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 с чётно-нечётным слиянием Бэтчера»**

**Выполнил:**

студент группы 381606-1

Батова Д.А.

**Проверил:**

Кустикова В.Д.

Нижний Новгород

2018

**Содержание**

[Постановка задачи 3](#_Toc531367741)

[Метод решения 4](#_Toc531367742)

[Схема распараллеливания 5](#_Toc531367743)

[Описание программной реализации 8](#_Toc531367744)

[Подтверждение корректности 10](#_Toc531367745)

[Результаты экспериментов 12](#_Toc531367746)

[Заключение 15](#_Toc531367747)

[Приложение 16](#_Toc531367748)

# Постановка задачи

Основная задача лабораторной работы ставится следующим образом:

Дан неупорядоченный массив некоторой длины n. Требуется реализовать параллельную версию сортировки данного массива, причём каждый процесс, участвующий в параллельной программе, должен отсортировать полученную им часть массива, используя сортировку Шелла, а слияние отсортированных на каждом процессе массивов в один полностью отсортированный массив, то есть в результирующий, должно основываться на чётно-нечётном слиянии Бэтчера.

В процессе разработки программы также необходимо выполнить следующие подзадачи:

* разработать последовательную версию алгоритма сортировки массива для проверки корректности работы параллельного алгоритма;
* сравнить время работы последовательного и параллельного алгоритмов сортировки при различных размерах входного массива;
* сделать вывод об эффективности работы параллельной программы.

Лабораторная работа считается выполненной, если представленные выше требования программно реализованы.

# Метод решения

Описание алгоритма «Сортировка Шелла».

Этот метод сортировки Д. Шелл предложил в 1959 г. Он использует минимум памяти и показывает высокие скорости при сортировке. По сути в методе Шелла применяются сравнения и перестановки элементов аналогичные методу вставок, но при этом порядок сравниваемых элементов совершенно другой.

Идея **сортировки методом Шелла** состоит в том, чтобы сортировать элементы, отстоящие друг от друга на некотором расстоянии ***step***. Затем сортировка повторяется при меньших значениях ***step***, и в конце процесс сортировки Шелла завершается при ***step = 1*** (а именно обычной сортировкой вставками).

До сих пор продолжает обсуждаться вопрос выбора шага сортировки ***step***. Шелл предложил такую последовательность:  …, где – количество элементов в сортируемом массиве.

Далее, на примере последовательности целых чисел, показан процесс сортировки массива методом Шелла. Для удобства и наглядности, элементы одной группы выделены одинаковым цветом.

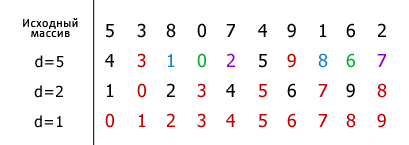


Рисунок 1 Пример сортировки Шелла

Первое значение, соответствующее расстоянию **step** равно **10/2=5**. На каждом шаге оно уменьшается вдвое. Элементы, входящие в одну группу, сравниваются и, если значение какого-либо элемента, стоящего левее того с которым он сравнивается, оказывается больше (сортировка по возрастанию), тогда они меняются местами. Так, элементы путем внутригрупповых перестановок постепенно становятся на свои позиции, и на последнем шаге (**step=1**) сортировка сводится к проходу по одной группе, включающей в себя все **N** элементов массива. При этом число требуемых обменов оказывается совсем небольшим.

# Схема распараллеливания

Для распараллеливания сортировки используется чётно-нечётное слияние Бэтчера. Идея параллельной реализации с использованием такого типа слияния заключается в выполнении следующих шагов:

* Каждый процесс отсортирует свою часть исходного массива.
* После сортировки частей массива выполняется слияние этих частей.

В первом пункте в качестве сортировки частей используется сортировка Шелла. Идея алгоритма описана в предыдущем разделе отчёта.

Во втором пункте используется алгоритм чётно-нечётного слияния Бэтчера. Идея слияния 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, стоящих на чётной и нечётной позициях. Первый и последний элементы массива проверять не надо, так как они являются минимальным и максимальным элементами массивов.

