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**

Отчет по лабораторной работе

«Сортировка Хоара с четно-нечетным слиянием Бэтчера»

Выполнил:
Студентка группы 381708-2
Коврижных М. А.

Проверил:
Ассистент кафедры МОСТ,
прикладной математики и информатики
Волокитин В. Д.

Нижний Новгород
2020 г.

Оглавление

Введение

Сортировка Хаора (или быстрая сортировка quicksort) - алгоритм сортировки, разработанный английским информатиком Чарльзом Хоаром. Представляет собой один из самых быстрых и универсальных алгоритмов сортировки массивов: в среднем $O(n \log n)$ обменов при упорядочении n элементов.

Популярность данного алгоритма сортировки обусловленна его быстродействием в среднем случае по сравнению с его конкурентами (MergeSort), а также меньшим использованием дополнительной памяти. Тем не менее, некоторые минусы: большая ассимптотика худшего случая ($O(n^2)$), а также риск переполнения стека, заставляют использовать на практике гибриды данного алгоритма с другими алгоритмами сортировки (например: с HeapSort в Std).

Целью данной работы является реализация последовательной версии алгоритма сортировки Хоара, а также распараллеливание данного алгоритма с использованием четно-нечетного слияния Бетчера и разных технологий выполнения параллельных вычислений.

1. Постановка задачи

В рамках данных лабораторных работ ставится задача: реализовать алгоритм сортировки Хоара с четно-нечетным слиянием Бетчера с использованием различных технологий для выполнения параллельных вычислений.

Для решения данной задачи необходимо выполнить следующие пункты:

- изучить алгоритм сортировки Хоара и реализовать его последовательную версию;
- рассмотреть в чем заключается четно-нечетное слияние Бетчера и написать параллельные реализации алгоритма Хоара с использованием данного слияния с помощью OpenMP, TBB и потоков из стандартной библиотеки языка C++ (`std::thread`);
- выполнить сравнение и сделать выводы об эффективности написанных реализаций;
- проверить корректность работы алгоритма.

2. Описание алгоритма

Быстрая сортировка относится к алгоритмам "разделяй и властвуй". Сам алгоритм можно записать в 3 шага:

1. Выбрать элемент из массива. Назовём его стержнем. В качестве стержневого элемента можно брать любой элемент сортируемого массива. Как правило за стержневый элемент берут средний элемент, либо элемент, индекс которого выбрается генератором рандомных чисел.

2. Разбиение: перераспределение элементов в массиве таким образом, что элементы меньше стержневого встают перед ним, а больше или равные после. Для реализации данного шага можно использовать разные виды разбиений, мы будем использовать разбиение Хоара. Оно заключается в следующем: идем по исходному массиву двумя индексами (с начала и с конца), как только нашлась пара элементов, где один больше опорного и стоит перед ним, а второй меньше и стоит после, то меняем эти элементы местами. Разбиение Хоара осуществляется, пока индексы не пересекуться.

3. Рекурсивно применить первые два шага к двум подмассивам слева и справа от стержневого элемента. Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы.

Для небольшой оптимизации алгоритма по используемой стековой памяти будем использовать прием, который называется "хвостовая элиминация". Суть данного приема в устранении одной ветви рекурсии: вместо того, чтобы после разделения массива вызывать рекурсивно процедуру разделения для обоих найденных подмассивов, рекурсивный вызов делается только для меньшего подмассива, а больший обрабатывается в цикле в пределах этого же вызова процедуры. С точки зрения эффективности в среднем случае разницы практически нет: накладные расходы на дополнительный рекурсивный вызов и на организацию сравнения длин подмассивов и цикла — примерно одного порядка. Зато глубина рекурсии ни при каких обстоятельствах не превысит $\log_2 n$, в то время как в стандартном алгоритме глубина рекурсии достигает n. Применение этого метода не спасёт от катастрофического падения производительности в худшем для алгоритма случае, но переполнения стека не будет.

3. Описание схемы распараллеливания

Для распараллеливания алгоритма сортировки будем делить исходный массив на N частей. Каждая отдельная часть будет сортироваться в отдельном потоке. То есть потоков будет столько же, сколько и частей массива - N.

