Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение
высшего образования
Национальный исследовательский Нижегородский государственный
университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Отчет по лабораторной работе

«Сортировка Шелла с четно-нечетным слиянием Бэтчера»

Выполнил:
студент группы 381806-1
Власов М. С.

Проверил:
доцент кафедры МОСТ,
кандидат технических наук
Сысоев А. В.

Нижний Новгород
2020

Оглавление

Введение

Как правило, как людям, так и компьютерам удобнее работать с такими наборами или списками данных, которые упорядочены по определенному свойству. Одним из способов упорядочения множества данных является сортировка.

Но обычно сортировки представляют собой алгоритмы, требующие многократных обходов сортируемого диапазона. Отсюда следует, что чем больше количество объектов, которые необходимо отсортировать, тем больше времени требуется для сортировки. Уменьшить затраты времени на сортировку может помочь распараллеливание реализации алгоритма с помощью его запуска на нескольких процессорах вычислительного оборудования.

Таким образом, мы можем разделить исходный набор данных на части, для каждой вычислительной единицы поровну, отсортировать их параллельно, а затем объединить, чтобы получить набор с начальным размером. Эффективным подходом для такого слияния в данном случае может служить применение алгоритма четно-нечетного слияния Бэтчера.

Итак, в данной лабораторной работе будет реализован один из алгоритмов сортировок - сортировка Шелла, а также алгоритм четно-нечетного слияния Бэтчера.

Постановка задачи

В рамках лабораторной работы необходимо реализовать последовательную и параллельную реализации сортировки Шелла для целых чисел с четно-нечетным слиянием Бэтчера, проверить корректность работы алгоритмов, провести эксперименты для оценки эффективности параллелизации. По полученным результатам сделать выводы.

Для реализации параллельной версии необходимо использовать библиотеку межпроцессорного взаимодействия MPI. Для проверки корректности работы алгоритмов используется Google Testing Framework.

Описание алгоритмов

Сортировка Шелла - алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками.

Идея метода заключается в сравнении разделенных на группы элементов последовательности, находящихся друг от друга на некотором расстоянии. Изначально это расстояние равно d или n/2, где n — общее число элементов. На первом шаге каждая группа включает в себя два элемента расположенных друг от друга на расстоянии n/2. Они сравниваются между собой и, в случае необходимости, меняются местами. На последующих шагах также происходят проверка и обмен, но расстояние d сокращается на d/2, и количество групп, соответственно, уменьшается. Постепенно расстояние между элементами уменьшается, и на d=1 проход по массиву происходит в последний раз.

Сортировка Шелла уступает в эффективности быстрой сортировке, но, очевидно, выигрывает у сортировки вставками.

Теперь расскажем про четно-нечетное слияние Бэтчера. Метод заключается в том, что два упорядоченных массива, которые необходимо слить, разделяются на чётные и нечётные элементы. Такое слияние может быть выполнено параллельно. Чтобы массив стал окончательно отсортированным, достаточно сравнить пары элементов, стоящие на нечётной и чётной позициях. В цикле по массиву, который хранит в себе компараторы (то есть, пары значений - номера процессов), процессы с номерами, входящие в такую пару, должны поделиться между собой данными. Затем каждый процесс выполняет упорядоченное слияние своего фрагмента массива с полученным, причем первый и второй процессы сохраняют в свои фрагменты первую и вторую половины получившегося массива соответственно.

Схема распараллеливания

Для распараллеливания алгоритма сортировки Шелла необходимо разделить массив на равные по размеру фрагменты, количество которых равно количеству процессов. Если размер исходного массива не делится нацело на количество процессов, необходимо добавить в его конец необходимое количество максимальных по значению для используемого типа данных элементов, которые на результат сортировки не повлияют, поскольку в итоге вновь окажутся в конце. Затем корневой процесс отдает каждому процессу фрагмент исходного массива, после чего все они должны быть отсортированы с помощью сортировки Шелла.

Далее необходимо параллельным образом выполнить четно-нечетное слияние Бэтчера. Таким образом, у каждого процесса оказывается фрагмент массива такой, что если соединить их со всех процессов друг за другом, то получим полностью отсортированный массив. Фиктивные элементы из конца массива необходимо удалить, если они были добавлены в начале.

Описание программной реализации

Алгоритм последовательной сортировки Шелла вызывается с помощью функции:

```
std::vector<int> shellSort(std::vector<int> arr);
```

Входным параметром функции является вектор, который необходимо отсортировать. Выходными данными является отсортированный по возрастанию вектор.

