Министерство образования и науки Российской Федерации

Государственное образовательное учреждение высшего профессионального образования «Нижегородский государственный университет им. Н.И. Лобачевского»

Институт информационных технологий, математики и механики

Кафедра: программная инженерия

Специальность (направление): Программная инженерия

Отчет

по лабораторной работе

по дисциплине «Параллельное программирование»

тема:

«Поразрядная сортировка для целых чисел с четно-нечетным слиянием Бэтчера»

Выполнил: студент группы 0828

Горожанин М.Ю.

_____Подпись

Научный руководитель:

Доцент, к.т.н. Сысоев А.В.

_____Подпись

Нижний Новгород 2018

Оглавление

Введение

Поразрядная сортировка для целых чисел

Поразрядная сортировка имеет две модификации:

- Most Significant Digit, MSD (Нисходящая поразрядная сортировка, от старшего разряда к младшему)
- Least Significant Digit. LSD (Восходящая поразрядная сортировка, от младшего разряда к старшему)

В данной работе был применен алгоритм LSD как наиболее эффективный (и подходящий).

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).
Эффективность алгоритма $O(k * (N + m))$ где,

$$k - \text{число разрядов}$$
$$N - \text{число элементов}$$
$$m - \text{чило возможных значений (число элементов системы счисления)}$$

Т.е. эффективность при правильно реализованном алгоритме линейная.

Общая идея алгоритма быстрой сортировки состоит в следующем:

- 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)

Чтобы решить проблему с сортировкой чисел разного знака, в данной работе сделано разделение массива на отрицательный и неотрицательный.

Разбиение «Четно-нечетное слияние Бэтчера»

Чётно-нечётное слияние Бэтчера заключается в том, что два упорядоченных массива, которые необходимо слить, разделяются на чётные и нечётные элементы. Такое слияние может быть выполнено параллельно.

Чтобы массив стал окончательно отсортированным, достаточно сравнить пары элементов, стоящие на нечётной и чётной позициях. Первый и последний элементы массива проверять не надо, т.к. они являются минимальным и максимальным элементов массивов.

Чётно-нечётное слияние Бэтчера позволяет задействовать 2 потока при слиянии двух упорядоченных массивов. В этом случае слияние N массивов могут выполнять N параллельных потоков. На следующем шаге слияние N/2 полученных массивов будут выполнять N/2 потоков и т.д. На последнем шаге два массива будут сливать 2 потока

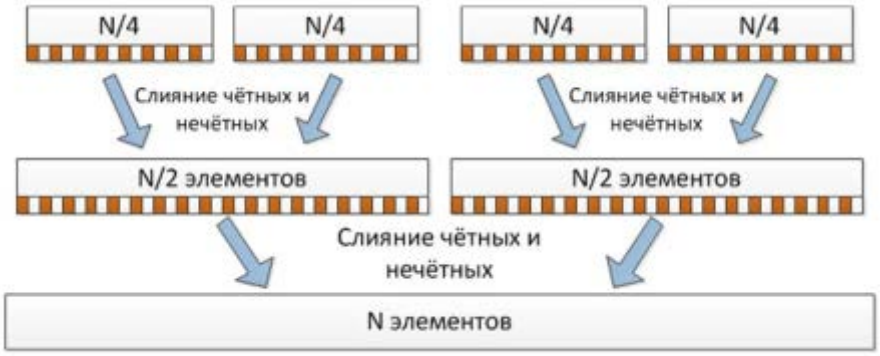


Рис. 1. Чётно-нечётное слияние Бэтчера

Характеристики компьютера, на котором писалась и выполнялась программа:

Intel Core I5-7600K(3800 MHz), RAM-16 ГБ, Windows 10 PRO Версия 1803, сборка 17133.1

Тестовая версия

Solver

Описание: Реализация восходящей поразрядной сортировки, по десятичным разрядам чисел файлы sol.cpp и before_code.cpp

Запуск: Команда в командной строке (или Powershell)
<имя скомпилированной программы> <входной бинарный файл> <выходной бинарный файл>

P.s. <входной бинарный файл> - должен существовать.

Реализация(*sol.cpp***):** Вызываемая функция сортировки называется *radix_sort_sol(vector<int>& array_for_sort, const int left_border, const int right_border)*, в которой первый параметр – сортируемый вектор, второй-третий параметры – границы сортируемого вектора. В функции вектор делится на положительную часть и отрицательную. Далее используются две дополнительные функции.

```
void radix_sort_sol(vector<int>& array_for_sort, const int left_border, const int right_border)
{
	auto num_of_numbers = right_border-left_border+1;
	vector<int> vec_pos;
	vector<int> vec_neg;

	auto num_of_pos_numbers = 0;
	auto num_of_neg_numbers = 0;

	for (size_t i = left_border; i < right_border+1; ++i)
		if (array_for_sort[i] >= 0)
			num_of_pos_numbers++;

	num_of_neg_numbers = num_of_numbers - num_of_pos_numbers;

	vec_pos.reserve(num_of_pos_numbers);
	vec_neg.reserve(num_of_neg_numbers);
	vector<int> temp = array_for_sort;
	vector<int>::iterator it_chunk_begin, it_chunk_end;
	it_chunk_begin = temp.begin() + left_border;
	it_chunk_end = temp.begin() + left_border + num_of_numbers;
	copy_if(it_chunk_begin, it_chunk_end, back_inserter(vec_pos), [](int i) { return i >= 0;
});
	copy_if(it_chunk_begin, it_chunk_end, back_inserter(vec_neg), [](int i) { return i < 0;
});

	const int max_elem = get_max(temp, left_border, right_border);
	for (int curr_digit = 1; max_elem / curr_digit > 0; curr_digit *= 10)
		count_sort(vec_pos, 0, num_of_pos_numbers-1, curr_digit);
	for (int curr_digit = 1; max_elem / curr_digit > 0; curr_digit *= 10)
		count_sort(vec_neg, 0, num_of_neg_numbers-1, curr_digit);
	reverse(vec_neg.begin(), vec_neg.end());

	for (int i = left_border,j=0; i < left_border + num_of_neg_numbers; i++,j++)
		array_for_sort[i] = vec_neg[j];
	for (int i = left_border+num_of_neg_numbers, j = 0; i < right_border+1; i++, j++)
		array_for_sort[i] = vec_pos[j];
}
```

Функция *int get_max(vector<int>& array_for_sort, const int left_border, const int right_border)* используется для нахождения в сортируемом фрагменте массива максимальное значение для определения глубины сортировки (кол-во разрядов)

```
int get_max(vector<int>& array_for_sort, const int left_border, const int right_border)
{
	int max = abs(array_for_sort[left_border]);
	for (int i = left_border; i <= right_border; i++)
		if (abs(array_for_sort[i]) > max)
			max = abs(array_for_sort[i]);
	return max;
}
```

А также функция *void count_sort(vector<int>& array_for_sort, const int left_border, const int right_border, int digit)* которая непосредственно сортирует по определенному разряду.

```
void count_sort(vector<int>& array_for_sort, const int left_border, const int right_border, int digit)
{
	vector<int> output(right_border - left_border + 1);
	int count[10] = { 0 };

	for (int i = left_border; i <= right_border; i++)
		count[(abs(array_for_sort[i]) / digit) % 10]++;

	for (int i = 1; i <10; i++)
		count[i] += count[i - 1];

	for (int i = right_border; i >= left_border; i--)
	{
		output[count[(abs(array_for_sort[i]) / digit) % 10] - 1] = array_for_sort[i];
		count[(abs(array_for_sort[i]) / digit) % 10]--;
	}

	for (int i = left_border; i <= right_border; i++)
		array_for_sort[i] = output[i];
}
```

