Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Параллельные вычисление

Отчёт по теме :

Создание программы сортировки последовательности чисел при помощи MPI

Работу выполнил:
Ковальков А.В.
Группа:
3540901/91501
Преподаватель:
Стручков И.В.

Санкт-Петербург
2020

Содержание

1. Цель работы

Задание. Вариант 3. MPI: Выполнить сортировку последовательности чисел.

В данное задание входит разбор алгоритма выполнения работы, проектирование распределенного и последовательного приложения и проведение тестирования скорости их работы различными методами.

2. Программа работы

- Для алгоритма из полученного задания написать последовательную программу на языке C или C++, реализующую этот алгоритм.
- Для созданной последовательной программы необходимо написать 3-5 тестов, которые покрывают основные варианты функционирования программы. Для создания тестов можно воспользоваться механизмом Unit-тестов среды NetBeans, или описать входные тестовые данные в файлах. При использовании NetBeans необходимо в свойствах проекта установить ключ компилятора -pthread.
- Проанализировать полученный алгоритм, выделить части, которые могут быть распараллелены, разработать структуру параллельной программы. Определить количество используемых потоков, а также правила и используемые объекты синхронизации.
- Согласовать разработанную структуру и детали реализации параллельной программы с преподавателем.
- Написать код параллельной программы и проверить ее корректность на созданном ранее наборе тестов. При необходимости найти и исправить ошибки.
- Провести эксперименты для оценки времени выполнения последовательной и параллельной программ. Проанализировать полученные результаты.
- Сделать общие выводы по результатам проделанной работы: Различия между способами проектирования последовательной и параллельной реализаций алгоритма, Возможные способы выделения параллельно выполняющихся частей, Возможные правила синхронизации потоков, Сравнение времени выполнения последовательной и параллельной программ, Принципиальные ограничения повышения эффективности параллельной реализации по сравнению с последовательной.

3. Сведения о системе

- intel core i7-4700HQ 2.4 GHz, 4 ядра, 8 потоков
- RAM 16gb DDR3
- Java 1.8_255 Addopted Open JDK

4. Теоретическая информация

Message Passing Interface (MPI, интерфейс передачи сообщений) — программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. Разработан Уильямом Гроуппом, Эвином Ласком (англ.) и другими.

MPI является наиболее распространённым стандартом интерфейса обмена данными в параллельном программировании, существуют его реализации для большого числа компьютерных платформ. Используется при разработке программ для кластеров и суперкомпьютеров. Основным средством коммуникации между процессами в MPI является передача сообщений друг другу.

Стандартизацией MPI занимается MPI Forum. В стандарте MPI описан интерфейс передачи сообщений, который должен поддерживаться как на платформе, так и в приложениях пользователя. В настоящее время существует большое количество бесплатных и коммерческих реализаций MPI. Существуют реализации для языков Фортран 77/90, Java, Си и C++.

В первую очередь MPI ориентирован на системы с распределенной памятью, то есть когда затраты на передачу данных велики, в то время как OpenMP ориентирован на системы с общей памятью (многоядерные с общим кешем). Обе технологии могут использоваться совместно, чтобы оптимально использовать в кластере многоядерные системы.

5. Ход выполнения работы

5.1. Алгоритм решения

5.1.1. Общие сведения и выбор алгоритма

Алгоритмов сортировки много, и изобретать свой собственный не имеет смысла, поэтому, в первую очередь необходимо было выбрать наиболее оптимальный алгоритм сортировки.

Оптимальность заключается в двух параметрах, первый - скорость работы. Оптимальным в данном случае будет выбор алгоритма работающего за O(n log n), к таким относятся быстрая сортировка, сортировка слиянием и др. Во вторых, данный алгоритм должен быть способен к работе с несколькими потоками. Например алгоритм Шэлла и некоторые другие алгоритмы реализовать значительно трудней при условии работы нескольких потоков, а некоторые и вовсе работают не производительно.

Таким образом, выбор пал на сортировку слиянием. Так как смысл данного алгоритма разбивать массивы на подмассивы, сортировать их и соединять. То если мы изначально разделим наш массив на подмассивы соизмеримо количеству потоков, а потом соединим их. Это удобно и кажется в контексте данной задачи неплохим решением.

Сама сортировка, как уже было сказано состоит из двух ключевых этапов (методов), этап рекурсивного разбиения на подмассивы, и этап их перестановки(сортировки) и слияния. Наиболее оптимальным является рекурсивная реализация, по этому и будет использована она. Но немного адаптированная под условия задания.

Рекурсивные методы наиболее удобно создавать при помощи возврата непосредственно рекурсивного вызова, но учитывая то, что мы должны вернуть массив чисел, а не просто отсортировав вывести, то нам нужно реализовать рекурсию немного по другому, например в полях переменных (подмассивов левой и правой части базового массива).

