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 простым слиянием»

Выполнил:
студент группы 381703-3
Лугин Михаил Дмитриевич
Проверил:
Доцент кафедры МОСТ,
кандидат технических наук

Сысоев А. В.

Нижний Новгород
2019

Содержание

1. Введение

С помощью параллельного программирования можно ускорить работу многих алгоритмов. Алгоритм Шелла сортировки массива входит в данное число.

Сортировка Шелла – алгоритм сортировки, идея которого состоит в сравнении элементов, стоящих на определённом расстоянии друг от друга. При данной сортировке сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии d, выбор которого зависит от версии алгоритма (в изначальном варианте сортировки Шелла оно равно n / 2, где n – число элементов в сортируемом массиве). После этого процедура сравнения повторяется для меньших значений d (опять же, первоначально размер d уменьшался вдвое на каждой итерации) и завершается тогда, когда d становится равным единице. Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы “быстрее” встают на свои места.

Для реализации параллельного алгоритма сортировки Шелла понадобится алгоритм слияния двух массивов. В данной лабораторной работе будет использовано обычное слияние.

Итак, параллельный алгоритм сортировки Шелла с обычным слиянием можно разделить на три этапа:

1. Разделение элементов массива на группы
2. Сортировка полученных групп
3. Слияние отсортированных групп

Более подробно о каждом этапе мы поговорим позднее.

* В лабораторной работе рассматривается реализация шаблонной функции сортировки с пользовательским оператором сравнения элементов.

2. Постановка задачи

В рамках лабораторной работы нужно реализовать параллельный алгоритм сортировки Шелла с обычным слиянием.

Необходимо реализовать следующие компоненты:

- Линейный алгоритм сортировки Шелла
- Алгоритм слияния двух массивов
- Параллельный алгоритм сортировки Шелла
- Вспомогательные функции для проверки работоспособности и эффективности
- Поддержка шаблонности алгоритма сортировки *
- Поддержка пользовательского оператора сравнения*

3. Метод решения

Программа состоит из модулей lugin_m_shell_sort_mpi и lugin_m_shell_sort _mpi_lib. Первый включает в себя файлы shell_sort.h, в котором описаны и реализованы шаблонные методы, использующиеся в программе, и main.cpp, который содержит набор тестов для проверки работоспособности написанной программы.

Метод линейной сортировки Шелла принимает итератор, указывающий на первый контейнер, итератор, указывающий на последний контейнер, оператор сравнения. Возвращает отсортированную структуру данных. Внутри метода каноничная реализация сортировки Шелла на c++.

Метод слияния имеет специфическую реализацию в угоду эффективности программы. На вход принимает ссылку на вектор с отсортированными элементами, массив с отсортированными элементами, размер массива, оператор сравнения. Возвращает отсортированный вектор с элементами из входных данных.

Метод параллельной сортировки Шелла IShellSort принимает вектор с произвольным типом*, оператор сравнения. Корневой процесс возвращает отсортированный вектор. Подробнее о данном методе поговорим в сл. пункте.

Из-за различия типов данных в языке c++ и типов данных в MPI, был реализован метод принимающий на вход произвольный элемент любого типа* c++ и возвращает похожий тип данных MPI.

* int, char, float, double.

4. Схема распараллеливания

Распараллеливание происходит в методе IshellSort. Все процессы коммуникатора MPI_COMM_WORLD получают на вход неотсортированный вектор. В каждом отдельном процессе создается свой буфер. В целях соответствия идее алгоритма сортировки Шелла, распределение элементов начального вектора по процессам выполняется выборкой через шаг равный количеству процессов в MPI_COMM_WORLD. Далее на каждом процессе вызывается метод линейной сортировки Шелла. Отсортированные векторы сливаются попарно с шагом 2^{n-1}, где n – итерация цикла слияния.

Рассмотрим пример слияния с 9 процессами:

Итерация 1: $\dot{0}\leftarrow 1, \dot{2}\leftarrow 3, \dot{4}\leftarrow 5, \dot{6}\leftarrow 7, \dot{8}$

Итерация 2: $\dot{0}\leftarrow 2, \dot{4}\leftarrow 6, \dot{8}$

Итерация 3: $\dot{0}\leftarrow 4, \dot{8}$

Итерация 4: $\dot{0}\leftarrow 8$

Условием остановки алгоритма слияния будет procNum < 2^{n-1}, где procNum – число процессов в MPI_COMM_WORLD.

