

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

«Поразрядная сортировка с простым слиянием для вещественных чисел»

Выполнил:

Студент группы
381606-1

_____ Толстиков М.В.

Проверил:

_____ Сысоев А.В.

Нижний Новгород

2019

Оглавление

Постановка задачи	3
Метод решения.....	3
Схема распараллеливания.....	5
Single and parallel sorting with OpenMP	6
Parallel sorting with TBB	10
Заключение	13
Литература.....	13
Приложение	14

Постановка задачи

Необходимо реализовать линейную и параллельную поразрядную сортировку вещественных чисел с простым слиянием с помощью библиотек OpenMP и TBB на языке C++, сравнить время выполнения на разном количестве запускаемых процессах, посчитать ускорение и эффективность.

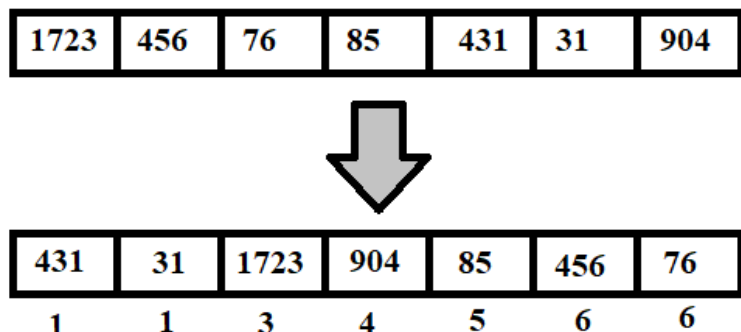
Метод решения

Поразрядная сортировка имеет две модификации: Most Significant Digit (MSD) и Least Significant Digit (LSD). В данной работе будет рассмотрен алгоритм LSD как наиболее эффективный. Идея поразрядной восходящей сортировки (Least Significant Digit (LSD) Radix Sort) заключается в том, что выполняется последовательная сортировка чисел по разрядам (от младшего разряда к старшему). Рассмотрим алгоритм работы сортировки на примере массива десятичных чисел из 7 элементов.

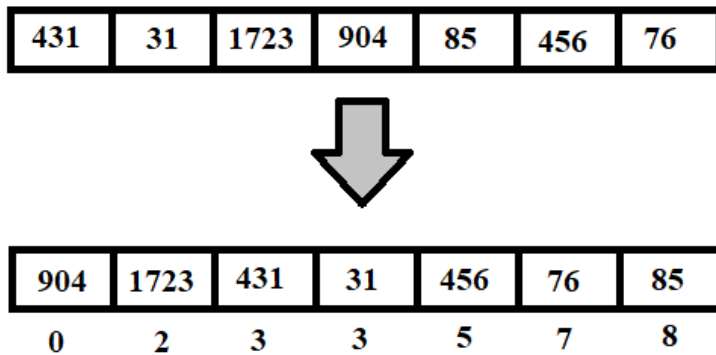
Пример массива:

1723	456	76	85	431	31	904
------	-----	----	----	-----	----	-----

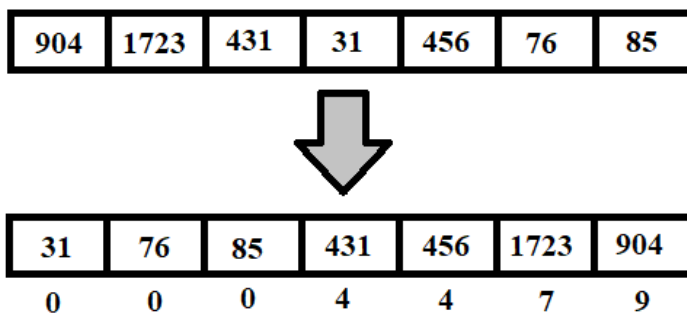
На первой итерации сортировки выполняется размещение элементов в отсортированном порядке по младшему разряду чисел:



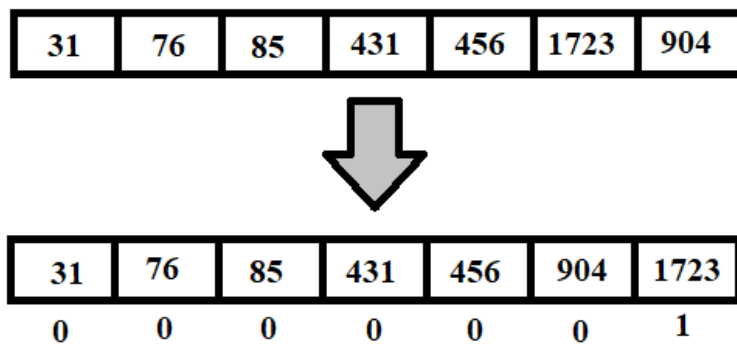
На следующей итерации сортируются элементы по второму разряду:



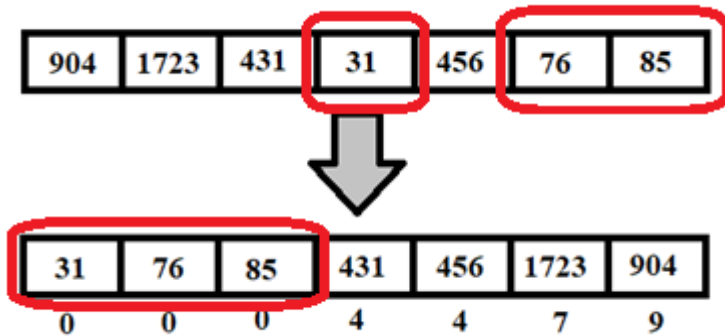
Далее выполняется упорядочивание элементов по третьему разряду:



На последнем шаге выполняется сортировка по старшему разряду:



Поразрядная сортировка массива будет работать только в том случае, если сортировка, выполняющаяся по разряду, является устойчивой (элементы равных разрядов не будут менять взаимного расположения при сортировке по очередному разряду):

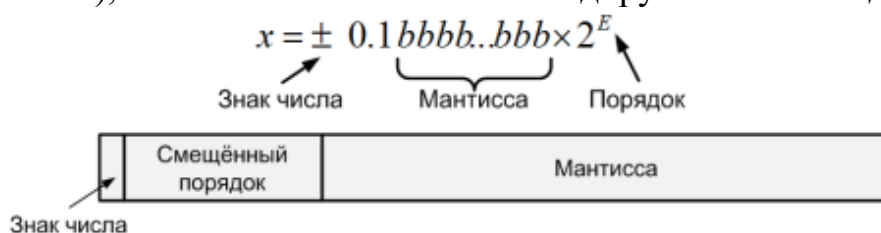


Современные процессоры предназначены для обработки данных, которые представлены в битах и байтах, поэтому выполнять сортировку по десятичным разрядам чисел не эффективно. Рассмотрим побайтовую реализацию поразрядной сортировки, т.е. будем рассматривать число как набор 256-значных цифр. В таком случае для сортировки разряда будет удобно использовать сортировку подсчетом (с модификацией, которая не будет менять взаимного расположения элементов равных разрядов).