Все нижеследующие функции, используемые в параллельной реализации алгоритма четно-нечетного слияния Бэтчера, помещены в одноименное пространство имен:

```
namespace BatcherMerge { ... }
```

Реализация распараллеливания сортировки с помощью слияния Бэтчера представлена в функции:

```
std::vector<int> parallelSort(std::vector<int> arr,
    std::function<std::vector<int>(std::vector<int>)> sort_func);
```

В качестве входных параметров передается массив, который необходимо отсортировать, а также сортирующий функтор, принимающий и возвращающий массив. В процессе распараллеливания сортировки этот функтор будет вызван, чтобы получить отсортированный фрагмент массива в каждом процессе.

Эта функция напрямую или косвенно вызывает внутри себя несколько других, перечисленных ниже.

Данная функция выполняет объединение (конкатенацию) двух массивов:

```
std::vector<int> join(const std::vector<int>& first, const std::vector<int>&
    second);
```

Входными параметрами являются ссылки на неизменяемые массивы, которые необходимо объединить. Выходными данными является объединенный массив.

Данная функция выполняет построение сети (массива) компараторов:

```
void buildNetwork(const std::vector<int>& ranks);
```

Входным параметром является ссылка на неизменяемый массив с номерами процессов, на которых нужно построить сеть (распараллелить слияние).

Данная функция выполняет рекурсивное слияние двух групп номеров процессов:

```
void mergeNetwork(const std::vector<int>& ranks_up, const std::vector<int>&
    ranks_down);
```

В сети нечетно-четного слияния отдельно объединяются элементы массивов с нечетными номерами и отдельно с четными, после чего с помощью заключительной группы компараторов обрабатываются пары соседних элементов. Данные пары записываются в массив компараторов для дальнейшего использования.

Входными параметрами функции являются ссылки на неизменяемые массивы с номерами процессов каждой группы.

Подтверждение корректности

Для подтверждения корректности в программе представлен набор тестов, разработанных с Google Testing Framework.

Набор представляет из себя тесты, которые проверяют корректность вычислений (сравнивается вектор, полученный благодаря параллельной сортировке, с вектором, отсортированным с помощью последовательной сортировки), а также эффективность (вычисление занимаемого последовательной и параллельной сортировок времени и сравнение полученных данных).

Успешное прохождение всех тестов подтверждает корректность работы написанной программы.

Результаты экспериментов

Эксперименты для оценки эффективности проводились на ПК со следующими характеристиками:

- Процессор: AMD A6-9225 Radeon R4, 2.60 GHz, 2 ядра;
- Оперативная память: 8 ГБ;
- ОС: Microsoft Windows 10 Pro Build 19042.685.

Для проведения экспериментов производилась сортировка Шелла - последовательно и параллельно (с четно-нечетным слиянием Бэтчера) - для векторов размером 524 288 (2^{19}) значений. Результаты представлены в таблице ниже.

Таблица 1: Результаты экспериментов

Процессы	Последовательно	Параллельно	Ускорение
1	3.41823	3.45914	0.98817
2	3.59880	1.90034	1.89377
4	3.92024	2.33415	1.67951

По данным, полученным в результате экспериментов, можно сделать вывод о том, что параллельный случай работает действительно быстрее, чем последовательный.

Если увеличить количество процессов до 4, возможно получить деградацию ускорения. Так происходит из-за увеличения накладных расходов: приходится выделять больше памяти под отсортированные части массива, происходит больше пересылок данных между процессами.

При дальнейшем увеличении количества процессов ускорения при данном количестве процессорных ядер компьютера, очевидно, не будет.

Заключение

В ходе выполнения лабораторной работы были разработаны последовательная реализация сортировки Шелла и параллелизация с четно-нечетным слиянием Бэтчера.

Задача работы была успешно достигнута, поскольку результаты проведенных экспериментов по оценке эффективности показывают, что параллельная реализация работает быстрее, чем последовательная.

Кроме того, были написаны тесты с использованием Google Testing Framework, необходимые для подтверждения корректности работы программы.

