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:

**«Сортировка Хоара с простым слиянием»**

**Выполнила:** студентка группы 381508

Зорина Екатерина

Нижний Новгород  
2018

Оглавление

[**Введение** 3](#_Toc514689437)

[**Постановка задачи** 4](#_Toc514689438)

[**Вспомогательные элементы** 5](#_Toc514689439)

[Typer 5](#_Toc514689440)

[Viewer 5](#_Toc514689441)

[Generator 5](#_Toc514689442)

[Checker 5](#_Toc514689443)

[**Метод решения** 6](#_Toc514689444)

[Последовательный алгоритм 6](#_Toc514689445)

[Параллельный алгоритм 7](#_Toc514689446)

[**Результаты экспериментов** 8](#_Toc514689447)

[**Вывод** 9](#_Toc514689448)

# **Введение**

Быстрая сортировка - один из самых быстрых известных универсальных алгоритмов сортировки массивов: в среднем *О(n \* log n) {\displaystyle O(n\log n)}*обменов при упорядочении {\displaystyle n} *n* элементов; из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

Общая идея алгоритма быстрой сортировки состоит в следующем:

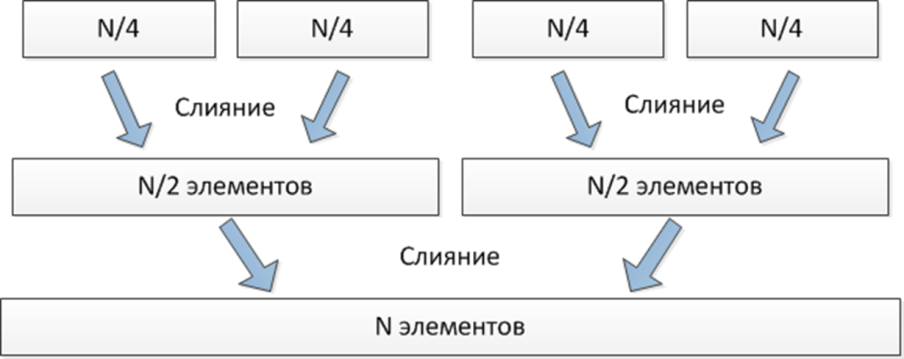
* выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность;
* сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: «меньшие опорного», «равные» и «большие»;
* для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

Разбиение Хоара

Данная схема использует два индекса (один в начале массива, другой в конце), которые приближаются друг к другу, пока не найдётся пара элементов, где один больше опорного и расположен перед ним, а второй меньше и расположен после. Эти элементы меняются местами. Обмен происходит до тех пор, пока индексы не пересекутся. Алгоритм возвращает последний индекс. Данная схема показывает эффективность в *O*(*n*2), когда входной массив уже отсортирован. Сортировка с использованием данной схемы нестабильна. Следует заметить, что конечная позиция опорного элемента необязательно совпадает с возвращённым индексом.

Простое слияние

Идея простого слияния заключается в том, что один поток может выполнять слияние двух отсортированных массивов по классическому алгоритму. В этом случае слияние *n* массивов могут выполнять *n*/2 параллельных потоков. На следующем шаге слияние *n*/2 полученных массивов будут выполнять *n*/4 потоков и т.д. (рис. 8.1).



* + - * 1. Простое слияние

Постановка задачи

В данной работе необходимо реализовать алгоритмы сортировки Хоара с простым слиянием, последовательный и с использованием библиотек OpenMP и Intel© TBB.

**Для выполнения задачи необходимо**:

* Разработать тестовую версию, включающую в себя:
  + solver - решение задачи последовательным алгоритмом, по возможности, максимально простым;
  + before-code – программа-обёртка для solver-а для получения бинарного файла с отсортированным массивом;
  + generator - программа для генерации набора тестовых данных и их сохранения в бинарные файлы;
  + checker - программа для проверки корректности параллельных версий;
  + набор тестов (не менее десяти, из которых не менее пяти "большие") для проверки корректности параллельных версий.
* Разработать параллельную OpenMP версию
* Разработать параллельную TBB версию

Вспомогательные элементы

### **Typer**

Читает текстовый файл, содержащий число - количество элементов массива и сами элементы, и преобразует его в бинарный файл.