После того, как части массива отсортированы по отдельности, мы можем приступить к этапу слияния. Для данного этапа будем использовать четно-нечетное слияние Бетчера, которое заключается в том, что два упорядоченных массива, которые необходимо слить, разделяются на чётные и нечётные элементы. Такое слияние может быть выполнено параллельно. Каждые две отдельные части массива будут сливаться с использованием двух потоков: первый поток будет сравнивать четные элементы, второй - нечетные. 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.

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.

4. Описание программной реализации

В данном разделе приведены краткие описания схем параллельного выполнения для различных технологий параллельного программирования. Полный код можно найти в приложении.

4.1. Описание реализации с использованием OpenMP

Для распараллеливания алгоритма при помощи технологии OpenMP использовалась директивы `#pragma omp parallel`, `#pragma omp for`, `#pragma omp single`, `#pragma omp task`, `#pragma omp taskwait`.
Чтобы выполнить данный алгоритм параллельно с помощью OMP, будем использовать механизм логических задач, директиву `#pragma omp single`. Каждая логическая задача содержит метод `#pragma omp task` в котором будут выполнятся вычисления.

Для нашего алгоритма быстрой сортировки нам понадобится две задачи для слияния: even и odd - слияние четных и нечетных элементов. А также создадим общую задачу quick, которая будет делать рекурсивные вызовы, разбивая исходный массив до тех пор, пока размер отдельной части не будет меньше установленной границы. Мы используем `#pragma omp for` для того, чтобы слить в финальный отсортированный массив параллельно.

4.2. Описание реализации с использованием TBB

Чтобы выполнить данный алгоритм параллельно с помощью TBB, будем использовать механизм логических задач `tbb::task`. Каждая логическая задача содержит метод `task::execute` в котором будут выполнятся вычисления.

Для нашего алгоритма быстрой сортировки нам понадобится две задачи для слияния: even и odd - слияние четных и нечетных элементов. А также создадим общую задачу quick, которая будет делать рекурсивные вызовы, разбивая исходный массив до тех пор, пока размер отдельной части не будет меньше установленной границы (граница зависит от числа потоков). Наконец, чтобы осуществлять последний этап слияния Бетчера (пробег по слитым массивам и сравнение пар элементов) параллельно, нам понадобиться использовать механизм распараллеливания циклов от TBB: `tbb::parallel_for`. Данная функция принимает объект `tbb::blocked_range`, задающий количество итераций цикла, а также объект функтор. В качестве функтора мы реализуем анонимную функ-

цию, которая и будет выполнять финальный пробег по массиву с сравнением необходимых элементов.

Таким образом, мы получили следующий алгоритм:

1. Задача quick рекурсивно делит исходный массив на порции определенного размера, зависящего от числа потоков. Когда размер части становится меньше граничного значения, вызываем метод последоватьной сортироски Хоара для данной части массива и возвращаемся из рекурсии.

2. После того как вызов подзадач на упорядочивание частей массивов завершен, мы можем приступить к слиянию: создаем и выполняем задачи: even и odd.

3. Наконец, для окончательной сортировки слитых частей вызываем метод `tbb::parallel_for`, передав ему границы обрабатываемой части массива и анонимной лямбда функции. После того как над частями массива выполнены шаги 2 и 3, передаем управление вверх по рекурсии.

Для запуска выполнения задач и ожидания их завершения используются методы: `run` и `run_and_wait`

4.3. Описание реализации с использованием std::thread

Схема распараллеливания с использованием технологии `std::thread` будет аналогична технологии OMP и TBB. Для создания потока используем конструктор `std::thread`, в который будем передавать аргументы к функции. А для ожидания завершения потока используем метод `join` для каждого вызванного потока.

5. Результаты экспериментов

Конфигурация системы:

- Процессор: AMD A8-6410 @ 2.00GHz
- Оперативная память: 6 GB
- Операционная система: Windows 7
- Число ядер: 4

Результаты вычислительных экспериментов для массива с $size = 2 * 10^5$ представлены в таблице.