Приложение(*before_code.cpp***):**

У приложения три аргумента:

- имя бинарного файла, представленного в специальном виде (поле времени выполнения типа double, размер сортируемого массива, сортируемый массив);
- имя выходного файла, в который будет записан результат в бинарном виде (поле времени выполнения типа double, размер отсортированного массива, отсортированный массив);
- Количество потоков (0 или не указано для последовательного алгоритма)

```
#define _CRT_SECURE_NO_WARNINGS
#include <cstdio>
#include <ctime>
#include <omp.h>
#include <iostream>
#include <vector>
#include <random>
#include <chrono>
using namespace std;

int get_max(vector<int>& array_for_sort, const int left_border, const int right_border);
void count_sort(vector<int>& array_for_sort, const int left_border, const int right_border, int digit);
void radix_sort_sol(vector<int>& array_for_sort, const int left_border, const int right_border);
void parallel_sort(vector<int>& vector_for_sorting, int _size, int _threads, double& _time);
```

```
int main(int argc, char * argv[])
{
	if (argc != 3) {
		if (argc != 4) {
		cout << "Enter the correct parameters:" << endl;
		cout << "1. The name of the source binary file" << endl;
		cout << "2. The name of the output binary file" << endl;
		cout << "3. The number of threads, type 2^n (if not, the default is a sequential
version of the sort)" << endl;
		return 1;
		}
	}

	int size;
	double time=0;
	int threads = 0;
	freopen(argv[1], "rb", stdin);
	freopen(argv[2], "wb", stdout);
	if (argc == 4) {
		threads = atoi(argv[3]);
	}
	fread(&time, sizeof(time), 1, stdin);
	fread(&size, sizeof(size), 1, stdin);
	vector<int> check(size);
	int *arr = new int[size];
	fread(arr, sizeof(*arr), size, stdin);

	for (int i = 0; i < size; i++)
		check[i] = arr[i];

	if (threads==0) {
		time = omp_get_wtime();
		radix_sort_sol(check, 0, size - 1);
		time = omp_get_wtime() - time;
	}
	else {
	parallel_sort(check, size, threads, time); }

	for (int i = 0; i < size; i++)
		arr[i] = check[i];
	fwrite(&time, sizeof(time), 1, stdout);
	fwrite(&size, sizeof(size), 1, stdout);
	fwrite(arr, sizeof(*arr), size, stdout);
	return 0;
}
```

Generator

Описание: Генератор тестов. Генерирует бинарный файл в виде: поле времени выполнения типа double, размер сортируемого массива, сортируемый массив типа double.
Может сгенерировать 20 тестов.

Запуск: generator <номер теста>

<номер теста> - принимает значения от `1` до `20`

Имя файла будет совпадать с номером теста (от 1 до 20)

Реализация(*generator.cpp*):

```
#define _CRT_SECURE_NO_WARNINGS
#include <cstdio>
#include <random>
#include <chrono>
#include <iostream>

using namespace std;

int n_tests[] = { 1, 2, 3, 4, 5, 6, 8, 10, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000,
1000000, 5000000, 10000000,
50000000};
int main(int argc, char * argv[])
{
	if (argc != 2 || atoi(argv[1]) < 1 || atoi(argv[1]) > 20)
	{
		cout << "You use the \"Generator\" to create tests. Enter the arguments in the
correct format !!" << endl;
		return 1;
	}
	int size = n_tests[atoi(argv[1]) - 1];
	double default_time = 0;

	// перенаправл¤ем поток stdout в файл
	freopen(argv[1], "wb", stdout);

	// создаЄм генератор случайных чисел с seed равным количеству времени с начала
эпохи
	default_random_engine generator(static_cast<unsigned
int>(chrono::system_clock::now().time_since_epoch().count()));
	// создаЄм равномерное распределение случайной величины типа int в диапазоне
	// [-999, 999]
	uniform_int_distribution <int> distribution(-999, 999);

	fwrite(&default_time, sizeof(default_time), 1, stdout); // –езервируем поле дл¤ времени в
создаваемом файле.
	fwrite(&size, sizeof(size), 1, stdout);

	int *arr = new int[size];
	for (int i = 0; i < size; i++)
		arr[i] = distribution(generator);

	fwrite(arr, sizeof(*arr), size, stdout);
	return 0;
}
```

Checker

Описание:

Считывает массив из бинарного файла специального вида (см. пункт **solver** или **generator**) и проверяет его на признак того, что он отсортирован: каждый последующий элемент больше или равен предыдущему.

Запуск:

checker <проверяемый файл>

На выходе будет бинарный файл `result.txt` с результатом выполнения.

Пояснение:

В реализации использовался класс *result*, который был в файле Подготовка_задач_для_автоматизированной_проверки.pdf (остался без изменений) Был использован по требованию к заданию.

Реализация:

```
#define _CRT_SECURE_NO_WARNINGS

#include <cstdio>
#include <cmath>
#include <string>
#include <iostream>

using namespace std;

// Используется для взаимодействия с тестирующей системой
////////////////////////////////////////////////////////////////////////////////////////

// Checker может устанавливать вот эти три вердикта:
//AC = Accepted = Решение выдаёт корректный результат на данном тесте
//WA = Wrong Answer = Решение выдаёт некорректный результат на данном тесте
//PE = Presentation Error = Ошибка формата выходных данных
// Остальные вердикты checker не может устанавливать
//NO = No verdict = Вердикт отсутствует
//CE = Compilation Error = Ошибка компиляции
//ML = Memory Limit Exceeded = Превышено ограничение по памяти
//TL = Time Limit Exceeded = Превышено ограничение по времени работы
//RE = Runtime Error = Ошибка времени исполнения программы
//IL = Idle Limit Exceeded = Превышено время простоя (бездействия) программы
//DE = Deadly Error = Ошибка тестирующей системы

enum verdict { NO = 1, AC, WA, CE, ML, TL, RE, IL, PE, DE };

class result {
private:
	FILE * bur;

public:
	enum ext_cls { NO = 1, VERDICT, MESSAGE, TIME, MEMORY };

	result(bool read = false) {
		if (read)
			bur = fopen("result.txt", "rt");
		else
			bur = fopen("result.txt", "wt");
	}
```

```
	~result() {
		fclose(bur);
	}

	void write_type(ext_cls t) {
		fwrite(&t, sizeof(t), 1, bur);
	}

	void write_verdict(verdict v) {
		write_type(ext_cls::VERDICT);
		fwrite(&v, sizeof(v), 1, bur);
	}

	void write_message(string str) {
		write_type(ext_cls::MESSAGE);
		int l = str.size();
		fwrite(&l, sizeof(l), 1, bur);
		fwrite(&str[0], sizeof(str[0]), l, bur);
	}

	void write_time(long long x) {
		write_type(ext_cls::TIME);
		fwrite(&x, sizeof(x), 1, bur);
	}

	void write_memory(unsigned long long x) {
		write_type(ext_cls::MEMORY);
		fwrite(&x, sizeof(x), 1, bur);
	}
} checker_result;

int main(int argc, char * argv[])
{
	if (argc != 2) {
		cout << "You use the \"Checker\" to check new data. Enter the arguments in the
correct format !!" << endl;
		return 1;
	}

	FILE * buo = fopen(argv[1], "rb");

	int res_size = 0;
	double res_time;

	fread(&res_time, sizeof(res_time), 1, buo);
	fread(&res_size, sizeof(res_size), 1, buo);
	//
	//Создаем флаг для проверки корректности решения.
	//Проверяем по очереди каждый элемент с правым относительно текущего до
конца.
	//При хотя бы одном нарушении меняем флаг и выходим из цикла.
	//Проверка отдельным алгоритмом сортировки потребует больше времени и
наличие исходных данных, а также больше памяти.
	//
```

```
	bool error = false;
	int previous = 0;
	int current = 0;
	fread(&previous, sizeof(previous), 1, buo);
	for (auto i = 1; i < res_size; i++) {
		fread(&current, sizeof(current), 1, buo);
		if (current < previous) {
			error = true;
			break;
		}
		previous = current;
	}
	if (error == false) {
		checker_result.write_message("AC. Array is sorted correctly.");
		checker_result.write_verdict(AC);
	}
	else {
		checker_result.write_message("WA. Array sorting has an error.");
		checker_result.write_verdict(WA);
	}
	checker_result.write_time(static_cast<long long>(res_time * 10000000));

	fclose(buo);
	return 0;
}
```

