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 MPI

ОБРАБОТКА МАССИВОВ.

Отчет по лабораторной работе №2

По дисциплине

«Параллельное программирование»

Студент гр. 431-3

\_\_\_\_\_\_\_\_\_\_\_\_\_ Е.П. Бекиш

(подпись)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(дата)

Руководитель:

Доцент кафедры АСУ

\_\_\_\_\_\_\_\_\_\_\_\_\_ С.М. Алфёров

(подпись)

Томск 2024

**Оглавление**

[Введение 3](#_Toc179550399)

[Ход работы 4](#_Toc179550400)

[Заключение 8](#_Toc179550401)

[Приложение А 9](#_Toc179550402)

# Введение

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

Индивидуальное задание по варианту №:23: переписать элементы каждой строки матрицы в обратном порядке.

# Ход работы

1. Для предложенного алгоритма составить и отладить последовательную программу обработки числовых массивов индивидуального задания. Использовать динамическое выделение памяти для массивов. Входные массивы заполнить случайными числами. Алгоритм обработки оформить внешней функцией.

Задача заключается в том, чтобы отразить позиции элементов по строкам. Для этого используется функция из стандартной библиотеки std::swap. Сам массив выделяется динамическим способом с помощью оператора new. Входной массив был заполнен рандомными числами посредством алгоритма «Вихрь Мерсенна».

2 Для параллельной обработки определить размер порции массива для каждого процесса и смещение порции от начала полного массива.

Так как в задаче фигурируют матрицы, то разделение задач будет осуществляться по строкам. Для этого вычисляется целое количество обрабатываемых строк одним процессом count и остаточное количество обрабатываемых строк rest, распределяемое по процессам. Далее в root процессе ранее инициализированные массивы rcounts и displs заполняются, в соответствии с распределяемыми буферами. Массив rcounts содержит длины отправляемых буферов данных, тогда как массив displs содержит индексы начала частей буфера.

3 В каждом процессе выделить память для размещения порции массива. Функцией MPI\_Scatter или MPI\_Scatterv распределить исходный массив(ы) на число процессов, выбранных при запуске программы.

Далее выделяется буфер памяти получаемого массива данных и посредством функции MPI\_Scatterv процесс root рассылает соответствующие части массива всем процессам коммуникатора.

5 В каждом процессе выполнить обработку части массива составленной внешней функцией и разместить результаты в массиве порции или в выходных переменных.

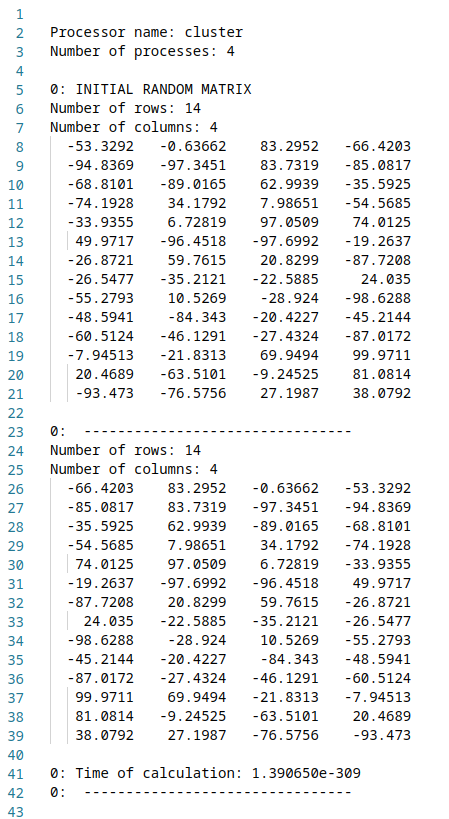
После получения части массива всеми процессами посредством той же самой функции MPI\_Scatterv, вызывается метод обработки массива reflect, выполняющий индивидуальное задание.

