Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Нижегородский государственный университет им. Н.И. Лобачевского»

**Институт информационных технологий, математики и механики**

**Отчет**

по лабораторной работе №3

**«Быстрая сортировка с простым слиянием»**

**Выполнил:**

студент группы 381606-1

Муравьев Д.А.

**Проверил:**

Кустикова В.Д.

Нижний Новгород

2018

**Содержание**

[Постановка задачи 3](#_Toc531367741)

[Метод решения 4](#_Toc531367742)

[Схема распараллеливания 6](#_Toc531367743)

[Описание программной реализации 8](#_Toc531367744)

[Подтверждение корректности 11](#_Toc531367745)

[Результаты экспериментов 12](#_Toc531367746)

[Заключение 14](#_Toc531367747)

[Приложение 15](#_Toc531367748)

# Постановка задачи

Сортировка массива хороший пример задачи, которую можно решать с помощью многих различных алгоритмов. Каждый из них имеет и свои достоинства, и свои недостатки, и выбирать алгоритмы нужно исходя из конкретной постановки задачи.

В общем, сортировку следует понимать как процесс перегруппировки заданного множества объектов в некотором определенном порядке. Цель сортировки – облегчить последующий поиск элементов в таком отсортированном множестве. Это почти универсальная, фундаментальная деятельность. Мы встречаемся с отсортированными объектами в телефонных книгах, в списках подоходных налогов, в оглавлениях книг, в библиотеках, в словарях, на складах – почти везде.

Наша задача формулируется следующим образом: требуется разработать программу, сортирующую массив элементов с помощью быстрой сортировки с использованием методов параллельного программирования. Для этого нужно разработать последовательный и параллельный алгоритмы данной сортировки. Затем сравнить время выполнения на различном числе процессов и определить ускорение.

Необходимо выполнить следующие задачи:

1. Изучить алгоритм быстрой сортировки
2. Разработать схему распределения данных между процессами
3. Придумать схему слияния данных
4. Реализовать эту программу посредством библиотеки MPI
5. Проверить корректность работы программы
6. Провести серию экспериментов с различным числом процессов и объема данных.
7. Проанализировать результаты экспериментов
8. Вычислить ускорение, полученное путем распараллеливания

# Метод решения

Изучим алгоритм быстрой сортировки. Псевдокод представлен далее:

* Функция принимает на вход массив, левую и правую границу массива.
* Объявляем 3 переменные целого типа i равное левой границе массива, j равное правой границе массива, «опорный» элемент, который равен элементу массива под номером х, где х равно среднему арифметическому от левой и правой границы.
* ЦИКЛ ПОКА i меньше либо равно j ТО
  + ЦИКЛ ПОКА элемент под номером i меньше «опорного» ТО
    - К i прибавляем 1.
  + ЦИКЛ ПОКА элемент под номером j больше «опорного» ТО
    - Из j вычитаем 1.
  + УСЛОВИЕ ЕСЛИ i меньше либо равно j, ТО
    - УСЛОВИЕ ЕСЛИ i меньше j, ТО
      * Меняем местами элементы с номерами i и j.
    - К i прибавляем 1.
    - Из j вычитаем 1.
* УСЛОВИЕ ЕСЛИ j больше левой границы, ТО
  + Вызываем функцию сортировки с параметрами: массив, левая граница, j.
* УСЛОВИЕ ЕСЛИ i меньше правой границы, ТО
  + Вызываем функцию сортировки с параметрами: массив, i, правая граница.

Как видно, алгоритм не зависит от данных всех процессов. То есть каждый процесс может отсортировать свой подмассив и вернуть его в главный процесс. Но и этап слияния можно распараллелить. Так как сливать результаты работы всех процессов в один не рационально, то будем сливать по два небольших подмассива, сокращая каждый раз, количество работающий процессов в 2 раза, пока у нас не сольются в главном процессе две отсортированные половины изначального массива. Получится бинарное дерево работы процессов.

