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 381606-1

Яковлев Д.Л.

**Проверил:**

Кустикова В.Д.

Нижний Новгород

2018

**Содержание**

[Постановка задачи 3](#_Toc533717515)

[Метод решения 4](#_Toc533717516)

[Схема распараллеливания 6](#_Toc533717517)

[Описание программной реализации 8](#_Toc533717518)

[Подтверждение корректности 10](#_Toc533717519)

[Результаты экспериментов 11](#_Toc533717520)

[Заключение 13](#_Toc533717521)

[Приложение 14](#_Toc533717522)

# Постановка задачи

Была поставлена задача разработать программу, сортирующую массив элементов с помощью сортировки Шелла с использованием методов параллельного программирования. Для этого разработать последовательный и параллельный алгоритм данной сортировки. При параллельном алгоритме сортировки, массив чисел разбивается на части, потом каждая часть сортируется независимо, затем выполняется слияние методом «Разделяй и властвуй».

Программа должна иметь параллельную реализацию, выполненную при помощи MPI.

В программе необходимо:

1. Выполнить проверку совпадения результатов последовательной и параллельной реализаций.
2. Обеспечить генерацию данных для задач произвольной размерности. Параметры задачи должен задавать пользователь.
3. Вывести время работы последовательного и параллельного алгоритмов.

# Метод решения

Описание алгоритма «Сортировка Шелла».

**Сортировка Шелла** — алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками. Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга. Иными словами — это сортировка вставками с предварительными «грубыми» проходами.

При сортировке Шелла сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии **d**. После этого процедура повторяется для некоторых меньших значений **d**, а завершается сортировка Шелла упорядочиванием элементов при **d=1** (то есть обычной сортировкой вставками). Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы «быстрее» встают на свои места (в простых методах сортировки, например, пузырьковой, каждая перестановка двух элементов уменьшает количество инверсий в списке максимум на 1, а при сортировке Шелла это число может быть больше).

Невзирая на то, что сортировка Шелла во многих случаях медленнее, чем быстрая сортировка, она имеет ряд преимуществ:

* отсутствие потребности в памяти под стек;
* отсутствие деградации при неудачных наборах данных — быстрая сортировка легко деградирует до O(n²), что хуже, чем худшее гарантированное время для сортировки Шелла.

Рассмотрим пример:

Пусть дан массив и выполняется его сортировка методом Шелла, а в качестве значений d выбраны 5,3,1.

На первом шаге сортируются подмассивы A, составленные из всех элементов A, различающихся на 5 позиций, то есть подмассивы

В полученном массиве на втором шаге вновь сортируются подмассивы из отстоящих на 3 позиции элементов.

Процесс завершается обычной сортировкой вставками получившегося списка. На рисунке представлена схема для рассмотренного примера.

