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

«Быстрая сортировка с четно-нечетным слиянием Бэтчера»

Выполнил:
студент группы 381706-1
Кукушкина К. О.

Проверил:
Доцент кафедры МОСТ
ИИТММ,
Сысоев А. В.

Нижний Новгород
2019

Содержание

1. Введение

Сортировка является одной из базовых операций обработки данных. Она используется во множестве задач в разных областях, например, в задачах вычислительной геометрии, таких как построение выпуклой оболочки множества точек или нахождение кратчайших путей в графе. На больших объемах данных даже наиболее оптимальные методы сортировки работают достаточно долго, а в случае рекурсивности алгоритма возможно переполнение стека вызовов. В связи с этим кажется сообразным разбивать исходные данные на группы меньшего размера и производить упорядочивание параллельно. Это значительно ускоряет сортировку, однако возникает проблема корректного слияния полученных фрагментов. Использование слияния Бэтчера с построением сетей сортировки решает эту проблему.

Целью настоящей работы является реализация параллельной быстрой сортировки со слиянием Бэтчера и сравнение ее с последовательной быстрой сортировкой.

2. Постановка задачи

Для выполнения цели работы были поставлены следующие задачи:

1. Реализация алгоритма быстрой сортировки
2. Реализация быстрой сортировки со слиянием Бэтчера с использованием средств Microsoft MPI
    a. Построение сети сортировки
    b. Попарное слияние фрагментов
3. Проведение вычислительных экспериментов
4. Сравнение времени работы полученных алгоритмов.

3. Метод решения

Алгоритм быстрой сортировки состоит в следующем:

- Выбрать ведущий элемент (последний в сортируемом подмассиве)
- Все элементы меньше ведущего перенести в левую часть относительно него, большие – в правую
- Рекурсивно повторить процедуру для левой и правой частей.

Алгоритм построения сети сортировки для слияния Бэтчера:

- Если длина входного массива меньше 2, завершить работу
- Разбить входной массив vector на два массива upvec и downvec
- Рекурсивно повторить процедуру для обоих массивов
- Выполнить для массивов следующую процедуру:
    - Если сумма размеров массивов равна 1, завершить работу
    - Если сумма размеров массивов равна 2, добавить пару элементов в сеть сортировки
    - Разбить каждый на два: элементы с четными (upvec_odd, downvec_odd) и нечетными (upvec_even, downvec_even) индексами
    - Рекурсивно выполнить процедуру для пар (upvec_odd, downvec_odd) и (upvec_even, downvec_even)
    - Слить исходные массивы в один
    - Добавить в сеть сортировки пары элементов с индексами (1, 2), (3, 4), ...

Алгоритм быстрой сортировки с использованием слияния Бэтчера предполагает предобработку входного массива – для корректной работы алгоритма количество элементов должно быть кратно количеству процессов – в массив добавляются фиктивные элементы, заведомо меньшие всех элементов. Следующее действие – разбиение входного массива на подмассивы, их независимую сортировку, а затем попарное слияние с использованием построенной сети сортировки.

Более подробно алгоритм рассмотрен в следующем разделе.

4. Схема распараллеливания

Для удобства будем считать, что сеть сортировки уже построена. Первый этап быстрой сортировки со слиянием Бэтчера:

1. Разбиение исходного массива на подмассивы, количество которых равно количеству параллельно работающих процессов
2. Рассылка подмассивов процессам
3. Быстрая сортировка каждого подмассива в своем процессе

Следующий этап – само слияние Бэтчера: в цикле по массиву, хранящему сеть сортировки (каждый элемент – пара номеров процессов или компаратор), процессы с номерами, входящими в компаратор (a, b), отправляют друг другу свои данные. Затем каждый процесс выполняет упорядоченное слияние полученных данных со своими, причем процесс с номером a сохраняет в своем подмассиве первую половину получившегося массива, а процесс с номером b – вторую.

Далее выполняется сбор подмассивов из всех процессов в массив корневого, удаление фиктивных элементов и рассылка полученного массива всем процессам.

5. Описание программной реализации

Быстрая сортировка представлена функцией *quickSort(vec, low, high)* и вспомогательной функцией *partition(vec, low, high)*, где *vec* – исходный массив, *low* и *high* – индексы первого и последнего элементов сортируемого фрагмента.

Построение сети сортировки также представлено двумя функциями: *biuldNet(allranks)* и *addComp(upvec, downvec)*, где *allranks* – массив номеров процессов (0, 1, ...).

Параллельная быстрая сортировка со слиянием Бэтчера реализована в функции *quickBatcher(vec)*, где *vec* – исходный массив.