Таблица 1: Результаты вычислительных экспериментов

Кол-во потоков	Последовательный алгоритм	Параллельный алгоритм					
		OpenMP		TBB		std::thread	
	t, c	t, c	speedup	t, c	speedup	t, c	speedup
2	7.35	4,147	1.7	4,147	1.7	4,34375	1.6
4	7.35	3.378	2.25	3.356	2.27	3.52	2.12

По результатам проделанных экспериментов можно сделать следующие выводы:

1. На 2 потоках видно, что алгоритм достаточно неплохо параллелится и дает ускорение порядка 1.7 отсносительно последовательной версии
2. Реализации задачи с использованием TBB и std::thread практически одинаковая, но TBB показывает более высокий результат. Это говорит о том, что в библиотеке TBB лучшего организована работа планировщика потоков.
3. Лучшего результата удалось достичь с помощью библиотек OpenMP и TBB.

Заключение

В процессе выполнения данных лабораторных работ была реализована последовательная версия сортировки Хоара, рассмотрен принцип четно-нечетного слияния Бетчера, с помощью которого на основе различных технологий реализована параллельная версия сортировки Хоара.

В результате самыми эффективными реализациями оказались те, которые написаны с помощью технологий OpenMP и TBB.

Литература

1. Википедия: свободная электронная энциклопедия на английском языке: URL: https://en.wikipedia.org/wiki/Batcher_odd%E2%80%93even_mergesort

2. Guide into OpenMP: Easy multithreading programming for C++. URL: https://bisqwit.iki.fi/story/howto/openmp/

3. Intel Threading Building Blocks User Guide. URL: https://www.threadingbuildingblocks.org/docs/help/tbb_userguide/parallel_for.html

Приложение

Исходный код

Лабораторная работа №1. Последовательная версия

Листинг 1: Файл с реализацией

```
#include <cstdlib>
#include <time.h>
#include <stdio.h>
#include <limits.h>
#include <iostream>

template<typename T>
void swp(T& a, T& b)
{ T temp = a;
  a = b;
  b = temp;
}

int partition(double* arr, int low, int high)
{
    double pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swp(arr[i], arr[j]);
        }
    }
    swp(arr[i + 1], arr[high]);
    return (i + 1);
}

void qsrt(double* arr, int low, int high) {
        if (low < high)
    {
        int pi = partition(arr, low, high);

        qsrt(arr, low, pi - 1);
        qsrt(arr, pi + 1, high);
    }
}

void getRandomArray(double* arr, int size)
{
        int i = 0;
        double number;

        while(i < size)
        {
                number = rand() / (RAND_MAX + 1.0);
                arr[i] = number;
```

```
50                  i += 1;
51          }
52  }
53
54  bool isSorted(double* ar, int size) {
55      const double *previous_value = ar;
56
57      while (size) {
58         if (*ar < *previous_value)
59               return false;
60         previous_value = ar;
61
62         ++ar;
63         --size;
64       }
65       return true;
66  }
67
68  int main(void) {
69          srand(time(NULL));
70
71          int size = 200000;
72          int threads = 4;
73
74      double* seq = new double[size];
75      getRandomArray(seq, size);
76
77      clock_t start = clock();
78      qsrt(seq, 0, size - 1);
79      clock_t end = clock();
80          float seconds = (float)(end - start) / CLOCKS_PER_SEC;
81      printf("(Sequential) time for quicksort = %f \n", seconds);
82
83      if (!isSorted(seq, size)) {
84          printf("Sorted incorrectly\n");
85      } else {
86                  printf("Sorted correctly\n");
87          }
88
89          //print everything
90  /*
91          for (int i = 0; i < size; i++)
92          std::cout << seq[i] << " ";
93      std::cout << std::endl;*/
94
95      delete[]seq;
96
97          return 0;
98  }
```