Тесты

20 тестов в папке *tests*. Имена представлены в виде чисел от 1 до 20. Имена соответствуют номеру теста в генераторе.

И 20 тестов .ans с отсортированными данными.

При загрузке более крупных тестов (если делать еще больше) потребовалось бы много памяти, наличие этих тестов, показывает работоспособность генератора и правильность работы сортировки.

Каждый тест представлен в виде бинарного файла, в котором:

- Поле времени типа double;
- Размер массива;
- Массив.

Размеры массивов:

1, 2, 3, 4, 5, 6, 8, 10, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000, 10000000, 50000000.

Typer

Описание: Преобразователь, конвертирует текстовый файл с временем, размером массива и массивом в бинарный файл специального вида:

- Поле времени типа *double*
- Поле размера массива типа *int*

- Массив типа *int*

Запуск:

typer <входной текстовый файл><выходной бинарный файл>

Реализация:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <vector>
using namespace std;


int main(int argc, char * argv[])
{
	if (argc != 3) {
		cout << "You use the \"Typer\" to process data. Enter the arguments in the correct format !!" << endl;
		return 1;
	}

	char* number_int = new char[15];
	int size = 0;
	double start_time=0;
	vector<int> check;
	FILE *in_txt = fopen(argv[1], "rt");
	FILE *out_bin = fopen(argv[2], "wb");
	if (in_txt != nullptr)
	{
		//Преобразуем строку т.е. каждое отдельное число типа int, и "пушим" его как значение типа int в конец вектора.
		while (fgets(number_int, 15, in_txt) != nullptr)
			check.push_back(atoi(number_int));
		// Фиксируем размер вектора в бинарном формате.
		size = check.size();

		fwrite(&start_time, sizeof(start_time), 1, out_bin);
		fwrite(&size, sizeof(size), 1, out_bin);
		for (int i = 0; i < size; ++i)
			fwrite(&check[i], sizeof(int), 1, out_bin);
	}
	fclose(in_txt);
	fclose(out_bin);
	delete[] number_int;
	return 0;
}
```

Viewer

Описание: Преобразователь, конвертирует бинарный файл специального вида:

- Поле времени типа *double*
- Поле размера массива типа *int*

- Массив типа *int*

в текстовый файл с временем, размером массива и массивом

Запуск:

viewer <входной бинарный файл><выходной текстовый файл>

Реализация:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <vector>
using namespace std;

int main(int argc, char * argv[])
{
	if (argc != 3) {
		cout << "You use the \"Viewer\" to process data. Enter the arguments in the correct format !!" << endl;
		return 1;
	}
	int size;
	double real_time;
	freopen(argv[1], "rb", stdin);
	freopen(argv[2], "wt", stdout);

	fread(&real_time, sizeof(real_time), 1, stdin);
	fread(&size, sizeof(size), 1, stdin);
	cout <<real_time << endl;
	cout << size << endl;
	int *arr = new int[size];
	fread(arr, sizeof(*arr), size, stdin);
	for (int i = 0; i < size; ++i)
		cout << arr[i] << endl;
	return 0;
}
```

Общая идея и структура параллельных версий

Реализованные OpenMP и TBB версии имеют одинаковую структуру, поэтому их можно представить в виде одной схемы:

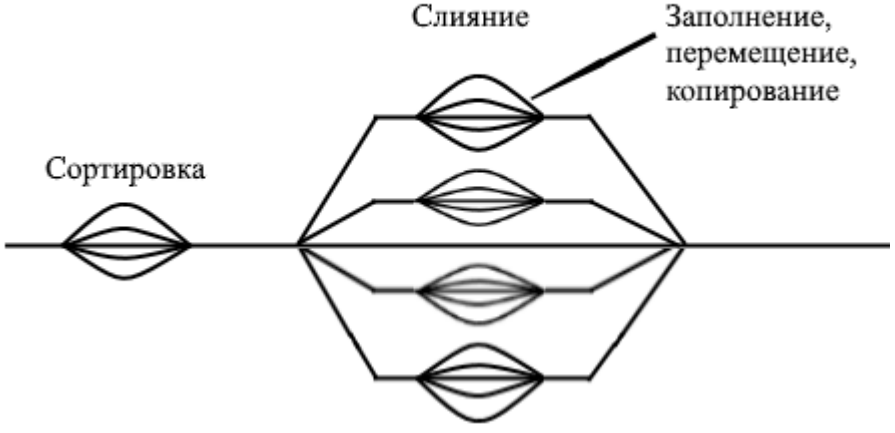


Рис. 2. Схема распараллеливания.

На этапе «сортировка» в OpenMP сразу массив делится на куски в количестве равном количеству потоков, в TBB при создании дочерних «заданий» когда размер «подмассива» соответствует необходимому разделению на потоки и в обоих случаях «подмассивы» сортируются последовательной версией сортировки (**solver**). После этой части весь массив элементов состоит из отсортированных кусков-«подмассивов».

На участке «слияние» происходит соответственное последовательное слияние подмассивов (1-2,3-4, и т.д.).

«Слияние» в данном случае подразумевает слияние четных и нечетных элементов подмассивов отдельно и их соответственное (четно-нечетное) слияние.

Схема слияния:

Первый подмассив	-1	10	25	31	48	50	52	64	Второй подмассив	-8	-5	0	2	7	10	19	60
Массив нечетных элементов	-8	-1	0	7	19	25	48	52	Массив четных элементов	-5	2	10	10	31	50	60	64
Почти отсортированный массив	-8	-5	-1	2	0	10	7	10		19	31	25	50	48	60	52	64
Отсортированный массив	-8	-5	-1	0	2	7	10	10		19	25	31	48	50	52	60	64

Рис.3. Схема слияния

Параллельная версия OpenMP

Описание: Реализация поразрядной сортировки для целых чисел с четно-нечетным слиянием Бэтчера с использованием средств OpenMP (Parallel.cpp и before_code.cpp)

Запуск: Команда в командной строке (или Powershell)
<имя скомпилированной программы> <входной бинарный файл> <выходной бинарный файл><Количество потоков>

P.s. <входной бинарный файл> - должен существовать.

