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 Бэтчера»**

**Выполнил:**

студент группы 0836-2

Круглов Е.О.

**Проверил:**

Козинов Е.А.

Нижний Новгород

2018

Оглавление

[Введение 3](#_Toc533129988)

[Постановка задачи 3](#_Toc533129989)

[Метод решения 4](#_Toc533129990)

[6](#_Toc533129991)

[Схема распараллеливания 7](#_Toc533129992)

[Описание программной реализации 7](#_Toc533129993)

[Подтверждение корректности 8](#_Toc533129994)

[Результаты экспериментов 9](#_Toc533129995)

[Заключение 10](#_Toc533129996)

[Приложение 10](#_Toc533129997)

# Введение

Сортировка (англ. sorting — классификация, упорядочение) — последовательное расположение или разбиение на группы чего-либо в зависимости от выбранного критерия.

Сортировка является одной из типовых проблем обработки, данных, и обычно понимается как задача размещения элементов неупорядоченного набора значений S = {a1, a2, ..., an} в порядке монотонного возрастания или убывания S ~ S' = {(a1', a2’, ..., an'): a1'<= a2' <=... <= an'}.

Возможные способы решения этой задачи широко обсуждаются в литературе. В данной работе рассмотрен метод поразрядной сортировки с четно-нечетным слиянием Бэтчера.

# Постановка задачи

Дан неупорядоченный массив, состоящий из n случайных чисел типа int. Требуется реализовать последовательную и параллельную версию алгоритма поразрядной сортировки для целых чисел и сравнить время их выполнения при различных значениях размера сортируемого массива.

# Метод решения

Идея поразрядной сортировки заключается в том, что выполняется последовательная сортировка чисел по разрядам (от младшего к старшему).

Рассмотрим алгоритм на примере массива из 8 элементов:

9

941

729

322

964

342

902

349

743

На первой итерации выполняется размещение элементов по младшему разряду

349

729

964

941

322

743

902

342

На второй итерации элементы упорядочиваются по второму разряду

941

729

322

902

964

349

743

342

На третьей итерации соответственно по третьему разряду(в нашем примере это старший разряд)

322

941

902

964

743

729

349

342

В итоге получаем отсортированный массив. Сортировка работает только в том случае, если элементы равных разрядов не меняют своего взаимного расположения при сортировке по очередному разряду.

Реализовывать алгоритм, используя десятичные числа, является нецелесообразным, так как современные процессоры предназначены для обработки данных, представленных в битах и байтах. Также использование десятичных чисел неэффективно, так как в максимальном числе типа int 10 разрядов, в то время как в двоичном представлении в нем всего 4 разряда (по количеству байт).

Сортировка имеет линейную сложность.

Целочисленные типы без знака представляются в памяти таким образом:

Целочисленный тип со знаком кодируется таким образом, что при сложении равных по модулю отрицательного и положительного чисел получается ноль.

Старший бит определяет знак числа

* Если старший бит 0, то число положительное
* Если старший бит 1, то число отрицательное

Целочисленный тип со знаком кодируется следующим образом

При таком представлении отрицательные числа кодируются в так называемом дополнительном коде:

* Записывается равное по модулю положительное число
* Выполняется инвертирование всех бит этого числа
* К полученному значению добавляется -1 по правилам сложения чисел без знака

Четно – нечетное слияние Бэтчера заключается в том, что два упорядоченных массива, которые необходимо слить, разделяются на четные и нечетные элементы. Чтобы массив стал полностью отсортированным, достаточно сравнить пары элементов, стоящих на четной и нечетной позициях.

Четно – нечетное слияние Бэтчера можно реализовать параллельно. Оно позволяет задействовать 2 процесса при слиянии двух массивов. Соответственно для произвольного числа элементов на 1 шаге мы сливаем n массивов при помощи n процессов, на следующем шаге – n/2 и так далее. На последнем шаге слияние выполняют два процесса.

N/4

N/4

N/4

N/4

N/2

N/2

# 

N элементов

Реализация алгоритма параллельного слияния предполагает, что программа будет работать на количестве процессов, равных степени двойки.

