Міністерство освіти і науки України
НТУУ «КПІ ім. Ігоря Сікорського»
Навчально-науковий інститут атомної та теплової енергетики
Кафедра цифрових технологій в енергетиці

Лабораторна робота №5

з дисципліни «Технології паралельних обчислень в енергетичних комплексах»

Тема «Алгоритми сортування»

Варіант №19

Студента 3-го курсу НН ІАТЕ гр. ТР-12
Ковальова Олександра
Перевірив: ас., Софієнко А. Ю.

КИЇВ 2024

Мета роботи. Розробити паралельні реалізації алгоритмів сортування даних з допомогою технології MPI.

Завдання: Розробити паралельну реалізацію швидкого сортування в середовищі MPI.

Хід роботи

Програмний код:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

// Swap two numbers
void swap(int* arr, int i, int j)
{
    int t = arr[i];
    arr[i] = arr[j];
    arr[j] = t;
}

// Quicksort function itself :D
void quicksort(int* arr, int start, int end)
{
    int pivot, index;

    // Basic case
    if (end <= 1)
        return;

    // Pick pivot and swap with first element Pivot is middle element
    pivot = arr[start + end / 2];
    swap(arr, start, start + end / 2);

    // Partitioning Steps
    index = start;

    // Iterate over the range [start, end]
    for (int i = start + 1; i < start + end; i++) {

        // Swap if the element is less than the pivot element
        if (arr[i] < pivot) {
            index++;
            swap(arr, i, index);
        }
    }

    // Swap the pivot into place
    swap(arr, start, index);

    // Recursive Call for sorting of quick sort function
    quicksort(arr, start, index - start);
    quicksort(arr, index + 1, start + end - index - 1);
}

// Function that merges the two arrays
int* merge(int* arr1, int n1, int* arr2, int n2)
{
    int* result = (int*)malloc((n1 + n2) * sizeof(int));
    int i = 0;
    int j = 0;
    int k;

    for (k = 0; k < n1 + n2; k++) {
        if (i >= n1) {
            result[k] = arr2[j];
            j++;
        }
        else if (j >= n2) {
            result[k] = arr1[i];
            i++;
        }
        // Indices in bounds as i < n1 && j < n2
        else if (arr1[i] < arr2[j]) {
            result[k] = arr1[i];
```

```
            i++;
        }
        // v2[j] <= v1[i]
        else {
            result[k] = arr2[j];
            j++;
        }
    }
    return result;
}

int main(int argc, char* argv[])
{
    int number_of_elements;
    int* data = NULL;
    int chunk_size, own_chunk_size;
    int* chunk;
    FILE* file = NULL;
    double time_taken;
    MPI_Status status;

    int number_of_process, rank_of_process;
    int rc = MPI_Init(&argc, &argv);

    if (rc != MPI_SUCCESS) {
        printf("Error in creating MPI program.\nTerminating...\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    MPI_Comm_size(MPI_COMM_WORLD, &number_of_process);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank_of_process);

    if (rank_of_process == 0) {
             if (argc != 3) {
                    printf("Waiting for 2 args: input and output files\n");
                    exit(-1);
             }

        // Opening the file
        file = fopen(argv[1], "r");

        // Printing Error message if any
        if (file == NULL) {
            printf("Error in opening file\n");
            exit(-1);
        }

        // Reading number of Elements in file ...
        // First Value in file is number of Elements
        fscanf(file, "%d", &number_of_elements);
        printf("Input array length: %d elements\n",
               number_of_elements);

        // Computing chunk size
        chunk_size
            = (number_of_elements % number_of_process == 0)
                  ? (number_of_elements / number_of_process)
                  : (number_of_elements / number_of_process
                     - 1);

        data = (int*)malloc(number_of_process * chunk_size
                            * sizeof(int));

        // Reading the rest elements in which operation is being performed
        for (int i = 0; i < number_of_elements; i++) {
            fscanf(file, "%d", &data[i]);
        }
        // Padding data with zero
        for (int i = number_of_elements;
             i < number_of_process * chunk_size; i++) {
            data[i] = 0;
        }

        printf("\n");

        fclose(file);
        file = NULL;

             printf("Started sorting process...\n\n");
    }
```

```
            // Blocks all process until reach this point
            MPI_Barrier(MPI_COMM_WORLD);