Пояснение: сам запуск приложения также производится через *before_code.cpp* аналогично solver версии (См. «Приложение» в главе Solver)

Реализация(*Parallel.cpp***):** Вызываемая функция сортировки называется *void parallel_sort(std::vector<int>& vector_for_sorting, int _size, int _threads, double& _time)*, в

которой первый параметр – сортируемый вектор, второй – размер сортируемого вектора, третий – количество потоков, четвертый – время.

Общая структура: сортируем последовательной версией столько частей сколько потоков, далее делаем слияние, пока шагов будет меньше потоков. Шаг рассчитывается специальной формулой. Внутри слияния делаем все по Рис.3.

P.s. алгоритм сделан так чтобы не создавать отдельно каждый проход/шаг векторы для каждого из подмассивов, вместо этого один раз идет конвертация из вектора в массив перед обработкой и из массива в вектор после обработки, это позволяет сэкономить память. Последовательная версия используется из **solver**

```
void parallel_sort(std::vector<int>& vector_for_sorting, int _size, int _threads, double& _time)
{
	int size = _size, threads = _threads;
	std::vector<int> vec_for_sort = vector_for_sorting;

	int* arr = new int[size];
	int step;
	std::vector<int>* typed_array = new std::vector<int>[threads];
	int* shift = new int[threads];
	int* dimension = new int[threads];
	double time = omp_get_wtime();
#pragma omp parallel shared(arr, step, shift, dimension, typed_array) num_threads(threads)
	{
		int t_id, thread_change;
		t_id = omp_get_thread_num();

		shift[t_id] = t_id * (size / threads);
		if (t_id == threads - 1)
			dimension[t_id] = size - t_id * (size / threads);
		else
			dimension[t_id] = size / threads;
		radix_sort_sol(vec_for_sort, shift[t_id], shift[t_id] + dimension[t_id] - 1);
#pragma omp barrier
#pragma omp single
		{
			for (int k = 0; k < size; k++) {
				arr[k] = vec_for_sort[k];
				//std::cout << vec_for_sort[k];
			}
			//std::cout << std::endl;
		}
		step = 1;
		while (step < threads)
		{
			thread_change = step;

			if (t_id % (thread_change * 2) == 0)
				select_splitter(EVEN, arr + shift[t_id], dimension[t_id], arr +
shift[t_id + thread_change], dimension[t_id + thread_change],
```

```
                                        typed_array[t_id]);
                        else if (t_id % thread_change == 0)
                                select_splitter(ODD, arr + shift[t_id], dimension[t_id], arr +
shift[t_id - thread_change], dimension[t_id - thread_change],
                                        typed_array[t_id]);
#pragma omp barrier
                        if (t_id % (thread_change * 2) == 0)
                        {
                                MergeAndSort(typed_array[t_id], typed_array[t_id +
thread_change], arr + shift[t_id]);
                                dimension[t_id] += dimension[t_id + thread_change];
                                typed_array[t_id].clear();
                                typed_array[t_id].shrink_to_fit();
                                typed_array[t_id + thread_change].clear();
                                typed_array[t_id + thread_change].shrink_to_fit();
                        }
#pragma omp single
                        {
                                step *= 2;
                        }
#pragma omp barrier
                }
        }
        _time = omp_get_wtime() - time;
        delete[] typed_array;
        delete[] dimension;
        delete[] shift;
        for (int k = 0; k < size; k++) {
                vector_for_sorting[k] = arr[k];
        }
}
```

Функция *void MergeAndSort(const std::vector<int> vec_one, const std::vector<int> vec_two, int* output_sorted_array)* используется для простого слияния массивов четных и нечетных элементов и финальное «причесывание» выходного массива

```
void MergeAndSort(const std::vector<int> vec_one, const std::vector<int> vec_two, int*
output_sorted_array)
{
        int i = 0, j = 0;
        int size1 = vec_one.size(), size2 = vec_two.size();

        while (i < size1 && j < size2) {
                output_sorted_array[i + j] = vec_one[i];
                output_sorted_array[i + j + 1] = vec_two[j];
                ++i; ++j;
        }

        while (i < size1) {
                output_sorted_array[size2 + i] = vec_one[i];
                i++;
        }
        while (j < size2) {
```

```
            output_sorted_array[size1 + j] = vec_two[j];
            j++;
        }
        //
        //Чтобы массив стал окончательно отсортированным, достаточно сравнить
        //пары элементов, стоящие на нечётной и чётной позициях.
        //Первый и последний элементы массива проверять не надо,
        //т.к.они являются минимальным и максимальным элементами массивов.
        //Взято из пояснения к алгоритму слияния.
        //
        i = 1;
        while (i < size1 + size2 - 1) {
            if (output_sorted_array[i] > output_sorted_array[i + 1]) {
                j = output_sorted_array[i];
                output_sorted_array[i] = output_sorted_array[i + 1];
                output_sorted_array[i + 1] = j;
            }
            ++i;
        }
}
```

Функция *void select_splitter(elemType type, const int* array_one, int size_one, const int* array_two, int size_two, std::vector<int>& result)* используется для выделения из двух массивов либо четных либо нечетных элементов в упорядоченном виде по enum EVEN или ODD

```
enum elemType {
        EVEN,
        ODD
};
//
//Выделение из двух массивов четных или нечетных элементов (в упорядоченном виде)
//
void select_splitter(elemType type, const int* array_one, int size_one, const int* array_two, int size_two, std::vector<int>& result) {
        int i, j;
        if (type == EVEN)
        {
            i = 0;
            j = 0;
        }
        else
        {
            i = 1;
            j = 1;
        }
        result.reserve(size_one + size_two);
        while (i < size_one && j < size_two) {
            if (array_one[i] <= array_two[j])
            {
                result.push_back(array_one[i]);
                i += 2;
            }
```

```
            else
            {
                result.push_back(array_two[j]);
                j += 2;
            }
        }
        if (i >= size_one)
            while (j < size_two) {
                result.push_back(array_two[j]);
                j += 2;
            }
        else
            while (i < size_one) {
                result.push_back(array_one[i]);
                i += 2;
            }
}
```

Параллельная версия TBB

Описание: Реализация поразрядной сортировки для целых чисел с четно-нечетным слиянием Бэтчера с использованием средств TBB (ParallelTBB.cpp)

Запуск: Команда в командной строке (или Powershell)
<имя скомпилированной программы> <входной бинарный файл> <выходной бинарный файл><Количество потоков>

P.s. <входной бинарный файл> - должен существовать.

Реализация(ParallelTBB.cpp**):** Вызываемая функция сортировки называется *void Start_TBB_Sorting(int *arr, int size, int threads)*, в которой первый параметр – сортируемый массив, второй – размер сортируемого массива, третий – количество потоков.