Для реализации был выбран язык Java. Версии 1.8, и библиотека соблюдающая спецификацию MPI - MPJexpress.

Общий код алгоритма продемонстрирован на листинге ниже:

Листинг 1: MergeSort.java

```
1  package Sorting;
2
3  public class MergeSort {
4
5
6      // merge sorting
7      public static int [] sortArray(int[] arrayA){
8          if (arrayA == null || arrayA.length < 2) {
9              return arrayA;
10         }
11
12         int [] arrayB = new int[arrayA.length / 2];
13         System.arraycopy(arrayA, 0, arrayB, 0, arrayA.length / 2);
14
15         int [] arrayC = new int[arrayA.length - arrayA.length / 2];
16         System.arraycopy(arrayA, arrayA.length / 2, arrayC, 0, arrayA.length -
       ↪ arrayA.length / 2);
17
18         arrayB = sortArray(arrayB);
19         arrayC = sortArray(arrayC);
20
21         return mergeArray(arrayB, arrayC);
22     }
23
24     public static int[] mergeArray(int[] a1, int[] a2) {
25         int[] b = new int[a1.length + a2.length];
26         int positionA1 = 0;
27         int positionA2 = 0;
28
29         for(int i = 0; i < b.length; i++) {
30             if(positionA1 == a1.length){
31                 b[i] = a2[positionA2];
32                 positionA2++;
33             } else if(positionA2 == a2.length){
34                 b[i] = a1[positionA1];
35                 positionA1++;
36             } else if(a1[positionA1] < a2[positionA2]){
37                 b[i] = a1[positionA1];
38                 positionA1++;
39             } else {
40                 b[i] = a2[positionA2];
41                 positionA2++;
42             }
43         }
44         return b;
45     }
46
47     // for final merge
48     public static int[][] spliting(int [][] subArr){
49         if (subArr.length == 1) {
50             return subArr;
51         }
52
53         int[][] arr = new int[subArr.length/2][subArr[0].length * 2];
54
55         for (int i = 0, j = 0; i < subArr.length; i+=2, j++) {
56             arr[j] = mergeArray(subArr[i], subArr[i + 1]);
```

```
57          }
58
59          return spliting(arr);
60      }
61  }
```

Как мы можем увидить из листинга 1. В данном классе всего 3 метода. Первый метод - sortArray. Принимает простой массив целых чисел, разбивает его на равные части и проверяет массив на пустоту (null) и на длину в 1 элемент, так как оба варианта таких массивов являются отсортированными. Также происходит разделение на два подмассива - левого и правого, и рекурсивно вызывается данный метод для каждой из половин. При этом возвращает метод второй метод mergeArray, который как раз таки, поочередно сравнивает элементы поступивших к нему массивов, укладывает это в один массив по возрастанию значений и возвращает обратно в метод sortArray.

Последний метод - spliting , изначально задумывался для финальной сборки и упорядочивания массивов. При помощи метода mergeArray. Но так как он также рекурсивный и мало чем отличен от метода sorting. К тому же в java не лучший язык для хвостовых рекурсий на данный момент, и данный метод был бы оптимальным в случае отсутствия рекурсии, по этому данный метод на момент написания отчёта не используется, но является альтернативным вариантом финальной сборки и сортировки данных для параллельного выполнения.

5.1.2. Unit-тестирование

Все основные случаи для сортировки были протестированы при помощи фреймворка JUnit версии 4.12.

Как видно из листинга ниже. Были протестированы отсортированный и обратно отсортированный массивы, пустой, с одним элементом, с дублированием , с соседними элементами и при этом пограничным количеством элементов с нижней стороны.

Всего 7 тестов, тестирующих 9 различных ситуаций при сортировке. Листинг ниже демонстрирует тесты для заданного алгоритма.

