Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение

высшего образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

# КОЛЛЕКТИВНЫЕ ФУНКЦИИ В MPI

Отчёт о лабораторной работе № 2 по дисциплине «Параллельное программирование»

Студент гр. 431-3

\_\_\_\_\_\_\_ Д.П. Андреев

«\_\_» \_\_\_\_\_\_\_\_\_ 2024

Проверил

Доцент каф. АСУ, к.т.н

\_\_\_\_\_\_\_ С.М. Алфёров

«\_\_» \_\_\_\_\_\_\_\_\_ 2024

Томск 2024

**1 Цель лабораторной работы**

Цель:освоить применение коллективных функций MPI для рассылки и сборки фрагментов массивов по процессам и параллельной их обработки по заданному алгоритму.

## 2 Задание

Задание на лабораторную работу (вариант 9): Применяя MPI\_Gather и MPI\_Scatter выполнить индивидуальное задание по работе с матрицами: вычислить поэлементное сложение элементов двух матриц Y = X + Y.

### 3 Использованные MPI функции

В этом коде используется несколько новых MPI функций, а именно MPI\_Scatter и MPI\_Gather. Опишем их подробнее ниже.

1. MPI\_Scatter

Функция MPI\_Scatter используется для равномерного распределения данных от одного ведущего процесса (обычно процесса с рангом 0) ко всем остальным процессам в коммуникаторе. В нашем случае эта функция выполняет рассылку порций исходных матриц X и Y каждому процессу для локальной обработки.

Параметры:

* sendbuf: Указатель на буфер отправляемых данных (в нашем

случае — это массивы X и Y на главном процессе).

* sendcount: Количество элементов, которые отправляются каждому процессу. В нашей программе это portion\_size, то есть число элементов матрицы, которые обрабатываются каждым процессом.
* sendtype: Тип данных, которые отправляются. В нашем случае — это тип MPI\_DOUBLE, так как элементы матриц — числа с плавающей точкой.
* recvbuf: Указатель на буфер, в который процесс будет принимать свою порцию данных. Это локальные массивы local\_X и local\_Y,

которые хранят части матриц для каждого процесса.

* recvcount: Количество элементов, которые принимаются каждым процессом (в данном случае также portion\_size).
* recvtype: Тип принимаемых данных, который совпадает с типом отправляемых данных — MPI\_DOUBLE.
* root: Ранг процесса, который выполняет рассылку данных.

Обычно это процесс с рангом 0.

* comm: Коммуникатор, определяющий группу процессов, между которыми происходит передача данных. В нашей программе используется MPI\_COMM\_WORLD, охватывающий все запущенные

процессы.

2. MPI\_Gather

Функция MPI\_Gather используется для сбора данных с каждого процесса обратно в ведущий процесс (обычно процесс с рангом 0). В нашей программе эта функция собирает обработанные порции матрицы Y, которая была обновлена каждым процессом после поэлементного сложения Y=X+Y.

Параметры:

* sendbuf: Указатель на буфер с данными, которые отправляет каждый процесс. В нашей программе — это локальный массив local\_Y, содержащий обработанную часть матрицы.
* sendcount: Количество элементов, которые отправляет каждый процесс (в нашем случае — portion\_size).
* sendtype: Тип отправляемых данных — MPI\_DOUBLE.
* recvbuf: Указатель на буфер, в который собираются данные на корневом процессе (ранг 0). В нашем случае это массив Y, где будут храниться обновленные элементы матрицы.
* recvcount: Количество элементов, которые принимает главный процесс от каждого процесса. В нашем случае это portion\_size.
* recvtype: Тип данных, которые принимает главный процесс — MPI\_DOUBLE.
* root: Ранг процесса, который собирает данные. В нашей программе это процесс с рангом 0.

comm: Коммуникатор, в рамках которого выполняется сбор данных — MPI\_COMM\_WORLD.

**4 Листинг программы**

Main.cpp:

#include <mpi.h>

#include <math.h>

#include <iostream>

#include <cstdint>

#include <cstdlib>

#include <ctime>

// Вариант 9: вычислить поэлементное сложение элементов двух матриц Y = X + Y

constexpr uint8\_t rows = 5;

constexpr uint8\_t cols = 20;

// Инициализация матрицы случайными числами

void initialize\_matrix(double\* matrix, int rows, int cols) {

for (int i = 0; i < rows \* cols; ++i) { matrix[i] = rand() % 100;

}

}

// Печать матрицы

void print\_matrix(const double\* matrix, int rows, int cols) { for (int i = 0; i < rows; ++i) { for (int j = 0; j < cols; ++j) {

std::cout << matrix[i \* cols + j] << " ";

}

std::cout << std::endl;

}

}

int main(int argc, char\*\* argv) { int numprocs, myid, namelen;

char processor\_name[MPI\_MAX\_PROCESSOR\_NAME];