Передачу/прием данных между процессами обеспечивает пара функций MPI_Send и MPI_Recv. В общем случаем не понятно сколько элементов надо принять процессу-получателю, поэтому исользуются функции MPI_Probe и MPI_Get_Count. Первая получает информацию о сообщении, а вторая количество элементов в сообщении.

5. Описание программной реализации

Методы, участвующие в параллельном алгоритме сортировки Шелла:

- MPI_Get_Datatype(…) – переводит тип данных c++ в тип данных MPI
- ShellSort(…) – представляет линейный алгоритм сортировки Шелла
- Merge(…) – производит слияние вектора и массива

Методы, генерирующие векторы для тестов:

- std::vector<T> getSortedRandomArrayBottomUp(T type, int size, int _percentSkipped, int _percentRepeated)
  type – переменная, определяющая тип возвращаемого вектора
  size – размер возвращаемого вектора
  _percentSkipped – значение в % шанса перескачить число в последовательности
  _percentRepeated – значение в % шанса повторить число в последовательности
  Данная функция генерирует рандомный вектор, элементы которого расположены по возрастанию от 0.
- std::vector<T> getSortedRandomArrayTopDown(T type, int size, int _percentSkipped, int _percentRepeated)
  type – переменная, определяющая тип возвращаемого вектора
  size – размер возвращаемого вектора
  _percentSkipped – значение в % шанса перескачить число в последовательности
  _percentRepeated – значение в % шанса повторить число в последовательности
  Данная функция генерирует рандомный вектор, элементы которого расположены по убыванию до 0.
- std::vector<T> RandomizeArray(std::vector<T> array, int iterations)
  array – отсортированный вектор
  iterations – количество итераций запутывания

6. Подтверждение корректности

Для подтверждения корректности в программе было реализованно 7 тестов.

Тест, проверяющий поведение программы в специальных ситуациях:

- TEST(shell_sort, return_empty_array_when_empty_array) – проверяет, нет ли ошибок, если в метод IshellSort(…) передают пустой вектор.

Тесты, проверяющие работоспособность алгоритма сортировки:

- TEST(shell_sort, sort_bottom_up_random_int_array)
- TEST(shell_sort, sort_bottom_up_random_float_array)
- TEST(shell_sort, sort_bottom_up_random_double_array)
- TEST(shell_sort, sort_top_down_random_int_array)
- TEST(shell_sort, sort_top_down_random_float_array)
- TEST(shell_sort, sort_top_down_random_double_array)

Данные тесты проверяют всевозможные комбинации входящих параметров в метод IshellSort(…). Тип входных данных, количество входных данных, запутанность элементов, сортировка по возрастанию/убыванию.

7. Результаты экспериментов

Характеристики компьютера:

- Intel® Core™ i5-6440HQ CPU @ 2.60 GHz (4 ядра)
- 8.00 GB RAM
- ОС Windows 10

Было проведено 12 экспериментов с разным количеством процессов и данных

Количество элементов	1 процесс	2 процесса	3 процесса	4 процесса
100000	3.67213 с	2.05103 с	1.6725 с	1.44883 с
500000	28.3949 с	17.4589 с	15.5429 с	13.4207 с
1000000	65.1756 с	51.9614 с	47.4694 с	40.552 с

По результатам экспериментов видно, что при при увеличении количества процессов, алгоритм работает в разы быстрее, а значит работает корректно и эффективно.

8. Заключение

Реализованный параллельный алгоритм сортировки Шелла не идеален, но работает корректно. Так же в приложении можно найти код программы и посмотреть реализацию на c++. В дополнение была реализована поддержка шаблонности и пользовательского оператора сравнения элементов. Чтобы отсортировать вектор со своим типом данных, нужно добавить обработку перевода типа c++ в тип MPI в методе MPI_Get_Datatype(…).

9. Список литературы

- Параллельные вычисления (базовый курс)

