# МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

## федеральное государственное автономное образовательное учреждение высшего образования

## **«Национальный исследовательский Нижегородский государственный**

## **университет им. Н. И. Лобачевского»**

## **Институт информационных технологий, математики и механики**

*Отчёт по лабораторной работе*

**"Сортировка Хоара с четно-нечетным слиянием Бэтчера"**

**Выполнила**: студентка гр. 381708-2

Коврижных М. А.

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

**Проверил**: ассистент кафедры МОСТ

прикладной математики и информатики

Волокитин В. Д.

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

## Нижний Новгород, 2020 г.

### Содержание

[Введение 2](#_Toc42950409)

[Постановка задачи 3](#_Toc42950410)

[Описание алгоритма 4](#_Toc42950411)

[Схема распараллеливания 5](#_Toc42950412)

[Эксперименты 6](#_Toc42950413)

[Заключение 7](#_Toc42950414)

[Литература 8](#_Toc42950415)

[Приложение 9](#_Toc42950416)

### Введение

#### **Быстрая сортировка, сортировка Хоара** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) quicksort), часто называемая qsort (по имени в [стандартной библиотеке языка Си](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D0%B0%D1%8F_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0_%D0%A1%D0%B8)) — [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8), разработанный английским информатиком [Чарльзом Хоаром](https://ru.wikipedia.org/wiki/%D0%A5%D0%BE%D0%B0%D1%80,_%D0%A7%D0%B0%D1%80%D0%BB%D1%8C%D0%B7_%D0%AD%D0%BD%D1%82%D0%BE%D0%BD%D0%B8_%D0%A0%D0%B8%D1%87%D0%B0%D1%80%D0%B4) в [1960 году](https://ru.wikipedia.org/wiki/1960_%D0%B3%D0%BE%D0%B4).

#### [Один из самых быстрых](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C) известных универсальных алгоритмов сортировки массивов.

#### QuickSort является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы.

#### Общая идея алгоритма состоит в следующем:

* 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.
* 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».
* Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

#### На практике массив обычно делят не на три, а на две части: например, «меньшие опорного» и «равные и большие»; такой подход в общем случае эффективнее, так как упрощает алгоритм разделения.

#### **Чётно-нечётная сортировка** — относительно простой алгоритм сортировки, разработанный для использования на параллельных процессорах, является модификацией [пузырьковой сортировки](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BF%D1%83%D0%B7%D1%8B%D1%80%D1%8C%D0%BA%D0%BE%D0%BC). Суть модификации в том, чтобы сравнивать элементы массива под чётными и нечётными индексами с последующими элементами независимо. Алгоритм был впервые представлен Н. Хаберманом (N. Haberman) в 1972 году.

#### Заводится флаг, определяющий отсортирован ли массив. В начале итерации ставится в состояние «истина», далее каждый нечётный элемент сверяется с последующим и если они стоят в неправильном порядке (предыдущий больше следующего), то они меняются местами, и флаг ставится в состояние «ложь». То же самое делается с чётными элементами. Алгоритм не прекращает работу, пока флаг не останется в состоянии «истина».

### Постановка задачи

#### Реализовать последовательную и параллельную работу сортировки Хоара с чётно-нечётным слиянием Бэтчера.

### Описание программной реализации

#### В данной лабораторной работе требуется реализовать последовательную и параллельную работу сортировки Хоара с чётно-нечётным слиянием Бэтчера. Механизм распараллеливания осуществляется при помощи технологий OpenMP и TBB.

### Схема распараллеливания

#### Основная задача состоит в выборе одним из процессоров ведущего элемента, который далее рассылается всем остальным процессам. После получения ведущего элемента процессы проводят разделение своих блоков, и получаемые части блоков передаются между попарно связанными процессами. В результате выполнения подобной итерации исходный массив оказывается разделенным на 2 массива меньшей размерности.

