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 вещественных чисел (тип double) со слиянием “Разделяй и властвуй”»**

**Выполнил:**

студент группы 381508

Казаков Андрей Михайлович

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

Подпись

**Проверил:**

Доцент, к.т.н. Сысоев А.

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

Подпись

Нижний Новгород  
2018

Введение

Сортировка данных является крайне важной задачей в области вычислительной техники. Но несмотря на кажущуюся относительную простоту задачи, придумать эффективный по памяти и времени работы алгоритм оказалось нелёгкой задачей. Первые приходящие на ум алгоритмы работают за квадратичное время, более сложные алгоритмы – уже за *.* Но также существуют алгоритмы, позволяющие сортировать данные почти за линейное время, и таким алгоритмом является поразрядная сортировка. За счёт чего получаются такие результаты? Во-первых, как обычно бывает, придётся пожертвовать памятью. Во-вторых, оказывается, поразрядная сортировка не может быть применена к произвольному типу без специальной модификации: изначально она была предназначена для сортировки целых чисел. В данной лабораторной работе будет рассмотрена одна из таких модификаций поразрядной сортировки для вещественного 64-битного типа double, а также реализация последовательной и параллельной (OpenMP и TBB) версий соответствующего алгоритма.

Постановка задачи

Необходимо реализовать последовательный и параллельный (с применением технологий OpenMP и TBB) алгоритм поразрядной сортировки для типа double (в параллельных версиях использовать сортировку «Разделяй и влавствуй»), а также:

* Написать вспомогательные программы для генерации набора данных для алгоритмов и записи данных в бинарный файл, а также для его последующего чтения и перевод в текстовый файл и наоборот;
* Вспомогательную программу checker для проверки корректности параллельных версий;
* Проверить эффективность параллельных версий на 2, 4, 8 потоках, сравнить с последовательной версией.

Описание алгоритма. Реализация последовательной версии

Чтобы понять, что «делать» с вещественными числами, надо сперва понять общий принцип работы поразрядной сортировки, а также принцип работы алгоритма для сортировки целых чисел.

## Общий принцип работы

Алгоритм, исходя из своего названия, оперирует «разрядами». Для целых чисел (не рассматривая их представления в памяти компьютера) «разрядами» будут являться настоящие разряды числа. В данной лабораторной работе здесь и далее будет упоминаться только «LSD» вариант сортировки (Least Significant Digit, с выравниванием по младшему разряду). «MSD» вариант (Most Significant Digit) не рассмотрен. Сравнение производится поразрядно: сначала сравниваются значения одного крайнего разряда, и элементы группируются по результатам этого сравнения, затем сравниваются значения следующего разряда, соседнего, и элементы либо упорядочиваются по результатам сравнения значений этого разряда внутри образованных на предыдущем проходе групп, либо переупорядочиваются в целом, но сохраняя относительный порядок, достигнутый при предыдущей сортировке. Затем аналогично делается для следующего разряда, и так до конца. Пример:

170, 45, 75, 90, 02, 802, 2, 66 ->

170, 90, 02, 802, 2, 45, 75, 66 ->

02, 802, *0*2, 45, 66, 170, 75, 90 ->

*0*02, *0*02, *0*45, *0*66, *0*75, *0*90, 170, 802

## Сортировка целых чисел: модификация

Чтобы отсортировать числа в памяти компьютера таким алгоритмом, нужно определить понятие «разряда». Вообще, чтобы отсортировать любой тип данных «поразрядно», нужно понять, что будет считаться разрядом. В этом и состоит идея любых модификаций поразрядной сортировки. Так, например, для целых беззнаковых чисел типа int разрядом можно считать каждый отдельный байт.

## Сортировка чисел типа double: модификация и последовательная версия

Оказывается, «трюк» для целых чисел также применим и для знаковых вещественных чисел (но с небольшими оговорками из-за знака): числа нужно также «делить» на байты и работать с ними как с разрядами. Почему это возможно? Из-за представления вещественных чисел по стандарту IEEE-754.

Тип double представляется в памяти компьютера следующим образом: сначала идёт знаковый бит (1 означает отрицательность), 11 битов отведено на порядок числа, остальные 52 на мантиссу. «Обычные» числа исходя из такой формы представления получаются следующим образом:

Число = (-1)[Знак] \* 1.[Мантисса] \* 2[Порядок]-127

Какие выводы можно сделать из данного представления? Для неотрицательных чисел:

* Чем больше порядок, тем больше число;
* При равенстве порядков, вышесказанное верно для мантиссы.

