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 целых чисел с четно-нечетным слиянием Бетчера»**

**Выполнил:**

студент группы 381608

Волков Д.Д.

**Проверил:**

Кустикова В.Ю.

Нижний Новгород

2018

**Содержание**

[Постановка задачи 3](#_Toc533618619)

[Метод решения 4](#_Toc533618620)

[Схема распараллеливания 6](#_Toc533618625)

[Описание программной реализации 7](#_Toc533618626)

[Подтверждение корректности 8](#_Toc533618628)

[Результаты экспериментов 9](#_Toc533618629)

[Заключение 10](#_Toc533618630)

[Приложение 11](#_Toc533618631)

# Постановка задачи

Задача формулируется следующим образом: требуется разработать программу, сортирующую массив целочисленных элементов с помощью поразрядной сортировки со слиянием Бетчера с использованием методов параллельного программирования. Для этого разработать последовательный и параллельный алгоритмы данной сортировки. Затем сравнить время выполнения на различном числе процессов и определить ускорение.

Программа должна использовать интерфейс MPI для эффективной работы на системах с распределенной памятью.

Выполнение задачи включает:

1. Освоение темы (постановка задачи).
2. Изучение метода решения.
3. Разработку схемы параллельных вычислений.
4. Реализацию программы.
5. Проведение вычислительных экспериментов.
6. Подготовку отчета с анализом результатов экспериментов

В программе необходимо:

1. Выполнить проверку совпадения результатов последовательной и параллельной реализаций.
2. Продемонстрировать корректность работы алгоритмов на задаче/задачах малой размерности.
3. Обеспечить генерацию данных для задач произвольной размерности. Параметры задачи должен задавать пользователь.
4. Вывести время работы последовательного и параллельного алгоритмов. Ускорение параллельного алгоритма.

# Метод решения

## Описание последовательного алгоритма сортировки.

Алгоритм состоит из трёх шагов:

1. Создать безразмерный буфер с 19 «кармашками» (да всех цифр -9…9), в который мы будем распределять числа и сортируемого массива
2. *Распределение*: смотрим на последний разряд числа и кладем его в «кармашек» с таким же номером
3. По порядку с -9 до 9 извлекаем числа из буфера, в том же порядке, в котором они были туда положены.
4. Повторяем для второго разряда и т.д. Пока у самого большого числа не кончаться разряды, для остальных недостающие разряды заменяем нулями

В последовательной реализации четно-нечетное слияние Бетчера будет отсутствовать, в связи с тем, что алгоритм не предполагает слияние отсортированных массивов.

## Описание параллельного алгоритма сортировки

Основа параллельной реализации данной сортировки заключается в следующем:

1. Распределение данных между процессами
2. Сортировка на каждом процессе собственного фрагмента массива, использую функцию *RadixSort()*
3. Отсортированные фрагменты объединяются методом четно-нечетного слияния Бэтчера.

## 

## Алгоритм слияния

Чётно-нечётное слияние Бэтчера заключается в том, что два упорядоченных массива, которые необходимо слить, разделяются на чётные и нечётные элементы. Такое слияние может быть выполнено параллельно. Чтобы массив стал окончательно отсортированным, достаточно сравнить пары элементов, стоящие на нечётной и чётной позициях. Первый и последний элементы массива проверять не надо, т.к. они являются минимальным и максимальным элементов массивов.

Чётно-нечётное слияние Бэтчера позволяет задействовать 2 потока при слиянии двух упорядоченных массивов. В этом случае слияние n массивов могут выполнять n параллельных потоков. На следующем шаге слияние n/2 полученных массивов будут выполнять n/2 потоков и т.д. На последнем шаге два массива будут сливать 2 потока.

# Схема распараллеливания

Схема построена для num\_proc = 8;

Rank 6

Rank 5

Rank 2

Rank 4

Rank 7

Rank 3

Rank 1

Rank 0

У каждого процесса есть 4 переменных:

1. Parent – процесс на уровень выше, для которого данный является ребенком, от родителя в начале работы программы, все и получают свою часть данных
2. Brother – процесс находящийся на том же уровне и имеющий того же родителя, с этим процессом и будет происходить слияние
3. Children1 – процесс которому в начале работы программы посылается часть данных, так же от него будет получена слитая часть массива, от всех потомков
4. Children2 – брат Children1 получает часть данных, сливается после обработки в Children1

