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 четно-нечетным слиянием Бэтчера. Параллельная реализация с OpenMP и TBB»**

**Выполнил:**

студент группы 381706-3

Маремьянин Иван Алексеевич

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

Подпись

**Преподаватель:**

старший преподаватель

кафедры МОСТ ИТММ

Козинов Евгений Александрович

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

Подпись

Нижний Новгород  
2020

**Оглавление**

[**1.Введение и постановка задачи.** 3](#_Toc39968045)

[**2.Теоретическая часть** 4](#_Toc39968046)

[**2.1 Быстрая сортировка (QuickSort)** 4](#_Toc39968047)

[**2.2 Алгоритм четно-нечетного слияния Бэтчера** 4](#_Toc39968048)

[**2.3 Схема распараллеливания** 5](#_Toc39968049)

[**3. Программная реализация** 6](#_Toc39968050)

[**3.1 Последовательная версия быстрой сортировки** 6](#_Toc39968051)

[**3.2 Параллельная версия с использованием OpenMP** 6](#_Toc39968052)

[**3.3** **Распараллеливание с помощью TBB** 8](#_Toc39968053)

[**3.4** **Проверка правильности сортировки массива** 9](#_Toc39968054)

[**4.Сравнение скорости выполнения алгоритмов** 11](#_Toc39968055)

[**5.Заключение** 12](#_Toc39968056)

# **1.Введение и постановка задачи.**

Сортировка данных – одна из самых важных задач в программировании. Создание алгоритмов параллельных сортировок было начато, как только появились методы параллельных вычислений. С тех пор было предложено множество параллельных методов сортировки.

Были поставлены следующие задачи:

1. Разработка последовательной версии быстрой сортировки.
2. Разработка параллельной версии быстрой сортировки с использованием OpenMP.
3. Разработка параллельной версии быстрой сортировки с использованием TBB.
4. Провести эксперименты по сравнению скорости выполнения сортировки.

# **2.Теоретическая часть**

**2.1 Быстрая сортировка (QuickSort)**

Быстрая сортировка – алгоритм упорядочивания данных, разработанный Чарльзом Хоаром в 1960 году. Данный алгоритм является одним из самых быстрых известных способов сортировки массивов, примерно обменов при *n* элементов. Краткое описание алгоритма:

1. Из массива выбирается элемент, который в последствии называется опорным. Этим элементом может быть любой элемент массива.
2. Сравнить все остальные элементы с опорным и переставить их так, чтобы элементы меньше опорного располагались левее от него, а все элементы больше опорного соответственно правее от него. После этого шага опорный элемент находится на своем месте в массиве.
3. Для обоих частей массива рекурсивно выполнить данную последовательность действий (если их размер больше единицы).

## **2.2 Алгоритм четно-нечетного слияния Бэтчера**

Данный алгоритм позволяет оптимизировать процедуру слияния отсортированных массивов. Краткое описание алгоритма:

1. На вход подаются 2 отсортированных массива.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 5 | 7 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 6 | 8 | 9 | 10 |

1. Из этих двух массивов в возрастающем порядке выбираются отдельно элементы, стоящие на четных местах и отдельно нечетных. В результате чего из двух исходных отсортированных массивов образуются два новых также отсортированных массива.

Массив четных элементов:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 7 | 8 | 10 |

Массив нечетных элементов:

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 5 | 6 | 9 |

1. Массивы, полученные на втором шаге, сливаются. Для этого нужно сравнить пары элементы, стоящие на нечетной и четной позициях.

Результат:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

## **2.3 Схема распараллеливания**

Реализация параллельной версии быстрой сортировки заключается в том, что:

1. Мы делим изначальный массив на равные части, каждая из которых обрабатывается своим потоком.
2. Каждый поток, независимо от остальных, сортирует свою часть массива последовательным методом быстрой сортировки.
3. Получив несколько отсортированных массивов, выполняется их объедение при помощи четно-нечетного слияния Бэтчера.

