**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**Федеральное государственное автономное образовательное**

**учреждение высшего образования**

**«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»**

**Институт информационных технологий, математики и механики**

**Кафедра математического обеспечения и суперкомпьютерных технологий**

**ОТЧЕТ ПО ЛАБАРОТОРНОЙ РАБОТЕ**

**Тема**

**Быстрая сортировка. Простое слияние.**

Выполнила:

студентка 3 курса

группы 081506-3

Санникова Виктория

Проверил:

Доцент кафедры МОСТ, иИТММ

Сысоев Александр Владимирович

**Нижний Новгород, 2018**

**Содержание**

[1. Введение 3](#_Toc514593935)

[2. Постановка задачи 4](#_Toc514593936)

[3. Руководство программиста 5](#_Toc514593937)

[3.1 Основные понятия предметной области. 5](#_Toc514593938)

[3.2 Описание линейного алгоритма. 6](#_Toc514593939)

[3.3 OpenMP версия. 7](#_Toc514593940)

[3.4 TBB версия 8](#_Toc514593945)

[3.5 Вспомогательные программы (?) 10](#_Toc514593946)

[4. Результаты работы. Сравнение. 11](#_Toc514593950)

[5. Заключение 12](#_Toc514593951)

[6. Список литературы 13](#_Toc514593952)

# Введение

Одной из основных задач в программировании является сортировка данных. Сортировка представляет собой расстановку объектов в определенном порядке, например, по убыванию или по возрастанию, или же по другому критерию. Вообще упорядочивание элементов – самая распространенная манипуляция с данными, облегчающая в дальнейшем поиск нужной информации. В настоящее время существует уже огромное количество различных алгоритмов для сортировки: от медленных (сортировка «пузырьком») до достаточно быстрых (сортировка Шелла, быстрая сортировка и др.). Но стоит учитывать, что скорость сортировки зависит от того какие данные мы сортируем и в каком количестве.  
Так же, для уменьшения времени выполнения сортировки можно использовать технологии для распараллеливания алгоритмовю.

В данной лабораторной работе будет рассмотрена быстрая сортировка с использованием простого слияния в трех версиях: последовательная, параллельная с использованием OpenMP, параллельная с использованием TBB.

# Постановка задачи

Для проверки эффективности параллельных алгоритмов было предложено использовать алгоритм быстрой сортировки, а для слияния отсортированных частей массива – стратегию простого слияния. Таким образом для выполнения данной лабораторной работы нам потребуется выполнить следующие задачи:

1. Реализовать последовательный алгоритм быстрой сортировки;
2. Реализовать параллельный алгоритм быстрой сортировки при помощи OpenMP;
3. Реализовать параллельный алгоритм быстрой сортировки при помощи TBB;
4. Реализовать дополнительные программы, позволяющие создавать неупорядоченные массивы, работать с файлами, проверять массив на упорядоченность после выполнения сортировки.
5. Выявить зависимость между количеством данных, количеством потоков и временм сортировки для каждой из трех версий.

# Руководство программиста

## Основные понятия предметной области.

Сортировка (англ. sorting — классификация, упорядочение) — последовательное расположение или разбиение на группы чего-либо в зависимости от выбранного критерия.

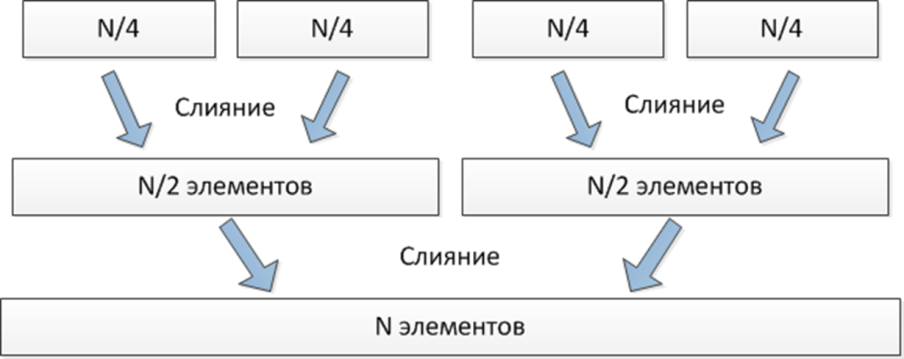
Идея простого слияния заключается в том, что один поток может выполнять слияние двух отсортированных массивов по классическому алгоритму. В этом случае слияние n массивов могут выполнять n/2 параллельных потоков. На следующем шаге слияние n/2 полученных массивов будут выполнять n/4 потоков и т.д. (Рис 1.).