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

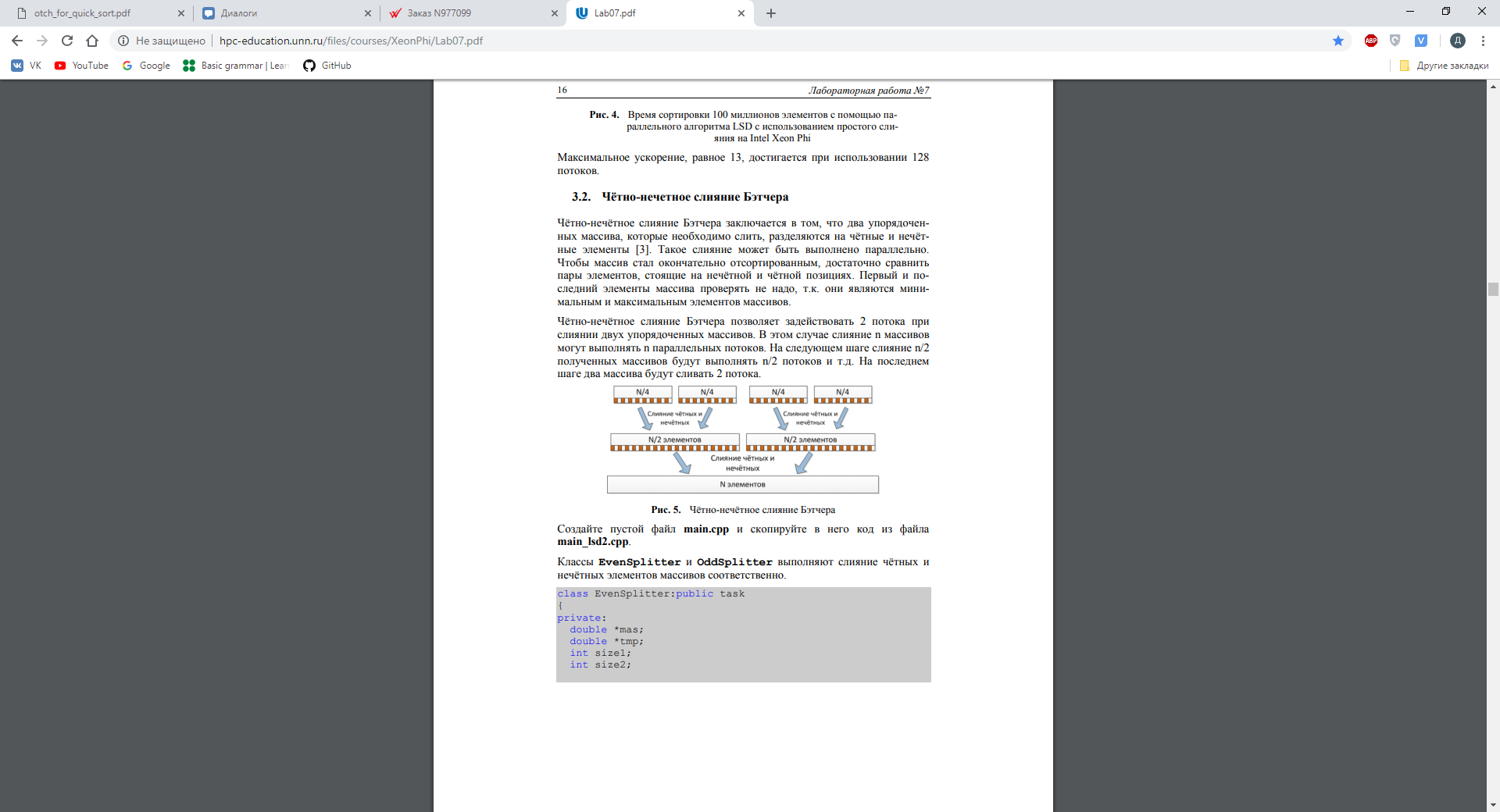


Рисунок 2 Чётно-нечётное слияние Бэтчера

Алгоритм слияния выглядит следующим образом:

1. Один из процессов отправляет другому свой массив.
2. Выполняется слияние массивов на одном из процессов, который получил массив от другого процесса.

