МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

Лабораторная работа

“ Быстрая сортировка со слиянием «Разделяй и властвуй»”

**Выполнила:** студентка группы 0836-1

Щукина Анна Евгеньевна

Подпись

**Проверил:**

Доцент кафедры МОСТ,ИИТММ

Сысоев Александр Владимирович

Подпись

Нижний Новгород  
2018

**Содержание**

[Введение 3](#_Toc514595452)

[1. Постановка задачи 4](#_Toc514595453)

[2. Метод решения 5](#_Toc514595454)

[3. Схема распараллеливания 7](#_Toc514595455)

[4. Описание программной реализации 8](#_Toc514595456)

[5. Дополнительные программы для автоматизированной проверки 9](#_Toc514595457)

[5.1. Генератор тестов 9](#_Toc514595458)

[5.2. Программа проверки (checker) 9](#_Toc514595459)

[5. Результаты экспериментов 10](#_Toc514595460)

[Заключение 11](#_Toc514595461)

[Приложение 12](#_Toc514595462)

# Введение

Сортировка представляет собой расстановку объектов в определенном порядке, например, по убыванию или по возрастанию. Вообще упорядочивание элементов – самая распространенная манипуляция с данными, облегчающая в дальнейшем поиск нужной информации. Это во многом относится к различным системам управления базами данных. Алгоритмы сортировки в настоящий момент времени существуют в большом количестве, хотя имеют сходные черты (этапы): сравнение и перестановку элементов попарно до тех пор, пока последовательность не станет упорядоченной. Возможные способы решения этой задачи широко обсуждаются в литературе. В данной работе рассмотрен метод быстрой сортировки со слиянием «Разделяй и властвуй».

# 1. Постановка задачи

Дан неупорядоченный массив, состоящий из n положительных элементов типа double. Цель данной работы – реализовать последовательную и параллельную версию алгоритма быстрой сортировки со слиянием «Разделяй и властвуй» для вещественных чисел и сравнить их время работы при различных значениях размера сортируемого массива.

Таким образом, для достижения поставленной цели необходимо выполнить следующие задачи:

1. Реализовать последовательный алгоритм быстрой сортировки.
2. Реализовать параллельный алгоритм быстрой сортировки средствами OpenMP.
3. Реализовать параллельный алгоритм быстрой сортировки средствами TBB.
4. Сравнить время работы последовательного и параллельных алгоритмов.

# 2. Метод решения

## 2.1. Описание последовательного алгоритма

Идея быстрой сортировки заключается в том, что в массиве выбирается некоторый элемент, называемый разрешающим. Затем он помещается в то место массива, где ему полагается быть после упорядочивания всех элементов. В процессе отыскания подходящего места для разрешающего элемента производятся перестановки элементов так, что слева от них находятся элементы, меньшие разрешающего, и справа — большие (предполагается, что массив сортируется по возрастанию).

Тем самым массив разбивается на две части:

* не отсортированные элементы слева от разрешающего элемента;
* не отсортированные элементы справа от разрешающего элемента.

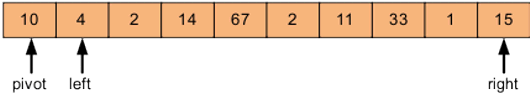
Чтобы отсортировать эти два меньших подмассива, алгоритм рекурсивно вызывает сам себя.

Рассмотрим сортировку на примере массива:

10, 4, 2, 14, 67, 2, 11, 33, 1, 15.

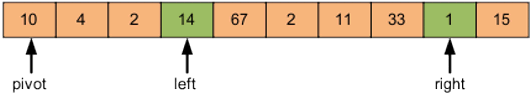
Для реализации алгоритма переупорядочения используем указатель left на крайний левый элемент массива. Указатель движется вправо, пока элементы, на которые он показывает, остаются меньше разрешающего. Указатель right поставим на крайний правый элемент массива, и он движется влево, пока элементы, на которые он показывает, остаются больше разрешающего.

