

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Кафедра Системотехніки

Звіт

з лабораторної роботи №3

з дисципліни: «Технології Високопродуктивних Обчислень»

з теми: «ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ
MPI»

Виконав:

здобувач освіти першого
(бакалаврського) рівня освіти гр.

КНТ-22-1

Орлов О. С.

Варіант: №9

Перевірів:

Професор кафедри СТ
Міщеряков Ю. В.

3 ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ MPI

3.1 Мета роботи

Ознайомлення з принципами роботи паралельних MPI-програм, засобами їх розробки і запуску.

3.2 Хід роботи

Варіант розраховується за формулою: $S = (14 - N)^2 \cdot 1000$, де N — номер студента у списку. Для $N = 9$:

$$S = (14 - 9)^2 \cdot 1000 = 5^2 \cdot 1000 = 25000 \quad (3.1)$$

Топологія «Кільце» характеризується тим, що кожен вузол з'єднаний рівно з двома сусідніми. Основні характеристики топології «Кільце» для p процесорів:

- діаметр: $\frac{p}{2}$;
- зв'язність: 2;
- ширина бінарного розподілу: 2;
- вартість: p .

Для сортування на кільцевій топології обрано алгоритм парного-непарного сортування (Odd-Even Transposition Sort). Масив розбивається на p частин. На кожній ітерації процеси обмінюються даними із сусідами (парні з непарними, потім навпаки), зливають отримані масиви та залишають собі відповідну (старшу або молодшу) частину даних.

Комунікаційна трудомісткість: Алгоритм потребує p етапів обміну. На кожному етапі передається блок розміром $\frac{N}{p}$.

$$T_{\text{comm}} \approx p \cdot \left(t_s + \frac{N}{p} \cdot t_w \right) \quad (3.2)$$

де t_s — латентність, t_w — час передачі одного слова. Оскільки у кільці обмін відбувається лише з безпосередніми сусідами, конфлікти за канал мінімізовані, але діаметр мережі впливає на кількість необхідних ітерацій для повного впорядкування даних.

Нижче наведено код програми, що реалізує паралельне сортування. Програма автоматично визначає кількість процесів та розподіляє масив.

```

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 25000

int compare_doubles(const void *a, const void *b) {
    double arg1 = *(const double *)a;
    double arg2 = *(const double *)b;
    if (arg1 < arg2)
        return -1;
    if (arg1 > arg2)
        return 1;
    return 0;
}

void compare_split(int n_local, double *local_data, double
*recv_data,
                    int keep_small) {
    int i = 0, j = 0, k = 0;

    double *merged = (double *)malloc(2 * n_local * sizeof(double));
    while (i < n_local && j < n_local) {
        if (local_data[i] < recv_data[j])
            merged[k++] = local_data[i++];
        else
            merged[k++] = recv_data[j++];
    }
    while (i < n_local)
        merged[k++] = local_data[i++];
    while (j < n_local)
        merged[k++] = recv_data[j++];

    if (keep_small) {
        for (i = 0; i < n_local; i++)
            local_data[i] = merged[i];
    } else {
        for (i = 0; i < n_local; i++)
            local_data[i] = merged[n_local + i];
    }
    free(merged);
}

int main(int argc, char **argv) {
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int n_local = N / size;
    double *local_data = (double *)malloc(n_local * sizeof(double));
    double *recv_data = (double *)malloc(n_local * sizeof(double));

    srand(time(NULL) + rank);
    for (int i = 0; i < n_local; i++) {
        local_data[i] = (double)(rand() % 1001);
    }

    double start_time = MPI_Wtime();

    qsort(local_data, n_local, sizeof(double), compare_doubles);