Отсюда незамедлительно следует то, что «приём» для целых беззнаковых чисел будет работать для вещественных беззнаковых чисел. Что верно для отрицательных чисел:

* Во-первых, если «знаковый» байт не интерпретировать по-особому, любое отрицательное число будет любого положительного;
* Все пункты для неотрицательных чисел верны с точностью до наоборот.

Вывод: условно имея правильно реализованный для целых неотрицательных чисел алгоритм, при применении к вещественным знаковым числам, сортировка будет произведена следующим образом:

<Отсортированные по возрастанию неотрицательные числа> <Отсортированные по убыванию отрицательные числа>

Сперва приведём псевдокод поразрядной сортировки для элементов произвольного типа:

// A – исходный массив

// С – вспомогательный массив

// m – кол-во разрядов

// k – кол-во возможных значений разрядов

// n – размер исходного и вспомогательного массивов

// digit – функция получения значения разряда числа

**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

Как видно, нам приходится жертвовать памятью как для создания вспомогательного массива, так и для создания массива для хранения количества значений разряда на данной итерации (т. н. сортировка подсчётом: среди набора сортируемых значений выделяется память, равная количеству всех возможных значений, и при проходе очередного элемента соответствующий ему счётчик увеличивается).

Далее представлен последовательный алгоритм для сортировки вещественных чисел с нужными модификациями и соответствующими комментариями.

// Вспомогательная функция для получения байта числа №digit\_num – байты интерпретируются

// как беззнаковые числа

unsigned char GetByte(const double\* number, const unsigned digit\_num) {

return (reinterpret\_cast<const unsigned char\*>(number))[digit\_num];

}

void LsdRadixSort(double\* arr, const size\_t arr\_size) {

const int kDigitsNumber = 8;

const size\_t kDigitPossibleValuesNumber = 256;

const size\_t kDigitPossibleValuesNumberHalf = kDigitPossibleValuesNumber >> 1;

// Жертвуем памятью ради скорости – вводим массивы для подсчёта значений разрядов

// (сортировка подсчётом)

int\* counters = new int[kDigitPossibleValuesNumber];

double\* arr\_temp = new double[arr\_size];

// Каждая итерация – очередной разряд

for (int i = 0; i < kDigitsNumber; i++) {

// По сути, это проверка для «знаковый байт», байт, содержащий

// первым битом знак числа

const bool is\_curr\_digit\_msd = i == kDigitsNumber - 1;

// Обнуление счётчиков должно производиться на каждой итерации

for (size\_t j = 0; j < kDigitPossibleValuesNumber; j++) {

counters[j] = 0;

}

// Сама сортировка подсчётом

for (size\_t j = 0; j < arr\_size; j++) {

const unsigned char curr\_digit\_unsigned = GetByte(&arr[j], i);

counters[curr\_digit\_unsigned]++;

}

int count = 0;

// Особый случай «знакового байта»

if (is\_curr\_digit\_msd) {

// Для начала, считаем число отрицательных чисел

for (size\_t j = kDigitPossibleValuesNumberHalf; j < kDigitPossibleValuesNumber; j++) {

count += counters[j];

}

// Только для неотрицательных чисел. Count сдвинут, поэтому

// отрицательные числа будут впереди

for (size\_t j = 0; j < kDigitPossibleValuesNumberHalf; j++) {

const int temp = counters[j];

counters[j] = count;

count += temp;

}

count = 0;

// А отрицательные числа необходимо расставить в ОБРАТНОМ ПОРЯДКЕ

for (size\_t j = kDigitPossibleValuesNumber - 1; j >= kDigitPossibleValuesNumberHalf; j--) {

counters[j] += count;

count = counters[j];

}

// Все остальные байты

} else {

for (size\_t j = 0; j < kDigitPossibleValuesNumber; j++) {

const int temp = counters[j];

counters[j] = count;

count += temp;

}

}

// Расставляем числа правильно во временном массиве

for (size\_t j = 0; j < arr\_size; j++) {

const unsigned char curr\_digit\_unsigned = GetByte(&arr[j], i);

// "Если текущее число отрицательное"

if (is\_curr\_digit\_msd && curr\_digit\_unsigned >= kDigitPossibleValuesNumberHalf) {

arr\_temp[--counters[curr\_digit\_unsigned]] = arr[j];

} else {

arr\_temp[counters[curr\_digit\_unsigned]++] = arr[j];

}

}

// Копирование результата

for (size\_t j = 0; j < arr\_size; j++) {

arr[j] = arr\_temp[j];

