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 381706-2
Мышкин А.А.

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

Нижний Новгород
2019

Содержание

Введение

Сортировкой - называется процесс упорядочивания множества объектов по какому-либо признаку. Так как данные могут храниться в разных структурах, то и алгоритмы для каждой структуры могут отличаться.

В данной лабораторной работе мы рассмотрим такую сортировку как сортировка Шелла. Так как её быстродействие на большом количестве элементов оставляет желать лучшего, предлагается ускорить эту сортировку за счет распараллеливания. Распараллеливание будет осуществляться с помощью такого программного интерфейса, как Microsoft MPI. Это значительно ускоряет сортировку, также важен тот факт, что сортировка Шелла поддается распараллеливанию, иначе параллельное исполнение может только замедлить работу алгоритма.

Постановка задачи

Для получения нужного результата, необходимо поставить несколько задач:

- ✓ Реализация последовательного алгоритма сортировки Шелла
- ✓ Реализация последовательного алгоритма сортировки простым слиянием
- ✓ Реализация параллельного алгоритма сортировки Шелла с простым слиянием
- ✓ Проведение экспериментов на правильность работы алгоритмов
- ✓ Проведение экспериментов на сравнение быстродействия последовательного и параллельного алгоритмов сортировки Шелла

Описание алгоритма

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 (в нашем случае изначальное значение $d=N/2$). 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.

Худшее время работы: $O(n^2)$.

Лучшее время работы: $O(n \log^2 n)$.

Пример. Пусть дан список A =(32, 95, 16, 82, 24, 66, 35, 19, 75, 54, 40, 43, 93, 68) и выполняется его сортировка методом Шелла, а в качестве значений d выбраны 5, 3, 1.

На первом шаге сортируются подсписки A, составленные из всех элементов A, различающихся на 5 позиций, то есть подсписки A(5,1) = (32, 66, 40), A(5,2) = (95,35,43), A(5,3) = (16, 19, 63), A(5,4) = (82, 75, 68), A(5,5) = (24, 54).

В полученном списке на втором шаге вновь сортируются подсписки из отстоящих на 3 позиции элементов.

Процесс завершается обычной сортировкой вставками получившегося списка.

Исходный массив	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
После сортировки с шагом 5	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 обменов
После сортировки с шагом 3	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 обменов
После сортировки с шагом 1	16	19	24	32	35	40	43	54	66	68	75	82	93	95	15 обменов

Рисунок 1. Пример алгоритма сортировки Шелла

Сортировка слиянием (англ. *merge sort*) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер

достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

- Массив рекурсивно разбивается пополам, и каждая из половин делиться до тех пор, пока размер очередного подмассива не станет равным единице;
- Далее выполняется операция алгоритма, называемая слиянием. Два единичных массива сливаются в общий результирующий массив, при этом из каждого выбирается меньший элемент (сортировка по возрастанию) и записывается в свободную левую ячейку результирующего массива. После чего из двух результирующих массивов собирается третий общий отсортированный массив, и так далее. В случае если один из массивов закончиться, элементы другого дописываются в собираемый массив;
- В конце операции слияния, элементы перезаписываются из результирующего массива в исходный.

Худшее время работы: *O (n log n).*

Лучшее время работы: *O (n log n)* обычно, *O (n)* на упорядоченном массиве

Исходный файл f: **5 7 3 2 8 4 1**

	Распределение	***Слияние***
1 проход	*f1*: 5 3 8 1 *f2*: 7 2 4	*f*: 5 7 2 3 4 8 1
2 проход	*f1*: 5 7 4 8 *f2*: 2 3 1	*f*: 2 3 5 7 1 4 8
3 проход	*f1*: 2 3 5 7 *f2*: 1 4 8	*f*: 1 2 3 4 5 7 8

Рисунок 2. Пример алгоритма сортировки простым слиянием

Описание схемы распараллеливания

Распараллеливание происходит в функции parallelShellSort. Изначально, возьмем шаг равный N/2, где N – количество элементов в массиве.