Сортировка подсчетом по i -му байту будет проходить в два прохода по исходному массиву:

- При первом проходе по исходному массиву выполняется подсчет i -ых байт в массиве `mas`, результат будет сохранен в массив подсчетов `counter` из 256 элементов;
- В массив `offset` на основании посчитанных данных выполняется подсчет смещений, по которым будут сохраняться элементы: **`offset[0]=0` для всех j от 1 до 255 `offset[j]=counter[j-1]+offset[j-1]`**
- При втором проходе по исходному массиву `mas` выполняется копирование элемента во вспомогательный массив по соответствующему индексу в массиве смещений `offset` и выполняется инкремент смещения.

Вещественный тип представляется в памяти так (рассмотрим 8-байтовый тип double), что число кодируется следующим образом:



Поразрядная сортировка таких чисел будет выполняться в 8 проходов, начиная от младшего байта до старшего.

Схема распараллеливания

Можно выделить два подхода к реализации параллельного алгоритма сортировки: внутренняя реализация параллельного алгоритма или внешнее распараллеливание за счет слияния отсортированных частей.

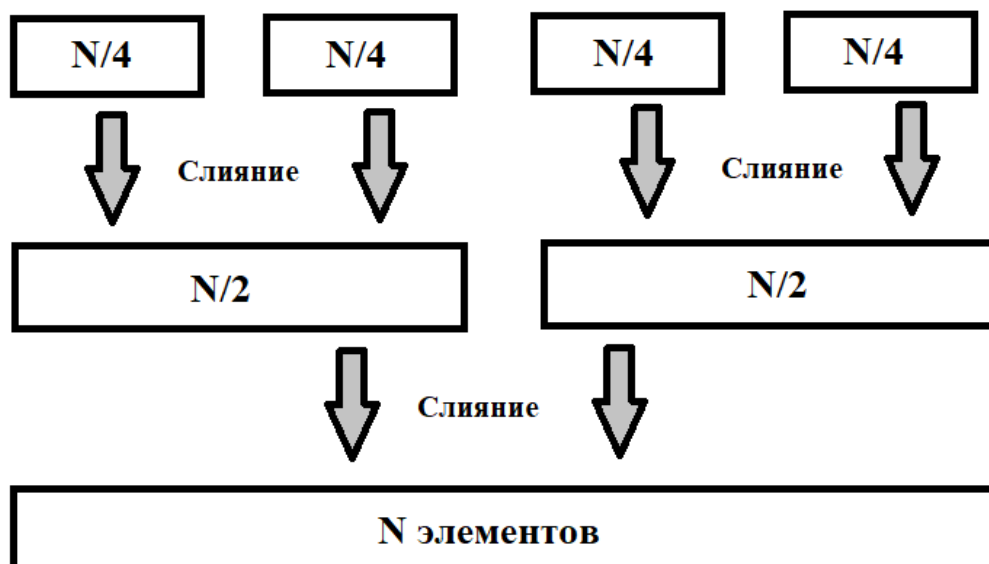
Алгоритм побайтовой восходящей сортировки заключается в последовательном применении сортировки подсчетом к каждому байту чисел, поэтому параллельно может выполняться только сортировка подсчетом для каждого байта. При этом параллельная версия сортировки подсчетом должна сохраняться свойство устойчивости сортировки (а это тяжелая задача).

Идея параллельной реализации с использованием слияния заключается в выполнении следующих шагов:

1. Сортировка частей массива.
2. Слияние отсортированных частей массива.

Сортировка частей массива может выполняться без каких-либо синхронизаций, поэтому теоретическое ускорение этого этапа является линейным. Этап слияние в зависимости от алгоритма может иметь различную эффективность.

Идея простого слияния заключается в том, что один поток может выполнять слияние двух отсортированных массивов по классическому алгоритму. В этом случае слияние n массивов могут выполнять $n/2$ параллельных потоков. На следующем шаге слияние $n/2$ полученных массивов будут выполнять $n/4$ потоков и т.д. Таким образом, последнее слияние будет выполнять один поток, а учитывая, что сортировка частей массива имеет линейную трудоемкость, то слияние вносит существенный вклад во время работы алгоритма.



Single and parallel sorting with OpenMP

В первом приложении я тестирую последовательную сортировку вместе с параллельной сортировкой написанной с помощью библиотеки OpenMP.

Здесь я сравниваю время работы, ускорение и эффективность сортировок, а так же проверяю их на корректность.

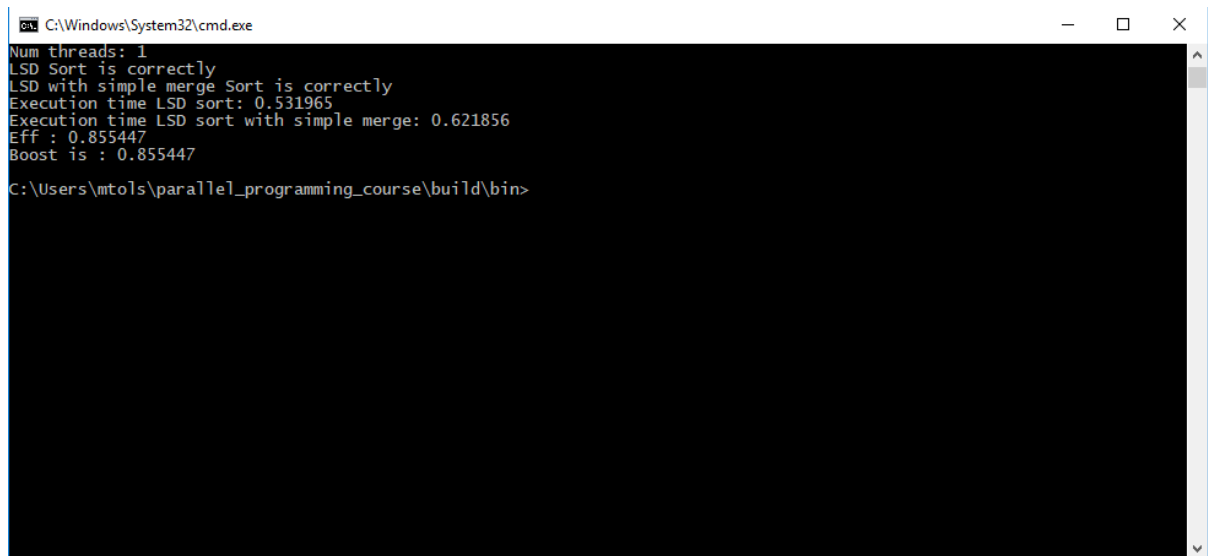
Пояснение каждой строки вывода:

1. Количество потоков
2. Проверка корректности последовательной сортировки
3. Проверка корректности параллельной сортировки
4. Время выполнения последовательной сортировки
5. Время выполнения параллельной сортировки
6. Эффективность
7. Ускорение

Программа запускается на компьютере с операционной системой Windows 10, процессором Intel Core i5-6200U и 4-мя ГБ оперативной памяти.

