МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

**«Нижегородский государственный университет им. Н.И. Лобачевского»**

**Национальный исследовательский университет**

**Институт информационных технологий математики и механики**

**ОТЧЕТ К ЛАБОРАТОРНОЙ РАБОТЕ ПО ПАРАЛЛЕЛЬНОМУ ПРОГРАММИРОВАНИЮ**

**«Поразрядная сортировка для целых чисел с простым слиянием»**

**Выполнил**: студент группы 381506-3

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Грибов П. Н.

Подпись

**Проверил:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Сысоев А. В.

Подпись

Нижний Новгород

2018

**Содержание**

[**Введение** 3](#_Toc514715367)

[**Постановка задачи** 4](#_Toc514715368)

[**Описание алгоритма сортировки** 5](#_Toc514715369)

[**Псевдокод** 5](#_Toc514715370)

[**Сложность LSD-сортировки** 6](#_Toc514715371)

[**Схема распараллеливания OpenMP версии алгоритма** 7](#_Toc514715372)

[**Схема распараллеливания TBB версии алгоритма** 7](#_Toc514715373)

[**Реализация генератора** 8](#_Toc514715374)

[**Реализация checker’a** 8](#_Toc514715375)

[**Подтверждение корректности** 8](#_Toc514715376)

[**Результаты экспериментов по оценке масштабируемости.** 9](#_Toc514715377)

[**Список литературы** 11](#_Toc514715378)

[**Приложение** 12](#_Toc514715379)

[Линейный алгоритм 12](#_Toc514715380)

[OpenMP алгоритм 13](#_Toc514715381)

[TBB алгоритм 15](#_Toc514715382)

**Введение**

В данной лабораторной работе рассматривается алгоритм поразрядной сортировки целых чисел с простым слиянием, работающий за время O(n).

Практика показывает, что указанный алгоритм не слишком сложен в программной реализации и хорошо себя показывает при упорядочивании огромных объемов данных. Одна из главных проблем – распараллеливание. Зачастую получается, что лучший последовательный алгоритм имеет сугубо последовательную природу, в связи с чем приходится либо брать менее эффективный, но распараллеливаемый алгоритм, либо делить массив на блоки с целью их сортировки и последующего слияния.

# **Постановка задачи**

Необходимо:

1. Придумать и запрограммировать простейшее решение задачи без использования распараллеливания.
2. Запрограммировать решение задачи на языке C++ с использованием OpenMP и TBB.
3. Написать программу-генератор тестов.
4. Написать программу checker, которая по входным данным и известному правильному ответу определяет корректность полученного результата.

# **Описание алгоритма сортировки**

В работе использован LSD вариант сортировки.

Имеем множество последовательностей одинаковой длины, состоящих из элементов, на которых задано отношение линейного порядка. Требуется отсортировать эти последовательности в лексикографическом порядке.

По аналогии с разрядами чисел будем называть элементы, из которых состоят сортируемые объекты, разрядами. Сам алгоритм состоит в последовательной сортировке объектов какой-либо устойчивой сортировкой по каждому разряду, в порядке от младшего разряда к старшему, после чего последовательности будут расположены в требуемом порядке.

Для чисел уже существует понятие разряда, поэтому будем представлять числа как последовательности разрядов. Конечно, в разных системах счисления разряды одного и того же числа отличаются, поэтому перед сортировкой представим числа в удобной для нас системе счисления.

В качестве устойчивой сортировки применим сортировку подсчетом.

## **Псевдокод**

На вход подается массив A размера n m-разрядных чисел. Сам по себе алгоритм представляет собой цикл по номеру разряда, на каждой итерации которого элементы массива A размещаются в нужном порядке во вспомогательном массиве B. Для подсчета количества объектов, i-й разряд которых одинаковый, а затем и для определения положения объектов в массиве B используется вспомогательный массив C. Функция digit(x,i) возвращает i-й разряд числа x. Также считаем, что значения разрядов меньше k.