# Схема распараллеливания

* Находится массив смещений (displs), массив количества отправляемых элементов (sendCounts) и массив полученного буффера (resultBuffer). В каждой ячейке массива будет содержаться соответствующие данные для соответствующего номеру ячейки процесса. Причем, если размер сортируемого массива не делится нацело на количество потоков, то, начиная с 0 потока, будет произведен инкремент значения, в массивах объявленных выше.
* Передаем в функцию MPI\_Scatterv соответствующие параметры, которые мы определили выше, тем самым разделяя массив, который нужно отсортировать, между всеми процессами
* Каждый процесс производит поразрядную сортировку со своим буфером
* После того, как каждый процесс отсортировал свою часть, производится четно – нечетное слияние Бэтчера до 0 процесса, который будет содержать уже отсортированный исходный массив данных.

# Описание программной реализации

Рассмотрим алгоритм для целых положительных чисел. Сортировка по i-му байту будет проходить в два прохода:

1. При первом проходе по исходному массиву выполняется подсчёт i-ых байт в массиве, результат будет сохранён в массив count из 256 элементов.
2. В массив offset, на основании посчитанных данных, выполняется подсчёт индексов, по которым будут сохраняться элементы:

offset[0]=0

для всех j от 1 до 256

offset [j]= offset [j-1] + count[j-1]

1. при втором проходе по исходному массиву выполняется копирование элемента во вспомогательный массив по соответствующему индексу в массиве подсчётов, выполняется инкремент индекса.
2. Из вспомогательного массива все копируются в исходный массив.

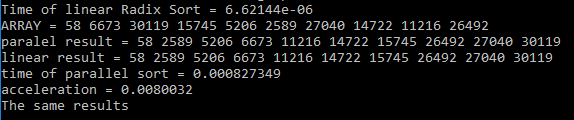
Для отрицательных чисел алгоритм не изменится. Старший бит знаковых типов определяет знак числа, поэтому сортировку нужно выполнять с учётом того, что числа, у которых старший бит равен 0 (положительные) больше тех чисел, у которых старший бит равен 1 (отрицательные). Исходя из этого, получается, что при сортировке знаковых чисел необходимо изменить алгоритм сортировки только по старшему байту. Для этого достаточно в сортировке интерпретировать старший байт, как знаковое число (диапазон знаковых однобайтовых чисел от -128 до 127) и прибавлять смещение 128 Таким образом, в массиве подсчётов по индексу 0 будет находиться количество элементов массива, у которых в старшем байте минимальное число со знаком (-128).

Функция слияния будет получать на вход 2 отсортированных массива, которые необходимо слить, результат будет записывать в буферный массив.

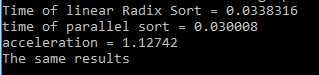
1. Сначала мы проходимся циклом по всем четным ячейкам сливаемых массивов и расставляем их на четные места в порядке возрастания в массив-результат.
2. Повторяем ту же процедуру с нечетными ячейками массивов.
3. Проходимся компаратором по массиву, сравнивая соседние четные и нечетные элементы и, при необходимости, меняя их местами.

# Подтверждение корректности

В ходе работы программы создаётся два одинаковых массива, один из них сортируется последовательно, другой параллельно, результаты их сортировки сравниваются. Проверим на массиве из 10 элементов.



Результаты сортировок больших массивов на экран не выводятся, но в случае правильной работы программы, на экране пишется “The same results”, что свидетельствует о том, что последовательная и параллельная реализации дали одинаковый результат. Пример для 1 000 000 элементов



# Результаты экспериментов

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Размер массива | 100000 | 100000 | 100000 | 1000000 | 1000000 | 1000000 | 10000000 | 10000000 | 10000000 |
| Количество процессов | 2 | 4 | 8 | 2 | 4 | 8 | 2 | 4 | 8 |
| Время работы, c | 0.0035 | 0.0048 | 0.0053 | 0.028 | 0.027 | 0.029 | 0.28 | 0.26 | 0.27 |
| Время последовательного выполнения, c | 0.0030 | 0.0029 | 0.0031 | 0.031 | 0.036 | 0.036 | 0.32 | 0.33 | 0.32 |
| Ускорение, c | 0.85 | 0.61 | 0.59 | 1.1 | 1.32 | 1.26 | 1.14 | 1.27 | 1.18 |

