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 вещественных чисел с простым слиянием»

Выполнил:
студент группы 381706-1
Нечаева Е.В

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

Нижний Новгород
2020.

Содержание

Введение

Алгоритм сортировки — это алгоритм для упорядочивания элементов в массиве.

В данной лабораторной работе нужно реализовать алгоритм поразрядной сортировки, рассмотрим подробнее, что же это.

Сначала рассмотрим сортировку подсчетом, дальше это пригодится.

Сортировка подсчётом - алгоритм сортировки, в котором используется диапазон чисел сортируемого массива (списка) для подсчёта совпадающих элементов.

Описание:

Исходная последовательность чисел длины n, а в конце отсортированная, хранится в массиве A. Также используется вспомогательный массив C с индексами от 0 до k−1, изначально заполняемый нулями.

- Последовательно пройдём по массиву A и запишем в C[i] количество чисел, равных i.
- Присвоить C[i] значение, равное сумме всех элементов до данного: C[i] = C[0]+C[1]+...C[i-1]. Эта сумма является количеством чисел исходного массива, меньших i.
- Для каждого числа A[i] мы знаем, сколько чисел меньше него - это значение хранится в C[A[i]]. Таким образом, нам известно окончательное место числа в упорядоченном массиве: если есть K чисел меньше данного, то оно должно стоять на позиции K+1. Осуществляем проход по массиву A слева направо, одновременно заполняя результирующий массив, который в конце будет отсортирован.

Алгоритм поразрядной сортировки предполагает поразрядное сравнение: сначала сравниваются значения одного крайнего разряда, и элементы группируются по результатам этого сравнения, затем сравниваются значения следующего разряда, соседнего, и элементы либо упорядочиваются по результатам сравнения значений этого разряда внутри образованных на предыдущем проходе групп, либо переупорядочиваются в целом, но сохраняя относительный порядок, достигнутый при предыдущей сортировке. Затем аналогично делается для следующего разряда, и так до конца.

Постановка задачи

Требуется реализовать программу, в которой выполняется параллельная сортировка вещественных чисел типа double с простым слиянием. Для этого необходимо реализовать следующие операции:

- Автоматическая генерация массива вещественных чисел
- Сортировка подсчетом для каждого байта числа
- Отдельная реализация сортировки для старшего байта числа
- Последовательная сортировка массива
- Параллельная сортировка массива
- Проверка на корректность сортировки
- Алгоритм слияния частей массива при распараллеливании

Метод решения

Рассмотрим кратко, как представляются числа типа double в памяти ЭВМ.

В соответствии со стандартом IEEE 754 для хранения числа отводится три поля: знак, смещенный порядок и мантисса.

Знак числа кодируется одним битом: если <знак числа> = 0, то число положительное, иначе – отрицательное. Мантисса представляет собой последовательность бит, которая задает значащие цифры числа. Порядок задает положение десятичного разделителя в числе.

В типе данных double для представления порядка используется 11 бит, для представления мантиссы – 52 бита.

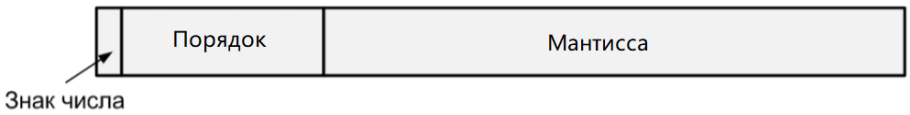

Рисунок 1 Хранение вещественного числа

Таким образом, для положительных чисел, большему байту соответствует большее число. В случае отрицательных чисел, первый бит равен 1, тогда большему байту соответствует меньшее число. Поэтому алгоритм сортировки должен предусматривать отдельный проход по старшим байтам чисел.

Рассмотрим теперь сам алгоритм поразрядной сортировки чисел:

Каждый байт может принимать в общем случае 256 значений. Начнем сортировку чисел от младшего байта к старшему.