Разрешающий элемент pivot выбирается случайным образом. Пусть им будет крайний левый элемент. Установим указатель left на следующий за ним элемент; right — на последний. Алгоритм должен определить правильное положение элемента 10 и по ходу дела поменять местами неправильно расположенные элементы.

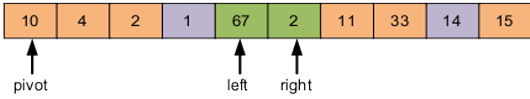


Движение указателей останавливается, как только встречаются элементы, порядок расположения которых относительно разрешающего элемента неправильный.

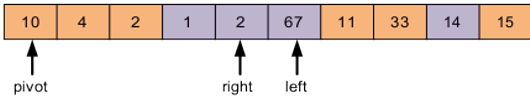
Указатель left перемещается до тех пор, пока не покажет элемент больше 10; right движется, пока не покажет элемент меньше 10.



Эти элементы меняются местами и движение указателей возобновляется.

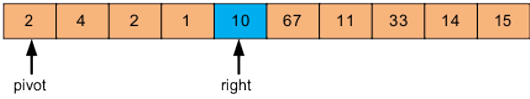


Процесс продолжается до тех пор, пока right не окажется слева от left.



Тем самым будет определено правильное место разрешающего элемента.

Осуществляется перестановка разрешающего элемента с элементом, на который указывает right.

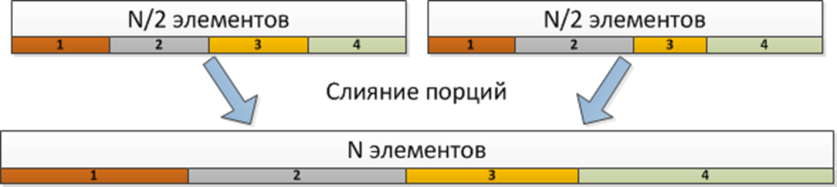


Разрешающий элемент находится в нужном месте: элементы слева от него имеют меньшие значения; справа — большие. Алгоритм рекурсивно вызывается для сортировки подмассивов слева от разрешающего элемента и справа от него.

## 2.2. Описание алгоритма слияния «Разделяй и властвуй»

Идея слияния по алгоритму «Разделяй и властвуй» заключается в разбие-нии массивов на участки, которые можно слить независимо [8]. 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 эле-мента разбивает второй массив на две части). После такого разбиения пер-вые и вторые половины массивов могут сливать независимо, т.к. в первых половинах находятся элементы меньшие элемента x, а во второй – большие (рис. 28). 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.



# 3. Схема распараллеливания быстрой сортировки

## **3.1. Параллельная реализация при помощи технологии OpenMP**

На данном этапе необходимо провести распараллеливание алгоритма быстрой сортировки при помощи технологии OpenMP. Библиотека OpenMP предоставляет разработчику набор директив и библиотечных функций, позволяющих при помощи небольших усилий распараллелить последовательный код.

### 3.1.1. Стратегия распараллеливания

Основной подход к параллельной реализации быстрой сортировки массива заключается в том, что мы делим массив на равные (с точностью до округления) части, каждая из которых обрабатывается своим потоком.

Каждый поток, независимо от остальных потоков выполняет сортировку своей части массива последовательным методом быстрой сортировки.

Получив несколько отсортированных массивов, выполняется их объединение при помощи слияния «Разделяй и властвуй».

### 3.1.1. Ход работы

1. Прежде всего была создана вспомогательная функция  
   int Bin\_Search\_ind(const std::vector<double> arr, int left, int right, double key),с помощью которой, используя бинарный поиск, в векторе arr будет найдена позиция наибольшего элемента меньшего key.
2. Функция vector<double> MergeAndSort(const std::vector<double> vec1, const std::vector<double> vec2) выполняет слияние двух векторов, последовательно выбирая наименьший элемент и записывая в локальный массив, который будет возвращать данная функция.
3. Функция void ShareAndMerge(std::vector<double> vec1, std::vector<double> vec2, double\* write\_to) выполняет слияние двух векторов, используя алгоритм «Разделяй и властвуй».

