Изображение выглядит как черно-белый, строительство, небо, монохромный

Автоматически созданное описание

Московский государственный университет имени М.В.Ломоносова

Факультет вычислительной математики и кибернетики

Кафедра системного программирования

Задание по курсу:

«Параллельные высокопроизводительные вычисления»

Отчёт по практическому заданию № 1:

«Расписание сети сортировки»

Арефьев Вениамин Андреевич

528 группа

Москва

11.11.2023

Содержание

[Описание условий 3](#_Toc150631214)

[Описание метода решения 5](#_Toc150631215)

[Описание метода проверки 7](#_Toc150631216)

[Приложение № 1: исходный текст программы 8](#_Toc150631217)

Описание условий

Разработать последовательную программу вычисления расписания сети сортировки, числа использованных компараторов и числа тактов, необходимых для её срабатывания при выполнении на n процессорах. Число тактов сортировки при параллельной обработке не должно превышать числа тактов, затрачиваемых четно-нечетной сортировкой Бетчера.

**Параметр командной строки запуска**:

* n, где n>=1 – количество элементов в упорядочиваемом массиве, элементы которого расположены на строках с номерами [0…n-1]

**Формат команды запуска**:

bsort n

**Требуется**:

1. вывести в файл стандартного вывода расписание и его характеристики в представленном далее формате;
2. обеспечить возможность вычисления сети сортировки для числа элементов 1<=n<=10000;
3. предусмотреть полную проверку правильности сети сортировки для значений числа сортируемых элементов 1<=n<=24;
4. представить краткий отчет удовлетворяющий указанным далее требованиям.

