МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ 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**

**Фундаментальная информатика и информационные технологии**

**Параллельное программирование**

**ОТЧЕТ**

по лабораторной работе

на тему:

**«Быстрая сортировка с простым слиянием.  
Параллельная реализация с OpenMP и TBB»**

**Выполнил:**

студент группы 381706-3

Егоров Данил Геннадьевич

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись

**Преподаватель:**

старший преподаватель

кафедры МОСТ ИТММ

Козинов Евгений Александрович

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись

Нижний Новгород  
2020

**Оглавление**

[1.Введение и постановка задачи. 3](#_Toc40994136)

[2.Теоретическая часть 4](#_Toc40994137)

[2.2 Алгоритм слияния 4](#_Toc40994138)

[2.3 Схема распараллеливания 4](#_Toc40994139)

[3. Программная реализация 5](#_Toc40994140)

[3.1 Последовательная версия быстрой сортировки 5](#_Toc40994141)

[3.2 Параллельная версия с использованием OpenMP 5](#_Toc40994142)

[3.3 Параллельная версия с использованием TBB 7](#_Toc40994143)

[4.Сравнение скорости выполнения алгоритмов 10](#_Toc40994144)

[5.Заключение 13](#_Toc40994145)

# **1.Введение и постановка задачи.**

Сортировка является одной из типовых проблем обработки данных и обычно понимается как задача размещения элементов неупорядоченного набора значений в порядке монотонного возрастания или убывания. "Быстрая сортировка", хоть и была разработана более 40 лет назад, является наиболее широко применяемым и одним их самых эффективных алгоритмов. В рамках выполнения данной лабораторной работы основное внимание уделяется изучению параллельного способа выполнения данного алгоритма сортировки.

**Постановка задачи:**

Разработать последовательный алгоритм быстрой сортировки и две параллельные его версии, с использованием OpenMP и TBB. Провести эксперименты и сравнить скорость выполнения сортировки.

# **2.Теоретическая часть**

**2.1 Быстрая сортировка (QuickSort)**

Быстрая сортировка — алгоритм сортировки, разработанный английским информатиком Чарльзом Хоаром во время его работы в 1960 году. Один из самых быстрых известных универсальных алгоритмов сортировки массивов: в среднем обменов при упорядочении элементов

Краткое описание алгоритма:

1. Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность (см. ниже).
2. Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного», «равные» и «большие».
3. Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

## **2.2 Алгоритм слияния**

Общая идея алгоритма состоит в следующем:

1. На вход подаются n отсортированных массивов.
2. Для слияния выделяется n/2 потоков.
3. Каждый поток выполняет слияние двух массивов в один.
4. В случае, если у нас остаётся более одного массива, возвращаемся на шаг (1). В ином случае, возвращаем результирующий массив.

## **2.3 Схема распараллеливания**

Общий алгоритм параллельной версии быстрой сортировки заключается в следующем:

1. На входе получаем количество элементов массива и количество потоков, которые будут участвовать в сортировке.
2. Делим полученный массив на равные части между потоками.
3. Каждый поток выполняет сортировку своей части последовательной версией алгоритма.
4. Применяем алгоритм параллельного слияния к полученным отсортированным частям.

# **3. Программная реализация**

## **3.1 Последовательная версия быстрой сортировки**

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

## **3.2 Параллельная версия с использованием OpenMP**

1. Параллельное слияние в дереве потоков организуется функцией parallel\_merge(). Принимает двумерный массив, содержащий части массива, к которым необходимо применить слияние. Создает потоки, которые будут производить, непосредственно, само слияние. После произведённого слияния, вызывает себя и передает в качестве параметров новый получившийся двумерный массив. Рекурсия продолжается до тех пор пока не останется две или три части, слияние которых, производится одним потоком и возвращается в виде результирующего массива.

int\* parallel\_merge(int numtasks, part\* parts) {

if (numtasks == 2) {

return merge(parts[0].array, parts[1].array, parts[0].size, parts[1].size);

}

else if (numtasks == 3) {

return merge(parts[2].array, merge(parts[0].array, parts[1].array, parts[0].size, parts[1].size),

parts[2].size, parts[0].size + parts[1].size);

}

else if (numtasks < 2) return nullptr;

int current\_parts = numtasks / 2 + numtasks % 2;

part\* sorted\_parts = new part[current\_parts];

int \_numtasks = numtasks;

int \_current\_parts = current\_parts;

int part\_size = parts[0].size;

for (size\_t i = 0; i < numtasks / 2; i++) {

sorted\_parts[i].array = new int[part\_size \* 2];

sorted\_parts[i].size = part\_size \* 2;