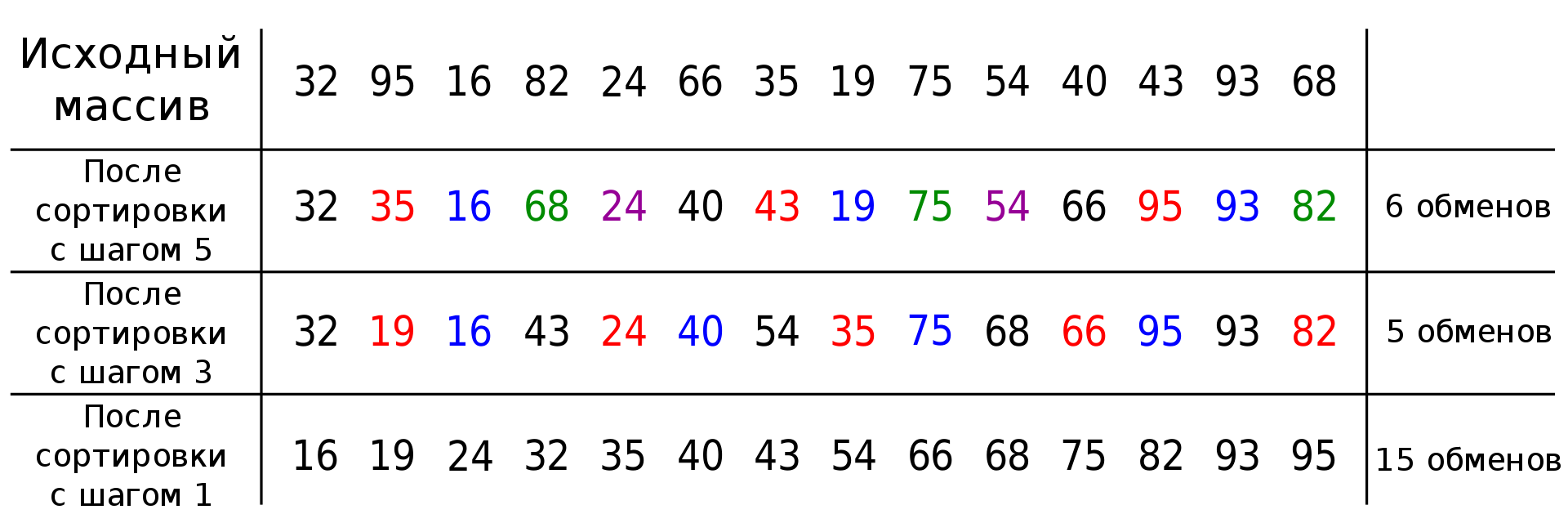


Рисунок 1. Сортировка Шелла

# Схема распараллеливания

Для распараллеливания сортировки используется слияние «Разделяй и властвуй». Идея слияния по алгоритму «Разделяй и властвуй» заключается в разбиении массивов на участки, которые можно слить независимо. В первом массиве выбирается центральный элемент x (он разбивает массив на две равные половины), а во втором массиве с помощью бинарного поиска находится позиция наибольшего элемента меньшего x (позиция этого элемента разбивает второй массив на две части). После такого разбиения первые и вторые половины массивов могут сливать независимо, т.к. в первых половинах находятся элементы меньшие элемента x, а во второй – большие (Рисунок 2). Для слияния двух массивов несколькими потоками можно в первом массиве выбрать несколько ведущих элементов, разделив его на равные порции, а во втором массиве найти соответствующие подмассивы. Каждый поток получит свои порции на обработку. Эффективность такого слияние во многом зависит от того, насколько равномерно произошло «разделение» второго массива.



Рисунок 2. Слияние "Разделяй и властвуй"

Массив данных разбивается на равные части. Если это невозможно, остаток распределяется равномерно между процессами. Каждый процесс сортирует свою часть независимо. Затем каждый второй процесс получает часть массива от предыдущего, бинарным поиском находит позицию наибольшего элемента меньшего первого элемента в полученной части массива, отправляет часть своего массив до элемента в данной позиции предыдущему. Процессы выполняют слияние. Далее каждый второй процесс отправляет предыдущему свою часть массива после слияния и завершает работу. Оставшиеся активные процессы объединяют свой массив и поученный от второго процесса в один. На следующем шаге взаимодействие происходит между оставшимися активными процессами. И так далее, пока активным не останется только корневой процесс.