10. Приложение

```
/*
 * Copyright (C) 2019 LOOGIN. All Rights Reserved.
 * Available types: char(Int8), int(Int32), float(Float32) and double(Float64).
 */
#ifndef MODULES_TASK_3_LUGIN_M_SHELL_SORT_SHELLSORT_H_
#define MODULES_TASK_3_LUGIN_M_SHELL_SORT_SHELLSORT_H_

#include <mpi.h>
#include <vector>
#include <cstdlib>
#include <string>
#include <ctime>
#include <utility>
#include <random>
#include <typeinfo>

template <class T>
std::string OutPutVector(std::vector<T> v) {
    std::string vec;
    vec = '{';
    for (const auto &curV : v) {
        vec += std::to_string(curV);
        vec += ',';
    }
    vec += '}';
    return vec;
}

template <class T>
MPI_Datatype MPI_Get_Datatype(T type) {
    if (typeid(type).name() == typeid(static_cast<double>(1.)).name()) {
        return MPI_DOUBLE;
    }
    if (typeid(type).name() == typeid(static_cast<float>(1.)).name()) {
        return MPI_FLOAT;
    }
    if (typeid(type).name() == typeid(static_cast<char>('1')).name()) {
        return MPI_CHAR;
    }
    if (typeid(type).name() == typeid(static_cast<int>(1)).name()) {
        return MPI_INT;
    }
    throw "Invalid Datatype";
}

template <typename Iterator, typename Compare>
void ShellSort(Iterator first, Iterator last, Compare comp) {
    for (typename std::iterator_traits<Iterator>::difference_type d = (last - first)
/ 2; d != 0; d /= 2)
        for (Iterator i = first + d; i != last; ++i)
            for (Iterator j = i; j - first >= d && comp(*j, *(j - d)); j -= d)
                std::swap(*j, *(j - d));
}

template <typename T, typename Compare>
std::vector<T> Merge(const std::vector<T>& mainVec, T massive[], int size, Compare
comp) {
    int i = 0;
    std::vector<T> mergedVect(mainVec.begin(), mainVec.end());
    auto IT = mergedVect.begin();
    for (int pos = 0; (IT + pos) != mergedVect.end() && i < size; pos++)
        if (comp(massive[i], *(IT + pos)) ||
```

```
            (!comp(massive[i], *(IT + pos)) &&
            !comp(*(IT + pos), massive[i]))) {
            mergedVect.insert((IT + pos), massive[i++]);
            IT = mergedVect.begin();
        }

    while (i < size)
        mergedVect.push_back(massive[i++]);

    return mergedVect;
}

template <class T, typename Compare>
std::vector<T> IShellSort(std::vector<T> massive, Compare comp) {
    int sizeOfMassive = massive.size();
    if (sizeOfMassive < 2)
        return massive;

    MPI_Datatype curDatatype = MPI_Get_Datatype(massive[0]);
    int procNum, procRank;

    MPI_Comm_size(MPI_COMM_WORLD, &procNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);

    std::vector<T> buffer;

    if (procRank < sizeOfMassive) {
        for (int i = procRank; i < sizeOfMassive; i += procNum) {
            buffer.push_back(massive[i]);
        }

        ShellSort(buffer.begin(), buffer.end(), comp);
    } else {
        return massive;
    }

    if (procNum == 1)
        return buffer;

    int mergeRange = 1;

    do {
        mergeRange *= 2;

        if (procRank % mergeRange == mergeRange / 2 && buffer.size()) {
            MPI_Ssend(&buffer[0], buffer.size(), curDatatype, procRank - mergeRange /
2, 0, MPI_COMM_WORLD);
            return buffer;
        }
        if (procRank % mergeRange == 0 && (procRank + mergeRange / 2) < procNum &&
buffer.size()) {
            int sizeCurBuffer;
            MPI_Status status;

            MPI_Probe(procRank + mergeRange / 2, 0, MPI_COMM_WORLD, &status);
            MPI_Get_count(&status, curDatatype, &sizeCurBuffer);
            T *curBuffer = new T[sizeCurBuffer];

            MPI_Recv(curBuffer, sizeCurBuffer, curDatatype, procRank + mergeRange /
2, 0, MPI_COMM_WORLD, &status);

            buffer = Merge(buffer, curBuffer, sizeCurBuffer, comp);
            delete[] curBuffer;
        }
    } while (mergeRange < procNum);
```

```
        return buffer;
}

template <typename T>
std::vector<T> getSortedRandomArrayBottomUp(T type, int size, int _percentSkipped =
20, int _percentRepeated = 35) {
    int procRank;

    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
    std::mt19937 generator(time(NULL));
    std::uniform_int_distribution<int> percentSkipped(0, _percentSkipped);
    std::uniform_int_distribution<int> percentRepeated(0, _percentRepeated);
    std::vector<T> array;
    if (procRank == 0) {
        if (typeid(type).name() == typeid(static_cast<double>(1.)).name()) {
            array.push_back(0.);
            for (int pos = 0; pos < size - 1; pos++) {
                double curElem = array.back();
                if (percentRepeated(generator)) {
                    array.push_back(curElem);
                    continue;
                }
                while (percentSkipped(generator))
                    curElem++;
                array.push_back(curElem);
            }
        }