}

}

delete[] counters;

delete[] arr\_temp;

}

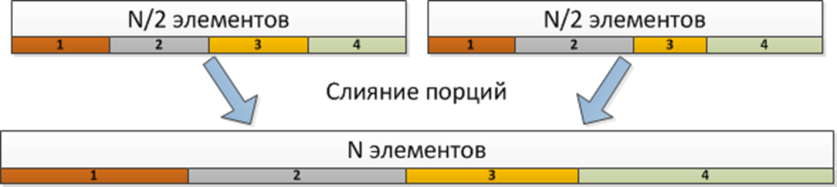
Как и было предположено, особое внимание потребовалось только байту, содержащему знак числа. В целом алгоритм крайне схож с представленным выше псевдокодом.

Параллельная версия с использованием OpenMP

## Описание слияния «Разделяй и властвуй»

Идея слияния по алгоритму «Разделяй и властвуй» заключается в разбиении массивов на участки, которые можно слить независимо. В первом массиве выбирается центральный элемент *x* (он разбивает массив на две равные половины), а во втором массиве с помощью бинарного поиска находится позиция наибольшего элемента, меньшего *x* (позиция этого элемента разбивает второй массив на две части). После такого разбиения первые и вторые половины массивов могут сливать независимо, т.к. в первых половинах находятся элементы меньшие элемента *x*, а во второй – большие (см. рис.). Для слияния двух массивов несколькими потоками можно в первом массиве выбрать несколько ведущих элементов, разделив его на равные порции, а во втором массиве найти соответствующие подмассивы. Каждый поток получит свои порции на обработку.

Эффективность такого слияние во многом зависит от того, насколько равномерно произошло «разделение» второго массива.



Выбранная стратегия распараллеливания данного алгоритма с предложенным слиянием при N потоках состоит в следующем:

1. Сама стратегия предполагает, что число потоков должно быть степенью двойки;
2. Массив разбивается на N равных частей, и каждая часть сортируется отдельным потоком;
3. Каждая пара массивов сливается по описанному выше способу двумя потоками. После всех слияний переход к пункту 2 с соответствующим изменением N.
4. Итого, первоначально будут задействованы N потоков, затем N/2, N/4, …, 2 потока для слияния последней пары.

Код параллельной версии (функция GetByte та же):

// Sorts an array within [index\_start, index\_end) interval

// (по сути та же последовательная версия, но сортирующая заданный участок массива;

// код не приводится из-за незначительности модификаций)

void LsdRadixSortPartial(double\* arr, double\* arr\_temp, const int index\_start, const int index\_end) {

...

}

// Modified binary search to always return something + searches

// within [left\_bound;right\_bound)

int BinarySearch(const double\* arr, const int left\_bound, const int right\_bound, const double elem) {

int left = left\_bound;

int right = right\_bound - 1;

int middle;

if (elem < arr[left]) {

return left;

} else if (elem > arr[right]) {

return right;

}

while (left <= right) {

middle = left + (right - left) / 2;

const double middle\_elem = arr[middle];

if (elem < middle\_elem) {

right = middle - 1;

} else if (elem > middle\_elem) {

left = middle + 1;

} else {

return middle;

}

}

return left;

}

// Performs "divide and conquer" "merge" within the array.

void DacMerge(const double\* arr\_in, double\* arr\_out, const int iteration, const int\* boundaries,

const int left\_bound\_first, const int right\_bound\_first,

const int left\_bound\_second, const int right\_bound\_second) {

const int arr\_len\_first = right\_bound\_first - left\_bound\_first;

const int arr\_len\_second = right\_bound\_second - left\_bound\_second;

const bool merge\_first\_parts\_of\_pair = iteration % 2 == 0;

int curr\_index\_first = 0;

int curr\_index\_second = 0;

int curr\_index\_sorted = merge\_first\_parts\_of\_pair

? boundaries[iteration \* 2]

: boundaries[iteration \* 2 + 1] - arr\_len\_first - arr\_len\_second;

while (curr\_index\_first < arr\_len\_first) {

while (curr\_index\_second < arr\_len\_second

&& arr\_in[left\_bound\_second + curr\_index\_second] < arr\_in[left\_bound\_first + curr\_index\_first]) {

arr\_out[curr\_index\_sorted] = arr\_in[left\_bound\_second + curr\_index\_second];

curr\_index\_second++;

curr\_index\_sorted++;

