**Лабораторная работа №5**

**Управление группами процессов и коммуникаторами**

**Цель:** изучить основные принципы управления группами процессов и коммуникаторами в технологии MPI на примере использования в рамках языка С++.

**Лабораторные задания** (№ варианта = 2)

**Задание.** Реализуйте на основе технологии MPI многопроцессную программу с использованием **групп процессов и коммуникаторов** в соответствии с вариантом задания**.** Проверьте корректность работы программы. **Результаты занесите в отчет.**

|  |  |
| --- | --- |
| **Вариант** | **Задание** |
| **2** | Реализуйте тип «длинный вектор».  Напишите программу, которая осуществляет сортировку слиянием большого массива N>10000 элементов. Элементы – случайные числа в промежутке от 0 до 100000. |

#include <stdio.h>

#include "mpi.h"

#include <random>

#include <iostream>

int generate\_int(int min\_v, int max\_v) {

return min\_v + std::rand() % (max\_v - min\_v);

}

int\* merge(int\* arr1, int n1, int\* arr2, int n2)

{

int\* arr3 = new int[n1 + n2];

int i = 0, j = 0, k = 0;

// Traverse both array

while ((i < n1) && (j < n2))

{

if (arr1[i] < arr2[j])

arr3[k++] = arr1[i++];

else

arr3[k++] = arr2[j++];

}

// Store remaining elements of first array

while (i < n1)

arr3[k++] = arr1[i++];

// Store remaining elements of second array

while (j < n2)

arr3[k++] = arr2[j++];

// sort the whole array arr3

std::sort(arr3, arr3 + n1 + n2);

return arr3;

}

void merge\_sort(int\* arr, int start, int end)

{

if (end > start) {

int mid = (start + end) / 2;

int left = mid - start + 1;

int right = end - mid;

merge\_sort(arr, start, mid);

merge\_sort(arr, mid + 1, end);

merge(arr + start, left, arr + mid + 1, right);

}

}

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

{

srand(time(0));

int MIN\_NUM = 0;

int MAX\_NUM = 10000;

int SIZE\_ARRAY = 100;

int recvcount = SIZE\_ARRAY;

int ProcRank, ProcNum;

MPI\_Status Status;

MPI\_Init(argc, &argv);

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

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

MPI\_Comm comm1;

MPI\_Comm comm2;

MPI\_Comm comm3;

MPI\_Comm comm4;

MPI\_Comm comm5;

MPI\_Comm comm6;

MPI\_Comm comm7;

MPI\_Group group;

MPI\_Group group1;

MPI\_Group group2;

MPI\_Group group3;

MPI\_Group group4;

MPI\_Group group5;

MPI\_Group group6;

MPI\_Group group7;

MPI\_Comm\_group(MPI\_COMM\_WORLD, &group);

const int ranks[6] = { 1, 2, 3, 4, 5, 6 };

const int ranks1[2] = { 1, 2 };

const int ranks2[2] = { 3, 4 };

const int ranks3[2] = { 5, 6 };

const int ranks4[2] = { 1, 3 };

const int ranks5[2] = { 1, 5 };

const int ranks6[2] = { 0, 1 };

MPI\_Group\_incl(group, 6, ranks, &group1);

MPI\_Group\_incl(group, 2, ranks1, &group2);

MPI\_Group\_incl(group, 2, ranks2, &group3);

MPI\_Group\_incl(group, 2, ranks3, &group4);

MPI\_Group\_incl(group, 2, ranks4, &group5);

MPI\_Group\_incl(group, 2, ranks5, &group6);

MPI\_Group\_incl(group, 2, ranks6, &group7);

MPI\_Comm\_create(MPI\_COMM\_WORLD, group1, &comm1);

MPI\_Comm\_create(MPI\_COMM\_WORLD, group2, &comm2);

MPI\_Comm\_create(MPI\_COMM\_WORLD, group3, &comm3);

MPI\_Comm\_create(MPI\_COMM\_WORLD, group4, &comm4);

MPI\_Comm\_create(MPI\_COMM\_WORLD, group5, &comm5);

MPI\_Comm\_create(MPI\_COMM\_WORLD, group6, &comm6);

MPI\_Comm\_create(MPI\_COMM\_WORLD, group7, &comm7);

int n\_part = SIZE\_ARRAY / (ProcNum - 1)+1;

int r = SIZE\_ARRAY % (ProcNum - 1);

int n\_r = SIZE\_ARRAY + (ProcNum - 1) - r;

int\* data\_array = new int [n\_r];

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

{

data\_array[i] = 0;

}

for (int i = r; i < n\_r; i++)

{

data\_array[i] = generate\_int(MIN\_NUM, MAX\_NUM);

}

if (ProcRank == 0)

{

printf("Unsorted:\n");

for (int i = r; i < n\_r; i++) {

printf("%d ", data\_array[i]);

