Multithreaded Merge Sort

In this lesson, we will implement merge sort using multiple 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, 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

$$T(n) = Cost\ to\ divide\ into\ 2\ unsorted\ arrays + 2 * T(\frac{n}{2}) + C$$

$ost\ to\ merge\ 2\ sorted\ arrays\ when\ n > 1$

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

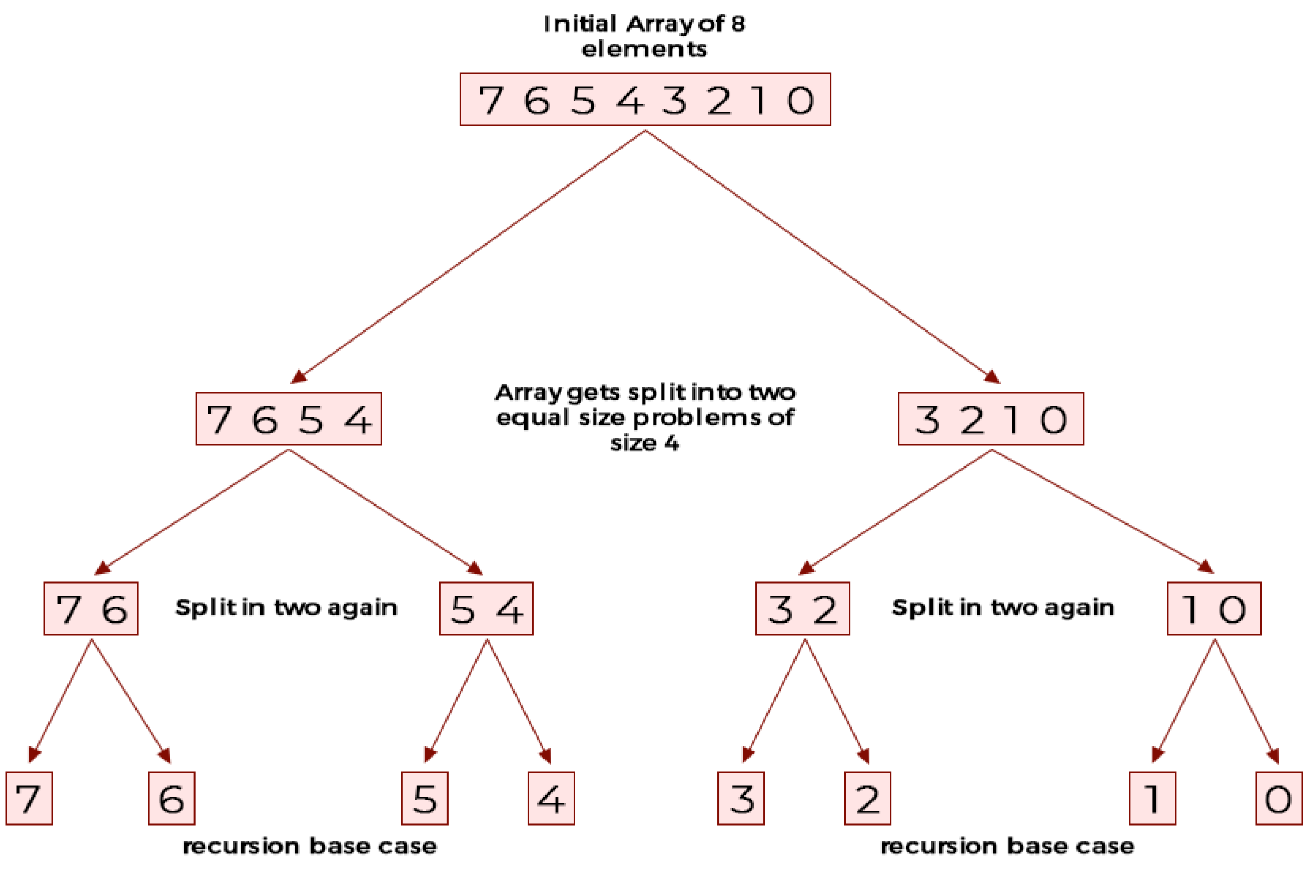
Remember, the *solution* to the recurrence equation will be the *running time* of the algorithm on an input of size n. Without getting into the details of how we'll solve the recurrence equation, the running time of merge sort is

$$O(n\ lg(n))$$

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 to 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.

```
def mergeSort(startIndex, endIndex, input)

   if startIndex == endIndex
     return
   end

   mid = startIndex + ((endIndex - startIndex) / 2)

   # sort the first half
   mergeSort(startIndex, mid, input)

   # sort the second half
   mergeSort(mid + 1, endIndex, input)

   # merge the two sorted arrays
   i = startIndex
   j = mid + 1

   for k in (0..input.length) do
     $scratch[k] = input[k]
   end

   k = startIndex

   while k <= endIndex do

    if i <= mid and j <= endIndex
      input[k] = [$scratch[i], $scratch[j]].min

      if input[k] == $scratch[i]
        i += 1
      else
      j += 1
      end

    elsif i <= mid and j > endIndex
      input[k] = $scratch[i]
      i += 1
    else
      input[k] = $scratch[j]
      j += 1
    end

    k += 1
  end
end
```

In the above single-threaded code, the opportunity to parallelize the processing of each sub-problem exists on **line 10** and **line 13**. We create two threads and allow them to carry on processing the two subproblems. When both are done 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.

```
def mergeSort(startIndex, endIndex, input)

  if startIndex == endIndex
    return
  end

  mid = startIndex + ((endIndex - startIndex) / 2)

  # sort the first half
  t1 = Thread.new do
    mergeSort(startIndex, mid, input)
  end

  # sort the second half
  t2 = Thread.new do
    mergeSort(mid + 1, endIndex, input)
  end

  t1.join()
  t2.join()

  # merge the two sorted arrays
  i = startIndex
  j = mid + 1

  for k in (0..input.length) do
    $scratch[k] = input[k]
  end

  k = startIndex


  while k <= endIndex do

    if i <= mid and j <= endIndex
      input[k] = [$scratch[i], $scratch[j]].min

      if input[k] == $scratch[i]
        i += 1
      else
        j += 1
      end

    elsif i <= mid and j > endIndex
      input[k] = $scratch[i]
      i += 1
    else
      input[k] = $scratch[j]
      j += 1
    end

    k += 1
  end
end

MAX_SIZE = 10

$scratch = Array.new(MAX_SIZE)
input = Array.new

MAX_SIZE.times do |m|
  input << (MAX_SIZE - m)
end

starting = Process.clock_gettime(Process::CLOCK_MONOTONIC)
mergeSort(0, input.length - 1, input)
ending = Process.clock_gettime(Process::CLOCK_MONOTONIC)

puts input
puts "Multiple threads took : #{ending - starting}"
```

As a crude experiment if we run the multithreaded vs single-threaded merge sort program on a MacBook, we observe the following runtimes for arrays of different sizes. It is apparent that the multithreaded code is significantly slower than the single-threaded code on account of GIL. A compute-intensive task such as in-memory sorting or matrix multiplication doesn't benefit from a multithreaded design because at any time only a single thread can execute even on a multicore machine. In

fact, multiple threads create management and communication overhead which slows down the program execution.

Input Size	Single Thread (secs)	Multiple Threads (secs)
1000	0.05	0.18
10,000	4.95	12.07
13,000	8.28	17.88