Московский государственный университет имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра системного программирования

Параллельные высокопроизводительные вычисления

Отчёт по заданию 1. Расписание сети сортировки

Выполнила:
студентка 527 группы
Парыгина Дарья Алексеевна

Дата подачи:
10.11.2023 г.

Москва, 2023

Содержание

1 Описание условия

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 Бэтчера.

Параметр командной строки запуска: $n >= 1$ – 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.

2 Описание метода решения

В качестве решения предлагается реализовать рекурсивный алгоритм нечётно-чётной сортировки, который применяется для построения сети обменной сортировки Бэтчера. Сети Бэтчера являются одними из наиболее быстродействующих из масштабируемых сетей сортировки.

Рекурсивный алгоритм заключается в следующем: для сортировки массива, содержащего p элементов с номерами от 0 до p - 1, нужно разделить его на две части. Первая из них должна содержать n = [p/2] (с округлением вверх) элементов с номерами [0, ..., n - 1], а вторая – m = p – n элементов с номерами [n, ..., p - 1]. Далее следует отсортировать каждую из частей и объединить результаты сортировки с помощью (n, m)-сети нечётно-чётного слияния Бэтчера. В сети нечётно-чётного слияния отдельно объединяются элементы массивов с нечётными номерами и отдельно – с чётными, после чего с помощью заключительной группы компараторов обрабатываются пары соседних элементов с номерами вида (2i, 2i + 1), где i — натуральные числа от 0 до [p/2] - 2 (с округлением вниз).

Для формирования сети достаточно знать номера линий данных, поэтому алгоритм является независимым от значений элементов упорядочиваемого массива. Группу линий удобно описывать тройкой чисел вида (idx, step, count) – линии с номерами idx, idx + step, ..., idx + (count - 1) * step. Для формирования множества применяемых компараторов используются рекурсивные процедуры build_net и merge_lines.

build_net(idx, step, count) – процедура рекурсивного построения сети сортировки группы линий (idx, step, count), выполняющая разделение группы линий на две части, их рекурсивную обработку и слияние результатов при помощи вызова функции merge_lines.

merge_lines(idx1, idx2, step, count1, count2) – рекурсивная процедура слияния двух групп линий (idx1, step, count1) и (idx2, step, count2) при помощи нечётно-чётного слияния Бэтчера.

Построение требуемой сети сортировки массива обеспечивается вызовом процедуры build_net(0, 1, p), так как тройка (0, 1, p) описывает все линии данных массива.

3 Описание метода проверки

Проверка правильности сети сортировки для значений числа сортируемых элементов $1 <= n <= 24$ может быть установлена с использованием принципа нулей и единиц, согласно которому: если сеть с p входами сортирует в порядке неубывания все 2^p последовательности из 0 и 1, то она будет сортировать в том же порядке любую последовательность p чисел. Таким образом, для выполнения проверки необходимо применить полученную сеть сортировки ко всем последовательностям из 0 и 1 длины p и проверить, что полученные в результате сортировки массивы являются упорядоченными по неубыванию. Успешное прохождение тестов характеризуется выводом строки "Tests passed!"на стандартный поток вывода.

4 Приложение 1

```
#include <iostream>
#include <vector>
#include <cstdint>

static std::vector<std::pair<size_t, size_t>> comparators;

void add_comp(size_t idx1, size_t idx2)
{
    comparators.push_back(std::make_pair(idx1, idx2));
}

void merge_lines(size_t idx1, size_t idx2, size_t step, size_t count1,
   size_t count2)
{
    if (count1 * count2 < 1)
    {
        return;
    }
    if (count1 == 1 && count2 == 1)
    {
        add_comp(idx1, idx2);
        return;
    }

    size_t count11 = count1 - count1 / 2;
    size_t count22 = count2 - count2 / 2;
    merge_lines(idx1, idx2, 2 * step, count11, count22);
    merge_lines(idx1 + step, idx2 + step, 2 * step, count1 - count11, count2
    - count22);
```

```
    size_t i;
    for (i = 1; i < count1 - 1; i += 2)
    {
        add_comp(idx1 + step * i, idx1 + step * (i + 1));
    }

    if (count1 % 2 == 0)
    {
        add_comp(idx1 + step * (count1 - 1), idx2);
        i = 1;
    }
    else
    {
        i = 0;
    }

    for (; i < count2 - 1; i += 2)
    {
        add_comp(idx2 + step * i, idx2 + step * (i + 1));
    }

}

void build_net(size_t idx, size_t step, size_t count)
{
    if (count < 2)
    {
        return;
    }

    if (count == 2)
```

```
    {
        add_comp(idx, idx + step);
        return;
    }

    size_t count1 = count / 2 + count % 2;
    build_net(idx, step, count1);
    build_net(idx + step * count1, step, count - count1);

    merge_lines(idx, idx + step * count1, step, count1, count - count1);
}

size_t count_tacts(size_t count)
{
    std::vector<size_t> comp_execution_tacts(count, 0);
    size_t tacts = 0;
    for (int32_t i = 0; i < comparators.size(); ++i)
    {
        size_t first = comparators[i].first;
        size_t second = comparators[i].second;
        size_t exec_tact = std::max(comp_execution_tacts[first],
   comp_execution_tacts[second]) + 1;
        comp_execution_tacts[first] = comp_execution_tacts[second] =
   exec_tact;
        if (exec_tact > tacts)
        {
            tacts = exec_tact;
        }
    }
    return tacts;
}
```

```
void compare(std::vector<int32_t>& data, std::pair<size_t, size_t>
   comparator)
{
    size_t first = comparator.first;
    size_t second = comparator.second;
    if (data[first] > data[second])
    {
        std::swap(data[first], data[second]);
    }
}

void check_sort(size_t count)
{
    if (count > 24)
    {
        return;
    }
    std::vector<int32_t> data(count);
    for (int32_t i = 0; i < 1 << count; ++i)
    {
        int32_t num = i;
        for (int32_t j = count - 1; j >= 0; --j)
        {
            data[j] = num & 1;
            num >>= 1;
        }

        for (size_t j = 0; j < comparators.size(); ++j)
        {
            compare(data, comparators[j]);
        }
        for (size_t j = 0; j < count - 1; ++j)
```

```
        {
            if (data[j] > data[j + 1])
            {
                std::cout << "Test failed for " << i << std::endl;
                return;
            }
        }
    }
    std::cout << "Tests passed!" << std::endl;

}

int main(int argc, char *argv[])
{
    size_t count;
    if (argc < 2)
    {
        return 1;
    }
    count = std::stoi(argv[1]);

    build_net(0, 1, count);

    std::cout << count << " 0 0" << std::endl;
    for (const auto &p : comparators)
    {
        std::cout << p.first << " " << p.second << std::endl;
    }
    std::cout << comparators.size() << std::endl;
    std::cout << count_tacts(count) << std::endl;

    check_sort(count);
```

```
    return 0;
}
```