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

«Поразрядная сортировка для целых чисел с четно-нечетным слиянием Бэтчера»

Выполнил:
студент группы 381708-1
Андронов М.Н.

Проверил:
Доцент кафедры МОСТ, к. т. н,
Сысоев А. В.

Нижний Новгород
2019

Содержание

Введение

Сортировка – одна из наиболее распространенных операций обработки данных. Она используется в множестве задач в разных областях. На больших объемах данных время работы, даже наиболее оптимальных методов сортировки, занимает достаточно много времени. Для достижения большей производительности необходимо разбить исходные данные на массивы меньшего размера, произвести упорядочивание подмассивов параллельно, а в конце слить их в один массив.

Целью данной работы является реализация параллельной поразрядная сортировки для целых чисел с четно-нечетным слиянием Бэтчера и сравнение ее производительности с последовательной поразрядной сортировкой.

Постановка задачи

1. Реализовать последовательный алгоритм поразрядной сортировки для целых чисел.
2. Реализовать параллельный алгоритм поразрядной сортировки для целых чисел с четно-нечетным слиянием Бэтчера.
3. Провести вычислительные эксперименты.
4. Сравнить время работы последовательного и параллельного алгоритмов.

Метод решения

Идея поразрядной сортировки заключается в том, что выполняется последовательная сортировка чисел по разрядам (от младшего разряда к старшему). Но так как современные процессоры предназначены для обработки данных, которые представлены в битах и байтах, выполнять сортировку по десятичным разрядам чисел не эффективно. Рассмотрим побайтовую реализацию поразрядной сортировки, т.е. будем рассматривать число как набор 256-значных цифр. Стоит заметить, что поразрядная сортировка массива будет работать только в том случае, если сортировка, выполняющаяся по разряду, является устойчивой. В случае побайтовой реализации для сортировки разряда будет удобно использовать сортировку подсчетом (с модификацией, которая не будет менять взаимного расположения элементов равных разрядов).

Сортировка подсчетом по i-му байту будет проходить в два прохода по исходному массиву:

1. при первом проходе по исходному массиву выполняется подсчет i-ых байт в массиве, результат будет сохранен в массив подсчетов counter из 256 элементов;
2. в массив offset на основании посчитанных данных выполняется подсчет смещений, по которым будут сохраняться элементы:

```
offset[0]=0
для всех j от 1 до 255
offset[j]=counter[j-1]+offset[j-1]
```

3. при втором проходе по исходному массиву выполняется копирование элемента во вспомогательный массив по соответствующему индексу в массиве смещений offset и выполняется инкремент смещения.

Старший бит знаковых типов определяет знак числа, поэтому сортировку нужно выполнять с учетом того, что числа у которых старший бит равен 0 (положительные) больше тех чисел у которых старший бит равен 1 (отрицательные). Исходя из этого, получается, что при сортировке знаковых чисел необходимо изменить алгоритм сортировки только по старшему байту. Для этого достаточно в сортировке подсчетом интерпретировать старший байт, как знаковое число (диапазон знаковых однобайтовых чисел от -128 до 127) и прибавлять смещение 128. Таким образом, в массиве подсчетов по индексу 0 будет находиться количество элементов массива, у которых в старшем байте минимальное число со знаком (-128).

Идея четно-нечетного слияния Бэтчера заключается в том, что два упорядоченных массива, которые необходимо слить, разделяются на четные и нечетные элементы. Далее выполняется слияние отдельно четных и нечетных элементов массивов. А после слияние четного и нечетного подмассивов в один. Чтобы массив стал окончательно отсортированным, достаточно сравнить пары элементов, стоящие на нечетной и четной позициях.

Схема распараллеливания

В начале, если количество элементов в массиве меньше чем количество процессов, создаётся новый коммуникатор с количеством процессов равным количеству элементов.

Далее происходит разбиение массива на группы меньшего размера. Если равномерно разбить массив не удалось, то оставшиеся элементы распределяются между первыми n (n – количество оставшихся элементов) процессами. Каждый процесс поразрядной сортировкой сортирует свой подмассив.