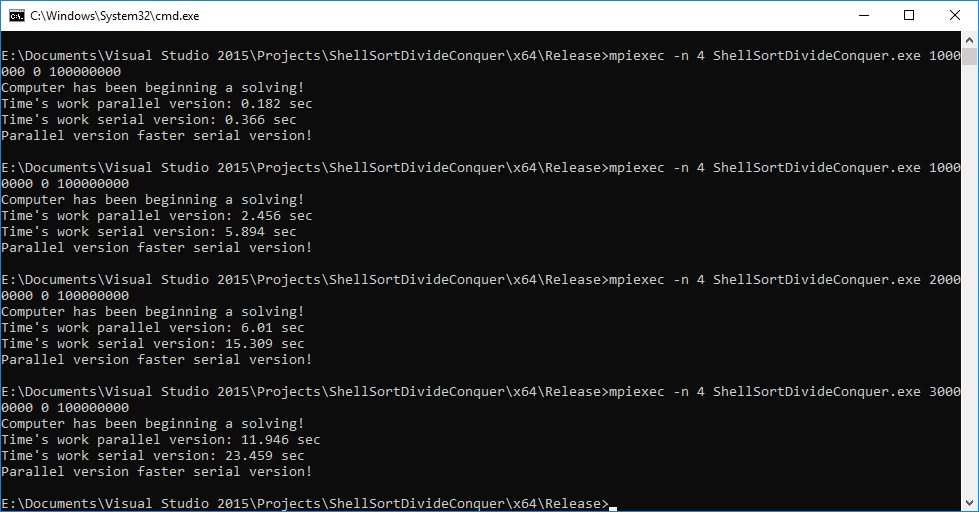
Для организации работы процессы выстраиваются в дерево обменов, которое представляет собой набор биномиальных деревьев. В представленной мной реализации процессы выстраиваются в правостороннее биномиальное дерево. В качестве обмена данными между процессами стоит понимать передачу одним процессом другому процессу своего массива и соответственный приём этого массива.

# Описание программной реализации

**Руководство пользователя**

Для запуска программы в консоли необходимо ввести следующую команду:

mpiexec –n <количество\_процессов> <путь\_до\_исполняемого\_файла> <размер\_массива> <начало\_диапазона> <конец\_диапазона>



**Руководство программиста**

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

Последовательность выполнения программы следующая:

* Генерируется массив из n случайных значений в некотором диапазоне типа double процессом с рангом ROOT, который выполняет последовательную версию сортировки массива и замеряет время работы алгоритма.
* Процесс ROOT сообщает всем другим процессам о размере массива. Для этого используем функцию MPI\_Bcast.
* Каждым процессом вычисляется массив работ и массив смещений. Массив работ определяется следующим образом: значение массива есть результат целочисленного деления размера массива на количество процессов, в случае если при делении получается остаток, то для каждого процесса, начиная с нулевого, к значению массива работ прибавляется 1 до тех пор, пока не закончится остаток.
* При помощи использования массивов работ и смещений, процесс с рангом ROOT рассылает соответствующие части исходного массива всем другим массивам и себе тоже. Здесь используется функция MPI\_Scatterv.
* Выполнение каждым процессом сортировки Шелла. На этом этапе происходит сортировка частей исходного массива, то есть каждый процесс сортирует свою часть массива.
* Выполнение слияния на основе дерева обменов, которое строится для процессов.
* Процесс ROOT выводит массив, который был получен им в результате выполнения слияния, также он сравнивает полученные результаты.

# Подтверждение корректности

Для подтверждения корректности в программе реализована функция сравнения буферов, полученных в результате последовательной и параллельной версий алгоритма. Данная функция вызывается после отработки обеих версий процессом с рангом ROOT. В случае, если мы не получаем одинаковые результаты, выводится сообщение: “The parallel and linear version not matching!”. В противном случае никаких сообщений в консоль не выводится (понимается, что никаких ошибок не возникло, результаты работы совпали).

# Результаты экспериментов

Одной из задач данной лабораторной работы было сравнение времени работы параллельной и последовательной версий сортировки одного и того же массива. В случае последовательной версии для сортировки массива используется алгоритм сортировки Шелла, сложность которого составляет на случайных данных.

Были проведены следующие эксперименты, на основании которых нужно сделать вывод об эффективности параллельной программы. Все результаты занесены в таблицы, также построены графики на основании результатов. Время вычисляется в секундах.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество процессов = 4 | n =100000 | n = 1000000 | n = 5000000 | n = 10000000 |
| Последовательная версия | 0.031 | 0.359 | 2.688 | 6.278 |
| Параллельная версия | 0.02 | 0.199 | 1.13 | 2.471 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество процессов = 2 | n =100000 | n = 1000000 | n = 5000000 | n = 10000000 |
| Последовательная версия | 0.025 | 0.345 | 2.404 | 5.715 |
| Параллельная версия | 0.015 | 0.222 | 1.475 | 3.35 |

На первом графике рассмотрим наглядно разницу во времени выполнения параллельной программы на 2-х и 4-х процессах соответственно. График строится на основе данных, занесенных в таблицы выше.