## **3.2. Параллельная реализация при помощи технологии TBB**

Для выполнения заключительной части работы было предложено реализовать параллельную версию быстрой сортировки со слиянием «Разделяй и властвуй» с помощью библиотеки TBB (Intel Threading Building Blocks — кроссплатформенная библиотека шаблонов С++, разработанная компанией Intel для параллельного программирования.).

Кроме набора высокоуровневых алгоритмов, которые предназначены для упрощения разработки параллельных программ, библиотека TBB предоставляет возможность писать параллельные программы на низком уровне – уровне «логических задач», работа с которыми, тем не менее, более удобна, чем напрямую с потоками. Логическая задача в библиотеке TBB представлена в виде класса tbb::task. Этот класс является базовым при реализации задач, т.е. должен быть унаследован всеми пользовательскими логическими задачами.

В данной части лабораторной работы были созданы дополнительный класс TBBSort, который выполняет рекурсивное слияние и сортировку, функция ParallelSort, которая создает корневую логическую задачу, с которой начнется разворачивание рекурсии, и еще один дополнительный класс Split, в котором был реализован алгоритм слияния «Разделяй и властвуй».

Таким образом, для того, чтобы выполнить сортировку будет достаточно выполнить следующую команду ParallelSort(arr, size, threads); где arr – входной массив данных, size – его размер, а threads – число потоков, которое планируется использовать в задаче. Параметр threads так же участвует в определении размера порции данных для каждого потока.

# 4. Дополнительные программы для автоматизированной проверки

## 4.1. Генератор тестов

Для автоматического создания тестов нужно написать генератор тестов. Генераторы необходимо делать разные. Цель написания генераторов состоит в том, чтобы проверить решения на корректность. То есть, если есть некоторое неправильное решение, то необходимо, чтобы был тест, на котором оно выдаёт неправильный ответ.

Необходимо проверить корректность выполнения быстрой сортировки массивов разной размерности. Таким образом, генератор может создавать массивы, в которых хранится от одного элемента до 100 миллионов элементов типа double.

## 4.2. Программа проверки (checker)

Цель данной программы, используя входные и выходные данные, и известный правильный ответ, определять корректность полученного результата.

Наша задача состоит в том, чтобы проверить корректность выполнения сортировки, сравнивая результат последовательного алгоритма и результат одного из двух методов распараллеливания. Более того, сравнивая эти два файла, мы так же проверяем корректность выполнения сортировки, чтобы элементы были расположены в порядке не убывания. Результат работы будет записан в файл result.txt.

## 4.3. Программы просмотра(viewer и typer)

Для удобства работы с бинарным файлом были разработаны две программы viewer и typer. Viewer переводит бинарный формат в текстовый, а typer наоборот

# 5. Результаты экспериментов

Для каждого эксперимента генерировался массив размерностью 1 миллион элементов, который заполнялся случайными вещественными числами от 0 до 1000.

Продемонстрируем результаты экспериментов при распараллеливании алгоритма с помощью 2 и 4 потоков.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Количество элементов | 1000 | 10000 | 100000 | 1000000 | 25000000 |
| Последовательная версия | 0,000125 | 0,000933 | 0,009265 | 0,126778 | 4,137321 |
| OpenMP версия(2) | 0,007426 | 0,001316 | 0,009994 | 0,135844 | 3,828655 |
| OpenMP версия(4) | 0,004718 | 0,001695 | 0,017041 | 0,149182 | 5,196018 |
| TBB версия(2) | 0,000736 | 0,001188 | 0,008939 | 0,079725 | 2,048567 |
| TBB версия(4) | 0,00043 | 0,001593 | 0,008723 | 0,046231 | 1,627061 |

Построим график зависимости времени работы алгоритма от размерности массива:

По графику видно, что параллельные версии, кроме OpenMP версии на 4 потоках, в общем быстрее, чем последовательная. Также можно заметить, что TBB версии работают быстрее, чем OpenMP версии. Лучшей среди всех параллельных версий при увеличении размера массива себя показала TBB версия на 4 потоках.