Лабораторная работа №2. Параллельная версия с использованием OpenMP

Листинг 2: Файл с реализацией

```
#include <algorithm>
#include <cstdlib>
#include <time.h>
#include <stdio.h>
#include <omp.h>

template<typename T>
void swp(T& a, T& b)
{ T temp = a;
  a = b;
  b = temp;
}

void qsrt(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) {
            qsrt(array, iterate_left, right);
            right = iterate_left - 1;
        } else {
            qsrt(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 += 2;
    }

    double* array2 = array + left_part;
    int a = 0, b = 0;
```

```
56      i = a;
57
58      for (i; (a < left_part) && (b < right_part); i += 2) {
59          array[i] = tmp[a];
60
61          if (tmp[a] <= array2[b]) {
62              a += 2;
63          } else {
64              array[i] = array2[b];
65              b += 2;
66          }
67      }
68
69      j = b;
70      if (a == left_part) {
71          while(j < right_part) {
72                  array[i] = array2[j];
73                  j += 2;
74                  i += 2;
75          }
76      } else {
77          j = a;
78          while(j < left_part) {
79                  array[i] = tmp[j];
80                  j += 2;
81                  i += 2;
82          }
83      }
84  }
85
86  void odd(double* array, double* tmp, int left_part, int right_part) {
87          int i = 1, j;
88      while(i < left_part) {
89          tmp[i] = array[i];
90          i += (1 << 1);
91      }
92
93      double* array2 = array + left_part;
94      int a = 1, b = 1;
95      i = a;
96
97      for (i; (a < left_part) && (b < right_part); i += (1 << 1)) {
98          array[i] = tmp[a];
99
100         if (tmp[a] <= array2[b]) {
101             a += 2;
102         } else {
103             array[i] = array2[b];
104             b += 2;
105         }
106     }
107
108     j = b;
109     if (a == left_part) {
110         while(j < right_part) {
111                 array[i] = array2[j];
112                 j += 2;
113                 i += 2;
114         }
115     } else {
```

```
116         j = a;
117         while(j < left_part) {
118                 array[i] = tmp[j];
119                 j += 2;
120                 i += 2;
121         }
122     }
123 }
124
125 void quick(double* array, double* tmp, int size, int part) {
126
127         #pragma omp parallel
128     #pragma omp single
129     {
130                 if (size <= part) {
131                 qsrt(array, 0, size - 1);
132         } else {
133                 int divide = size >> 1;
134                 int partial = divide + divide % 2;
135                 #pragma omp task
136                 {
137                         quick(array, tmp, partial, part);
138                 }
139
140                 #pragma omp task
141                 {
142                         quick(array + partial, tmp + partial, size - partial, part)
                            ;
143                 }
144                 #pragma omp taskwait
145
146                 #pragma omp task
147                 {
148                         even(array, tmp, partial, size - partial);
149                 }
150
151                 #pragma omp task
152                 {
153                 odd(array, tmp, partial, size - partial);
154                 }
155                 #pragma omp taskwait
156
157                         #pragma omp parallel num_threads(4)
158                         {
159                                 #pragma omp for
160                                 for (int i = 1; i < (size + 1) >> 1; i += 1) {
161                                         if (array[i << 1] < array[(i << 1) - 1]) {
162                                         swp(array[(i << 1) - 1], array[i << 1]);
163                                 }
164                                 }
165                         }
166         }
167     }
168 }
169
170 void quickSort__OMP(double* array, int threads, int size) {
171     double* temporary = new double[size];
172
173     int portion = size / threads;
174     if (size % threads)
```

```
175         portion += 1;
176
177         #pragma omp parallel
178         #pragma omp single
179         {
180         #pragma omp task
181         {
182                 quick(array, temporary, size, portion);
183         }
184     }
185     delete[]temporary;
186 }
187
188 void getRandomArray(double* arr, int size) {
189         int i = 0;
190         double number;
191
192         while(i < size)
193         {
194                 number = rand() / (RAND_MAX + 1.0);
195                 arr[i] = number;
196                 i += 1;
197         }
198 }
199
200 bool isSorted(double* ar, int size) {
201     const double *previous_value = ar;
202
203     while (size) {
204        if (*ar < *previous_value)
205              return false;
206        previous_value = ar;
207
208        ++ar;
209        --size;
210      }
211      return true;
212 }
213
214
215 int main(void) {
216         srand(time(NULL));
217
218         int size = 200000;
219         int threads = 4;
220
221     double* omp = new double[size];
222     double* seq = new double[size];
223     getRandomArray(omp, size);
224
225     for (int i = 0; i < size; i++)
226         {
227                 seq[i] = omp[i];
228         }
229
230     double begin;
231     double finish;
232     begin = omp_get_wtime();
233     quickSort__OMP(omp, threads, size);
234     finish = omp_get_wtime();
```

```
235     printf("(OMP) time for quicksort = %f seconds\n", finish - begin);
236 
237     clock_t start = clock();
238     qsrt(seq, 0, size - 1);
239     clock_t end = clock();
240         float seconds = (float)(end - start) / CLOCKS_PER_SEC;
241     printf("(Sequential) time for quicksort = %f \n", seconds);
242 
243     if ( isSorted(omp, size) )
244         printf("Correctly sorted\n");
245     else
246         printf("Incorretly sorted\n");
247 
248     if ( isSorted(seq, size) )
249         printf("Correctly sorted\n");
250     else
251         printf("Incorretly sorted\n");
252 
253 
254     for (int i = 0; i < size; i++)
255         {
256         if (omp[i] != seq[i])
257                 {
258                 puts("not equal"); break;
259         }
260     }
261 
262     delete[]omp;
263     delete[]seq;
264 
265         return 0;
266 }
```