**Формат файла результата**:

Начало файла результата

n 0 0

cu0 cd0

cu1 cd1

…

cun\_comp-1 cdn\_comp-1

n\_comp

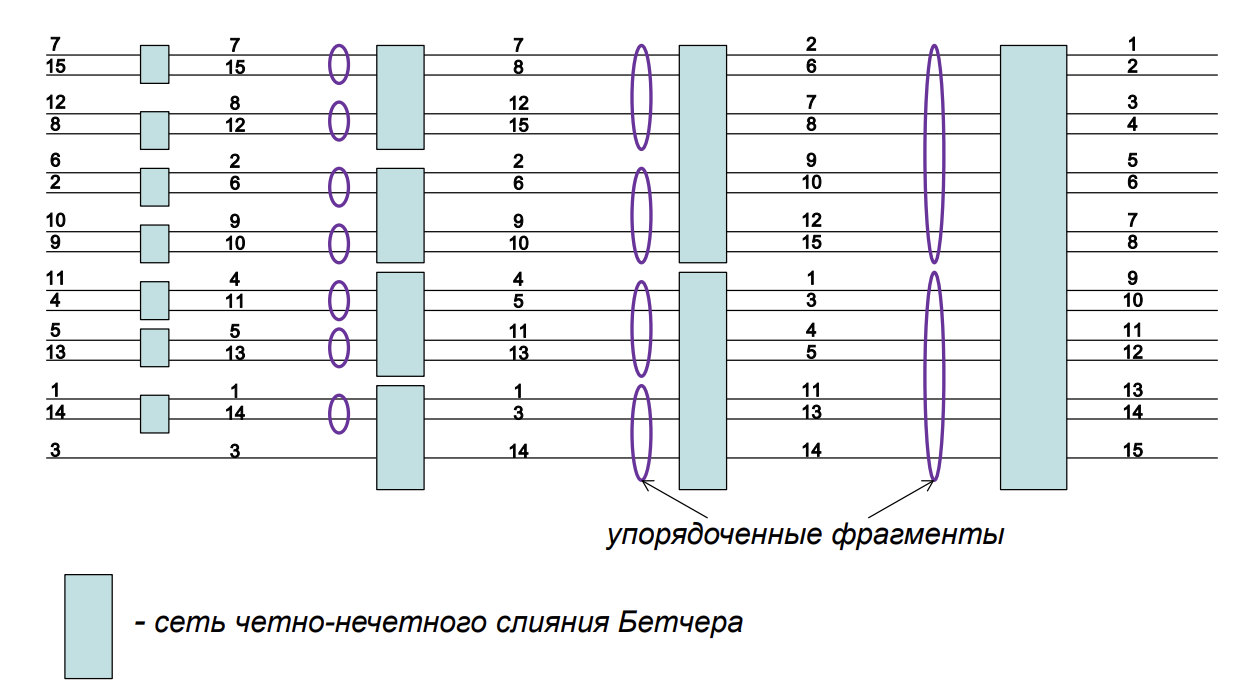
n\_tact

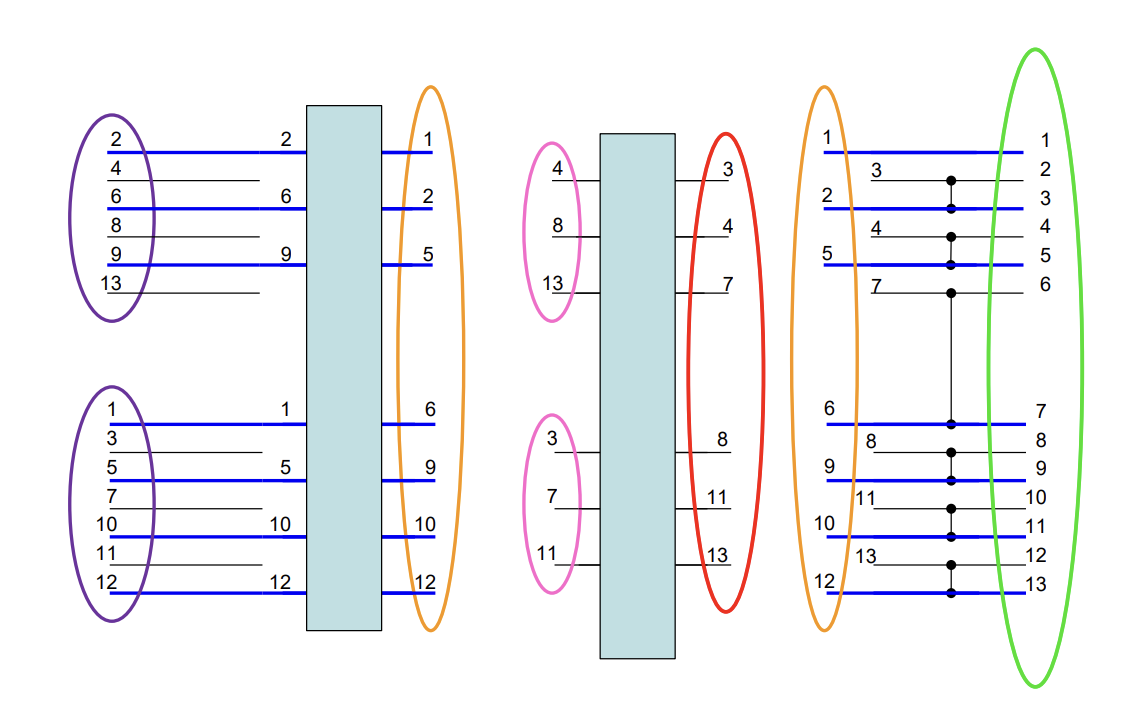
Конец файла результата

Здесь:

* n 0 0 – число сортируемых элементов, ноль, ноль. Да, вывести число элементов и два нуля.
* cui cdi – номера строк, соединяемых i-м компаратором сравнения перестановки.
* n\_comp – число компараторов
* n\_tact – число тактов сети сортировкифывфыв

Описание метода решения

В ходе решения был выбрал и реализован рекурсивный алгоритм сортировки на основе четно-нечетного слияния Бэтчера. Общая схема работы проиллюстрирована на рисунках 1 и 2.Рисунок 1. Сортировка массива из 15 элементов.

 Рисунок 2. Сортировка массива из 13 элементов.

Работа алгоритма описывается следующими шагами:

1. Разделить исходный массив длины P на две части. Первая часть длины n =p/2, округленная вниз. А вторая часть длины m=p-n
2. Отсортировать получившиеся подмассивы, причём в каждой части происходит выделение чётных и нечётных элементов и отдельных сортировка этих пар
3. Произвести слияние отсортированных подмассивов с помощью сети чётно-нечётного слияния Бэтчера

В реализации первому пункту соответствует функция batcherSort, а второму и третьему – sortAndMergeTwoArrays.

Разработка велась на языке C++ с использованием стандарта С++20.   
Компиляция осуществлялась с помощью CMake, либо с помощью Makefile.