В цикле пока шаг больше 0, выполняем следующие действия:

- При каждом изменении шага создаем временный контейнер размером N/step для каждого процесса, где step – шаг на данном этапе основного цикла.
- В цикле пока мы не выходим за границы шага, (внутри стоит счетчик, который подсчитывает количество отданных массивов, если этот счетчик становится больше размера шага, происходит выход из цикла), заходим в цикл по количеству процессов из MPI_COMM_WORLD.
- В нулевом процессе во временный контейнер записываем данные из основного массива. Эти данные мы выбираем в зависимости от размера текущего шага, значения счетчика (пояснение см. ниже) и от процесса, которому он будет передан для дальнейшей сортировки.
- Если мы находимся в первой итерации цикла, в нулевом процессе, сортируем временный массив сортировкой простым слиянием и записываем отсортированные данные в изначальный массив на те места, откуда мы брали значения с учетом изменения их в порядке записи.
- Если же мы в нулевом процессе, но не в первой итерации, передаем временный массив процессу (сопоставимому с итерацией в цикле), в котором он его отсортирует с помощью сортировки простым слиянием и вернет обратно в нулевой. В нулевом процессе происходит вставка отсортированного временного массива в изначальные позиции, откуда были взяты эти значения с учетом изменения порядка записи.

Таким образом, была сохранена идея алгоритма последовательной сортировки Шелла. Передачу/прием данных между процессами обеспечивает пара функций MPI_Send и MPI_Recv.

Описание программной реализации

Для корректной работы программы требуется несколько функций:

int getRandomArray(int* buffer, int size) – функция создания случайного массива

int sortingCheck(int* buffer1, int* buffer2, int size) – функция проверки двух массивов на сортировку по возрастанию и поэлементного совпадения между собой

int ShellSortSenq(int* buffer, int length) – последовательный алгоритм сортировки Шелла

int mergeSort(int* buffer, int size) – алгоритм сортировки простым слиянием

int parallelShellSort(int* buffer, int length) – параллельный алгоритм сортировки Шелла

Подтверждение корректности

Для подтверждения корректности работы параллельного алгоритма сортировки Шелла реализованы несколько тестов с использованием библиотеки для модульного тестирования Google C++ Testing Framework.

TEST (Parallel_Shell_Sort, Test_With_Shell_Sort) – проверка корректности работы параллельной сортировки и последовательной сортировки Шелла

TEST (Parallel_Shell_Sort, Test_With_Merge_Sort) - проверка корректности работы параллельной сортировки и сортировки простым слиянием

TEST (Parallel_Shell_Sort, Test_With_Shell_Sort_Size_Zero) – проверка создания массива при размере 0 элементов

Тесты проверки корректности работы параллельной сортировки и последовательной сортировки Шелла при различных размерах массивов:

TEST (Parallel_Shell_Sort, Test_With_Shell_Sort_Size_One)

TEST (Parallel_Shell_Sort, Test_With_Shell_Sort_Size_Two)

TEST (Parallel_Shell_Sort, Test_With_Shell_Sort_Size_Three)

Результаты экспериментов

Эксперименты проводились на ультрабуке со следующими параметрами:

1. Операционная система: Windows 10 Pro
2. Процессор: Intel(R) Core™ i7-3517U CPU @ 1.90 GHz
3. ОЗУ: 4 ГБ
4. Версия Visual Studio 2019

Количество элементов	**Последовательный алгоритм сортировки Шелла**		**Параллельны алгоритм сортировки Шелла**	
	1 процесс	2 процесса	1 процесс	2 процесса
50000	0.426 мс	0.429 мс	0.311 мс	0.287 мс
100000	1.838 мс	1.893 мс	0.606 мс	0.564 мс
130000	2.946 мс	2.876 мс	1.154 мс	0.808 мс

Таблица 1. Сравнение времени работы последовательного и параллельного сортировок

Исходя из данных проведенных экспериментов, можно сказать, что при увеличении числа процессов алгоритм параллельной сортировки ускоряется. Также параллельный алгоритм работает быстрее, чем его последовательный конкурент. Следовательно, можно сделать вывод, что алгоритм работает правильно и достигнуто цель ускорения данного алгоритма сортировки.

Можно сделать предположение, что при увеличении количества элементов массива ускорение также будет сохранено.

Заключение

В ходе данной лабораторной работы были реализованы параллельный алгоритм сортировки Шелла с простым слиянием и последовательный алгоритм сортировки Шелла. Благодаря проведенным экспериментам, удалось установить правильность работы созданного алгоритма и удостовериться, что параллельный алгоритм данной сортировки эффективнее последовательной.

