作业 2 报告

韩云飞 SA13226297

1、 Please write a parallel program which uses the divide-and-conquer idea. You are allowed to use the program taught in the class, but to solve a new problem using the divide-and-conquer idea is highly recommended.

以下代码用 ForkJoin 框架实现了快速排序，需要说明的是，当 数组长度小于 THRESHOLD 时，会调用 sequentiallySort，不再做并行化。

//ForkJoinDemo.java :

```
import java.util.Arrays;
import java.util.Random;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

class SortTask extends RecursiveAction {
        private static final long serialVersionUID = 1L;
        final long[] array;
        final int lo;
        final int hi;
        private int THRESHOLD = 200;

        public SortTask(long[] array) {
            this.array = array;
            this.lo = 0;
            this.hi = array.length - 1;
         }

        public SortTask(long[] array, int lo, int hi) {
             this.array = array;
            this.lo = lo;
            this.hi = hi;
        }

        protected void compute() {
             if (lo - hi < THRESHOLD)
                 sequentiallySort(array, lo, hi);
             else {
                 int pivot = partition(array, lo,  hi);
                 new SortTask(array, lo, pivot - 1).fork();
                 new SortTask(array, pivot + 1, hi).fork();
              }
          }

        private int partition(long[] array, int lo, int hi) {
```

```
                long x = array[hi];
                int i = lo - 1;
                for (int j = lo; j < hi; j++) {
                if (array[j] <= x) {
                        i++;
                        swap(array, i, j);
                 }
         }
        swap(array, i + 1, hi);
          return i + 1;
          }

          private void swap(long[] array, int i, int j) {
         if (i != j) {
          long temp = array[i];
                array[i] = array[j];
                array[j] = temp;
         }
         }

   private void sequentiallySort(long[] array, int lo, int hi) {
         Arrays.sort(array, lo, hi + 1);
          }
}

public class ForkJoinDemo {
         public static void main(String[] args) throws
InterruptedException {
                  ForkJoinPool forkJoinPool = new ForkJoinPool();
                    Random rnd = new Random();
                    final int SIZE = 15000;
                    long[] array = new long[SIZE];
                    for (int i = 0; i < SIZE; i++) {
                       array[i] = rnd.nextInt();
                    }
                    forkJoinPool.submit(new SortTask(array));
                    forkJoinPool.shutdown();
                    forkJoinPool.awaitTermination(50, TimeUnit.SECONDS);

                    for (int i = 1; i < SIZE; i++) {
                       assert(array[i - 1] < array[i]);
                    }
   }

}
```

2、 A sequential program for calculating Pi is given as below (It is downloaded from http://blog.csdn.net/xuxian361/article/details/8130948). Please rewrite it into a concurrent program, and you can use either the parallel idea or the concurrent idea learned in the class. Then try to compare the efficiency between the original sequential program and your concurrent programs when the input "n" is the same. In addition to the source code, a report for the comparison is required.

//Pi.java， NaivePI 为串行方法，MultiPi 为并行方法

```
import java.util.Scanner;
public class  Pi {
        public static class NaivePi {
                static double MontePI(long n) {
                        double PI;
                        double x, y;
                        long  sum = 0;
                        for (long i = 1; i < n; i++) {
                                x = Math.random();
                                y = Math.random();
                                if ((x * x + y * y) <= 1) {
                                        sum++;
                                }
                        }
                        PI = 4.0 * sum / n;
                        return PI;
                }
        }

        class MultiPi extends Thread {
                long n;
           int sum = 0;
           public MultiPi(long n) {
                this.n = n;
           }
                public void run() {
             double x, y;
             for (int i = 1; i < n; i++) {
                x = Math.random();
                y = Math.random();
                if ((x * x + y * y) <= 1) {
                        sum++;
                }
             }
          }
        }
```

```
    void  Compare(long n, int chunk) throws InterruptedException {
            double PI;
            long start = System.currentTimeMillis();
            PI = NaivePi.MontePI(n);
            long end = System.currentTimeMillis();
            System.out.println("NaivePi: "+PI  + "\t Time: " + (end -
                    start));

            start = System.currentTimeMillis();

            long size = n/chunk;
            MultiPi[] mp = new MultiPi[chunk];
            start = System.currentTimeMillis();
            for (int i =0; i < chunk; i++) {
                    mp[i] = new MultiPi(size);
                    mp[i].start();
            }
            int sum = 0;
          for(int i=0; i < chunk; i++) {
                mp[i].join();
                sum += mp[i].sum;
            }
          PI = 4.0*sum/n;
          end = System.currentTimeMillis();
          System.out.println("MultiPi: "+PI  + "\t Time: " + (end - start));

        }

  public static void main(String [] args) {
            System.out.println("蒙特卡洛概率算法计算圆周率:");
            Scanner input = new Scanner(System.in);
            System.out.println("输入点的数量：");
            long n = input.nextLong();
            input.close();
            Pi p = new Pi();
            try {
                    p.Compare(n,  4);
            } catch (InterruptedException e) {
                    e.printStackTrace();
            }
  }
}
```

对比结果为：

并行程序的运行时间反而比串行要长，一个原因是计算 Pi 的过程而且为 CPU 密集的计算 ，时间开销比较小，所以并行程序在线程的创建与管理上反而多花了时间; 另一个原因为运行程序的电脑为双核，并行的优势发挥不明显。

3．Running your application on *n* processors yields a speed up of S_n. Use Amdahl's Law to derive a formula for S_m, the speedup on *m* processors, in terms of *m* and S_n.

设可并行部分的比例为 p, 由 Amdahl's Law 知：

$S_n = 1 / ((1-p) + p/n) \quad => \quad p = n/(n-1) *(1 – 1/S_n)$

$S_m = 1 / ((1-p) + p/m)$，代入 p 可得：

$S_m = (n-1)/n * 1/((m-1)/m * 1/S_n + 1/m)$