В результате экспериментов удалось добиться ускорения в 2,5 раза, что считается хорошим показателем.

# Заключение

Поставленная задача полностью выполнена. Были реализованы и протестированы последовательный и параллельный алгоритм сортировки Шелла. В параллельном алгоритме слияние выполнялось по принципу «Разделяй и властвуй». По данным экспериментов видно, что время работы сортировки в параллельной версии зависит от механизмов взаимодействия между процессами и от самой схемы распараллеливания т.е. имеет смысл применять параллельную версию данной сортировки при больших размерах массива.

# Приложение

#include <mpi.h>

#include <string>

#include <iostream>

#include <memory>

#include <ctime>

#include <limits>

#include <cassert>

#include <chrono>

#pragma warning (disable: 4703)

**void** **FillArray**(**double** \* pd2dArray, **size\_t** unSizeArr, **double** dBegin, **double** dEnd);

**void** **ShowArray**(**double** \* pdArray, **size\_t** unSizeArr, **int** ProcRank);

**void**\* **PointerOffset**(MPI\_Datatype type, **void**\* buf, **unsigned** **int** nOffset);

**int** **OlderBit**(**int** nNum);

**int** **NumberOlderBit**(**int** nNum);

**template** <**typename** Type>

**int** Shell\_Reduce(Type \*sendbuf, **int** \*sendcount, Type \*recvbuf, **int** sizearr, MPI\_Datatype type, **int** root, MPI\_Comm comm);

**template** <**typename** BaseType>

**void** ShellsSort(BaseType \*A, **unsigned** N);

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

{

**using** **namespace** std;

**using** **namespace** std::chrono;

**using** time = chrono::steady\_clock::time\_point;

**int** nSizeArr;

**int** ProcRank, ProcNum;

MPI\_Init(&argc, &argv);

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

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

**if** (ProcRank == **0**)

{

**if** (argc == **4**)

{

nSizeArr = atoi(argv[**1**]);

}

**else**

{

nSizeArr = **1000**;

}

}

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

**double** \* pdArr, \*pdRecvArr;

**double** \* pdRes = **new** **double**[nSizeArr] {**0**};

high\_resolution\_clock::time\_point startTime;

high\_resolution\_clock::time\_point endTime;

**double** dStartTime;

**if** (ProcRank == **0**)

{

pdArr = **new** **double**[nSizeArr];

**if** (argc == **5**)

FillArray(pdArr, nSizeArr, atof(argv[**2**]), atof(argv[**3**]));

**else**

FillArray(pdArr, nSizeArr, **0.f**, **10.f**);

dStartTime = MPI\_Wtime();

std::cout << "Computer has been beginning a solving!" << std::endl;

}

**else**

{

pdArr = nullptr;

}

**int** \* pnArrCounts = **new** **int**[ProcNum]; // 4 3 3

// 1 2 3 4 5 6 7 8 9 10

**int** \* pnArrDispls = **new** **int**[ProcNum]; // 0 4 7

**if** (ProcRank == **0**)

{

**int** nRecvSum = **0**;

**int** nRemSizePerProc = nSizeArr % ProcNum;

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

{

pnArrCounts[i] = nSizeArr / ProcNum ;

**if** (nRemSizePerProc > **0**)

{

pnArrCounts[i]++;

nRemSizePerProc--;

}

pnArrDispls[i] = nRecvSum;

nRecvSum += pnArrCounts[i];

startTime = chrono::high\_resolution\_clock::now();

}

}

MPI\_Bcast(pnArrCounts, ProcNum, MPI\_INT, **0**, MPI\_COMM\_WORLD);