            // Starts Timer
            time_taken -= MPI_Wtime();

            // BroadCast the Size to all the process from root process
            MPI_Bcast(&number_of_elements, 1, MPI_INT, 0, MPI_COMM_WORLD);

            // Computing chunk size
            chunk_size
                = (number_of_elements % number_of_process == 0)
                      ? (number_of_elements / number_of_process)
                      : number_of_elements
                            / (number_of_process - 1);

            // Calculating total size of chunk according to bits
            chunk = (int*)malloc(chunk_size * sizeof(int));

            // Scatter the chuck size data to all process
            MPI_Scatter(data, chunk_size, MPI_INT, chunk,
                        chunk_size, MPI_INT, 0, MPI_COMM_WORLD);
            free(data);
            data = NULL;

            // Compute size of own chunk and then sort them using quick sort
            own_chunk_size = (number_of_elements
                              >= chunk_size * (rank_of_process + 1))
                                 ? chunk_size
                                 : (number_of_elements
                                    - chunk_size * rank_of_process);

            // Sorting array with quick sort for every chunk as called by process
            quicksort(chunk, 0, own_chunk_size);

            for (int step = 1; step < number_of_process;
                 step = 2 * step) {
                if (rank_of_process % (2 * step) != 0) {
                    MPI_Send(chunk, own_chunk_size, MPI_INT,
                             rank_of_process - step, 0,
                             MPI_COMM_WORLD);
                    break;
                }

                if (rank_of_process + step < number_of_process) {
                    int received_chunk_size
                        = (number_of_elements
                           >= chunk_size
                                  * (rank_of_process + 2 * step))
                              ? (chunk_size * step)
                              : (number_of_elements
                                 - chunk_size
                                       * (rank_of_process + step));
                    int* chunk_received;
                    chunk_received = (int*)malloc(
                        received_chunk_size * sizeof(int));
                    MPI_Recv(chunk_received, received_chunk_size,
                             MPI_INT, rank_of_process + step, 0,
                             MPI_COMM_WORLD, &status);

                    data = merge(chunk, own_chunk_size,
                                 chunk_received,
                                 received_chunk_size);

                    free(chunk);
                    free(chunk_received);
                    chunk = data;
                    own_chunk_size
                        = own_chunk_size + received_chunk_size;
                }
            }

            // Stop the timer
            time_taken += MPI_Wtime();

            // Opening the other file as taken form input and writing it to the file and giving it as the
output
            if (rank_of_process == 0) {
                // Opening the file
                file = fopen(argv[2], "w");
```

```
        if (file == NULL) {
            printf("Error in opening file... \n");
            exit(-1);
        }

        // Printing total number of elements in the file
        fprintf(file, "Total number of Elements in the array : %d\n", own_chunk_size);

        // Printing the value of array in the file
        for (int i = 0; i < own_chunk_size; i++) {
            fprintf(file, "%d ", chunk[i]);
        }

        // Closing the file
        fclose(file);

        printf("Results printed in output file\n");
        printf("Quick-sorted %d ints on %d processor cores: %.6f secs\n",
            number_of_elements, number_of_process,
            time_taken);
    }

    MPI_Finalize();
    return 0;
}
```

Результати.

Послідовна версія:

```
┌─alex@host ~/PCT/Lab5
└─$ sudo ./start.sh
 __         ___      .______    _____
|  |       /   \     |   _  \  |  ___|
|  |      /  ^  \    |  |_)  | | |__
|  |     /  /_\  \   |   _  <  |___ \
|  `----./  _____  \  |  |_)  |  ___) |
|_______/__/     \__\ |______/  |____/

---------- NON-PARALLEL ----------
Input array length: 1000000 elements

Started sorting process...

Results printed in output file
Quick-sorted 1000000 ints on 1 processor cores: 0.116561 secs

 Performance counter stats for 'mpirun --allow-run-as-root -np 1 ./out/non-parallel ./data/input.txt ./data/output.txt':

            265.39 msec task-clock                #    0.512 CPUs utilized
               271      context-switches          #    1.021 K/sec
                 7      cpu-migrations            #   26.376 /sec
             7,925      page-faults               #   29.862 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

       0.518144759 seconds time elapsed

       0.225822000 seconds user
       0.041679000 seconds sys


Result: (last 100 bytes)

3 999984 999984 999984 999988 999988 999989 999989 999990 999991 999991 999994 999996 999998 999999
```

Паралельна версія:

```
----------   PARALLEL   ----------
Input array length: 1000000 elements

Started sorting process...