# Описание программной реализации

**Руководство пользователя**

Для запуска программы через командную строку необходимо задать дополнительный параметр – размер массива. Например, для сортировки массива из 10000 элементов на 4-х процессах программу следует запустить следующим образом:

mpiexec -n 4 Testpo

Please, enter vector size: 50000000

**Руководство программиста**

На корневом процессе выделяется память для всего массива размера *all\_size*. После массив распределяется между процессами, каждый процесс получает от родителя половину или половину + 1, хранимого у него массива, в итоге процессы нижнего уровня получают примерно равные части, и сортируют их методом RadixSort();

void RadixSort(int\* arr, int size);

Далее каждый массив работает со своим братом а именно: они выполняют четно-нечетное слияние Бэтчера, после чего левый Брат отправляет слитый массив родителю, тот выполняет слияние со своим братом и т.д. пока не дойдут до процесса у которого нет брата (rank 1). Он делает завершающий проход по массиву и отправляет сортированный массив 0 процессу.

# Подтверждение корректности

Для подтверждения корректности в программе реализована автоматическая проверка которая возвращает истину, если каждый элемент с индексом *i* массива отсортированного параллельноравен элементу с индексом i массива отсортированного последовательно.

# Результаты экспериментов

Эксперименты проводились на компьютере со следующими характеристиками

* Процессор Intel Core i7 8750
* ОЗУ 8 Гб
* 6 физических/12 логических процессоров
* Операционная система Windows 10

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | N = 100000 | N = 1000000 | N = 10000000 | N = 100000000 |
| num\_proc = 2 | 0.05 | 0.45 | 5.6 | 54.9 |
| num\_proc = 4 | 0,03 | 0.27 | 3.08 | 29.12 |
| num\_proc = 8 | 0.02 | 0.18 | 1.87 | 18.3 |

Рисунок 2 Ускорение сортировки

Из результатов экспериментов можно сделать вывод, что самое наибольшее ускорение достигается на 8-х процессах. На большом кол-ве процессов мы будем наблюдать замедление и связанно это с там что кол-во созданных процессов превышает физ/лог кол-во процессоров, занимаются процессы примерно одним и тем же, отсюда могут образовываться «пробки» и время может даже ухудшиться.

# Заключение

Была изучена технология построение сетей сортировки, в частности – сетей сортировки с четно-нечетным слиянием Бетчера. В тоже время, была разработана схема распараллеливания быстрой сортировки, дающая оптимальное ускорение на 8 процессах >2 раз.

# Приложение

#include<iostream>

#include<queue>

#include <mpi.h>

#include<time.h>

using namespace std;

void swap(int\* a, int \* b)

{

int tmp = \*a;

\*a = \*b;

\*b = tmp;

}

void RadixSort(int\* arr, int size)

{

int maxRadix = 0;

int tmp, count, max=abs(arr[0]);

queue<int> buf[20];

for (size\_t i = 0; i < size; i++)

{

/\*tmp = arr[i];

count = (tmp < 0) ? 0 : 9;

while (tmp != 0)

{

tmp /= 10;

count++;

}

if (count > maxRadix) maxRadix = count;\*/

if (max < abs(arr[i])) max = abs(arr[i]); }

while (max != 0)

{

max /= 10;

maxRadix++;

}

for (size\_t i = 0; i < maxRadix; i++)

{

for (size\_t j = 0; j < size; j++)

{

tmp = arr[j];

count = 0;

while (count < i)

{

tmp /= 10;

count++;

}

buf[tmp % 10 + 10].push(arr[j]);

}

count = 0;

for (size\_t num = 0; num < 20; num++)

{

while (!buf[num].empty())

{

arr[count++] = buf[num].front();

buf[num].pop();

}

}

}

}

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

{

srand(time(NULL));

const int info = 0, sizeData = 1, data = 2;

int rank, size, num\_proc;

int parent = -1, brother = -1, children1 = -1, children2 = -1;

int VectorSize;

double time;

int \* part, \*part2;

int tmp[4];

int SizeOfParts, SizeOfParts2;

MPI\_Comm comm\_cart;

MPI\_Status status;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_proc);

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

if (rank == 0) {

cout << num\_proc << endl;

cout << "Please, enter vector size: ";

cin >> VectorSize;

part = new int[VectorSize];

part2 = new int[VectorSize];

//std::cout << "Vector:";

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

part[i] = part2[i] = rand() % 1000 - 500;

//cout << part[i] << " ";

}

time = clock();

RadixSort(part2, VectorSize);

/\*std::cout << "\nVector sort:";

for (int a = 0; a < VectorSize; a++)

{

cout << part2[a] << " ";

}

cout << endl;\*/

cout << "Serial time: " << (clock() - time)/CLOCKS\_PER\_SEC << endl;

cout << endl;

time = MPI\_Wtime();

children1 = 1;

tmp[0] = 0;

tmp[1] = -1;

tmp[2] = 2;

tmp[3] = 3;