В программе реализованы вспомогательные функции:
generateRand(vec) – заполняет входной массив *vec* случайными целыми числами от 1 до 99;
isSorted(vec, n) – проверяет, отсортирован ли входной массив *vec* длины *n*.

6. Подтверждение корректности

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

Random_Generation – проверяет возможность случайной генерации вектора большой длины;

Empty_Vector – считается пройденным, если при попытке вызвать функцию quickBatcher для вектора нулевого размера возникает исключение;

Odd_Size – проверка работы quickBatcher на исходном массиве нечетной длины;

Pow_2_Size – проверка работы quickBatcher на исходном массиве с длиной, равной степени 2;

Quick_Sort – с помощью функции isSorted проверяет корректность работы quickSort;

Batcher_Sort – с помощью функции isSorted проверяет корректность работы batcherSort;

Equal_Result – проверка результатов работы quickSort и batcherSort на идентичность;

Efficiency – проверка эффективности параллельной схемы. Считается пройденным, если время работы quickBatcher оказалось меньше времени работы quickSort.

7. Результаты экспериментов

Эксперименты проводились на ПК с следующими параметрами:

1. Операционная система: Windows 10 Домашняя
2. Процессор: Intel(R) Core™ i5-7200U CPU @ 2.50 GHz 2.71 GHz
3. Версия Visual Studio: 2019

В рамках эксперимента было вычислено время работы параллельного и последовательного алгоритмов (усреднение по трем экспериментам) для исходного массива размера 100 000 и количества процессов от 1 до 10, а также полученное ускорение. Результаты представлены в таблице 1.

Таблица 1. Время работы алгоритмов

Количество процессов	Время работы quickSort, сек	Время работы batcherSort, сек	Ускорение
1	1.0735	0.9772	1.099
2	1.0287	0.2789	3.688
3	1.2047	0.2716	4.436
4	1.5169	0.1577	9.619
5	1.8770	0.1933	9.418
6	2.2209	0.2657	8.359
7	2.5991	0.2554	10.177
8	2.9001	0.3749	7.736
9	3.2835	0.4608	7.126
10	3.3683	0.5154	6.535

Полученное ускорение превосходит количество физических ядер ПК, на котором проводилось тестирование, что объясняется следующими факторами:

1. Скорость доступа к памяти: с уменьшением размера массива, обрабатываемого одним процессом, уменьшается частота кэш-промахов;
2. Глубина рекурсии: сложность быстрой сортировки составляет $O(n \log n)$. При разделении вычислений на q процессов получаем

$$\frac{n}{q}\log\frac{n}{q} = \frac{n}{q}(\log n - \log q)$$

Найдем соотношение:

$$\frac{n \log n}{\frac{n}{q}(\log n - \log q)} = q + \log_q n$$

Таким образом, если сортировка n элементов происходит за время t, сортировка $\frac{n}{q}$ элементов происходит быстрее, чем за время $\frac{t}{q}$.

На основе полученных данных можно сделать вывод, что использование быстрой сортировки со слиянием Бэтчера значительно более эффективно, чем использование классического алгоритма быстрой сортировки.

8. Заключение

В ходе работы были реализованы два алгоритма сортировки массивов: классическая быстрая сортировка и параллельная быстрая сортировка со слиянием Бэтчера. Вычислительные эксперименты показали, что классический алгоритм значительно уступает алгоритму с использованием параллельных вычислений в эффективности.

Быстрая сортировка в параллельном алгоритме работает на меньшем объеме данных, что сказывается как на времени работы, так и на глубине рекурсии. Этот факт делает ее более предпочтительной для использования в реальных вычислительных задачах.

9. Список литературы

1. Якобовский М. В. *Параллельные алгоритмы сортировки больших объемов данных* // Фундаментальные физико-математические проблемы и моделирование технико-технологических систем: Сб. науч. тр.: Янус-К, 2004, с. 235 – 249.
2. Быстрая сортировка [Электронный ресурс]: Википедия, свободная энциклопедия. –https://ru.wikipedia.org/wiki/Быстрая_сортировка
3. Сеть Бэтчера [Электронный ресурс]: Викиконспекты. – Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=Сеть_Бетчера
4. Сеть обменной сортировки со слиянием Бэтчера [Электронный ресурс]: Хабр, сообщество IT-специалистов. – https://habr.com/ru/post/275889/