|  |
| --- |
| **function** radixSort(int[] A):  **for** i = 1 **to** m  **for** j = 0 **to** k - 1  C[j] = 0  **for** j = 0 **to** n - 1  d = digit(A[j], i)  C[d]++  count = 0  **for** j = 0 **to** k - 1  tmp = C[j]  C[j] = count  count += tmp  **for** j = 0 **to** n - 1  d = digit(A[j], i)  B[C[d]] = A[j]  C[d]++  A = B |

## **Сложность LSD-сортировки**

Пусть m — количество разрядов, n — количество объектов, которые нужно отсортировать, T(n)— время работы устойчивой сортировки. Поразрядная сортировка выполняет k итераций, на каждой из которой выполняется устойчивая сортировка и не более O(1) других операций. Следовательно, время работы поразрядной сортировки — O(kT(n))

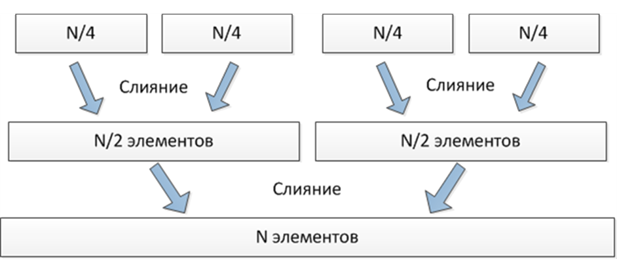
Рассмотрим отдельно случай сортировки чисел. Пусть в качестве аргумента сортировке передается массив, в котором содержатся n m-значных чисел, и каждая цифра может принимать значения от 0 до k−1. Тогда цифровая сортировка позволяет отсортировать данный массив за время  O(m(n+k)), если устойчивая сортировка имеет время работы O(n+k). Если k небольшое, то оптимально выбирать в качестве устойчивой сортировки сортировку подсчетом.

Если количество разрядов — константа, а k=O(n), то сложность поразрядной сортировки составляет O(n), то есть она линейно зависит от количества сортируемых чисел.

# **Схема распараллеливания OpenMP версии алгоритма**

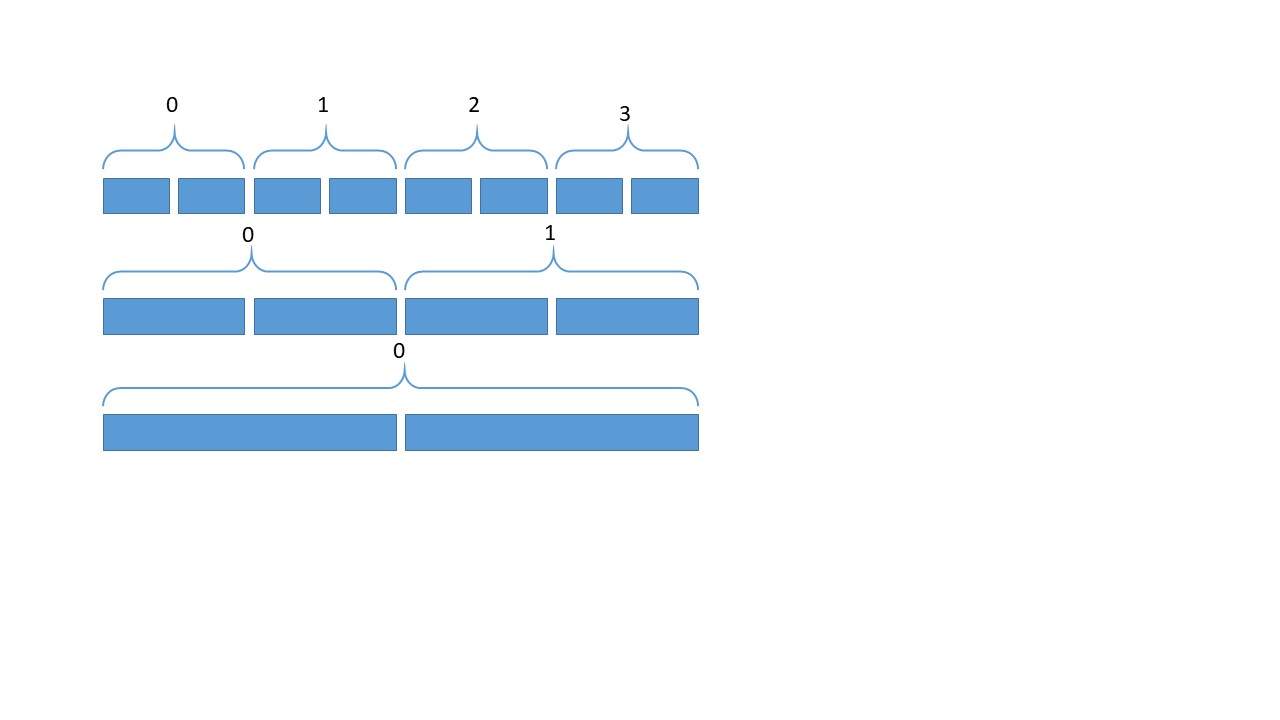
Исходный массив делится поровну между всеми потоками, затем каждый поток сортирует свою часть. Далее используется простое слияния:

Для каждого потока в функцию слияния передаются индексы начала и конца пары соседних отсортированных частей массива (расчет индексов производится по номеру потока), и каждый поток сливает свою пару. В этом случае слияние n блоков могут выполнять n/2 параллельных потоков. На следующем шаге слияние n/2 полученных блоков будут выполнять n/4 потоков и т.д.



**Схема распараллеливания TBB версии алгоритма**

Схема распараллеливания для TBB версии алгоритма в целом аналогична OpenMP версии, с той лишь разницей, что вводится вспомогательный вектор, хранящий индексы начала и конца для каждой отсортированной части массива, так как размер блока вычислений для каждого потока может быть различным при использовании стратегий планирования auto\_partitioner и simple\_partitioner. Затем для каждого потока в функцию слияния передается номер пары для слияния и шаг-размер очередного слияния (в отличие от openmp версии).



# **Реализация генератора**

Для тестирования алгоритма поразрядной сортировки целых чисел необходимо генерировать массивы случайных чисел различного размера. На вход программе-генератору Generator.exe через аргумент командной строки подается номер теста (Generator.exe [номер теста]). По индексу [номер теста – 1] берется размер для очередного генерируемого массива из вспомогательного массива размеров для тестов. Всего доступно 25 размеров теста:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| № | 1 | | 2 | 3 | 4 | 5 | 6 | 7 | | 8 | 9 | | 10 | | 11 | | 12 | 13 | |
| размер | 1 | | 2 | 2 | 5 | 5 | 10 | 10 | | 100 | 100 | | 500 | | 1000 | | 2000 | 5000 | |
| № | 14 | | | 15 | | 16 | | 17 | | | 18 | | | 19 | | | 20 | | 21 | |
| размер | 10000 | | | 25000 | | 50000 | | 100000 | | | 200000 | | | 500000 | | | 1000000 | | 5000000 | |
| № | | 22 | | | 23 | | | | 24 | | | 25 | | | |
| размер | | 5000000 | | | 10000000 | | | | 50000000 | | | 100000000 | | | |

На выходе Generator.exe генерирует бинарный файл, в котором сначала пишется время типа double (фиктивное при генерации; сделано для унификации при проверке отсортированного результата Viewer’ом), затем размер массива типа int, затем сам генерируемый случайный массив. Файл без расширения с именем по номеру теста.

# **Реализация checker’a**

На вход программе Checker.exe через аргументы командной строки подается номер теста, и программа берет два файла <номер\_теста>.ans и <номер\_теста>\_true.ans из папки ./tests/ . Бинарный файл <номер\_теста>.ans содержит отсортированный параллельным алгоритмом (программой Solution.exe) массив, а бинарный файл <номер\_теста>\_true.ans содержит массив, отсортированный доверительной сортировкой std::sort (программой TrueSolution.exe). Checker делает обход почисловой обход по массивам из этих файлов и сравнивает пары соответствующих чисел. Если какая-либо пара чисел не совпадает, то тест с текущим номером не проходит проверку на корректность, иначе тест корректен. Результат для каждого теста приписывается в конец файла ./tests/result.txt

# **Подтверждение корректности**

В ходе тестирования, за несколько циклов проверок по различным группам из 25 тестов, ни один тест не падает, что подтверждает корректность алгоритма.