Характеристики компьютера:

* Intel Core i5 – 3450 3.10GHz
* 4 Gb ОЗУ
* OS Windows 10 Pro

# Заключение

В лабораторной работе был разработан последовательный и параллельный алгоритмы поразрядной сортировки с использованием четно-нечетного слияния Бэтчера. Было замерено время их работы. Стоит заметить, что время работы последовательной и параллельной реализации примерно одинаковое, однако параллельная версия требует больше затрат оперативной памяти для работы. Отсюда можно сделать вывод, что на рассмотренных примерах выгоднее использовать последовательный алгоритм. Небольшая потеря во времени особой роли не играет, зато это позволит нам эффективно расходовать оперативную память.

# Приложение

#include <iostream>

#include <time.h>

#include "mpi.h"

using namespace std;

void Radix(int byte, int size, int \*sourceArr, int \*destArr)

{

int count[256];

int offset[256];

memset(count, 0, sizeof(count));

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

{

if (byte == 3)

count[((sourceArr[i] >> (byte \* 8)) + 128) & 0xff]++;

else

count[((sourceArr[i]) >> (byte \* 8)) & 0xff]++;

}

offset[0] = 0;

for (int i = 1; i < 256; ++i)

offset[i] = offset[i - 1] + count[i - 1];

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

{

if (byte == 3)

destArr[offset[((sourceArr[i] >> (byte \* 8)) + 128) & 0xff]++] = sourceArr[i];

else

destArr[offset[((sourceArr[i]) >> (byte \* 8)) & 0xff]++] = sourceArr[i];

}

}

void RadixSort(int \*sourceArr, int size)

{

int \*temp = new int[size];

Radix(0, size, sourceArr, temp);

Radix(1, size, temp, sourceArr);

Radix(2, size, sourceArr, temp);

Radix(3, size, temp, sourceArr);

delete[] temp;

}

void ArrayFill(int \*arr, int size)

{

//srand(time(NULL));

srand(6);

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

arr[i] = rand();

}

void EvenSplit(int\* arr, int\* tmp, int size1, int size2)

{

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

//tmp[i] = arr[i];

int \*arr2 = arr + size1;

int a = 0;

int b = 0;

int i = 0;

while ((a < size1) && (b < size2))

{

if (tmp[a] < arr2[b])

{

arr[i] = tmp[a];

a += 2;

}

else

{

arr[i] = arr2[b];

b += 2;

}

i += 2;

}

if (a == size1)

for (int j = b; j < size2; j += 2, i += 2)

arr[i] = arr2[j];

else

for (int j = a; j<size1; j += 2, i += 2)

arr[i] = tmp[j];

}

void OddSplit(int\* arr, int\* tmp, int size1, int size2)

{

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

// tmp[i] = arr[i];

int \*arr2 = arr + size1;

int a = 1;

int b = 1;

int i = 1;

while ((a < size1) && (b < size2))

{

if (tmp[a] < arr2[b])

{

arr[i] = tmp[a];

a += 2;

}

else

{

arr[i] = arr2[b];

b += 2;

}

i += 2;

}

if (a == size1)

for (int j = b; j < size2; j += 2, i += 2)

arr[i] = arr2[j];

else

for (int j = a; j<size1; j += 2, i += 2)

arr[i] = tmp[j];

}

void Comparator(int \*arr, int size)

{

for (int i = 1; i < size; i++)

if (arr[i] < arr[i - 1])

std::swap(arr[i], arr[i - 1]);

}

double TimeOfLinearSort(int \*arr, int size, int \*linearmas)

{

int \*tmp = new int[size];

double startTime;

double finishTime;

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

tmp[i] = arr[i];

startTime = MPI\_Wtime();

RadixSort(tmp, size);

finishTime = MPI\_Wtime();

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

linearmas[i] = tmp[i];

delete[] tmp;

return (finishTime - startTime);

}

int\* BatcherMerge(int \*arr1, int \*arr2, int size1, int size2)

{

int \*res = new int[size1 + size2];

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

//res[i] = arr1[i];