Теперь необходимо слить k подмассивов (k – количество процессов). Для этого необходимо m итераций. Число итераций будет равно сумме степеней 2 на которые можно разложить k (любое число можно разложить на степени двойки). Вводиться понятие “шага слияния”, для того чтобы знать с каким подмассивом сливать подмассив текущего процесса. Результат слияния на текущей итерации всегда будет храниться в четном процессоре текущего шага слияния. Если на текущей итерации процесс не учавствует в слиянии, то он переходит на следующую итерацию.

По итогу отсортированный массив будет расположен в процессе с рангом 0.

Описание программной реализации

В программе реализовано 5 вспомогательных методов:

unsigned char GetByte(int value, int number) – возвращает байт с номером number числа value.

std::vector<int> EvenSplit(std::vector<int> arr1, std::vector<int> arr2) – выполняет выборку и слияние четных элементов двух отсортированных массивов.

std::vector<int> OddSplit(std::vector<int> arr1, std::vector<int> arr2) – выполняет выборку и слияние нечетных элементов двух отсортированных массивов.

std::vector<int> EvenOddSplit(std::vector<int> arr1, std::vector<int> arr2) – выполняет слияние массива с четными элементами arr1 и нечетными элементами arr2.

std::vector<int> GetRandomVector(int size) – возвращает случайный вектор размера size.

А также основные методы:

std::vector<int> RadixSort(std::vector<int> array) – выполняет последовательную поразрядную сортировки для целых чисел.

std::vector<int> ParallelRadixSortBatcherSplit(std::vector<int> array, int size_arr) – выполняет параллельную поразрядной сортировки для целых чисел с четно-нечетным слиянием Бэтчера.

Подтверждение корректности

Для подтверждения корректности в программе реализован набор тестов с использованием библиотеки для модульного тестирования Google C++ Testing Framework.

Тесты проверяют корректную работу последовательного и параллельного алгоритмов сортировки на массивах с четным и нечетным количество элементов. Полученный массив сравнивается с массивом, отсортированным при помощи функции sort из библиотеки algorithm.

Результаты экспериментов

Эксперименты проводились на ПК с следующей конфигурацией:

- Операционная система: Windows 7 Максимальная
- Процессор: Intel(R) Core™ i5-3210M CPU @ 2.50 GHz
- ОЗУ 4 гб
- Версия Visual Studio: 2017

Размер массива 1000000 элементов.

Таблица 1. Результаты экспериментов

Количество процессов	Время работы последовательного алгоритма, сек	Время работы параллельного алгоритма, сек	Ускорение
1	1,999	2,003	0,998
2	2,006	1,601	1,253
3	1,997	1,510	1,323
4	2,008	1,319	1,523
5	1,999	1,312	1,524

По данным экспериментов видно, что использование параллельной поразрядной сортировки для целых чисел с четно-нечетным слиянием Бэтчера более эффективно, чем использование последовательного алгоритма поразрядной сортировки.

Заключение

В ходе работы были реализованы два алгоритма сортировки массивов: последовательная поразрядная сортировка и параллельная поразрядной сортировки с четно-нечетным слиянием Бэтчера. Вычислительные эксперименты показали, что последовательный алгоритм уступает в производительности параллельному алгоритму, так как работает на большем объеме данных. Этот факт делает параллельную сортировку более предпочтительной для использования в реальных вычислительных задачах.

Список литературы

1. Национальный открытый университет ИНТУИТ [Электронный ресурс]. – Режим доступа: https://www.intuit.ru/studies/courses/10612/1096/lecture/22924?page=3
2. Хабр, сообщество IT-специалистов [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/261777/

Приложение

radix_with_batcher_split.h

```
// Copyright 2019 Andronov Maxim
#ifndef MODULES_TASK_3_ANDRONOV_M_RADIX_WITH_BATCHER_SPLIT_RADIX_WITH_BATCHER_SPLIT_H_
#define MODULES_TASK_3_ANDRONOV_M_RADIX_WITH_BATCHER_SPLIT_RADIX_WITH_BATCHER_SPLIT_H_

#include <mpi.h>
#include <vector>

std::vector<int> GetRandomVector(int size);

std::vector<int> RadixSort(std::vector<int> array);

std::vector<int> ParallelRadixSortBatcherSplit(std::vector<int> array,
                                               int size_arr);

#endif  // MODULES_TASK_3_ANDRONOV_M_RADIX_WITH_BATCHER_SPLIT_RADIX_WITH_BATCHER_SPLIT_H_
```

