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

**«**Поразрядная сортировка для вещественных чисел (тип double) с простым слиянием**»**

**Выполнил:**

студент группы 381608

Ермаченко Б.А.

**Проверил:**

Кустикова В.Д.

Нижний Новгород

2018

**Содержание**

[Постановка задачи 2](#_Toc531367741)

[Метод решения 3](#_Toc531367742)

[Схема распараллеливания 4](#_Toc531367743)

[Описание программной реализации 5](#_Toc531367744)

[Подтверждение корректности 6](#_Toc531367745)

[Результаты экспериментов 7](#_Toc531367746)

[Заключение 8](#_Toc531367747)

[Приложение 9](#_Toc531367748)

# Постановка задачи

В рамках данной лабораторной работы необходимо разработать программу, использующую поразрядную сортировку вещественных чисел и простое слияние для сортировки массива. Требуется реализовать как параллельную, так и линейную версию программы. Массив генерируется в начале работы программы на нулевом процессе. По завершении выполнения программа выводит на консоль время работы линейной и параллельной версий программы в секундах (в линейной версии программы используется только поразрядная сортировка).

# Метод решения

Каждый процесс сортирует полученную часть исходного массива с помощью поразрядной сортировки.

Затем каждый процесс отправляет свою отсортированную часть массива всем остальным процессам (себе в том числе). Далее циклично происходит слияние. Сначала слияние происходит на половине процессов, затем на четверти и т.д. до того момента, пока не получим две отсортированные части, которые отсортируем слиянием на нулевом процессе

В результате работы на нулевом процессе оказывается отсортированный массив. (Его уже нет смысла отсылать всем)

Так как каждая сортировка слиянием требует выделения памяти на дополнительный массив, будем выделять массив максимальной необходимой длины для каждого процесса в самом начале работы программы, это уменьшит время работы программы.

# Схема распараллеливания

Нулевой процесс поровну разделяет массив между всеми процессами (остаток равномерно распределяется между процессами). Затем каждый процесс линейно сортирует полученную часть массива с помощью поразрядной сортировки. Далее данные собираются на всех процессах, после чего, фактически по схеме биномиального дерева, где корнем является нулевой процесс, происходит простое слияние. После каждой итерации значения массива отправляются остальным процессам, количество процессов, участвующих в итерации сокращается вдвое

Для написания параллельной программы используется библиотека MS-MPI.

# Описание программной реализации

**Руководство пользователя**

Для запуска программы необходимо выполнить в консоли следующую команду:

path/to/mpiexec.exe –n PNum "pathto/program.exe", где PNum – количество процессов

Далее программа запросит размер массива

После того, как Вы ввели размер массива необходимо нажать Enter

В результате работы программы выводится время выполнения линейной и параллельной версий программы в секундах.

**Руководство программиста**

Программа реализована в одном модуле - main

Функции:

show\_array -вывод массива

MergeSort - сортировка слиянием

LSDSortDouble - поразрядная сортировка

Rand\_arr - генерирование случайного массива

Массивы и переменные:

ProcNum - количество процессов

ProcRank - номер текущего процесса

bufSize - размер массива

portion - порция на одном процессе

start\_time - переменная для счета времени

end\_time - переменная для счета времени

section - переменная для генерирования массива

frac - переменная для генерирования массива

arrayMain - сортируемый массив

arrayLoc - куски сортируемого массива

tmp1 - вспомогательный массива для сортировки с несчетными процессами

recarr - массив для функции Allgatherv, показывает размер передаваемой части

recarrind - массив для функции Allgatherv, показывает отступ передаваемой части от начала принимающего буфера

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

# Подтверждение корректности

Для подтверждения корректности в программе реализована линейная и параллельная версия. Показано время, с которым работает линейная и параллельная версии, также результаты можно вывести и сравнить результаты сортировок (при больших размерах массивов это делать нецелесообразно )   
Кроме этого запоминается отсортированный массив на линейной версии и сравнивается с сортировкой на параллельной, после это выдается сообщение о том, что ошибки нет, или о том, что найдены не одинаковые элементы

# Результаты экспериментов

Программа запускалась на компьютере со следующими характеристиками:

* Процессор: Intel Core i5 2.50GHz;
* Оперативная память: 6ГБ;
* ОС: Windows 8.1 Профессиональная

График с результатами: Время от количества процессов, ускорение от количества процессов

количество процессов

В таблицу заносились средние значения по 5 экспериментам, во всех экспериментах сортировка происходила на массиве типа double длиной в 10000000 элементов.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Количество процессов | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Время работы(пар), **сек** | 0,8 | 0,788 | 0,805 | 0,811 | 0,795 | 0,790 | 0,803 | 0,8 | 0,815 | 0,819 |
| Время работы(лин), **сек** | 0,8 | 0,688 | 0,668 | 0,739 | 0,868 | 0,896 | 0,790 | 1,050 | 1,33 | 1,476 |
| Ускорение (пар) | 1,00 | 1,14 | 1,2 | 1,10 | 0,91 | 0,88 | 1,02 | 0,76 | 0,6 | 0,55 |

# Заключение

Мы не получили большого ускорения за счет распараллеливания программы.  
Возможно, это произошло из-за использования функции Allgatherv . Также это могло произойти из-за того, что сортировка слиянием работает недостаточно быстро

# Приложение

#include "stdafx.h"

#include <stdio.h>

#include "mpi.h"

#include <math.h>

#include <conio.h>

#include <time.h>

#include <stdlib.h>

void show\_array(double \*arr, int size) {

int n = 0;

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

printf("%d)\_", i);

if (n++> 4) {

n = 0;

printf("%.3f\n ", arr[i]);

}

else

{

printf("%.3f ", arr[i]);

}

}

printf("\n");

printf("\n");

}

double\* MergeSort(double\* mesA, double\* mesB, int size) {

double \* rez = new double[2\*size];

int k = 0;

int z = 0;

for (int i = 0; i < 2\*size; i++) {

if (mesA[k] > mesB[z]) {

rez[i] = mesB[z++];

}

else {

rez[i] = mesA[k++];

}

if (k == size-1 || z == size-1) break;

}

if (k >= z) {

while (z != k) {

rez[z + k] = mesB[z++];

}

}

else {

while (z != k) {

rez[z + k] = mesA[k++];

}

}

return rez;

}

double\* merge(double \*m1, double\* m2, int size1,int size2)

{

double\* ret = new double[size1+size2];

int n = 0;

// Сливаем массивы, пока один не закончится

while (size1 && size2) {

if (\*m1 < \*m2) {

ret[n] = \*m1;

m1++;

--size1;

}

else {

ret[n] = \*m2;

++m2;

--size2;

}

++n;

}

// Если закончился первый массив

if (size1 == 0) {

for (int i = 0; i < size2; ++i) {

ret[n++] = \*m2++;

}

}

else { // Если закончился второй массив

for (int i = 0; i < size1; ++i) {

ret[n++] = \*m1++;

}

}

return ret;

}

void CountingSort(double \*inp, double \*out, int byteNum,

int size)

{

unsigned char \*mas = (unsigned char \*)inp;

int counter[256];

int tem;

memset(counter, 0, sizeof(int) \* 256);

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

counter[mas[8 \* i + byteNum]]++;

int j = 0;

for (; j<256; j++)

{

if (counter[j] != 0)

break;

}

tem = counter[j];

counter[j] = 0;

j++;

for (; j<256; j++)

{

int b = counter[j];

counter[j] = tem;

tem += b;

}

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

{

out[counter[mas[8 \* i + byteNum]]] = inp[i];

counter[mas[8 \* i + byteNum]]++;

}

}

void LSDSortDouble(double \*inp, int size)

{

double \*out = new double[size];

CountingSort(inp, out, 0, size);

CountingSort(out, inp, 1, size);

CountingSort(inp, out, 2, size);

CountingSort(out, inp, 3, size);

CountingSort(inp, out, 4, size);

CountingSort(out, inp, 5, size);

CountingSort(inp, out, 6, size);

CountingSort(out, inp, 7, size);

delete[] out;

}

int Rand\_arr(double\* arrayMain, int bufSize, int section, int frac) {

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

{

double tmp = rand() % section;

tmp += rand() % frac / (double)frac;

arrayMain[i] = tmp;

}

return 1;