MPI\_Send(tmp, 4, MPI\_INT, 1, info, MPI\_COMM\_WORLD);

tmp[0] = 1;

tmp[1] = 3;

tmp[2] = 4;

tmp[3] = 5;

MPI\_Send(tmp, 4, MPI\_INT, 2, info, MPI\_COMM\_WORLD);

tmp[0] = 1;

tmp[1] = 2;

tmp[2] = 6;

tmp[3] = 7;

MPI\_Send(tmp, 4, MPI\_INT, 3, info, MPI\_COMM\_WORLD);

tmp[0] = 2;

tmp[1] = 5;

tmp[2] = -1;

tmp[3] = -1;

MPI\_Send(tmp, 4, MPI\_INT, 4, info, MPI\_COMM\_WORLD);

tmp[0] = 2;

tmp[1] = 4;

tmp[2] = -1;

tmp[3] = -1;

MPI\_Send(tmp, 4, MPI\_INT, 5, info, MPI\_COMM\_WORLD);

tmp[0] = 3;

tmp[1] = 7;

tmp[2] = -1;

tmp[3] = -1;

MPI\_Send(tmp, 4, MPI\_INT, 6, info, MPI\_COMM\_WORLD);

tmp[0] = 3;

tmp[1] = 6;

tmp[2] = -1;

tmp[3] = -1;

MPI\_Send(tmp, 4, MPI\_INT, 7, info, MPI\_COMM\_WORLD);

MPI\_Send(&VectorSize, 1, MPI\_INT, children1, sizeData, MPI\_COMM\_WORLD);

MPI\_Send(part, VectorSize, MPI\_INT, children1, data, MPI\_COMM\_WORLD);

delete[] part;

part = new int[VectorSize];

MPI\_Recv(part, VectorSize, MPI\_INT, children1, data, MPI\_COMM\_WORLD, &status);

time = MPI\_Wtime() - time;

cout << "Parallel sort: " << time << endl;

bool check = true;

for (size\_t i = 0; i < VectorSize; i++)

{

if (part[i] != part2[i]) check = false;

}

if (check)cout << "\nOK\n" << endl;

else

cout << "\nnotOK\n" << endl;

/\* cout << "\nSORT (" << rank << "): ";

for (int a = 0; a < VectorSize; a++)

{

cout << part[a] << " ";

}

cout << endl;\*/

}

else {

//if(parent==-1)MPI\_Finalize();

MPI\_Recv(tmp, 4, MPI\_INT, 0, info, MPI\_COMM\_WORLD, &status);

parent = tmp[0];

brother = tmp[1];

children1 = tmp[2];

children2 = tmp[3];

/\*cout << "Process number: " << rank << " info :\nParent: " << parent << "\nBrother: " << brother << "\nClidren: " << children1 <<", " <<children2<< std::endl;\*/

MPI\_Recv(&SizeOfParts, 1, MPI\_INT, parent, sizeData, MPI\_COMM\_WORLD, &status);

part = new int[SizeOfParts];

MPI\_Recv(part, SizeOfParts, MPI\_INT, parent, data, MPI\_COMM\_WORLD, &status);

/\*cout << "Process number: " << rank << " works ... " << std::endl;

cout << "\nPart (" << rank << "): ";

for (int a = 0; a < SizeOfParts; a++)

{

cout << part[a] << " ";

}

cout << endl;\*/

if (children1 != -1 && children2 != -1)

{

size = SizeOfParts / 2;

MPI\_Send(&size, 1, MPI\_INT, children1, sizeData, MPI\_COMM\_WORLD);

MPI\_Send(part, size, MPI\_INT, children1, data, MPI\_COMM\_WORLD);

size = SizeOfParts - size;

MPI\_Send(&size, 1, MPI\_INT, children2, sizeData, MPI\_COMM\_WORLD);

MPI\_Send(part + (SizeOfParts - size), size, MPI\_INT, children2, data, MPI\_COMM\_WORLD);

delete[] part;

part = new int[SizeOfParts];

MPI\_Recv(part, SizeOfParts, MPI\_INT, children1, data, MPI\_COMM\_WORLD, &status);

}

else

{

RadixSort(part, SizeOfParts);

/\*cout << "\nPart after sort(" << rank << "): ";

for (int a = 0; a < SizeOfParts; a++)

{

cout << part[a] << " ";

}

cout << endl;\*/

}

if (brother != -1)

{

if (brother > rank)

{

MPI\_Send(&SizeOfParts, 1, MPI\_INT, brother, sizeData, MPI\_COMM\_WORLD);