radix_with_batcher_split.cpp

```
// Copyright 2019 Andronov Maxim
#include <random>
#include <vector>
#include <utility>
#include
"../../../modules/task_3/andronov_m_radix_with_batcher_split/radix_with_batcher_split.h"

std::vector<int> GetRandomVector(int size) {
    if (size < 1)
        throw - 1;

    std::vector <int> vec;

    vec.resize(size);

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution <int> dist(-100, 100);

    for (int i = 0; i < size; i++)
        vec[i] = dist(gen);

    return vec;
}

unsigned char GetByte(int value, int number) {
    return (value >> (number * 8)) & 255;
}

std::vector<int> RadixSort(std::vector<int> array) {
    if (array.size() < 1)
        throw - 1;
    if (array.size() == 1)
        return array;

    std::vector<int> tmp(array.size());
    std::vector<int> counter(256, 0);
    std::vector<int> offset(256, 0);
    bool flag = true;

    for (int i = 0; i < 3; i++) {
        for (size_t j = 0; j < array.size(); j++) {
            unsigned char byte = GetByte(array[j], i);
            counter[byte]++;
```

```
        }
        offset[0] = 0;
        for (int k = 1; k < 256; k++) {
            offset[k] = counter[k - 1] + offset[k - 1];
        }
        for (size_t j = 0; j < array.size(); j++) {
            if (flag) {
                unsigned char value = GetByte(array[j], i);
                tmp[offset[value]] = array[j];
                offset[value]++;
            } else {
                int value = GetByte(tmp[j], i);
                array[offset[value]] = tmp[j];
                offset[value]++;
            }
        }
        flag = !flag;
        for (int k = 0; k < 256; k++) {
            counter[k] = 0;
        }
    }
    for (size_t j = 0; j < array.size(); j++) {
        char byte = static_cast<char>(GetByte(array[j], 3));
        counter[byte + 128]++;
    }
    offset[0] = 0;
    for (int k = 1; k < 256; k++) {
        offset[k] = counter[k - 1] + offset[k - 1];
    }
    for (size_t j = 0; j < array.size(); j++) {
        char value = GetByte(tmp[j], 3);
        array[offset[value + 128]] = tmp[j];
        offset[value + 128]++;
    }

    return array;
}

std::vector<int> EvenSplit(std::vector<int> arr1, std::vector<int> arr2) {
    int arr1_size = arr1.size();
    int arr2_size = arr2.size();
    int result_size = arr1_size / 2 + arr2_size / 2
                          + arr1_size % 2 + arr2_size % 2;
    std::vector<int> result(result_size);
    int i = 0;
    int j = 0, k = 0;

    while ((j < arr1_size) && (k < arr2_size)) {
        if (arr1[j] <= arr2[k]) {
            result[i] = arr1[j];
            j += 2;
        } else {
            result[i] = arr2[k];
            k += 2;
        }
        i++;
    }

    if (j >= arr1_size) {
        for (int l = k; l < arr2_size; l += 2) {
            result[i] = arr2[l];
            i++;
        }
    } else {
        for (int l = j; l < arr1_size; l += 2) {
            result[i] = arr1[l];
```

```
            i++;
        }
    }

    return result;
}

std::vector<int> OddSplit(std::vector<int> arr1, std::vector<int> arr2) {
    int arr1_size = arr1.size();
    int arr2_size = arr2.size();
    int result_size = arr1_size / 2 + arr2_size / 2;
    std::vector<int> result(result_size);
    int i = 0;
    int j = 1, k = 1;

    while ((j < arr1_size) && (k < arr2_size)) {
        if (arr1[j] <= arr2[k]) {
            result[i] = arr1[j];
            j += 2;
        } else {
            result[i] = arr2[k];
            k += 2;
        }
        i++;
    }

    if (j >= arr1_size) {
        for (int l = k; l < arr2_size; l += 2) {
            result[i] = arr2[l];
            i++;
        }
    } else {
        for (int l = j; l < arr1_size; l += 2) {
            result[i] = arr1[l];
            i++;
        }
    }

    return result;
}

// arr1 - even elements | arr2 - odd elements
std::vector<int> EvenOddSplit(std::vector<int> arr1, std::vector<int> arr2) {
    int arr1_size = arr1.size();
    int arr2_size = arr2.size();
    int result_size = arr1_size + arr2_size;
    std::vector<int> result(result_size);
    int i = 0, j = 0, k = 0;

    while ((j < arr1_size) && (k < arr2_size)) {
        result[i] = arr1[j];
        result[i + 1] = arr2[k];
        i += 2;
        j++;
        k++;
    }

    if ((k >= arr2_size) && (j < arr1_size)) {
        for (int l = i; l < result_size; l++) {
            result[l] = arr1[j];
            j++;
        }
    }

    for (int i = 0; i < result_size - 1; i++) {
        if (result[i] > result[i + 1])
```

```
            std::swap(result[i], result[i + 1]);
    }

    return result;
}

std::vector<int> ParallelRadixSortBatcherSplit(std::vector<int> array,
    int size_arr) {
    int size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size == 1)
