Program Structures and Algorithms

Spring 2023(SEC –08)

NAME: Priyal Vimal Gudhka

NUID: 002747680

# Assignment-5 Parallel Sorting

**Task:** 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).

**Code of Main.java Class:**

package edu.neu.coe.info6205.sort.par;  
  
import java.io.BufferedWriter;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.io.OutputStreamWriter;  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Random;  
import java.util.concurrent.ForkJoinPool;  
import java.util.HashMap;  
import java.util.Map;  
  
*/\*\*  
 \* This code has been fleshed out by Ziyao Qiao. Thanks very much.  
 \* CONSIDER tidy it up a bit.  
 \*/*public class Main {  
  
 public static void main(String[] args) {  
 *processArgs*(args);  
// System.out.println("Degree of parallelism: " + ForkJoinPool.getCommonPoolParallelism());  
 Random random = new Random();  
 for(int arr = 1000000; arr < 18000000; arr \*= 2) {  
 System.*out*.println("Length of array that needs to be sorted is :" + arr);  
 int[] array = new int[arr];  
 List<Long> timeList = new ArrayList<>();  
 ParSort.*threadCount* = 1;  
 while (ParSort.*threadCount* <= 16) {  
 ParSort.*threadPool* = new ForkJoinPool(ParSort.*threadCount*);  
 System.*out*.println("Degree of parallelism: " + ParSort.*threadPool*.getParallelism());  
 for (int j = 0; j < 20; j++) {  
// ParSort.cutoff = 10000 \* (j + 1);  
 ParSort.*cutoff* = arr / 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");  
  
 }  
 System.*out*.println("\n");  
  
 try {  
 FileOutputStream fis = new FileOutputStream("./src/results/result" + arr + "\_" + ParSort.*threadCount* + ".csv");  
 OutputStreamWriter isr = new OutputStreamWriter(fis);  
 BufferedWriter bw = new BufferedWriter(isr);  
 int j = 0;  
 for (long i : timeList) {  
 String content = (double) arr / 20 \* (j + 1) + "," + (double) i / 10 + "\n";  
 j++;  
 bw.write(content);  
 bw.flush();  
 }  
 bw.close();  
  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 ParSort.*threadCount* \*= 2;  
 }  
 }  
  
 }  
  
 private static void processArgs(String[] args) {  
 String[] xs = args;  
 while (xs.length > 0)  
 if (xs[0].startsWith("-")) xs = *processArg*(xs);  
 }  
  
 private static String[] processArg(String[] xs) {  
 String[] result = new String[0];  
 System.*arraycopy*(xs, 2, result, 0, xs.length - 2);  
 *processCommand*(xs[0], xs[1]);  
 return result;  
 }  
  
 private static void processCommand(String x, String y) {  
 if (x.equalsIgnoreCase("N")) *setConfig*(x, Integer.*parseInt*(y));  
 else  
 // *TODO sort this out* if (x.equalsIgnoreCase("P")) //noinspection ResultOfMethodCallIgnored  
 ForkJoinPool.*getCommonPoolParallelism*();  
 }  
  
 private static void setConfig(String x, int i) {  
 *configuration*.put(x, i);  
 }  
  
 @SuppressWarnings("MismatchedQueryAndUpdateOfCollection")  
 private static final Map<String, Integer> *configuration* = new HashMap<>();  
  
  
}

**Code of ParSort.java Class:**

package edu.neu.coe.info6205.sort.par;  
  
import java.util.Arrays;  
import java.util.concurrent.CompletableFuture;  
import java.util.concurrent.ForkJoinPool;  
  
*/\*\*  
 \* This code has been fleshed out by Ziyao Qiao. Thanks very much.  
 \* CONSIDER tidy it up a bit.  
 \*/*class ParSort {  
  
 public static int *cutoff* = 1000;  
 public static int *threadCount* = 3;  
 public static ForkJoinPool *threadPool* = new ForkJoinPool(*threadCount*);  
  
  
 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;  
 },*threadPool* );  
 }  
}

**Evidence to Support Conclusion:** Various graphs are plotted for different array lengths that compares the time taken to sort the array with respective to the threads and cutoff value