Также корректность данных суждений подтверждает проверка с помощью Google C++ Testing Framework.

Литература

1. Википедия: https://ru.wikipedia.org/wiki/Сортировка_Шелла
2. Интернет-статья: https://neerc.ifmo.ru/wiki/index.php?title=Сортировки
3. Интернет-статья: https://habr.com/ru/post/274017/
4. Интернет-статья: http://aliev.me/runestone/SortSearch/TheShellSort.html

Приложение

shell_sort.h

```
// Copyright 2019 Myshkin Andrew
#ifndef MODULES_TASK_3_MYSHKIN_A_SHELL_SORT_SHELL_SORT_H_
#define MODULES_TASK_3_MYSHKIN_A_SHELL_SORT_SHELL_SORT_H_

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

int getRandomArray(int* buffer, int size);
int sortingCheck(int* buffer1, int* buffer2, int size);

int ShellSortSenq(int* buffer, int length);
int mergeSort(int* buffer, int size);

int parallelShellSort(int* buffer, int length);


#endif  // MODULES_TASK_3_MYSHKIN_A_SHELL_SORT_SHELL_SORT_H_
```

shell_sort.cpp

```
// Copyright 2019 Myshkin Andrew
#include <mpi.h>
#include <random>
#include <stdexcept>
#include <ctime>
#include <algorithm>
#include <iostream>
#include "../../../modules/task_3/myshkin_a_shell_sort/shell_sort.h"

int getRandomArray(int* buffer, int size) {
  if ((size <= 0) || (buffer == nullptr)) return -1;
  std::mt19937 gen;
  gen.seed(static_cast<unsigned int>(time(0)));
  for (int i = 0; i < size; i++) {
     buffer[i] = gen() % 100;
  }
  return 0;
}

int sortingCheck(int* buffer1, int* buffer2, int size) {
  if ((size <= 0) || (buffer1 == nullptr) || (buffer2 == nullptr)) return -1;
  for (int i = 1; i < size; i++) {
    if ((buffer1[i - 1] <= buffer1[i]) && (buffer2[i - 1] <= buffer2[i])) {
       i++;
    } else {
      return -1;
    }
```

```
  }

  for (int i = 0; i < size; i++) {
     if (buffer1[i] == buffer2[i]) {
        i++;
     } else {
        return -1;
     }
  }
  return 0;
}

int ShellSortSenq(int* buffer, int length) {
  if ((length <= 0) || (buffer == nullptr)) return -1;
  int i, j, temp;
  int step = length / 2;

  while (step > 0) {
    for (i = 0; i < (length - step); i++) {
      j = i;
      while (j >= 0 && buffer[j] > buffer[j + step]) {
        temp = buffer[j];
        buffer[j] = buffer[j + step];
        buffer[j + step] = temp;
        j--;
      }
    }
    step = step / 2;
  }
  return 0;
}

void ownMergeBound(int* buffer, int left, int right, int middle)
{
    if (left >= right || middle < left || middle > right) return;
    if (right == left + 1 && buffer[left] > buffer[right]) {
        int value = buffer[right];
        buffer[right] = buffer[left];
        buffer[left] = value;
        return;
    }
    int* tmp = (int*)malloc(((right - left) + 1) * sizeof(int));
    if (tmp == nullptr) return;
    for (int i = 0; i < ((right - left) + 1); i++) tmp[i] = buffer[left + i];

    for (int i = left, j = 0, k = (middle - left + 1); i <= right; ++i) {
        if (j > middle - left) {
            buffer[i] = tmp[k++];