# **3. Программная реализация**

## **3.1 Последовательная версия быстрой сортировки**

void quickSort(int arr[], int first, int last)

{

int f = first;

int l = last;

int pivot = arr[(f+l)/2];

while (f <= l)

{

while (arr[f] < pivot)

f++;

while (arr[l] > pivot)

l--;

if (f <= l)

swap(&arr[f++], &arr[l--]);

}

if (first < l)

quickSort(arr, first, l);

if(last>f)

quickSort(arr, f, last);

}

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

1. Изначально была создана функция void SeparationOddEven, принимающая на вход 2 массива, которая выбирает из них четные или нечетные элементы (в зависимости от первого параметра) и сохраняющая их в вектор.  
void SeparationOddEven(int chetnost, int\* first\_arr, int size1, int\* second\_arr, int size2,vector<int>& parts)

{

int i, j;

if (chetnost == 0)

{

i = 0, j = 0;

}

else

{

i = 1, j = 1;

}

parts.reserve(size1 + size2);

while (i < size1 && j < size2)

{

if (first\_arr[i] <= second\_arr[j])

{

parts.push\_back(first\_arr[i]);

i += 2;

}

else

{

parts.push\_back(second\_arr[j]);

j += 2;

}

}

if (i >= size1)

while (j < size2)

{

parts.push\_back(second\_arr[j]);

j += 2;

}

else

while (i < size1)

{

parts.push\_back(first\_arr[i]);

i += 2;

}

}

2. Функция void Merger выполняет слияние двух векторов, полученные в результате первого шага, последовательно выбирая наименьший элемент и записывая его в массив, затем происходит сортировка парно стоящих элементов массива.

void Merger(vector<int> first\_vec,vector<int> second\_vec, int\* result)

{

int i = 0, j = 0;

int size1 = first\_vec.size(), size2 = second\_vec.size();

while (i < size1 && j < size2)

{

result[i + j] = first\_vec[i];

result[i + j + 1] = second\_vec[j];

++i; ++j;

}

while (i < size1)

{

result[size2 + i] = first\_vec[i];

i++;

}

while (j < size2)

{

result[size1 + j] = second\_vec[j];

j++;

}

i = 1;

while (i < size1 + size2 - 1)

{

if (result[i] > result[i + 1])

{

j = result[i];

result[i] = result[i + 1];

result[i + 1] = j;

}

++i;

}

}

3. Основной блок кода распараллеливания

int step;

vector<int>\* temparr = new vector<int>[numtasks];

int\* sdvig = new int[numtasks];

int\* fragments\_size = new int[numtasks];

#pragma omp parallel shared(arr, step, sdvig, fragments\_size, temparr) num\_threads(numtasks)

{

int task\_id;

int thread\_pair;

task\_id = omp\_get\_thread\_num();

sdvig[task\_id] = task\_id \* (size / numtasks);

fragments\_size[task\_id] = (task\_id == numtasks - 1) ? size - task\_id \* (size / numtasks) : size / numtasks;

quickSort(arr + sdvig[task\_id], 0, fragments\_size[task\_id] - 1);

#pragma omp barrier

step = 1;

while (step < numtasks)

{

thread\_pair = (int)pow(2, step - 1);

if (task\_id % (thread\_pair \* 2) == 0)

{

SeparationOddEven(0, arr + sdvig[task\_id], fragments\_size[task\_id], arr + sdvig[task\_id + thread\_pair], fragments\_size[task\_id + thread\_pair], temparr[task\_id]);

}

else if (task\_id % thread\_pair == 0)

{

SeparationOddEven(1, arr + sdvig[task\_id], fragments\_size[task\_id], arr + sdvig[task\_id - thread\_pair], fragments\_size[task\_id - thread\_pair], temparr[task\_id]);

}

#pragma omp barrier

if (task\_id % (thread\_pair \* 2) == 0)

{

Merger(temparr[task\_id], temparr[task\_id + thread\_pair], arr + sdvig[task\_id]);

fragments\_size[task\_id] += fragments\_size[task\_id + thread\_pair];

temparr[task\_id].clear();

temparr[task\_id].shrink\_to\_fit();

temparr[task\_id + thread\_pair].clear();

temparr[task\_id + thread\_pair].shrink\_to\_fit();

}

#pragma omp single

{

step \*= 2;

}

#pragma omp barrier

}

}

delete[] temparr;

delete[] fragments\_size;

delete[] sdvig;

## **Распараллеливание с помощью TBB**

1. Было создано два практически идентичных класса SeparationEven и SeparationOdd, представляющие собой задачи по выделению из массива четных и нечетных элементов соответственно. Ниже представлен только код класса SeparationEven, так как они почти идентичны.

