Concurrent Implementation of Divide-and-Conquer Sorting Algorithms in Java

University of Central Florida

Johnny Ngo, Anthony Galbo, Kevin Abreu, Jessica Hawkins, Mykola Shpota

Abstract—Sorting has been predominantly one of the most important concepts computer scientists consider today. Some of the best sorting algorithms are quick sort and merge sort. Both of the algorithms follow the concept of "divide and conquer" where it breaks down a list into sub lists until it gets down to 1 element and then recombines back in an orderly sorted matter. In that behavior, the design runs log linear complexity or O(nlogn). Up until now we are aware that both of the implementations run on a single and sequential process. However, as technology advances we are starting to look in different ways to make sorting algorithms run much more efficiently than our prior concept of sequential sorting.The idea of implementing parallel multi-threaded processes on quick sort and merge sort can perhaps, quickly improve its run-time. If the parallel implementation does gain some drastic results, the scene revolving sorting can improve the current programs that utilize it. In this report we will implement and test the original sequential algorithm of merge and quick sort with our multi-threaded methods to find whether multi-threaded has better efficiency in run-time.

1 Introduction

Our research report focuses mainly on the merge sort and quick sort algorithms. Two of our group members are tasked to create and document the source code for testing. They will implement two different parallel approaches to the sequential method and compare both of the results. The source codes will be written in Java where we will utilize one of the concurrent frameworks Java possesses, Fork Join framework and its subclass Recursive Action. Testing component is controlled for both implementation where we are working with array size of

$$\sum_{n=1}^{8} 10^n$$

Research will be carefully documented with additional charts to substantiate our analysis between the run-time execution of sequential and parallel.

2 Background and Related Work

In this section, we will cover the general overview and preliminaries of the sorting algorithm merge sort and quick sort.

2.1 General History

Quick-sort has been around since the mid 1900's when a British computer scientist, Tony Hoare developed it [1]. In 1959, Tony Hoare was working on a translation project at the time, as a student of Moscow State University. He initially used insertion sort to support the restructuring of Russian words but later realized it was too slow. After several attempts, he came up with an idea of utilizing partitioning of the data structure and the idea of quick sort was brought to life. [2] Tony Hoare first programmed quick sort in a language called Mercury Autocode, however it was a failed attempt due to lack of recursion and partitioning. then studied Algol 60 where the language includes the feature of allowing the functions to call itself which led to successful publication of quick sort. Merge sort on the other hand, was much older than quick sort by 15 years. It was first invented by a mathematician John Von Neumann in 1945 [3].

2.2 Overview of Quicksort

Quick sort is classified as a divide-and-conquer algorithm. It takes an array and recursively breaks it down into sub-array by the partitioned element. Depending on programmer preference, the partition point can be placed at random, low, max, or mid of the array. However the choice of the pivot point can immensely affect the run-time, having a low pivot point can prolong the sorting since the left side would be sorted already in the middle of the progress. It is preferable to proceed with a random pivot point: (pivot = new Random().nextInt(end - start) + start) or the middle point: (pivot = start + (end - start) / 2) to fully optimize the sorting. Compared to merge sort, quick sort is an in-place algorithm where it does not need additional memory for storing the elements since it overwrites the original array with the output.

2.3 Pseudocode for Quicksort Sequential

```
quickSort(A,L,H)
1. If L is less than H
Get the partitioning index, P
quickSort(A,L,P - 1)
quickSort(A,P + 1, H)
```

partition(A,L,H)
```
1. Get pivot by A[H]
2. Get index of smaller element (low - 1), i
3. For loop from low to high
If current element is smaller than pivot then increment i and swap A[i] and A[j] 4. Afterwards swap A[i + 1] and A[H] 5. return i + 1
```

2.4 Overview of Merge Sort

Merge sort go hand to hand with quick sort as divide-and-conquer by taking an array and recursively breaking it down into smaller sub-array until we get to one to two elements sub array and then return the valid order of the two or one element from the recursive stack. It does not use partition points like quick sort and divides the array by using the midpoint algorithm.