}

arr\_out[curr\_index\_sorted] = arr\_in[left\_bound\_first + curr\_index\_first];

curr\_index\_first++;

curr\_index\_sorted++;

}

while (curr\_index\_second < arr\_len\_second) {

arr\_out[curr\_index\_sorted] = arr\_in[left\_bound\_second + curr\_index\_second];

curr\_index\_second++;

curr\_index\_sorted++;

}

}

// САМА ПОРАЗРЯДНАЯ СОРТИРОВКА (число потоков указывается внешней программой)

void LsdRadixSort(double\* arr, const size\_t arr\_size) {

if (arr\_size == 1) {

return;

}

int num\_threads;

// This checks if number of threads is 1, OR greater than arr\_size, // OR indivisible by 2^n

#pragma omp parallel

{

if (omp\_get\_thread\_num() == 0) {

num\_threads = omp\_get\_num\_threads();

if (num\_threads == 1 || num\_threads > (int)arr\_size || (num\_threads & (num\_threads - 1))) {

num\_threads = 2;

}

}

}

omp\_set\_num\_threads(num\_threads);

const size\_t arr\_part\_size = arr\_size / num\_threads;

int\* boundaries = new int[num\_threads \* 2];

// We need this array twice, on both steps. To not allocate memory twice, we do it once, here

double\* arr\_temp = new double[arr\_size];

// Step I. Sorting

#pragma omp parallel shared(arr, arr\_temp, arr\_part\_size, arr\_size, num\_threads, boundaries)

{

const int thread\_num = omp\_get\_thread\_num();

const int index\_start = thread\_num \* arr\_part\_size;

int index\_end = index\_start + arr\_part\_size;

// "Last" thread: we must fix index\_end

if (thread\_num == num\_threads - 1) {

index\_end += arr\_size % num\_threads;

}

boundaries[2 \* thread\_num] = index\_start;

boundaries[2 \* thread\_num + 1] = index\_end;

LsdRadixSortPartial(arr, arr\_temp, index\_start, index\_end);

}

// Step II. Merging

int\* middle\_indexes = new int[num\_threads];

int array\_pairs\_num = num\_threads / 2;

while (array\_pairs\_num != 0) {

// Merging preparations

for (int i = 0; i < array\_pairs\_num; i++) {

const int left\_bound\_first = boundaries[i \* 4];

const int right\_bound\_first = boundaries[i \* 4 + 1];

const int left\_bound\_second = boundaries[i \* 4 + 2];

const int right\_bound\_second = boundaries[i \* 4 + 3];

// Find two middle points in the current pair of arrays

const int middle\_index\_first = (left\_bound\_first + right\_bound\_first) / 2;

const int middle\_index\_second = BinarySearch(

arr,

left\_bound\_second,

right\_bound\_second,

arr[middle\_index\_first]

);

middle\_indexes[i \* 2] = middle\_index\_first;

middle\_indexes[i \* 2 + 1] = middle\_index\_second;

}

#pragma omp parallel for schedule(static,1)

for (int i = 0; i < array\_pairs\_num \* 2; i++) {

const bool merge\_first\_parts\_of\_pair = i % 2 == 0;

const int left\_bound\_first = merge\_first\_parts\_of\_pair

? boundaries[i \* 2]

: middle\_indexes[i - 1];

const int right\_bound\_first = merge\_first\_parts\_of\_pair

? middle\_indexes[i]

: boundaries[i \* 2 - 1];

const int left\_bound\_second = merge\_first\_parts\_of\_pair

? boundaries[(i + 1) \* 2]

: middle\_indexes[i];

const int right\_bound\_second = merge\_first\_parts\_of\_pair

? middle\_indexes[i + 1]

: boundaries[i \* 2 + 1];

DacMerge(

arr, arr\_temp, i, boundaries,

left\_bound\_first, right\_bound\_first, left\_bound\_second, right\_bound\_second

);

}

for (int i = 0; i < arr\_size; i++) {

arr[i] = arr\_temp[i];

}

array\_pairs\_num /= 2;

// We must fix pairs boundaries

if (array\_pairs\_num != 0) {

for (int i = 0; i < array\_pairs\_num; i++) {

boundaries[i \* 4] = boundaries[i \* 8];

boundaries[i \* 4 + 1] = boundaries[i \* 8 + 3];

boundaries[i \* 4 + 2] = boundaries[i \* 8 + 4];

boundaries[i \* 4 + 3] = boundaries[i \* 8 + 7];

}

}

} // while (array\_pairs\_num != 0)

delete[] boundaries;

delete[] arr\_temp;

delete[] middle\_indexes;

}

Параллельная версия с использованием TBB

Библиотека TBB диктует другие правила распараллеливания и предоставляет совершенно другие сущности для распараллеливания кода. В данном случае используется сущность TBB «задача»: массив изначально разбивается на 2 части – создаются две «задачи» для сортировки отдельных частей. Далее это разбиение продолжается рекурсивно (соответственно, говоря языком TBB, «спавнятся» соответствующие задачи) до тех пор, пока размер массива не станет меньше или равным 'размер изначального массива поделить на количество потоков'. Такой подход библиотеки TBB также разрешает нам использовать любое количество потоков.

