Multithreaded Merge Sort

Learn how to perform Merge sort using threads.

Merge Sort

Merge sort is a typical text-book example of a recursive algorithm and the poster-child of the divide and conquer strategy. The idea is very simple, we divide the array into two equal parts, sort them recursively and then combine the two sorted arrays. The base case for recursion occurs when the size of the array reaches a single element. An array consisting of a single element is already sorted.

The running time for a recursive solution is expressed as a *recurrence equation*. An equation or inequality that describes a function in terms of its own value on smaller inputs is called a recurrence equation. The running time for a recursive algorithm is the solution to the recurrence equation. The recurrence equation for recursive algorithms usually takes on the following form:

Running Time = Cost to divide into n subproblems + n * Cost to solve each of the n problems + Cost to merge all n problems

In the case of merge sort, we divide the given array into two arrays of equal size, i.e. we divide the original problem into sub-problems to be solved recursively.

Following is the recurrence equation for merge sort.

Running Time = Cost to divide into 2 unsorted arrays + 2 * Cost to sort half the original array + Cost to merge 2 sorted arrays

> $$T(n) = Cost\ to\ divide\ into\ 2\ unsorted\ arrays + 2 * T(\frac{n}{2}) + Cost\ to\ merge\ 2\ sorted\ arrays when n > 1$$

$$T(n) = O(1) \ \ when\, n = 1$$

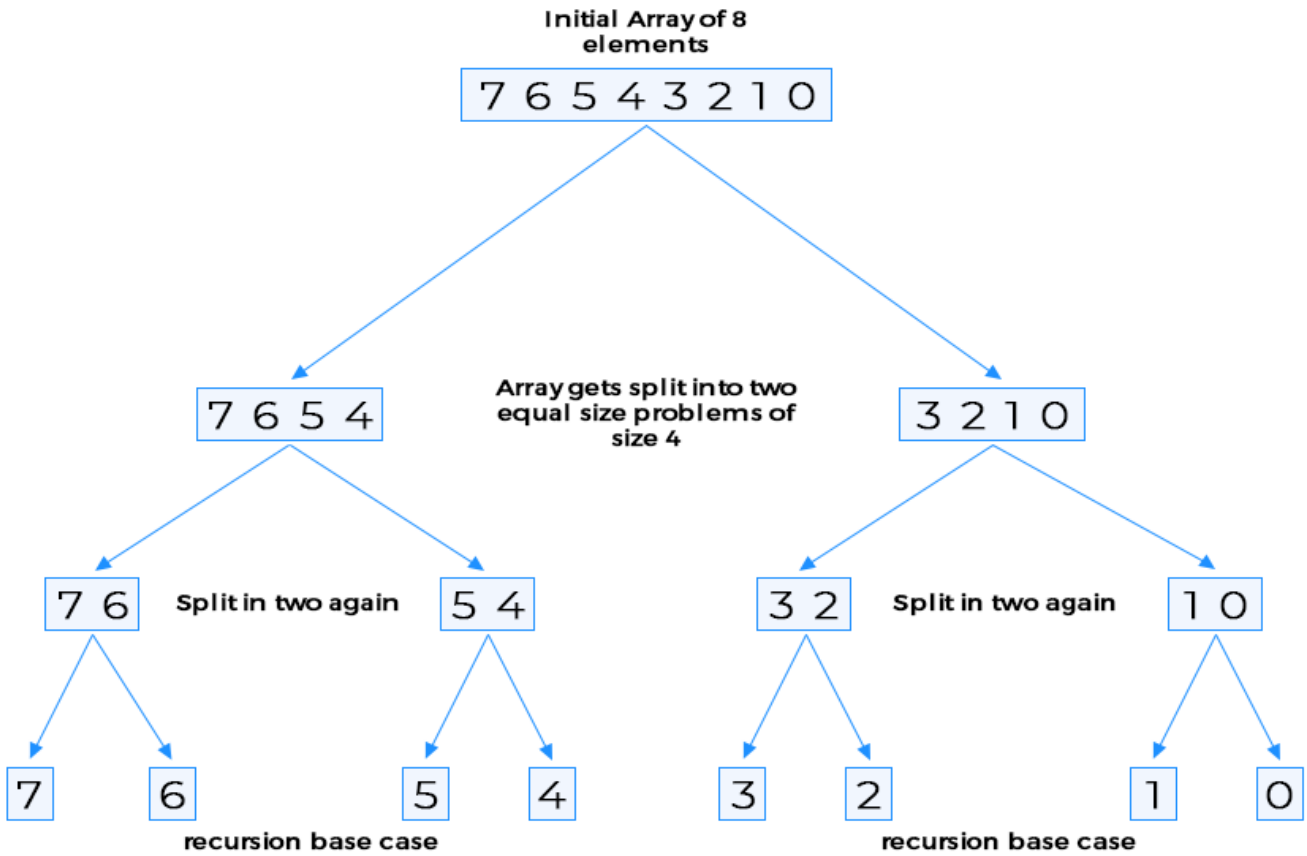
Remember the *solution* to the recurrence equation will be the *running time* of the algorithm on an input of size n. Without getting into the details of how we'll solve the recurrence equation, the running time of merge sort is

$$O(nlgn)$$

where *n* is the size of the input array.

Merge Sort Recursion Tree

Below is a pictorial representation of how the merge sort algorithm works



Merge sort lends itself very nicely for parallelism. Note that the subdivided problems or subarrays don't overlap with each other so each thread can work on its assigned subarray without worrying about synchronization with other threads. There is no data or state being shared between threads. There's only one caveat, we need to make sure that peer threads at each level of recursion finish before we attempt to merge the subproblems.