#### Оптимальная ситуация состоит в выборе такого значения ведущего элемента, при котором данные на процессорах разделяются на части одинакового размера. В общем случае, при произвольно сгенерированных исходных данных достижение такой ситуации является достаточно сложной задачей. Для этого предлагается выбирать ведущий элемент, например, как среднее арифметическое элементов на управляющем процессе. Тогда данные на каждом процессоре предварительно упорядочиваются с тем, чтобы взять средний элемент блока данных как ведущее значение.

### Эксперименты

#### По данным экспериментов видно, что при увеличении числа процессов параллельной программы, время выполнения сортировки сокращается.

#### 

### Заключение

#### В рамках лабораторной работы была реализована последовательная и параллельная сортировка Хоара с чётно-нечётным слиянием Бэтчера.

### Литература

* + 1. Кнут Д. «Искусство программирования»
    2. Страуструп Б. «Язык программирования С++»
    3. https://ru.wikipedia.org/wiki/Быстрая\_сортировка
    4. https://ru.wikipedia.org/wiki/Сортировка\_чёт-нечет

### Приложение

Последовательный код:

|  |
| --- |
|  |
|  | #include<vector>  #include <cstdlib> |
|  | #include <time.h> |
|  | #include <stdio.h> |
|  |  |
|  | void quickSort(std::vector <double>\* vector, int left, int right) { |
|  | while (right > left) |
|  | { |
|  | int iterate\_left = left; |
|  | int iterate\_right = right; |
|  | double pivot = (\*vector)[(left + right) >> 1]; |
|  |  |
|  | while (iterate\_left <= iterate\_right) { |
|  | while ((\*vector)[iterate\_left] < pivot) |
|  | { |
|  | iterate\_left += 1; |
|  | } |
|  | while ((\*vector)[iterate\_right] > pivot) |
|  | { |
|  | iterate\_right -= 1; |
|  | } |
|  | if (iterate\_left <= iterate\_right) |
|  | { |
|  | std::swap((\*vector)[iterate\_left], (\*vector)[iterate\_right]); |
|  | iterate\_left += 1; |
|  | iterate\_right -= 1; |
|  | } |
|  | } |
|  |  |
|  | if ((iterate\_left << 1) > left + right) |
|  | { |
|  | quickSort(vector, iterate\_left, right); |
|  | right = iterate\_left - 1; |
|  | } |
|  | else |
|  | { |
|  | quickSort(vector, left, iterate\_left - 1); |
|  | left = iterate\_left; |
|  | } |
|  | } |
|  | } |
|  |  |
|  | void randomVec(int size, std::vector <double>\* vector) |
|  | { |
|  | int i = 0; |
|  | double number; |
|  | while (i < size) |
|  | { |
|  | number = rand() / (RAND\_MAX + 1.0); |
|  | (\*vector).push\_back(number); |
|  | i += 1; |
|  | } |
|  | } |
|  |  |
|  | void isSorted(std::vector <double>\* vector) |
|  | { |
|  | for (int i = 0; i < (\*vector).size() - 1; i++) |
|  | { |
|  | if ( (\*vector)[i] > (\*vector)[i+1]) {printf("Incorrect sort %f %f", (\*vector)[i], (\*vector)[i + 1]); return;} |
|  | } |
|  | printf("Correct sort"); |
|  | } |
|  |  |
|  | int main() |
|  | { |
|  | srand(time(NULL)); |
|  |  |
|  | int n = 100000; |
|  | std::vector <double> vec; |
|  |  |
|  | randomVec(n, &vec); |
|  | quickSort(&vec, 0, n - 1); |
|  | isSorted(&vec); |
|  |  |
|  | return 0; |
|  | } |