Код параллельной версии (функции GetByte и Binary Search те же):

// Просто немного улучшенная функция LsdRadixSortPartial (эту же функцию можно

// использовать и в OpenMP версии, просто изначально был написан именно тот вариант)

void LsdRadixSortPartial(double\* arr, double\* arr\_temp, const size\_t arr\_size) {

...

}

int BinarySearch(const double\* arr, const int left\_bound, const int right\_bound, const double elem) {

...

}

// Задача слияния двух массивов

class DacMergeTask : public tbb::task {

public:

DacMergeTask(const double\* arr1, const double\* arr2, double\* arr\_temp, const size\_t arr\_size\_1, const size\_t arr\_size\_2) : arr1\_(arr1), arr2\_(arr2), arr\_temp\_(arr\_temp), arr\_size\_1\_(arr\_size\_1), arr\_size\_2\_(arr\_size\_2) {}

tbb::task\* execute() {

int curr\_index\_first = 0;

int curr\_index\_second = 0;

int curr\_index\_sorted = 0;

while (curr\_index\_first < arr\_size\_1\_) {

while (curr\_index\_second < arr\_size\_2\_ && arr2\_[curr\_index\_second] < arr1\_[curr\_index\_first]) {

arr\_temp\_[curr\_index\_sorted] = arr2\_[curr\_index\_second];

curr\_index\_second++;

curr\_index\_sorted++;

}

arr\_temp\_[curr\_index\_sorted] = arr1\_[curr\_index\_first];

curr\_index\_first++;

curr\_index\_sorted++;

}

while (curr\_index\_second < arr\_size\_2\_) {

arr\_temp\_[curr\_index\_sorted] = arr2\_[curr\_index\_second];

curr\_index\_second++;

curr\_index\_sorted++;

}

return nullptr;

}

private:

const double\* arr1\_;

const double\* arr2\_;

double\* arr\_temp\_;

const size\_t arr\_size\_1\_;

const size\_t arr\_size\_2\_;

};

class LsdRadixSortTask : public tbb::task {

public:

LsdRadixSortTask(double\* arr, double\* arr\_temp, const size\_t arr\_size, const size\_t arr\_part\_size) :

arr\_(arr), arr\_temp\_(arr\_temp), arr\_size\_(arr\_size), arr\_part\_size\_(arr\_part\_size) {}

tbb::task\* execute() {

if (arr\_size\_ <= arr\_part\_size\_) {

LsdRadixSortPartial(arr\_, arr\_temp\_, arr\_size\_);

return nullptr;

}

// Need to split the array, create & spawn tasks

LsdRadixSortTask& sort\_task\_1 = \*(new(allocate\_child()) LsdRadixSortTask(

arr\_,

arr\_temp\_,

arr\_size\_ / 2,

arr\_part\_size\_

));

LsdRadixSortTask& sort\_task\_2 = \*(new(allocate\_child()) LsdRadixSortTask(

arr\_ + arr\_size\_ / 2,

arr\_temp\_ + arr\_size\_ / 2,

arr\_size\_ - arr\_size\_ / 2,

arr\_part\_size\_

));

set\_ref\_count(2 + 1); // two tasks above + 1

spawn(sort\_task\_1);

spawn\_and\_wait\_for\_all(sort\_task\_2);

// At this point our current part of the initial array consists of two separate sorted parts.