2.5 Pseudocode for Merge Sort Sequential

mergeSort(A)
```
1. Let n be the length of A
2. if n ¡= 1 return
3. $n_1$ = $n/2$
4. $n_2$ = n - $n_1$
5. Let L[1...$n_1$ + 1] and R[1...$n_2$ + 1] be new arrays
6. for i to $n_1$ L[i] = A[$n_1$]
7. for j to $n_2$ R[j] = A[$n_2$]
8. mergeSort(L)
9. mergeSort(R)
10.merge(A,L,R)
```

merge(A,L,R)
```
1. Let $n_1$ be the length of L
2. Let $n_2$ be the length of R
3. i = 0
4. j = 0
5. k = 0
6. while i ¡ $n_1$ and j ¡ $n_2$
if L[i] ¡ R[j]
A[k] = L[i]
i = i + 1
else L[i] ¿ R[j]
A[k] = R[j]
k = k + 1
j = j + 1
7. while i ¡ $n_1$
A[k] = L[i]
k = k + 1
i = i + 1
8. while j ¡ $n_2$
A[k] = R[j]
k = k + 1
j = j + 1
```

3 Methodology Overview

Our research was conducted to understand sorting algorithms and how a parallel implementation would impact them. Converting algorithms standardized by optimizing run-time and efficiency on a single thread to a multi-threaded implementation can have a wide range of impacts on the algorithm's performance. To better compare and observe changes of multi-threaded sorting, two sorting algorithms of a similar style were chosen. Merge sort and Quick sort, two "divide and conquer" style sorting algorithms with similar average run-times [5] were chosen to parallelize and analyze the change in algorithms run-time, and structure. The best place to utilize multiple threads in the case of both Quick Sort and Merge Sort is in the arrangement of array values. In this phase of divide and conquer sort methods, a (traditionally) single thread must recursively run through the array by splitting segments. With a single thread this leaves Merge Sort with a run-time best, average, and worst case of O(nlogn) [4], and Quick Sort with an O(nlogn) average case and a worst-case scenario for pre-sorted arrays of O(n2) [3]. The best place for improvement of run-time is through the dividing portion, rather than sending a thread through every value, the tasks can be split amongst other threads to not only divide and conquer the array but to divide the workload concurrently and shorten the runtime.

3.1 Using Java

Our group has decided to use Java instead of Python because it provides the available tools such as libraries and documents to assist in our implementation of concurrent sorting. Python on the other hand has concurrency but it is much slower than Java. The concurrent features such as the **java.util.concurrent** library will be our main usage for this report. Although concurrent syntax for Java can be difficult, the **java.util.concurrent** makes writing concurrent programs much easier because of pre-existing functions (in which we will explain more in the next section). Overall the group was competent in writing with Java.

3.2 ForkJoinTask

Both the merge Sort and quick sort utilize the abstract Java ForkJoinTask class to implement thread-like behavior in the algorithm. The ForkJoinTask class was chosen over traditional threads because it is lighter weight and can complete the pre-designated work as requested without leaving any sitting threads making its use more efficient than traditional threads or thread pools [5]. The ForkJoinTask class also provides a standard implementation that returns all results together [5], which works easily with recursion, a large part of both sorting algorithms.