Лабораторная работа №3. Параллельная версия с использованием TBB

Листинг 3: Файл с реализацией

```
#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);
```

```
56          }
57
58          double* array2 = array + left_part;
59          int a = 0, b = 0;
60          i = a;
61
62          for (i; (a < left_part) && (b < right_part); i += (1 << 1))
63                  {
64                  array[i] = tmp[a];
65
66                  if (tmp[a] <= array2[b])
67                          {
68                  a += (1 << 1);
69                  }
70                          else
71                          {
72                  array[i] = array2[b];
73                  b += (1 << 1);
74                  }
75          }
76
77          j = b;
78          if (a == left_part)
79                  {
80                  while(j < right_part)
81                          {
82                          array[i] = array2[j];
83                          j += (1 << 1);
84                          i += (1 << 1);
85                  }
86          }
87                  else
88                  {
89                  j = a;
90                  while(j < left_part)
91                          {
92                          array[i] = tmp[j];
93                          j += (1 << 1);
94                          i += (1 << 1);
95                  }
96          }
97  }
98
99  void odd(double* array, double* tmp, int left_part, int right_part)
100 {
101         int i = 1, j;
102         while(i < left_part)
103                 {
104                 tmp[i] = array[i];
105                 i += (1 << 1);
106         }
107
108         double* array2 = array + left_part;
109         int a = 1, b = 1;
110         i = a;
111
112         for (i; (a < left_part) && (b < right_part); i += (1 << 1))
113                 {
114                 array[i] = tmp[a];
115
```

```
116                 if (tmp[a] <= array2[b])
117                         {
118                 a += (1 << 1);
119                 }
120                         else
121                         {
122                 array[i] = array2[b];
123                 b += (1 << 1);
124                 }
125         }
126
127         j = b;
128         if (a == left_part)
129                 {
130                 while(j < right_part)
131                         {
132                         array[i] = array2[j];
133                         j += (1 << 1);
134                         i += (1 << 1);
135                 }
136         }
137                 else
138                 {
139                 j = a;
140                 while(j < left_part)
141                         {
142                         array[i] = tmp[j];
143                         j += (1 << 1);
144                         i += (1 << 1);
145                 }
146         }
147 }
148
149 void quick(double* array, double* tmp, int size, int part)
150 {
151         if (size <= part)
152         {
153                 quickSort(array, 0, size - 1);
154     }
155         else
156         {
157                 int divide = size >> 1;
158                 int partial = divide + divide % 2;
159
160                         tbb::task_group sort;
161                         tbb::task_group batcher;
162
163                         sort.run([&]{quick(array, tmp, partial, part);});
164                         sort.run_and_wait([&]{quick(array + partial, tmp + partial,
                                size - partial, part);});
165
166                         batcher.run([&]{even(array, tmp, partial, size - partial)
                            ;});
167                         batcher.run_and_wait([&]{odd(array, tmp, partial, size -
                            partial);});
168
169                 tbb::parallel_for(tbb::blocked_range<int>(1, (size + 1) >> 1),[&](
                    const tbb::blocked_range<int>& r)
170                         {
171                         int i = r.begin();
```

```
172                         while (i < r.end())
173                                 {
174                                 if (array[i << 1] < array[(i << 1) - 1])
175                                         {
176                                         std::swap(array[(i << 1) - 1], array[i <<
                                                1]);
177                                         }
178                                         i += 1;
179                         }
180             });
181         }
182 }
183 void quickSort__TBB(double* array, int threads, int size)
184 {
185     double temporary[size];
186
187     int portion = size / threads;
188     if (size % threads)
189         portion += 1;
190
191     tbb::task_group g;
192     g.run_and_wait([&]{quick(array, temporary, size, portion);});
193 }
194
195 void getRandomArray(double* arr, int size)
196 {
197         int i = 0;
198         double number;
199
200         while(i < size)
201         {
202                 number = rand() / (RAND_MAX + 1.0);
203                 arr[i] = number;
204                 i += 1;
205         }
206 }
207
208 void isSorted(double* arr, int size)
209 {
210         int i = 0;
211
212         while(i < size - 1)
213         {
214                 if (arr[i] > arr[i + 1]) { printf("\n Incorretly sorted %f > %f \n"
                        , arr[i], arr[i + 1]); return; }
215                 i += 1;
216         }
217         printf("\nCorrectly sorted\n");
218 }
219
220 int main(void) {
221         srand(time(NULL));
222
223         int size = 200000;
224         int threads = 4;
225
226     double* tbb = new double[size];
227     double* seq = new double[size];
228     getRandomArray(tbb, size);
229
```

```
230     for (int i = 0; i < size; i++)
231         {
232                 seq[i] = tbb[i];
233         }
234 
235     tbb::tick_count t0 = tbb::tick_count::now();
236         quickSort__TBB(tbb, threads, size);
237     tbb::tick_count t1 = tbb::tick_count::now();
238     printf("(TBB) time for quicksort = %g seconds\n", (t1 - t0).seconds() );
239 
240     clock_t start = clock();
241     quickSort(seq, 0, size - 1);
242     clock_t end = clock();
243         float seconds = (float)(end - start) / CLOCKS_PER_SEC;
244     printf("(Sequential) time for quicksort = %f \n", seconds);
245 
246     isSorted(tbb, size);
247     isSorted(seq, size);
248 
249     for (int i = 0; i < size; i++)
250         {
251         if (tbb[i] != seq[i])
252                 {
253                 puts("not equal"); break;
254         }
255     }
256 
257     delete[]tbb;
258     delete[]seq;
259 
260         return 0;
261 }
```