Общая структура: Запускаем корневое задание, которое, по сути, рекурсивно вызовет столько дочерних задний сколько есть потоков и когда заданная порция (которая зависит от кол-ва потоков) будет больше числа элементов в подмассиве запустится последовательная версия сортировки из solver. Далее созданные задания также, по сути, делают слияния аналогично пункту **«Параллельная версия OpenMP»**

P.s. В алгоритме используется функция-конвертер которая преобразует массив в вектор и наоборот. Она необходима т.к. получилось реализовать программу только используя массив, а solver версия рассчитана на вектор. Последовательная версия используется из **solver**

```
void Start_TBB_Sorting(int *arr, int size, int threads)
{
    int portion = size / threads;
```

```
        if (size%threads != 0)
                portion++;

        SortingBody& sorter = *new (task::allocate_root())
                SortingBody(arr, size, portion);
        task::spawn_root_and_wait(sorter);
}
```

Класс используемый в работе *class SortingBody* необходим для запуска самого тела сортировки: таких же заданий для следующих уровней, выделения массивов четных и нечетных элементов и их распределения в массив, а также запуск последовательной сортировки для кусков необходимого размера.

```
class SortingBody : public task
{
private:
        int *arr;
        int size;
        int portion;

public:
        SortingBody(int *_arr, int _size, int _portion) : arr(_arr), size(_size), portion(_portion)
{}

        task* execute()
        {
                if (size <= portion)
                {
                        transformed_sort(arr, size);
                }
                else
                {
                        int s = size / 2;
                        SortingBody &sorter_one = *new (allocate_child()) SortingBody(arr, s,
portion);
                        SortingBody &sorter_two = *new (allocate_child()) SortingBody(arr + s,
size - s, portion);
                        set_ref_count(3);
                        spawn(sorter_one);
                        spawn_and_wait_for_all(sorter_two);
                        EvenSelector &evenSelector = *new (allocate_child()) EvenSelector(arr,
s, size - s);
                        OddSelector &oddSelector = *new (allocate_child()) OddSelector(arr, s,
size - s);
                        set_ref_count(3);
                        spawn(evenSelector);
                        spawn_and_wait_for_all(oddSelector);

                        parallel_for(blocked_range<int>(1, size), Comparator(arr));
                }
                return NULL;
        }
};
```

Классы *class OddSelector* и *class EvenSelector* каждый из которых выбирает из массивов нечетные и четные элементы соответственно в упорядоченном виде и сразу же распределяет их по нечетным и четным местам массива соответственно.

```
class EvenSelector : public task
{
private:
	int *arr;
	int size1, size2;

public:
	EvenSelector(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;
	}
};

class OddSelector : public task
{
private:
	int *arr;
	int size1, size2;
```

```
public:
	OddSelector(int* _arr, int _size1, int _size2) : arr(_arr), size1(_size1), size2(_size2) {
	}

	task* execute()
	{
		int* arr2 = arr + size1;

		int num = (size1 + size2) - (size1 + size2 + 1) / 2;
		int* tmp = new int[num];

		int a = 1, b = 1, 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 + 1] = tmp[j];
		return NULL;
	}
};
```

Функция *void transformed_sort(int* arr, int size)* производящая конвертирование и сортировку последовательной версией

```
void transformed_sort(int* arr, int size)
{
	tick_count start = tick_count::now();
	vector<int> convert(size);
	for (int i = 0; i < size; i++)
		convert[i] = arr[i];
	tick_count f_st = tick_count::now();
	radix_sort_sol(convert, 0, size - 1);
	tick_count f_fin = tick_count::now();
	for (int i = 0; i < size; i++)
		arr[i]= convert[i];
	tick_count finish = tick_count::now();
	time_error+=((finish - start) - (f_fin - f_st)).seconds();
}
```

Класс *class Comparator* необходимый для пробега по массиву и проверки что массив отсортирован, если соседние элементы стоят неправильно он их обменивает

```
class Comparator
{
private:
	int *arr;
public:
	Comparator(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;
			}
	}
};
```

Приложение:

У приложения три аргумента:

- имя бинарного файла, представленного в специальном виде (поле времени выполнения типа double, размер сортируемого массива, сортируемый массив);
- имя выходного файла, в который будет записан результат в бинарном виде (поле времени выполнения типа double, размер отсортированного массива, отсортированный массив);
- Количество потоков

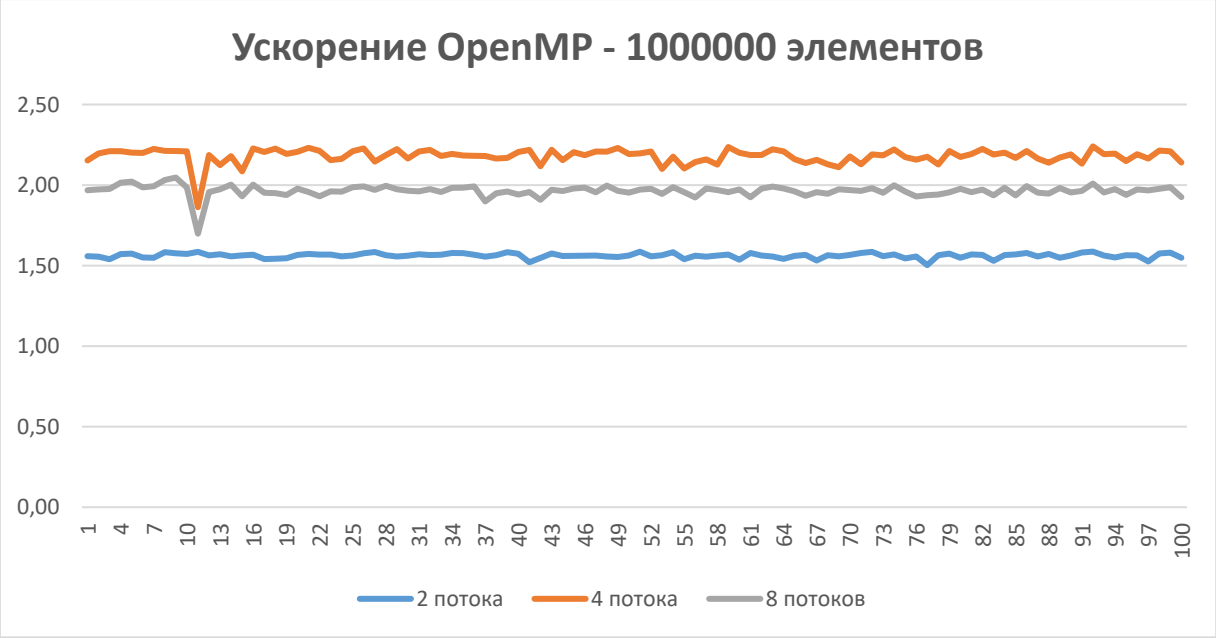
```
int main(int argc, char * argv[])
{
	if (argc != 4) {
		cout << "Enter the correct parameters:" << endl;
		cout << "1. The name of the source binary file" << endl;
		cout << "2. The name of the output binary file" << endl;
		cout << "3. The number of threads" << endl;
		return 1;
	}

	int size, threads = atoi(argv[3]);
	int* arr;
	double fict_time;
	time_error = 0;

	freopen(argv[1], "rb", stdin);
	fread(&fict_time, sizeof(fict_time), 1, stdin);
```

```
fread(&size, sizeof(size), 1, stdin);

arr = new int[size];
fread(arr, sizeof(*arr), size, stdin);
double time;

tick_count start_main = tick_count::now();
Start_TBB_Sorting(arr, size, threads);
tick_count finish_main = tick_count::now();

time =(finish_main-start_main).seconds()-time_error;
printf("TBB sorting time: %f\n", time);

freopen(argv[2], "wb", stdout);
fwrite(&time, sizeof(time), 1, stdout);
fwrite(&size, sizeof(size), 1, stdout);
fwrite(arr, sizeof(*arr), size, stdout);
return 0;
```