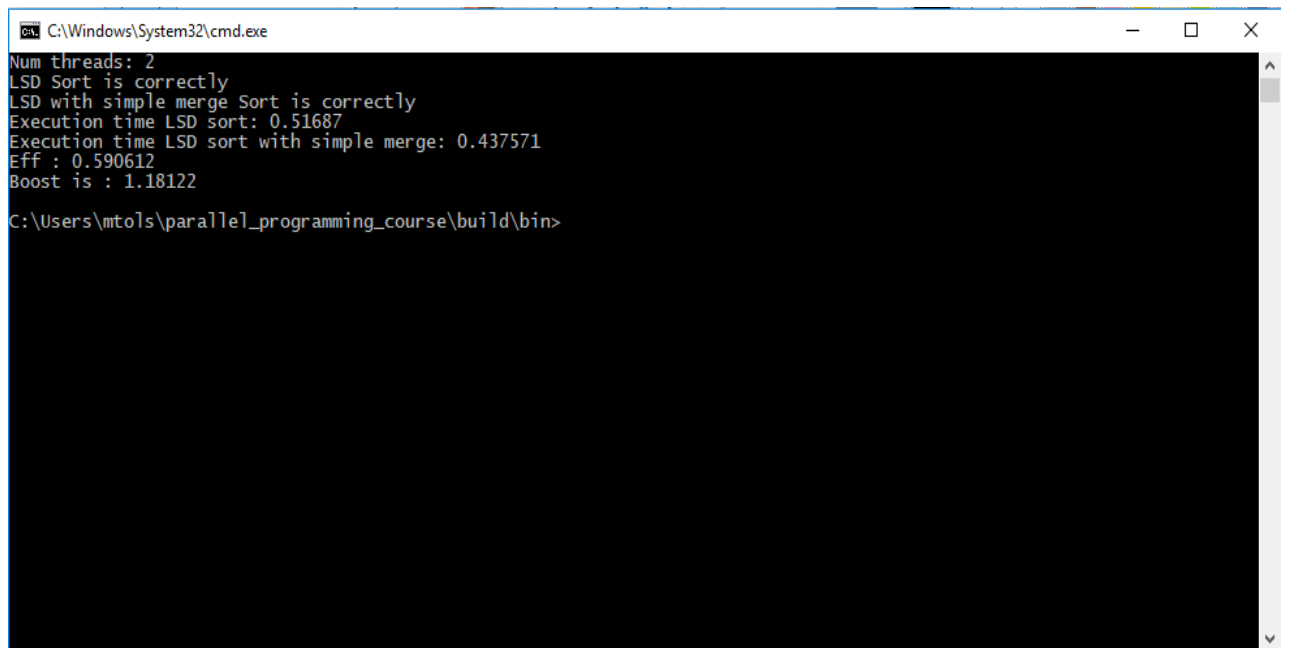
Запустим программу с размером массива 10 миллионов элементов на 1, 2, 4, 8 и 16 потоках и измерим ускорение и эффективность.

На 1 потоке:



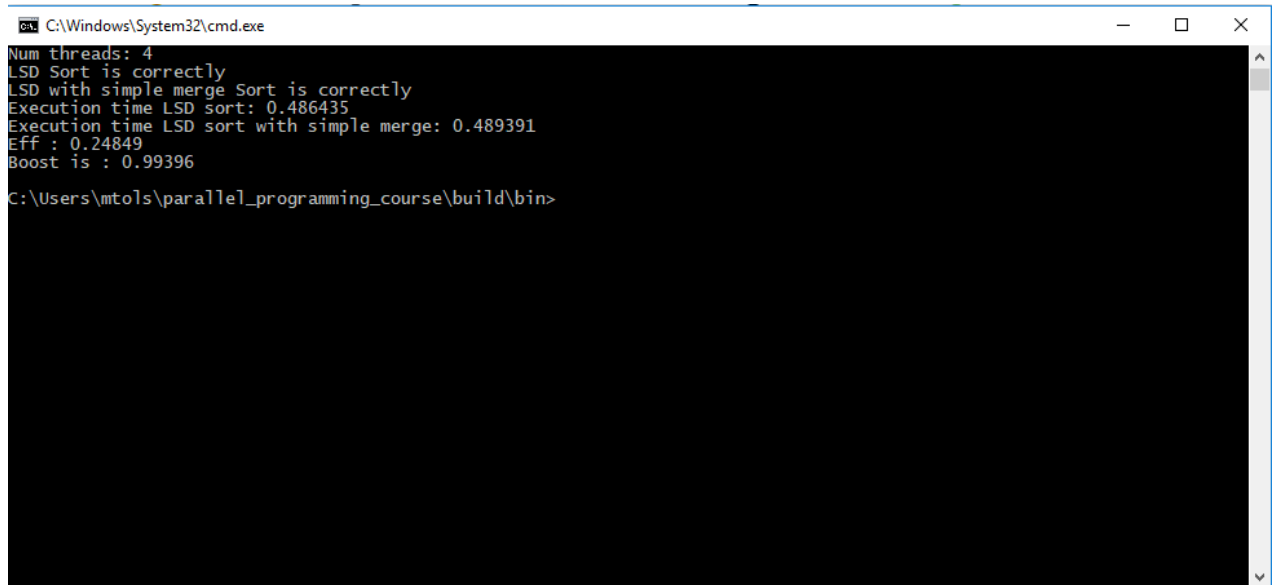
```
C:\Windows\System32\cmd.exe
Num threads: 1
LSD Sort is correctly
LSD with simple merge Sort is correctly
Execution time LSD sort: 0.531965
Execution time LSD sort with simple merge: 0.621856
Eff : 0.855447
Boost is : 0.855447
C:\Users\mtols\parallel_programming_course\build\bin>
```

На 2 потоках:



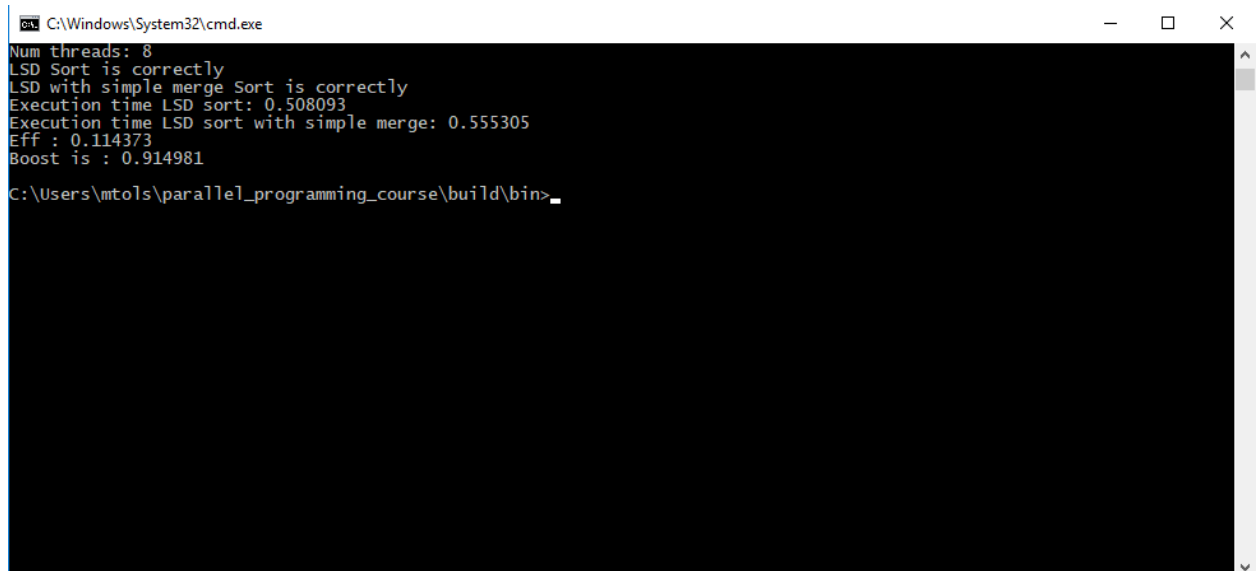
```
C:\Windows\System32\cmd.exe
Num threads: 2
LSD Sort is correctly
LSD with simple merge Sort is correctly
Execution time LSD sort: 0.51687
Execution time LSD sort with simple merge: 0.437571
Eff : 0.590612
Boost is : 1.18122
C:\Users\mtols\parallel_programming_course\build\bin>
```