# **Результаты экспериментов по оценке масштабируемости.**

Для наглядности в рассмотрение взяты тесты с номерами 14 – 25, т.к тесты меньшего размера не отражают реальное ускорение.

**2 потока:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер теста | Время работы алгоритма (сек.) | | | Ускорение | |
| Линейный | OpenMP | TBB | OpenMP | TBB |
| 10000 | 0.001648 | 0.001737 | 0.002693 | 0.94876 | 0.61196 |
| 25000 | 0.004354 | 0.002488 | 0.004848 | 1.75 | 0.8991 |
| 50000 | 0.008439 | 0.005545 | 0.006244 | 1.52191 | 1.35154 |
| 100000 | 0.019170 | 0.010014 | 0.015368 | 1.91432 | 1.2474 |
| 200000 | 0.038115 | 0.024096 | 0.022787 | 1.5818 | 1.67226 |
| 500000 | 0.088516 | 0.052376 | 0.054936 | 1.69001 | 1.61126 |
| 1000000 | 0.179893 | 0.103553 | 0.111347 | 1.73721 | 1.61561 |
| 5000000 | 0.349760 | 0.201615 | 0.222601 | 1.73479 | 1.57124 |
| 5000000 | 0.868701 | 0.496734 | 0.520690 | 1.74883 | 1.66837 |
| 10000000 | 1.721182 | 1.003419 | 1.030678 | 1.71532 | 1.66995 |
| 50000000 | 8.543601 | 5.318440 | 5.360443 | 1.60641 | 1.59382 |
| 100000000 | 17.083335 | 9.780027 | 10.418956 | 1.74676 | 1.63964 |

**4 потока:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер теста | Время работы алгоритма (сек.) | | | Ускорение | |
| Линейный | OpenMP | TBB | OpenMP | TBB |
| 10000 | 0.001648 | 0.001386 | 0.002498 | 1.18903 | 0.65973 |
| 25000 | 0.004354 | 0.001624 | 0.004640 | 2.68103 | 0.93836 |
| 50000 | 0.008439 | 0.006936 | 0.005505 | 1.2167 | 1.53297 |
| 100000 | 0.019170 | 0.013165 | 0.010455 | 1.45613 | 1.83357 |
| 200000 | 0.038115 | 0.025612 | 0.031932 | 1.48817 | 1.19363 |
| 500000 | 0.088516 | 0.070489 | 0.067064 | 1.25574 | 1.31987 |
| 1000000 | 0.179893 | 0.088452 | 0.122859 | 2.03379 | 1.46422 |
| 5000000 | 0.349760 | 0.211739 | 0.235237 | 1.65184 | 1.48684 |
| 5000000 | 0.868701 | 0.436433 | 0.366875 | 1.99046 | 2.36784 |
| 10000000 | 1.721182 | 0.666710 | 1.015959 | 2.58161 | 1.69415 |
| 50000000 | 8.543601 | 5.198945 | 4.878450 | 1.64333 | 1.75129 |
| 100000000 | 17.083335 | 7.497081 | 6.936800 | 2.27866 | 2.46271 |

**8 потоков:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер теста | Время работы алгоритма (сек.) | | | Ускорение | |
| Линейный | OpenMP | TBB | OpenMP | TBB |
| 10000 | 0.001648 | 0.001022 | 0.002437 | 1.61252 | 0.67624 |
| 25000 | 0.004354 | 0.002632 | 0.002878 | 1.65426 | 1.51286 |
| 50000 | 0.008439 | 0.006873 | 0.005028 | 1.22785 | 1.6784 |
| 100000 | 0.019170 | 0.008942 | 0.009263 | 2.14382 | 2.06952 |
| 200000 | 0.038115 | 0.021343 | 0.017012 | 1.78583 | 2.24048 |
| 500000 | 0.088516 | 0.042257 | 0.040800 | 2.09471 | 2.16951 |
| 1000000 | 0.179893 | 0.081637 | 0.077452 | 2.20357 | 2.32264 |
| 5000000 | 0.349760 | 0.159980 | 0.153457 | 2.18627 | 2.27921 |
| 5000000 | 0.868701 | 0.431986 | 0.320360 | 2.01095 | 2.71164 |
| 10000000 | 1.721182 | 0.765048 | 0.671339 | 2.24977 | 2.5638 |
| 50000000 | 8.543601 | 3.689839 | 3.417000 | 2.31544 | 2.50032 |
| 100000000 | 17.083335 | 5.499068 | 5.334953 | 3.10659 | 3.20215 |