OpenMP:

|  |
| --- |
|  |
|  | #include <algorithm>  #include <cstdlib> |
|  | #include <time.h> |
|  | #include <stdio.h> |
|  | #include <omp.h> |
|  |  |
|  | void quickSort(double\* array, int left, int right) |
|  | { |
|  | while (right > left) |
|  | { |
|  | int iterate\_left = left; |
|  | int iterate\_right = right; |
|  | double pivot = array[(left + right) >> 1]; |
|  |  |
|  | while (iterate\_left <= iterate\_right) |
|  | { |
|  | while (array[iterate\_left] < pivot) |
|  | { |
|  | iterate\_left += 1; |
|  | } |
|  | while (array[iterate\_right] > pivot) |
|  | { |
|  | iterate\_right -= 1; |
|  | } |
|  | if (iterate\_left <= iterate\_right) |
|  | { |
|  | double temp = array[iterate\_left]; |
|  | array[iterate\_left] = array[iterate\_right]; |
|  | array[iterate\_right] = temp; |
|  |  |
|  | iterate\_left += 1; |
|  | iterate\_right -= 1; |
|  | } |
|  | } |
|  |  |
|  | if ((iterate\_left << 1) > left + right) |
|  | { |
|  | quickSort(array, iterate\_left, right); |
|  | right = iterate\_left - 1; |
|  | } |
|  | else |
|  | { |
|  | quickSort(array, left, iterate\_left - 1); |
|  | left = iterate\_left; |
|  | } |
|  | } |
|  | } |
|  |  |
|  | void even(double\* array, double\* tmp, int left\_part, int right\_part) |
|  | { |
|  | int i = 0, j; |
|  | while(i < left\_part) |
|  | { |
|  | tmp[i] = array[i]; |
|  | i += (1 << 1); |
|  | } |
|  |  |
|  | double\* array2 = array + left\_part; |
|  | int a = 0, b = 0; |
|  | i = a; |
|  |  |
|  | for (i; (a < left\_part) && (b < right\_part); i += (1 << 1)) |
|  | { |
|  | array[i] = tmp[a]; |
|  |  |
|  | if (tmp[a] <= array2[b]) |
|  | { |
|  | a += (1 << 1); |
|  | } |
|  | else |
|  | { |
|  | array[i] = array2[b]; |
|  | b += (1 << 1); |
|  | } |
|  | } |
|  |  |
|  | j = b; |
|  | if (a == left\_part) |
|  | { |
|  | while(j < right\_part) |
|  | { |
|  | array[i] = array2[j]; |
|  | j += (1 << 1); |
|  | i += (1 << 1); |
|  | } |
|  | } |
|  | else |
|  | { |
|  | j = a; |
|  | while(j < left\_part) |
|  | { |
|  | array[i] = tmp[j]; |
|  | j += (1 << 1); |
|  | i += (1 << 1); |
|  | } |
|  | } |
|  | } |
|  |  |
|  | void odd(double\* array, double\* tmp, int left\_part, int right\_part) |
|  | { |
|  | int i = 1, j; |
|  | while(i < left\_part) |
|  | { |
|  | tmp[i] = array[i]; |
|  | i += (1 << 1); |
|  | } |
|  |  |
|  | double\* array2 = array + left\_part; |
|  | int a = 1, b = 1; |
|  | i = a; |
|  |  |
|  | for (i; (a < left\_part) && (b < right\_part); i += (1 << 1)) |
|  | { |
|  | array[i] = tmp[a]; |
|  |  |
|  | if (tmp[a] <= array2[b]) |
|  | { |
|  | a += (1 << 1); |
|  | } |
|  | else |
|  | { |
|  | array[i] = array2[b]; |
|  | b += (1 << 1); |
|  | } |
|  | } |
|  |  |
|  | j = b; |
|  | if (a == left\_part) |
|  | { |
|  | while(j < right\_part) |
|  | { |
|  | array[i] = array2[j]; |
|  | j += (1 << 1); |
|  | i += (1 << 1); |
|  | } |
|  | } |
|  | else |
|  | { |
|  | j = a; |
|  | while(j < left\_part) |
|  | { |
|  | array[i] = tmp[j]; |
|  | j += (1 << 1); |
|  | i += (1 << 1); |
|  | } |
|  | } |
|  | } |
|  |  |