Лабораторная работа №4. Параллельная версия с использованием std::thread

Листинг 4: Файл с реализацией

```
#include <cstdlib>
#include <time.h>
#include <stdio.h>
#include <limits.h>
#include <thread>

template<typename T>
void swp(T& a, T& b)
{ T temp = a;
  a = b;
  b = temp;
}

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;
        }
    }
}

```

```
56 void even(double* array, double* tmp, int left_part, int right_part)
57 {
58         int i = 0, j;
59         while(i < left_part)
60                 {
61                 tmp[i] = array[i];
62                 i += (1 << 1);
63         }
64
65         double* array2 = array + left_part;
66         int a = 0, b = 0;
67         i = a;
68
69         for (i; (a < left_part) && (b < right_part); i += 2)
70                 {
71                 array[i] = tmp[a];
72
73                 if (tmp[a] <= array2[b])
74                         {
75                 a += 2;
76                 }
77                         else
78                         {
79                 array[i] = array2[b];
80                 b += 2;
81                 }
82         }
83
84         j = b;
85         if (a == left_part)
86                 {
87                 while(j < right_part)
88                         {
89                         array[i] = array2[j];
90                         j += 2;
91                         i += 2;
92                 }
93         }
94                 else
95                 {
96                 j = a;
97                 while(j < left_part)
98                         {
99                         array[i] = tmp[j];
100                        j += 2;
101                        i += 2;
102                }
103        }
104 }
105
106 void odd(double* array, double* tmp, int left_part, int right_part)
107 {
108        int i = 1, j;
109        while(i < left_part)
110                {
111                tmp[i] = array[i];
112                i += 2;
113        }
114
115        double* array2 = array + left_part;
```

```
116         int a = 1, b = 1;
117         i = a;
118
119         for (i; (a < left_part) && (b < right_part); i += 2)
120                 {
121                 array[i] = tmp[a];
122
123                 if (tmp[a] <= array2[b])
124                         {
125                 a += 2;
126                 }
127                         else
128                         {
129                 array[i] = array2[b];
130                 b += 2;
131                 }
132         }
133
134         j = b;
135         if (a == left_part)
136                 {
137                 while(j < right_part)
138                         {
139                         array[i] = array2[j];
140                         j += 2;
141                         i += 2;
142                 }
143         }
144                 else
145                 {
146                 j = a;
147                 while(j < left_part)
148                         {
149                         array[i] = tmp[j];
150                         j += 2;
151                         i += 2;
152                 }
153         }
154 }
155
156 void quick(double* array, double* tmp, int size, int part) {
157         if (size <= part) {
158                 quickSort(array, 0, size - 1);
159     } else {
160         int divide = size >> 1;
