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 чётно-нечётным слиянием Бэтчера»

Выполнил:
студент группы 381706-2
Паузин Л. П.

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

Нижний Новгород
2019

Содержание

Введение

Сортировка является одной из базовых операций обработки данных. Она используется во множестве задач в разных областях (решение систем линейных уравнений, упорядочивание графов, базы данных и др.). При работе с большим количеством элементов наиболее оптимальные методы сортировки работают достаточно долго, а в случае рекурсивности алгоритма возможно переполнения стека вызовов. Уже давно часто приходится работать с большим объемом данных. Мы можем повысить эффективность выполняя эти вычисления параллельно.

Целью настоящей работы является реализация сортировки Шелла с чётно-нечётным слиянием Бэтчера.

Постановка задачи

Для выполнения цели работы были поставлены следующие задачи:

- Реализация последовательного алгоритма сортировки Шелла
- Реализация параллельного алгоритма сортировки Шелла со слиянием Бэтчера
- Проведение вычислительных экспериментов
- Сравнение времени работы полученных алгоритмов

Метод решения

Сортировка Шелла (англ. Shell sort) — алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками. Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга. Иными словами — это сортировка вставками с предварительными «грубыми» проходами. Аналогичный метод усовершенствования пузырьковой сортировки называется сортировка расчёской.

При сортировке Шелла сначала сравниваются и сортируются между собой значения, стоящие один от другого на некотором расстоянии d (о выборе значения d см. ниже). После этого процедура повторяется для некоторых меньших значений d, а завершается сортировка Шелла упорядочиванием элементов при d=1 (то есть обычной сортировкой вставками). Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы «быстрее» встают на свои места (в простых методах сортировки, например, пузырьковой, каждая перестановка двух элементов уменьшает количество инверсий в списке максимум на 1, а при сортировке Шелла это число может быть больше).

Невзирая на то, что сортировка Шелла во многих случаях медленнее, чем быстрая сортировка, она имеет ряд преимуществ:

- отсутствие потребности в памяти под стек;
- отсутствие деградации при неудачных наборах данных — быстрая сортировка легко деградирует до $O(n^2)$, что хуже, чем худшее гарантированное время для сортировки Шелла.

Чётно-нечётное слияние Бэтчера 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. Первый и последний элементы массива проверять не надо, т.к. они являются минимальным и максимальным элементов массивов.

Чётно-нечётное слияние Бэтчера позволяет задействовать 2 потока при слиянии двух упорядоченных массивов. В этом случае слияние n массивов могут выполнять n параллельных потоков. На следующем шаге слияние n/2 полученных массивов будут выполнять n/2 потоков и т.д. На последнем шаге два массива будут сливать 2 потока.

Схема распараллеливания

Сначала мы разделяем исходный массив на количество частей равному количеству процессов. Далее передаем каждому процессу свою часть элементов из исходного массива и сортируем их с помощью сортировки Шелла.

Следом нужно слить все эти массивы с помощью слияния Бэтчера.

На i-ой итерации будут попарно сливаться процессы, для которых выполняется $j\%2^i = 0$ с процессами, для которых $j + 2^{i-1}$, где j- ранг процесса. Для каждого слияния необходимо выполнить следующие действия:

1. Разделить массивы в первом и втором сливаемых процессах на четные и нечетные. Второй процесс передает первому четные элементы, а первый второму нечетные.
2. Первый процесс сливает в один массив четные элементы, а второй нечетные.
3. Второй процесс отправляет первому свой отсортированный массив и первый эти массивы объединяет.
4. Первый процесс проходит по массиву и сравнивает четные и нечетные элементы.

На последней итерации слияния необходимо вместо шага 4 нужно пересортировать массив так чтобы четные и нечетные элементы были на своих местах, а затем так же сделать проход по массиву и сравнить нечетные элементы с четными.

После всех итераций отсортированный массив будет находится в 0 процессе.

Описание программной реализации

Программа содержит 7 методов:

`getRandomVector`– создает вектор из случайных элементов

`ShellSort`– последовательный алгоритм сортировки Шелла

`evenFunc` – функция для сливания четных элементов

`oddFunc` – функция для сливания нечетных элементов.

`ransposition` – проход по массиву и сравнение нечетных и четных элементов.

`permutation` – разделяет массив на четные и нечетные элементы.

`batcherParallel` – основной алгоритм, использующийся для работы программы

Подтверждение корректности

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

`Can_Throw_Assert` – проверка вызова ошибки при вызове функции для вектора размером 0

`Can_Not_Throw_Assert` – проверка корректности работы исключения в алгоритме

`Test_On_Big_Data` – проверка корректности работы алгоритма для вектора большого размера

`Test_On_Simple_Vec` – проверка корректности работы алгоритма для вектора четной длины.