Для организации распределения работы между процессами понадобится массив реверсов. Это массив, в ячейках которого хранится перевернутое двоичное представление номера этой самой ячейки.

Алгоритм получения массива реверсов:

* Создаем массив размером 2k элементов.
* В каждую ячейку кладем ее же номер в двоичной системе счисления.
* Переворачиваем двоичное представление каждого значения и сохраняем результат.
* Переводим обратно в десятичную систему счисления.

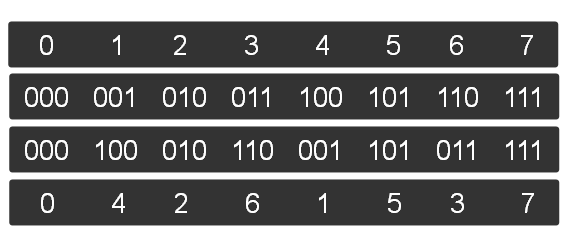


Рис.1 Схема получения массива реверсов для 8 элементов

Свойства массива реверсов, которые нужны для распараллеливания, будут описаны в следующем разделе.

# Схема распараллеливания

Рассмотрим для начала случай, когда число процессов равняется 2k.

Строим массив реверсов и выполняем распределение частей сортируемого массива, а именно вычисляем точку начала копируемого подмассива и его длину, так что бы для каждого процесса было выделено примерно одинаковое количество данных.

Отправляем каждому процессу выделенный подмассив для сортировки. Первая часть вектора отправляется 0-му процессу, вторая часть 4-му, третья 2-му, четвертая часть 6-му процессу и т.д. Каждый процесс сортирует свою порцию данных. Слияние отсортированных данных происходит по следующей схеме:

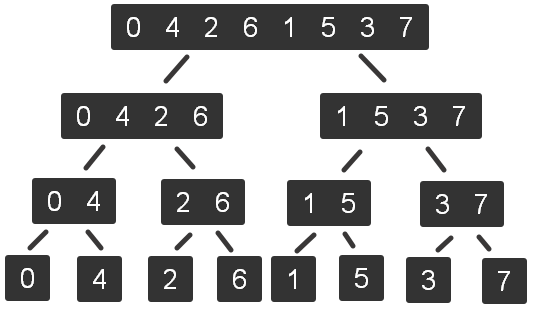


Рис.2 Схема слияния данных для 8 процессов

Если число процессов не равно 2k, то некоторые процессы будут выполнять двойную норму. Например, если у нас 7 процессов, то 3-й процесс вынужден выполнить сортировку своих данных и данных процесса под номером 7.

Параметр k отвечает за максимальный уровень погружения. В данном примере у нас 4 уровня погружения (от нулевого до третьего). На k уровне происходит сортировка, а на последующих происходит слияние данных. На них выполняем операцию:

**(Номер процесса) XOR (1 << i)**

Где i это текущий уровень погружения. Если на i-ой позиции двоичного представления числа есть единица, то получаем номер процесса, куда нужно посылать данные. Если этот бит равен нулю, то получаем номер процесса, откуда будем принимать данные. В данном примере на 2-м уровне у нас останутся работать 0,1,2,3 процессы, на 1-м уровне 0 и 1 процессы, а на 0-м уровне останется только 0-й процесс.

Преимущество данной схемы распараллеливания в легком нахождении соседних процессов для передачи информации и в не перемешивании данных. То есть если подать на вход отсортированный массив, то мы просто пройдемся по подмассивам в сортировках и сольем их в том же порядке, в котором они были распределены. Все элементы массива останутся на тех же позициях, на которых были в изначальном массиве.