}

if (numtasks % 2) {

sorted\_parts[current\_parts - 1].size = parts[numtasks - 1].size;

sorted\_parts[current\_parts - 1].array = new int[sorted\_parts[current\_parts - 1].size];

for (size\_t i = 0; i < sorted\_parts[current\_parts - 1].size; i++) {

sorted\_parts[current\_parts - 1].array[i] = parts[numtasks - 1].array[i];

}

\_numtasks--;

\_current\_parts--;

}

omp\_set\_num\_threads(\_current\_parts);

#pragma omp parallel shared(\_current\_parts)

{

int taskid = omp\_get\_thread\_num();

int part\_id = ((taskid + 1) \* 2) - 1;

sorted\_parts[taskid].array = merge(parts[part\_id].array, parts[part\_id - 1].array, parts[part\_id].size, parts[part\_id - 1].size);

}

for (size\_t i = 0; i < numtasks; i++) {

delete[] parts[i].array;

}

return parallel\_merge(current\_parts, sorted\_parts);

}

2. Функция merge() выполняет слияние двух массивов в один, проходя по обоим массивам, сравнивая элементы, и копируя в выходной массив меньший из них, до тех пор, пока не дойдём до конца одного из массивов, после чего, копируем оставшиеся элементы.

int\* merge(int\* a, int\* b, int n, int m) {

int\* c;

int size = n + m;

c = new int[size];

int i = 0, j = 0, k = 0;

while (i < n && j < m)

{

if (a[i] <= b[j]) {

c[k] = a[i];

i++;

}

else {

c[k] = b[j];

j++;

}

k++;

}

while (i < n) {

c[k] = a[i];

k++;

i++;

}

while (j < m) {

c[k] = b[j];

k++;

j++;

}

return c;

}

3. Параллельный вызовов QuickSort()

#pragma omp parallel shared(numtasks)

{

int taskid = omp\_get\_thread\_num();

quickSort(parts[taskid], 0, part\_size - 1);

}

4. Проверка результата сортировки на корректность производится попарным сравнением элементов массива отсортированным линейной версией алгоритма и массива отсортированным параллельной версией TBB.

bool same = true;

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

if (sorted\_start\_array[i] != result\_array[i]) {

same = false;

break;

}

}

## **3.3 Параллельная версия с использованием TBB**

1. Параллельное слияние в дереве потоков организуется функцией parallel\_merge(). Принимает двумерный массив, содержащий части массива, к которым необходимо применить слияние. Создает потоки, которые будут производить, непосредственно, само слияние. После произведённого слияния, вызывает себя и передает в качестве параметров новый получившийся двумерный массив. Рекурсия продолжается до тех пор пока не останется две или три части, слияние которых, производится одним потоком и возвращается в виде результирующего массива.

int\* parallel\_merge(int numtasks, part\* parts) {

if (numtasks == 2) {

int\* result\_array = new int[parts[0].size + parts[1].size];

merge(result\_array, parts[0].array, parts[1].array, parts[0].size, parts[1].size);

return result\_array;

}

else if (numtasks == 3) {

int\* tmp\_array = new int[parts[0].size + parts[1].size];

merge(tmp\_array, parts[0].array, parts[1].array, parts[0].size, parts[1].size);

int\* result\_array = new int[parts[0].size + parts[1].size + parts[2].size];

merge(result\_array, parts[2].array, tmp\_array, parts[2].size, parts[0].size + parts[1].size);

delete[] tmp\_array;

return result\_array;

}

else if (numtasks < 2) return nullptr;

int current\_parts = numtasks / 2 + numtasks % 2;

part\* sorted\_parts = new part[current\_parts];

int \_numtasks = numtasks;

int \_current\_parts = current\_parts;

int part\_size = parts[0].size;

for (size\_t i = 0; i < numtasks / 2; i++) {

int part\_id = ((i + 1) \* 2) - 1;

sorted\_parts[i].size = parts[part\_id].size + parts[part\_id - 1].size;

sorted\_parts[i].array = new int[sorted\_parts[i].size];

}

if (numtasks % 2) {

sorted\_parts[current\_parts - 1].size = parts[numtasks - 1].size;

sorted\_parts[current\_parts - 1].array = new int[sorted\_parts[current\_parts - 1].size];

for (size\_t i = 0; i < sorted\_parts[current\_parts - 1].size; i++)

{

sorted\_parts[current\_parts - 1].array[i] = parts[numtasks - 1].array[i];

}

\_numtasks--;

\_current\_parts--;

}

ParallelMerger& parallelmerger = \*new (task::allocate\_root()) ParallelMerger(\_current\_parts, parts, sorted\_parts);

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

for (size\_t i = 0; i < numtasks; i++) {

delete[] parts[i].array;

