Multithreaded Sort

Report by Vincent Chan, masc0264

Prewords

Unfortunately, I forgot that multithreading makes use of cores, so I ran tests on a **dual core** (Intel Core 2 Duo) machine, which may give confusing results. Let me know if you would rather me run tests on an 8 core machine and redo this report.

The Program

The program written is a multithreaded sorter that will accept and array and a number of threads, and sort the array using that many threads. It will use shell sort as it's main sort mechanism, with merge sort to merge the threads into a new array. Since the algorithm uses merge sort, our sort will no longer be in place, nor will it be stable.

ThreadedSorter.java

```
1  /* Vincent Chan
2   * masc0264
3   */
4
5  /*ThreadedSorter
6   * This program will sort an array using the specified number of threads
7   * It will do so using shell sort and merge sort.
8   * This program is NOT in place or stable.
9   * To use:
10  * array = ThreadedSorter.shellSort(toSortArray, threads)
11  */
12
13 public class ThreadedSorter{
14   /* Class ThreadedSorter
15    * === PUBLIC ===
16    * shellSort(E[], threads) //Ln.29
17    * === PRIVATE ===
18    * sort(E[])               //Ln.74
19    * merge(E[])              //Ln.102
20    * getMin(E[])             //Ln.113
21    *
22    * === CLASSES ===
23    * WorkerThread            //Ln.134
24    */
25
26   private static int[] indices;
27   private static int threadCount;
28
29   //This is the function that is called for sorting.
30   //It will sort using the array and threads specified.
31   public static <E> E[] shellSort(E[] array, int threads) {
32     //This will calculate the chunk to be used for each array.
33     //if the number of threads exceeds half the array length (chunk=2),
34     //or exceeds the max number of threads, we will use a different number of
35 threads.
36     if(threads>array.length) threads = array.length/2;
37     if(threads>256) threads=256;
38     else if(threads<=0) threads=1;
39
40     //This will initialize the thread count and indices for use
41     //in the merge() function.
42     threadCount = threads;
43     WorkerThread[] threadArray = new WorkerThread[threads];
44     indices = new int[threads*2];
45
46     //This will create the threads and start them.
47     //They will also record the indices they start at for use
48     //in the merge() function.
```

```
49      int arrayChunk = array.length/threads;
50      int first = 0, last = arrayChunk, curIndex = 0;
51      for(int i=0; i<threads; i++) {
52        if(i==threads-1) last = array.length-1;
53        threadArray[i] = new WorkerThread<E>(array, first, last);
54        threadArray[i].start();
55        indices[curIndex++] = first;
56        indices[curIndex++] = last;
57        first = last+1;
58        last += arrayChunk;
59      }
60
61      //Waiting for the threads to join
62      //and the merge to finish
63      try{
64        for(int i=0; i<threads; i++) threadArray[i].join();
65      }
66      catch(Exception e) {
67        System.out.println("ERROR, " + 2);
68      }
69
70      //This will return the merged array
71      merge(array);
72      return array;
73    } //End shellSort()
74
75   //This will sort the array via shell sort.
76    private static <E> void sort(E[] array, int start, int last) {
77      int toSort, current;
78      int gap = 1;
79      int length = last-start+1;
80      E temp;
81
82      //Calculate the gaps using the Knuth's sequence
83      while(gap<=length/3) gap = gap*3+1;
84
85      //Sort the array
86      while(gap>0) {
87        //Sort with the given gap
88        for(toSort=start+gap; toSort<last+1; toSort++) {
89          temp = array[toSort];
90          current = toSort;
91          //Shift the array over until we find the place to insert the temp
92          while(current>start+gap-1 && ((Comparable<E>)temp).compareTo(array[cur
93   rent-gap])<=0) {
94            array[current] = array[current-gap];
95            current -= gap;
96          }
97          array[current] = temp;
98        }
99        //Reduce the gap and resort using the new gap.
100       gap = (gap-1)/3;
101     }
102   } //End sort()
103
104   //This will merge the arrays.
105   private static <E> void merge(E[] array) {
106     E[] aux = (E[])new Object[array.length];
107     for(int i=0; i<array.length; i++) {
108       aux[i] = getMin(array);
109     }
110     for(int i=0; i<array.length; i++) {
111       array[i] = aux[i];
112     }
113   } //End merge()
114
115   //This will return the minimum of the all the threads
116   //This is used for merging at the end of the sort
```

```
117   private static <E> E getMin(E[] array) {
118     E minimum = null;
119     int index = -1;
120     for(int i=0;i<threadCount;i++) {
121       if(indices[i*2]>indices[i*2+1]) continue;
122       if(minimum==null) {
123         minimum = array[indices[i*2]];
124         index = i*2;
125         continue;
126       }
127       if(((Comparable<E>)minimum).compareTo(array[indices[i*2]])>0) {
128         minimum = array[indices[i*2]];
129         index = i*2;
130       }
131     }
132     indices[index]++;
133     return minimum;
134   } //End getMin()
135
136   //This is the thread that we will use to sort.
137   static class WorkerThread<T> extends Thread{
138     int start, end;
139     T[] tmp;
140
141     //Constructor that will take an array, and the start/end indices
142     public WorkerThread(T[] arr, int s, int e) {
143       tmp = arr;
144       start = s;
145       end = e;
146     } //End constructor
147
148     public void run() {
149       sort(tmp, start, end);
150     } //End run()
  } //End WorkerThread
}//End ThreadedSorter
```

Report

Expectations

Since the sort is making use of parallel processing, it should be safe to assume that the processing time decreases as the number of threads goes up, providing the array size is large enough to overcome the overhead associated with spawning threads and merging.

Computation Analysis

It seems that the benefit of the threading improves until the leap from 16 to 32 threads, at which it starts to lose it's benefits, returning gradually to it's former one threaded effiency In most cases, the 256 thread sort is worse than it's single threaded sort. One explaination for this is that the overhead may start to outweigh the benefits. Because the array is split into smaller chunks, the merge portion (Shown on line 102), which is the portion that must be done in a single thread, has more work in sorting the array. Thus, the algorithm reduces in speed if too much work is put on the merge section. However, when the array is making use of a lesser thread count, we can see improvements as much as a halved sort time. This is because the array spends more time in the multithreaded sort then it does in the single thread merge.

Conclusion

Multithreading does provide a huge benefit to computing if successfully done. However, attention must be paid to how many threads to use, as the overhead will outweigh the benefits if too many threads are used, as the merging of the threads presents a more and more signifigant overhead as thread count increases.