```

```

for (int step = 0; step < size; step++) {
    int partner = -1;

    if (step % 2 == 0) {
        if (rank % 2 == 0)
            partner = rank + 1;
        else
            partner = rank - 1;
    } else {
        if (rank % 2 != 0)
            partner = rank + 1;
        else
            partner = rank - 1;
    }

    if (partner >= 0 && partner < size) {
        MPI_Sendrecv(local_data, n_local, MPI_DOUBLE, partner, 0,
recv_data,
                    n_local, MPI_DOUBLE, partner, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);

        if (rank < partner) {
            compare_split(n_local, local_data, recv_data, 1);
        } else {
            compare_split(n_local, local_data, recv_data, 0);
        }
    }
}

double end_time = MPI_Wtime();

double *global_data = NULL;
if (rank == 0) {
    global_data = (double *)malloc(N * sizeof(double));
}

MPI_Gather(local_data, n_local, MPI_DOUBLE, global_data, n_local,
MPI_DOUBLE,
          0, MPI_COMM_WORLD);

if (rank == 0) {
    printf("Processes: %d\n", size);
    printf("N: %d\n", N);
    printf("Time: %f s\n", end_time - start_time);

    int sorted = 1;
    for (int i = 1; i < N; i++) {
        double next_el = global_data[i];
        double current_el = global_data[i - 1];
        // printf("[%0f %0f] i:%d", current_el, next_el, i);
        if (current_el > next_el) {
            sorted = 0;
            break;
        }
    }
    printf("Verification: %s\n", sorted ? "OK" : "FAILED");
    free(global_data);
}

free(local_data);
free(recv_data);

```

```

MPI_Finalize();
return 0;
}

```

Нижче наведено результати запуску програми для різної кількості процесів ($p = 1, 2, 4$).

```

dxrkness@pc ~/u/t/lab3> mpiexec -n 1 main
Processes: 1
N: 25000
Time: 0.006011 s
Verification: OK

```

Рисунок 3.1 – Результат виконання програми для одного процесу

```

dxrkness@pc ~/u/t/lab3> mpiexec -n 2 main
Processes: 2
N: 25000
Time: 0.006070 s
Verification: OK

```

Рисунок 3.2 – Результат виконання програми для двох процесів

```

dxrkness@pc ~/u/t/lab3> mpiexec -n 4 main
Processes: 4
N: 25000
Time: 0.003211 s
Verification: OK

```

Рисунок 3.3 – Результат виконання програми для чотирьох процесів

Для масиву розміром $N = 25000$:

К-сть процесів	Час (с)	Прискорення S_p	Ефективність E_p
1	0.00601	1.0	100%
2	0.00607	0.99	0.495
4	0.00321	1.87	0.5

3.3 Висновки

У ході лабораторної роботи було розроблено програмний засіб мовою C з використанням бібліотеки MPI для сортування масиву методом парного-непарного обміну (Odd-Even Sort), що відповідає кільцевій топології.

За результатами експериментів для розмірності завдання $N = 25000$ було встановлено наступне:

За результатами експериментів для розмірності завдання $N = 25000$ було отримано наступні показники:

- а) послідовне виконання ($p = 1$): Час виконання склав 0.006011 с. Це є базовим показником для розрахунку прискорення;
- б) виконання на 2 процесорах ($p = 2$): Час склав 0.006070 с. Спостерігається незначне уповільнення ($S_2 \approx 0.99$). Це свідчить про те, що для двох процесів виграш від розподілу обчислень (сортування двох підмасивів по 12 500 елементів) був повністю нівельований накладними витратами на ініціалізацію MPI та передачу даних між процесорами;
- в) виконання на 4 процесорах ($p = 4$): Час склав 0.003211 с. Отримано прискорення $S_4 = \frac{0.006011}{0.003211} \approx 1.87$. Ефективність становить $E_4 \approx 46.8\%$.

На відміну від запуску на 2 процесорах, при розбитті задачі на 4 частини (по 6 250 елементів на процес) зменшення часу локального сортування (яке має складність $O(N \log N)$) виявилось достатньо суттєвим, щоб перекрити комунікаційні витрати. Це дозволило отримати майже двократний приріст швидкодії порівняно з послідовною версією навіть на відносно малому масиві даних.

Таким чином, експеримент підтвердив, що ефективність MPI-програми залежить не тільки від розміру даних, але й від балансу між обчислювальною складністю локальної задачі та інтенсивністю обмінів.