`Test_On_Not_Equal` – проверка корректности работы алгоритма для заранее заданного вектора.

Результаты экспериментов

Эксперименты проводились на ПК со следующими параметрами:

1. Операционная система: Windows 8.1.
2. Процессор: Intel(R) Core™ i5-6500 CPU @ 3.20 GHz.
3. ОЗУ 8 ГБ.
4. Версия Visual Studio: 2017.

Эксперимент проводился на 45 000 000 элементов.

Количество процессов	Время последовательного алгоритма (сек.)	Время параллельного алгоритма (сек.)	Ускорение
1	5.86085	6.24077	0.9391
2	5.74708	3.74694	1.5338
4	5.81337	2.78101	2.0904
8	5.87689	2.88584	2.0365

Таблица 1. Сравнение времени работы программы при параллельном и последовательном алгоритме.

Проведены эксперименты, в ходе которых было доказано, что алгоритм реализован верно и получена действительно эффективно работающая программа. При количестве процессов, больше 4, увеличение ускорения не наблюдается, даже наоборот, заметна небольшая деградация. Это связано как с накладными расходами, так и с ограниченностью параметров процессора, на котором проводились эксперименты.

На основе полученных данных можно сделать вывод, что использование параллельного алгоритмы лучше сортировки Шелла.

Заключение

В ходе работы была реализована сортировка Шелла и параллельная сортировка Шелла с чётно-нечётным слиянием Бэтчера. Вычислительные эксперименты показали, что сортировка Шелла уступает алгоритму с использованием параллельных вычислений в эффективности.

Корректность работы подтверждается с помощью библиотеки модульного тестирования Google C++ Testing Framework.

Литература

Ссылки в Internet:

- Интуит. https://www.intuit.ru/studies/courses/1156/190/info
- Статья на Хабр - https://habr.com/ru/post/261777/
- Википедия. https://ru.wikipedia.org/wiki/Сортировка_Шелла

Приложение

main.cpp

```
// Copyright 2019 Pauzin Leonid
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <stdio.h>
#include <mpi.h>
#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>
#include "./pauzin_l_batcher_shell.h"


TEST(Batcher_Sort, Can_Throw_Assert) {
  std::vector<int> testVec(0);
  const int sizeVec = testVec.size();
  ASSERT_ANY_THROW(batcherParallel(testVec, sizeVec));
}

TEST(Batcher_Sort, Can_Not_Throw_Assert) {
  std::vector<int> testVec(10);
  const int sizeVec = testVec.size();
  ASSERT_NO_THROW(batcherParallel(testVec, sizeVec));
}


TEST(Batcher_Sort, Test_On_Big_Data) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  std::vector<int> testVec;
  const int vecSize = 45000000;
  if (rank == 0) {
    testVec = getRandomVector(vecSize);
  }
  std::vector<int> batchVec = batcherParallel(testVec, vecSize);
  if (rank == 0) {
    testVec = ShellSort(testVec);
    ASSERT_EQ(batchVec, testVec);
  }
}

TEST(Batcher_Sort, Test_On_Simple_Vec) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  const int vecSize = 10;
  std::vector<int> testVec(vecSize);
  if (rank == 0) {
    testVec = { 55, 36, 37, 28, 49, 90, 3, 12, 19, 14 };
  }
  std::vector<int> batchVec = batcherParallel(testVec, vecSize);
  if (rank == 0) {
    testVec = ShellSort(testVec);
    ASSERT_EQ(batchVec, testVec);
  }
}

TEST(Batcher_Sort, Test_On_Not_Equal) {
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
  const int vecSize = 10;
  std::vector<int> testVec(vecSize);
  if (rank == 0) {
    testVec = { 55, 36, 37, 28, 49, 90, 3, 12, 19, 14 };
  }
  std::vector<int> batchVec = batcherParallel(testVec, vecSize);
  if (rank == 0) {
    testVec = ShellSort(testVec);
    testVec[0] = -1;
    ASSERT_NE(batchVec, testVec);
  }
}

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();
}
```

pauzin_l_batcher_shell.h

```
// Copyright 2019 Pauzin Leonid
#ifndef MODULES_TASK_3_PAUZIN_L_BATCHER_SHELL_PAUZIN_L_BATCHER_SHELL_H_
#define MODULES_TASK_3_PAUZIN_L_BATCHER_SHELL_PAUZIN_L_BATCHER_SHELL_H_

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

std::vector<int> getRandomVector(int size);
std::vector <int> ShellSort(const std::vector <int> &vec);
std::vector<int> evenFunc(const std::vector<int>& vec1, const std::vector<int>& vec2);
std::vector<int> oddFunc(const std::vector<int>& vec1, const std::vector<int>& vec2);
std::vector<int> transposition(std::vector<int> vec, int even_size, int odd_size);
std::vector<int> merge(std::vector<int> vec, int even_size, int odd_size);
std::vector<int> permutation(std::vector<int> vec);
std::vector<int> batcherParallel(std::vector<int> global_vec, int size_vec);
#endif  // MODULES_TASK_3_PAUZIN_L_BATCHER_SHELL_PAUZIN_L_BATCHER_SHELL_H_
```

