000

| Реалізація | Послідовний MergeSort | MPI (2 процеси) | MPI (4 процеси) | MPI (8 процесів) | OpenMP (2 потоки) | OpenMP (4 потоки) | OpenMP (8 потоків) |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Час (мс) | 245 | 97.4 | 56 | 47 | 156.8 | 122.2 | 78 |

#### Для n = 10 000 000

| Реалізація | Послідовний MergeSort | MPI (2 процеси) | MPI (4 процеси) | MPI (8 процесів) | OpenMP (2 потоки) | OpenMP (4 потоки) | OpenMP (8 потоків) |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Час (мс) | 2834.2 | 1069 | 626.8 | 456 | 1768.6 | 1370.2 | 776.6 |

**Висновки**

В результаті виконання лабораторної роботи було розроблено та проаналізовано реалізації алгоритму MergeSort у послідовному, багатопроцесійному (MPI) та багатопотоковому (OpenMP) варіантах. Було проведено експерименти для різних обсягів даних та кількості потоків/процесів, що дозволило оцінити ефективність кожного підходу.

Перспективами розвитку роботи є оптимізація використання пам’яті та зменшення накладних витрат на комунікацію в MPI-реалізації, а також вдосконалення стратегій балансування навантаження для OpenMP. Також можливе впровадження гібридного підходу, що поєднує багатопоточність OpenMP та розподілені обчислення MPI для ще більшого прискорення алгоритму.

Для аналізу продуктивності було виконано такі завдання:

* Реалізовано алгоритм MergeSort у трьох варіантах: послідовному, MPI та OpenMP.
* Згенеровано вхідні дані з рівномірним розподілом значень у заданому діапазоні.
* Виконано експериментальні вимірювання часу сортування для різних n та кількості потоків/процесів.
* Проведено порівняльний аналіз отриманих результатів.

Отримані результати показали, що MPI забезпечує найкращу продуктивність на великих обсягах даних, тоді як OpenMP є зручним рішенням для багатоядерних процесорів без міжпроцесної комунікації. Послідовний MergeSort, хоч і працює коректно, демонструє значно гіршу швидкодію та підходить лише для малих розмірів вхідних даних.

**Додатки**

**Послідовний MergeSort:**

#include <iostream>

#include <vector>

#include <fstream>

#include <sstream>

#include <ctime>

#include <iomanip>

void merge(std::vector<int>& arr, std::vector<int>& temp, int left, int mid, int right) {

int i = left, j = mid + 1, k = left;

while (i <= mid && j <= right) {

if (arr[i] <= arr[j]) {

temp[k++] = arr[i++];

} else {

temp[k++] = arr[j++];

}

}

while (i <= mid) {

temp[k++] = arr[i++];

}

while (j <= right) {

temp[k++] = arr[j++];

}

for (i = left; i <= right; i++) {

arr[i] = temp[i];

}

}

void mergeSort(std::vector<int>& arr, std::vector<int>& temp, int left, int right) {

if (left < right) {

int mid = (left + right) / 2;

mergeSort(arr, temp, left, mid);

mergeSort(arr, temp, mid + 1, right);

merge(arr, temp, left, mid, right);

}

}

void readFile(const std::string& filename, std::vector<int>& arr) {

std::ifstream file(filename);

int num;

while (file >> num) {

arr.push\_back(num);

}

file.close();

}

int main() {

std::vector<int> arr;

readFile("input.txt", arr);

std::vector<int> temp(arr.size());

clock\_t start\_time = clock();

mergeSort(arr, temp, 0, arr.size() - 1);

clock\_t end\_time = clock();

double execution\_time = double(end\_time - start\_time) / CLOCKS\_PER\_SEC;

std::cout << "Execution time: " << execution\_time << " seconds" << std::setprecision(10) << std::endl;

return 0;

}

**MergeSort з MPI:**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <time.h>

#define INPUT\_FILE "input.txt"

int \*mergeArrays(int \*leftArray, int leftSize, int \*rightArray, int rightSize) {

int leftIndex = 0, rightIndex = 0, mergedIndex = 0;

int \*mergedArray;

int mergedSize = leftSize + rightSize;

mergedArray = (int \*)malloc(mergedSize \* sizeof(int));

while ((leftIndex < leftSize) && (rightIndex < rightSize)) {

if (leftArray[leftIndex] <= rightArray[rightIndex]) {

mergedArray[mergedIndex++] = leftArray[leftIndex++];

} else {

mergedArray[mergedIndex++] = rightArray[rightIndex++];

}

}

while (leftIndex < leftSize)

mergedArray[mergedIndex++] = leftArray[leftIndex++];

while (rightIndex < rightSize)

mergedArray[mergedIndex++] = rightArray[rightIndex++];

return mergedArray;

}

void performSort(int \*arr, int left, int right) {

if (left >= right) return;

int mid = (left + right) / 2;

performSort(arr, left, mid);

performSort(arr, mid + 1, right);

int leftPartSize = mid - left + 1;

int rightPartSize = right - mid;

int \*sortedArray = mergeArrays(arr + left, leftPartSize, arr + mid + 1, rightPartSize);

for (int i = 0; i < leftPartSize + rightPartSize; i++)

arr[left + i] = sortedArray[i];

free(sortedArray);

}

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

int \*globalArray = NULL;

int \*localArray;

int \*receivedArray;

int totalElements = 0, localSize;

int processRank, totalProcesses;

int mergeStep;

double startTime, endTime;