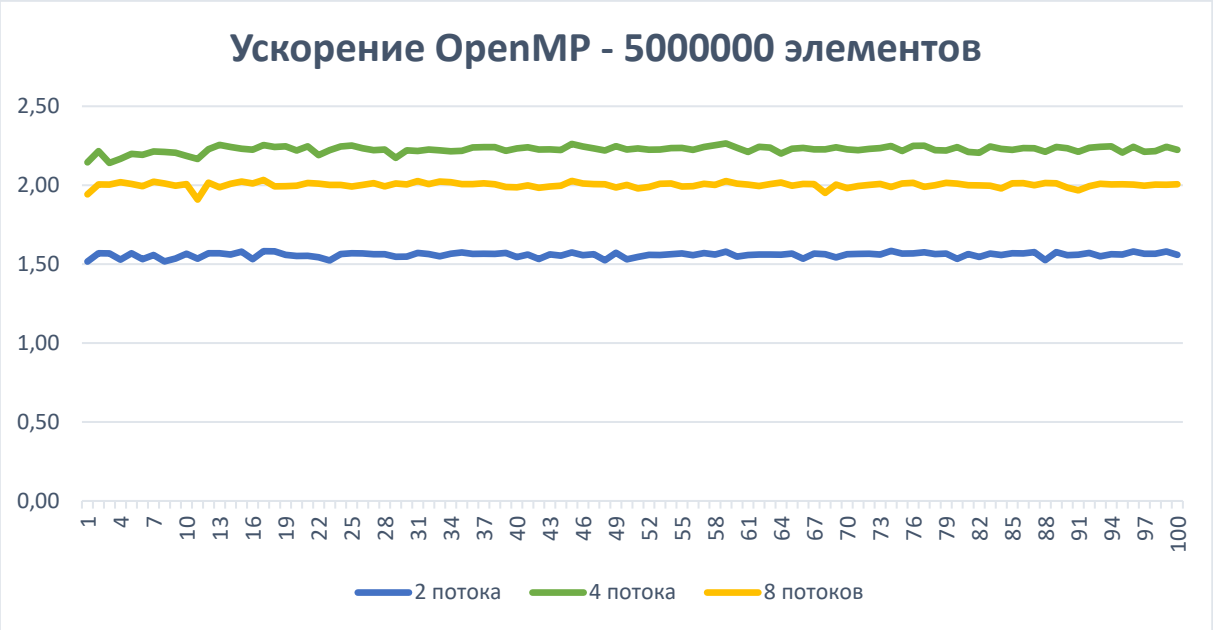
Статистика:

Сортировка проводилась для массивов из 5 млн. и из 1 млн. элементов с использованием двух, четырех и восьми потоков на 100 разных тестах. И соответственно с использованием OpenMP и TBB инструментов. По графикам ниже можно отследить изменение ускорения по каждой из групп (кроме числа элементов по причинам, описанным ниже)

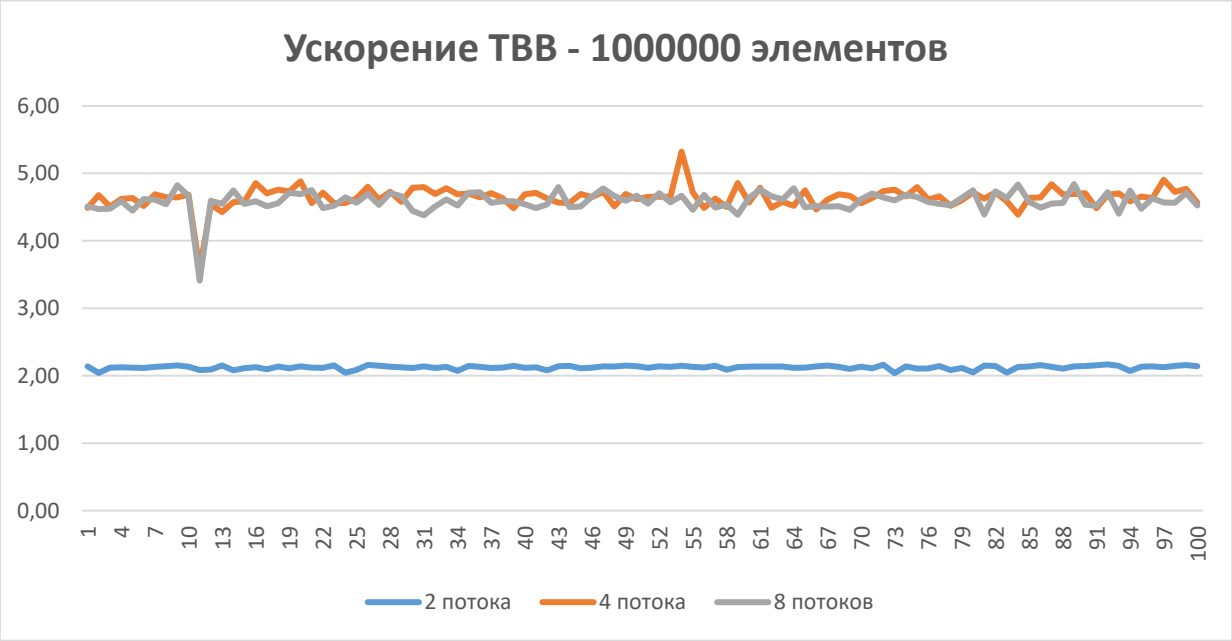
Как видно из графика оптимальным количеством потоков для наилучших показателей ускорения у OpenMP в среднем является 4 потока, причина может быть, как в аппаратных составляющих (4-х ядерный процессор), так и в возможно малом количестве элементов.



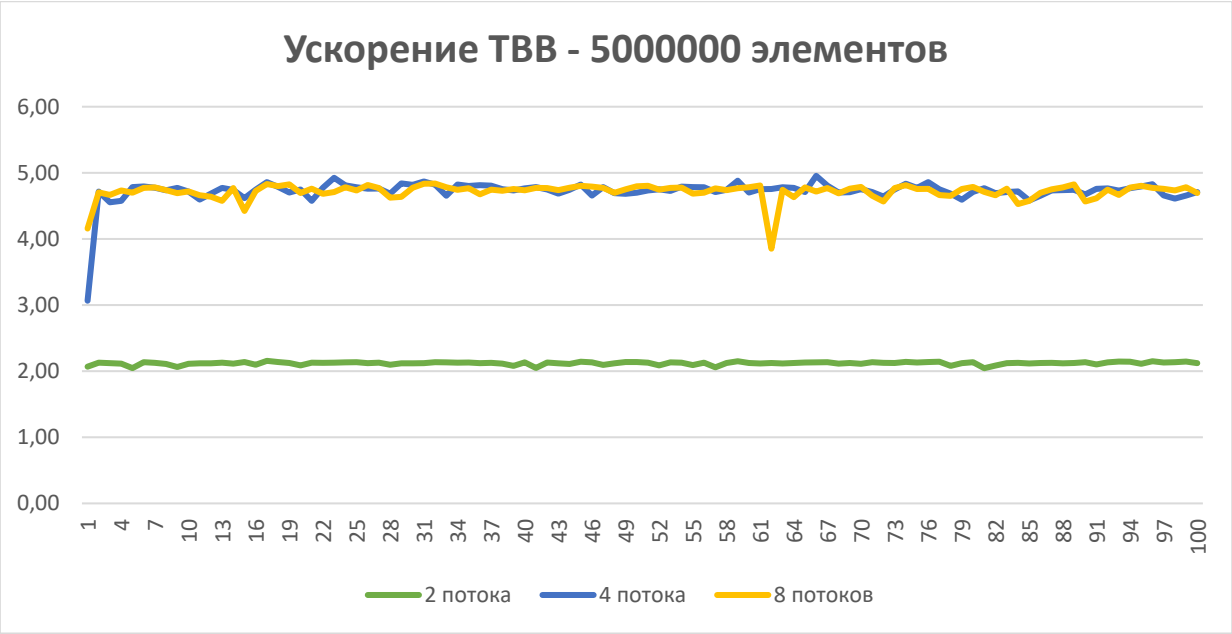
Такая же ситуация и при большем количестве элементов