MPI\_Bcast(pnArrDispls, ProcNum, MPI\_INT, **0**, MPI\_COMM\_WORLD);

pdRecvArr = **new** **double**[pnArrCounts[ProcRank]];

MPI\_Scatterv(pdArr, pnArrCounts, pnArrDispls, MPI\_DOUBLE, pdRecvArr, pnArrCounts[ProcRank], MPI\_DOUBLE, **0**, MPI\_COMM\_WORLD);

Shell\_Reduce(pdRecvArr, pnArrCounts, pdRes, nSizeArr, MPI\_DOUBLE, **0**, MPI\_COMM\_WORLD);

**if** (ProcRank == **0**)

{

endTime = chrono::high\_resolution\_clock::now();

**auto** WorkTimeParallel = chrono::duration\_cast<chrono::milliseconds>(endTime - startTime).count();

cout << "Time's work parallel version: " << (**double**)WorkTimeParallel/**1000** << " sec" << endl;

startTime = chrono::high\_resolution\_clock::now();

ShellsSort(pdArr, nSizeArr);

endTime = chrono::high\_resolution\_clock::now();

**auto** WorkTimeSerial = chrono::duration\_cast<chrono::milliseconds>(endTime - startTime).count();

cout << "Time's work serial version: " << (**double**)WorkTimeSerial/**1000** << " sec" << endl;

string sLess = (WorkTimeParallel < WorkTimeSerial ? "faster" : "slower");

cout << "Parallel version " << sLess << " serial version!" << std::endl;

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

{

**if** (pdArr[i] != pdRes[i])

{

cout << "Serial and parallel versions are not matching!" << endl;

**break**;

}

}

}

MPI\_Finalize();

**delete**[] pnArrCounts, pnArrDispls, pdRecvArr;

**delete**[] pdArr, pdRes;

**return** **0**;

}

**void** **FillArray**(**double** \* pd2dArray, **size\_t** unSizeArr, **double** dBegin, **double** dEnd)

{

srand(time(nullptr));

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

{

pd2dArray[i] = rand() % (**int**)(dEnd - dBegin) + dBegin + **double**(rand()) / **1000000.f**;

}

}

**void** **ShowArray**(**double** \* pdArray, **size\_t** unSizeArr, **int** ProcRank)

{

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

{

std::cout << "ProcRank " << ProcRank << " has a folowing arr[" << i << "] = " << pdArray[i] << std::endl;

}

}

**template**<**typename** Type>

**void** RealocArray(Type\*& pArr, **size\_t** nOldSize, **size\_t** nNewSize)

{

assert(nOldSize <= nNewSize);

Type\* pNewArr = **new** Type[nNewSize]{ **0** };

memcpy(pNewArr, pArr, nOldSize \* **sizeof**(Type));

**delete**[] pArr;

pArr = nullptr;

pArr = pNewArr;

}

**template** <**typename** BaseType>

**void** ShellsSort(BaseType \*A, **unsigned** N)

{

**unsigned** i, j, k;

BaseType t;

**for** (k = N / **2**; k > **0**; k /= **2**)

**for** (i = k; i < N; i++)

{

t = A[i];

**for** (j = i; j >= k; j -= k)

{

**if** (t < A[j - k])

A[j] = A[j - k];

**else**

**break**;

}

A[j] = t;

}

}

**template**<**typename** T>

**void** Merge(T\* mas1, T\* mas2, T\* tmp, **size\_t** size1, **size\_t** size2)

{

**int** a = **0**;

**int** b = **0**;

**int** i = **0**;

**while** ((a != size1) && (b != size2))

{

**if** (mas1[a] <= mas2[b])

{

tmp[i] = mas1[a];

a++;

}

**else**

{

tmp[i] = mas2[b];

b++;

}

i++;

}

**if** (a == size1)

{

**int** j = b;

**for** (; j<size2; j++, i++)

tmp[i] = mas2[j];

}

**else**

{

**int** j = a;

**for** (; j<size1; j++, i++)

tmp[i] = mas1[j];