На 4 потоках:



```
C:\Windows\System32\cmd.exe
Num threads: 4
LSD Sort is correctly
LSD with simple merge Sort is correctly
Execution time LSD sort: 0.486435
Execution time LSD sort with simple merge: 0.489391
Eff : 0.24849
Boost is : 0.99396
C:\Users\mtols\parallel_programming_course\build\bin>
```

На 8 потоках:



```
C:\Windows\System32\cmd.exe
Num threads: 8
LSD Sort is correctly
LSD with simple merge Sort is correctly
Execution time LSD sort: 0.508093
Execution time LSD sort with simple merge: 0.555305
Eff : 0.114373
Boost is : 0.914981
C:\Users\mtols\parallel_programming_course\build\bin>
```

На 16 потоках:

```
C:\Windows\System32\cmd.exe
Num threads: 16
LSD Sort is correctly
LSD with simple merge Sort is correctly
Execution time LSD sort: 0.484687
Execution time LSD sort with simple merge: 0.635842
Eff : 0.0476423
Boost is : 0.762276
C:\Users\mtols\parallel_programming_course\build\bin>
```

Результаты:

Количество потоков	Ускорение	Эффективность
1	0.855447	0.855447
2	1.18122	0.590612
4	0.99396	0.24849
8	0.914981	0.114373
16	0.762276	0.0476423

По результатам можно сделать вывод, что на данном компьютере максимального ускорения можно добиться на 2 потоках.

Максимальная эффективность достигается только на 1 потоке.

Но из - за слияния всех отсортированных частей массива в один, время сортировки на нескольких процессах приближается к времени сортировки на одном процессе, так как слияние занимает линейное время, что делает параллельную реализацию поразрядной сортировки неэффективной.

Для подтверждения корректности сортировки после параллельной и последовательной части идет проверка:

Массив (копия исходного массива) сортируется одним потоком через функцию **std::sort** после чего идет поэлементная проверка уже отсортированных массивов. В результате проверки во всех случаях все массивы прошли проверку на корректность, что означает правильность написанной сортировки.

Parallel sorting with TBB

Во втором приложении я тестирую поразрядную сортировку написанной с помощью библиотеки TBB.

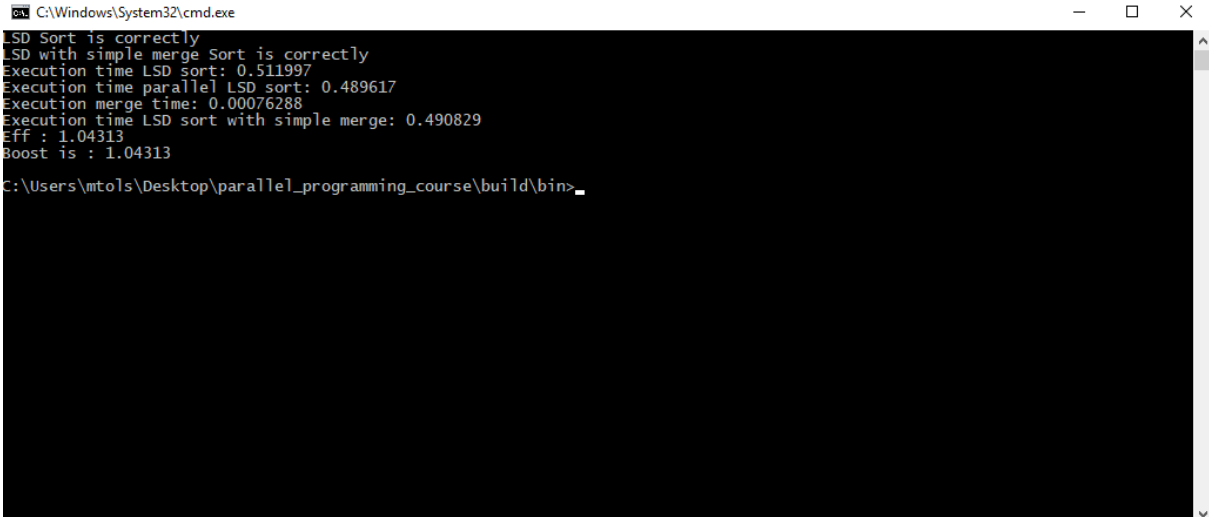
Здесь я сравниваю время работы, ускорение и эффективность сортировок, а так же проверяю их на корректность.

Пояснение каждой строки вывода:

1. Проверка корректности последовательной сортировки
2. Проверка корректности параллельной сортировки
3. Время выполнения последовательной сортировки
4. Время выполнения параллельно поразрядной сортировки
5. Время выполнения слияния
6. Время выполнения параллельной сортировки
7. Эффективность
8. Ускорение

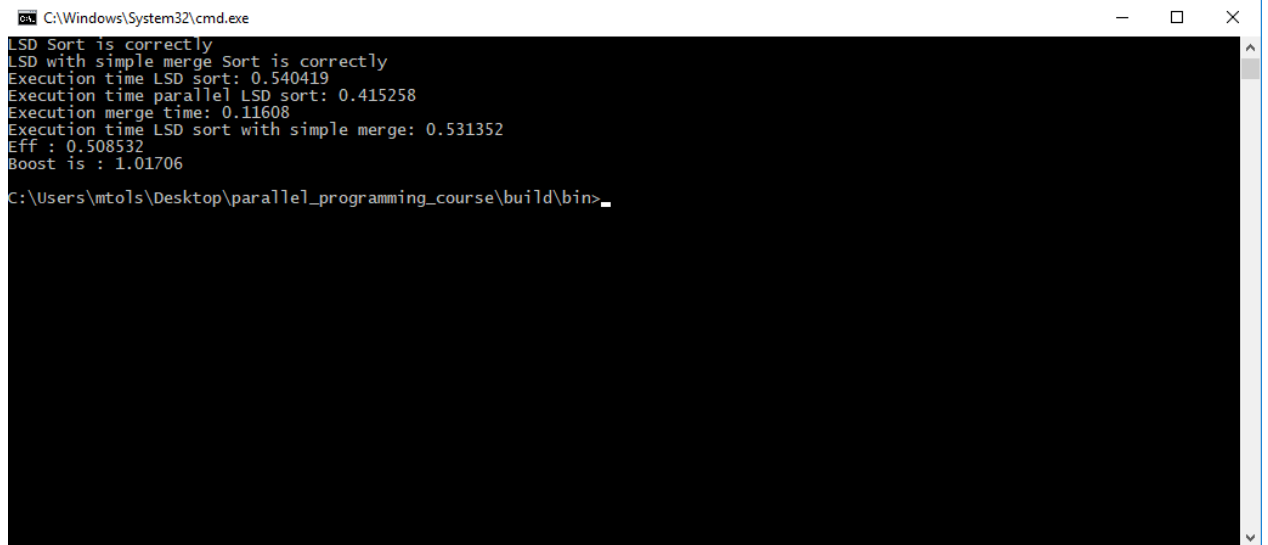
Программа запускается на компьютере с операционной системой Windows 10, процессором Intel Core i5-6200U и 4-мя ГБ оперативной памяти.

