МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Нижегородский государственный университет им. Н.И. Лобачевского»
Национальный исследовательский университет

Институт информационных технологий, математики и механики

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ
«Поразрядная сортировка для целых чисел с простым слиянием.»

Выполнил: студент группы 381706-2
Гущин Александр Владимирович
_____ Подпись

Проверил: доцент кафедры МОСТ,
кандидат технических наук
Сысоев Александр Владимирович

_____ Подпись

Нижний Новгород
2019

Оглавление

Введение

Алгоритм сортировки — это алгоритм для упорядочивания элементов. Существует множество алгоритмов сортировки, но на больших объемах данных далеко не все они являются эффективными. Конкретно для целых чисел можно взять алгоритм поразрядной сортировки.

Суть параллельной сортировки заключается в разбиении массива на несколько частей. К каждой части применяется сортировка, после чего они сливаются в отсортированный массив.

В данной лабораторной применяется метод простого слияния.

Постановка задачи

1. Реализовать последовательной алгоритм поразрядной сортировки целых чисел.
2. Реализовать параллельной алгоритм сортировки с использованием поразрядной сортировки целых чисел и простым слиянием.
3. Провести эксперименты.
4. Сравнить время работы последовательного и параллельного алгоритма.

Метод решения

Идея заключается в последовательной сортировке по разрядам. В данной лабораторной реализован проход от младшего разряда к старшему. Так как целое число представляется в памяти компьютера как несколько последовательно расположенных байт, то логично взять размер разряда 256. Так как в некоторых системах используется big-endian, а в других little-endian, то до момента начала сортировки необходимо определить порядок байт. Также стоит заранее узнать количество байт в типе данных (к примеру, long long = 8 байт) и является ли тип данных знаковым или беззнаковым.

На каждом разряде:

1. Заносим количество всех значений разряда в массив count.

   Если разряд является старшим и тип данных является знаковым, то необходимо посчитать все отрицательные числа в переменную shift для последующего смещения.

2. Изменяем массив count в массив смещения

   count[i] = sum;
   sum += tmp;

3. Проходим по основному массиву и подставляем числа в дополнительный массив в зависимости от массива смещений count и переменной shift.

Смысл простого слияния заключается в выборе минимального элемента из 2 массивов. Так как массивы являются отсортированными, то необходимо только сравнить их начальные элементы, если элемент является минимальным, то дальше сравнивается следующий элемент из этого массива.

Схема распараллеливания

Сортируемый массив разбивается на части в зависимости от количества процессов. Остаток присоединяется к нулевому процессу. Каждый процесс сортирует свою часть.

Представим процессы как бинарное дерево. Потомками i процесса являются i * 2 + 1 и i * 2 + 2. Если у процесса нет потомка, то он только отправляет отсортированный массив своему родителю. Если потомки всё же есть, то сначала происходит прием массива потомков, после чего происходит слияние и уже потом отправка родителю.

Описание программной реализации

std::vector<T> Radix_sort(std::vector<T> st) – поразрядная шаблонная сортировка для целых типов данных.

void Merge(T* res, T* vec_1, T* vec_2, int first_size, int second_size) – слияние 2 массивов в массив res.

std::vector<T> P_radix_sort(std::vector<T> st) – параллельная поразрядная шаблонная сортировка для целых чисел.

void Fill_random(T* vec, int size) – заполняет массив случайными числами. Используется для тестов.

Подтверждение корректности

В данной лабораторной используются Google C++ Testing Framework. Все тесты можно разделить на 2 типа. Первый тип — это тесты на случайно генерируемых массивах для подтверждения работы на любых целочисленных типах данных. Второй тип — это тесты на проверку на заранее заданном массиве.

Эксперименты

Процессор: Ryzen 5 3600u

Память: 8 гб

Система: Windows 10

Массив: 10000000 элементов типа long long

Кол-во процессов	Время последовательного алгоритма	Время параллельного алгоритма	Коэффициент ускорения
1	1.27096	1.49953	0.84
2	1.21343	0.917295	1.32
3	1.29896	0.708598	1.83
4	1.23253	0.644556	1.91

Параллельная сортировка показывает себя лучше при любом количестве процессов кроме 1. Малую разницу в ускорении между 3 и 4 процессами можно объяснить в неравномерном распределении массива (на левую ветку корня приходиться половина массива).