Results printed in output file
Quick-sorted 1000000 ints on 4 processor cores: 0.050321 secs

 Performance counter stats for 'mpirun --allow-run-as-root -np 4 ./out/parallel ./data/input.txt ./data/output.txt':

            546.09 msec task-clock                #    1.053 CPUs utilized
             1,407      context-switches          #    2.576 K/sec
                28      cpu-migrations            #   51.273 /sec
            15,793      page-faults               #   28.920 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

       0.518465536 seconds time elapsed

       0.485680000 seconds user
       0.069377000 seconds sys


Result: (last 100 bytes)

3 999984 999984 999984 999988 999988 999989 999989 999990 999991 999991 999994 999996 999998 999999 %
```

Контрольні запитання:

1) **У чому полягає постановка задачі сортування даних?**
   Постановка задачі сортування даних передбачає розташування елементів послідовності у відповідності з певним критерієм, який може бути зростанням, спаданням або іншим порядком. Мета полягає в організації даних для полегшення їх подальшого пошуку, аналізу або обробки.

2) **Наведіть кілька прикладів алгоритмів сортування. Яка обчислювальна складність наведених алгоритмів?**
   Деякі приклади алгоритмів сортування: бульбашкою (Bubble Sort), вибором (Selection Sort), вставками (Insertion Sort), злиттям (Merge Sort), швидке сортування (Quick Sort).
   Обчислювальна складність залежить від конкретного алгоритму та властивостей вхідних даних, але в основному може бути $O(n^2)$ для простих алгоритмів, таких як бульбашкове сортування, та $O(n * log_n)$ для більш ефективних, таких як швидке сортування (Quick Sort) чи злиття (Merge Sort).

3) **Яка операція є базовою для задачі сортування даних?**
   Базова операція для задачі сортування даних – це порівняння між елементами даних та їх обмін місцями для визначення відносного порядку.

4) **У чому суть паралельного узагальнення базової операції задачі сортування даних?**
   Суть паралельного узагальнення базової операції сортування даних полягає у розділенні вхідних даних між різними процесами для одночасного виконання порівнянь та обміну інформацією між ними з метою швидшого сортування.

5) **Чим є алгоритм парно-непарної перестановки?**
   Алгоритм парно-непарної перестановки – це метод сортування, де використовується процес рекурсивного поділу та обміну елементів між процесами MPI за допомогою парних та непарних ітерацій, щоб відсортувати дані.

6) **У чому полягає паралельний варіант алгоритму Шелла? Які основні відмінності цього варіанта паралельного алгоритму сортування від методу парно-непарної перестановки?**
   Паралельний варіант алгоритму Шелла використовує той самий алгоритм, що і класичний алгоритм Шелла, але розділяє вхідні дані між різними процесами MPI для одночасного сортування різних частин послідовності. Основною відмінністю від методу парно-непарної перестановки є те, що в алгоритмі Шелла використовується специфічна послідовність кроків сортування, що може призводити до кращої продуктивності в деяких випадках.

7) **Чим є паралельний варіант алгоритму швидкого сортування?**
   Паралельний варіант алгоритму швидкого сортування – це метод, який розподіляє вхідні дані між різними процесами MPI для одночасного розділення та сортування частин послідовності, що дозволяє прискорити процес сортування.

8) **Що залежить від правильного вибору головного елемента для паралельного алгоритму швидкого сортування?**
   Від правильного вибору головного елемента для паралельного алгоритму швидкого сортування залежить ефективність розділення вхідних даних між процесами MPI та швидкість сортування.

9) **Які способи вибору головного елемента можуть бути запропоновані?**
   Можна використовувати різні способи для вибору головного елемента в паралельному алгоритмі швидкого сортування, такі як вибір випадкового елемента, медіана серед трьох, вибір елемента з фіксованої позиції або використання статистики про вхідні дані для визначення оптимального елемента.

10) **Для яких топологій можуть застосовуватися розглянуті алгоритми сортування?**
    Розглянуті алгоритми сортування можуть застосовуватися для будь-яких топологій, які дозволяють процесам обмінюватися даними між собою, таких як лінійна, кубічна, гіперкуб, кільце тощо.