На 1 потоке:



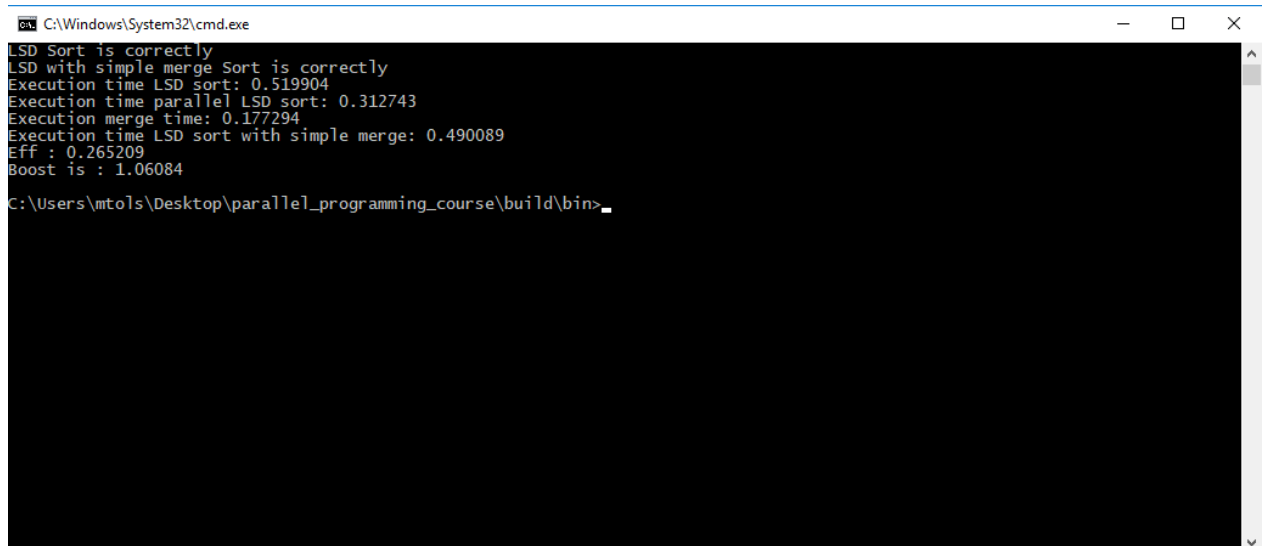
```
C:\Windows\System32\cmd.exe
LSD Sort is correctly
LSD with simple merge Sort is correctly
Execution time LSD sort: 0.511997
Execution time parallel LSD sort: 0.489617
Execution merge time: 0.00076288
Execution time LSD sort with simple merge: 0.490829
Eff : 1.04313
Boost is : 1.04313
C:\Users\mtols\Desktop\parallel_programming_course\build\bin>
```

На 2 потоках:



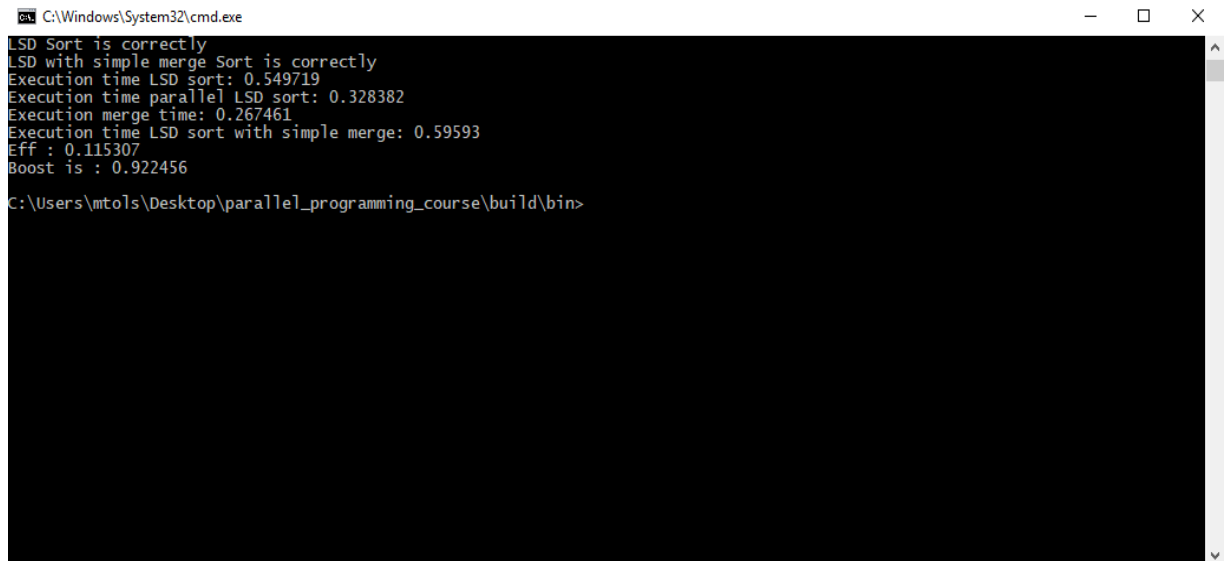
```
C:\Windows\System32\cmd.exe
LSD Sort is correctly
LSD with simple merge Sort is correctly
Execution time LSD sort: 0.540419
Execution time parallel LSD sort: 0.415258
Execution merge time: 0.11608
Execution time LSD sort with simple merge: 0.531352
Eff : 0.508532
Boost is : 1.01706
C:\Users\mtols\Desktop\parallel_programming_course\build\bin>
```

На 4 потоках:



```
C:\Windows\System32\cmd.exe
LSD Sort is correctly
LSD with simple merge Sort is correctly
Execution time LSD sort: 0.519904
Execution time parallel LSD sort: 0.312743
Execution merge time: 0.177294
Execution time LSD sort with simple merge: 0.490089
Eff : 0.265209
Boost is : 1.06084
C:\Users\mtols\Desktop\parallel_programming_course\build\bin>
```