        return RadixSort(array);

    MPI_Comm MPI_COMM_TASK = MPI_COMM_NULL;

    if (size_arr < size) {
        MPI_Group global_group, new_group;
        MPI_Comm_group(MPI_COMM_WORLD, &global_group);
        std::vector<int> new_group_ranks;
        for (int i = 0; i < size_arr; i++)
            new_group_ranks.push_back(i);
        MPI_Group_incl(global_group, size_arr, new_group_ranks.data(),
            &new_group);
        MPI_Comm_create(MPI_COMM_WORLD, new_group, &MPI_COMM_TASK);
    }
    else {
        MPI_COMM_TASK = MPI_COMM_WORLD;
    }

    if (MPI_COMM_TASK == MPI_COMM_NULL)
        return array;

    std::vector<int> local_array;

    if (MPI_COMM_TASK != MPI_COMM_NULL) {
        MPI_Comm_size(MPI_COMM_TASK, &size);
        MPI_Comm_rank(MPI_COMM_TASK, &rank);

        int delta = size_arr / size;
        int delta_rem = size_arr % size;

        std::vector<int> sendcounts(size, delta);

        for (int i = 0; i < delta_rem; i++)
            sendcounts[i]++;

        std::vector<int> displ(size, 0);
        for (int i = 1; i < size; i++) {
            displ[i] = displ[i - 1] + sendcounts[i - 1];
        }

        local_array.resize(sendcounts[rank]);
        MPI_Scatterv(array.data(), sendcounts.data(),
            displ.data(), MPI_INT, local_array.data(), sendcounts[rank],
            MPI_INT, 0, MPI_COMM_TASK);

        local_array = RadixSort(local_array);

        int iteration_size = 0;
        int bitmask_size = sizeof(int) * 8;
        if ((1 & size) != 0)
            iteration_size++;
        for (int i = 1; i < bitmask_size; i++) {
            if (((1 << i) & size) != 0)
                iteration_size += i;
```

```
}

int count_curr_splited_array = 0;
int curr_last_rank = size - 1;
int curr_split = 1;
int current_array_count = 0;
for (int i = 0; i < iteration_size; i++) {
    if (rank % curr_split == 0) {
        current_array_count = 0;
        for (int k = 0; k < static_cast<int>(sendcounts.size()); k++) {
            if (sendcounts[k] != 0)
                current_array_count++;
        }
        if (((rank / curr_split) % 2 == 0) && (rank == curr_last_rank)
            && (current_array_count % 2 != 0)) {
            count_curr_splited_array = 0;
            for (int j = 0; j < size; j += (curr_split * 2)) {
                if (j + curr_split < size) {
                    sendcounts[j] += sendcounts[j + curr_split];
                    sendcounts[j + curr_split] = 0;
                    count_curr_splited_array++;

                    if ((j + curr_split * 2) >= size) {
                        if ((count_curr_splited_array - 1) % 2 == 0)
                            curr_last_rank = j;
                        else
                            curr_last_rank = -1;
                    }
                }
                else {
                    if (count_curr_splited_array % 2 == 0)
                        curr_last_rank = j;
                    else
                        curr_last_rank = -1;
                }
            }
            curr_split *= 2;
            continue;
        }
        if ((rank / curr_split) % 2 == 0) {
            MPI_Status status;
            std::vector<int> recv_elem(sendcounts[rank + curr_split]);
            std::vector<int> even_elem;
            std::vector<int> odd_elem;

            MPI_Recv(recv_elem.data(), sendcounts[rank + curr_split],
                MPI_INT, rank + curr_split,
                curr_split, MPI_COMM_TASK, &status);

            even_elem = EvenSplit(local_array, recv_elem);
            odd_elem = OddSplit(local_array, recv_elem);

            local_array = EvenOddSplit(even_elem, odd_elem);
        }
        else {
            MPI_Send(local_array.data(), sendcounts[rank],
                MPI_INT, rank - curr_split, curr_split,
                MPI_COMM_TASK);
        }
    }
    else {
        return local_array;
    }

    count_curr_splited_array = 0;
    for (int j = 0; j < size; j += (curr_split * 2)) {
```

```
                if (j + curr_split < size) {
                    sendcounts[j] += sendcounts[j + curr_split];
                    sendcounts[j + curr_split] = 0;
                    count_curr_splited_array++;

                    if ((j + curr_split * 2) >= size) {
                        if ((count_curr_splited_array - 1) % 2 == 0)
                            curr_last_rank = j;
                        else
                            curr_last_rank = -1;
                    }
                }
                else {
                    if (count_curr_splited_array % 2 == 0)
                        curr_last_rank = j;
                    else
                        curr_last_rank = -1;
                }
            }
            curr_split *= 2;
        }
    }
    return local_array;
}
```