Let's first implement the single threaded version of Merge Sort and then attempt to make it multithreaded. Note that merge sort can be implemented without using extra space but the implementation becomes complex so we'll allow ourselves the luxury of using extra space and stick to a simple-to-follow implementation.

```
1. class SingleThreadedMergeSort {
2.
3.   private static int[] scratch = new int[10];
4.
5.   public static void main( String args[] ) {
6.     int[] input = new int[]{ 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
7.     printArray(input,"Before: ");
8.     mergeSort(0, input.length-1, input);
9.     printArray(input,"After:  ");
10.
11.   }
12.
13.   private static void mergeSort(int start, int end, int[] input) {
14.
15.     if (start == end) {
16.       return;
17.     }
18.
19.     int mid = start + ((end - start) / 2);
21.
22.     // sort first half
23.     mergeSort(start, mid, input);
24.
25.     // sort second half
26.     mergeSort(mid + 1, end, input);
27.
```

```
28.     // merge the two sorted arrays
29.     int i = start;
30.     int j = mid + 1;
31.     int k;
32.
33.     for (k = start; k <= end; k++) {
34.       scratch[k] = input[k];
35.     }
36.
37.     k = start;
38.     while (k <= end) {
39.
40.       if (i <= mid && j <= end) {
41.         input[k] = Math.min(scratch[i], scratch[j]);
42.
43.         if (input[k] == scratch[i]) {
44.           i++;
45.         } else {
46.           j++;
47.         }
48.       } else if (i <= mid && j > end) {
49.         input[k] = scratch[i];
50.         i++;
51.       } else {
52.         input[k] = scratch[j];
53.         j++;
54.       }
55.       k++;
56.     }
57.   }

  private static void printArray(int[] input, String msg) {
    System.out.println();
    System.out.print(msg + " ");
    for (int i = 0; i < input.length; i++)
      System.out.print(" " + input[i] + " ");
    System.out.println();
  }
}
```

In the above single threaded code, the opportunity to parallelize the processing of each sub-problem exists on **line 23** and **line 26**. We create

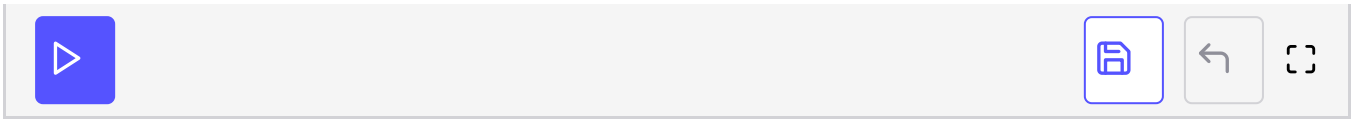
two threads and allow them to carry on processing the two subproblems. When both are done, then we combine the solutions. Note that the threads work on the same array but on completely exclusive portions of it, there's no chance of synchronization issues coming up.

Below is the multithreaded code for Merge sort. Note the code is slightly different than the single threaded version to account for changes required for concurrent code.

```
import java.util.Random;

class Demonstration {

    private static int SIZE = 25;
    private static Random random = new Random(System.currentTimeMillis());
    private static int[] input = new int[SIZE];

    static private void createTestData() {
        for (int i = 0; i < SIZE; i++) {
            input[i] = random.nextInt(10000);
        }
    }

    static private void printArray(int[] input) {
        System.out.println();
        for (int i = 0; i < input.length; i++)
            System.out.print(" " + input[i] + " ");
        System.out.println();
    }

    public static void main( String args[] ) {
        createTestData();

        System.out.println("Unsorted Array");
        printArray(input);
        long start = System.currentTimeMillis();
        (new MultiThreadedMergeSort()).mergeSort(0, input.length - 1, input);
        long end = System.currentTimeMillis();
        System.out.println("\n\nTime taken to sort = " + (end - start) + " milliseconds");
        System.out.println("Sorted Array");
        printArray(input);
    }
}

class MultiThreadedMergeSort {

    private static int SIZE = 25;
    private int[] input = new int[SIZE];
    private int[] scratch = new int[SIZE];

    void mergeSort(final int start, final int end, final int[] input) {

        if (start == end) {
            return;
```

```
        }

        final int mid = start + ((end - start) / 2);


        // sort first half
        Thread worker1 = new Thread(new Runnable() {

            public void run() {
                mergeSort(start, mid, input);
            }
        });

        // sort second half
        Thread worker2 = new Thread(new Runnable() {

            public void run() {
                mergeSort(mid + 1, end, input);
            }
        });

        // start the threads
        worker1.start();
        worker2.start();

        try {

            worker1.join();
            worker2.join();
        } catch (InterruptedException ie) {
            // swallow
        }

        // merge the two sorted arrays
        int i = start;
        int j = mid + 1;
        int k;

        for (k = start; k <= end; k++) {
            scratch[k] = input[k];
        }

        k = start;
        while (k <= end) {

            if (i <= mid && j <= end) {
                input[k] = Math.min(scratch[i], scratch[j]);

                if (input[k] == scratch[i]) {
                    i++;
                } else {
                    j++;
                }
            } else if (i <= mid && j > end) {
                input[k] = scratch[i];
                i++;
            } else {
                input[k] = scratch[j];
                j++;
            }
            k++;
        }
    }
```

```
}
```



Multithreaded Merge Sort

We create two threads on lines **51** and **59** and then wait for them to finish on **lines 67-68**. On smaller datasets the speed-up achieved may not be visible but larger datasets which are processed on multiprocessor machines, the speed-up effect will be much more pronounced.