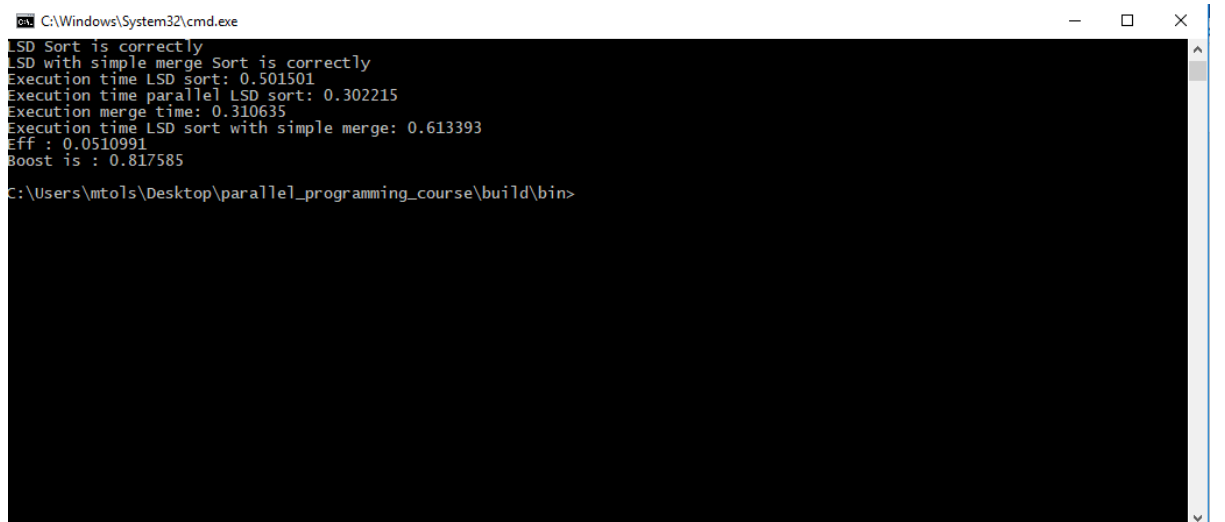
На 8 потоках:



```
C:\Windows\System32\cmd.exe
LSD Sort is correctly
LSD with simple merge Sort is correctly
Execution time LSD sort: 0.549719
Execution time parallel LSD sort: 0.328382
Execution merge time: 0.267461
Execution time LSD sort with simple merge: 0.59593
Eff : 0.115307
Boost is : 0.922456

C:\Users\mtols\Desktop\parallel_programming_course\build\bin>
```

На 16 потоках:



```
C:\Windows\System32\cmd.exe
LSD Sort is correctly
LSD with simple merge Sort is correctly
Execution time LSD sort: 0.501501
Execution time parallel LSD sort: 0.302215
Execution merge time: 0.310635
Execution time LSD sort with simple merge: 0.613393
Eff : 0.0510991
Boost is : 0.817585

C:\Users\mtols\Desktop\parallel_programming_course\build\bin>
```

Результаты:

Количество потоков	Ускорение	Эффективность
1	1.04313	1.04313
2	1.01706	0.508532
4	1.06084	0.265209
8	0.922456	0.115307
16	0.817585	0.0510991

По результатам можно сделать такой же вывод что и при реализации параллельной сортировки с помощью библиотеки OpenMP, т.е. эффективность параллельной версии достаточно мала, чтобы конкурировать с последовательной сортировкой.

Для подтверждения корректности сортировки после параллельной и последовательной части идет проверка:

Массив (копия исходного массива) сортируется одним потоком через функцию **std::sort** после чего идет поэлементная проверка уже отсортированных массивов. В результате проверки во всех случаях все массивы прошли проверку на корректность, что означает правильность написанной сортировки.

Заключение

Была реализована последовательная и параллельная реализация поразрядной сортировки вещественных чисел с простым слиянием. Было замерено время работы этих сортировок, эффективность и протестирована их корректность. В результате было выяснено, что последовательная версия от параллельной по времени почти не различается. Это говорит нам о том, что параллельная реализация является неэффективна по сравнению с последовательной.

Литература

1. Сиднев А.А., Сысоев А.В., Мееров И.Б. Параллельные численные методы: «Лабораторная работа: Сортировки».
2. Сиднев А.А., Сысоев А.В., Мееров И.Б. Учебный курс «Технологии параллельного программирования»

OpenMP main.cpp

```
                                // Copyright 2019 Tolstikov Maksim
// RadixSort with simple merge (double)
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <omp.h>
#include <iostream>
#include <ctime>
#include <algorithm>
#include <cstring>
#include <climits>
#include <vector>

void CountingSort(double* inp, double* out, int byteNum, int size) {
    unsigned char* mas = (unsigned char*)inp;
    int counter[256];
    int tem;
    memset(counter, 0, sizeof(int) * 256);
    for (int i = 0; i < size; i++)
        counter[mas[8 * i + byteNum]]++;
    int j = 0;
    for (; j < 256; j++) {
        if (counter[j] != 0)
            break;
    }
    tem = counter[j];
    counter[j] = 0;
    j++;
    for (; j < 256; j++) {
        int b = counter[j];
        counter[j] = tem;
        tem += b;
    }
    for (int i = 0; i < size; i++) {
        out[counter[mas[8 * i + byteNum]]] = inp[i];
        counter[mas[8 * i + byteNum]]++;
    }
}
```

```

void LastCountingSort(double* inp, double* out, int byteNum, int size) {
    unsigned char* mas = (unsigned char*)inp;
    int counter[256];
    int tem;
    memset(counter, 0, sizeof(int) * 256);
    for (int i = 0; i < size; i++)
        counter[mas[8 * i + byteNum] + 128]++;
    int j = 0;
    for (; j < 256; j++) {
        if (counter[j] != 0)
            break;
    }
    tem = counter[j];
    counter[j] = 0;
    j++;
    for (; j < 256; j++) {
        int b = counter[j];
        counter[j] = tem;
        tem += b;
    }
    for (int i = 0; i < size; i++) {
        out[counter[mas[8 * i + byteNum] + 128]] = inp[i];
        counter[mas[8 * i + byteNum] + 128]++;
    }
}

void merge(double* mas, int sizel, int sizer) {
    int size = sizel + sizer;
    double* tempMas = new double[size];
    int i = 0, j = sizel, k = 0;

    while (i != sizel && j != size) {
        if (mas[i] <= mas[j]) {
            tempMas[k] = mas[i];
            ++i;
            ++k;
        } else {
            tempMas[k] = mas[j];
            ++j;
            ++k;
        }
    }

    if (i < sizel) {
        for (; i < sizel; ++i) {
            tempMas[k] = mas[i];
            ++k;
        }
    }

    if (j < size) {
        for (; j < size; ++j) {
            tempMas[k] = mas[j];
            ++k;
        }
    }

    for (i = 0; i < size; ++i) {
        mas[i] = tempMas[i];
    }

    delete[] tempMas;
}