# **Список литературы**

1. <http://neerc.ifmo.ru/wiki/index.php?title=Цифровая_сортировка>
2. [www.hpcc.unn.ru/file.php?id=724](http://www.hpcc.unn.ru/file.php?id=724)
3. <http://redmine.software.unn.ru/attachments/12125/Подготовка_задач_для_автоматизированной_проверки.pdf>
4. <http://redmine.software.unn.ru/attachments/12126/Стратегии_слияния_в_параллельных_сортировках.docx>

# **Приложение**

## Линейный алгоритм

|  |
| --- |
| // Функция получения требуемого разряда числа  int get\_dig(int num, int dig\_num)  {  num = abs(num);  for (int i = 0; i < (dig\_num - 1); ++i)  {  num /= 10;  }  return num % 10;  }  // Поразрядная сортировка (Алгоритм №2)  void radix\_sort(vector<int>& vec)  {  // максимально возможное число разрядов в числе  int max\_dig\_count = 10;  // максимально возможное значение разряда числа (0..9)  int max\_dig = 10;  // количество чисел  int num\_of\_numbers = static\_cast<int>(vec.size());  // вспомогательные переменные и векторы  int count, tmp;  vector<int> buffer(num\_of\_numbers);  vector<int> pos\_vec(10);  for (int i = 0; i < max\_dig\_count; ++i)  {  for (int j = 0; j < max\_dig; ++j)  pos\_vec[j] = 0;  for (int j = 0; j < num\_of\_numbers; ++j)  {  ++pos\_vec[get\_dig(vec[j], i)];  }  count = 0;  for (int j = 0; j < max\_dig; ++j)  {  tmp = pos\_vec[j];  pos\_vec[j] = count;  count += tmp;  }  for (int j = 0; j < num\_of\_numbers; ++j)  {  buffer[pos\_vec[get\_dig(vec[j], i)]++] = vec[j];  }  vec = buffer;  }  vec.clear();  std::copy\_if(buffer.rbegin(), buffer.rend(), std::back\_inserter(vec), [](int &i) { return i < 0; });  std::copy\_if(buffer.begin(), buffer.end(), std::back\_inserter(vec), [](int &i) { return i >= 0; });  } |