MPI\_Send(part, SizeOfParts, MPI\_INT, brother, data, MPI\_COMM\_WORLD);

MPI\_Recv(&SizeOfParts2, 1, MPI\_INT, brother, sizeData, MPI\_COMM\_WORLD, &status);

part2 = new int[SizeOfParts2];

MPI\_Recv(part2, SizeOfParts2, MPI\_INT, brother, data, MPI\_COMM\_WORLD, &status);

int \*res = new int[SizeOfParts + SizeOfParts2];

int i1, i2, ires;

i1 = i2 = ires = 0;

while (i1 < SizeOfParts && i2 < SizeOfParts2)

{

if (part[i1] < part2[i2])

{

res[ires] = part[i1];

i1 += 2;

}

else

{

res[ires] = part2[i2];

i2 += 2;

}

ires += 2;

}

if (i1 >= SizeOfParts) {

while (i2 < SizeOfParts2)

{

res[ires] = part2[i2];

ires += 2;

i2 += 2;

}

}

if (i2 >= SizeOfParts2)

{

while (i1 < SizeOfParts)

{

res[ires] = part[i1];

ires += 2;

i1 += 2;

}

}

int SizeOfRes = SizeOfParts + SizeOfParts2;

int \*res2 = new int[SizeOfRes];

delete[] part;

delete[] part2;

MPI\_Recv(res2, SizeOfRes, MPI\_INT, brother, 5, MPI\_COMM\_WORLD, &status);

part = new int[SizeOfRes];

SizeOfParts = SizeOfRes;

for (size\_t x = 0; x < SizeOfRes; x += 2)

{

part[x] = res[x];

part[x + 1] = res2[x + 1];

}

/\*cout << "\nPart after merge(" << rank << "): ";

for (int a = 0; a < SizeOfParts; a++)

{

cout << part[a] << " ";

}

cout << endl;\*/

delete[] res;

delete[] res2;

MPI\_Send(part, SizeOfParts, MPI\_INT, parent, data, MPI\_COMM\_WORLD);

//MPI\_Finalize();

}

else

{

MPI\_Recv(&SizeOfParts2, 1, MPI\_INT, brother, sizeData, MPI\_COMM\_WORLD, &status);

part2 = new int[SizeOfParts2];

MPI\_Recv(part2, SizeOfParts2, MPI\_INT, brother, data, MPI\_COMM\_WORLD, &status);

MPI\_Send(&SizeOfParts, 1, MPI\_INT, brother, sizeData, MPI\_COMM\_WORLD);

MPI\_Send(part, SizeOfParts, MPI\_INT, brother, data, MPI\_COMM\_WORLD);

int \*res = new int[SizeOfParts + SizeOfParts2];

int i1, i2, ires;

i1 = i2 = ires = 1;

while (i1 < SizeOfParts && i2 < SizeOfParts2)

{

if (part[i1] < part2[i2])

{

res[ires] = part[i1];

i1 += 2;

}

else

{

res[ires] = part2[i2];

i2 += 2;

}

ires += 2;

}

if (i1 >= SizeOfParts) {

while (i2 < SizeOfParts2)

{

res[ires] = part2[i2];

ires += 2;

i2 += 2;

}

}

if (i2 >= SizeOfParts2)

{

while (i1 < SizeOfParts)

{

res[ires] = part[i1];

ires += 2;

i1 += 2;

}

}

int SizeOfRes = SizeOfParts + SizeOfParts2;

MPI\_Send(res, SizeOfRes, MPI\_INT, brother, 5, MPI\_COMM\_WORLD);

delete[] part;

delete[] part2;

delete[] res;

//MPI\_Finalize();

}

}

else {

int i = 1, j = 2, ires=0;

int \*result = new int[SizeOfParts];

result[ires++] = part[0];

result[SizeOfParts - 1] = part[SizeOfParts - 1];

while (i <= SizeOfParts - 2 && j <= SizeOfParts - 1)

{

if (part[i] < part[j]) {

result[ires++] = part[i];

i += 2;

}

else

{

result[ires++] = part[j];

j += 2;

}

}

if (i >= SizeOfParts - 2)

{

while (j < SizeOfParts - 1)

{

result[ires++] = part[j];

j += 2;

}

}

if (j >= SizeOfParts - 1)

{

while (i < SizeOfParts - 1)

{

result[ires++] = part[i];

i += 2;

}

}

MPI\_Send(result, SizeOfParts, MPI\_INT, parent, data, MPI\_COMM\_WORLD);

//MPI\_Finalize();

}

}

MPI\_Finalize();

}