Построим график зависимости ускорения от размерности массива:

Как видно из графика, что наилучшее ускорение при увеличении размера матриц показывает TBB при 4 потоках.

# Заключение

Были реализованы как последовательная, так и параллельные версии быстрой сортировки со слиянием «Разделяй и властвуй», с использованием ThreadingBuildingBlocks и OpenMP. С помощью проверки на корректность было проверено, что все три версии дают одинаковый результат. Анализируя графики, можно сделать вывод, что наилучшего ускорения достигает TBB версия на 4 потоках. Также стоит отметить, что все TBB версии работают значительно быстрее чем, OpenMP версии. Однако можно заметить, что версия OpenMP на 4 потоках работает дольше всех.

# Список литературы

* 1. Рабочие материалы преподавателя по общему курсу "Параллельное программирование", подготовленные В.П.Гергелем.
  2. «Инструменты параллельного программирования для систем с общей памятью Библиотека Intel Threading Building Blocks – краткое описание» Мееров И.Б., Сысоев А.В., Сиднев А.А. Кафедра математического обеспечения ЭВМ
  3. Лекционные материалы по курсу «Технологии параллельного программирования» Сиднев А.А., Сысоев А.В., Мееров И.Б.

# Приложение

Последовательный алгоритм быстрой сортировки:

void quicksort(double \*array, int size)

{

int l = 0;

int r = size - 1;

double temp = array[size / 2];

do

{

while (array[l] < temp)

l++;

while (array[r] > temp)

r--;

if (l <= r)

{

double tmp = array[l];

array[l] = array[r];

array[r] = tmp;

l++;

r--;

}

} while (l <= r);

if (r > 0)

quicksort(array, r + 1);

if (l < size)

quicksort(&array[l], size - l);

}

Параллельная реализация при помощи технологии OpenMP:

int Bin\_Search\_ind(const std::vector<double> arr, int left, int right, double key)

{

int midd = 0;

while (1)

{

midd = (left + right) / 2;

if (key < arr[midd])

right = midd - 1;

else if (key > arr[midd])

left = midd + 1;

else

return midd;

if (left > right){

if (left - 1 >= 0)

return left - 1;

else

return left;

}

}

}

vector<double> MergeAndSort(const std::vector<double> vec1, const std::vector<double> vec2) //функция сортировки и слияния двух векторов

{

int i = 0, j = 0;

int size1 = vec1.size(), size2 = vec2.size();

vector<double> write\_to;

write\_to.resize(size1 + size2);

while (i < size1 && j < size2) {

if (vec1[i] <= vec2[j]) {

write\_to[i + j] = vec1[i];

i++;

}

else {

write\_to[i + j] = vec2[j];

j++;

}

}

while (i < size1) {

write\_to[size2 + i] = vec1[i];

i++;

}

while (j < size2) {

write\_to[size1 + j] = vec2[j];

j++;

}

return write\_to;

}

void ShareAndMerge(std::vector<double> vec1, std::vector<double> vec2, double\* write\_to)

