**Jyoti Vaswani (NUID -001005711)**

**INFO 6205**

**Program Structures and Algorithms**

**Fall 2020**

**Assignment No -5**

**Task :** Parallel merge sort analysis with experiments to find suitable cut-off value.

1. **I have added code in ParSort.java for :**

* **sort () –** I have added the code for merging the partitions post they are been sorted.
* **parsort ()-** Added the code for utilizing the threads from the custom thread pool and joining the result back.

1. **I have modified the loops for providing different input parameters in Main.java**

* Changed the input array size as per the experiment and the looping to find the suitable cut off value.

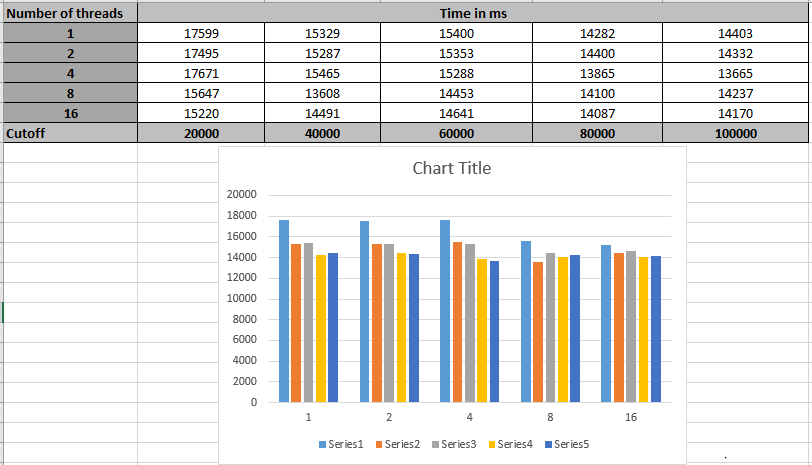
1. **Output:**

Below are the screenshots of output generated when 50 Million int where taken in an array over which parallel merge sort was performed.

Also, the screenshots of the exhaustive cut-off experiment over different input values are attached.

Note: Kindly check the folder (Output\_Screenshots) as I am not able to insert them as Objects here.

1. **Observations and Analysis:**



* **Execution time against number of threads:** For the experiment, we have kept the cutoff (from 20000 to 100000) and input size of random array fixed (10 million primitive int), the execution time is recorded against the number of threads provided in the pool. The same is plotted using a bar graph (as shown the above screenshot).

It is observed that the execution time to sort 10 million primitive integers over threads 1,2,4 give almost the same performance whereas it falls when the thread pool is increased to 8. However, it shows no significant gain when the thread pool is further increased to hold 16 threads.

To reason the above observation, there are few factors that should be taken into account:

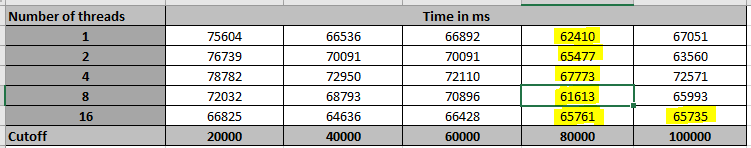
1. **Merge operation dependency**: Even though we have parallelised the divide strategy of merge sort, the conquer part depends completely on the sorted sub-arrays being provided as input. Due to this the independent thread allocated for merger has to wait/stays blocked until it gets the result (sorted sub-array) from other threads.
2. **Time slicing and switching between threads**: When the number of threads exceeds the CPU cores, the execution of tasks switches frequently between threads adding to the communication cost by saving over sequential computational costs. Also, not all the extra threads get the dedicated CPU for execution rather they share with other threads using various time slicing strategies.

If we ignore the communication cost then the actual computation happens only while merging the sub-arrays. In worst case, it would require 2s - 1 compares and may be swaps (considering s as the size of sub-array). Since the recursion depth is atmost lg N (as we switch to system.sort when we hit cut-off, the recursion depth will hardly reach its max value of lg N), there would be atmost lg N merging steps:

Merging corresponds to O(N) time complexity.

* **Execution time against cut off values :**

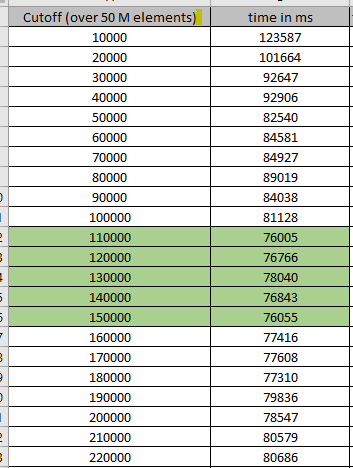
Over various input values (10 M to 50 M) different cut-off values were set and execution time against those were recorded (data attached in analysis.xlsx). It was observed that the minimum execution time was recorded when the cut-off was in the **range of 80000 to 10000** for input array sizes like 10, 20, 30, 40 M (as highlighted in screenshot below – 40 M int values taken in input).



However, the same range did not give the similar performance for larger input arrays like 50 M. For such values (50 M and above), the cut-off value was found most suitable when it was over 100000 but yet below 150000.

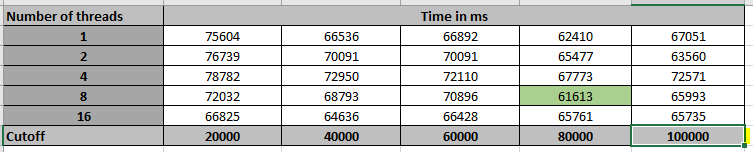
Below is the screenshot for the execution time captured for 50 M elements

Note: The exhaustive detail of execution time is attached in .xlsx



* **Combination of suitable cut off value and number of threads :**

The strategy used in this parallel merge sort is that it sorts divisions in parallel recording lesser execution time when the threads are equal to the CPU core of the machine and the cut-off value is in the range of 80000 to 100000. As shown in the below screenshot, the best execution time value is achieved for 40 M elements with the cut-off of 80000 and 8 threads.



1. **Evidence to support relationship observations and conclusions:**

To support the observations and analysis described above, I took different values of elements from 10 million to 50 million.

Also, an experiment was done to find the suitable value of cut off for 50 million int values in random array.

Data thus collected was plotted and graph analysis was done to make the above conclusions.

All the data values are added in the below attached spreadsheet.



1. **Screenshot of Unit test passing:**

No unit test cases were available/asked for this assignment.