Рис. 1

## Описание линейного алгоритма.

Быстрая сортировка представляет собой усовершенствованный метод сортировки, основанный на принципе обмена. Пузырьковая сортировка является самой неэффективной из всех алгоритмов прямой сортировки. Однако усовершенствованный алгоритм является лучшим из известных методом сортировки массивов. Он обладает столь блестящими характеристиками, что его изобретатель Ч. Хоар назвал его быстрой сортировкой.

Для достижения наибольшей эффективности желательно производить обмен элементов на больших расстояниях. 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:

1. Не отсортированные элементы слева от разрешающего элемента;
2. Не отсортированные элементы справа от разрешающего элемента.

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

Если требуется сортировать больше одного элемента, то нужно:

1. Выбрать в массиве разрешающий элемент;
2. Переупорядочить массив, помещая элемент на его окончательное место;
3. Отсортировать рекурсивно элементы слева от разрешающего;
4. Отсортировать рекурсивно элементы справа от разрешающего.

Ниже приведена реализация данного алгоритма:

void qSort\_array\_recursion(double\* arr, int size)

{

int first = 0;

int last = size - 1;

double elem = arr[size / 2];

do {

while (arr[first] < elem)

first++;

while (arr[last] > elem)

last--;

if (first <= last) {

double tmp = arr[first];

arr[first] = arr[last];

arr[last] = tmp;

first++;

last--;}

} while (first <= last);

if (last > 0) qSort\_array\_recursion(arr, last + 1);

if (first < size) qSort\_array\_recursion(&arr[first], size - first);}

## OpenMP версия.

В данной части лабораторной работы требовалось распараллелить приведенный выше последовательный алгоритм при помощи средств библиотеки OpenMP. OpenMP (Open Multi-Processing) — открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран. Дает описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

Параллельную версию алгоритма быстрой сортировки можно реализовать следующий образом: каждому потоку будет выделена часть массива, которую он должен отсортировать с помощью последовательной версии быстрой сортировки. После получения отсортированных подмассивов происходит их слияние в один отсортированный массив при помощи стратегии просто слияния.

Функция void MergeAndSort(const std::vector<double> vec1, const std::vector<double> vec2, double\* write\_to) сливает два отсортированных вектора в один. Два итератора ставятся на начало каждого из векторов, дальше идет сравнение соответствующих элементов. Элемент, который оказался меньше, записывается в результирующий масиив.

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

{

int i = 0, j = 0;

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

//write\_to = new double[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++;

}

}

Основной блок параллелизма содержится в функции main. Основной его идеей является разделение массива на 2^n частей. Далее каждому из 2^n потоков достается часть массива, которую он должен отсортировать. Далее 2^(n-1) потока выполняют слияние отсортированных частей и т.д., пока не останется 1 поток, который сольет две отсортированные половины исходного массива.

std::vector<double>\* tempArray = new std::vector<double>[threads]; //массив 4х векторов

int \*shift = new int[threads], \*chunk = new int[threads]; //сдвиг относительно начала, хвост относительно сдвига

#pragma omp parallel shared(arr, step, shift, chunk, tempArray) num\_threads(threads)

{

int tid, thread\_index; //ид текущего потока

tid = omp\_get\_thread\_num();

shift[tid] = tid\*(size / threads); //начало с которого будем сортировать

chunk[tid] = (tid == threads - 1) ? size - tid \* (size / threads) : size / threads;

qSort\_array\_recursion(arr + shift[tid], chunk[tid]);

#pragma omp barrier //ждешь все потоки

step = 1; //шаг рекурсии

while (step < threads) {

thread\_index = (int)pow(2, step - 1); //позволяет найти следующиий поток

for (int i = 0; i < chunk[tid]; ++i) { //записываю каждый кусох исходного массива в нужный вектор

tempArray[tid].push\_back(\*(arr + shift[tid] + i));

}

#pragma omp barrier //ждем все потоки

if (tid % (thread\_index \* 2) == 0) { //в каждой паре берем левый

MergeAndSort(tempArray[tid], tempArray[tid + thread\_index], &arr[shift[tid]]);

chunk[tid] += chunk[tid + thread\_index];

tempArray[tid].clear(); tempArray[tid].shrink\_to\_fit(); //удаление памяти

tempArray[tid + thread\_index].clear(); tempArray[tid + thread\_index].shrink\_to\_fit();