- Производится сортировка подсчетом, описанная во введении, последнего байта каждого числа в исходном массиве sourse и сохранение результирующей последовательности чисел в другой массив dest.
- Теперь в качестве исходного массива используется dest, и выполняется сортировка предпоследнего байта числа, и также производится сохранение результата в массив sourse.
- Данные действия повторяются, меняя исходный массив после каждой байтовой сортировки до первого байта числа.
- Так как числа могут быть как положительными, так и отрицательными, для первого байта должен быть свой алгоритм сортировки. Отрицательные числа расположены в обратном порядке, поэтому при вычислении их расположения необходимо идти с конца C[255]=0, соответствующего наименьшему

отрицательному числу до C[128]. При этом в массив нужно записывать сумму всех чисел после текущего, включая само текущее. В случае C от [0] до [127] производится та же сортировка подсчета, что и для положительных чисел.

- Таким образом, в конце прохода по всем байтам числа, мы получим отсортированный массив чисел.

Схема распараллеливания

Так как поразрядная сортировка выполняется для каждого числа последовательно, можно ускорить данный алгоритм, распараллелив программу, сортируя кусочки массива по отдельности, а потом выполнить их слияние.

Тогда общий алгоритм распараллеливания выглядит так:

1. Сортировка частей массива.
2. Слияние отсортированных частей массива.

Идея простого слияния заключается в том, что один процесс может выполнять слияние двух отсортированных массивов по классическому алгоритму. В этом случае слияние n массивов могут выполнять n/2 параллельных процессов. На следующем шаге слияние n/2 полученных массивов будут выполнять n/4 процессов и т.д. Таким образом, последнее слияние будет выполнять один процесс.

Рисунок 2 Простое слияние

Таким образом, каждый процесс получает size/k элементов, где size- размер массива, который необходимо отсортировать, k- кол-во процессов, причем, если кол-во процессов не кратно числу элементов, то первый процесс получает дополнительно число, равное остатку от деления size/k. После чего, каждый процесс сортирует свою часть массива, а дальше происходит простое слияние процессов и получается итоговый отсортированный массив.

Описание программной реализации

Рассмотрим подробнее, какие функции были разработаны в данной программе:

- RazrSort (std::vector<double> src, std::vector<double> dest, int byte, int size) - сортировка каждого байта числа, возвращает сортированный по выбранному байту массив. Сортировка выполняется подсчетом, описанным выше.
- RazrSortLast (std::vector<double> src, std::vector<double> dest, int byte, int size) – сортировка первого байта числа, возвращает отсортированный массив по первому байту.
- DoubleSortWin (std::vector<double> src, int size) – сортировка массива, возвращает полностью отсортированный массив. Идет сортировка от 7 до 1 байт числа, каждый раз свапая два массива. На последнем шаге для 0 байта выполняется своя сортировка.
- Rand (std::vector<double> mas, int size) – формирование автоматического массива чисел, возвращает сформированный массив.
- Tru (std::vector<double> mas, int size) – проверяет на корректность сортировки, возвращает истину или ложь.
- ParallSort (std::vector<double> src, int size) – параллельная реализация поразрядной сортировки, также возвращает отсортированный массив.
- Merge (std::vector<double> mas1, std::vector<double> mas2, int size1, int size2) – выполняет слияние двух отсортированных массивов.

Подтверждение корректности

Для подтверждения корректности в программе были реализованы тесты, разработанные с помощью использования Google C++ Testing Framework.

В данных тестах проверяется:

- Корректность на ввод данных, то есть размер массива должен быть положительным.
- Сортировка массива, созданного с помощью автоматической генерации чисел.
- Сортировка положительных чисел.
- Сортировка отрицательных чисел.
- Сортировка и отрицательных и положительных чисел.

Так как все тесты работают и показывают ожидаемый результат, программа работает корректно.

Результаты экспериментов

Эксперименты проводились на ПК с такими параметрами:

- Процессор: Intel Core i3-8130U, 2.20 GHz
- ОЗУ 6 ГБ
- Версия VS: 2017

Количество элементов равно 10 000 000.

Кол-во процессов	Последовательный алгоритм	Параллельный алгоритм	Ускорение
1	1.16514	1.223	0.952
2	1.0652	0.77763	1.369
3	1.08165	0.735653	1.47
4	1.09235	0.703683	1.552
5	1.09462	0.731241	1.496

Проведя эксперименты, можно увидеть, что при некотором увеличении числа процессов, а именно до 4, наблюдается ускорение алгоритма, но из-за увеличения накладных расходов, это ускорение невелико.

Также при увеличении количества процессов, а именно от 5, наблюдается ухудшение ускорения. Это связано, как было сказано раннее, с увеличением накладных расходов, и с параметрами ПК, а именно с их ограниченностью.