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <vector>

int main(int argc, char\* argv[])

{

if (argc != 3)

{

std::cout << "Enter [input.txt] [output.bin] " << std::endl;

return 1;

}

FILE \*in, \*out;

in = fopen(argv[1], "rt");

out = fopen(argv[2], "wb");

char\* num = new char[20];

std::vector<int> arr;

int size = 0;

double time = 0;

if (in != nullptr)

{

while (fgets(num, 20, in) != nullptr)

arr.push\_back(atoi(num));

size = arr.size();

fwrite(&time, sizeof(double), 1, out);

fwrite(&size, sizeof(int), 1, out);

for (int i = 0; i < size; i++)

fwrite(&arr[i], sizeof(int), 1, out);

}

fclose(in);

fclose(out);

delete[] num;

return 0;

}

### **Viewer**

Читает массив из бинарного файла и записывает его в текстовый файл.

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

using namespace std;

int main(int argc, char \* argv[])

{

if (argc != 3)

{

cout << "Enter [output.bin] [output.txt] " << endl;

return 1;

}

double time;

int\* arr;

int size;

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

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

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

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

arr = new int[size];

fread(arr, sizeof(\*arr), size, stdin);

cout << time << endl;

cout << size << endl;

for (int i = 0; i < size; i++)

cout << arr[i] << endl;

return 0;

}

### **Generator**

Генератор тестов. Создает бинарный файл и записывает в него тестовые данные. Принимает на вход целое число от 1 до 20, соответствующее номеру элемента массива { 1, 2, 3, 4, 6, 8, 10, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000, 10000000, 50000000, 100000000 } и создает целочисленный массив с соответствующим количеством элементов. Диапазон элементов: от 0 до 1000000.

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <cstdio>

#include <random>

#include <ctime>

#include <chrono>

#include <iostream>

using namespace std;

int n\_tests[] = { 1, 2, 3, 4, 6, 8, 10, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000, 10000000, 50000000, 100000000 };

int main(int argc, char \* argv[])

{

if ( argc != 2)

{

cout << "Enter number from 1 to 20" << endl;

return 1;

}

int n = n\_tests[atoi(argv[1]) - 1];

double time = 0;

string name = argv[1];

string type = ".bin";

name += type;

freopen(name.c\_str(), "wb", stdout);

default\_random\_engine generator(static\_cast<unsigned int>(chrono::system\_clock::now().time\_since\_epoch().count()));

uniform\_int\_distribution <int> distribution(0, 1000000);

fwrite(&time, sizeof(double), 1, stdout); //фиктивное время

fwrite(&n, sizeof(int), 1, stdout);

int\* array = new int[n];

for (int i = 0; i < n; i++)

{

array[i] = distribution(generator);

}

fwrite(array, sizeof(\*array), n, stdout);

return 0;

}

### **Checker**

Проверяет корректность работы алгоритма, сравнивая результаты работы алгоритма на тестовых изображениях с готовыми ответами и выдает вердикт:

* AC = Accepted = Решение выдаёт корректный результат на данном тесте
* WA = Wrong Answer = Решение выдаёт некорректный результат на данном тесте
* PE = Presentation Error = Ошибка формата выходных данных

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <cstdio>

#include <cmath>

#include <string>

#include <iostream>

using namespace std;

// Используется для взаимодействия с тестирующей системой

////////////////////////////////////////////////////////////////////////////////////////////

/\*

// Checker может устанавливать вот эти три вердикта:

AC = Accepted = Решение выдаёт корректный результат на данном тесте

WA = Wrong Answer = Решение выдаёт некорректный результат на данном тесте

PE = Presentation Error = Ошибка формата выходных данных

// Остальные вердикты checker не может устанавливать

NO = No verdict = Вердикт отсутствует

CE = Compilation Error = Ошибка компиляции

ML = Memory Limit Exceeded = Превышено ограничение по памяти

TL = Time Limit Exceeded = Превышено ограничение по времени работы

RE = Runtime Error = Ошибка времени исполнения программы

IL = Idle Limit Exceeded = Превышено время простоя (бездействия) программы

DE = Deadly Error = Ошибка тестирующей системы

\*/

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

class result {

private:

FILE \* bur;

public:

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

result(bool read = false)

{

if (read)

bur = fopen("result.txt", "r");

else

bur = fopen("result.txt", "w");

}

~result()

{

fclose(bur);

}

void write\_type(ext\_cls t)

{

fwrite(&t, sizeof(t), 1, bur);

}

void write\_verdict(verdict v)