```



```

}

std::vector<int> merge_size(std::vector<int> counts, int num_th) {
    std::vector<int> tmp;
    if (num_th % 2 == 1) {
        for (int i = 0; i < num_th / 2; ++i) {
            tmp.push_back(counts[2 * i] + counts[2 * i + 1]);
        }
        tmp.push_back(counts[counts.size() - 1]);
    } else {
        for (int i = 0; i < num_th / 2; ++i) {
            tmp.push_back(counts[2 * i] + counts[2 * i + 1]);
        }
    }
    return tmp;
}

int displacement_M(std::vector<int> counts, int num_th) {
    int sum = 0;
    for (int i = 0; i < num_th; ++i) {
        sum += counts[2 * i] + counts[2 * i + 1];
    }
    return sum;
}

int displacement_S(std::vector<int> counts, int num_th) {
    int sum = 0;
    for (int i = 0; i < num_th; ++i) {
        sum += counts[i];
    }
    return sum;
}

void PrintArray(double* array, int size) {
    if (size < 20) {
        for (int i = 0; i < size; i++) {
            std::cout << array[i] << " ";
        }
        std::cout << std::endl;
    }
}

void LSDSortDouble(double* inp, int size) {
    double* out = new double[size];

    for (int i = 0; i < 6; i += 2) {
        CountingSort(inp, out, i, size);
        CountingSort(out, inp, i + 1, size);
    }
    CountingSort(inp, out, 6, size);
    LastCountingSort(out, inp, 7, size);
    delete[] out;
}

void CopyArray(double* mas, double* tmp, int size) {
    for (int i = 0; i < size; i++)
        tmp[i] = mas[i];
}

void CheckingSort(double* mas, double* tmp, int size) {
    for (int i = 0; i < size; i++) {
        if (mas[i] != tmp[i]) {
            std::cout << "Sort is incorrectly" << std::endl;
        }
    }
}

```

```

        break;
    }
}
std::cout << "Sort is correctly" << std::endl;
}

void GenerateArray(double* mas, int size) {
    int b = 100;
    int a = 0;
    for (int i = 0; i < size; i++) {
        mas[i] = static_cast<double>(std::rand()) * (b - a + 1) / RAND_MAX + a;
    }
}

int main(int argc, char* argv[]) {
    double time_1sd = 0;
    double ptime_1sd = 0;
    std::vector<int> counts;
    int size = 10;
    int n = 2;
    std::srand((unsigned)time(NULL));
    double* mas, * tmp, * lmas;
    if (argc == 4) {
        n = atoi(argv[1]);
        if (strcmp(argv[2], "-size") == 0)
            size = atoi(argv[3]);
    }
    mas = new double[size];
    tmp = new double[size];
    lmas = new double[size];
    int tail = size % n;
    for (int i = 0; i < n; ++i) {
        if (i == 0) {
            counts.push_back(size / n + tail);
        } else {
            counts.push_back(size / n);
        }
    }
    if (size < 20) {
        for (int i = 0; i < n; ++i) {
            std::cout << counts[i] << " ";
        }
        std::cout << std::endl;
    }
    if (size < 20)
        std::cout << "Array: ";
    GenerateArray(mas, size);
    CopyArray(mas, tmp, size);
    std::sort(tmp, tmp + size);
    CopyArray(mas, lmas, size);
    if (mas == NULL) {
        std::cout << "Error! Incorrect input data for array";
        return -1;
    }
    PrintArray(mas, size);
    ptime_1sd = omp_get_wtime();
    std::cout << "Num threads: " << n << std::endl;
    omp_set_num_threads(n);
#pragma omp parallel
    {
        LSDSortDouble(mas + displacement_S(counts, omp_get_thread_num()),
            counts[omp_get_thread_num()]);
#pragma omp barrier
    }
}

```

```

    int f = n;
    int k = n / 2 + n % 2;
    while (k > 0) {
        omp_set_num_threads(k);
#pragma omp parallel
        {
            if (f % 2 == 1) {
                if (omp_get_thread_num() != k - 1) {
                    merge(mas + displacement_M(counts, omp_get_thread_num()),
                        counts[2 * omp_get_thread_num()],
                        counts[2 * omp_get_thread_num() + 1]);
                }
            } else {
                merge(mas + displacement_M(counts, omp_get_thread_num()),
                    counts[2 * omp_get_thread_num()],
                    counts[2 * omp_get_thread_num() + 1]);
            }
        }
#pragma omp barrier
        if (omp_get_thread_num() == 0) {
            counts = merge_size(counts, f);
            if (k == 1) {
                k = 0;
            } else {
                k = k / 2 + k % 2;
            }
            f = f / 2 + f % 2;
        }
    }
}

ptime_lsd = omp_get_wtime() - ptime_lsd;
time_lsd = omp_get_wtime();
LSDSortDouble(lmas, size);
time_lsd = omp_get_wtime() - time_lsd;
if (size < 20)
    std::cout << "Array after LSD sort: ";
PrintArray(lmas, size);
std::cout << "LSD ";
CheckingSort(lmas, tmp, size);
if (size < 20)
    std::cout << "Array after LSD sort with simple merge: ";
PrintArray(mas, size);
std::cout << "LSD with simple merge ";
CheckingSort(mas, tmp, size);
std::cout << "Execution time LSD sort: " << time_lsd << std::endl;
std::cout << "Execution time LSD sort with simple merge: " << ptime_lsd << std::endl;
std::cout << "Eff : " << (time_lsd / ptime_lsd) / n << std::endl;
std::cout << "Boost is : " << time_lsd / ptime_lsd << std::endl;
delete[] mas;
delete[] tmp;
delete[] lmas;
return 0;
}

```

TBB main.cpp

// Copyright 2019 Tolstikov Maksim

```
// RadixSort with simple merge (double)
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <tbb/tbb.h>
#include <iostream>
#include <ctime>
#include <algorithm>
#include <cstring>
#include <climits>
#include <vector>

void CountingSort(double *inp, double *out, int byteNum, int size) {
    unsigned char *mas = (unsigned char *)inp;
    int counter[256];
    int tem;
    memset(counter, 0, sizeof(int) * 256);
    for (int i = 0; i < size; i++)
        counter[mas[8 * i + byteNum]]++;
    int j = 0;
    for (; j < 256; j++) {
        if (counter[j] != 0)
            break;
    }
    tem = counter[j];
    counter[j] = 0;
    j++;
    for (; j < 256; j++) {
        int b = counter[j];
        counter[j] = tem;
        tem += b;
    }
    for (int i = 0; i < size; i++) {
        out[counter[mas[8 * i + byteNum]]] = inp[i];
        counter[mas[8 * i + byteNum]]++;
    }
}

void LastCountingSort(double *inp, double *out, int size) {
    unsigned char *mas = (unsigned char *)inp;
    int counter[256];
    int tem;
```

```

memset(counter, 0, sizeof(int) * 256);
for (int i = 0; i < size; i++)
    counter[mas[8 * i + 7] + 128]++;
int j = 0;
for (; j < 256; j++) {
    if (counter[j] != 0)
        break;
}
tem = counter[j];
counter[j] = 0;
j++;
for (; j < 256; j++) {
    int b = counter[j];
    counter[j] = tem;
    tem += b;
}
for (int i = 0; i < size; i++) {
    out[counter[mas[8 * i + 7] + 128]] = inp[i];
    counter[mas[8 * i + 7] + 128]++;
}
}

void merge(double* mas, int sizel, int sizer) {
    int size = sizel + sizer;
    double* tempMas = new double[size];
    int i = 0, j = sizel, k = 0;

    while (i != size && j != size) {
        if (mas[i] <= mas[j]) {
            tempMas[k] = mas[i];
            ++i;
            ++k;
        } else {
            tempMas[k] = mas[j];
            ++j;
            ++k;
        }
    }

    if (i < size) {
        for (; i < size; ++i) {
            tempMas[k] = mas[i];
            ++k;
        }
    }

    if (j < size) {
        for (; j < size; ++j) {
            tempMas[k] = mas[j];
            ++k;
        }
    }

    for (i = 0; i < size; ++i) {
        mas[i] = tempMas[i];
    }

    delete[] tempMas;
}

std::vector<int> merge_size(std::vector<int> counts, int num_th) {
    std::vector<int> tmp;
    if (num_th % 2 == 1) {