* **Tabular & Graphical representation for array of size 1000000**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Array Length of 1000000** | | | | | |
| **Cut-Off Value** | **Thread Count 1** | **Thread Count 2** | **Thread Count 4** | **Thread Count 8** | **Thread Count 16** |
| 50000 | 565 | 355 | 368 | 372 | 345 |
| 100000 | 387 | 340 | 305 | 284 | 280 |
| 150000 | 471 | 380 | 313 | 279 | 286 |
| 200000 | 476 | 391 | 316 | 278 | 278 |
| 250000 | 473 | 388 | 315 | 281 | 281 |
| 300000 | 581 | 463 | 299 | 298 | 299 |
| 350000 | 575 | 422 | 301 | 297 | 298 |
| 400000 | 615 | 423 | 299 | 298 | 299 |
| 450000 | 619 | 418 | 296 | 303 | 299 |
| 500000 | 605 | 415 | 298 | 298 | 297 |
| 550000 | 645 | 405 | 402 | 399 | 403 |
| 600000 | 640 | 410 | 399 | 400 | 399 |
| 650000 | 641 | 402 | 401 | 401 | 401 |
| 700000 | 643 | 401 | 401 | 406 | 402 |
| 750000 | 642 | 402 | 402 | 406 | 399 |
| 800000 | 641 | 398 | 399 | 423 | 400 |
| 850000 | 650 | 399 | 406 | 401 | 402 |
| 900000 | 643 | 398 | 400 | 400 | 400 |
| 950000 | 641 | 401 | 401 | 399 | 401 |
| 1000000 | 644 | 399 | 398 | 399 | 399 |

Chart, line chart

Description automatically generated

* **Tabular & Graphical representation for array of size 2000000**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Array Length of 2000000** | | | | | |
| **Cut-Off Value** | **Thread Count 1** | **Thread Count 2** | **Thread Count 4** | **Thread Count 8** | **Thread Count 16** |
| 100000 | 851 | 734 | 759 | 719 | 774 |
| 200000 | 789 | 689 | 620 | 575 | 594 |
| 300000 | 973 | 782 | 647 | 560 | 560 |
| 400000 | 990 | 786 | 644 | 559 | 568 |
| 500000 | 977 | 770 | 641 | 567 | 557 |
| 600000 | 1332 | 872 | 613 | 612 | 615 |
| 700000 | 1304 | 866 | 611 | 614 | 613 |
| 800000 | 1257 | 859 | 627 | 610 | 612 |
| 900000 | 1269 | 866 | 614 | 616 | 663 |
| 1000000 | 1307 | 864 | 613 | 613 | 699 |
| 1100000 | 1347 | 841 | 830 | 828 | 832 |
| 1200000 | 1346 | 831 | 831 | 828 | 828 |
| 1300000 | 1349 | 833 | 826 | 831 | 830 |
| 1400000 | 1342 | 830 | 828 | 830 | 830 |
| 1500000 | 1345 | 830 | 831 | 828 | 829 |
| 1600000 | 1341 | 830 | 829 | 830 | 830 |
| 1700000 | 1343 | 832 | 832 | 830 | 829 |
| 1800000 | 1346 | 828 | 833 | 833 | 829 |
| 1900000 | 1343 | 832 | 829 | 829 | 827 |
| 2000000 | 1337 | 831 | 827 | 830 | 829 |

Chart, line chart

Description automatically generated

* **Tabular & Graphical representation for array of size 4000000**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Array Length of 4000000** | | | | | |
| **Cut-Off Value** | **Thread Count 1** | **Thread Count 2** | **Thread Count 4** | **Thread Count 8** | **Thread Count 16** |
| 200000 | 1702 | 1647 | 1506 | 1575 | 1515 |
| 400000 | 1594 | 1438 | 1254 | 1150 | 1174 |
| 600000 | 2040 | 1584 | 1312 | 1125 | 1120 |
| 800000 | 2025 | 1643 | 1299 | 1133 | 1111 |
| 1000000 | 2048 | 1607 | 1298 | 1119 | 1119 |
| 1200000 | 2555 | 1799 | 1249 | 1249 | 1281 |
| 1400000 | 2717 | 1800 | 1257 | 1246 | 1256 |
| 1600000 | 2658 | 1786 | 1248 | 1249 | 1248 |
| 1800000 | 2590 | 1784 | 1249 | 1245 | 1245 |
| 2000000 | 2617 | 1824 | 1253 | 1255 | 1248 |
| 2200000 | 2825 | 1725 | 1731 | 1736 | 1711 |
| 2400000 | 2814 | 1722 | 1722 | 1874 | 1725 |
| 2600000 | 2797 | 1720 | 1742 | 1713 | 1721 |
| 2800000 | 2798 | 1713 | 1723 | 1726 | 1721 |
| 3000000 | 2794 | 1757 | 1865 | 1721 | 1716 |
| 3200000 | 2801 | 1724 | 1718 | 1718 | 1732 |
| 3400000 | 2806 | 1724 | 1721 | 1734 | 1719 |
| 3600000 | 2824 | 1721 | 1721 | 1715 | 1731 |
| 3800000 | 2789 | 1721 | 1722 | 1723 | 1728 |
| 4000000 | 2794 | 1718 | 1730 | 1723 | 1719 |