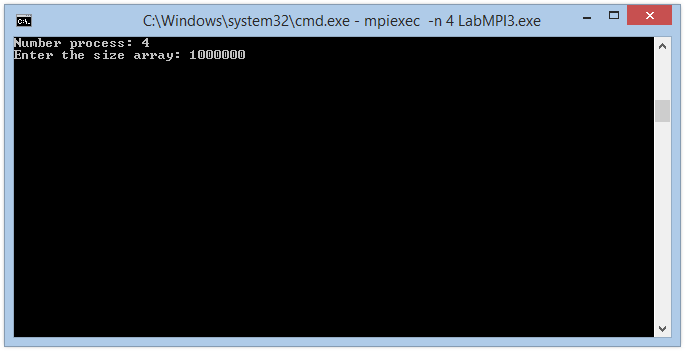
# Описание программной реализации

**Руководство пользователя**

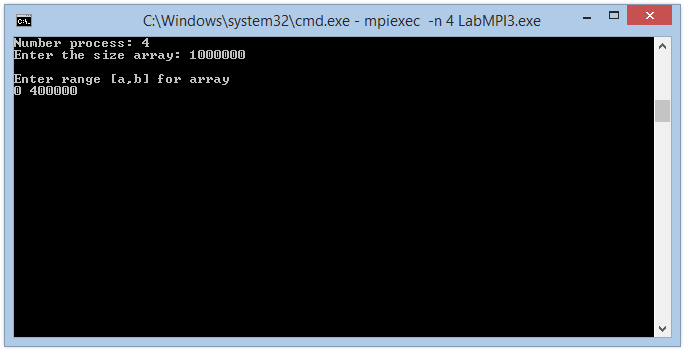
Запуск программы происходит с помощью следующего запроса:

*mpiexec –n 4 LabMPI3.exe*

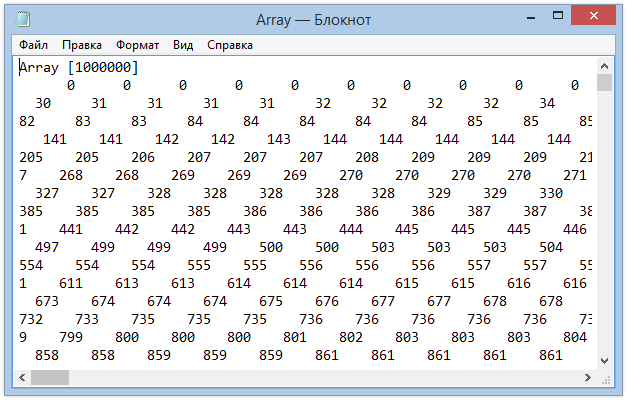
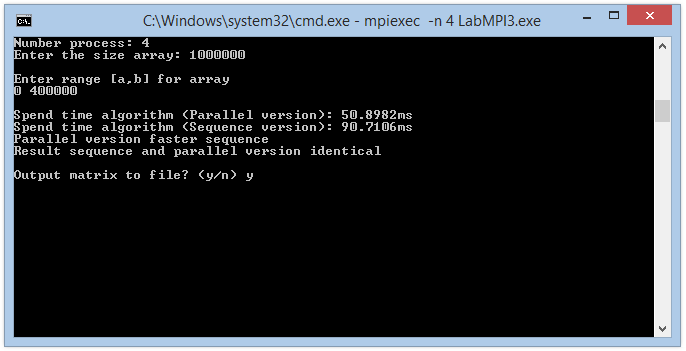
Где 4 это количество запускаемых процессов. Можно задать и другое натуральное число. Далее вводим размер массива.



Числа для массива будут генерироваться из определенного диапазона. Вводим его.



Анализируем результаты работы программы и записываем отсортированный массив в .txt файл.



**Руководство программиста**

Код программы можно просмотреть в разделе «[Приложение](#_Приложение)».