Литература

1. Википедия: Сортировка Шелла [Электронный ресурс] // URL: `https://en.wikipedia.org/wiki/Shellsort` (дата обращения: 09.12.2020)

2. Habr: Сеть обменной сортировки со слиянием Бэтчера [Электронный ресурс] // URL: `https://habr.com/ru/post/275889/` (дата обращения: 09.12.2020)

Приложение

В этом разделе находится листинг всего кода, написанного в рамках лабораторной работы.

```
// shell_sort_batcher_merge.h

// Copyright 2020 Vlasov Maksim
#ifndef
    MODULES_TASK_3_VLASOV_M_SHELL_SORT_BATCHER_MERGE_SHELL_SORT_BATCHER_MERGE_H_
#define
    MODULES_TASK_3_VLASOV_M_SHELL_SORT_BATCHER_MERGE_SHELL_SORT_BATCHER_MERGE_H_
#include <functional>
#include <vector>

using Vector = std::vector<int>;

Vector createRandomVector(int size);

Vector shellSort(Vector arr);

namespace BatcherMerge {
    Vector parallelSort(Vector arr, std::function<Vector(Vector)> sort_func);
}  // namespace BatcherMerge

#endif  //
    MODULES_TASK_3_VLASOV_M_SHELL_SORT_BATCHER_MERGE_SHELL_SORT_BATCHER_MERGE_H_
```

```
// shell_sort_batcher_merge.cpp
// Copyright 2020 Vlasov Maksim
#include <mpi.h>
#include <cmath>
#include <algorithm>
#include <functional>
#include <limits>
#include <numeric>
#include <random>
#include <utility>
#include <vector>
#include "./shell_sort_batcher_merge.h"


Vector createRandomVector(int elements_count) {
    std::random_device rd;
    std::mt19937 generator(rd());
    Vector result(elements_count);
    for (int& elem : result)
        elem = static_cast<int>(generator() % 100u);
    return result;
}

Vector shellSort(Vector arr) {
    auto size = arr.size();
```

```
    for (auto step = size / 2; step > 0; step /= 2)
        for (auto i = step; i < size; i++)
            for (auto j = i; j >= step && arr[j] < arr[j - step]; j -= step)
                std::swap(arr[j], arr[j - step]);
    return arr;
}

namespace BatcherMerge {
    using Comparator = std::pair<int, int>;
    std::vector<Comparator> comparators;

    Vector join(const Vector& first, const Vector& second) {
        Vector temp(0);
        temp.reserve(first.size() + second.size());
        temp.insert(temp.end(), first.begin(), first.end());
        temp.insert(temp.end(), second.begin(), second.end());
        return temp;
    }

    void mergeNetwork(const Vector& ranks_up, const Vector& ranks_down) {
        size_t size = ranks_up.size() + ranks_down.size();
        if (size == 1)
            return;
        if (size == 2) {
            comparators.emplace_back(ranks_up.front(), ranks_down.front());
            return;
        }

        Vector ranks_up_odd, ranks_up_even;
        for (size_t i = 0; i < ranks_up.size(); i++) {
            if (i % 2 == 0)
                ranks_up_odd.push_back(ranks_up[i]);
            else
                ranks_up_even.push_back(ranks_up[i]);
        }
        Vector ranks_down_odd, ranks_down_even;
        for (size_t i = 0; i < ranks_down.size(); i++) {
            if (i % 2 == 0)
                ranks_down_odd.push_back(ranks_down[i]);
            else
                ranks_down_even.push_back(ranks_down[i]);
        }

        mergeNetwork(ranks_up_odd, ranks_down_odd);
        mergeNetwork(ranks_up_even, ranks_down_even);

        Vector temp_comp = join(ranks_up, ranks_down);
        for (size_t i = 1; i < temp_comp.size() - 1; i += 2)
            comparators.emplace_back(temp_comp[i], temp_comp[i + 1]);
    }

    void buildNetwork(const Vector& ranks) {
        size_t size = ranks.size();
        if (size < 2)
            return;

        size_t ranks_up_size = size / 2;
        Vector ranks_up{ ranks.begin(), ranks.begin() + ranks_up_size };
```

```
    Vector ranks_down{ ranks.begin() + ranks_up_size, ranks.end() };

    buildNetwork(ranks_up);
    buildNetwork(ranks_down);
    mergeNetwork(ranks_up, ranks_down);
}

Vector parallelSort(Vector arr, std::function<Vector(Vector)> sort_func) {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int arr_size = static_cast<int>(arr.size());
    if (arr_size < 2)
        return arr;
    if (arr_size <= size)
        return sort_func(arr);

    int extra_size = static_cast<int>(std::pow(2, std::ceil(std::log2(arr_size
        + arr_size % size)))) - arr_size;
    arr_size += extra_size;
    arr.resize(arr_size, std::numeric_limits<int>::max());
    int part_size = arr_size / size;

    Vector ranks(size);
    std::iota(ranks.begin(), ranks.end(), 0);
    buildNetwork(ranks);

    Vector part(part_size), part_curr(part_size), part_temp(part_size);