// Let's merge it via DAC merge

const int middle\_first\_offset = arr\_size\_ / 4;

const double\* middle\_first = arr\_ + middle\_first\_offset;

const double\* middle = arr\_ + arr\_size\_ / 2;

const int middle\_second\_offset = BinarySearch(

middle,

0,

arr\_size\_ - arr\_size\_ / 2,

\*middle\_first

);

const double\* middle\_second = middle + middle\_second\_offset;

DacMergeTask& merge\_task\_1 = \*(new(allocate\_child()) DacMergeTask(

arr\_,

middle,

arr\_temp\_,

middle\_first\_offset,

middle\_second\_offset

));

DacMergeTask& merge\_task\_2 = \*(new(allocate\_child()) DacMergeTask(

middle\_first,

middle\_second,

arr\_temp\_ + (middle\_first\_offset) + (middle\_second\_offset),

(arr\_size\_ / 2) - (middle\_first\_offset),

(arr\_size\_ - arr\_size\_ / 2) - (middle\_second\_offset)

));

set\_ref\_count(1 + 2);

spawn(merge\_task\_1);

spawn\_and\_wait\_for\_all(merge\_task\_2);

for (int i = 0; i < arr\_size\_; i++) {

arr\_[i] = arr\_temp\_[i];

}

return nullptr;

}

private:

double\* arr\_;

double\* arr\_temp\_;

const size\_t arr\_size\_;

const size\_t arr\_part\_size\_;

};

void LsdRadixSort(double\* arr, const size\_t arr\_size, int num\_threads) {

if (arr\_size == 1) {

return;

}

if (num\_threads == 1 || num\_threads > (int)arr\_size) {

num\_threads = 2;

}

double\* arr\_temp = new double[arr\_size];

size\_t arr\_part\_size = arr\_size / num\_threads;

if (arr\_size % num\_threads != 0) {

arr\_part\_size++;

}

tbb::task\_scheduler\_init init(num\_threads);

LsdRadixSortTask& sort\_task = \*(new(tbb::task::allocate\_root()) LsdRadixSortTask(

arr, arr\_temp, arr\_size, arr\_part\_size

));

tbb::task::spawn\_root\_and\_wait(sort\_task);

init.terminate();

delete[] arr\_temp;

}

Вспомогательные программы

Для удобства тестирования параллельных и последовательной версий было предложено создать следующие вспомогательные программы:

* Generator – генерация входных данных для задач в виде бинарных файлов;
* Специальные программы для запуска алгоритмов на данном бинарном файле, которые также генерируют бинарный файл с ответом;
* Checker – программа для сравнения двух бинарных файлов, сгенерированных программами из предыдущего пункта;
* Viewer и Typer – программы для перевода из бинарного формата и текстовый и наоборот соответственно.

Бинарный файл имеет следующий формат: double для хранения времени исполнения, int для размера массива, затем идёт сам массив из чисел типа double.

## Generator

Generator позволяет генерировать массивы вещественных чисел следующих размеров: 2^0, 2^0+1, 2^1, 2^1+1, 2^2, 2^2+1, ..., 2^23, 2^23+1 в произвольном порядке, порядке по возрастанию и порядке по убыванию. Справка по использованию утилиты предоставляется при её вызове из командной строки без аргументов.

Код:

int main(int argc, char\*\* argv) {

const int max\_number\_of\_tests = 46;

if (argc < 3) {

std::cout

<< "Generator\n"

<< "Usage: [.exe name] [order] [test set number 1] [test set number 2] ...\n"

<< "Order of elements: 0 - random, any int > 0 - ascending, any int < 0 - descending\n"

<< "NOTE: set numbers count from 0 to " << max\_number\_of\_tests - 1

<< "; you can type as many test set numbers as you want; "

<< "if the current set doesn't exists nothing will happen."

<< std::endl;

return 1;

}

int number\_of\_tests[max\_number\_of\_tests];

for (int i = 0; i < max\_number\_of\_tests / 2; i++) {

// gonna be 1, 1, 2, 3, 4, 5, 8, 9, 16, 17, 32, 33, 64, 65, 128, 129,...

number\_of\_tests[2 \* i] = 1 << i;

number\_of\_tests[2 \* i + 1] = 1 << i + 1;

}

const int order = atoi(argv[1]);

std::default\_random\_engine generator(std::chrono::system\_clock::now().time\_since\_epoch().count());

std::uniform\_real\_distribution<double> distribution(-1e4, 1e4);

std::set<int> active\_tests;

for (int i = 2; i < argc; i++) {

const int curr\_set\_number = atoi(argv[i]);

if (curr\_set\_number < 0

|| curr\_set\_number > max\_number\_of\_tests - 1) {

continue;

}

active\_tests.insert(curr\_set\_number);

}

for (auto it = active\_tests.begin(); it != active\_tests.end(); ++it) {

char name[32];

\_itoa(\*it, name, 10);

// first arguments means just \*it in string form

FILE\* f = fopen(name, "wb");

const int curr\_tests\_num = number\_of\_tests[\*it];

