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

**«Сортировка Шелла со слиянием «Разделяй и властвуй»»**

**Выполнил:**

студент группы 1608

Чванов Л.Л.

**Проверил:**

Кустикова В.Д.

Нижний Новгород

2018

**Содержание**

[Постановка задачи 3](#_Toc531367741)

[Метод решения 4](#_Toc531367742)

[Схема распараллеливания 5](#_Toc531367743)

[Описание программной реализации 6](#_Toc531367744)

[Подтверждение корректности 7](#_Toc531367745)

[Результаты экспериментов 8](#_Toc531367746)

[Заключение 10](#_Toc531367747)

[Приложение 11](#_Toc531367748)

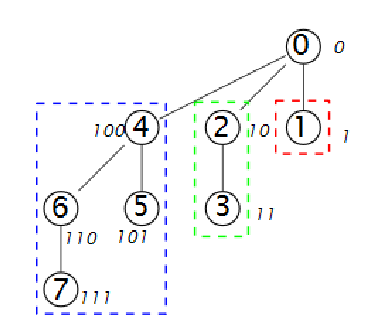
# Постановка задачи

В рамках данной лабораторной работы необходимо разработать программу, использующую сортировку Шелла и слияние по принципу «Разделяй и властвуй» 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 сортировка Шелла).

# Метод решения

Каждый процесс сортирует полученную часть исходного массива с помощью сортировки Шелла с использованием промежутков из последовательности Шелла, где за изначальный промежуток берется половина длины массива, каждый последующий равен половине предыдущего, пока размер промежутка не станет равен 1.

Затем каждый процесс отправляет свою отсортированную часть массива родителю в соответствии биномиальному дереву, где в корне находится нулевой процесс, а номера дочерних процессов отличаются от номеров родительских на значения, равные степеням двойки.



Родительский процесс, получив массив, сортирует его с имеющейся в нем частью исходного массива с помощью сортировки слиянием. Затем ожидает получения массива от следующего ребенка, если таковой существует.

В результате работы на нулевом процессе оказывается отсортированный массив.

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 первыми процессами). Также на нулевом процессе вычисляются массив максимальных размеров массивов для всех процессов и отправляется каждому процессу. Затем каждый процесс линейно сортирует полученную часть массива с помощью сортировки Шелла. Сбор данных происходит по схеме биномиального дерева с сортировкой слиянием на каждом шаге.

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 nP "path/to/program.exe" aS,

где nP – количество процессов, aS – размер массива. В результате работы программы выводится время выполнения линейной и параллельной версий программы в секундах.

**Руководство программиста**

Программа разделена на 2 модуля:

* ShellSort
* main

Модуль ShellSort содержит следующие функции:

* shellSort – сортировка Шелла с использованием последовательности промежутков Шелла;
* merge – сортировка слиянием двух массивов с записью результата в третий массив.

Модуль main содержит следующие функции:

* findFirstSet – возвращает номер первого установленного бита числа (используется для нахождения номера родительского процесса);
* generateRandomArraw – генерирует случайный массив;
* printArray – выводит массив на консоль;
* calcSendCountsDispls – вычисляет массивы, необходимые для использования функции MPI\_Scatterv;
* calcMergeBufferSize – вычисляет максимальные размеры массивов для каждого процесса;
* main – основная функция программы.

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

# Подтверждение корректности

Для подтверждения корректности работы программы нулевой процесс сравнивает результаты работы параллельного и линейного вариантов программы и выводит на консоль номера элементов массива, значения которых не совпали.

# Результаты экспериментов

Эксперименты проводились на машине со следующими характеристиками:

* Процессор: AMD Athlon X4 860K Quad Core 3.70GHz;
* Оперативная память: 16ГБ Dual-Channel DDR3 1866MHz;
* ОС: Windows 10

В таблицу заносились средние значения по 10 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, сек | 9,02 | 4,38 | 2,70 | 2,31 | 2,72 | 2,23 | 2,44 | 2,35 | 2,20 | 2,30 |
| Ускорение | 1,00 | 2,06 | 3,34 | 3,90 | 3,32 | 4,04 | 3,70 | 3,84 | 4,10 | 3,92 |

По данным экспериментов видно, что время выполнения снижается с ростом числа процессов. Нелинейность зависимости можно объяснить ростом накладных расходов при увеличении числа процессов (пересылка данных между процессами, создание дополнительных массивов для приема данных). Падения ускорения при использовании 5 и 7 процессов можно объяснить схемой слияния данных.

# Заключение

В результате работы была написана программа, сортирующая массив с помощью сортировок Шелла и слияния по схеме «Разделяй и властвуй». По результатам экспериментов видно, что параллельная версия программы работает быстрее линейной. Рост ускорение быстро достигает насыщения, что объясняется накладными расходами на обмен данными между процессами.