## OpenMP алгоритм

|  |
| --- |
| // Поразрядная сортировка части массива  void radix\_sort\_by(vector<int>& vec, int begin, int end, vector<int>& buffer)  {  if (end - begin == 1) return;  // максимально возможное число разрядов в числе  int max\_dig\_count = 10;  // максимально возможное значение разряда числа (0..9)  int max\_dig = 10;  // количество чисел  int num\_of\_numbers = static\_cast<int>(vec.size());  // вспомогательные переменные и векторы  int count, tmp;  //vector<int> buffer(num\_of\_numbers);  vector<int> pos\_vec(10);  for (int i = 0; i < max\_dig\_count; ++i)  {  for (int j = 0; j < max\_dig; ++j)  pos\_vec[j] = 0;  for (int j = begin; j < end; ++j)  {  ++pos\_vec[get\_dig(vec[j], i)];  }  count = 0;  for (int j = 0; j < max\_dig; ++j)  {  tmp = pos\_vec[j];  pos\_vec[j] = count;  count += tmp;  }  for (int j = begin; j < end; ++j)  {  buffer[begin + pos\_vec[get\_dig(vec[j], i)]++] = vec[j];  }  for (int j = begin; j < end; ++j)  {  vec[j] = buffer[j];  }  }  int nneg = 0;  for (int i = end - 1, j = begin; i >= begin; --i)  {  if (buffer[i] < 0)  {  ++nneg;  vec[j++] = buffer[i];  }  }  for (int i = begin, j = begin + nneg; i < end; ++i)  {  if (buffer[i] >= 0)  {  vec[j++] = buffer[i];  }  }  }  void vec\_merge(vector<int>& vec, const int& begin1, const int& end1,  const int& begin2, const int& end2, vector<int>& buffer)  {  vector<int>::iterator start1 = vec.begin() + begin1;  vector<int>::iterator finish1 = vec.begin() + end1;  vector<int>::iterator start2 = vec.begin() + begin2;  vector<int>::iterator finish2 = vec.begin() + end2;  vector<int>::iterator buffer\_start = buffer.begin() + begin1;  int count = end2 - begin1;  while (start1 != finish1 && start2 != finish2)  \*buffer\_start++ = \*start1 <= \*start2 ? \*start1++ : \*start2++;  while (start1 != finish1)  \*buffer\_start++ = \*start1++;  while (start2 != finish2)  \*buffer\_start++ = \*start2++;  buffer\_start = buffer.begin() + begin1;  start1 = vec.begin() + begin1;  for (int i = 0; i < count; ++i)  \*start1++ = \*buffer\_start++;  }  void radix\_sort\_with\_simple\_merge(vector<int>& vec, int nthreads = 1)  {  if (nthreads > 1)  omp\_set\_num\_threads(nthreads);  int num\_of\_numbers = static\_cast<int>(vec.size());  vector<int> buffer(num\_of\_numbers);  #pragma omp parallel shared(buffer, num\_of\_numbers)  {  int num\_of\_threads = omp\_get\_num\_threads();  int bucket\_size = num\_of\_numbers / num\_of\_threads;  if (bucket\_size != 0 && bucket\_size != 1)  {  int tid = omp\_get\_thread\_num();  int begin = tid \* bucket\_size;  int end;  if (tid == num\_of\_threads - 1)  end = num\_of\_numbers;  else end = begin + bucket\_size;  radix\_sort\_by(vec, begin, end, buffer);  #pragma omp barrier  if (num\_of\_threads > 1)  {  for (int size = bucket\_size; size < num\_of\_numbers; size \*= 2)  {  int begin1 = 2 \* tid \* size;  int end1 = begin1 + size;  int begin2 = end1;  int end2 = begin2 + size;  if (abs(num\_of\_numbers - end2) > 0 && abs(num\_of\_numbers - end2) < size)  end2 = num\_of\_numbers;  if (end2 <= num\_of\_numbers)  vec\_merge(vec, begin1, end1, begin2, end2, buffer);  #pragma omp barrier  }  }  }  else  {  #pragma omp single  {  radix\_sort\_by(vec, 0, num\_of\_numbers, buffer);  }  }  }  } |