161         int partial = divide + divide % 2;
162
163                 std::thread sortL(quick, array, tmp, partial, part);
164                 std::thread sortR(quick, array + partial, tmp + partial, size -
                        partial, part);
165                 sortL.join();
166                 sortR.join();
167
168                 std::thread batcherE(even, array, tmp, partial, size - partial);
169                 std::thread batcherO(odd, array, tmp, partial, size - partial);
170                 batcherE.join();
171                 batcherO.join();
172
173                 auto lambda = [&]() {
174                         int i = 1;
```

```
175             while (i < (size + 1) >> 1) {
176                 if (array[i << 1] < array[(i << 1) - 1]) {
177                         swp(array[(i << 1) - 1], array[i << 1]);
178                                 }
179                         i += 1;
180             }
181                 };
182                 std::thread prll(lambda);
183                 prll.join();
184     }
185 }
186 void quickSort__STD(double* array, int threads, int size)
187 {
188     double* temporary = new double[size];
189 
190     int portion = size / threads;
191     if (size % threads)
192         portion += 1;
193 
194     std::thread start(quick, array, temporary, size, portion);
195     start.join();
196     delete[]temporary;
197 }
198 
199 void getRandomArray(double* arr, int size)
200 {
201         int i = 0;
202         double number;
203 
204         while(i < size) {
205                 number = rand() / (RAND_MAX + 1.0);
206                 arr[i] = number;
207                 i += 1;
208         }
209 }
210 
211 bool isSorted(double* ar, int size) {
212     const double *previous_value = ar;
213 
214     while (size) {
215        if (*ar < *previous_value)
216             return false;
217        previous_value = ar;
218 
219        ++ar;
220        --size;
221      }
222      return true;
223 }
224 
225 int main(void) {
226         srand(time(NULL));
227 
228         int size = 20000;
229         int threads = 4;
230 
231     double* std = new double[size];
232     double* seq = new double[size];
233     getRandomArray(std, size);
234 
```

```
235     for (int i = 0; i < size; i++) {
236                 seq[i] = std[i];
237         }
238 
239         clock_t start, end;
240         float seconds;
241 
242     start = clock();
243         quickSort__STD(std, threads, size);
244     end = clock();
245     seconds = (float)(end - start) / CLOCKS_PER_SEC;
246     printf("(STD) time for quicksort = %f \n", seconds);
247 
248     start = clock();
249     quickSort(seq, 0, size - 1);
250     end = clock();
251         seconds = (float)(end - start) / CLOCKS_PER_SEC;
252     printf("(Sequential) time for quicksort = %f \n", seconds);
253 
254     if (isSorted(std, size) )
255         printf("Correctly sorted\n");
256     else
257         printf("Incorretly sorted\n");
258 
259     if (isSorted(seq, size) )
260         printf("Correctly sorted\n");
261     else
262         printf("Incorretly sorted\n");
263 
264     for (int i = 0; i < size; i++) {
265         if (std[i] != seq[i]) {
266                 puts("not equal"); break;
267         }
268     }
269 
270     delete[] std;
271     delete[] seq;
272 
273         return 0;
274 }
```