Так мы можем наблюдать что небольшое (всего в 5 раз) увеличение ничуть не меняет поведение ускорения и также лучше всего алгоритм показал себя на 4-х потоках.

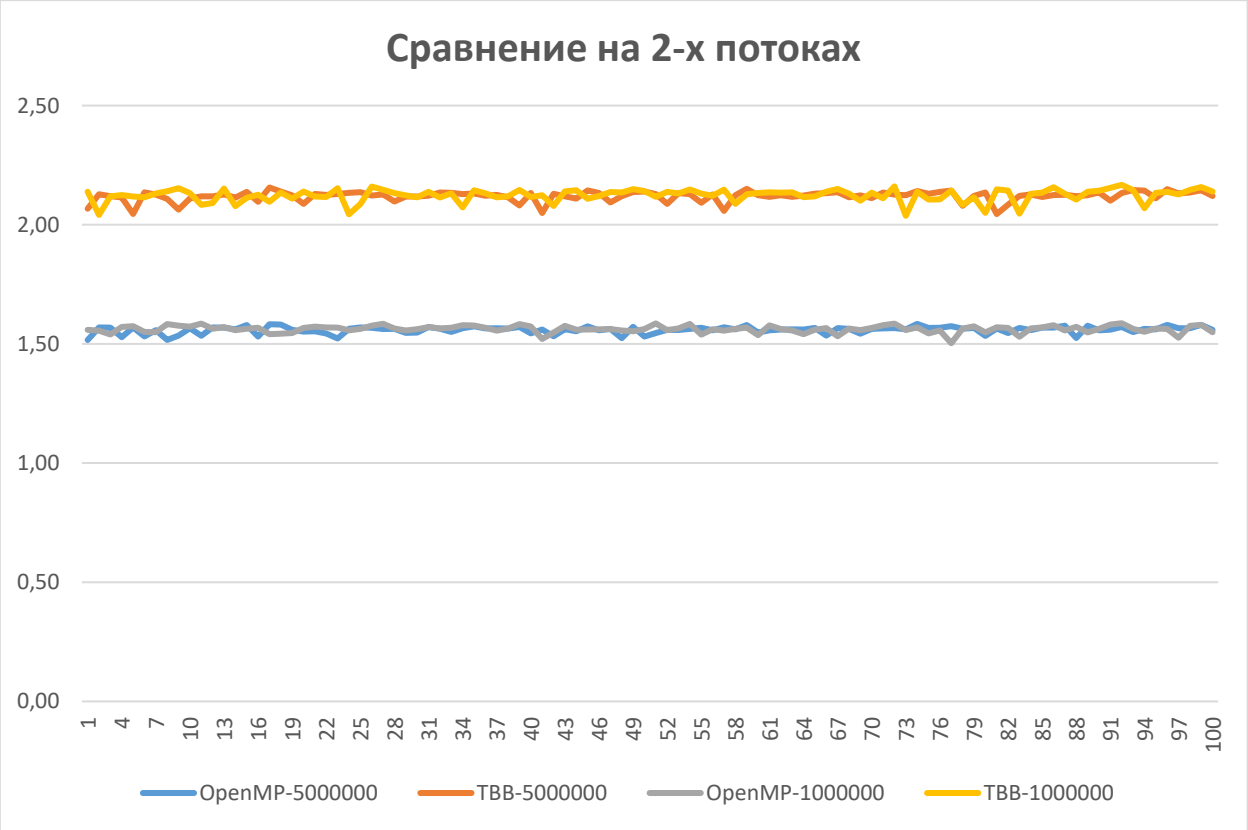


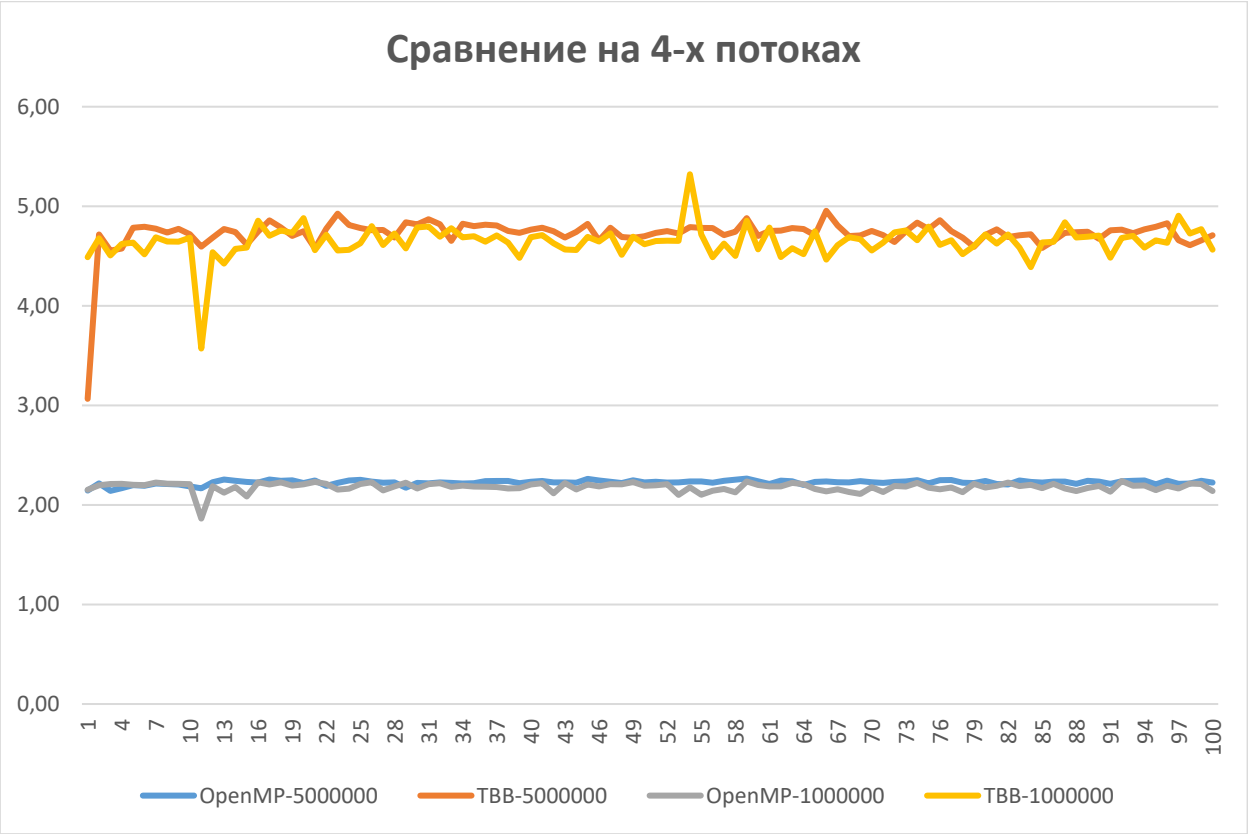
У средств TBB похожая ситуация, хотя, здесь отрыв 4 потоков не так очевиден, однако он все же есть. Так же можно заметить, что более высокое ускорение при 4-8 потоках хотя и выдает хорошие результаты, весьма нестабильно и может выдавать как хорошие показатели – 5.32, так и не очень – 3.41. В целом же ускорение весьма приличное для проводимого опыта.