main.cpp

```
// Copyright 2019 Andronov Maxim
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <vector>
#include <algorithm>
#include <iostream>
#include
"../../../modules/task_3/andronov_m_radix_with_batcher_split/radix_with_batcher_split.h"

TEST(Radix_With_Batcher_Split, throw_radix_sort_with_0_elements) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        std::vector<int> vec;
        ASSERT_ANY_THROW(RadixSort(vec));
    }
}

TEST(Radix_With_Batcher_Split, radix_sort_with_even_number_elements) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        std::vector<int> vec, exp_vec;
        vec = GetRandomVector(30);
        exp_vec = vec;
        std::sort(exp_vec.begin(), exp_vec.end());
        vec = RadixSort(vec);

        EXPECT_EQ(exp_vec, vec);
    }
}

TEST(Radix_With_Batcher_Split, radix_sort_with_odd_number_elements) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
```

```
        std::vector<int> vec, exp_vec;
        vec = GetRandomVector(35);
        exp_vec = vec;
        std::sort(exp_vec.begin(), exp_vec.end());
        vec = RadixSort(vec);

        EXPECT_EQ(exp_vec, vec);
    }
}

TEST(Radix_With_Batcher_Split, sort_with_even_number_elements) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<int> array;
    const int size = 30;

    if (rank == 0) {
        array = GetRandomVector(size);
    }

    std::vector<int> result_array = ParallelRadixSortBatcherSplit(array, size);

    if (rank == 0) {
        sort(array.begin(), array.end());
        EXPECT_EQ(array, result_array);
    }
}

TEST(Radix_With_Batcher_Split, sort_with_odd_number_elements) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<int> array;
    const int size = 35;

    if (rank == 0) {
        array = GetRandomVector(size);
    }

    std::vector<int> result_array = ParallelRadixSortBatcherSplit(array, size);

    if (rank == 0) {
        sort(array.begin(), array.end());
        EXPECT_EQ(array, result_array);
    }
}

TEST(Radix_With_Batcher_Split, performance_test) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    std::vector<int> array;
    const int size = 1000000;
    double start_parall, end_parall, time_parall;

    if (rank == 0) {
        array = GetRandomVector(size);
    }

    start_parall = MPI_Wtime();
    std::vector<int> parall_result_array = ParallelRadixSortBatcherSplit(array, size);
    end_parall = MPI_Wtime();
    time_parall = end_parall - start_parall;

    if (rank == 0) {
        double start_seq, end_seq, time_seq;
        start_seq = MPI_Wtime();
        std::vector<int> seq_result_array = RadixSort(array);
```

```
        end_seq = MPI_Wtime();
        time_seq = end_seq - start_seq;

        std::cout << "Parallel result: " << time_parall << std::endl;
        std::cout << "Sequential result: " << time_seq << std::endl;
    }
}

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();
}
```