{

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

fwrite(&v, sizeof(v), 1, bur);

}

void write\_message(string str)

{

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

int l = str.size();

fwrite(&l, sizeof(l), 1, bur);

fwrite(&str[0], sizeof(str[0]), l, bur);

}

void write\_time(long long x)

{

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

fwrite(&x, sizeof(x), 1, bur);

}

void write\_memory(unsigned long long x)

{

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

fwrite(&x, sizeof(x), 1, bur);

}

} checker\_result;

int main(int argc, char \* argv[])

{

if (argc != 2)

{

std::cout << "Enter [result.bin] [log.txt]" << std::endl;

return 1;

}

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

int res\_size = 0;

double res\_time = 0;

fread(&res\_time, sizeof(double), 1, buo);

fread(&res\_size, sizeof(int), 1, buo);

bool error = false;

double prev = 0, cur = 0;

fread(&prev, sizeof(prev), 1, buo);

for (int i = 1; i < res\_size; ++i)

{

fread(&cur, sizeof(cur), 1, buo);

if (cur < prev)

{

error = true;

break;

}

prev = cur;

}

if (error == false)

{

checker\_result.write\_message("AC. Sorting was correct.");

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

}

else

{

checker\_result.write\_message("WA. Sorting was incorrect.");

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

}

checker\_result.write\_time(static\_cast<long long>(res\_time \* 1e7));

fclose(buo);

return 0;

}

Реализация

### **Последовательный алгоритм**

#include <omp.h>

void HoaraQuickSort(int\* arr, int first, int last)

{

int i = first, j = last;

int temp, mid = arr[(first + last) / 2];

do {

while (arr[i] < mid)

i++;

while (arr[j] > mid)

j--;

if (i <= j)

{

if (i < j)

{

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

i++;

j--;

}

} while (i <= j);

if (i < last)

HoaraQuickSort(arr, i, last);

if (first < j)

HoaraQuickSort(arr, first, j);

}

### **Параллельный алгоритм**

Здесь, помимо последовательного алгоритма сортировки Хоара, добавлены вспомогательные функции, такие как CopyArray (копирует содержимого массива) и Merge (объединяет отсортированные массивы в один).

#include <tbb\parallel\_for.h>

#include <tbb\tick\_count.h>

#include <tbb\task\_scheduler\_init.h>

#define \_SCL\_SECURE\_NO\_WARNINGS

#include "conio.h"

#include <math.h>

#include <omp.h>

#include <ctime>

#include "iostream"

#include <algorithm>

#include "sol.cpp"

void copyArray(int\* first, int\* last, int\* res)

{

while (first != last + 1)

{

(\*res) = (\*first);

++first;

++res;

}

}

void Merge(int\* arr1\_begin, int\* arr1\_end, int\* arr2\_begin, int\* arr2\_end, int\* dest)

{

while ((arr1\_begin <= arr1\_end) && (arr2\_begin <= arr2\_end))

{

auto elem1 = \*arr1\_begin;

auto elem2 = \*arr2\_begin;

if (elem1 < elem2)

{

\*dest = elem1;

++arr1\_begin;

}

else

{

\*dest = elem2;

++arr2\_begin;

}

++dest;

}

if (arr1\_begin > arr1\_end)

{

copyArray(arr2\_begin, arr2\_end + 1, dest);

}

else /\*arr2\_begin > arr2\_end\*/

{

copyArray(arr1\_begin, arr1\_end + 1, dest);

}

}

void HoaraQuickSort\_OMP(int\* a, int\* sorted\_a, int n)

{

int threads\_count = omp\_get\_max\_threads();

int chunk\_size = n / threads\_count ? n / threads\_count : 1;

#pragma omp parallel for

for (int i = 0; i < threads\_count; ++i)

{

auto first = i \* chunk\_size;

auto last = first + chunk\_size - 1;

HoaraQuickSort(a, first, last);

}

auto ret = n % threads\_count;

if (ret)

{

HoaraQuickSort(a, threads\_count \* chunk\_size, n - 1);

}

//reduction

int array\_size = chunk\_size;

int count = array\_size << 1;

int j = 0;

while (array\_size < n)

{

for (j = 0; j <= n - count; j += count)

{

int\* first\_array\_start = a + j;

int\* first\_array\_end = first\_array\_start + array\_size - 1;

int\* second\_array\_start = a + j + array\_size;

int\* secon\_array\_end = second\_array\_start + array\_size - 1;