Слияние основывается на отправке одним процессом другому процессу своего буфера и соответственным получением этого буфера вторым процессом. Для организации работы алгоритма процессы выстраиваются в дерево обменов, которое представляет собой совокупность биномиальных деревьев, степени которых определяются номерами значащих битов в двоичном представлении количества процессов, участвующих в операции. Степени всех деревьев различны. Каждому узлу дерева соответствует номер процесса. Корнем дерева обмена является корень биномиального дерева наибольшей степени. Распределение процессов по дереву происходит согласно прямому обходу всего дерева. Таким образом, если полностью отсортированный массив должен собираться на нулевом процессе, то узлам дерева обмена присваиваются номера (– число процессов). Если же отсортированный массив должен собраться на каком-то другом процессе, то нумерация процессов смещается на позиций относительно 0: Поэтому необходимо для каждого процесса вычислить относительный номер – номер процесса относительно корневого процесса.

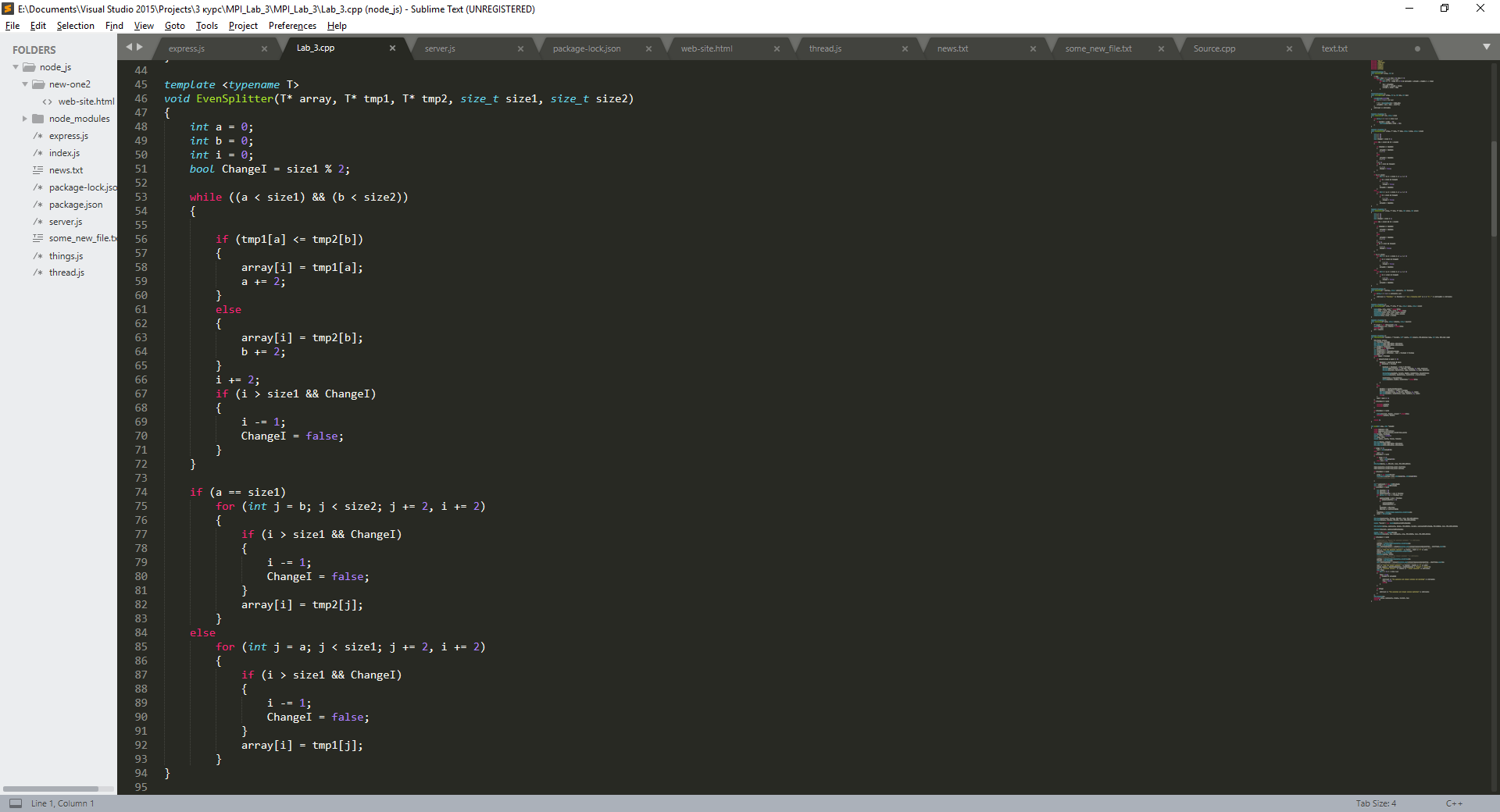
Каждый процесс выполняет цикл операций, число которых определяется двоичным логарифмом от количества процессов. На каждой итерации цикла процесс передаёт или получает данные. Правила отправки и передачи данных устанавливаются следующим образом:

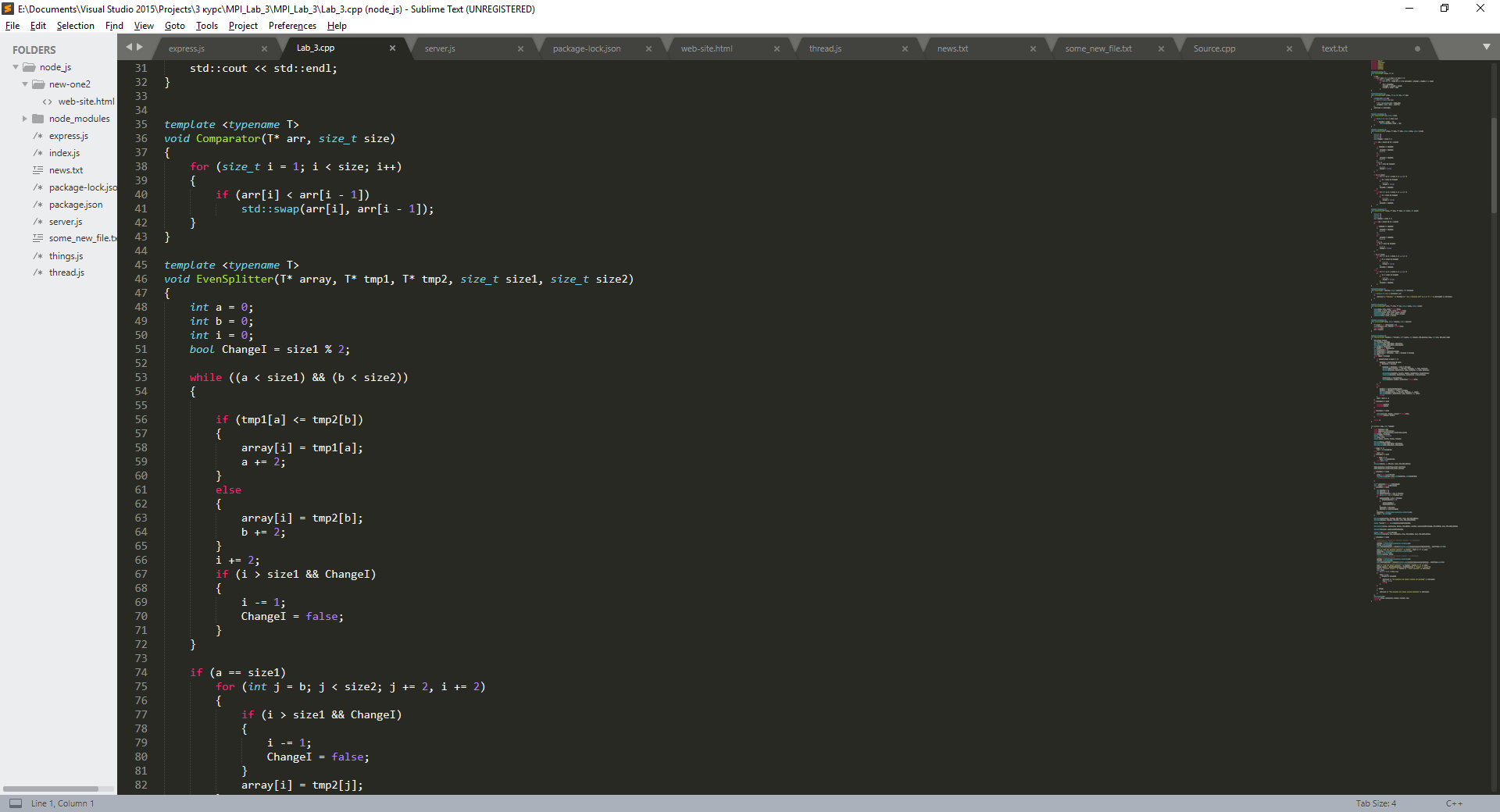
* Если у процесса i-й бит установлен, то он передаёт данные процессу, у которого этот бит сброшен.
* Если у процесса i-й бит сброшен, то он получает данные от процесса, у которого этот бит установлен.