Таким образом, показано, что параллельная программа была выполнена верно.

Заключение

В ходе лабораторной работы были изучены основные формулировки алгоритмов сортировки подсчетом и поразрядной сортировки. Был изучен способ эффективной реализации алгоритма, то есть параллельный.

Были проведены эксперименты, которые показали, что параллельный алгоритм сортировки работает быстрее, чем последовательным, таким образом, можно утверждать, что поставленная задача была выполнена.

Также были написаны методы, с помощью которых возможна сортировка массивов, как последовательно, так и параллельно, и тесты, которые проверяют их корректность.

Литература

1. Поразрядная сортировка целых чисел с плавающей точкой [Электронный ресурс] // URL:

   http://algolist.manual.ru/sort/radix_sort.php

2. Сортировка подсчетом [Электронный ресурс] // URL:

   https://neerc.ifmo.ru/wiki/index.php?title=Сортировка_подсчетом

3. Поразрядная сортировка [Электронный ресурс] // URL:

   https://ru.wikipedia.org/wiki/Поразрядная_сортировка

Приложение

razr_sort.h

```
// Copyright 2019 Nechaeva Ekaterina
#ifndef MODULES_TASK_3_NECHAEVA_E_RAZR_SORT_RAZR_SORT_H_
#define MODULES_TASK_3_NECHAEVA_E_RAZR_SORT_RAZR_SORT_H_

#include <string>
#include <vector>
#include <iostream>
#include <cstdlib>

std::vector<double> RazrSort(std::vector<double> src, std::vector<double> dest, int byte,
int size);
std::vector<double> RazrSortLast(std::vector<double> src, std::vector<double> dest, int
byte, int size);
std::vector<double> DoubleSortWin(std::vector<double> src, int size);
std::vector<double> DoubleSortLin(std::vector<double> src, int size);
std::vector<double> Rand(std::vector<double> mas, int size);
bool Tru(std::vector<double> mas, int size);
std::vector<double> ParallSort(std::vector<double> src, int size);
std::vector<double> Merge(std::vector<double> mas1, std::vector<double> mas2, int size1,
int size2);


#endif  // MODULES_TASK_3_NECHAEVA_E_RAZR_SORT_RAZR_SORT_H_
```

main.cpp

```
// Copyright 2019 Nechaeva Ekaterina
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <algorithm>
#include <random>
#include <cassert>
#include <vector>
#include "../../../modules/task_3/nechaeva_e_razr_sort/razr_sort.h"

TEST(Razr_Sort, sort_random_vec) {
    int nrank;
    MPI_Comm_rank(MPI_COMM_WORLD, &nrank);
    std::vector<double> mas(10000000);
    if (nrank == 0) {
        Rand(mas, 10000000);
        double t1 = MPI_Wtime();
        mas = DoubleSortWin(mas, 10000000);
        t1 = MPI_Wtime() - t1;
        std::cout << t1 << "\n";
    }
    double t = MPI_Wtime();
    mas = ParallSort(mas, 10000000);
    t = MPI_Wtime() - t;
    if (nrank == 0) {
        std::cout << t;
        ASSERT_TRUE(Tru(mas, 10000000) == true);
    }
}

TEST(Razr_Sort, sort_pol_vec) {
    int nrank;
    MPI_Comm_rank(MPI_COMM_WORLD, &nrank);
    std::vector<double> mas = { 3.7, 30.1, 11.5, 1.6 };
    std::vector<double> sort = { 1.6, 3.7, 11.5, 30.1 };
    mas = ParallSort(mas, mas.size());
```

```
    if (nrank == 0) {
        ASSERT_EQ(mas, sort);
    }
}

TEST(Razr_Sort, error_size) {
    std::vector<double> mas = { 3.7, -30.1, -1.5, 1.6 };
    std::vector<double> sort = { -30.1 , -1.5, 1.6, 3.7 };
    ASSERT_ANY_THROW(ParallSort(mas, 0));
}

TEST(Razr_Sort, sort_neg_vec) {
    int nrank;
    MPI_Comm_rank(MPI_COMM_WORLD, &nrank);
    std::vector<double> mas = { -30.7, -300.1, -1.7, -25.6 };
    std::vector<double> sort = { -300.1, -30.7, -25.6, -1.7 };
    mas = ParallSort(mas, mas.size());
    if (nrank == 0) {
        ASSERT_EQ(mas, sort);
    }
}

TEST(Razr_Sort, sort_neg_and_pol_vec) {
    int nrank;
    MPI_Comm_rank(MPI_COMM_WORLD, &nrank);
    std::vector<double> mas = { 3.7, -30.1, -10.5, 1.6 };
    std::vector<double> sort = { -30.1 , -10.5, 1.6, 3.7 };
    mas = ParallSort(mas, mas.size());
    if (nrank == 0) {
        ASSERT_EQ(mas, sort);
    }
}



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();

     }
```