|  | void quick(double\* array, double\* tmp, int size, int part) |
|  | { |
|  | if (size <= part) |
|  | { |
|  | quickSort(array, 0, size - 1); |
|  | } |
|  | else |
|  | { |
|  | int divide = size >> 1; |
|  | int partial = divide + divide % 2; |
|  |  |
|  | #pragma omp task |
|  | { |
|  | quick(array, tmp, partial, part); |
|  | } |
|  | #pragma omp task |
|  | { |
|  | quick(array + partial, tmp + partial, size - partial, part); |
|  | } |
|  | #pragma omp taskwait |
|  |  |
|  | #pragma omp task |
|  | { |
|  | even(array, tmp, partial, size - partial); |
|  | } |
|  | #pragma omp task |
|  | { |
|  | odd(array, tmp, partial, size - partial); |
|  | } |
|  | #pragma omp taskwait |
|  |  |
|  | #pragma omp parallel |
|  | for (int i = 1; i < (size + 1) >> 1; i += 1) { |
|  | if (array[i << 1] < array[(i << 1) - 1]) { |
|  | std::swap(array[(i << 1) - 1], array[i << 1]); |
|  | } |
|  | } |
|  | } |
|  | } |
|  |  |
|  | void quickSort\_\_OMP(double\* array, int threads, int size) |
|  | { |
|  | double\* temporary = new double[size]; |
|  |  |
|  | int portion = size / threads; |
|  | if (size % threads) |
|  | portion += 1; |
|  |  |
|  |  |
|  | #pragma omp task |
|  | { |
|  | quick(array, temporary, size, portion); |
|  | } |
|  | } |
|  |  |
|  | void getRandomArray(double\* arr, int size) |
|  | { |
|  | int i = 0; |
|  | double number; |
|  |  |
|  | while(i < size) |
|  | { |
|  | number = rand() / (RAND\_MAX + 1.0); |
|  | arr[i] = number; |
|  | i += 1; |
|  | } |
|  | } |
|  |  |
|  | bool isSorted(double\* ar, int size) { |
|  |  |
|  | const double \*previous\_value = ar; |
|  |  |
|  | while (size) { |
|  | if (\*ar < \*previous\_value) |
|  | return false; |
|  | previous\_value = ar; |
|  |  |
|  | ++ar; |
|  | --size; |
|  | } |
|  | return true; |
|  | } |
|  |  |
|  |  |
|  | int main(void) { |
|  | srand(time(NULL)); |
|  |  |
|  | int size = 200; |
|  | int threads = 4; |
|  |  |
|  | double\* omp = new double[size]; |
|  | double\* seq = new double[size]; |
|  | getRandomArray(omp, size); |
|  |  |
|  | for (int i = 0; i < size; i++) |
|  | { |
|  | seq[i] = omp[i]; |
|  | } |
|  |  |
|  |  |
|  | double begin; |
|  | double finish; |
|  | begin = omp\_get\_wtime(); |
|  | quickSort\_\_OMP(omp, threads, size); |
|  | finish = omp\_get\_wtime(); |
|  | printf("(OMP) time for quicksort = %f seconds\n", finish - begin); |
|  |  |
|  |  |
|  | clock\_t start = clock(); |
|  | quickSort(seq, 0, size - 1); |
|  | clock\_t end = clock(); |
|  | float seconds = (float)(end - start) / CLOCKS\_PER\_SEC; |
|  | printf("(Sequential) time for quicksort = %f \n", seconds); |
|  |  |
|  | if ( isSorted(omp, size) ) |
|  | printf("Correctly sorted\n"); |
|  | else |
|  | printf("Incorretly sorted\n"); |
|  |  |
|  | if ( isSorted(seq, size) ) |
|  | printf("Correctly sorted\n"); |
|  | else |
|  | printf("Incorretly sorted\n"); |
|  |  |
|  |  |
|  | for (int i = 0; i < size; i++) |
|  | { |
|  | if (omp[i] != seq[i]) |
|  | { |
|  | puts("not equal"); break; |
|  | } |
|  | } |
|  |  |
|  | delete[]omp; |
|  | delete[]seq; |
|  |  |
|  | return 0; |
|  | } |