Заключение

Эксперимент показал, что при использование последовательных алгоритмов на большом объеме данных является неэффективным по сравнению с их распараллеленными аналогами. Это также относится и к поразрядной сортировке массивов.

Литература

1. Вики-конспекты [Электронный ресурс]. – Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=Цифровая_сортировка
2. ИНТУИТ [Электронный ресурс]. – Режим доступа: https://www.intuit.ru/studies/courses/12181/1174/lecture/25257

Приложения

radix_sort_s_m.h

```
// Copyright 2019 Guschin Alexander
#ifndef MODULES_TASK_3_GUSCHIN_A_RADIX_SORT_S_M_RADIX_SORT_S_M_H_
#define MODULES_TASK_3_GUSCHIN_A_RADIX_SORT_S_M_RADIX_SORT_S_M_H_
#include <mpi.h>
#include <ctime>
#include <random>
#include <string>
#include <vector>

int D_heap_cntr(int root, int size);

template <class T>
void Fill_random(T* vec, int size) {
  std::mt19937 gen(time(0));
  for (int i = 0; i < size; ++i) vec[i] = static_cast<T>(gen());
}

template <class T>
std::vector<T> Radix_sort(std::vector<T> st) {
  int b_len = sizeof(T);
  bool is_signed = std::is_signed<T>::value;

  std::uint16_t* test_int16 = new std::uint16_t(1);
  bool is_lit_end =
      *(reinterpret_cast<std::uint8_t*>(test_int16)) == 0 ? false : true;
  delete test_int16;
  std::uint8_t* ptr = reinterpret_cast<std::uint8_t*>(&st[0]);
  int size = st.size();
  std::vector<T> res(size);
  for (int k = 0; k < b_len; ++k) {
    int count[256] = {0};
    if (is_lit_end) {
      for (int i = 0; i < size; ++i) count[*(ptr + k + i * b_len)]++;
    } else {
      for (int i = 0; i < size; ++i)
        count[*(ptr + b_len - 1 - k + i * b_len)]++;
    }

    int shift = 0;
    if (is_signed && k == b_len - 1) {
      for (int i = 128; i < 256; ++i) shift += count[i];
    }

    int sum = 0;
    for (int i = 0; i < 256; ++i) {
      int tmp = count[i];
      count[i] = sum;
      sum += tmp;
    }

    if (is_lit_end) {
      for (int i = 0; i < size; ++i) {
        res[(count[*(ptr + k + i * b_len)] + shift) % size] = st[i];
        count[*(ptr + k + i * b_len)]++;
      }
    } else {
      for (int i = 0; i < size; ++i) {
```

```
        res[(count[*(ptr + b_len - 1 - k + i * b_len)] + shift) % size] = st[i];
        count[*(ptr + b_len - 1 - k + i * b_len)]++;
      }
    }
    st = res;
  }

  return res;
}

template <class T>
void Merge(T* res, T* vec_1, T* vec_2, int first_size, int second_size) {
  int i = 0, j = 0;
  while (i < first_size && j < second_size) {
    if (vec_1[i] < vec_2[j]) {
      res[i + j] = vec_1[i];
      ++i;
    } else {
      res[i + j] = vec_2[j];
      ++j;
    }
  }
  while (i < first_size) {
    res[i + j] = vec_1[i];
    ++i;
  }
  while (j < second_size) {
    res[i + j] = vec_2[j];
    ++j;
  }
}

template <class T>
std::vector<T> P_radix_sort(std::vector<T> st) {
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  int b_len = sizeof(T);