}

return parallel\_merge(current\_parts, sorted\_parts);

}

2. Функция merge() выполняет слияние двух массивов в один, проходя по обоим массивам, сравнивая элементы, и копируя в выходной массив меньший из них, до тех пор, пока не дойдём до конца одного из массивов, после чего, копируем оставшиеся элементы.

void merge(int\* c, int\* a, int\* b, int n, int m) {

int i = 0, j = 0, k = 0;

while (i < n && j < m) {

if (a[i] <= b[j]) {

c[k] = a[i];

i++;

}

else {

c[k] = b[j];

j++;

}

k++;

}

while (i < n) {

c[k] = a[i];

k++;

i++;

}

while (j < m) {

c[k] = b[j];

k++;

j++;

}

}

3. Класс Merger наследуется от класса tbb::task. Реализует выполнение слияния одной задачей.

class Merger : public task {

private:

unsigned task\_id;

part\* parts;

part\* sorted\_parts;

public:

Merger(unsigned \_task\_id, part\* \_parts, part\* \_sorted\_parts)

: task\_id(\_task\_id), parts(\_parts), sorted\_parts(\_sorted\_parts) {}

task\* execute() {

int part\_id = ((task\_id + 1) \* 2) - 1;

merge(sorted\_parts[task\_id].array, parts[part\_id].array, parts[part\_id - 1].array, parts[part\_id].size, parts[part\_id - 1].size);

return NULL;

}

};

4. Класс Sorter наследуется от класса tbb::task. Реализует выполнение QuickSort() одной задачей.

class Sorter : public task {

private:

unsigned task\_id;

unsigned part\_size;

part\* parts;

public:

Sorter(unsigned \_task\_id, part\* \_parts, unsigned \_part\_size)

: task\_id(\_task\_id), parts(\_parts), part\_size(\_part\_size) {}

task\* execute() {

quickSort(parts[task\_id].array, 0, part\_size - 1);

return NULL;

}

};

5. Класс ParallelMerger наследуется от класса tbb::task. Организует параллельное выполнение QuickSort().

class ParallelMerger : public task {

private:

unsigned num\_tasks;

part\* parts;

part\* sorted\_parts;

vector<Merger\*> mergers;

public:

ParallelMerger(unsigned \_num\_tasks, part\* \_parts, part\* \_sorted\_parts)

: num\_tasks(\_num\_tasks), parts(\_parts), sorted\_parts(\_sorted\_parts) {

mergers.resize(num\_tasks);

}

task\* execute() {

unsigned task\_id = 0;

for (auto& x : mergers) {

x = new (allocate\_child()) Merger(task\_id, parts, sorted\_parts);

task\_id++;

}

set\_ref\_count(num\_tasks + 1);

for (auto& x : mergers) {

spawn(\*x);

}

wait\_for\_all();

return NULL;

}

};

5. Проверка результата сортировки на корректность производится попарным сравнением элементов массива отсортированным линейной версией алгоритма и массива отсортированным параллельной версией TBB.

bool same = true;

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

if (sorted\_start\_array[i] != result\_array[i]) {

same = false;

break;

}

}

# **4.Сравнение скорости выполнения алгоритмов**

Эксперименты проводились на персональном компьютере со следующими характеристиками:

* Память - 16 GB RAM
* Процессор - AMD FX(tm)-8350 Eight-Core c 4 ядрами и 8 логическими процессорами.

Было проведено 50 тестов на массиве размером 500000 элементов. Данные усреднены. Тесты на правильность сортировки во всех экспериментах были пройдены.

Результаты OpenMP версии:

|  |  |  |
| --- | --- | --- |
| Число потоков | Время работы | Ускорение |
| 1 | 0.204 | 1 |
| 2 | 0.065 | 3.1 |
| 4 | 0.032 | 6.3 |
| 8 | 0,018 | 9,6 |

Результаты TBB версии:

|  |  |  |
| --- | --- | --- |
| Число потоков | Время работы | Ускорение |
| 1 | 0.204 | 1 |
| 2 | 0.061 | 3.3 |
| 4 | 0.313 | 6.5 |
| 8 | 0.019 | 10.7 |

# 

# **5.Заключение**

В ходе выполнения работы был разработан алгоритм последовательный версии быстрой сортировки и две параллельные его версии, с использованием OpenMP и TBB. Были проведены серии экспериментов по сравнению скорости выполнения работы данных алгоритмов, в следствии которых, было установлено, что параллельные методы дают существенный прирост в скорости выполнения сортировки. Все алгоритмы были реализованы корректно. В сравнении параллельных версий алгоритмов было выявлено, что TBB версия показывает больший прирост в скорости, чем OpenMP версия.