TBB:

|  |
| --- |
|  |
|  | #include <cstdlib>  #include <time.h> |
|  | #include <stdio.h> |
|  | #include <limits.h> |
|  | #include <tbb/tbb.h> |
|  |  |
|  | void quickSort(double\* array, int left, int right) |
|  | { |
|  | while (right > left) |
|  | { |
|  | int iterate\_left = left; |
|  | int iterate\_right = right; |
|  | double pivot = array[(left + right) >> 1]; |
|  |  |
|  | while (iterate\_left <= iterate\_right) |
|  | { |
|  | while (array[iterate\_left] < pivot) |
|  | { |
|  | iterate\_left += 1; |
|  | } |
|  | while (array[iterate\_right] > pivot) |
|  | { |
|  | iterate\_right -= 1; |
|  | } |
|  | if (iterate\_left <= iterate\_right) |
|  | { |
|  | double temp = array[iterate\_left]; |
|  | array[iterate\_left] = array[iterate\_right]; |
|  | array[iterate\_right] = temp; |
|  |  |
|  | iterate\_left += 1; |
|  | iterate\_right -= 1; |
|  | } |
|  | } |
|  |  |
|  | if ((iterate\_left << 1) > left + right) |
|  | { |
|  | quickSort(array, iterate\_left, right); |
|  | right = iterate\_left - 1; |
|  | } |
|  | else |
|  | { |
|  | quickSort(array, left, iterate\_left - 1); |
|  | left = iterate\_left; |
|  | } |
|  | } |
|  | } |
|  |  |
|  | void even(double\* array, double\* tmp, int left\_part, int right\_part) |
|  | { |
|  | int i = 0, j; |
|  | while(i < left\_part) |
|  | { |
|  | tmp[i] = array[i]; |
|  | i += (1 << 1); |
|  | } |
|  |  |
|  | double\* array2 = array + left\_part; |
|  | int a = 0, b = 0; |
|  | i = a; |
|  |  |
|  | for (i; (a < left\_part) && (b < right\_part); i += (1 << 1)) |
|  | { |
|  | array[i] = tmp[a]; |
|  |  |
|  | if (tmp[a] <= array2[b]) |
|  | { |
|  | a += (1 << 1); |
|  | } |
|  | else |
|  | { |
|  | array[i] = array2[b]; |
|  | b += (1 << 1); |
|  | } |
|  | } |
|  |  |
|  | j = b; |
|  | if (a == left\_part) |
|  | { |
|  | while(j < right\_part) |
|  | { |
|  | array[i] = array2[j]; |
|  | j += (1 << 1); |
|  | i += (1 << 1); |
|  | } |
|  | } |
|  | else |
|  | { |
|  | j = a; |
|  | while(j < left\_part) |
|  | { |
|  | array[i] = tmp[j]; |
|  | j += (1 << 1); |
|  | i += (1 << 1); |
|  | } |
|  | } |
|  | } |
|  |  |
|  | void odd(double\* array, double\* tmp, int left\_part, int right\_part) |
|  | { |
|  | int i = 1, j; |
|  | while(i < left\_part) |
|  | { |
|  | tmp[i] = array[i]; |
|  | i += (1 << 1); |
|  | } |
|  |  |
|  | double\* array2 = array + left\_part; |
|  | int a = 1, b = 1; |
|  | i = a; |
|  |  |
|  | for (i; (a < left\_part) && (b < right\_part); i += (1 << 1)) |
|  | { |
|  | array[i] = tmp[a]; |
|  |  |
|  | if (tmp[a] <= array2[b]) |
|  | { |
|  | a += (1 << 1); |
|  | } |
|  | else |
|  | { |
|  | array[i] = array2[b]; |
|  | b += (1 << 1); |
|  | } |
|  | } |
|  |  |
|  | j = b; |
|  | if (a == left\_part) |
|  | { |
|  | while(j < right\_part) |
|  | { |
|  | array[i] = array2[j]; |
|  | j += (1 << 1); |
|  | i += (1 << 1); |
|  | } |
|  | } |
|  | else |
|  | { |
|  | j = a; |
|  | while(j < left\_part) |
|  | { |
|  | array[i] = tmp[j]; |
|  | j += (1 << 1); |