4 Quicksort Parallel Implementation

Quick Sort, like Merge Sort, is a divide and conquer algorithm. Instead of recursively dividing the array in half into smaller segments, Quick sort "randomly" picks a value and uses the chosen value as a pivot to order and partition the other elements around. The partitions are not necessarily in half or in any predesignated fraction. Quick Sort should not require any additional storage to arrange the values like Merge Sort. The multi-threaded optimization of Quick Sort should aim to reduce the portion of the algorithm with the longest runtime, the arrangement of values in order around the pivot (the handling of the partitions). Implementing multi-threading in this area draws Quick Sort and Merge Sort closer in implementation as the arrangement around the pivot breaks the array into smaller portions to order the values. In this case, we chose to use the ForkJoinPool and its task scheduling to handle the branching from splitting the array of values. In implementation, the threaded approach used the ForkJoinPool's executor to arrange and execute the quick sort actions and obtain the Future, which contained each ForkJoinTask's result/return and helps to split the work.Initially the methods were confused and had the program running sequentially but on a single thread. Once the initial issue was addressed, the test cases were run through the algorithm. In these results, for every individual testing the algorithm, any test array containing over 100000 elements would trigger a StackOverflowError. This error class is commonly thrown when recursion is too deep or is infinite, overflowing the stack frames/JVM Memory (Java Virtual Machine) [7]. We believe the issue may be due to the arranging of FJT (ForkJoinTasks) then calling in the implementation after which behaved recursively could be the cause of the "overflow". To address the stack overflow error, the group began working on debugging the algorithm only to realize that the Quick Sort was still behaving sequentially however, it was not running in parallel with multiple threads as it was originally thought. It was observed that the algorithm was still running using only a single thread and there was a misunderstanding with the methodology the multi-threaded Quick sort was attempting to use. While changes are being made, the test cases were run through a standard quick sort to observe the run-times and to allow us to re-evaluate the methodology we will choose going forward with the multi-threaded implementation of Quick Sort.

4.1 Pseudocode for Quicksort Parallel

```
class ParallelQuicksort
compute()
1. if start is greater than end, return
2. get partition index by partition function
3. call sequential quicksort on left and right partition
4. create task for left fork()
5. sort current partition on right compute()

partition(Arr,L,H)
1. select pivot index using H
2. get small index, i
for loop using j from L to H
increment i and swap i and j in A
3. swap i + i with high in A
4. return i + 1
```

5 Merge Sort Parallel Implementation

Merge sort's divide and conquer strategy utilizes a method of dividing and subdividing the contents in an array in half, recursively (a function implements steps and calls itself from those steps), to separate and sort values. A merge function is used to reassemble the values in the sorted order back into a single array. The multi-threaded optimization of Merge Sort would be best utilized when dividing and sorting the array because the array arrives whole and typically unsorted. The divide and conquer quality of Merge Sort means dividing the array into the individual indexes by splitting the array in half, then the halves in half and so on. A single thread traverses the algorithm working up and down instances of the split array. Supplying a method of splitting tasks so that they can work individually instead of one thread traversing the tasks could speed up the sorting portion of the Merge Sort. To parallelize the merge sort, a thread pool would be best utilized to split workload amongst "threads" and avoid over-spawning threads. ForkJoinPool was chosen to be used for Merge Sort's parallelization. The ForkJoinPool allows for multi-threaded tasking in cases like Merge Sort where the algorithm uses recursion to divide the array. Each task completes a split of the array given to it and passes the split values to two new forked task threads, one taking the left set and the other taking the right set. This continues until a task no longer has more than one value and thus cannot split the array any further and then the tasks begin to return out of the recursion. This allows for as many threads or "tasks" as needed to complete the traversal and handle the divisions.

5.1 Pseudocode for Merge Sort Parallel

```
class ParallelMergeSort
compute()
1. if length of array is less than or equal to 1, return
2. get size of elements left and right of the array
3. create temp arrays for left and right
4. copy data from original array to the temp arrays
5. recursively sort into two halves
6. invoke the declared left and right sorting tasks
7. merge left and right of the array

merge(Arr,LA,RA)
1.create first, second index, and i. initialize all value to 0
2.While loop with condition of first index is less than length of LA and second index less than length of LA
In the while loop: If left part of our arr at first index is smaller than right part, put left part into main array and increment both indexes. Otherwise we put the right part into main array and increment both indexes
3. After we are done looping through, we will fill the remainder of elements for our left part and right part of our array.
```