}

#pragma omp single

{

step \*= 2;

}

#pragma omp barrier

}

}

## TBB версия

Для выполнения заключительной части работы было предложено реализовать параллельную версию быстрой сортировки с простым слиянием 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 был создан дополнительный класс ParallelSorter, который выполняет рекурсивное слияние и сортировку и функция RootSorter, которая создает корневую логическую задачу, с которой начнется разворачивание рекурсии. Ниже приведен код класса и функции с комментариями и пояснениями:

class ParallelSorter : public task {

private:

double\* mas; //исходный массив

int size; //размер

int portion; //порция

void MergeAndSort(int size1, int size2) //слияние двух подряд идущих отсортированных кусков массива

{

double \*tmp = new double[size1];

for (int i = 0; i<size1; i++)

tmp[i] = mas[i];

double \*mas2 = mas + size1;

int a = 0;

int b = 0;

int i = 0;

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

{

if (tmp[a] <= mas2[b])

{

mas[i] = tmp[a];

a++;

}

else

{

mas[i] = mas2[b];

b++;

}

i++;

}

if (a == size1)

for (int j = b; j<size2; j++)

mas[size1 + j] = mas2[j];

else

for (int j = a; j<size1; j++)

mas[size2 + j] = tmp[j];

}

public:

ParallelSorter(double \*\_mas, int \_size, int \_portion) : //конструктор

mas(\_mas), size(\_size), portion(\_portion) {};

task\* execute() //объявление логической задачи

{

if (size <= portion) // если размер массива меньше или равен порции разделенной между массивами

{

QuickSort(mas, size); //выполняем сортировку этого куска

}

else

{

ParallelSorter &sorter1 = \*new (allocate\_child())ParallelSorter(mas, size / 2, portion); //создаем объект класса для первого куска +1 таск

ParallelSorter &sorter2 = \*new (allocate\_child())ParallelSorter(mas + size / 2, size / 2, portion); //для второго +1 таск

set\_ref\_count(3); //числоо подчиненных задач +1, т.е. 2+1=3

spawn(sorter1); //добавляет задачу в очередь готовых к выполнению

spawn\_and\_wait\_for\_all(sorter2); //добавляет задачу в очередь готовых к выполнению и ждет выполнения всех подчиненных задач

MergeAndSort(size / 2, size - size / 2); //слияние двух отсортированных массивов

}

return NULL;

}

};

void RootSorter(double \*inp, int size, int nThreads) //функция, задающая корневой таск

{

int portion = size / nThreads; //размер порции

if (size%nThreads != 0)

portion++;

ParallelSorter& sorter = \*new (task::allocate\_root()) //корневой таск

ParallelSorter(inp, size, portion); //для цельного массива

task::spawn\_root\_and\_wait(sorter); //добавляем в очередь и ждем выполнения подзадач

}

Таким образом, для того, чтобы выполнить сортировку будет достаточно выполнить следующую команду RootSorter(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. В классе ParallelSorter была реализована дополнительная функция MergeAndSort, которая сливает два отсортированных массива в один методом простого слияния. Для работы с файлами была использована та же схема, что и в OMP версии данной сортировки.

## Вспомогательные программы

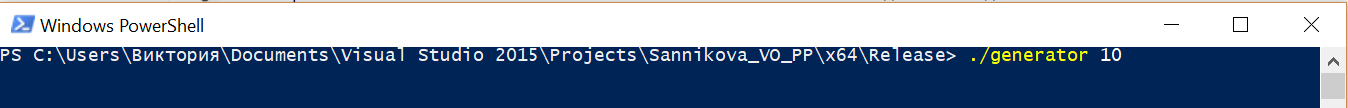
Для проверки эффективности каждой из версий быстрой сортировки требуется создание тестов – бинарных файлов, содержащих неупорядоченный массив указанной длины. Для этих целей была разработана программа generator. На вход поступает число – номер теста, которому соответствует количество элементов в генерируемом массиве (Рис. 2). Разброс генерируемого числа от 0 до 1000.