}

}

**template**<**typename** T>

**void** MergeArrays(T\* mas1, T\* mas2, T\* tmp, **size\_t** size1, **size\_t** size2)

{

**int** nMedIndex = BinSearch(mas2, **0**, size2, (mas1[size1 / **2**]));

**int** nTmp1Size = nMedIndex + size1 / **2**;

**int** nTmp2Size = size1 + size2 - nTmp1Size;

T\* tmp1 = **new** T[nTmp1Size];

T\* tmp2 = **new** T[nTmp2Size];

Merge(mas1, mas2, tmp1, size1 / **2**, nMedIndex);

Merge(mas1 + size1 / **2**, mas2 + nMedIndex , tmp2, size1 - size1 / **2**, size2 - (nMedIndex));

memcpy(tmp, tmp1, nTmp1Size \* **sizeof**(T));

memcpy(tmp + nTmp1Size, tmp2, nTmp2Size \* **sizeof**(T));

**delete**[] tmp1, tmp2;

}

**template**<**typename** T>

**int** BinSearch(T \*mas, **int** l, **int** r, T x)

{

**if** (l == r)

**return** l;

**if** (l + **1** == r)

**if** (x < mas[l])

**return** l;

**else**

**return** r;

**int** m = (l + r) / **2**;

**if** (x < mas[m])

r = m;

**else**

**if** (x > mas[m])

l = m;

**else**

**return** m;

**return** **BinSearch**(mas, l, r, x);

}

**size\_t** TypeSize(MPI\_Datatype type)

{

**size\_t** result;

**switch** (type)

{

**default:**

std::cout << "Incorrect datatype" << std::endl;

exit(**1**);

**case** MPI\_CHAR:

result = **sizeof**(**char**);

**break**;

**case** MPI\_UNSIGNED\_CHAR:

result = **sizeof**(**unsigned** **char**);

**break**;

**case** MPI\_SHORT:

result = **sizeof**(**short**);

**break**;

**case** MPI\_UNSIGNED\_SHORT:

result = **sizeof**(**unsigned** **short**);

**break**;

**case** MPI\_INT:

result = **sizeof**(**int**);

**break**;

**case** MPI\_UNSIGNED:

result = **sizeof**(**unsigned** **int**);

**break**;

**case** MPI\_LONG:

result = **sizeof**(**long**);

**break**;

**case** MPI\_UNSIGNED\_LONG:

result = **sizeof**(**unsigned** **long**);

**break**;

**case** MPI\_LONG\_LONG\_INT:

result = **sizeof**(**long** **long**);

**break**;

**case** MPI\_FLOAT:

result = **sizeof**(**float**);

**break**;

**case** MPI\_DOUBLE:

result = **sizeof**(**double**);

**break**;

**case** MPI\_LONG\_DOUBLE:

result = **sizeof**(**long** **double**);

**break**;

}

**return** result;

}

**void**\* PointerOffset(MPI\_Datatype type, **void**\* buf, **unsigned** **int** nOffset)

{

**void**\* buffer;

**if** (!buf)

{

std::cout << "Pointer must not be nullptr!" << std::endl;

exit(**1**);

}

// проверить константность?

**switch** (type)

{

**default:**

std::cout << "Incorrect datatype" << std::endl;

exit(**1**);

**case** MPI\_CHAR:

buffer = **reinterpret\_cast**<**char**\*>(buf) + nOffset;

**break**;

**case** MPI\_UNSIGNED\_CHAR:

buffer = **reinterpret\_cast**<**unsigned** **char**\*>(buf) + nOffset;

**break**;

**case** MPI\_SHORT:

buffer = **reinterpret\_cast**<**short**\*>(buf) + nOffset;

**break**;

**case** MPI\_UNSIGNED\_SHORT:

buffer = **reinterpret\_cast**<**unsigned** **short**\*>(buf) + nOffset;

**break**;

**case** MPI\_INT:

buffer = **reinterpret\_cast**<**int**\*>(buf) + nOffset;

**break**;

**case** MPI\_UNSIGNED:

buffer = **reinterpret\_cast**<**unsigned** **int**\*>(buf) + nOffset;