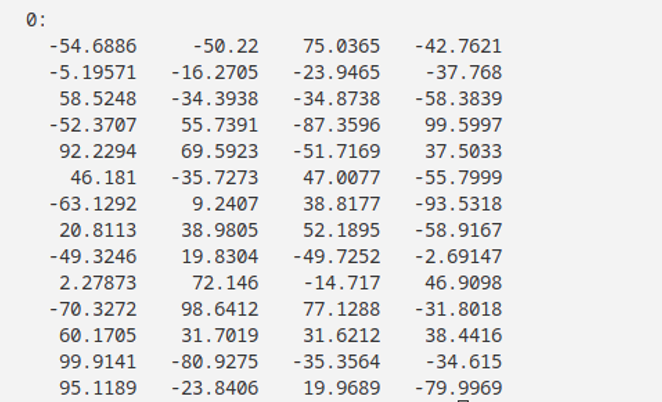
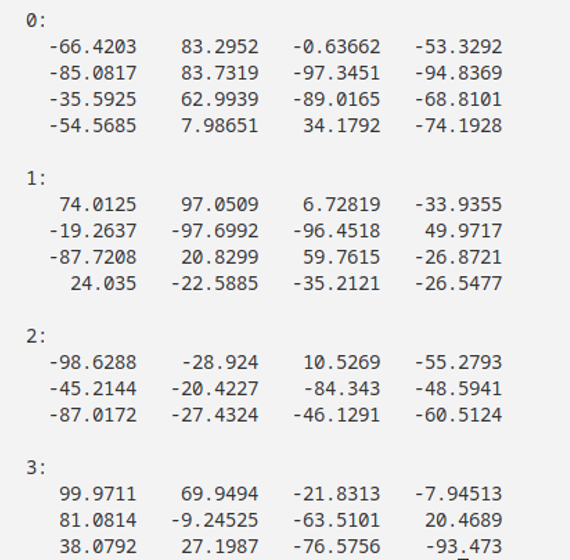
6 Собрать в главном процессе окончательные результаты (функции MPI\_Gather(v) или MPI\_Reduce.

Обработав массив, вызывается функция MPI\_Gatherv, которая на всех процессах посылает root процессу свой буфер данных, который объединяет части буферов в единый буфер.

7 Вывести окончательные результаты.

Окончательные результаты работы параллельной программы выводятся в зависимости от переданных параметров, задающих количество строк и столбцов матрицы. Вывод осуществляется в файл output.txt, содержимое которого можно посмотреть на рисунке 1.1. Листинг программы представлен в приложении А.1. Результаты работы функции отображения для каждого процесса представлены на рисунке 1.2.

Рисунок 1.1 — Результат работы программы, выполненной на 4 процессах

Рисунок 1.2 — Результат работы функции отражения для каждого процесса

# Заключение

В результате выполнения лабораторной работы я освоил применение основных функций MPI на примере параллельной программы численного интегрирования.

# Приложение А

(обязательное)

**Листинг программы**

Листинг А.1

#pragma once

#include "mpi.h"

class Init\_Process {

protected:

MPI\_Comm \_comm;

int numprocs;

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

int namelen;

protected:

Init\_Process(int argc, char\* argv[], MPI\_Comm comm = MPI\_COMM\_WORLD)

{

MPI\_Init(&argc, &argv);

\_comm = comm;

MPI\_Comm\_size(\_comm, &numprocs);

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

}

~Init\_Process() {

MPI\_Finalize();

}

public:

virtual void printInfo(std::string accompanying\_message = "", std::ostream& out = std::cout)

{

out << "\nИмя процесса: " << processor\_name

<< "\nНомер процесса: " << numprocs

<< "\n" << accompanying\_message;

fflush(NULL);

}

};

#pragma once

#include "mpi.h"

#include "init\_process.h"

class Process : public Init\_Process

{

protected:

int process\_ID;

enum PIDs

{

INIT

};

public:

Process(int argc, char\* argv[], MPI\_Comm comm = MPI\_COMM\_WORLD) : Init\_Process(argc, argv, comm) {

MPI\_Comm\_rank(comm, &process\_ID);

}

int get\_process\_ID() const {

return process\_ID;

}

void printInfo(std::string accompanying\_message = "", std::ostream& out = std::cout) {

out << "\n"

<< process\_ID << ": " << accompanying\_message;

fflush(NULL);

}

void send(void\* buffer, int to\_process\_ID, MPI\_Datatype type = MPI\_INT, int count = 1, int tag = 1) {

MPI\_Send(buffer, count, type, to\_process\_ID, tag, \_comm);

}

MPI\_Status recv(void\* buffer, int from\_process\_ID, MPI\_Datatype type = MPI\_INT, int count = 1, int tag = 1) {

MPI\_Status status;

MPI\_Recv(buffer, count, type, from\_process\_ID, tag, \_comm, &status);

return status;