razr_sort.cpp

```
// Copyright 2019 Nechaeva Ekaterina
#include <mpi.h>
#include <random>
#include <utility>
#include <vector>
#include "../../../modules/task_3/nechaeva_e_razr_sort/razr_sort.h"

std::vector<double> RazrSort(std::vector<double> src, std::vector<double> dest, int byte,
int size) {
    unsigned char *mas = (unsigned char *)src.data();
    int counter[256] = { 0 };
    int tek = 0;
```

```
    for (int i = 0; i < size; i++)
        counter[mas[8 * i + byte]]++;

    for (int j = 0; j < 256; j++) {
        int b = counter[j];
        counter[j] = tek;
        tek += b;
    }

    for (int i = 0; i < size; i++) {
        dest[counter[mas[8 * i + byte]]] = src[i];
        counter[mas[8 * i + byte]]++;
    }
    return dest;
}

std::vector<double> RazrSortLast(std::vector<double> src, std::vector<double> dest, int
byte, int size) {
    unsigned char *mas = (unsigned char *)src.data();
    int counter[256] = { 0 };
    int tek = 0;

    for (int i = 0; i < size; i++)
        counter[mas[8 * i + byte]]++;

    for (int j = 255; j >= 128; j--) {
        counter[j] += tek;
        tek = counter[j];
    }
    for (int j = 0; j < 128; j++) {
        int b = counter[j];
        counter[j] = tek;
        tek += b;
    }
    for (int i = 0; i < size; i++) {
        if (mas[8 * i + byte] < 128) {
            dest[counter[mas[8 * i + byte]]] = src[i];
            counter[mas[8 * i + byte]]++;
        }
        else {
            counter[mas[8 * i + byte]]--;
            dest[counter[mas[8 * i + byte]]] = src[i];
        }
    }
    return dest;
}

std::vector<double> DoubleSortWin(std::vector<double> src, int size) {
    std::vector<double> dest(size);

    for (int i = 0; i < 7; i++) {
        dest = RazrSort(src, dest, i, size);
        std::swap(src, dest);
    }
    dest = RazrSortLast(src, dest, 7, size);
    std::swap(src, dest);
    return src;
}

std::vector<double> DoubleSortLin(std::vector<double> src, int size) {
    std::vector<double> dest(size);

    for (int i = 7; i > 0; i--) {
```

```
        RazrSort(src, dest, i, size);
        std::swap(src, dest);
    }
    RazrSort(dest, src, 0, size);
    return dest;
}

std::vector<double> Rand(std::vector<double> mas, int size) {
    std::mt19937 generator;
    std::random_device device;
    generator.seed(device());
    std::uniform_real_distribution<double> distribution(10, 100);

    for (int i = 0; i < size; i++) {
        double f = distribution(generator);
        mas[i] = f;
    }
    return mas;
}

bool Tru(std::vector<double> mas, int size) {
    int a = 0;
    for (int i = 0; i < size - 1; i++) {
        if (mas[i] > mas[i + 1])
            a = 1;
        if (a == 1)
            return false;
    }
    return true;
}

std::vector<double> Merge(std::vector<double> mas1, std::vector<double> mas2, int size1,
int size2) {
    std::vector<double> temp(size1 + size2);
    int i = 0, j = 0;
    for (int k = 0; k < size1 + size2; k++) {
        if (i > size1 - 1) {
            double a = mas2[j];
            temp[k] = a;
            j++;
        }
        else if (j > size2 - 1) {
            double a = mas1[i];
            temp[k] = a;
            i++;
        }
        else if (mas1[i] < mas2[j]) {
            double a = mas1[i];
            temp[k] = a;
            i++;
        }
        else {
            double b = mas2[j];
            temp[k] = b;
            j++;
        }
    }
    return temp;
}

std::vector<double> ParallSort(std::vector<double> src, int size) {
    int nsize, nrank;
    MPI_Comm_size(MPI_COMM_WORLD, &nsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &nrank);
```

```
    if (size <= 0)
        throw - 1;
    int n = size / nsize;
    int p = size % nsize;
    std::vector<double> dest;

    if (n == 0) {
        dest.resize(size);