class SeparationEven : public task

{

private:

int\* arr;

int size1, size2;

public:

SeparationEven(int\* \_arr, int \_size1, int \_size2) : arr(\_arr), size1(\_size1), size2(\_size2) {

}

task\* execute()

{

int\* arr2 = arr + size1;

int num = (size1 + size2 + 1) / 2;

int\* tmp = new int[num];

int a = 0, b = 0, i = 0;

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

if (arr[a] <= arr2[b])

{

tmp[i] = arr[a];

a += 2;

}

else

{

tmp[i] = arr2[b];

b += 2;

}

i++;

}

if (a >= size1)

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

tmp[i] = arr2[j];

else

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

tmp[i] = arr[j];

for (int j = 0; j < num; ++j)

arr[j \* 2] = tmp[j];

return NULL;

}

};

2. Для последнего этапа слияния был создан класс Checker, выполняющий попарное сравнение элементов полученного массива.

class Checker

{

private:

int\* arr;

public:

Checker(int\* \_arr) : arr(\_arr) {}

void operator()(const blocked\_range<int>& r) const

{

int begin = r.begin(), end = r.end();

for (int i = begin; i < end; i++)

if (arr[i - 1] > arr[i])

{

int tmp = arr[i - 1];

arr[i - 1] = arr[i];

arr[i] = tmp;

}

}

};

3. Класс Sorting реализует основную часть распараллеливания быстрой сортировки, производя разделение массива на части, сортируемые последовательно и объединяя результаты при помощи четно-нечетного слияния Бэтчера.

class Sorting : public task

{

private:

int\* arr;

int size;

int parts;

public:

Sorting (int\* \_arr, int \_size, int \_parts) : arr(\_arr), size(\_size), parts(\_parts) {}

task\* execute()

{

if (size <= parts)

{

quickSort(arr,0,size-1);

}

else

{

int s = size / 2 + (size / 2) % 2;

Sorting& sorting1 = \*new (allocate\_child()) Sorting(arr, s, parts);

Sorting& sorting2 = \*new (allocate\_child()) Sorting(arr + s, size - s, parts);

set\_ref\_count(3);

spawn(sorting1);

spawn\_and\_wait\_for\_all(sorting2);

SeparationEven& separationEven = \*new (allocate\_child()) SeparationEven(arr, s, size - s);

SeparationOdd& separationOdd = \*new (allocate\_child()) SeparationOdd(arr, s, size - s);

set\_ref\_count(3);

spawn(separationEven);

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

parallel\_for(blocked\_range<int>(1, size), Checker(arr));

}

return NULL;

}

};

## **Проверка правильности сортировки массива**

Для проверки правильности выполнения алгоритмов был создан алгоритм, проверяющий конечный массив.

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

{

if (arr[i] >= arr[i - 1])

continue;

else

{

check = true;

break;

}

}

if (check == true)

std::cout << "Array isnt sorted";

else

std::cout << "Array is sorted";

# **4.Сравнение скорости выполнения алгоритмов**

Эксперименты проводились на компьютере с 8гб оперативной памяти и процессором Intel Core i5 2.40GHz(4 ядра, 4 потока).

Тесты проводились на массиве размером 10 000 000 элементов, тесты проводились многократно и данные по времени усреднены. Тесты на правильность сортировки всегда показывали корректность работы алгоритмов.

Результаты OpenMP версии:

|  |  |  |
| --- | --- | --- |
| Число потоков | Время работы | Ускорение |
| 1 | 2,48 | - |
| 2 | 1.77 | 1.4 |
| 4 | 1.18 | 2.1 |

Результаты TBB версии:

|  |  |  |
| --- | --- | --- |
| Число потоков | Время работы | Ускорение |
| 1 | 2,48 | - |
| 2 | 1.12 | 2.2 |
| 4 | 0.73 | 3.4 |

# **5.Заключение**

В этой работе были реализованы 3 программы, выполняющие быструю сортировку. Были проведены эксперименты по сравнению скорости работы, по окончанию которых, было доказано, что параллельные методы быстрой сортировки дают существенный прирост в скорости выполнения алгоритма. Так же было доказано что все три алгоритма выполняют свою задачу корректно, и из результатов экспериментов видно, что TBB-версия программы показывает более эффективные результаты, чем OpenMP-версия.