{

int left1 = 0, left2 = 0;

int last1 = vec1.size() - 1, last2 = vec2.size() - 1;

int mid1 = (left1 + last1) / 2;//индекс среднего элемента первого вектора

double key = vec1[mid1];//средний элемент первого вектора, относительно него выполняем бинарный посик во втором векторе

int mid2 = Bin\_Search\_ind(vec2, left2, last2, key);

vector<double> vec\_temp1\_1, vec\_temp1\_2;

vec\_temp1\_1.insert(vec\_temp1\_1.begin(), vec1.begin(), vec1.begin() + mid1 + 1);

vec\_temp1\_2.insert(vec\_temp1\_2.begin(), vec2.begin(), vec2.begin() + mid2 + 1);

vector<double> vec\_temp2\_1, vec\_temp2\_2;

vec\_temp2\_1.insert(vec\_temp2\_1.begin(), vec1.begin() + mid1 + 1, vec1.begin() + last1 + 1);

vec\_temp2\_2.insert(vec\_temp2\_2.begin(), vec2.begin() + mid2 + 1, vec2.begin() + last2 + 1);

vec1 = MergeAndSort(vec\_temp1\_1, vec\_temp1\_2);

vec2 = MergeAndSort(vec\_temp2\_1, vec\_temp2\_2);

vector<double> Merged;

Merged.resize(vec1.size() + vec2.size());

merge(vec1.begin(), vec1.end(), vec2.begin(), vec2.end(), Merged.begin());

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

write\_to[i] = Merged[i];

}

Параллельная реализация при помощи технологии TBB:

class Split :public task

{

private:

double \*mas1;

double \*mas2;

double \*tmp;

int size1;

int size2;

public:

Split(double \*\_mas1, double \*\_mas2, double \*\_tmp, int \_size1, int \_size2) : mas1(\_mas1), mas2(\_mas2), tmp(\_tmp), size1(\_size1), size2(\_size2) {}

task\* execute()

{

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];

}

return NULL;

}

};

class TBBSort : public task { //выполняет сортировку либо части массива либо двух частей а после делает сплит

private:

double\* mas;

double\* tmp;

int size;

int blok;

int threads;

int BinSearch(double \*mas, int l, int r, double 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);

}

public:

TBBSort(double \*\_mas, double \*\_tmp, int \_size, int \_blok, int \_threads) : mas(\_mas), tmp(\_tmp), size(\_size), blok(\_blok), threads(\_threads) {}

task\* execute()

{

if (size <= blok)

{

QuickSort(mas, size);

}

else

{

TBBSort &sorter1 = \*new (allocate\_child()) TBBSort(mas, tmp, size / 2, blok, threads / 2);

TBBSort &sorter2 = \*new (allocate\_child()) TBBSort(mas + size / 2, tmp + size / 2, size - size / 2, blok, threads / 2);

set\_ref\_count(3);

spawn(sorter1);

spawn\_and\_wait\_for\_all(sorter2);

Split \*\*sp = new Split\*[threads - 1];

int s = size / 2;

s /= threads;

int l = 0, r = s;

int l2 = 0, r2;

for (int i = 0; i<threads - 1; i++)

{

double x = mas[r];

r2 = BinSearch(mas + size / 2, 0, size - size / 2, x);

sp[i] = new (allocate\_child())

Split(mas + l, mas + size / 2 + l2,

tmp + l + l2, r - l, r2 - l2);

l += s;

r += s;

l2 = r2;

}

Split &spl = \*new (allocate\_child())

Split(mas + l, mas + size / 2 + l2,

tmp + l + l2, size / 2 - l,

size - size / 2 - l2);

set\_ref\_count(threads + 1);

for (int i = 0; i<threads - 1; i++)

spawn(\*(sp[i]));

spawn\_and\_wait\_for\_all(spl);

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

mas[i] = tmp[i];

delete[] sp;

}

return NULL;

}

};

void ParallelSort(double \*inp, int size, int nThreads)

{

double \*out = new double[size];

int blok = size / nThreads;

if (size%nThreads != 0)

blok++;

TBBSort& sorter = \*new (task::allocate\_root()) TBBSort(inp, out, size, blok, nThreads);

task::spawn\_root\_and\_wait(sorter);

delete[] out;

}