Описание метода проверки

Для запуска проверки с помощью 0–1 принципа необходимо вместо числа, как первого аргумента командной строки, подать строку «test». Тогда будут отсортированы всевозможные массивы длины n, где n принимает значения в диапазоне [1,24]. При правильной работе сортиров будет выведена 1, что она и вывела при запуске.

Для проверки правильности количества тактов и числа компараторов был написать скрипт, который запускает программу с аргументом n, который принимает значения в диапазоне [1,24]. Таким образом получается две последовательности – количество компараторов и тактов для каждой итерации. В таблицах 1, 2 и 3 можно увидеть количество тактов (cycles) и число использованных компараторов (comp) в зависимости от n, выведенные программой, а также аналитическую оценку число тактов (cycles\_max).

Аналитическая оценка получена с помощью формулы:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| comp | 0 | 1 | 3 | 5 | 9 | 12 | 16 | 19 |
| cycles | 0 | 1 | 3 | 3 | 5 | 6 | 6 | 6 |
| cycles\_max | 0 | 1 | 3 | 3 | 6 | 6 | 6 | 6 |

Таблица 1.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| comp | 26 | 31 | 37 | 41 | 48 | 53 | 59 | 63 |
| cycles | 8 | 9 | 10 | 10 | 10 | 10 | 10 | 10 |
| cycles\_max | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |

Таблица 2.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| Comp | 74 | 82 | 91 | 97 | 107 | 114 | 122 | 127 |
| Cycles | 12 | 13 | 14 | 14 | 15 | 15 | 15 | 15 |
| cycles\_max | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 |

Таблица 3.

Как видно из таблицы, фактическое количество задействованных тактов не превосходит аналитической оценки, что подтверждает правильность алгоритма.

Приложение № 1: исходный текст программы

#include <iostream>  
#include <cmath>  
#include <vector>  
#include <set>  
#include <memory>  
  
int comparator\_count = 0;  
bool is\_printing = true;  
std::vector<std::set<size\_t> > compare\_cycles;  
  
enum {  
 MAX\_TEST\_ARRAY\_SIZE = 24  
};  
  
void addPairToCycles(size\_t first, size\_t second) {  
 if (compare\_cycles.empty()) {  
 compare\_cycles.push\_back(std::set<size\_t>{first, second});  
 return;  
 }  
 int cycles\_count = (int) compare\_cycles.size();  
 for (int i = cycles\_count - 1; i >= 0; i--) {  
 // check last comparison conflict  
 if (compare\_cycles[i].contains(first) or compare\_cycles[i].contains(second)) {  
 // check whether a new cycle needs to be added  
 if (i < cycles\_count - 1) {  
 compare\_cycles[i + 1].insert(first);  
 compare\_cycles[i + 1].insert(second);  
 } else {  
 compare\_cycles.push\_back(std::set<size\_t>{first, second});  
 }  
 return;  
 }  
 }  
 // without any conflicts suspected -> add elements to first cycle  
 compare\_cycles[0].insert(first);  
 compare\_cycles[0].insert(second);  
}  
  
void myComparator(std::pair<long, long> &first, std::pair<long, long> &second) {  
 if (is\_printing) {  
 std::cout << first.first << " " << second.first << std::endl;  
 }  
 if (first.second > second.second) {  
 std::swap(first.second, second.second);  
 }  
 comparator\_count++;  
 addPairToCycles(first.first, second.first);  
}  
  
void sortAndMergeTwoArrays(const std::shared\_ptr<std::pair<long, long>[]> &array,  
 long first\_size,  
 long second\_size,  
 long start = 0) {  
 if (first\_size <= 0 or second\_size <= 0) {  
 return;  
 } else if (first\_size == 1 and second\_size == 1) {  
 myComparator(array[start + 0], array[start + 1]);  
 return;  
 } else {  
 // calculate index  
 auto first\_even\_size = (long) round((double) first\_size / 2);  
 long first\_odd\_size = first\_size - first\_even\_size;  
 auto second\_even\_size = (long) round((double) second\_size / 2);  
 long second\_odd\_size = second\_size - second\_even\_size;  
  