        if (typeid(type).name() == typeid(static_cast<float>(1.)).name()) {
            array.push_back(0.);
            for (int pos = 0; pos < size - 1; pos++) {
                float curElem = array.back();
                if (percentRepeated(generator)) {
                    array.push_back(curElem);
                    continue;
                }
                while (percentSkipped(generator))
                    curElem++;
                array.push_back(curElem);
            }
        }
        if (typeid(type).name() == typeid(static_cast<char>('1')).name()) {
            for (char pos = 0; pos < size; pos++) {
                int curCount = pos < size % 256 ? size / 256 + 1 : size / 256;
                while ( curCount-- > 0)
                    array.push_back(pos);
            }
        }
        if (typeid(type).name() == typeid(static_cast<int>(1)).name()) {
            array.push_back(0);
            for (int pos = 0; pos < size - 1; pos++) {
                int curElem = array.back();
                if (percentRepeated(generator)) {
                    array.push_back(curElem);
                    continue;
                }
                while (percentSkipped(generator))
                    curElem++;
                curElem++;
                array.push_back(curElem);
            }
        }
    }
    if (procRank != 0)
        array.resize(size);

    MPI_Bcast(&array[0], size, MPI_Get_Datatype(type), 0, MPI_COMM_WORLD);
```

```
    return array;
}

template <typename T>
std::vector<T> getSortedRandomArrayTopDown(T type, int size, int _percentSkipped =
20, int _percentRepeated = 35) {
    int procRank;

    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
    std::mt19937 generator(time(NULL));
    std::uniform_int_distribution<int> percentSkipped(0, _percentSkipped);
    std::uniform_int_distribution<int> percentRepeated(0, _percentRepeated);
    std::vector<T> array;
    if (procRank == 0) {
        if (typeid(type).name() == typeid(static_cast<double>(1.)).name()) {
            array.push_back(0.);
            for (int pos = 0; pos < size - 1; pos++) {
                double curElem = array.front();
                if (percentRepeated(generator)) {
                    array.insert(array.begin(), curElem);
                    continue;
                }
                while (percentSkipped(generator))
                    curElem++;
                array.insert(array.begin(), curElem);
            }
        }
        if (typeid(type).name() == typeid(static_cast<float>(1.)).name()) {
            array.push_back(0.);
            for (int pos = 0; pos < size - 1; pos++) {
                float curElem = array.front();
                if (percentRepeated(generator)) {
                    array.insert(array.begin(), curElem);
                    continue;
                }
                while (percentSkipped(generator))
                    curElem++;
                array.insert(array.begin(), curElem);
            }
        }
        if (typeid(type).name() == typeid(static_cast<char>('1')).name()) {
            for (char pos = 0; pos < size; pos++) {
                int curCount = pos < size % 256 ? size / 256 + 1 : size / 256;
                while (curCount-- > 0)
                array.insert(array.begin(), pos);
            }
        }
        if (typeid(type).name() == typeid(static_cast<int>(1)).name()) {
            array.push_back(0);
            for (int pos = 0; pos < size - 1; pos++) {
                int curElem = array.front();
                if (percentRepeated(generator)) {
                    array.insert(array.begin(), curElem);
                    continue;
                }
                while (percentSkipped(generator))
                    curElem++;
                array.insert(array.begin(), curElem);
            }
        }
    }

    if (procRank != 0)
        array.resize(size);
```

```
    MPI_Bcast(&array[0], size, MPI_Get_Datatype(type), 0, MPI_COMM_WORLD);

    return array;
}

template <typename T>
std::vector<T> RandomizeArray(std::vector<T> array, int iterations) {
    int procRank;

    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
    std::mt19937 generator(time(NULL));
    std::uniform_int_distribution<int> pos(0, array.size() - 1);
    std::vector<T> randomArray(array.begin(), array.end());
    if (procRank == 0) {
        int size = array.size();
        while (iterations--)
            for (int i = 0; i < size ; i++)
                std::swap(randomArray[i], randomArray[pos(generator)]);
    } else {
        randomArray.resize(array.size());
    }

    MPI_Bcast(&randomArray[0], randomArray.size(), MPI_Get_Datatype(randomArray[0]),
0, MPI_COMM_WORLD);

    return randomArray;
}

#endif  // MODULES_TASK_3_LUGIN_M_SHELL_SORT_SHELLSORT_H_
```