Для удобства, часть кода вынесена в несколько функций:

double\* Create\_vec(int \_size)// генерация вектора

void Show\_vec(double\* \_vec, int \_size) // вывод массива в поток и в файл

int Equality\_test(double\* \_mas\_seque, double\* \_mas\_paral, int \_size)// проверка идентичности результатов

void Quick\_Sort(double \* m, int l, int r)// алгоритм быстрой сортировки

void Merg(double\* mas, double\* tmp\_mas, int l1, int r1, int l2, int r2)// слияние двух отсортированных массивов

int\* Reverse\_notation(int \_n, int \_k)// n=2^k // создание массива реверсов

int\* Create\_list\_work(int \_size, int \_n, int \_k)// нахождение точек начала подмассивов

int\* Calculation\_length\_mas(int\* \_sendcounts, int \_k, int \_proc\_num)// вычисление размера буфера подмассивов

Код главной функции можно разделить на несколько этапов:

1. Вызов функций **MPI\_Init(), MPI\_Comm\_size(), MPI\_Comm\_rank()**.
2. Генерация в главном процессе массива данных и подсчет размеров передаваемых подмассивов и их расположение в массиве.
3. Вызов функций **MPI\_Scatter(), MPI\_Scatterv()** для передачи подмассивов, их размеров и размеров выделяемого буфера, для хранения своих данных и тех, что приняли посредством слияния.
4. Вызов функции быстрой сортировки для подмассива на каждом процессе
5. Вычисление **proc\_rank ^ (1 << i)** для передачи (приема) данных для (от) соседних процессов в зависимости от текущего уровня погружения (i). Применяемые функции: **MPI\_Send(),MPI\_Recv()**, а так же функции слияния.
6. Запуск в главном процессе последовательного алгоритма
7. Подсчет времени работы обоих алгоритмов и проверка идентичности результатов.
8. Освобождение выделенной памяти.
9. Вызов **MPI\_Finalize().**

# Подтверждение корректности

Для подтверждения корректности в программе реализована функция, которая проверяет 2 массива. Первый массив из элементов, отсортированных последовательным алгоритмом, а второй параллельным алгоритмом. Поскольку последовательный алгоритм всегда дает правильный результат, то мы проверяем соответствующие элементы 2-х массивов. Если у нас есть хотя бы одна пара значений, которые не совпадают, то функция возвращает отрицательный результат. Это означает, что алгоритм параллельной версии неправильно отсортировал массив. Если функция не смогла найти различий в этих 2-х массивах, то наш алгоритм отработал правильно.

# Результаты экспериментов

По данным экспериментов видно, что ускорение происходит на массивах длиной более 50000 элементов. Далее представлены графики зависимости времени от длины массива.





При использовании 2-х процессов коэффициент ускорения равен **1,5**. При 4-х процессах ускорение равнялось **2**. Ускорения, в 2 и 4 раза соответственно, не происходило из-за многих тормозящих факторов. Например, передача / прием данных между процессами, вычисление размеров подмассивов, их буферов, выделение памяти, слияние данных и т.д.

# Заключение

Был реализован алгоритм быстрой сортировки с простым слиянием. Разработана схема распределения работы и слияния данных. В результате серии экспериментов было обнаружено ускорение работы программы. Это означает, что алгоритм быстрой сортировки можно распараллелить и получить при этом ускорение.

# Приложение

#include <mpi.h> // Задача 3 №11 Быстрая сортировка с простым слиянием

#include <ctime>

#include <iostream>

#include <fstream>

using namespace std;

#define Main\_Process 0

#define Port\_Merge 2

#define Port\_size\_vec 4

double\* Create\_vec(int \_size) {// генерация вектора

if (\_size < 1)

return NULL;

double a = 0, b = 0;

cout << endl << "Enter range [a,b] for array" << endl;

cin >> a >> b;

double\* \_matr\_as\_vec = new double[\_size];

for (int i = 0; i < \_size; i++)

\_matr\_as\_vec[i] = (long long)(rand() \* rand()) % (long long)(b - a + 1) + a;

return \_matr\_as\_vec;