Chart, line chart

Description automatically generated

* **Tabular & Graphical representation for array of size 8000000**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Array Length of 8000000** | | | | | |
| **Cut-Off Value** | **Thread Count 1** | **Thread Count 2** | **Thread Count 4** | **Thread Count 8** | **Thread Count 16** |
| 400000 | 3354 | 3268 | 3504 | 3033 | 3091 |
| 800000 | 3303 | 2940 | 2555 | 2354 | 2461 |
| 1200000 | 4283 | 3265 | 2693 | 2259 | 2308 |
| 1600000 | 4136 | 3245 | 2692 | 2247 | 2335 |
| 2000000 | 4210 | 3344 | 2699 | 2895 | 2320 |
| 2400000 | 5408 | 3742 | 2555 | 4015 | 2572 |
| 2800000 | 5318 | 3699 | 2555 | 27471 | 2560 |
| 3200000 | 5422 | 3755 | 2559 | 3591 | 2555 |
| 3600000 | 5522 | 3737 | 2554 | 3207 | 2612 |
| 4000000 | 4989 | 3745 | 2557 | 3020 | 2553 |
| 4400000 | 5865 | 3558 | 3545 | 4625 | 3543 |
| 4800000 | 5821 | 3550 | 3544 | 4117 | 3547 |
| 5200000 | 5820 | 3545 | 3545 | 3617 | 3585 |
| 5600000 | 5809 | 3549 | 3540 | 3542 | 3569 |
| 6000000 | 5811 | 3551 | 3564 | 3549 | 3593 |
| 6400000 | 5846 | 3532 | 3530 | 3544 | 3573 |
| 6800000 | 5821 | 3554 | 3552 | 3553 | 3548 |
| 7200000 | 5809 | 3557 | 3545 | 3543 | 3567 |
| 7600000 | 5813 | 3540 | 3566 | 3540 | 3549 |
| 8000000 | 5803 | 3554 | 3563 | 3555 | 3564 |

Chart, line chart

Description automatically generated

* **Tabular & Graphical representation for array of size 16000000**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Array Length of 16000000** | | | | | |
| **Cut-Off Value** | **Thread Count 1** | **Thread Count 2** | **Thread Count 4** | **Thread Count 8** | **Thread Count 16** |
| 800000 | 6752 | 7151 | 6039 | 6001 | 6078 |
| 1600000 | 6854 | 6000 | 5200 | 4799 | 5048 |
| 2400000 | 8714 | 6891 | 5734 | 4542 | 4773 |
| 3200000 | 8607 | 6701 | 5514 | 4617 | 4641 |
| 4000000 | 8701 | 6769 | 5517 | 4659 | 4635 |
| 4800000 | 11533 | 7710 | 5213 | 5366 | 5380 |
| 5600000 | 11729 | 7576 | 5238 | 5261 | 5342 |
| 6400000 | 11758 | 7655 | 5226 | 5260 | 5545 |
| 7200000 | 11256 | 7614 | 5259 | 5348 | 5523 |
| 8000000 | 11116 | 7602 | 5236 | 5266 | 5364 |
| 8800000 | 12374 | 7415 | 7479 | 7527 | 7559 |
| 9600000 | 12204 | 7339 | 7620 | 7519 | 7528 |
| 10400000 | 12546 | 7485 | 7529 | 7817 | 7570 |
| 11200000 | 12482 | 7433 | 7347 | 7727 | 7609 |
| 12000000 | 12359 | 7432 | 7507 | 7426 | 7363 |
| 12800000 | 12357 | 7348 | 7474 | 7660 | 7284 |
| 13600000 | 12175 | 7428 | 7576 | 7713 | 7571 |
| 14400000 | 12460 | 7476 | 7320 | 10517 | 7545 |
| 15200000 | 12140 | 7323 | 7713 | 10144 | 7626 |
| 16000000 | 12234 | 7899 | 7903 | 7951 | 7506 |

Chart, line chart

Description automatically generated

**Conclusion:** From the above graphs plotted by considering arrays with different length it is evident for time taken to sort with single thread is much more in comparison to multithreads. So, it can be concluded that multi-threading is better for sorting larger arrays in comparison to single thread.

Also, it can be observed that when thread count is 2 the ideal cutoff value is approximately at 50% of the array size and similarly when thread count is 4 the ideal cutoff is approximately at 25% of the array size

Therefore, the ideal cutoff value is approximately equal to array size / number of threads used to sort the array.

Looking at the graphs, the ideal value of threads required to get the best output seems to be 4