// double is for the execution time, nothing here now

fwrite(&curr\_tests\_num, sizeof(double), 1, f);

fwrite(&curr\_tests\_num, sizeof(curr\_tests\_num), 1, f);

std::vector<double> temp(curr\_tests\_num);

for (auto it\_temp = temp.begin(); it\_temp != temp.end(); ++it\_temp) {

\*it\_temp = distribution(generator);

}

if (order) {

std::sort(

temp.begin(),

temp.end(),

[order](const int a, const int b) -> bool { return order > 0 ? a < b : a > b; }

);

}

fwrite(temp.data(), sizeof(temp.data()), curr\_tests\_num, f);

fclose(f);

}

return 0;

}

## Программы для запуска версий

Программы для запуска тестовой и OpenMP одинаковые (указание количества потоков при запуске тестовой версии ни к чему не приводит); программа для запуска TBB-версии лишь немногим отличается от первых двух.

Код:

void LsdRadixSort(double\* arr, const size\_t arr\_size);

int main(int argc, char\*\* argv) {

if (argc != 4) {

std::cout

<< "Solver\n"

<< "Usage: [.exe name] [input file] [output file] [number of threads]\n"

<< "Note that the number of threads MUST be divisible by 2."

<< std::endl;

return 1;

}

int number\_of\_threads = atoi(argv[3]);

freopen(argv[1], "rb", stdin);

freopen(argv[2], "wb", stdout);

double time;

int arr\_size;

double\* arr;

fread(&time, sizeof(time), 1, stdin);

fread(&arr\_size, sizeof(arr\_size), 1, stdin);

arr = new double[arr\_size];

fread(arr, sizeof(arr), arr\_size, stdin);

// Часть, специфичная для OMP

omp\_set\_num\_threads(number\_of\_threads);

double elapsed\_time = omp\_get\_wtime();

LsdRadixSort(arr, arr\_size);

elapsed\_time = omp\_get\_wtime() - elapsed\_time;

// Часть, специфичная для TBB

tbb::tick\_count time\_start = tbb::tick\_count::now();

LsdRadixSort(arr, arr\_size, number\_of\_threads);

tbb::tick\_count time\_end = tbb::tick\_count::now();

const double elapsed\_time = (time\_end - time\_start).seconds();

// Продолжение «общего» кода

fwrite(&elapsed\_time, sizeof(elapsed\_time), 1, stdout);

fwrite(&arr\_size, sizeof(arr\_size), 1, stdout);

fwrite(arr, sizeof(arr), arr\_size, stdout);

delete[] arr;

return 0;

}

## Checker

Чекер просто сравнивает два бинарных файла с ответами и записывает результат в текстовый файл в специальном формате.

Код:

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

class Result {

public:

enum ext\_cls { NO = 1, VERDICT, MESSAGE, TIME };

Result() {

result\_file\_.open("result.txt");

}

~Result() {

result\_file\_.close();

}

void write\_verdict(const verdict verdict) {

write\_type(ext\_cls::VERDICT);

result\_file\_ << verdict << " ";

}

void write\_message(const std::string message) {

write\_type(ext\_cls::MESSAGE);

result\_file\_ << message << " ";

}

void write\_time(const long long time) {

write\_type(ext\_cls::TIME);

result\_file\_ << time << " ";

}

private:

std::ofstream result\_file\_;

void write\_type(const ext\_cls type) {

result\_file\_ << type << " ";

}

};

Result checker\_result;

int main(int argc, char\*\* argv) {

if (argc != 3) {

std::cout

<< "Checker\n"

<< "Usage: [.exe name] [input file] [answer file]\n"

<< "Result will be written in result.txt"

<< std::endl;

return 1;

}

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

FILE\* answer = fopen(argv[2], "rb");

double input\_time;

double answer\_time;

int input\_size;

int answer\_size;

bool correct = true;

fread(&input\_time, sizeof(input\_time), 1, input);

fread(&input\_size, sizeof(input\_size), 1, input);

fread(&answer\_time, sizeof(answer\_time), 1, answer);

fread(&answer\_size, sizeof(answer\_size), 1, answer);

if (input\_size == answer\_size) {

double\* inp\_arr = new double[input\_size];

double\* ans\_arr = new double[input\_size];

fread(inp\_arr, sizeof(inp\_arr), input\_size, input);

fread(ans\_arr, sizeof(ans\_arr), input\_size, answer);

for (int i = 0; i < input\_size; i++) {

if (inp\_arr[i] != ans\_arr[i]) {

correct = false;

checker\_result.write\_message("WA: arrays are equal in sizes, but different by themselves");

break;