Merge(first\_array\_start, first\_array\_end, second\_array\_start, secon\_array\_end, sorted\_a + j);

}

copyArray(sorted\_a, sorted\_a + j, a);

array\_size = (array\_size << 1) > n ? n : array\_size << 1;

count = array\_size << 1;

}

if (ret)

{

int array\_size = n - ret;

int\* first\_array\_start = a;

int\* first\_array\_end = first\_array\_start + array\_size - 1;

int\* second\_array\_start = a + array\_size;

int\* second\_array\_end = second\_array\_start + ret - 1;

Merge(first\_array\_start, first\_array\_end, second\_array\_start, second\_array\_end, sorted\_a);

}

copyArray(sorted\_a, sorted\_a + n, a);

}

void HoaraQuickSort\_TBB(int\* a, int\* sorted\_a, int n)

{

int threads\_count = tbb::task\_scheduler\_init::default\_num\_threads();

int chunk\_size = n / threads\_count ? n / threads\_count : 1;

tbb::parallel\_for(tbb::blocked\_range<int>(0, threads\_count), [&a, chunk\_size](auto range)

{

for (auto i = range.begin(); i != range.end(); ++i)

{

auto first = i \* chunk\_size;

auto last = first + chunk\_size - 1;

HoaraQuickSort(a, first, last);

}

}

);

auto ret = n % threads\_count;

if (ret)

{

HoaraQuickSort(a, threads\_count \* chunk\_size, n - 1);

}

//reduction

int array\_size = chunk\_size;

int count = array\_size << 1;

int j = 0;

while (array\_size < n)

{

for (j = 0; j <= n - count; j += count)

{

int\* first\_array\_start = a + j;

int\* first\_array\_end = first\_array\_start + array\_size - 1;

int\* second\_array\_start = a + j + array\_size;

int\* secon\_array\_end = second\_array\_start + array\_size - 1;

Merge(first\_array\_start, first\_array\_end, second\_array\_start, secon\_array\_end, sorted\_a + j);

}

copyArray(sorted\_a, sorted\_a + j, a);

array\_size = (array\_size << 1) > n ? n : array\_size << 1;

count = array\_size << 1;

}

if (ret)

{

int array\_size = n - ret;

int\* first\_array\_start = a;

int\* first\_array\_end = first\_array\_start + array\_size - 1;

int\* second\_array\_start = a + array\_size;

int\* secon\_array\_end = second\_array\_start + ret - 1;

Merge(first\_array\_start, first\_array\_end, second\_array\_start, secon\_array\_end, sorted\_a);

}

copyArray(sorted\_a, sorted\_a + n, a);

}

Результаты экспериментов

|  |  |  |  |
| --- | --- | --- | --- |
| Размер сортируемого массива | Последовательная версия | OpenMP версия | TBB версия |
| 100 | 3.7235e-06 | 0.00368875 | 0.011164 |
| 1 000 | 4.71644e-05 | 0.00372009 | 0.0112329 |
| 10 000 | 0.00547665 | 0.00376787 | 0.0111469 |
| 100 000 | 0.00600663 | 0.00569696 | 0.0129696 |
| 1 000 000 | 0.0613481 | 0.0314133 | 0.0507206 |
| 10 000 000 | 0.607631 | 0.227472 | 0.21845 |
| 100 000 000 | 6.08653 | 2.17494 | 2.19073 |

Эксперимент проводился на компьютере с процессором Intel® Core™ i7-3960X, обладающим 12 ядрами. Из полученных данных можно заключить, что параллельные алгоритмы работают в 2-3 раза быстрее последовательной версии при достаточно большом размере сортируемого массива (от миллиона элементов) и медленнее на маленьких массивах, при этом TBB-реализация проигрывает во времени на порядок и OpenMP-реализации. Однако, на самых больших тестах видно, что TBB алгоритм превосходит в скорости OpenMP.

Вывод

В результате работы были созданы программы, в которых реализованы как последовательная, так и параллельная версии сортировки Хоара с простым слиянием.

В результате замера времени работы различных версий алгоритма было выяснено, что параллельные версии с использованием библиотек OpenMP и TBB работают быстрее последовательной версии, когда длина массива превышает миллион элементов (для данной машины). При этом стоит отметить неэффективность параллельных алгоритмов для сортировки массивов небольшой длины.