|  | i += (1 << 1); |
|  | } |
|  | } |
|  | } |
|  |  |
|  | void quick(double\* array, double\* tmp, int size, int part) |
|  | { |
|  | if (size <= part) |
|  | { |
|  | quickSort(array, 0, size - 1); |
|  | } |
|  | else |
|  | { |
|  | int divide = size >> 1; |
|  | int partial = divide + divide % (1 << 1); |
|  |  |
|  | tbb::task\_group sort; |
|  | tbb::task\_group batcher; |
|  |  |
|  | sort.run([&]{quick(array, tmp, partial, part);}); |
|  | sort.run\_and\_wait([&]{quick(array + partial, tmp + partial, size - partial, part);}); |
|  |  |
|  | batcher.run([&]{even(array, tmp, partial, size - partial);}); |
|  | batcher.run\_and\_wait([&]{odd(array, tmp, partial, size - partial);}); |
|  |  |
|  | tbb::parallel\_for(tbb::blocked\_range<int>(1, (size + 1) >> 1),[&](const tbb::blocked\_range<int>& r) |
|  | { |
|  | int i = r.begin(); |
|  | while (i < r.end()) |
|  | { |
|  | if (array[i << 1] < array[(i << 1) - 1]) |
|  | { |
|  | std::swap(array[(i << 1) - 1], array[i << 1]); |
|  | } |
|  | i += 1; |
|  | } |
|  | }); |
|  | } |
|  | } |
|  | void quickSort\_\_TBB(double\* array, int threads, int size) |
|  | { |
|  | double temporary[size]; |
|  |  |
|  | int portion = size / threads; |
|  | if (size % threads) |
|  | portion += 1; |
|  |  |
|  | tbb::task\_group g; |
|  | g.run\_and\_wait([&]{quick(array, temporary, size, portion);}); |
|  | } |
|  |  |
|  | void getRandomArray(double\* arr, int size) |
|  | { |
|  | int i = 0; |
|  | double number; |
|  |  |
|  | while(i < size) |
|  | { |
|  | number = rand() / (RAND\_MAX + 1.0); |
|  | arr[i] = number; |
|  | i += 1; |
|  | } |
|  | } |
|  |  |
|  | void isSorted(double\* arr, int size) |
|  | { |
|  | int i = 0; |
|  |  |
|  | while(i < size - 1) |
|  | { |
|  | if (arr[i] > arr[i + 1]) { printf("\n Incorretly sorted %f > %f \n", arr[i], arr[i + 1]); return; } |
|  | i += 1; |
|  | } |
|  | printf("\nCorrectly sorted\n"); |
|  | } |
|  |  |
|  | int main(void) { |
|  | srand(time(NULL)); |
|  |  |
|  | int size = 200; |
|  | int threads = 4; |
|  |  |
|  | double\* tbb = new double[size]; |
|  | double\* seq = new double[size]; |
|  | getRandomArray(tbb, size); |
|  |  |
|  | for (int i = 0; i < size; i++) |
|  | { |
|  | seq[i] = tbb[i]; |
|  | } |
|  |  |
|  | tbb::tick\_count t0 = tbb::tick\_count::now(); |
|  | quickSort\_\_TBB(tbb, threads, size); |
|  | tbb::tick\_count t1 = tbb::tick\_count::now(); |
|  | printf("(TBB) time for quicksort = %g seconds\n", (t1 - t0).seconds() ); |
|  |  |
|  | clock\_t start = clock(); |
|  | quickSort(seq, 0, size - 1); |
|  | clock\_t end = clock(); |
|  | float seconds = (float)(end - start) / CLOCKS\_PER\_SEC; |
|  | printf("(Sequential) time for quicksort = %f \n", seconds); |
|  |  |
|  | isSorted(tbb, size); |
|  | isSorted(seq, size); |
|  |  |
|  | for (int i = 0; i < size; i++) |
|  | { |
|  | if (tbb[i] != seq[i]) |
|  | { |
|  | puts("not equal"); break; |
|  | } |
|  | } |
|  |  |
|  | delete[]tbb; |
|  | delete[]seq; |
|  |  |
|  | return 0; |
|  | } |