        }
        else if (k > right - left) {
            buffer[i] = tmp[j++];
        }
```

```
        else {
            buffer[i] = (tmp[j] < tmp[k]) ? tmp[j++] : tmp[k++];
        }
    }
    if (tmp) { free(tmp); tmp = nullptr; }

    return;
}

void ownMergeSort(int *buffer, int left, int right) {
  if (left >= right) return;

  int middle = left + (right - left) / 2;

  ownMergeSort(buffer, left, middle);
  ownMergeSort(buffer, (middle + 1), right);
  ownMergeBound(buffer, left, right, middle);
  return;
}

int mergeSort(int *buffer, int size) {
  if ((buffer == nullptr) || (size < 2)) return -1;
  int left = 0, right = size - 1;

  ownMergeSort(buffer, left, right);
  return 0;
}


int parallelShellSort(int* buffer, int length) {
  int size, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int sizeBuf = length;

  if (sizeBuf <= 0) {
    return -1;
  }

  if (sizeBuf == 1) {
    return 0;
  }

  int sts = 0;
  int step = sizeBuf / 2;
  int sizeP, residue;
  int sizeProc;
  int counter = 0;

  while (step > 0) {
    residue = step % size;
```

```
if (size > step) {
  sizeP = step;
} else {
  sizeP = size;
}

sizeProc = sizeBuf / step;
int* bufForProc = (int*)(malloc(sizeProc * sizeof(int)));

counter = 0;

while (counter < step + sizeP - residue || counter <= step) {
  if (counter + residue == step) { sizeP = residue; }
  for (int proc = 0; proc < sizeP; proc++) {
    if (rank == 0) {
      for (int i = 0; i < sizeProc; i++) {
        bufForProc[i] = buffer[proc + counter + step * i];
      }
      if (proc == 0) {
        sts = mergeSort(bufForProc, sizeProc);
        if (sts == -1) {
            return -1;
        }
        for (int i = 0; i < sizeProc; i++) {
          buffer[counter + step * i] = bufForProc[i];
        }
      } else {
        MPI_Send(&bufForProc[0], sizeProc, MPI_INT, proc, 0, MPI_COMM_WORLD);
      }
    } else {
      if (proc == rank) {
        MPI_Status Status;
        MPI_Recv(&bufForProc[0], sizeProc, MPI_INT, 0, 0, MPI_COMM_WORLD, &Status);
        sts = mergeSort(bufForProc, sizeProc);
        if (sts == -1) {
            return -1;
        }
        MPI_Send(&bufForProc[0], sizeProc, MPI_INT, 0, 0, MPI_COMM_WORLD);
      }
    }
  }
  counter = counter + size;
}

if (size > step) {
  sizeP = step;
} else {
  sizeP = size;
}

counter = 0;
```

```
  while (counter < step + sizeP - residue || counter <= step) {
    if (counter + residue == step) { sizeP = residue; }
    for (int proc = 1; proc < sizeP; proc++) {
      if (rank == 0) {
        MPI_Status Status;
        MPI_Recv(&bufForProc[0], sizeProc, MPI_INT, proc, 0, MPI_COMM_WORLD,
&Status);
        for (int i = 0; i < sizeProc; i++) {
          buffer[proc + step * i + counter] = bufForProc[i];
        }
      }
    }
    counter = counter + size;
  }
  step = step / 2;
  if (bufForProc) { free(bufForProc); bufForProc = nullptr; }
 }

 return 0;
}
```