  int vec_size;
  if (rank == 0) {
    vec_size = st.size();
  }
  MPI_Bcast(&vec_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  int sort_size, sort_rem;
  sort_size = vec_size / size;
  sort_rem = vec_size % size;

  std::vector<T> local_vector(sort_size);
  if (rank == 0) local_vector.resize(sort_size + sort_rem);

  MPI_Scatter(reinterpret_cast<std::uint8_t*>(&st[0]) + sort_rem * b_len,
              sort_size * b_len, MPI_CHAR, &local_vector[0], sort_size * b_len,
              MPI_CHAR, 0, MPI_COMM_WORLD);

  if (rank == 0) {
    for (int i = sort_size; i < sort_rem + sort_size; ++i)
      local_vector[i] = st[i - sort_size];
  }
  std::vector<T> sort_res(Radix_sort(local_vector));
  int self_cntr = sort_size;
  if (rank == 0) self_cntr += sort_rem;
```

```
  T* left_cntr = nullptr;
  T* rigth_cntr = nullptr;
  T* rigth_merge = nullptr;

  std::vector<T> total(
      (D_heap_cntr(rank * 2 + 2, size) + D_heap_cntr(rank * 2 + 1, size)) *
          sort_size + self_cntr);

  if (rank * 2 + 1 < size) {
    int child_w = D_heap_cntr(rank * 2 + 1, size);
    left_cntr = new T[child_w * sort_size];
    MPI_Status status;
    MPI_Recv(reinterpret_cast<std::uint8_t*>(left_cntr),
             child_w * sort_size * b_len, MPI_CHAR, rank * 2 + 1, 0,
             MPI_COMM_WORLD, &status);
  }
  if (rank * 2 + 2 < size) {
    int child_w = D_heap_cntr(rank * 2 + 2, size);
    rigth_cntr = new T[child_w * sort_size];
    MPI_Status status;
    MPI_Recv(reinterpret_cast<std::uint8_t*>(rigth_cntr),
             child_w * sort_size * b_len, MPI_CHAR, rank * 2 + 2, 0,
             MPI_COMM_WORLD, &status);
  }

  if (rank * 2 + 1 >= size && rank * 2 + 2 >= size && rank != 0) {
    MPI_Send(reinterpret_cast<std::uint8_t*>(&sort_res[0]),
             sort_res.size() * b_len,
             MPI_CHAR, (rank - 1) / 2, 0, MPI_COMM_WORLD);
  } else {
    if (rank * 2 + 2 < size) {
      rigth_merge =
          new T[D_heap_cntr(rank * 2 + 2, size) * sort_size + self_cntr];
      Merge(rigth_merge, rigth_cntr, &sort_res[0],
            D_heap_cntr(rank * 2 + 2, size) * sort_size, self_cntr);
      Merge(&total[0], rigth_merge, left_cntr,
            D_heap_cntr(rank * 2 + 2, size) * sort_size + self_cntr,
            D_heap_cntr(rank * 2 + 1, size) * sort_size);
    } else {
      Merge(&total[0], &sort_res[0], left_cntr,
            self_cntr,
            D_heap_cntr(rank * 2 + 1, size) * sort_size);
    }
    if (rank != 0) {
      MPI_Send(reinterpret_cast<std::uint8_t*>(&total[0]), total.size() * b_len,
               MPI_CHAR, (rank - 1) / 2, 0, MPI_COMM_WORLD);
    }
  }
  if (left_cntr != nullptr) delete[] left_cntr;
  if (rigth_cntr != nullptr) delete[] rigth_cntr;
  if (rigth_merge != nullptr) delete[] rigth_merge;
  return total;
}
#endif  // MODULES_TASK_3_GUSCHIN_A_RADIX_SORT_S_M_RADIX_SORT_S_M_H_
```