}

void Show\_vec(double\* \_vec, int \_size) {// вывод массива в поток и в файл

if (\_vec == NULL || \_size < 1)

return;

int delta = 7;

if (\_size < 20) {

cout << "Array [" << \_size << "]" << endl;

for (int i = 0; i < \_size; i++) {

cout.width(delta);

cout << \_vec[i];

}

}

cout << endl << "Output matrix to file? (y/n) ";

char answer = 'n';

cin >> answer;

if (answer == 'y') {

ofstream strm("Array.txt");

strm << "Array [" << \_size << "]" << endl;

for (int i = 0; i < \_size; i++) {

strm.width(delta);

strm << \_vec[i];

}

strm.close();

}

}

int Equality\_test(double\* \_mas\_seque, double\* \_mas\_paral, int \_size) {// проверка идентичности результатов

for (int i = 0; i < \_size; i++)

if (\_mas\_seque[i] != \_mas\_paral[i])

return 0;

return 1;

}

void Merg(double\* mas, double\* tmp\_mas, int l1, int r1, int l2, int r2){

int i = l1, j = l2, k = l1;

while ((i <= r1) && (j <= r2)) {

if (mas[i] < mas[j])

tmp\_mas[k++] = mas[i++];

else

tmp\_mas[k++] = mas[j++];

}

if (i > r1)

for (j; j <= r2; j++)

tmp\_mas[k++] = mas[j];

else

for (i; i <= r1; i++)

tmp\_mas[k++] = mas[i];

for (i = l1; i <= r2; i++)

mas[i] = tmp\_mas[i];

}

void Quick\_Sort(double \* m, int l, int r)

{

int i = l, j = r, e, t;

e = m[(r + l) / 2];

while (i <= j) {

while (m[i] < e)

i++;

while (m[j] > e)

j--;

if (i <= j) {

if (i < j) {

t = m[i];

m[i] = m[j];

m[j] = t;

}

i++;

j--;

}

}

if (j > l)

Quick\_Sort(m, l, j);

if (r > i)

Quick\_Sort(m, i, r);

}

int\* Reverse\_notation(int \_n, int \_k) { // n=2^k

int higher = -1;

int\* \_rev = new int[\_n];

\_rev[0] = 0;

for (int i = 1; i < \_n; i++) {

if ((i & (i - 1)) == 0)

higher++;

\_rev[i] = \_rev[i ^ (1 << higher)];

\_rev[i] |= 1 << (\_k - higher - 1);

}

return \_rev;

}

void Dive(int \*\_border, int id, int l, int r, int num\_dive, int max\_dive) {

if (num\_dive == max\_dive) {

\_border[id] = l;

return;

}

num\_dive++;

int l1, r1;

l1 = (l + r) / 2;

r1 = l1 + 1;

Dive(\_border, id, l, l1, num\_dive, max\_dive);

Dive(\_border, id + (1 << (max\_dive - num\_dive)), r1, r, num\_dive, max\_dive);

}

int\* Create\_list\_work(int \_size, int \_n, int \_k) { // n=2^k

int\* \_border = new int[\_n];

Dive(\_border, 0, 0, \_size - 1, 0, \_k);

return \_border;

}

int\* Calculation\_length\_mas(int\* \_sendcounts, int \_k, int \_proc\_num) {

int\* \_size = new int[\_proc\_num];

for (int i = 0; i < \_proc\_num; i++)

\_size[i] = \_sendcounts[i];

int mask;

for (\_k--; \_k >= 0; \_k--) {

mask = 1 << \_k;

for (int i = 0; i < mask; i++) {

if ((i | mask) < \_proc\_num)

\_size[i] += \_size[i | mask];

}

}

return \_size;