В результате буфером на корневом процессе будет являться полностью отсортированный массив, который нужно было получить. Сортировка – это коммутативная операция, таким образом, в качестве корня мы можем использовать любой процесс и сливать процессы в любом порядке, ведь порядок никак не повлияет на результат всей операции сортировки.

Помимо построения дерева процессов, которое необходимо для организации передачи процессами массивов, нужно реализовать слияние массивов, имеющихся на отправляемом и получаемом процессах. Результат слияния становится новым массивом получающего процесса. Отправляющий массив в операциях слияния больше не участвует.

Функции EvevSplitter и OddSplitter отвечают за слияние двух приходящих от разных процессов массивов в один результирующий массив. Первая сравнивает элементы приходящих массивов на чётных позициях, вторая – на нечётных позициях. В результате работы этих функций полученный массив является частично отсортированным, поскольку в нём отсортированы отдельно чётные элементы и отдельно нечётные. Следующим шагом является сортировка всего результирующего массива, то есть мы сравниваем между собой пары чётных и нечётных элементов, получая полностью отсортированный массив. За это отвечает функция Comparator.





# Описание программной реализации

**Руководство пользователя**

Для запуска программы в консоли необходимо ввести следующую команду:

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

Значения массива генерируются в некотором диапазоне, который указывает пользователь. Для тестирования предлагается использовать значения типа double, однако программа рассчитана и на другие типы данных.

Пользователь в праве указать номер процесса, на который придёт результат в качестве отсортированного массива.

В результате запуска в консоли выводятся:

* Время выполнения последовательной сортировки сгенерированного массива. В этом случаем массив сортируется выбранным пользователем процессом с помощью алгоритма сортировки Шелла.
* Время выполнения параллельной сортировки сгенерированного массива. Части массива на каждом из процессов сортируются с помощью алгоритма сортировки Шелла, а потом сливаются в один с помощью чётно-нечётного слияния Бэтчера.
* Вывод о том, совпадают ли массивы, полученные в результате последовательной и параллельной сортировок.

**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 (указывается пользователем), который выполняет последовательную версию сортировки массива и замеряет время работы алгоритма (он также замеряет время работы параллельной версии 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 чётно-нечётного слияния Бэтчера 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 (сравнение буферов, полученных в последовательной и параллельной версиях алгоритма).

# 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. Он указывается пользователем. В случае, если мы не получаем одинаковые результаты, выводится сообщение: “The parallel and linear version not matching!”. В случае, если массивы в обоих случаях совпадают, выводится сообщение: “The parallel and linear version matching!”.

Чтобы продемонстрировать корректность, запустим параллельную программу, рассматривая случай, когда количество процессов равно 4, размер массива равен 10 (специально берём маленький размер, чтобы можно было вывести результаты). Результат работы программы представлен на рис. 3. В данном случае в консоли выводятся массивы, полученные в параллельной версии и в линейной версии. Поскольку размер массива мал, можно сделать вывод о правильности работы программы, ведь массивы совпали.