**break**;

**case** MPI\_LONG:

buffer = **reinterpret\_cast**<**long**\*>(buf) + nOffset;

**break**;

**case** MPI\_UNSIGNED\_LONG:

buffer = **reinterpret\_cast**<**unsigned** **long**\*>(buf) + nOffset;

**break**;

**case** MPI\_LONG\_LONG\_INT:

buffer = **reinterpret\_cast**<**long** **long** **int**\*>(buf) + nOffset;

**break**;

**case** MPI\_FLOAT:

buffer = **reinterpret\_cast**<**float**\*>(buf) + nOffset;

**break**;

**case** MPI\_DOUBLE:

buffer = **reinterpret\_cast**<**double**\*>(buf) + nOffset;

**break**;

**case** MPI\_LONG\_DOUBLE:

buffer = **reinterpret\_cast**<**long** **double**\*>(buf) + nOffset;

**break**;

}

**return** buffer;

}

**int** OlderBit(**int** nNum)

{

**int** nTmp = **1** << **30**;

**while** (nNum < nTmp)

{

nTmp >>= **1**;

}

**return** nTmp;

}

**int** NumberOlderBit(**int** nNum)

{

**int** nTmp = **1** << **30**;

**int** nCount = **31**;

**while** (nNum < nTmp)

{

nTmp >>= **1**;

nCount--;

}

**return** nCount;

}

**template** <**typename** Type>

**int** Shell\_Reduce(Type \*sendbuf, **int** \*sendcount, Type \*recvbuf, **int** sizearr, MPI\_Datatype type, **int** root, MPI\_Comm comm)

{

MPI\_Status status;

**int** ProcRank, ProcNum;

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

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

**int** nRankSender{ **0** };

Type \* pvResBuf = **new** Type[sizearr];

**int** nFinish = std::log(ProcNum) / std::log(**2**);

**if** (ProcNum % **2** > **0**)

nFinish++;

**int** nSendArrSize = sendcount[ProcRank];

**int** nRecvArrSize = **0**;

ShellsSort((**double**\*)sendbuf, sendcount[ProcRank]);

**int** newProcRank = (ProcRank < root ? ProcRank + **1** : ProcRank);

**int** nStart = (ProcRank == root ? **0** : NumberOlderBit(newProcRank));

**for** (**int** i = nFinish; i >= nStart; i--)

{

**if** (ProcRank == root)

{

nRankSender = **1** << i;

nRankSender <= root ? nRankSender-- : nRankSender;

}

**else**

{

nRankSender = (**1** << i) + newProcRank;

**if** (nRankSender <= root)

nRankSender--;

}

**if** (nRankSender < ProcNum)

{

MPI\_Recv(&nRecvArrSize, **1**, MPI\_INT, nRankSender, **0**, comm, &status);

MPI\_Recv(recvbuf, nRecvArrSize, type, nRankSender, **0**, comm, &status);

MergeArrays(sendbuf, recvbuf, pvResBuf, nSendArrSize, nRecvArrSize);

RealocArray(sendbuf, nSendArrSize, nSendArrSize + nRecvArrSize);

nSendArrSize += nRecvArrSize;

memcpy(sendbuf, pvResBuf, nSendArrSize \* **sizeof**(Type));

}

}

**if** (ProcRank != root)

{

**int** oldnum = OlderBit(newProcRank);

**int** nRankReciever = newProcRank & (~oldnum);

**if** (nRankReciever == **0**)

nRankReciever = root;

**else** **if** (nRankReciever <= root)

nRankReciever--;

MPI\_Send(&nSendArrSize, **1**, MPI\_INT, nRankReciever, **0**, comm);

MPI\_Send(sendbuf, nSendArrSize, type, nRankReciever, **0**, comm);

}

**if** (ProcRank == root)

{

memcpy(recvbuf, sendbuf, sizearr\***sizeof**(Type));

}

**delete**[] pvResBuf;

**return** **0**;

}