**Program Structures and Algorithms**

**Assignment-5**

**Summer-2022**

**Dimpleben Kanjibhai Patel – 002965372**

**Task-1:**

Please see the presentation on *Assignment on Parallel Sorting* under the *Exams. etc.*module.

Your task is to implement a parallel sorting algorithm such that each partition of the array is sorted in parallel. You will consider two different schemes for deciding whether to sort in parallel.

1. A cutoff (defaults to, say, 1000) which you will update according to the first argument in the command line when running. It's your job to experiment and come up with a good value for this cutoff. If there are fewer elements to sort than the cutoff, then you should use the system sort instead.
2. Recursion depth or the number of available threads. Using this determination, you might decide on an ideal number (*t*) of separate threads (stick to powers of 2) and arrange for that number of partitions to be parallelized (by preventing recursion after the depth of *lg t* is reached).
3. An appropriate combination of these.

There is a *Main* class and the *ParSort* class in the *sort.par* package of the INFO6205 repository. The *Main* class can be used as is but the *ParSort* class needs to be implemented where you see "TODO..." [it turns out that these TODOs are already implemented].

Unless you have a good reason not to, you should just go along with the Java8-style future implementations provided for you in the class repository.

You must prepare a report that shows the results of your experiments and draws a conclusion (or more) about the efficacy of this method of parallelizing sort. Your experiments should involve sorting arrays of sufficient size for the parallel sort to make a difference. You should run with many different array sizes (they must be sufficiently large to make parallel sorting worthwhile, obviously) and different cutoff schemes.

For varying the number of threads available, you might want to consult the following resources:

**Code:**

|  |
| --- |
| public static void main(String[] args) {  *processArgs*(args); *// System.out.println("Degree of parallelism: " + ForkJoinPool.getCommonPoolParallelism());* Random random = new Random();  for(int size = 1000000; size < 16000000; size \*= 2) {  int[] array = new int[size];  ArrayList<Long> timeList = new ArrayList<>();  System.*out*.println("Array Size :" + size);  ParSort.*thread\_count* = 1;  while (ParSort.*thread\_count* <= 16) {  ParSort.*thread\_Pool* = new ForkJoinPool(ParSort.*thread\_count*);  System.*out*.println("Degree of parallelism: " + ParSort.*thread\_Pool*.getParallelism());  for (int j = 0; j < 20; j++) { *// ParSort.cutoff = 10000 \* (j + 1);* ParSort.*cutoff* = size / 20 \* (j + 1);  *// for (int i = 0; i < array.length; i++) array[i] = random.nextInt(10000000);* long time;  long startTime = System.*currentTimeMillis*();  for (int t = 0; t < 10; t++) {  for (int i = 0; i < array.length; i++) array[i] = random.nextInt(10000000);  ParSort.*sort*(array, 0, array.length);  }  long endTime = System.*currentTimeMillis*();  time = (endTime - startTime);  timeList.add(time);    System.*out*.println("cutoff:" + (ParSort.*cutoff*) + "\t\t10times Time:" + time + "ms");   }  try {  FileOutputStream fis = new FileOutputStream("./src/result" + size + "-" + ParSort.*thread\_count* + ".csv");  OutputStreamWriter isr = new OutputStreamWriter(fis);  BufferedWriter bw = new BufferedWriter(isr);  int j = 0;  for (long i : timeList) {  String content = (double) size / 20 \* (j + 1) + "," + (double) i / 10 + "\n";  j++;  bw.write(content);  bw.flush();  }  bw.close();   } catch (IOException e) {  e.printStackTrace();  }  ParSort.*thread\_count* \*= 2;  }  }   } |

Parsort.java

|  |
| --- |
| public static int *cutoff* = 1000;  public static int *thread\_count* = 2;  public static ForkJoinPool *thread\_Pool* = new ForkJoinPool(*thread\_count*);   public static void sort(int[] array, int from, int to) {  if (to - from < *cutoff*) Arrays.*sort*(array, from, to);  else {  *// FIXME next few lines should be removed from public repo.* CompletableFuture<int[]> parsort1 = *parsort*(array, from, from + (to - from) / 2); *// TO IMPLEMENT* CompletableFuture<int[]> parsort2 = *parsort*(array, from + (to - from) / 2, to); *// TO IMPLEMENT* CompletableFuture<int[]> parsort = parsort1.thenCombine(parsort2, (xs1, xs2) -> {  int[] result = new int[xs1.length + xs2.length];  *// TO IMPLEMENT* int i = 0;  int j = 0;  for (int k = 0; k < result.length; k++) {  if (i >= xs1.length) {  result[k] = xs2[j++];  } else if (j >= xs2.length) {  result[k] = xs1[i++];  } else if (xs2[j] < xs1[i]) {  result[k] = xs2[j++];  } else {  result[k] = xs1[i++];  }  }  return result;  });   parsort.whenComplete((result, throwable) -> System.*arraycopy*(result, 0, array, from, result.length)); *// System.out.println("# threads: "+ ForkJoinPool.commonPool().getRunningThreadCount());* parsort.join();  }  }   private static CompletableFuture<int[]> parsort(int[] array, int from, int to) {  return CompletableFuture.*supplyAsync*(  () -> {  int[] result = new int[to - from];  *// TO IMPLEMENT* System.*arraycopy*(array, from, result, 0, result.length);  *sort*(result, 0, to - from);  return result;  },*thread\_Pool* );  } |

**Comparison Graph:**

Graph for array size = 1000000, and Threads value= 1,2,4,8,16

Table

Description automatically generated with medium confidence

Chart, line chart

Description automatically generated

Graph for array size = 2000000, and Threads value= 1,2,4,8,16

Table

Description automatically generated

Chart, line chart

Description automatically generated

Graph for array size = 4000000, and Threads value= 1,2,4,8,16

Table

Description automatically generated

Chart, line chart

Description automatically generated

Graph for array size = 8000000, and Threads value= 1,2,4,8,16

Table

Description automatically generated

Chart, line chart

Description automatically generated

**Conclusion :**

• From the above simulation, it can be inferred that multi-threading is better than single thread.

• For 2 Threads, good cutoff value is at approximately 50% of the array size and for 4 Threads, good cutoff value is at approximately 25% of the array size.

• Thus, At my observation, Good cutoff can be obtained at

cutoff = Array Size / Number of Threads

• In my laptop(Windows i5 16GB RAM) best result is obtained for the number of threads = 4