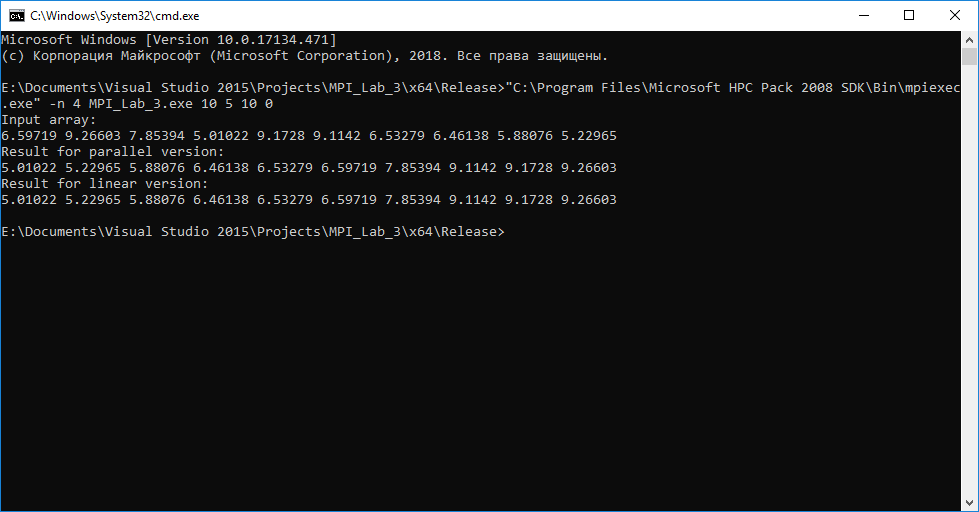


Рисунок 3 Вывод результатов

Однако, опыты могут проводиться на достаточно больших размерах массива, что делает вывод результатов на консоль неразумным. Поэтому в таких случаях в программе поэлементно сравниваются массивы параллельной и последовательной версии с дальнейшим выводом на консоли сообщении о совпадении или же несовпадении.

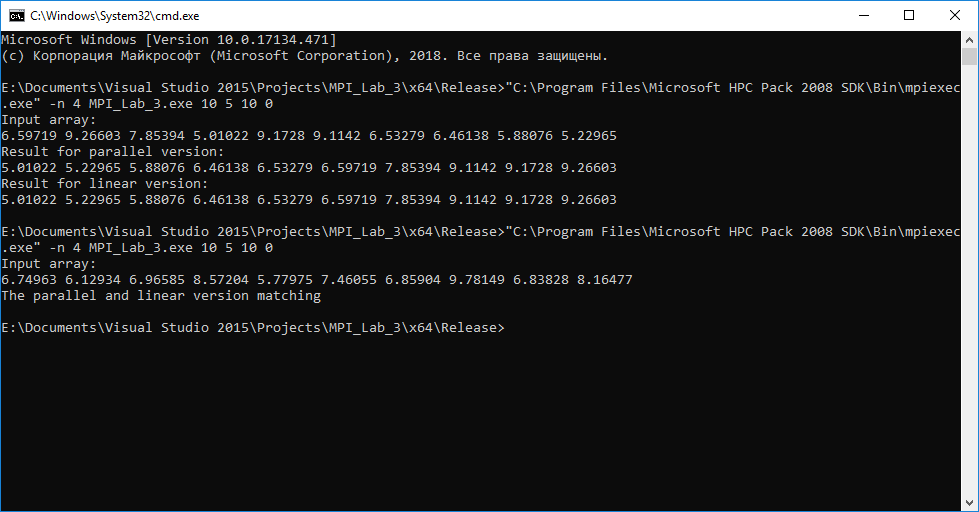


Рисунок 4 Вывод сообщения о корректности

# 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 | 0.031844 | 0.421227 | 3.05718 | 6.66787 |
| Параллельная версия | 0.037427 | 0.325554 | 1.56689 | 3.13262 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество процессов = 2 | n =100000 | n = 1000000 | n = 5000000 | n = 10000000 |
| Последовательная версия | 0.0251695 | 0.372163 | 2.50584 | 6.29431 |
| Параллельная версия | 0.0197368 | 0.238977 | 1.51998 | 3.48946 |

На первом графике рассмотрим наглядно разницу во времени выполнения параллельной программы на 2-х и 4-х процессах соответственно. (Больше 4-х процессов рассматривать не имеет смысла, поскольку машина 4-х ядерная, а значит, при больших значениях программа не будет по-настоящему параллельно выполняться). График строится на основе данных, занесенных в таблицы выше.

Чтобы сделать вывод об эффективности параллельной программы будем использовать такое понятие, как ускорение. Ускорение – отношение времени выполнения последовательного алгоритма к времени выполнения параллельного алгоритма.