Рис. 2

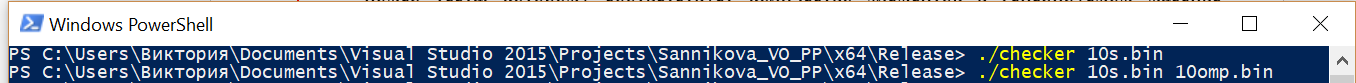
Так же для проверки правильности результатов была разработана программа checker. В зависимости от количества аргументов командной строки данная программа будет проверять на правильность сортировки или один файл, или сравнивать и проверять на упорядоченность два файла (Рис 3.). Результат работы будет записан в файл result.txt

Рис. 3

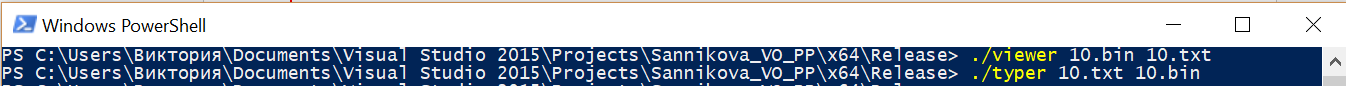
Для удобства работы с бинарным файлом были разработаны две программы viewer и typer. Viewer переводит бинарный формат в текстовый, а typer наоборот (Рис 4.).

Рис. 4

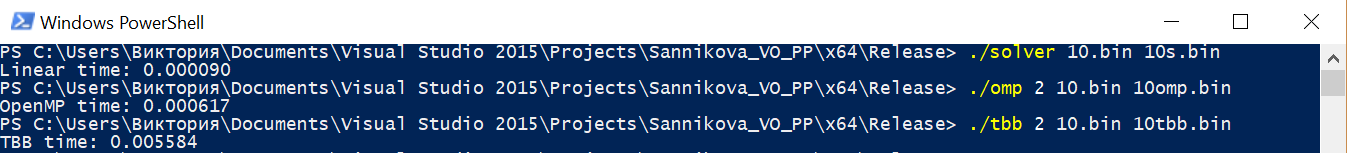
Для работы с основными программами в командную строку нужно ввести следующие параметры (Рис. 5). Где основными параметрами являются файлы откуда и куда читается/записывается массив данных. Дополнительным параметром для параллельных версий является количество потоков.

Рис. 5

# Результаты работы.

Результаты работы приведены в графиках ниже (Рис. 6-7). Данные приведены в таблицах 1 (время) и 2 (ускорения) в зависимости от длины обрабатываемого массива и количества потоков. Эксперименты выполнены на 64-битной 2х ядерной системе.

Рис. 7

Рис. 6

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 100 | 1000 | 10000 | 1000000 | 25000000 | 100000000 |
| linear | 0,000012 | 0,00009 | 0,001862 | 0,185997 | 6,289531 | 32,15273 |
| OpenMP 2 | 0,070794 | 0,000464 | 0,002273 | 0,14426 | 5,496602 | 14,909548 |
| OpenMP 4 | 0,000452 | 0,000659 | 0,001577 | 0,18894 | 5,719666 | 22,906785 |
| TBB 2 | 0,001141 | 0,000724 | 0,008495 | 0,128893 | 3,968348 | 17,463548 |
| TBB 4 | 0,002713 | 0,00094 | 0,001036 | 0,094308 | 2,847325 | 12,964553 |

Таблица 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 100 | 1000 | 10000 | 1000000 | 25000000 | 100000000 |
| OpenMP 2 | 0,00017 | 0,193966 | 0,819182 | 1,289318 | 1,144258 | 2,1525395 |
| OpenMP 4 | 0,026549 | 0,136571 | 1,180723 | 0,984424 | 1,099633 | 1,4036335 |
| TBB 2 | 0,010517 | 0,124309 | 0,219188 | 1,443034 | 1,584924 | 1,8411339 |
| TBB 4 | 0,004423 | 0,095745 | 1,797297 | 1,972229 | 2,208926 | 2,4800493 |

Таблица 2

# Заключение

В ходе данной лабораторной работы была реализованы последовательная и параллельная версии алгоритма быстрой сортировки с простым слиянием. 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. Стоит отметить, что все TBB версии работают гораздо быстрее чем, OpenMP версии, а на размере массива до 10000000 все три версии дают примерно одинаковый результат. OpenMP версия на 2 потоках работает быстрее, чем на 4, а TBB на 4 больше, чем на 2.

# 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.