MPI\_Status mpiStatus;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &processRank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &totalProcesses);

if (processRank == 0) {

FILE \*inputFile = fopen(INPUT\_FILE, "r");

int temp;

while (fscanf(inputFile, "%d", &temp) == 1) {

totalElements++;

}

globalArray = (int \*)malloc(totalElements \* sizeof(int));

rewind(inputFile);

for (int i = 0; i < totalElements; i++)

fscanf(inputFile, "%d", &globalArray[i]);

fclose(inputFile);

}

MPI\_Bcast(&totalElements, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

localSize = totalElements / totalProcesses;

int remainder = totalElements % totalProcesses;

if (processRank < remainder) localSize++;

localArray = (int \*)malloc(localSize \* sizeof(int));

int \*sendCounts = NULL;

int \*displacements = NULL;

if (processRank == 0) {

sendCounts = (int \*)malloc(totalProcesses \* sizeof(int));

displacements = (int \*)malloc(totalProcesses \* sizeof(int));

int currentIndex = 0;

for (int i = 0; i < totalProcesses; i++) {

sendCounts[i] = totalElements / totalProcesses + (i < remainder ? 1 : 0);

displacements[i] = currentIndex;

currentIndex += sendCounts[i];

}

}

MPI\_Scatterv(globalArray, sendCounts, displacements, MPI\_INT, localArray, localSize, MPI\_INT, 0, MPI\_COMM\_WORLD);

startTime = MPI\_Wtime();

performSort(localArray, 0, localSize - 1);

mergeStep = 1;

while (mergeStep < totalProcesses) {

if (processRank % (2 \* mergeStep) == 0) {

if (processRank + mergeStep < totalProcesses) {

int receivedSize;

MPI\_Recv(&receivedSize, 1, MPI\_INT, processRank + mergeStep, 0, MPI\_COMM\_WORLD, &mpiStatus);

receivedArray = (int \*)malloc(receivedSize \* sizeof(int));

MPI\_Recv(receivedArray, receivedSize, MPI\_INT, processRank + mergeStep, 0, MPI\_COMM\_WORLD, &mpiStatus);

localArray = mergeArrays(localArray, localSize, receivedArray, receivedSize);

localSize += receivedSize;

free(receivedArray);

}

} else {

int targetProcess = processRank - mergeStep;

MPI\_Send(&localSize, 1, MPI\_INT, targetProcess, 0, MPI\_COMM\_WORLD);

MPI\_Send(localArray, localSize, MPI\_INT, targetProcess, 0, MPI\_COMM\_WORLD);

break;

}

mergeStep \*= 2;

}

endTime = MPI\_Wtime();

if (processRank == 0) {

printf("\nNumber of elements: %d; Number of processes: %d; Time: %f\n\n", totalElements, totalProcesses, endTime - startTime);

free(globalArray);

}

free(localArray);

if (processRank == 0) {

free(sendCounts);

free(displacements);

}

MPI\_Finalize();

return 0;

}

**MergeSort з OpenMP:**

#include <iostream>

#include <fstream>

#include <vector>

#include <omp.h>

void merge(std::vector<int>& arr, int l, int m, int r, std::vector<int>& tmp) {

int i = l, j = m + 1, k = l;

while (i <= m && j <= r) {

tmp[k++] = (arr[i] < arr[j]) ? arr[i++] : arr[j++];

}

while (i <= m) tmp[k++] = arr[i++];

while (j <= r) tmp[k++] = arr[j++];

for (int x = l; x <= r; ++x) {

arr[x] = tmp[x];

}

}

void mergeSortParallel(std::vector<int>& arr, int l, int r, std::vector<int>& tmp, int depth = 0) {

if (l < r) {

int m = l + (r - l) / 2;

#pragma omp task shared(arr, tmp) if(depth < 3)

mergeSortParallel(arr, l, m, tmp, depth + 1);

#pragma omp task shared(arr, tmp) if(depth < 3)

mergeSortParallel(arr, m + 1, r, tmp, depth + 1);

#pragma omp taskwait

merge(arr, l, m, r, tmp);

}

}

void readFile(const std::string& filename, std::vector<int>& arr) {

std::ifstream file(filename);

int num;

while (file >> num) {

arr.push\_back(num);

}

file.close();

}

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

if (argc != 3) {

std::cerr << "Usage: ./merge\_sort\_omp <input\_file> <num\_threads>" << std::endl;

return 1;

}

std::vector<int> arr;

readFile(argv[1], arr);

int n = arr.size();

int num\_threads = std::stoi(argv[2]);

omp\_set\_num\_threads(num\_threads);

std::vector<int> tmp(n);

double start\_time = omp\_get\_wtime();

#pragma omp parallel

{

#pragma omp single

mergeSortParallel(arr, 0, n - 1, tmp);

}

double end\_time = omp\_get\_wtime();

std::cout << "Execution time: " << (end\_time - start\_time) << " seconds" << std::endl;

return 0;

}

**Джерела**

1. The OpenMP API specification for parallel programming [Електронний ресурс] - Режим доступу до ресурсу: https://www.openmp.org/
2. Microsoft MPI [Електронний ресурс] - Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi>
3. MPI Reference [Електронний ресурс] - Режим доступу до ресурсу: https://learn.microsoft.com/en-us/message-passing-interface/mpi-reference