10. Приложение

Приложение 1. quickBatcher.h

```
//
Copyright
2019
Kukushkina
Ksenia
```

```
#ifndef MODULES_TASK_3_KUKUSHKINA_K_QUICK_BATCHER_QUICK_BATCHER_H_
#define MODULES_TASK_3_KUKUSHKINA_K_QUICK_BATCHER_QUICK_BATCHER_H_
#include <mpi.h>
#include <vector>
void generateRand(std::vector<int>* vec);
bool isSorted(const std::vector<int>& vec, int n);
void quickSort(std::vector<int>* vec, int low, int high);
void addComp(std::vector<int> upvec, std::vector<int> downvec);
void buildNet(std::vector<int> allranks);
void quickBatcher(std::vector<int>* vec);
#endif  // MODULES_TASK_3_KUKUSHKINA_K_QUICK_BATCHER_QUICK_BATCHER_H_
```

Приложение 2. quickBatcher.cpp

```
//
Copyright
2019
Kukushkina
Ksenia
```

```cpp
#include <../../../modules/task_3/kukushkina_k_quick_batcher/quick_batcher.h>
#include <mpi.h>
#include <algorithm>
#include <ctime>
#include <iostream>
#include <random>
#include <utility>
#include <vector>
std::vector<std::pair<int, int>> comparators;
static int offset = 0;
void generateRand(std::vector<int>* vec) {
  std::mt19937 gen;
  gen.seed(static_cast<unsigned int>(time(0)) + offset++);
  for (int i = 0; i < static_cast<int>(vec->size()); i++)
    (*vec)[i] = gen() % 100;
  return;
}
bool isSorted(const std::vector<int>& vec, int n) {
  for (int i = 0; i < n - 1; i++)
    if (vec[i] > vec[i + 1])
      return false;
  return true;
}
int partition(std::vector<int>* vec, int low, int high) {
  int pivot = (*vec)[high];
  int tmp;
  int i = (low - 1);
  for (int j = low; j <= high - 1; j++) {
    if ((*vec)[j] < pivot) {
      i++;
      tmp = (*vec)[i];
      (*vec)[i] = (*vec)[j];
      (*vec)[j] = tmp;
    }
  }
  tmp = (*vec)[i + 1];
  (*vec)[i + 1] = (*vec)[high];
  (*vec)[high] = tmp;
  return (i + 1);
}
void quickSort(std::vector<int>* vec, int low, int high) {
  if (low < high) {
```

```
    int pivot = partition(vec, low, high);
    quickSort(vec, low, pivot - 1);
    quickSort(vec, pivot + 1, high);
  }
}
void addComp(std::vector<int> upvec, std::vector<int> downvec) {
  int ressize = static_cast<int>(upvec.size())
    + static_cast<int>(downvec.size());
  if (ressize == 1) return;
  if (ressize == 2) {
    std::pair<int, int> tmp{ upvec[0], downvec[0] };
    comparators.push_back(tmp);
    return;
  }
  std::vector<int> upvec_odd, downvec_odd, upvec_even, downvec_even,
vecres(ressize);
  for (int i = 0; i < static_cast<int>(upvec.size()); i++) {
    if (i % 2)
      upvec_even.push_back(upvec[i]);
    else
      upvec_odd.push_back(upvec[i]);
  }
  for (int i = 0; i < static_cast<int>(downvec.size()); i++) {
    if (i % 2)
      downvec_even.push_back(downvec[i]);
    else
      downvec_odd.push_back(downvec[i]);
  }
  addComp(upvec_odd, downvec_odd);
  addComp(upvec_even, downvec_even);
  std::copy(upvec.begin(), upvec.end(), vecres.begin());
  std::copy(downvec.begin(), downvec.end(), vecres.begin() + upvec.size());
  for (int i = 1; i < static_cast<int>(vecres.size()) - 1; i += 2) {
    std::pair<int, int> tmp{ vecres[i], vecres[i + 1] };
    comparators.push_back(tmp);
  }
}
void buildNet(std::vector<int> allranks) {
  if (allranks.size() < 2) return;
  std::vector<int> upvec(allranks.size() / 2);
  std::vector<int> downvec(allranks.size() / 2 + allranks.size() % 2);
  std::copy(allranks.begin(), allranks.begin() + upvec.size(), upvec.begin());
  std::copy(allranks.begin() + upvec.size(), allranks.end(), downvec.begin());
  buildNet(upvec);
  buildNet(downvec);
  addComp(upvec, downvec);
}
void quickBatcher(std::vector<int>* vec) {
  MPI_Status status;
```

```
  int rank, size;
  int vecsize = static_cast<int>(vec->size());
  int n = vecsize;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  if (n <= 0)
    throw "Negative size";
  while (n % size) {
    vec->push_back(-1000);
    n++;
  }
  int len = n / size;
  std::vector<int> allranks(size);
  for (int i = 0; i < static_cast<int>(allranks.size()); i++) {
    allranks[i] = i;
  }
  buildNet(allranks);
  std::vector<int> resvec(len), curvec(len), tmpvec(len);
  MPI_Scatter(&(*vec)[0], len, MPI_INT, &resvec[0], len, MPI_INT, 0,
MPI_COMM_WORLD);
  quickSort(&resvec, 0, len - 1);
  for (int i = 0; i < static_cast<int>(comparators.size()); i++) {
    int a = comparators[i].first, b = comparators[i].second;
    if (rank == a) {
      MPI_Send(&resvec[0], len, MPI_INT, b, 0, MPI_COMM_WORLD);
      MPI_Recv(&curvec[0], len, MPI_INT, b, 0, MPI_COMM_WORLD, &status);
      for (int resi = 0, curi = 0, tmpi = 0; tmpi < len; tmpi++) {
        int res = resvec[resi];
        int cur = curvec[curi];
        if (res < cur) {
          tmpvec[tmpi] = res;
          resi++;
        } else {
          tmpvec[tmpi] = cur;
          curi++;
        }
      }
      resvec.swap(tmpvec);
    } else if (rank == b) {
      MPI_Recv(&curvec[0], len, MPI_INT, a, 0, MPI_COMM_WORLD, &status);
      MPI_Send(&resvec[0], len, MPI_INT, a, 0, MPI_COMM_WORLD);
      int start = len - 1;
      for (int resi = start, curi = start, tmpi = start; tmpi >= 0; tmpi--) {
        int res = resvec[resi];
        int cur = curvec[curi];
        if (res > cur) {
          tmpvec[tmpi] = res;
          resi--;
        } else {
```

```
            tmpvec[tmpi] = cur;
            curi--;
          }
        }
        resvec.swap(tmpvec);
      }
    }
    MPI_Gather(&resvec[0], len, MPI_INT, &(*vec)[0], len, MPI_INT, 0, MPI_COMM_WORLD);
    int elDiff = n - vecsize;
    if (rank == 0 && elDiff) {
      vec->erase(vec->begin(), vec->begin() + elDiff);
    }
    MPI_Bcast(&(*vec)[0], vecsize, MPI_INT, 0, MPI_COMM_WORLD);
}
```