}

int main(int argc, char \* argv[]) {

int proc\_num; // число процессов

int proc\_rank; // номер текущего процесса

double\* mas = NULL;// массив

int size = 0; // длина массива

double\* mas\_seque = NULL; // отсортированный последовательной версии алгоритма

double\* mas\_paral = NULL; // отсортированный параллельной версии алгоритма

double time\_seque = 0, time\_paral = 0; // время работы последовательной и параллельной версии алгоритма

double time\_start\_seque = 0, time\_start\_paral = 0; // время начала работы -//-

double time\_end\_seque = 0, time\_end\_paral = 0; // время конца работы -//-

int\* displs = NULL; // массив смещений в векторе для начальной передачи данных

int\* sendcounts = NULL; // массив длин подвекторов для начальной передачи данных

int\* size\_buf = NULL; // массив длин подвекторов для работы сортировки

int n = 1, k = 0; // n=2^k // параметры бинарного деления массива

double\* part\_mas = NULL; // массив мортируемый каждым процессом

double\* tmp\_mas = NULL; // побочный массив для слияния массивов

int part\_size\_vec, part\_size\_buf; // размер вектора и его заполненость

int err\_code = 0;

MPI\_Status stat; // структура атрибутов сообщений для "общения" процессов

err\_code = MPI\_Init(&argc, &argv); // передача всем процессам аргументов командной строки

if (err\_code != MPI\_SUCCESS) {// проверка на успешность инициализации процессов

cout << "Error code MPI\_Init: " << err\_code;

return -1;

}

MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num); // функция определения числа процессов

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &proc\_rank); // функция определения номера текущего процесса

if (proc\_rank == Main\_Process) {// код главного процесса

cout << endl << "Number process: " << proc\_num << endl;

cout << "Enter the size array: ";

cin >> size;

// генерация матрицы, вектора, а также представление матрицы в виде вектора

srand((unsigned)time(NULL));

mas = Create\_vec(size);

if (mas == NULL) {

cout << "Error! Incorrect input data for array";

MPI\_Finalize();

return -1;

}

time\_start\_paral = MPI\_Wtime();

while (n < proc\_num) {

n = n << 1;

k++;

}

int \*rev = Reverse\_notation(n, k);

int \*border = Create\_list\_work(size, n, k);

displs = new int[proc\_num];

sendcounts = new int[proc\_num];

for (int i = 0; i < proc\_num; i++)

displs[i] = border[rev[i]];

int \*tmp = new int[n];

for (int i = 0; i < n - 1; i++)

tmp[i] = border[i + 1] - border[i];

tmp[n - 1] = size - border[n - 1];

for (int i = 0; i < n; i++) {

if (rev[i] < proc\_num)

sendcounts[rev[i]] = tmp[i];

else

sendcounts[rev[i - 1]] += tmp[i];

}

size\_buf = Calculation\_length\_mas(sendcounts, k, proc\_num);

delete[] tmp;

delete[] rev;

delete[] border;

}

// НАЧАЛО ПАРАЛЛЕЛЬНОГО АЛГОРИТМА

// отправка каждому процессу массива данных и их кол-во

MPI\_Bcast(&k, 1, MPI\_INT, Main\_Process, MPI\_COMM\_WORLD);

MPI\_Scatter(sendcounts, 1, MPI\_INT, &part\_size\_vec, 1, MPI\_INT, Main\_Process, MPI\_COMM\_WORLD);

MPI\_Scatter(size\_buf, 1, MPI\_INT, &part\_size\_buf, 1, MPI\_INT, Main\_Process, MPI\_COMM\_WORLD);

part\_mas = new double[part\_size\_buf];

tmp\_mas = new double[part\_size\_buf];

MPI\_Scatterv(mas, sendcounts, displs, MPI\_DOUBLE, part\_mas, part\_size\_vec, MPI\_DOUBLE, Main\_Process, MPI\_COMM\_WORLD);

Quick\_Sort(part\_mas, 0, part\_size\_vec - 1);

int max\_rank = 1 << (k - 1);

int source = proc\_rank ^ max\_rank;

if (k > 0) {

if (proc\_rank < max\_rank) {

if (source < proc\_num) {

int add\_size;