}

printf("\n");

for (int i = 1; i < ProcNum; i++)

{

// 0 -> 1, 2, 3, 4, 5, 6

MPI\_Send(&n\_part, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD);

}

}

else

{

//1, 2, 3, 4, 5, 6 <- 0

MPI\_Recv(&n\_part, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &Status);

int\* arr\_part = new int[n\_part];

//data -> 1, 2, 3, 4, 5, 6

MPI\_Scatter(data\_array, n\_part, MPI\_INT, arr\_part, n\_part, MPI\_INT, 1, comm1);

if (ProcRank == 1)

{

//1 <- 2

MPI\_Bcast(&recvcount, 1, MPI\_INT, 1, comm2);

int\* recvArr = new int[recvcount];

MPI\_Scatter(arr\_part, n\_part, MPI\_INT, recvArr, recvcount, MPI\_INT, 1, comm2);

arr\_part = merge(arr\_part, n\_part, recvArr, recvcount);

n\_part += recvcount;

//1 <- 3

MPI\_Bcast(&recvcount, 1, MPI\_INT, 1, comm5);

recvArr = new int[recvcount];

MPI\_Scatter(arr\_part, n\_part, MPI\_INT, recvArr, recvcount, MPI\_INT, 1, comm5);

arr\_part = merge(arr\_part, n\_part, recvArr, recvcount);

n\_part += recvcount;

//1 <- 5

MPI\_Bcast(&recvcount, 1, MPI\_INT, 1, comm6);

recvArr = new int[recvcount];

MPI\_Scatter(arr\_part, n\_part, MPI\_INT, recvArr, recvcount, MPI\_INT, 1, comm6);

arr\_part = merge(arr\_part, n\_part, recvArr, recvcount);

n\_part += recvcount;

//1 -> 0

MPI\_Send(&n\_part, 1, MPI\_INT, 0, 0, comm7);

MPI\_Send(arr\_part, n\_part, MPI\_INT, 0, 0, comm7);

}

if (ProcRank == 2)

{

//2 -> 1

MPI\_Bcast(&n\_part, 1, MPI\_INT, 1, comm2);

int\* recvArr = new int [n\_part];

MPI\_Scatter(arr\_part, n\_part, MPI\_INT, recvArr, n\_part, MPI\_INT, 1, comm2);

}

if (ProcRank == 3)

{

// 3 <- 4

MPI\_Bcast(&recvcount, 1, MPI\_INT, 1, comm3);

int\* recvArr = new int[n\_part];

MPI\_Scatter(arr\_part, n\_part, MPI\_INT, recvArr, recvcount, MPI\_INT, 1, comm3);

arr\_part = merge(arr\_part, n\_part, recvArr, recvcount);

n\_part += recvcount;

printf("-----%d-", recvcount);

//3 -> 1

MPI\_Bcast(&n\_part, 1, MPI\_INT, 1, comm5);

recvArr = new int[n\_part];

MPI\_Scatter(arr\_part, n\_part, MPI\_INT, recvArr, n\_part, MPI\_INT, 1, comm5);

}

if (ProcRank == 4)

{

//4 -> 3

MPI\_Bcast(&n\_part, 1, MPI\_INT, 1, comm3);

int\* recvArr = new int[n\_part];

MPI\_Scatter(arr\_part, n\_part, MPI\_INT, recvArr, n\_part, MPI\_INT, 1, comm3);

}

if (ProcRank == 5)

{

//5 <- 6

MPI\_Bcast(&recvcount, 1, MPI\_INT, 1, comm4);

int\* recvArr = new int[recvcount];

MPI\_Scatter(arr\_part, n\_part, MPI\_INT, recvArr, recvcount, MPI\_INT, 1, comm4);

arr\_part = merge(arr\_part, n\_part, recvArr, recvcount);

n\_part += recvcount;

//5 -> 1

MPI\_Bcast(&n\_part, 1, MPI\_INT, 1, comm6);

recvArr = new int[n\_part];

MPI\_Scatter(arr\_part, n\_part, MPI\_INT, recvArr, n\_part, MPI\_INT, 1, comm6);

}

if (ProcRank == 6)

{

//6 -> 5

MPI\_Bcast(&n\_part, 1, MPI\_INT, 1, comm4);

int\* recvArr = new int[n\_part];

MPI\_Scatter(arr\_part, n\_part, MPI\_INT, recvArr, n\_part, MPI\_INT, 1, comm4);

}

}

if (ProcRank == 0)

{

//0 <- 1

MPI\_Recv(&recvcount, 1, MPI\_INT, 1, 0, comm7, &Status);

MPI\_Recv(data\_array, recvcount, MPI\_INT, 1, 0, comm7, &Status);

printf("Sorted:\n");

for (int i = r; i < n\_r; i++) {

printf("%d ", data\_array[i]);

}

printf("\n");

}

MPI\_Finalize();

}