Как видно из графиков, на исходных данных размером от 100000 до 2000000 параллельная версия алгоритма на 2-х процессах работает с большим ускорением, чем аналогичная версия на 4-х процессах. Это связано с тем, что больше времени уходит на взаимодействие между процессами, то есть на операции слияния, нежели чем на сортировку массивов. С ростом размера массива ситуация изменяется, поскольку теперь больше времени уходит на сортировку массивов, чем на взаимодействие процессов, таким образом, большее ускорение получается на 4-х процессах по сравнению с версией, отработанной на 2-х процессах.

# Заключение

Поставленная задача полностью выполнена. Были реализованы и протестированы последовательный и параллельный алгоритм сортировки Шелла. В параллельном алгоритме использовалось чётно-нечётное слияние Бэтчера. 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 <iostream>

#include <time.h>

#include <chrono>

#include <limits>

#include <string>

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

**void** ShellSort(T\* array, **int** n)

{

T tmp;

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

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

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

{

tmp = array[j];

array[j] = array[j + step];

array[j + step] = tmp;

}

}

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

**void** FillingArray(T\* array, **int** n, **int** min, **int** max)

{

srand(time(nullptr));

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

{

T ri = (**double**)rand() / RAND\_MAX;

array[i] = min + (max - min)\*ri;

}

std::cout << std::endl;

}

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

**void** Comparator(T\* arr, **size\_t** size)

{

**for** (**size\_t** i = **1**; i < size; i++)

{

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

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

}

}

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

**void** EvenSplitter(T\* array, T\* tmp1, T\* tmp2, **size\_t** size1, **size\_t** size2)

{

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

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

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

**bool** ChangeI = size1 % **2**;

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

{

**if** (tmp1[a] <= tmp2[b])

{

array[i] = tmp1[a];

a += **2**;

}

**else**

{

array[i] = tmp2[b];

b += **2**;

}

i += **2**;

**if** (i > size1 && ChangeI)

{

i -= **1**;

ChangeI = false;

}

}

**if** (a == size1)

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

{

**if** (i > size1 && ChangeI)

{

i -= **1**;

ChangeI = false;

}

array[i] = tmp2[j];

}

**else**

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

{

**if** (i > size1 && ChangeI)

{

i -= **1**;

ChangeI = false;

}

array[i] = tmp1[j];

}

}

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

**void** OddSplitter(T\* array, T\* tmp1, T\* tmp2, **int** size1, **int** size2)

{

**int** a = **1**;

**int** b = **1**;

**int** i = **1**;

**bool** ChangeI = size1 % **2**;

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

{

**if** (tmp1[a] <= tmp2[b])

{

array[i] = tmp1[a];

a += **2**;

}

**else**

{

array[i] = tmp2[b];

b += **2**;

}

i += **2**;

**if** (i >= size1 && ChangeI)

{

i += **1**;

ChangeI = false;

}

}

**if** (a == size1)

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

{

**if** (i >= size1 && ChangeI)

{

i += **1**;

ChangeI = false;

}

array[i] = tmp2[j];

}

**else**

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

{

**if** (i >= size1 && ChangeI)

{

i += **1**;

ChangeI = false;

}

array[i] = tmp1[j];

}

}

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

**void** BatcherMerge(T\* arr1, T\* arr2, T\* res, **size\_t** size1, **size\_t** size2)

{

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

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

EvenSplitter(res, arr1, arr2, size1, size2);

OddSplitter(res, arr1, arr2, size1, size2);

Comparator(res, size1 + size2);

}

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

**void** reallocate(T\* &arr, **size\_t** oldsize, **size\_t** newsize)

{

T\* newarr = **new** T[newsize]{ **0** };

memcpy(newarr, arr, oldsize \* **sizeof**(T));

**delete**[] arr;

arr = newarr;

}

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

**int** TREE\_Betcher(T \*sendbuf, T \*recvbuf, **int**\* counts, **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** RankRecv, RankSend;

T \* ResBuf = **new** T[sizearr];

**int** RecvArrSize = **0**;

**int** SendArrSize = counts[ProcRank];

**int** newProcRank = (ProcRank - root + ProcNum) % ProcNum;

**int** mask = **1**;

**while** (mask < ProcNum)

{

**if** ((newProcRank & mask) == **0**)

{

RankSend = newProcRank | mask;

**if** (RankSend < ProcNum)

{

RankSend = (RankSend + root) % ProcNum;

MPI\_Recv(&RecvArrSize, **1**, MPI\_INT, RankSend, **0**, comm, &status);