```

```

        for (int i = 0; i < num_th / 2; ++i) {
            tmp.push_back(counts[2 * i] + counts[2 * i + 1]);
        }
        tmp.push_back(counts[counts.size() - 1]);
    } else {
        for (int i = 0; i < num_th / 2; ++i) {
            tmp.push_back(counts[2 * i] + counts[2 * i + 1]);
        }
    }
    return tmp;
}

int displacement_M(std::vector<int> counts, int num_th) {
    int sum = 0;
    for (int i = 0; i < num_th; ++i) {
        sum += counts[2 * i] + counts[2 * i + 1];
    }
    return sum;
}

int displacement_S(std::vector<int> counts, int num_th) {
    int sum = 0;
    for (int i = 0; i < num_th; ++i) {
        sum += counts[i];
    }
    return sum;
}

void PrintArray(double *array, int size) {
    if (size < 15) {
        for (int i = 0; i < size; i++) {
            std::cout << array[i] << " ";
        }
        std::cout << std::endl;
    }
}

void LSDSortDouble(double *inp, int size) {
    double *out = new double[size];
    CountingSort(inp, out, 0, size);
    CountingSort(out, inp, 1, size);
    CountingSort(inp, out, 2, size);
    CountingSort(out, inp, 3, size);
    CountingSort(inp, out, 4, size);
    CountingSort(out, inp, 5, size);
    CountingSort(inp, out, 6, size);
    LastCountingSort(out, inp, size);
    delete[] out;
}

void CopyArray(double *mas, double* tmp, int size) {
    for (int i = 0; i < size; i++)
        tmp[i] = mas[i];
}

void CheckingSort(double *mas, double* tmp, int size) {
    for (int i = 0; i < size; i++) {
        if (mas[i] != tmp[i]) {
            std::cout << "Sort is incorrectly" << std::endl;
            break;
        }
    }
}

```

```

        std::cout << "Sort is correctly" << std::endl;
    }

void GenerateArray(double *mas, int size) {
    int b = 100;
    int a = 0;
    for (int i = 0; i < size; i++) {
        mas[i] = static_cast<double>(std::rand())*(b - a + 1) / RAND_MAX + a;
    }
}

int main(int argc, char* argv[]) {
    double time_lsd = 0;
    double ptime_lsd = 0;
    double merge_time = 0;
    double time_plsd = 0;
    std::vector<int> counts;
    int size = 10;
    int n = 1;
    std::srand((unsigned)time(NULL));
    double* mas, * tmp, * lmas;
    if (argc == 4) {
        n = atoi(argv[1]);
        if (strcmp(argv[2], "-size") == 0)
            size = atoi(argv[3]);
    }
    mas = new double[size];
    tmp = new double[size];
    lmas = new double[size];
    int tail = size % n;
    for (int i = 0; i < n; ++i) {
        if (i == 0) {
            counts.push_back(size / n + tail);
        } else {
            counts.push_back(size / n);
        }
    }
    if (size < 20) {
        for (int i = 0; i < n; ++i) {
            std::cout << counts[i] << " ";
        }
        std::cout << std::endl;
    }
    if (size < 15)
        std::cout << "Array: ";
    GenerateArray(mas, size);
    CopyArray(mas, tmp, size);
    std::sort(tmp, tmp + size);
    CopyArray(mas, lmas, size);
    if (mas == NULL) {
        std::cout << "Error! Incorrect input data for array";
        return -1;
    }
    PrintArray(mas, size);
    tbb::task_scheduler_init init(n);
    tbb::tick_count t1 = tbb::tick_count::now();
    tbb::tick_count t3 = tbb::tick_count::now();
    tbb::parallel_for(tbb::blocked_range<int>(0, n), [=, &mas](const
tbb::blocked_range<int> &r) {
        for (int f = r.begin(); f != r.end(); ++f) {
            LSDSortDouble(mas + displacement_S(counts, f), counts[f]);
        }
    });
    tbb::tick_count t4 = tbb::tick_count::now();

```

```

init.terminate();
int j = n;
int k = n / 2 + n % 2;
tbb::tick_count t5 = tbb::tick_count::now();
while (k > 0) {
    init.initialize(k);
    tbb::parallel_for(tbb::blocked_range<int>(0, k), [=, &mas](const
tbb::blocked_range<int> &r) {
        for (int f = r.begin(); f != r.end(); ++f) {
            if (j % 2 == 1) {
                if (f != k - 1) {
                    merge(mas + displacement_M(counts, f), counts[2 * f], counts[2 *
f + 1]);
                }
            } else {
                merge(mas + displacement_M(counts, f), counts[2 * f], counts[2 * f +
1]);
            }
        }
    });
    counts = merge_size(counts, j);
    if (k == 1) {
        k = 0;
    } else {
        k = k / 2 + k % 2;
    }
    j = j / 2 + j % 2;
    init.terminate();
}
tbb::tick_count t6 = tbb::tick_count::now();
tbb::tick_count t2 = tbb::tick_count::now();
ptime_lsd = (t2 - t1).seconds();
tbb::tick_count t11 = tbb::tick_count::now();
LSDSortDouble(lmas, size);
tbb::tick_count t22 = tbb::tick_count::now();
time_lsd = (t22 - t11).seconds();
merge_time = (t6 - t5).seconds();
time_plsd = (t4 - t3).seconds();
if (size < 15)
    std::cout << "Array after LSD sort: ";
PrintArray(lmas, size);
std::cout << "LSD ";
CheckingSort(lmas, tmp, size);
if (size < 15)
    std::cout << "Array after LSD sort with simple merge: ";
PrintArray(mas, size);
std::cout << "LSD with simple merge ";
CheckingSort(mas, tmp, size);
std::cout << "Execution time LSD sort: " << time_lsd << std::endl;
std::cout << "Execution time parallel LSD sort: " << time_plsd << std::endl;
std::cout << "Execution merge time: " << merge_time << std::endl;
std::cout << "Execution time LSD sort with simple merge: " << ptime_lsd << std::endl;
std::cout << "Eff : " << (time_lsd / ptime_lsd) / n << std::endl;
std::cout << "Boost is : " << time_lsd / ptime_lsd << std::endl;
delete[] mas;
delete[] tmp;
delete[] lmas;
return 0;
}

```