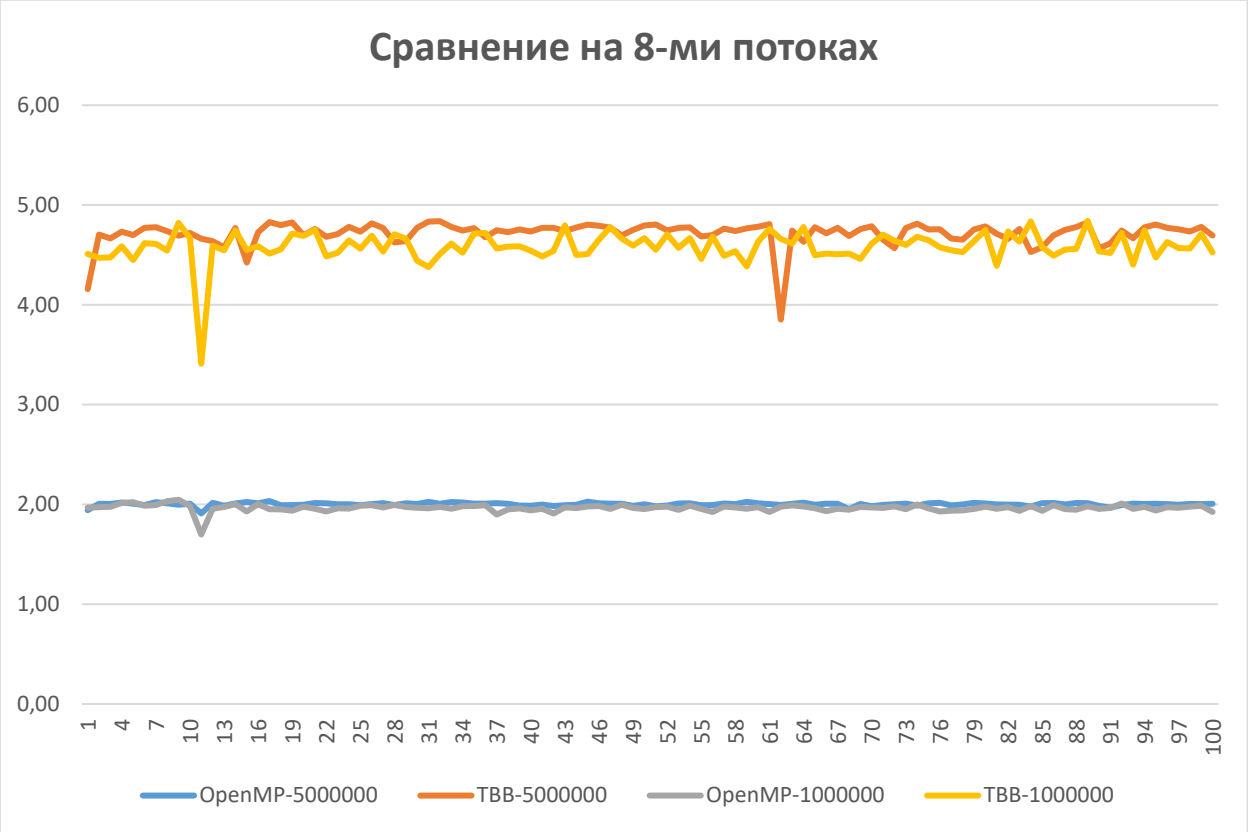


И, как и у OpenMP у TBBнаблюдается так же отсутствие каких-либо сильных зависимостей от размера (однако опять же возможно дело в ограничениях «железа» или в «малом» количестве элементов). Однако, в целом само поведение ускорения здесь более спокойное чем на предыдущем графике (несмотря на наличие экстремумов)

Также интересно выяснить какие из средств параллельного программирования – TBB или OpenMP оказались на высоте в данном алгоритме.







Посмотрев на эти графики сразу становится очевидно, что во всех случаях более высокие показатели у средств TBB, и даже если сравнивать на разных размерах массивов, картина не меняется. И на 2-х, и на 4-х, и на 8-ми потоках TBB более эффективно выполняли сортировку и можно заметить, что при увеличении числа потоков ускорение у TBB становится не в (среднем) 1,33 больше, а стабильно в 2, а бывает и более (до 2,5) раз больше

Вывод

В ходе работы была реализована **Поразрядная сортировка для целых чисел с четно-нечетным слиянием Бэтчера** в последовательной и 2-х параллельных версиях, изучены и опробованы такие средства параллельной разработки как ***Open Multi-Processing(OpenMP)*** и ***Intel Threading Building Blocks(TBB)*** а также были написаны программы помогающие структурировать работу с сортировкой и сделать ее более простой для оценки результатов (generator, checker, typer, viewer, и скрипты для полуавтоматической обработки, которые нигде не указаны но очень помогли провести 1400 измерений)

По самой сортировке видно, что ее эффективность вполне соответствует теоретическим ожиданиям, а также видно, что сама последовательная сортировка (Восходящая поразрядная) очень устойчивая и генерация исходных данных массивов не влияют на результат (нет худшего случая при случайном распределении данных)

Также сама параллельная часть т.е. слияние тоже является очень неплохим для варианта слияний.

Масштабируемость по данным:
Сложно оценить, т.к. не хватает данных, и при текущих условиях сложно более глубоко проверить масштабируемость. По текущим же данным при росте размера массива ускорение остается в целом стабильным.

Масштабируемость по потокам:
По результатам эксперимента видно, что при увеличении числа потоков для данной задачи, до 4 ускорение алгоритма быстро растет, при увеличении до 8 либо немного падает, либо остается на том же уровне. Скорее всего причина в том, как именно аппаратная часть реализует работу при восьми потоках на 4 ядрах. (Скорее всего идет обработка как будто бы в 4 потока)

В результате работы была написана программа (набор программ), позволяющая создавать случайные массивы, сортировать их, как последовательным, так и параллельным алгоритмом, а также вести обработку полученных по ним данных. Для любого количества данных и потоков алгоритм является корректным. Проведены серии экспериментов с анализом масштабируемости, показывающие ускорение параллельного алгоритма, достоинства и важность изучения методов параллельных вычислений.