        copy(src.begin(), src.end(), dest.begin());
        dest = DoubleSortWin(dest, n + p);
        return dest;
    }
    else {
        if (nrank == 0) {
            dest.resize(n + p);
        }
        else {
            dest.resize(n);
        }
        MPI_Scatter(src.data(), n, MPI_DOUBLE, dest.data(), n, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
        if (nrank == 0) {
            int a = size - p;
            for (int i = n; i < n + p; ++i)
                dest[i] = src[a++];
            dest = DoubleSortWin(dest, n + p);
        }
        else {
            dest = DoubleSortWin(dest, n);
        }

        int locsize = nsize;
        int i = 0;
        int patner;
        while (locsize > 1) {
            if (locsize % 2 == 0) {
                if (nrank % static_cast<int>(pow(2.0, i + 1)) != 0) {
                    patner = nrank - static_cast<int>(pow(2.0, i));
                    MPI_Send(dest.data(), n*static_cast<int>(pow(2.0, i)),
                        MPI_DOUBLE, patner, 0, MPI_COMM_WORLD);
                    return std::vector<double>(0);
                }
                else {
                    std::vector<double> temp(n*static_cast<int>(pow(2.0, i)));
                    patner = nrank + static_cast<int>(pow(2.0, i));
                    MPI_Status status;
                    MPI_Recv(temp.data(), n*static_cast<int>(pow(2.0, i)), MPI_DOUBLE,
patner, 0, MPI_COMM_WORLD, &status);
                    if (nrank == 0) {
                        dest = Merge(dest, temp, dest.size(), n*static_cast<int>(pow(2.0,
i)));
                    }
                    else {
                        dest = Merge(dest, temp, n*static_cast<int>(pow(2.0, i)),
n*static_cast<int>(pow(2.0, i)));
                    }
                }
            }
            else {
                if ((nrank % static_cast<int>(pow(2.0, i + 1)) != 0) && (nrank !=
(locsize - 1)*static_cast<int>(pow(2.0, i)))) {
                    patner = nrank - static_cast<int>(pow(2.0, i));
                    MPI_Send(dest.data(), n*static_cast<int>(pow(2.0, i)),
```

```
                        MPI_DOUBLE, patner, 0, MPI_COMM_WORLD);
                    return std::vector<double>(0);
                }
                else if (nrank == (locsize - 1)*static_cast<int>(pow(2.0, i))) {
                    MPI_Send(dest.data(), n*static_cast<int>(pow(2.0, i)),
                        MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
                    return std::vector<double>(0);
                }
                else {
                    if (nrank == 0) {
                        std::vector<double> temp(n*static_cast<int>(pow(2.0, i)));
                        patner = (locsize - 1)*static_cast<int>(pow(2.0, i));
                        MPI_Status status;
                        MPI_Recv(temp.data(), n*static_cast<int>(pow(2.0, i)),
MPI_DOUBLE, patner, 0, MPI_COMM_WORLD, &status);
                        dest = Merge(dest, temp, dest.size(), n*static_cast<int>(pow(2.0,
i)));
                    }
                    std::vector<double> temp(n*static_cast<int>(pow(2.0, i)));
                    patner = nrank + static_cast<int>(pow(2.0, i));
                    MPI_Status status;
                    MPI_Recv(temp.data(), n*static_cast<int>(pow(2.0, i)), MPI_DOUBLE,
patner, 0, MPI_COMM_WORLD, &status);
                    if (nrank == 0) {
                        dest = Merge(dest, temp, dest.size(), n*static_cast<int>(pow(2.0,
i)));
                    }
                    else {
                        dest = Merge(dest, temp, n*static_cast<int>(pow(2.0, i)),
n*static_cast<int>(pow(2.0, i)));
                    }
                }
            }
            locsize = locsize / 2;
            i++;
        }
    }
    return dest;
     }
```