# Приложение

Код файла «ShellSort.h»:

#ifndef \_\_SHELLSORT\_H\_\_

#define \_\_SHELLSORT\_H\_\_

#define TYPE double

#define MPI\_TYPE MPI\_DOUBLE

void shellSort(TYPE\* arr, int n)

{

int step;

double tmp;

for (step = n/2; step > 0; step /=2)

for (int i = step; i < n; ++i)

for (int j = i - step; (j >= 0) && (arr[j] > arr[j + step]); j -= step)

{

tmp = arr[j];

arr[j] = arr[j + step];

arr[j + step] = tmp;

}

}

void merge(TYPE\* arr1, int n1, TYPE\* arr2, int n2, TYPE\* res)

{

int i, j, k;

i = 0; j = 0; k = 0;

while ((i < n1) && (j < n2))

if (arr1[i]<arr2[j])

{

res[k] = arr1[i];

i++; k++;

}

else

{

res[k] = arr2[j];

j++; k++;

}

if (i == n1)

while (j<n2)

{

res[k] = arr2[j];

j++; k++;

}

else

while (i<n1)

{

res[k] = arr1[i];

i++; k++;

}

}

#endif // \_\_SHELLSORT\_H\_\_

Код файла «main.cpp»:

#include <cstdlib>

#include <stdlib.h>

#include <iostream>

#include <time.h>

#include <mpi.h>

#include "ShellSort.h"

inline int findFirstSet(int n)

{

if (!n)

return 0;

int i;

for (i = 0; !(n & 1); ++i)

n >>= 1;

return i;

}

void generateRandomArray(TYPE\* arr, int n)

{

if (n > 0)

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

arr[i] = (TYPE)rand();

}

void printArray(int\* arr, int size)

{

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

std::cout << arr[i] << ' ';

std::cout << std::endl;

}

void calcSendCountsDispls(int\* sendcounts, int\* displs, int n, int procNum)

{

int sum = 0, rem = n % procNum;

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

sendcounts[i] = n / procNum;

if (rem > 0) {

sendcounts[i]++;

rem--;

}

displs[i] = sum;

sum += sendcounts[i];

}

}

void calcMergeBuffSize(int\* mergeBuffSize, int\* sendcounts, int procNum, int procRank)

{

for (int i = 1; ((procRank | i) > procRank) && ((procRank | i) < procNum); i <<= 1)

{

if (mergeBuffSize[procRank | i] == 0)

calcMergeBuffSize(mergeBuffSize, sendcounts, procNum, procRank | i);

mergeBuffSize[procRank] += mergeBuffSize[procRank | i];

}

mergeBuffSize[procRank] += sendcounts[procRank];

}

using namespace std;

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

{

TYPE\* mergeBuff = nullptr, \*recBuff = nullptr, \*arr = nullptr;

double startTime, parTime, linTime;

int procRank, procNum;

int\* sendcounts, \* displs, \* mergeBuffSize;

MPI\_Status status;

MPI\_Init(&argc, &argv);

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

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

int ARR\_SIZE = atoi(argv[1]);

if (procRank == 0)

{

srand(time(NULL));

arr = new TYPE[ARR\_SIZE];

generateRandomArray(arr, ARR\_SIZE);

startTime = MPI\_Wtime();

}

sendcounts = new int[procNum];

displs = new int[procNum];

mergeBuffSize = new int[procNum];

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

mergeBuffSize[i] = 0;

if (procRank == 0)

{

calcSendCountsDispls(sendcounts, displs, ARR\_SIZE, procNum);

calcMergeBuffSize(mergeBuffSize, sendcounts, procNum, 0);

}

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

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

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

mergeBuff = new TYPE[mergeBuffSize[procRank]];

TYPE\* curBuff = mergeBuff + mergeBuffSize[procRank] - sendcounts[procRank];

int curSize = sendcounts[procRank];

MPI\_Scatterv(arr, sendcounts, displs, MPI\_TYPE,

curBuff, sendcounts[procRank], MPI\_TYPE, 0, MPI\_COMM\_WORLD);

shellSort(curBuff, sendcounts[procRank]);

for (int i = 1; ((procRank | i) > procRank) && ((procRank | i) < procNum); i <<= 1)

{

int pSender = procRank | i;

int recCount = mergeBuffSize[pSender];

recBuff = new TYPE[recCount];

MPI\_Recv(recBuff, recCount, MPI\_TYPE, pSender, 0, MPI\_COMM\_WORLD, &status);

merge(curBuff, curSize, recBuff, recCount, curBuff - recCount);

curSize += recCount;

curBuff -= recCount;

delete[] recBuff;

}

if (procRank)

{

int pReceiver = procRank - (1 << findFirstSet(procRank));

MPI\_Send(mergeBuff, mergeBuffSize[procRank],

MPI\_TYPE, pReceiver, 0, MPI\_COMM\_WORLD);

}

if (procRank == 0)

{

parTime = MPI\_Wtime() - startTime;

startTime = MPI\_Wtime();

shellSort(arr, ARR\_SIZE);

linTime = MPI\_Wtime() - startTime;

for (int i = 0; i < ARR\_SIZE; ++i)

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

cout << i << ' ';

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

cout << "Linear sort time: " << linTime << endl;

delete[] arr;

}

delete[] mergeBuff;

MPI\_Finalize();

}