pauzin_l_batcher_shell.cpp

```
// Copyright 2019 Pauzin Leonid
#include <mpi.h>
#include <vector>
#include <random>
#include <ctime>
#include <algorithm>
#include <iostream>
#include <utility>
#include "../../../modules/task_3/pauzin_l_batcher_shell/pauzin_l_batcher_shell.h"



std::vector <int> ShellSort(const std::vector <int> &vec) {
  int step, i, j, tmp;
  int size = vec.size();
  std::vector <int> resulVec(vec);
```

```
  for (step = size / 2; step > 0; step /= 2)
    for (i = step; i < size; i++)
      for (j = i - step; j >= 0 && resulVec[j] > resulVec[j + step]; j -= step) {
        tmp = resulVec[j];
        resulVec[j] = resulVec[j + step];
        resulVec[j + step] = tmp;
      }
  return resulVec;
}

std::vector<int> getRandomVector(int size) {
  std::vector<int> vector(size);
  std::mt19937 gen;
  gen.seed(static_cast<unsigned int>(time(0)));
  for (int i = 0; i < size; i++) {
    vector[i] = gen() % 100;
  }
  return vector;
}

std::vector<int> batcherParallel(std::vector<int> globalVec, int vecSize) {
  int size, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (vecSize <= 0)
    throw "Wrong vector size";

  const int delta = vecSize / size;
  const int remainder = vecSize % size;
  int evenS, oddS, tag, sendLenghtNew;
  std::vector<int> localVec;

  if (vecSize < size) {
    if (rank == 0)
      localVec = ShellSort(globalVec);
    return localVec;
  }

  if (size == 1) {
    localVec = ShellSort(globalVec);
    return localVec;
  }
  if (rank == 0) {
    localVec.resize(delta + remainder);
  } else {
    localVec.resize(delta);
  }

  if (rank == 0) {
    for (int i = 0; i < delta + remainder; i++) {
      localVec[i] = globalVec[i];
    }
    for (int i = 1; i < size; i++)
      MPI_Send(&globalVec[0] + delta*i + remainder, delta, MPI_INT, i, 0, MPI_COMM_WORLD);
  } else {
    MPI_Status status;
    MPI_Recv(&localVec.front(), delta, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  }

  int count = 1;
  for (count = 1; pow(2, count) < size; count++) {}
  localVec = ShellSort(localVec);
  localVec = permutation(localVec);
```

```
  int offset = 1, proc = 2, lengthS, lenghtRcv;
  std::vector<int> promVec;
  std::vector<int> vecOdd;
  std::vector<int> vecEven;
  MPI_Status status;

  for (int i = 0; i < count; i++) {
    if (rank % proc == 0 && rank + offset < size) {
      lengthS = localVec.size() / 2;
      tag = rank + offset;
      MPI_Send(&lengthS, 1, MPI_INT, tag, 0, MPI_COMM_WORLD);
      MPI_Recv(&lenghtRcv, 1, MPI_INT, tag, 0, MPI_COMM_WORLD, &status);

      promVec.resize(lenghtRcv / 2 + lenghtRcv % 2);
      MPI_Send(&localVec[localVec.size() / 2 + localVec.size() % 2], lengthS, MPI_INT, tag, 0,
MPI_COMM_WORLD);
      MPI_Recv(&promVec.front(), lenghtRcv / 2 + lenghtRcv % 2, MPI_INT, tag, 0, MPI_COMM_WORLD, &status);

      vecEven = evenFunc(localVec, promVec);
      vecOdd.resize(lenghtRcv / 2 + localVec.size() / 2);
      evenS = vecEven.size();
      oddS = vecOdd.size();

      MPI_Recv(&vecOdd.front(), lenghtRcv / 2 + localVec.size() / 2, MPI_INT,
        tag, 0, MPI_COMM_WORLD, &status);

      localVec.resize(evenS + oddS);
      std::copy(vecEven.begin(), vecEven.end(), localVec.begin());
      std::copy(vecOdd.begin(), vecOdd.end(), localVec.begin() + evenS);

      if (i + 1 == count)
        localVec = merge(localVec, evenS, oddS);
      else
        localVec = transposition(localVec, evenS, oddS);
    }

    if (rank - offset >= 0 && (rank - offset) % proc == 0) {
      lengthS = localVec.size();
      tag = rank - offset;
      MPI_Send(&lengthS, 1, MPI_INT, tag, 0, MPI_COMM_WORLD);
      MPI_Recv(&lenghtRcv, 1, MPI_INT, tag, 0, MPI_COMM_WORLD, &status);

      promVec.resize(lenghtRcv);

      sendLenghtNew = lengthS / 2 + lengthS % 2;