MPI\_Recv(recvbuf, RecvArrSize, type, RankSend, **0**, comm, &status);

BatcherMerge(sendbuf, recvbuf, ResBuf, SendArrSize, RecvArrSize);

reallocate(sendbuf, SendArrSize, SendArrSize + RecvArrSize);

SendArrSize += RecvArrSize;

memcpy(sendbuf, ResBuf, SendArrSize \* **sizeof**(T));

}

}

**else**

{

RankRecv = newProcRank&(~mask);

RankRecv = (RankRecv + root) % ProcNum;

MPI\_Send(&SendArrSize, **1**, MPI\_INT, RankRecv, **0**, comm);

MPI\_Send(sendbuf, SendArrSize, type, RankRecv, **0**, comm);

**break**;

}

mask = mask << **1**;

}

**if** (ProcRank != root)

{

**delete**[] sendbuf;

**delete**[] ResBuf;

}

**if** (ProcRank == root)

{

memcpy(recvbuf, ResBuf, sizearr \* **sizeof**(T));

**delete**[] sendbuf, ResBuf;

}

**return** **0**;

}

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

{

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

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

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

**int** ProcNum, ProcRank;

**double** \*array = nullptr;

**int** size, root;

**double** start, start1, finish, finish1;

MPI\_Init(&argc, &argv);

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

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

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

root = atoi(argv[**4**]);

**else**

root = **0**;

**if** (ProcRank == root)

{

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

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

**else** size = **10**;

}

MPI\_Bcast(&size, **1**, MPI\_INT, root, MPI\_COMM\_WORLD);

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

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

**if** (ProcRank == root)

{

array = **new** **double**[size];

FillingArray(array, size, atoi(argv[**2**]), atoi(argv[**3**]));

//ShowArray(array, size);

}

**int** \* sendcounts = **new** **int**[ProcNum];

**int** \* displs = **new** **int**[ProcNum];

**if** (ProcRank == root)

{

**int** nSendSum = **0**;

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

**int** nRemSizePerProc = size % ProcNum;

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

{

sendcounts[i] = size / ProcNum;

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

{

sendcounts[i]++;

nRemSizePerProc--;

}

displs[i] = nRecvSum;

nRecvSum += sendcounts[i];

}

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

start = MPI\_Wtime();

}

MPI\_Bcast(sendcounts, ProcNum, MPI\_INT, root, MPI\_COMM\_WORLD);

MPI\_Bcast(displs, ProcNum, MPI\_INT, root, MPI\_COMM\_WORLD);

**double** \*recvbuf = **new** **double**[sendcounts[ProcRank]];

MPI\_Scatterv(array, sendcounts, displs, MPI\_DOUBLE, recvbuf, sendcounts[ProcRank], MPI\_DOUBLE, root, MPI\_COMM\_WORLD);

ShellSort(recvbuf, sendcounts[ProcRank]);

**double** \* res = **new** **double**[size];

TREE\_Betcher(recvbuf, res, sendcounts, size, MPI\_DOUBLE, root, MPI\_COMM\_WORLD);

**if** (ProcRank == root)

{

//std::cout << "Result for parallel version: " << std::endl;

//ShowArray(res, size);

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

finish = MPI\_Wtime();

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

//cout << "Time for parallel version: " << WorkTimeParallel << "ns" << endl;

cout << "Time for parallel version: " << finish - start << "s" << endl;

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

start1 = MPI\_Wtime();

ShellSort(array, size);

//std::cout << "Result for linear version: " << std::endl;

//ShowArray(array, size);

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

finish1 = MPI\_Wtime();

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

//cout << "Time for linear version: " << WorkTimeSerial << "ns" << endl;

cout << "Time for linear version: " << finish1 - start1 << "s" << endl;

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

cout << "Parallel version " << compare << " linear version!" << std::endl;

**bool** flag;

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

{

flag = true;

**if** (res[i] != array[i])

{

std::cout << "The parallel and linear version not matching" << std::endl;

flag = false;

**break**;

}

}

**if** (flag)

{

std::cout << "The parallel and linear version matching" << std::endl;

}

}

MPI\_Finalize();

**delete**[] array, sendcounts, displs, recvbuf, res;

**return** **0**;

}