MPI\_Recv(&add\_size, 1, MPI\_INT, source, Port\_size\_vec, MPI\_COMM\_WORLD, &stat);

MPI\_Recv(part\_mas + part\_size\_vec, add\_size, MPI\_DOUBLE, source, Port\_Merge, MPI\_COMM\_WORLD, &stat);

Merg(part\_mas, tmp\_mas, 0, part\_size\_vec - 1, part\_size\_vec, part\_size\_vec + add\_size - 1);

part\_size\_vec += add\_size;

}

}

else {

MPI\_Send(&part\_size\_vec, 1, MPI\_INT, source, Port\_size\_vec, MPI\_COMM\_WORLD);

MPI\_Send(part\_mas, part\_size\_vec, MPI\_DOUBLE, source, Port\_Merge, MPI\_COMM\_WORLD);

}

}

for (int i = k - 2; (i >= 0) && (proc\_rank < max\_rank); i--) {

source = proc\_rank ^ (1 << i);

max\_rank = max\_rank >> 1;

if (proc\_rank < max\_rank) {

int add\_size;

MPI\_Recv(&add\_size, 1, MPI\_INT, source, Port\_size\_vec, MPI\_COMM\_WORLD, &stat);

MPI\_Recv(part\_mas + part\_size\_vec, add\_size, MPI\_DOUBLE, source, Port\_Merge, MPI\_COMM\_WORLD, &stat);

Merg(part\_mas, tmp\_mas, 0, part\_size\_vec - 1, part\_size\_vec, part\_size\_vec + add\_size - 1);

part\_size\_vec += add\_size;

}

else {

MPI\_Send(&part\_size\_vec, 1, MPI\_INT, source, Port\_size\_vec, MPI\_COMM\_WORLD);

MPI\_Send(part\_mas, part\_size\_vec, MPI\_DOUBLE, source, Port\_Merge, MPI\_COMM\_WORLD);

}

}

// КОНЕЦ ПАРАЛЛЕЛЬНОГО АЛГОРИТМА

if (proc\_rank == Main\_Process) {

time\_end\_paral = MPI\_Wtime();

time\_paral = 1000 \* (time\_end\_paral - time\_start\_paral); // посчет времени работы алгоритма в миллисекудах

cout << endl << "Spend time algorithm (Parallel version): " << time\_paral << "ms";

// НАЧАЛО ПОСЛЕДОВАТЕЛЬНОГО АЛГОРИТМА

time\_start\_seque = MPI\_Wtime();

Quick\_Sort(mas, 0, size - 1);

time\_end\_seque = MPI\_Wtime();

time\_seque = 1000 \* (time\_end\_seque - time\_start\_seque); // посчет времени работы алгоритма в миллисекудах

cout << endl << "Spend time algorithm (Sequence version): " << time\_seque << "ms" << endl;

// КОНЕЦ ПОСЛЕДОВАТЕЛЬНОГО АЛГОРИТМА

mas\_paral = part\_mas;

mas\_seque = mas;

// вывод результатов работы алгоритмов

if (time\_seque < time\_paral)

cout << "Sequence version faster parallel" << endl;

else

cout << "Parallel version faster sequence" << endl;

if (Equality\_test(mas\_seque, mas\_paral, size))

cout << "Result sequence and parallel version identical" << endl;

else

cout << "Result sequence and parallel version not identical" << endl;

// вывод в поток матрицы небольших размеров + вывод мартицы в файл

Show\_vec(mas, size);

// очистка памяти

delete[] mas;

delete[] displs;

delete[] sendcounts;

delete[] size\_buf;

}

delete[] part\_mas;

delete[] tmp\_mas;

MPI\_Finalize();// уничтожение всех MPI процессов и их связей

return 0;

}