      MPI_Sendrecv(&localVec[0], sendLenghtNew, MPI_INT, tag, 0, &promVec[0], lenghtRcv, MPI_INT,
        tag, 0, MPI_COMM_WORLD, &status);
      vecOdd = oddFunc(localVec, promVec);
      sendLenghtNew = vecOdd.size();
      MPI_Send(&vecOdd[0], sendLenghtNew, MPI_INT, tag, 0, MPI_COMM_WORLD);
    }
    proc *= 2;
    offset *= 2;
  }
  return localVec;
}


std::vector<int> permutation(std::vector<int> localVec) {
  std::vector<int> promVec(localVec.size());

  for (unsigned int i = 0; i < localVec.size() / 2 + localVec.size() % 2; i++) {
    promVec[i] = localVec[2 * i];
  }
```

```
  for (unsigned int i = 1; i < localVec.size(); i += 2) {
    promVec[localVec.size() / 2 + localVec.size() % 2 + i / 2] = localVec[i];
  }
  for (unsigned int i = 0; i < localVec.size(); i++) {
    localVec[i] = promVec[i];
  }
  return localVec;
}


std::vector<int> evenFunc(const std::vector<int>& localVec, const std::vector<int>& promVec) {
  int size1 = localVec.size() / 2 + localVec.size() % 2;
  int size2 = promVec.size();
  int size = localVec.size() / 2 + localVec.size() % 2 + promVec.size();
  std::vector <int> result(size);
  int j = 0, k = 0, l = 0;
  while ((j < size1) && (k < size2)) {
    if (localVec[j] < promVec[k]) {
      result[l] = localVec[j];
      j++;
    } else {
      result[l] = promVec[k];
      k++;
    }
    l++;
  }
  if (j >= size1) {
    for (int a = k; a < size2; a++) {
      result[l] = promVec[a];
      l++;
    }
  } else {
    for (int a = j; a < size1; a++) {
      result[l] = localVec[a];
      l++;
    }
  }
  return result;
}

std::vector<int> oddFunc(const std::vector<int>& localVec, const std::vector<int>& promVec) {
  int size1 = localVec.size();
  int size2 = promVec.size();
  int size = localVec.size() / 2 + promVec.size();
  std::vector <int> result(size);
  int j = localVec.size() / 2 + localVec.size() % 2, k = 0, l = 0;
  while ((j < size1) && (k < size2)) {
    if (localVec[j] < promVec[k]) {
      result[l] = localVec[j];
      l++;
      j++;
    } else {
      result[l] = promVec[k];
      k++;
      l++;
    }
  }

  if (j < size1) {
    for (int a =j; a < size1; a++) {
      result[l] = localVec[a];
      l++;
    }
  }
```

```
  if (k < size2) {
    for (int t = k; t < size2; t++) {
      result[l] = promVec[t];
      l++;
    }
  }
  return result;
}


std::vector<int> transposition(std::vector<int> localVec, int evenSize, int oddSize) {
  int i;
  if (evenSize - oddSize == 2) {
    std::vector<int> result(localVec.size());
    int a = 0;
    int b = 0;
    int c = 0;
    for (a = 0, b = 0; a < evenSize && b < oddSize; a++, b++) {
      result[c] = localVec[a];
      c++;
      result[c] = localVec[evenSize + b];
      c++;
    }

    for (int t = a; t < evenSize; t++, c++) {
      result[c] = localVec[t];
    }
    int size = result.size();
    for (i = 1; i < size - 1; i += 2) {
      if (result[i] > result[i + 1])
        std::swap(result[i], result[i + 1]);
    }

    result = permutation(result);
    return result;

  } else {
    for (i = 0; i < evenSize - 1; i++)
      if (localVec[1 + i] < localVec[evenSize + i])
        std::swap(localVec[1 + i], localVec[evenSize + i]);
    return localVec;
  }
}

std::vector<int>  merge(std::vector<int> localVec, int evenSize, int oddSize) {
  std::vector<int> result(localVec.size());
  int a = 0;
  int b = 0;
  int c = 0;

  for (a = 0; a < evenSize && b < oddSize; a++, b++) {
    result[c] = localVec[a];
    c++;
    result[c] = localVec[evenSize + b];
    c++;
  }

  for (int t = a; t < evenSize; t++, c++) {
    result[c] = localVec[t];
  }

  int size = result.size();
  for (int i = 1; i < size - 1; i += 2) {
    if (result[i] > result[i + 1])
```

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