Листинг 2: MergeSortTests.java

```
1  package tests;
2
3  import Sorting.MergeSort;
4  import org.junit.Assert;
5  import org.junit.Test;
6
7  public class MergeSortTests {
8
9      private static final int[] ORDERED =
       ↪ {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};          //
       упорядоченный
10     private static final int[] REVERS_ORDERED =
       ↪ {20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1}; //
       обратно
11     private static final int[] ALL_THE_SAME =
       ↪ {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};              // Все числа
       одинаковы
12     private static final int[] TWO_NEAR =
       ↪ {2,2,2,2,2,2,2,2,2,2,2,2,2,1,1,1,1,1,1,1,1,1,1,1};              // Два ближних
       числа
13     private static final int[] TWO_NEAR_2 = {1,2};
       ↪                          // Два ближних
       числа
14     private static final int[] TWO_NEAR_3 = {2,1};
       ↪                          // Два ближних
       числа
```

```
15      private static final int[] ONE_ELEM = {1};
        ↪                          // Один
        элемент
16      private static final int[] EMPTY_ELEM = {};
        ↪                          // Пустой
        элемент
17      private static final int[] NULL_ELEM = null;
        ↪                          // Пустой
        элемент
18
19
20      @Test
21      public void empty(){
22          Assert.assertArrayEquals(new int[]{}, EMPTY_ELEM);
23      }
24
25      @Test
26      public void nullArr(){
27          Assert.assertNull(MergeSort.sortArray(NULL_ELEM));
28      }
29      @Test
30      public void oneElement(){
31          Assert.assertArrayEquals(new int[]{1}, MergeSort.sortArray(ONE_ELEM));
32      }
33      @Test
34      public void nears(){
35          Assert.assertArrayEquals(new int
        ↪ []{1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,2,2}, MergeSort.sortArray(
        ↪ TWO_NEAR));
36          Assert.assertArrayEquals(new int[]{1,2}, MergeSort.sortArray(TWO_NEAR_2)
        ↪ );
37          Assert.assertArrayEquals(new int[]{1,2}, MergeSort.sortArray(TWO_NEAR_3)
        ↪ );
38      }
39
40      @Test
41      public void same(){
42          Assert.assertArrayEquals(new int
        ↪ []{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}, MergeSort.sortArray(
        ↪ ALL_THE_SAME));
43      }
44
45      @Test
46      public void oredered(){
47          MergeSort.sortArray(ORDERED);
48          Assert.assertArrayEquals(ORDERED, MergeSort.sortArray(ORDERED));
49      }
50      @Test
51      public void reversOrdered(){
52          Assert.assertArrayEquals(ORDERED, MergeSort.sortArray(REVERS_ORDERED));
53      }
54  }
```

5.1.3. Вспомогательные методы

Для того, чтобы можно было адекватно оценить производительность данного алгоритма, было также принято решение написать класс, способный генерировать массив заданной длинны.

Массив собирается на основе случайных чисел, в методе generate. Также существует несколько методов для работы с данным массивом. Во первых метод toSimpleArray, кото-

рый переводит массив чисел из листа (коллекции) в массив фиксированного размера.

А также методы предоставляющий массив чисел и метод разбивающий массив на подмассивы равно длинны. Подобный бинарный массив в начале использовался в паре с методом spliting листинга 1. Но так как тот метод сейчас не используется, то данный метод оставлен для просто для демонстрации, альтернативного пути исполнения.

Листинг 3: ListGenerator.java

```
1  package helpers;
2
3  import java.util.List;
4  import java.util.Random;
5  import java.util.stream.Collectors;
6  import java.util.stream.Stream;
7
8  public class ListGenerator {
9      private int[] ints;
10
11     public ListGenerator(int c) {
12         ints = toSimpleArray(generate(c));
13     }
14
15     // generate list of random unsigned int from 0 to 100
16     private List<Integer> generate(int c){
17         return Stream.generate(() -> new Random(47).nextInt(100)).limit(c).
       ↪ collect(Collectors.toList());
18     }
19
20     // transform list to int[]
21     private int[] toSimpleArray(List<Integer> list){
22         ints = new int[list.size()];
23         for (int i = 0; i < list.size(); i++) {
24             ints[i] = list.get(i);
25         }
26         return ints;
27     }
28
29     public int[] getInts() {
30         return ints;
31     }
32
33     // divide one array into equal parts
34     public static int[][] getRankedInts(int worldSize, int[] ints){
35         int rankSize = ints.length/worldSize;
36         int[][] ints1 = new int[worldSize][rankSize];
37         for (int i =0, x = 0; i < worldSize; i++, x+=rankSize) {
38             if (rankSize >= 0) System.arraycopy(ints, x, ints1[i], 0, rankSize);
39         }
40         return ints1;
41     }
42 }
```

5.1.4. Последовательная реализация

Последовательная генерация выполняется крайне просто. В начале и в конце метода снимаются показатели о текущем времени(8,15), и в середине создаётся массив заданной длинны (10) и вызывается метод его сортировки (11).

Листинг 4: Serial.java

```java
import Sorting.MergeSort;
import helpers.ListGenerator;

import java.util.Arrays;

public class Serial {
    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();

        int[] shuf = new ListGenerator(40_000_000).getInts();
        int[] sort = MergeSort.sortArray(shuf);
        System.out.printf("Shuffled: %s \n",Arrays.toString(shuf));
        System.out.printf("Sorted  : %s \n", Arrays.toString(sort));

        long finishTime = System.currentTimeMillis();
        System.out.printf("Time is : %d ms", finishTime - startTime);
    }
}
```

5.1.5. Параллельная реализация

Параллельная реализация намного более многословна. Так как по мимо описания и реализации специфичных для MPI методов, необходимо было позаботится о финальной сортировке данных.