    MPI_Scatter(arr.data(), part_size, MPI_INT, part.data(), part_size,
        MPI_INT, 0, MPI_COMM_WORLD);
    part = sort_func(part);

    for (const auto& comp : comparators) {
        if (rank == comp.first) {
            MPI_Send(part.data(), part_size, MPI_INT, comp.second, 0,
                MPI_COMM_WORLD);
            MPI_Recv(part_curr.data(), part_size, MPI_INT, comp.second, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            for (int i = 0, i_curr = 0, i_temp = 0; i_temp < part_size;
                i_temp++) {
                int value = part[i];
                int value_curr = part_curr[i_curr];
                if (value < value_curr) {
                    part_temp[i_temp] = value;
                    i++;
                } else {
                    part_temp[i_temp] = value_curr;
                    i_curr++;
                }
            }
            std::swap(part, part_temp);
        } else if (rank == comp.second) {
            MPI_Recv(part_curr.data(), part_size, MPI_INT, comp.first, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            MPI_Send(part.data(), part_size, MPI_INT, comp.first, 0,
                MPI_COMM_WORLD);
            size_t i_start = part_size - 1;
```

```
                for (int i = i_start, i_curr = i_start, i_temp = part_size; i_temp
                    > 0; i_temp--) {
                    int value = part[i];
                    int value_curr = part_curr[i_curr];
                    if (value > value_curr) {
                        part_temp[i_temp - 1] = value;
                        i--;
                    } else {
                        part_temp[i_temp - 1] = value_curr;
                        i_curr--;
                    }
                }
                std::swap(part, part_temp);
            }
        }
        MPI_Gather(part.data(), part_size, MPI_INT, arr.data(), part_size,
            MPI_INT, 0, MPI_COMM_WORLD);
        arr_size -= extra_size;
        arr.resize(arr_size);
        return arr;
    }
}  // namespace BatcherMerge



// main.cpp
// Copyright 2020 Vlasov Maksim
#include <mpi.h>
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <iostream>
#include <vector>
#include "./shell_sort_batcher_merge.h"

TEST(Parallel_Shell_Sort_Batcher_Merge_MPI, Size_10) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<int> arr(10);
    if (rank == 0)
        arr = createRandomVector(10);
    auto check_arr = BatcherMerge::parallelSort(arr, shellSort);
    MPI_Barrier(MPI_COMM_WORLD);
    if (rank == 0) {
        auto exp_arr = shellSort(arr);
        ASSERT_EQ(exp_arr, check_arr);
    }
}

TEST(Parallel_Shell_Sort_Batcher_Merge_MPI, Size_15) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<int> arr(15);
    if (rank == 0)
        arr = createRandomVector(15);
    auto check_arr = BatcherMerge::parallelSort(arr, shellSort);
    if (rank == 0) {
        auto exp_arr = shellSort(arr);
        ASSERT_EQ(exp_arr, check_arr);
```

```
    }
}

TEST(Parallel_Shell_Sort_Batcher_Merge_MPI, Size_100) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<int> arr(100);
    if (rank == 0)
        arr = createRandomVector(100);
    auto check_arr = BatcherMerge::parallelSort(arr, shellSort);
    if (rank == 0) {
        auto exp_arr = shellSort(arr);
        ASSERT_EQ(exp_arr, check_arr);
    }
}

TEST(Parallel_Shell_Sort_Batcher_Merge_MPI, Size_500) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<int> arr(500);
    if (rank == 0)
        arr = createRandomVector(500);
    auto check_arr = BatcherMerge::parallelSort(arr, shellSort);
    if (rank == 0) {
        auto exp_arr = shellSort(arr);
        ASSERT_EQ(exp_arr, check_arr);
    }
}

TEST(Parallel_Shell_Sort_Batcher_Merge_MPI, Size_1000) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<int> arr(1000, 0);
    if (rank == 0)
        arr = createRandomVector(1000);
    auto check_arr = BatcherMerge::parallelSort(arr, shellSort);
    if (rank == 0) {
        auto exp_arr = shellSort(arr);
        ASSERT_EQ(exp_arr, check_arr);
    }
}

int main(int argc, char *argv[]) {
    ::testing::InitGoogleTest(&argc, argv);
    MPI_Init(&argc, &argv);

    ::testing::AddGlobalTestEnvironment(new GTestMPIListener::MPIEnvironment);
    ::testing::TestEventListeners &listeners =
        ::testing::UnitTest::GetInstance()->listeners();

    listeners.Release(listeners.default_result_printer());
    listeners.Release(listeners.default_xml_generator());

    listeners.Append(new GTestMPIListener::MPIMinimalistPrinter);
    return RUN_ALL_TESTS();
}
```