main.cpp

```
// Copyright 2019 Myshkin Andrew
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <algorithm>
#include <random>
#include <ctime>
#include "./shell_sort.h"


TEST(Parallel_Shell_Sort, Test_With_Shell_Sort) {
 int rank;
 int sts = 0;
 int sts2 = 0;

 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 const int length = 130000;
 int *tmp = reinterpret_cast<int*>(malloc(length * sizeof(int)));
 int *tmp2 = reinterpret_cast<int*>(malloc(length * sizeof(int)));

 if (rank == 0) {
  sts = getRandomArray(tmp, length);
  if (tmp && tmp2) {
    memcpy(tmp2, tmp, (sizeof(int) * length));
  }

  sts2 = ShellSortSenq(tmp2, length);
 }
```

```
  sts = parallelShellSort(tmp, length);

  if (rank == 0) {
      sts = sortingCheck(tmp, tmp2, length);
  }

  if (tmp) { free(tmp); tmp = nullptr; }
  if (tmp2) { free(tmp2); tmp2 = nullptr; }

  if (rank == 0) {
      ASSERT_EQ(sts, 0);
  }
}


TEST(Parallel_Shell_Sort, Test_With_Merge_Sort) {
    int rank;
    int sts = 0;
    int sts2 = 0;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const int length = 130000;
    int* tmp = reinterpret_cast<int*>(malloc(length * sizeof(int)));
    int* tmp2 = reinterpret_cast<int*>(malloc(length * sizeof(int)));

    if (rank == 0) {
        sts = getRandomArray(tmp, length);
        if (tmp && tmp2) {
            memcpy(tmp2, tmp, (sizeof(int) * length));
        }

        sts2 = mergeSort(tmp2, length);
    }

    sts = parallelShellSort(tmp, length);

    if (rank == 0) {
        sts = sortingCheck(tmp, tmp2, length);
    }

    if (tmp) { free(tmp); tmp = nullptr; }
    if (tmp2) { free(tmp2); tmp2 = nullptr; }

    if (rank == 0) {
        ASSERT_EQ(sts, 0);
    }
}


TEST(Parallel_Shell_Sort, Test_With_Shell_Sort_Size_One) {
```

```
    int rank;
    int sts = 0;
    int sts2 = 0;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const int length = 50000;
    int* tmp = reinterpret_cast<int*>(malloc(length * sizeof(int)));
    int* tmp2 = reinterpret_cast<int*>(malloc(length * sizeof(int)));

    if (rank == 0) {
        sts = getRandomArray(tmp, length);
        if (tmp && tmp2) {
            memcpy(tmp2, tmp, (sizeof(int) * length));
        }

        sts2 = ShellSortSenq(tmp2, length);
    }

    sts = parallelShellSort(tmp, length);

    if (rank == 0) {
        sts = sortingCheck(tmp, tmp2, length);
    }

    if (tmp) { free(tmp); tmp = nullptr; }
    if (tmp2) { free(tmp2); tmp2 = nullptr; }

    if (rank == 0) {
        ASSERT_EQ(sts, 0);
    }
}


TEST(Parallel_Shell_Sort, Test_With_Shell_Sort_Size_Two) {
    int rank;
    int sts = 0;
    int sts2 = 0;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const int length = 10000;
    int* tmp = reinterpret_cast<int*>(malloc(length * sizeof(int)));
    int* tmp2 = reinterpret_cast<int*>(malloc(length * sizeof(int)));

    if (rank == 0) {
        sts = getRandomArray(tmp, length);
        if (tmp && tmp2) {
            memcpy(tmp2, tmp, (sizeof(int) * length));
        }

        sts2 = ShellSortSenq(tmp2, length);
```

```
    }

    sts = parallelShellSort(tmp, length);

    if (rank == 0) {
        sts = sortingCheck(tmp, tmp2, length);
    }

    if (tmp) { free(tmp); tmp = nullptr; }
    if (tmp2) { free(tmp2); tmp2 = nullptr; }

    if (rank == 0) {
        ASSERT_EQ(sts, 0);
    }
}


TEST(Parallel_Shell_Sort, Test_With_Shell_Sort_Size_Three) {
    int rank;
    int sts = 0;
    int sts2 = 0;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const int length = 35687;
    int* tmp = reinterpret_cast<int*>(malloc(length * sizeof(int)));
    int* tmp2 = reinterpret_cast<int*>(malloc(length * sizeof(int)));

    if (rank == 0) {
        sts = getRandomArray(tmp, length);
        if (tmp && tmp2) {
            memcpy(tmp2, tmp, (sizeof(int) * length));
        }

        sts2 = ShellSortSenq(tmp2, length);
    }

    sts = parallelShellSort(tmp, length);

    if (rank == 0) {
        sts = sortingCheck(tmp, tmp2, length);
    }

    if (tmp) { free(tmp); tmp = nullptr; }
    if (tmp2) { free(tmp2); tmp2 = nullptr; }

    if (rank == 0) {
        ASSERT_EQ(sts, 0);
    }
}
```

```
TEST(Parallel_Shell_Sort, Test_With_Shell_Sort_Size_Zero) {
    int rank;
    int sts = 0;
    int sts2 = 0;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    const int length = 0;
    int* tmp = reinterpret_cast<int*>(malloc(length * sizeof(int)));
    int* tmp2 = reinterpret_cast<int*>(malloc(length * sizeof(int)));

    if (rank == 0) {
        sts = getRandomArray(tmp, length);
        if (tmp && tmp2) {
            memcpy(tmp2, tmp, (sizeof(int) * length));
        }

        sts2 = ShellSortSenq(tmp2, length);
    }

    sts = parallelShellSort(tmp, length);

    if (rank == 0) {
        sts = sortingCheck(tmp, tmp2, length);
    }

    if (tmp) { free(tmp); tmp = nullptr; }
    if (tmp2) { free(tmp2); tmp2 = nullptr; }

    if (rank == 0) {
        ASSERT_EQ(sts, -1);
    }
}


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();
}
```