Код параллельной реализации продемонстрирован ниже:

Листинг 5: Parallel.java

```java
import Sorting.MergeSort;
import helpers.ListGenerator;
import mpi.MPI;
import java.util.Arrays;
public class Parallel {
    public static void main(String[] args) {
        long timeStart= System.currentTimeMillis();

        // Init MPI
        MPI.Init(args);
        // rank - number of process
        // size - quantities of process
        int rank = MPI.COMM_WORLD.Rank(), size = MPI.COMM_WORLD.Size(), root =
    ↪ 0;
        // length - length of array which we wanna sort
        // unitSize - length of one process
        int length = 40_000_000, unitSize = length / size;
        // send - what we send from root to all
        // recv -what we receive from root
        // newrecv - what we receive in root from all
        int[] sendBuffer = new int[length], recvBuffer = new int[unitSize],
    ↪ newRecBuffer = new int[length];


        if(rank == root){
            // generate random array
            sendBuffer = new ListGenerator(length).getInts();
            System.out.printf("Shuffled: %s \n", Arrays.toString(sendBuffer));
        }

        // sending parts of sendBuffer to all process
```

```
30          MPI.COMM_WORLD.Scatter(sendBuffer, 0, unitSize, MPI.INT, recvBuffer, 0,
        ↪ unitSize, MPI.INT, root);
31
32          // sort in each process
33          recvBuffer = MergeSort.sortArray(recvBuffer);
34
35          // assembly all parts to one array
36          MPI.COMM_WORLD.Gather(recvBuffer, 0, unitSize, MPI.INT, newRecBuffer, 0,
        ↪  unitSize, MPI.INT, root);
37
38          if (rank == root) {
39              // finally merge
40  // int[][] result = Sorting.MergeSort.spliting(helpers.ListGenerator.getRankedInts(size, newRecBuffer));
41  // System.out.printf("Sorted :System.out.printf("Sorted :
42              // check time
43              long timeStop = System.currentTimeMillis();
44              System.out.printf("Time is: %d ms", timeStop - timeStart);
45          }
46          // finalization for garbage collector
47          MPI.Finalize();
48      }
49  }
```

Данная программа также как и последовательная начинается и заканчивается показанием текущего времени, для оценки скорости работы программы.

Далее происходит инициализация MPI при помощи метода Init, затем создание переменных необходимых для управления потоками, size - общее количество потоков, rank - номер текущего процесса от 0 до n-1, а также мы сразу объявляем о нашем коревом процессе (root), по умолчанию корневым процессом мы считаем процесс с rank = 0.

Затем мы инициализируем переменные, говорящие о объемах массивов данных. length - говорит об общей длине сгенирированных данных, а unitSize - говорит о том, сколько элементов будет включать в себя подмассив у каждого потока. Очевидно, что данный подход не защищает от нечётного числа потоков, но в пользу упрощения логики было принято решение оставить данное решение, так как планировалось использовать четное количество потоков.

Последним этапом создания переменных является создание буферов приема и передачи данных. sendBuffer - буфер, который отправит корневой процесс всем остальным, recvBuffer - то что примут процессы, newRecvBuffer - то что примет от потоков обратно корневой процесс.

Следующим этапом, на строках 23-27 мы создаем массив нужно нам длинны в корневом процессе.

Затем при помощи Scatter (основной метод коллективного обмена данными в mpi), мы делим исходный массив на равные части и раздаём его части нашим потокам.

Далее производим сортировку во всех потоках сразу. А затем используем метод Gather, который обратен методу Scatter (передает данные от всех процессов к корневому соединяя их в один массив).

Затем, мы производим финальную сортировку массива в корневом процессе. И закрываем MPI при помощи Finalize, вызывая сборщик мусора.

5.2. Эксперименты

5.2.1. Вариативность скорости работы от величины исходных данных

5.2.2. Оценка результатов прогонов при помощи статистических показателей

Рисунок 5.1. Зависимость времени выполнения от длинны массива

6. Заключение

В результате выполнения данной работы был произведён анализ различных алгоритмов сортировки, а также реализация последовательной и параллельной программы с использованием MPI.

Сортировка была протестирована модульными тестами, а также были произведены многократные замеры при различных объёмах данных. Для каждого размера массива, каждая реализация была запущенна 50 раз, с фиксацией времени выполнения.

В ходе экспериментов, было выяснено, что параллельный вариант стоит использовать при достаточно больших размерах входных данных, так как много ресурсов будет затрачено на само распаралелливание потоков и программа выйдет существенно дольше последовательно. При этом нужно брать в учёт специфику работы JVM. Так как выяснилось, что со слишком большим размером входных данных происходило переполнение кучи данных, при том что последовательная реализация справлялась с таким же объёмом успешно.