//for (int j = 0; j < size2; j++)

//res[j + size1] = arr2[j];

memcpy(res, arr1, size1 \* sizeof(int));

memcpy(res + size1, arr2, size2 \* sizeof(int));

//int \*tmp = new int[size1];

EvenSplit(res, arr1 /\*tmp\*/, size1, size2);

OddSplit(res, arr1 /\*tmp\*/, size1, size2);

Comparator(res, size1 + size2);

//delete[] tmp;

return res;

}

void reallocate(int\* &arr, int oldsize, int newsize)

{

int \*newArr = new int[newsize]{ 0 };

memcpy(newArr, arr, oldsize \* sizeof(int));

delete[] arr;

arr = newArr;

}

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

{

int procNum, procRank;

int size = 10000000;

int \*arr = new int[size];

int \*linearmas = new int[size];

int root = 0;

double stime, ftime;

double linTime, parTime;

MPI\_Init(&argv, &argc);

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

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

if (procRank == 0)

{

ArrayFill(arr, size);

linTime = TimeOfLinearSort(arr, size, linearmas);

cout << "Time of linear Radix Sort = " << linTime << endl;

}

int \*sendCounts = new int[procNum];

int \*displs = new int[procNum];

if (procRank == 0)

{

int localBuf = size / procNum;

int reminder = size % procNum;

sendCounts[0] = localBuf + reminder;

displs[0] = 0;

for (int i = 1; i < procNum; i++)

{

sendCounts[i] = localBuf;

displs[i] = reminder + i\*localBuf;

}

}

MPI\_Bcast(sendCounts, procNum, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(displs, procNum, MPI\_INT, 0, MPI\_COMM\_WORLD);

//int \*resultBuffer = new int[sendCounts[procRank]];

int \*resultBuffer = new int[size];

MPI\_Scatterv(arr, sendCounts, displs, MPI\_INT, resultBuffer,

sendCounts[procRank],MPI\_INT, 0, MPI\_COMM\_WORLD);

if (procRank == 0)

stime = MPI\_Wtime();

RadixSort(resultBuffer, sendCounts[procRank]);

int n = procNum;

int m = 1;

int length;

//int \*tempMas = new int[size];

//int \*resMas = new int[size];

while (n > 1)

{

n = n / 2 + n % 2;

if ((procRank - m) % (2 \* m) == 0)

{

MPI\_Send(&sendCounts[procRank], 1, MPI\_INT, procRank - m, 0, MPI\_COMM\_WORLD);

MPI\_Send(resultBuffer, sendCounts[procRank], MPI\_INT, procRank - m, 0, MPI\_COMM\_WORLD);

break;

}

if ((procRank % (2 \* m) == 0) && (procNum - procRank > m))

{

MPI\_Status status;

MPI\_Recv(&length, 1, MPI\_INT, procRank + m, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status);

int \*tempMas = new int[length];

MPI\_Recv(tempMas, length, MPI\_INT, procRank + m, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status);

resultBuffer = BatcherMerge(resultBuffer, tempMas, sendCounts[procRank], length);

sendCounts[procRank] += length;

/\* if (size == sendCounts[procRank])

{

for (int i = 0; i < length + sendCounts[procRank]; i++)

{

resultBuffer[i] = resMas[i];

}

}\*/

}

m \*= 2;

}

if (procRank == 0)

{

ftime = MPI\_Wtime();

parTime = ftime - stime;

}

if (procRank == 0)

{

//cout << "ARRAY = ";

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

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

//cout << endl;

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

arr[i] = resultBuffer[i];

delete[] resultBuffer;

bool flag = true;

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

if (arr[i] != linearmas[i])

{

flag = false;

break;

}

//cout << "paralel result = ";

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

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

//cout << endl;

//cout << "linear result = ";

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

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

//cout << endl;

cout << "time of parallel sort = " << parTime << endl;

double uskor = linTime / parTime;

cout << "acceleration = " << uskor << endl;

if (flag)

cout << "The same results" << endl;

else

cout << "Error" << endl;

}

delete[] sendCounts;

delete[] displs;

//delete[] tempMas;

//delete[] resMas;

MPI\_Finalize();

delete[] arr;

}