}

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

{

int ProcNum, ProcRank, RecvRank;

int bufSize=0;

int portion=0;

double start\_time = 0;

double end\_time = 0;

const int section = 10000;

const int frac = 1000;

double \*arrayMain = nullptr;

double \*arrayLoc = nullptr;

double \*tmp1 = nullptr;

MPI\_Init(&argc, &argv);

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

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

int \*recarr = new int[ProcNum];

int \*recarrind = new int[ProcNum];

if (ProcRank == 0) {

printf("Enter size of array: ");

fflush(stdout);

scanf\_s("%d", &bufSize);

if (bufSize <= 0 ) {

printf("Error: Wrong number");

return 0;

}

if (bufSize < ProcNum) {

bufSize=ProcNum;

}

if (bufSize % ProcNum!=0) {

bufSize = bufSize+ (ProcNum-bufSize % ProcNum);

}

portion = bufSize / ProcNum;

srand(time(0));

arrayMain = new double[bufSize];

arrayLoc = new double[bufSize];

tmp1 = new double[bufSize];

Rand\_arr(arrayMain, bufSize, section, frac);

}

MPI\_Bcast(&portion, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(&bufSize, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

arrayLoc = new double[portion];

if(ProcRank!=0)

arrayMain = new double[bufSize];

MPI\_Scatter(arrayMain, portion, MPI\_DOUBLE, arrayLoc, portion, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

//vertion 1, liner

if (ProcRank == 0) {

start\_time = MPI\_Wtime();

LSDSortDouble(arrayMain, bufSize);

printf("Liner program\n"); //show

fflush(stdout);

end\_time = MPI\_Wtime();// end line

printf("Time = %.3f\n", (end\_time - start\_time));

fflush(stdout);

Rand\_arr(arrayMain, bufSize, section, frac);

}

MPI\_Barrier(MPI\_COMM\_WORLD);

start\_time = 0;

end\_time = 0;

//version 2, parallel

if (ProcRank == 0)

{

start\_time = MPI\_Wtime(); //start parallel

}

LSDSortDouble(arrayLoc, portion);

MPI\_Allgather(arrayLoc, portion, MPI\_DOUBLE, arrayMain, portion, MPI\_DOUBLE, MPI\_COMM\_WORLD);

int x = ProcNum;

int t = 1;

while (x>=1) {

if (ProcRank < x / 2) {

///arrayLoc = new double[portion\*t \* 2];

arrayLoc = merge(&arrayMain[2 \* ProcRank \* portion\*t], &arrayMain[0 + (2 \* ProcRank + 1) \* portion\*t], portion\*t, portion\*t);

if ((x / 2 == ProcRank + 1 )) {

if (bufSize - 2 \* portion\*t\*(x / 2) > 0) {

tmp1 = merge(arrayLoc, &arrayMain[0 + (2 \* ProcRank + 2) \* portion\*t], portion\*t \* 2, bufSize - 2 \* portion\*t\*(x / 2));

delete[] arrayLoc;

arrayLoc = tmp1;

}

}

}

t = t \* 2;

int k = portion \* t;

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

if (i < x/2) {

if (i + 1 == x/2 ) {

recarr[i] = bufSize - portion\*t\*(x / 2) + portion \* t;

}

else

{

recarr[i] = portion \* t;

}

recarrind[i] = portion \* t\*i;

}

else {

recarr[i] = 0;

recarrind[i] = bufSize;

}

}

if (ProcRank + 1 == x/2 ) {

k = bufSize - portion \* t\*(x / 2) + portion \* t;

}

if (ProcRank >= x / 2)k = 0;

x = x / 2;

if(x!=1)

MPI\_Allgatherv(arrayLoc, k, MPI\_DOUBLE, arrayMain, recarr, recarrind, MPI\_DOUBLE, MPI\_COMM\_WORLD);

}

if (ProcRank == 0)

{

end\_time = MPI\_Wtime();// end line

printf("Parallel program \n"); //show

fflush(stdout);

//show\_array(arrayLoc, bufSize);

//fflush(stdout);

printf("Time = %.3f\n", (end\_time - start\_time));

fflush(stdout);

}

delete[] arrayMain;

delete[] arrayLoc;

delete[] recarr;

delete[] recarrind;

MPI\_Finalize();

return 0;

}