}

}

delete[] inp\_arr;

delete[] ans\_arr;

} else {

correct = false;

checker\_result.write\_message("WA: arrays are different in sizes");

}

if (correct) {

checker\_result.write\_message("AC: arrays are equal");

checker\_result.write\_verdict(verdict::AC);

} else {

checker\_result.write\_verdict(verdict::WA);

}

checker\_result.write\_time(answer\_time);

fclose(input);

fclose(answer);

return 0;

}

## Typer и Viewer

Простые программы для перевода бинарных файлов в текстовые и наоборот.

Код:

int main(int argc, char\*\* argv) {

if (argc != 3) {

std::cout

<< "Typer\n"

<< "Usage: [.exe name] [input file] [output file]"

<< std::endl;

return 1;

}

std::ifstream input(argv[1]);

FILE\* output = fopen(argv[2], "wb");

double time;

int arr\_size;

double\* arr;

input >> time >> arr\_size;

arr = new double[arr\_size];

for (int i = 0; i < arr\_size; i++) {

input >> arr[i];

}

fwrite(&time, sizeof(time), 1, output);

fwrite(&arr\_size, sizeof(arr\_size), 1, output);

fwrite(arr, sizeof(arr), arr\_size, output);

fclose(output);

input.close();

delete[] arr;

return 0;

}

int main(int argc, char\*\* argv) {

if (argc != 3) {

std::cout

<< "Viewer\n"

<< "Usage: [.exe name] [input file] [output file]"

<< std::endl;

return 1;

}

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

std::ofstream output(argv[2]);

double time;

int arr\_size;

double\* arr;

fread(&time, sizeof(time), 1, input);

fread(&arr\_size, sizeof(arr\_size), 1, input);

std::cout << arr\_size;

arr = new double[arr\_size];

fread(arr, sizeof(arr), arr\_size, input);

output << time << "\n" << arr\_size << "\n";

std::cout << time << "\n" << arr\_size;

for (int i = 0; i < arr\_size; i++) {

output << " " << arr[i];

}

output.close();

fclose(input);

delete[] arr;

return 0;

}

Тестирование последовательных и параллельных версий

Тестирование проводилось с использованием приведённых выше вспомогательных программ на 2-ядерной и 6-ядерной машинах на массивах с 2^21+1 элементами (примерно 2.1 млн. элементов, номер теста – 43 при запуске из «генератора»). Время работы представлено в секундах, в скобках указано ускорение по сравнению с последовательной версией алгоритма.

Результаты на 2 ядрах:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 поток | 2 потока | 4 потока | 8 потоков |
| Послед. | 1.775 | - | - | - |
| OpenMP | - | 0.794 (2.23) | 0.401 (**4.42**) | 0.496 (3.57) |
| TBB | - | 0.496 (**3.57**) | 0.613 (2.9) | 0.857 (2.07) |

Результаты на 6 ядрах (2^22+1 элементов, в 2 раза больше):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 поток | 2 потока | 4 потока | 8 потоков |
| Послед. | 0.37 | - | - | - |
| OpenMP | - | 0.267 (1.39) | 0.216 (**1.71**) | 0.216 (**1.71**) |
| TBB | - | 0.25 (1.48) | 0.25 (1.48) | 0.249 (**1.49**) |

Выводы

В среднем в обоих случаях лучшее ускорение происходило на 4 потоках, на 8 потоках ускорение было даже ниже из-за бОльших накладных расходов. На 2-ядерной машине распараллелирование приносило намного большую выгоду. Но главное, оно всегда и при любом количестве потоков ускоряло программу в 1,5-4 раза.

Несмотря на то, что в большинстве случаев «распараллелить» программу легче с помощью OpenMP (по крайней мере так может показаться интуитивно), в данном случае это оказалось не так. Данная задача проще решалась с помощью рекурсии, что TBB «любит»: писать программы с рекурсией благодаря концепции задач TBB намного легче, чем параллелить их, изобретая собственные эвристики. Да, TBB также выдаёт более плохие результаты, но это опять же связано с ещё бОльшими накладными расходами на внутренние структуры библиотеки. На двух же потоках TBB показал себя лучше OpenMP. Соответственно, для задач с рекурсией, особенно сложной, лучше подойдёт TBB. Для остальных задач рекомендуется написать обе версии и сравнить их эффективность.