radix_sort_s_m.cpp

```
// Copyright 2019 Guschin Alexander
#include <mpi.h>
#include <string>
#include <vector>
#include "../../../modules/task_3/guschin_a_radix_sort_s_m/radix_sort_s_m.h"

int D_heap_cntr(int root, int size) {
  if (root >= size) return 0;
  int base = 1;
  if (root * 2 + 1 < size) base += D_heap_cntr(root * 2 + 1, size);
  if (root * 2 + 2 < size) base += D_heap_cntr(root * 2 + 2, size);
  return base;
}
```

main.cpp

```
// Copyright 2019 Guschin Alexander
#include <gtest/gtest.h>
#include <gtest-mpi-listener.hpp>
#include <string>
#include <vector>
#include "../../../modules/task_3/guschin_a_radix_sort_s_m/radix_sort_s_m.h"

TEST(radix_sort, radix_sort) {
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  std::vector<std::int64_t> vec;

  if (rank == 0) {
    vec.resize(1000);
    Fill_random(&vec[0], vec.size());
    std::vector<std::int64_t> res(Radix_sort(vec));
    bool is_sort = true;
    int length = vec.size();
    for (int i = 1; i < length; ++i)
      if (res[i] < res[i - 1]) {
        is_sort = false;
        break;
      }
    EXPECT_EQ(is_sort, 1);
  }
}

TEST(radix_sort, can_sort_int64) {
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  std::vector<std::int64_t> vec(1000);
  Fill_random(&vec[0], vec.size());
  std::vector<std::int64_t> res(P_radix_sort(vec));
  if (rank == 0) {
    bool is_sort = true;
    int length = vec.size();
    for (int i = 1; i < length; ++i)
      if (res[i] < res[i - 1]) {
        is_sort = false;
        break;
      }

    EXPECT_EQ(is_sort, 1);
  }
}

TEST(radix_sort, can_sort_uint64) {
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  std::vector<std::uint64_t> vec(1000);
  Fill_random(&vec[0], vec.size());
  std::vector<std::uint64_t> res(P_radix_sort(vec));
  if (rank == 0) {
    bool is_sort = true;
    int length = vec.size();
    for (int i = 1; i < length; ++i)
```

```
      if (res[i] < res[i - 1]) {
        is_sort = false;
        break;
      }

    EXPECT_EQ(is_sort, 1);
  }
}

TEST(radix_sort, can_sort_int32) {
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  std::vector<std::int32_t> vec(1000);
  Fill_random(&vec[0], vec.size());
  std::vector<std::int32_t> res(P_radix_sort(vec));
  if (rank == 0) {
    bool is_sort = true;
    int length = vec.size();
    for (int i = 1; i < length; ++i)
      if (res[i] < res[i - 1]) {
        is_sort = false;
        break;
      }

    EXPECT_EQ(is_sort, 1);
  }
}

TEST(radix_sort, can_sort_uint32) {
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  std::vector<std::uint32_t> vec(1000);
  Fill_random(&vec[0], vec.size());
  std::vector<std::uint32_t> res(P_radix_sort(vec));
  if (rank == 0) {
    bool is_sort = true;
    int length = vec.size();
    for (int i = 1; i < length; ++i)
      if (res[i] < res[i - 1]) {
        is_sort = false;
        break;
      }

    EXPECT_EQ(is_sort, 1);
  }
}

TEST(radix_sort, can_sort_int16) {
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  std::vector<std::int16_t> vec(1000);
  Fill_random(&vec[0], vec.size());
  std::vector<std::int16_t> res(P_radix_sort(vec));
  if (rank == 0) {
    bool is_sort = true;
    int length = vec.size();
    for (int i = 1; i < length; ++i)
      if (res[i] < res[i - 1]) {
```

```
        is_sort = false;
        break;
      }

    EXPECT_EQ(is_sort, 1);
  }
}

TEST(radix_sort, can_sort_uint16) {
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  std::vector<std::uint16_t> vec(1000);
  Fill_random(&vec[0], vec.size());
  std::vector<std::uint16_t> res(P_radix_sort(vec));
  if (rank == 0) {
    bool is_sort = true;
    int length = vec.size();
    for (int i = 1; i < length; ++i)
      if (res[i] < res[i - 1]) {
        is_sort = false;
        break;
      }

    EXPECT_EQ(is_sort, 1);
  }
}

TEST(radix_sort, can_sort_defined_array_int32) {
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  std::vector<std::int32_t> vec{-32, 11, 332, 42, 1 << 28, -8, 1 << 26, -(1 << 26), 55, -
24, -(1 << 27), 1111};
  std::vector<std::int32_t> expected{-(1 << 27), -(1 << 26), -32, -24, -8, 11, 42, 55,
332, 1111, 1 << 26, 1 << 28};
  std::vector<std::int32_t> res(P_radix_sort(vec));
  if (rank == 0) {
    EXPECT_EQ(res, expected);
  }
}

TEST(radix_sort, can_sort_defined_array_uint16) {
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  std::vector<std::uint16_t> vec{1, 4, 2, 16, 4, 64, 8, 256, 16, 1024};
  std::vector<std::uint16_t> expected{1, 2, 4, 4, 8, 16, 16, 64, 256, 1024};
  std::vector<std::uint16_t> res(P_radix_sort(vec));
  if (rank == 0) {
    EXPECT_EQ(res, expected);
  }
}

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();
}
```