 // create sub array and fill them  
 auto even\_prt = std::shared\_ptr<std::pair<long, long>[]>(  
 new std::pair<long, long>[first\_even\_size + second\_even\_size]);  
  
 for (long i = 0; i < first\_even\_size; i++) {  
 even\_prt[i] = array[start + 2 \* i];  
 }  
 for (long i = 0; i < second\_even\_size; i++) {  
 even\_prt[first\_even\_size + i] = array[start + first\_size + 2 \* i];  
 }  
  
 auto odd\_prt = std::shared\_ptr<std::pair<long, long>[]>(  
 new std::pair<long, long>[first\_odd\_size + second\_odd\_size]);  
  
 for (long i = 0; i < first\_odd\_size; i++) {  
 odd\_prt[i] = array[start + 2 \* i + 1];  
 }  
 for (long i = 0; i < second\_odd\_size; i++) {  
 odd\_prt[first\_odd\_size + i] = array[start + first\_size + 2 \* i + 1];  
 }  
  
 // sort sub arrays  
 sortAndMergeTwoArrays(even\_prt, first\_even\_size, second\_even\_size);  
 sortAndMergeTwoArrays(odd\_prt, first\_odd\_size, second\_odd\_size);  
  
 // merge sub arrays  
 for (long i = 0; i < first\_even\_size; i++) {  
 array[start + 2 \* i] = even\_prt[i];  
 }  
 for (long i = 0; i < second\_even\_size; i++) {  
 array[start + first\_size + 2 \* i] = even\_prt[first\_even\_size + i];  
 }  
 for (long i = 0; i < first\_odd\_size; i++) {  
 array[start + 2 \* i + 1] = odd\_prt[i];  
 }  
 for (long i = 0; i < second\_odd\_size; i++) {  
 array[start + first\_size + 2 \* i + 1] = odd\_prt[first\_odd\_size + i];  
 }  
  
 // count comparators in merging  
 for (long i = 1; i < first\_size + second\_size - 1; i += 2) {  
 myComparator(array[start + i], array[start + i + 1]);  
 }  
 }  
}  
  
void batcherSort(const std::shared\_ptr<std::pair<long, long>[]> &array\_ptr, long cur\_arr\_size, long start = 0) {  
 // stop recursion for all small array  
 if (cur\_arr\_size < 2) {  
 return;  
 }  
 long mid\_size = cur\_arr\_size / 2;  
  
 batcherSort(array\_ptr, mid\_size, start);  
 batcherSort(array\_ptr, cur\_arr\_size - mid\_size, start + mid\_size);  
 sortAndMergeTwoArrays(array\_ptr, mid\_size, cur\_arr\_size - mid\_size, start);  
}  
  
bool zeroOneTestBatcherSort() {  
 is\_printing = false;  
 auto array\_prt = std::shared\_ptr<std::pair<long, long>[]>(new std::pair<long, long>[MAX\_TEST\_ARRAY\_SIZE]);  
  
 for (long cur\_array\_size = 1, cur\_max\_value = 2;  
 cur\_array\_size <= MAX\_TEST\_ARRAY\_SIZE; cur\_array\_size++, cur\_max\_value \*= 2) {  
 for (long i = 0; i < cur\_max\_value; i++) {  
 auto cur\_temp = i;  
 for (long j = 0; j < cur\_array\_size; j++) {  
 array\_prt[j] = std::make\_pair(j, cur\_temp % 2);  
 cur\_temp /= 2;  
 }  
 batcherSort(array\_prt, cur\_array\_size);  
 for (long j = 1; j < cur\_array\_size; j++) {  
 if (array\_prt[j] < array\_prt[j - 1]) {  
 return false;  
 }  
 }  
 }  
 }  
 return true;  
}  
  
int main(int argc, char \*\*argv) {  
 if (argc != 2) {  
 std::cout << "Program accept only one additional argument" << std::endl;  
 return 0;  
 }  
 if (std::string(argv[1]) == "test") {  
 std::cout << zeroOneTestBatcherSort();  
 return 0;  
 }  
 long array\_size = strtol(argv[1], nullptr, 10);  
 if (array\_size <= 0) {  
 std::cout << "First argument must be positive" << std::endl;  
 return 0;  
 }  
 std::cout << array\_size << " 0 0" << std::endl;  
  
 auto array\_prt = std::shared\_ptr<std::pair<long, long>[]>(new std::pair<long, long>[array\_size]);  
  
 for (long i = 0; i < array\_size; i++) {  
 array\_prt[i] = std::make\_pair(i, i \* (i + 23) % 1653);  
 }  
  
 batcherSort(array\_prt, array\_size);  
 std::cout << comparator\_count << std::endl << compare\_cycles.size() << std::endl;  
 return 0;  
}