MPI\_Init(&argc, &argv);

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

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

MPI\_Get\_processor\_name(processor\_name, &namelen);

const int matrix\_size = rows \* cols; // Размер матрицы

double\* X = nullptr; // Матрица X double\* Y = nullptr; // Матрица Y

// Порция матриц для каждого процесса int portion\_size = matrix\_size / numprocs; double\* local\_X = new double[portion\_size];

double\* local\_Y = new double[portion\_size];

if (myid == 0) {

// Инициализация матриц на главном процессе

X = new double[matrix\_size]; Y = new double[matrix\_size]; initialize\_matrix(X, rows, cols);

initialize\_matrix(Y, rows, cols);

std::cout << "Matrix X:" << std::endl; print\_matrix(X, rows, cols);

std::cout << "Matrix Y:" << std::endl; print\_matrix(Y, rows, cols);

}

// Рассылка порций матриц X и Y по процессам

MPI\_Scatter(X, portion\_size, MPI\_DOUBLE, local\_X, portion\_size, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD); MPI\_Scatter(Y, portion\_size, MPI\_DOUBLE, local\_Y, portion\_size, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Поэлементное сложение локальных частей

for (int i = 0; i < portion\_size; ++i) { local\_Y[i] += local\_X[i];

}

// Сбор результатов в главном процессе

MPI\_Gather(local\_Y, portion\_size, MPI\_DOUBLE, Y, portion\_size, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Вывод результата на главном процессе if (myid == 0) {

std::cout << "Resultant matrix Y = X + Y:" << std::endl; print\_matrix(Y, rows, cols);

}

// Освобождение памяти delete[] local\_X; delete[] local\_Y; if (myid == 0) { delete[] X;

delete[] Y;

}

MPI\_Finalize();

return 0;

}

### 5 Примеры работы программы

Разберём работу программы. Пример работы изображён на рисунке 5.1.

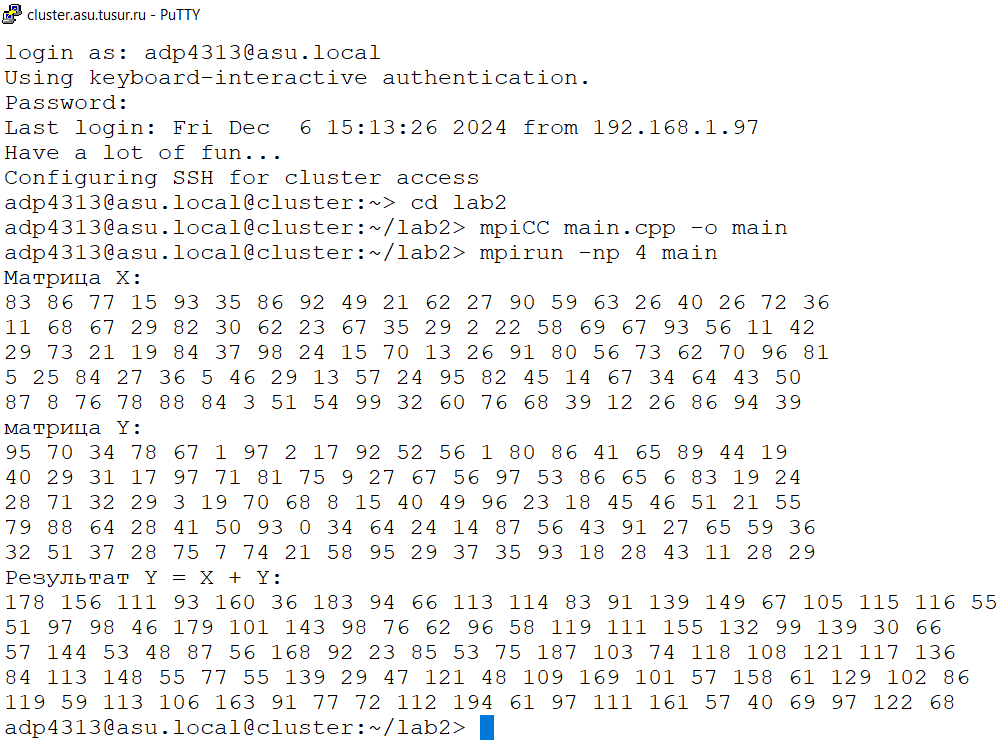


Рисунок 5.1 — Пример работы программы

На рисунке изображён процесс сложения матриц 5 на 20 посредством функций MPI Scatter и Gather.

### 6 Выводы

Таким образом, я изучил основы работы со MPI\_Scatter и MPI\_Gather и реализовал сложение матриц посредством этих функций. Этот метод позволяет вычислять матрицы ещё больших размеров.