Приложение 3. main.cpp

```
//
Copyright
2019
Kukushkina
Ksenia
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <vector>
#include "./quick_batcher.h"
TEST(Quick_Batcher_Sort, Random_Generation) {
  std::vector<int> vec(100000);
  ASSERT_NO_THROW(generateRand(&vec));
}
TEST(Quick_Batcher_Sort, Empty_Vector) {
  std::vector<int> vec(0);
  ASSERT_ANY_THROW(quickBatcher(&vec));
}
TEST(Quick_Batcher_Sort, Odd_Size) {
  std::vector<int> vec(1001);
  generateRand(&vec);
  ASSERT_NO_THROW(quickBatcher(&vec));
}
TEST(Quick_Batcher_Sort, Pow_2_Size) {
  std::vector<int> vec(256);
  generateRand(&vec);
  ASSERT_NO_THROW(quickBatcher(&vec));
}
TEST(Quick_Batcher_Sort, Quick_Sort) {
  std::vector<int> vec(100);
  generateRand(&vec);
  quickSort(&vec, 0, 99);
  ASSERT_EQ(isSorted(vec, 99), true);
}
TEST(Quick_Batcher_Sort, Batcher_Sort) {
  std::vector<int> vec(1000);
  generateRand(&vec);
  quickBatcher(&vec);
  ASSERT_EQ(isSorted(vec, 999), true);
}
TEST(Quick_Batcher_Sort, Equal_Result) {
  std::vector<int> vec(1000), vec1(1000);
  generateRand(&vec);
  vec1 = vec;
  quickSort(&vec, 0, 999);
  quickBatcher(&vec1);
  bool eq = true;
  for (int i = 0; i < 1000; i++)
    if (vec1[i] != vec[i])
```

```
        eq = false;
    ASSERT_EQ(eq, true);
}
/* TEST(Quick_Batcher_Sort, Efficiency) {
    std::vector<int> vec(100000), vec1(100000);
    double t1, t2, finish;
    generateRand(vec);
    vec1 = vec;
    double start = MPI_Wtime();
    quickSort(vec, 0, 99999);
    finish = MPI_Wtime();
    t1 = finish - start;
    start = MPI_Wtime();
    quickBatcher(&vec1);
    finish = MPI_Wtime();
    t2 = finish - start;
    std::cout << "Qsort: " << t1 << "s\nBatcher: " << t2 << "s\n";
    ASSERT_EQ(t1 > t2, true);
}  */
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();
}
```