## TBB алгоритм

|  |
| --- |
| // Функция получения требуемого разряда числа  int get\_dig(int num, int dig\_num)  {  num = abs(num);  for (int i = 0; i < (dig\_num - 1); ++i)  {  num /= 10;  }  return num % 10;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*Сортировка с простым слиянием\*\*\*\*\*\*\*\*\*\*\*\*/  using Iterator = vector<int>::iterator;  struct VecPart  {  int start;  int partsize;  VecPart(int start\_, int partsize\_) : start(start\_), partsize(partsize\_) {}  };  concurrent\_vector<VecPart> parts;  // Поразрядная сортировка части массива  void radix\_sort\_by(vector<int>& vec, const int& begin, const int& end, vector<int>& buffer)  {  VecPart vp(begin, end - begin);  parts.push\_back(vp);  if (end - begin == 1) return;  // максимально возможное число разрядов в числе  int max\_dig\_count = 10;  // максимально возможное значение разряда числа (0..9)  int max\_dig = 10;  // количество чисел  int num\_of\_numbers = static\_cast<int>(vec.size());  // вспомогательные переменные и векторы  int count, tmp;  vector<int> pos\_vec(10);  for (int i = 0; i < max\_dig\_count; ++i)  {  for (int j = 0; j < max\_dig; ++j)  pos\_vec[j] = 0;  for (int j = begin; j < end; ++j)  {  ++pos\_vec[get\_dig(vec[j], i)];  }  count = 0;  for (int j = 0; j < max\_dig; ++j)  {  tmp = pos\_vec[j];  pos\_vec[j] = count;  count += tmp;  }  for (int j = begin; j < end; ++j)  {  buffer[begin + pos\_vec[get\_dig(vec[j], i)]++] = vec[j];  }  for (int j = begin; j < end; ++j)  {  vec[j] = buffer[j];  }  }  int nneg = 0;  for (int i = end - 1, j = begin; i >= begin; --i)  {  if (buffer[i] < 0)  {  ++nneg;  vec[j++] = buffer[i];  }  }  for (int i = begin, j = begin + nneg; i < end; ++i)  {  if (buffer[i] >= 0)  {  vec[j++] = buffer[i];  }  }  }  void vec\_merge(vector<int>& vec, const int& merge\_num, const int& step, vector<int>& buffer)  {  int current\_part\_num = merge\_num \* step \* 2;  if (current\_part\_num + step >= parts.size()) return;  Iterator start1 = vec.begin() + parts[current\_part\_num].start;  Iterator finish1 = start1 + parts[current\_part\_num].partsize;  Iterator start2 = finish1;  Iterator finish2 = start2 + parts[current\_part\_num + step].partsize;  Iterator buffer\_start = buffer.begin() + parts[current\_part\_num].start;  int count = static\_cast<int>(std::distance(start1, finish2));  while (start1 != finish1 && start2 != finish2)  \*buffer\_start++ = \*start1 <= \*start2 ? \*start1++ : \*start2++;  while (start1 != finish1)  \*buffer\_start++ = \*start1++;  while (start2 != finish2)  \*buffer\_start++ = \*start2++;  buffer\_start = buffer.begin() + parts[current\_part\_num].start;  start1 = vec.begin() + parts[current\_part\_num].start;  for (int i = 0; i < count; ++i)  \*start1++ = \*buffer\_start++;  parts[current\_part\_num].partsize += parts[current\_part\_num + step].partsize;  }  // Сортировка с простым слиянием (TBB)  void radix\_sort\_with\_simple\_merge(vector<int>& vec, int nthreads = 1)  {  task\_scheduler\_init init(task\_scheduler\_init::deferred);  int num\_of\_threads;  if (nthreads >= 1)  {  num\_of\_threads = nthreads;  init.initialize(nthreads);  }  else  {  num\_of\_threads = task\_scheduler\_init::default\_num\_threads();  init.initialize(num\_of\_threads);  }  int num\_of\_numbers = static\_cast<int>(vec.size());  int grainsize1 = num\_of\_numbers / num\_of\_threads;  if (grainsize1 == 0)  {  radix\_sort(vec);  return;  }  vector<int> buffer(num\_of\_numbers);  parallel\_for(blocked\_range<int>(0, num\_of\_numbers, grainsize1),  [&vec, &buffer](const blocked\_range<int>& r)  {  radix\_sort\_by(vec, r.begin(), r.end(), buffer);  }  );    parallel\_sort(parts.begin(), parts.end(), [](const VecPart& vp1, const VecPart& vp2) { return vp1.start < vp2.start; });  int parts\_count = static\_cast<int>(parts.size());  int cur\_merge\_count = parts\_count;  for (int step = 1; step < parts\_count; step \*= 2)  {  cur\_merge\_count /= 2;  if (cur\_merge\_count == 0) ++cur\_merge\_count;  parallel\_for(blocked\_range<int>(0, cur\_merge\_count, 1),  [&vec, &buffer, &num\_of\_numbers, &step](const blocked\_range<int>& r)  {  vec\_merge(vec, r.begin(), step, buffer);  }  );  }    if (num\_of\_threads >= 1)  init.terminate();  } |