}

void bcast(void\* buffer, MPI\_Datatype type = MPI\_INT, int count = 1, int root = Process::INIT) {

MPI\_Bcast(buffer, count, type, root, \_comm);

}

void reduce(void\* send\_buffer, void\* recv\_buffer, MPI\_Datatype type = MPI\_INT, int root = Process::INIT, MPI\_Op op = MPI\_SUM, int count = 1) {

MPI\_Reduce(send\_buffer, recv\_buffer, count, type, op, root, \_comm);

}

void pack(void\* in\_buffer, int count, void\* out\_buffer, int count\_buffer, MPI\_Datatype type = MPI\_INT, int\* position = NULL) {

MPI\_Pack(in\_buffer, count, type, out\_buffer, count\_buffer, position, \_comm);

}

void unpack(void\* in\_buffer, int insize, void\* out\_buffer, int\* position, MPI\_Datatype type = MPI\_INT, int count = 1) {

MPI\_Unpack(in\_buffer, insize, position, out\_buffer, count, type, \_comm);

}

};

#pragma once

#include "Process.h"

#include <fstream>

#include <iomanip>

#include <iostream>

#include <random>

#include <algorithm>

#include <iterator>

#include <vector>

template<typename T>

class Matrix: public Process

{

private:

int rows, columns;

T\* data = NULL;

double startwtime, endwtime;

std::ofstream fout;

public:

Matrix(int argc, char \*argv[],

int \_rows=5, int \_columns=5,

MPI\_Comm comm = MPI\_COMM\_WORLD,

std::string filename = "output.txt")

: Process(argc, argv, comm)

{

fout.open(filename, std::ios::out);

rows = \_rows;

columns = \_columns;

if (PID==Process::INIT)

{

Communicator::printInfo("", fout);

data = new T[rows \* columns];

fillRandom();

Process::printInfo("INITIAL RANDOM MATRIX", fout);

fout << \*this;

Process::printInfo("\t--------------------------------", fout);

}

startwtime=MPI\_Wtime();

}

~Matrix() { fout.close(); delete data;}

void fillRandom(T min=-100.0, T max=100.0)

{

std::random\_device rnd\_device;

std::mt19937 mersenne\_engine {rnd\_device()};

std::uniform\_real\_distribution<T> dist {min, max};

auto gen = [&dist, &mersenne\_engine]()

{return dist(mersenne\_engine);};

std::generate(data, data + rows \* columns, gen);

}

private:

static T\* reflect(T\* array, int len, int cols)

{

for(int i = 0; i < len / cols; ++i)

for(int j = 0; j < cols/2; ++j)

std::swap(array[i\*cols + j],

array[i\*cols + (cols -1-j)]);

return array;

}

public:

void scatterVec()

{

int count = rows / numprocs;

int rest = rows % numprocs;

int \*displs = new int[numprocs],

\*rcounts = new int[numprocs];

if (PID==Process::INIT)

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

{

rcounts[i] = i < rest ? columns \* (count+1) : columns\*count;

displs[i] = displs[i-1] + rcounts[i-1];

}

int length = PID < rest ? columns \* (count+1) : columns\*count;

int startIndex = PID \* length + (PID >= rest ? rest : 0);

T \*partOfArray = new T[length];

MPI\_Scatterv(data, rcounts, displs, MPI\_FLOAT,

partOfArray, length, MPI\_FLOAT, Process::INIT, comm);

reflect(partOfArray, length, columns);

MPI\_Gatherv(partOfArray, length, MPI\_FLOAT,

data, rcounts, displs, MPI\_FLOAT, Process::INIT, comm);

delete rcounts, displs, partOfArray;

if (PID == Process::INIT)

{

fout << \*this;

std::stringstream str;

str << std::scientific << "Time of calculation: " << endwtime - startwtime;

Process::printInfo(str.str(), fout);

Process::printInfo("\t--------------------------------", fout);

fflush(NULL);

str.clear();

str.str("");

}

}

friend std::ostream& operator<<(std::ostream& out, const Matrix<T>& matrix)

{

out << "\nNumber of rows: " << matrix.rows

<< "\nNumber of columns: " << matrix.columns

<< "\n";

for (int i = 0; i < matrix.rows; ++i)

{

for (int j = 0; j < matrix.columns; ++j)

out << std::setw(10)<<matrix.data[i \* matrix.columns + j] << " ";

out << "\n";

}

fflush(NULL);

return out;

}

};