6 Testing and Results

To analyze the results of the project code, our team used a set of universal test cases. Each of the test cases generate an array filled with random values via Java's random class. The random values are used to populate the multiple arrays generated by the test cases. To provide an adequate range of performance array lengths of 10, 100, 1000, 10000, 100000, 1000000, 10000000, and 100000000 are used for both algorithms. Testing with this range and random variables allows for the group to get the best view of performance without bias in ordering or size by individual testing. The time is recorded just before each test case array is passed through to the algorithm and then the final time is taken once the algorithm completes to get the algorithm's most accurate run-time. Each run-time is printed out to the command line at the end to be viewed by the user/coder running the code. In Merge Sort the parallelized version is slower for six of the eight test cases consistently. The reasoning for this is that the Merge Sort algorithm is optimized for a single thread, which while it does get gradually slower as the array size increases it can still handle a fairly large number of values while the multi-threaded version takes additional time to spin up the threads and handle the tasks appropriately. However, at 10000000 (10 million) value arrays and larger, the parallelized version begins to outperform the single threaded version. At this point it is likely that the time it takes to produce the threads to handle their respective tasks is made up by the threads having a capability to handle more tasks in a concurrent fashion.

7 Conclusion

In our work for this project, we researched the topic of sorting. Particularly on how sorting could be improved with parallel computation. We looked at two popular sorting algorithms: merge sort and quick sort. It is a known fact that it is impossible to achieve worst-case run-time better than O(nlog n) when performing comparison-based sorting. Thus, applying parallel processing to merge and quick sorts won't change the time complexity overall since the same amount of work will need to be performed as in traditional sorting. However, it doesn't mean that it would not be of any benefit at all. As it can be seen through our research conducted above that applying multi-threading techniques reduces the overall time needed to perform the sorting of data. Merge and quick sorts belong to divide and conquer family. In their implementations, the main idea was to take an array of data and recursively break it down into smaller sub-arrays. Merge sort is a clear example here, where during each division step an array is split exactly in half. With quick sort, things are a bit more complex but the general principle still stays the same. To adapt the merge sort algorithm to the multi-threaded environment, in our project we used Java's Fork/Join Framework. It provides the necessary API with a high level of abstraction to make this relatively easy. With its help, when breaking down an array into sub-arrays, the new tasks are created as instances of RecursiveAction class. Creating smaller tasks and the ability to manage and evenly distribute these tasks between the threads is the key to the proper functioning parallel sorting algorithm. Task management is done with the use of the ForkJoinPool class that creates a default pool. It supports a level of parallelism equal to the number of processors available in the system. However, there is no limit on how many tasks can be managed by the pool. Thus, the level of parallelism is limited by the number of processors/cores in the system when using the Fork/Join framework. Workload distribution between the threads is achieved by the ability of idle threads to execute tasks from the global queue or steal tasks submitted by other active tasks (term called work-stealing).

7.1 Observation of Merge Sort

n	10	10^2	10^3	10^4	10^5	10^6	10^7	10^8
P, (ms)	3	5	1	14	23	39	365	3947
S, (ms)	0	0	1	1	12	126	1192	13226

TABLE 1
Comparison between parallel(P) and sequential(S) for Merge Sort
CPU: 6 cores, 12 threads

Through a practical approach, we confirmed that the divide and conquer strategy that underlines merge sort and quick sort algorithms translates very well into a parallel processing world. That's because dividing an array into pieces and creating sub-tasks creates a perfect environment where pieces of data could be simultaneously processed. That is also true according to Amdahl's law. In our merge sort experiments, we saw the speed up in sorting however it wasn't instantaneous. What we observed is that the execution time of the parallel algorithm was only faster when array sizes were close to 10,000,000 and above. Below that value, the single-threaded sorting algorithm outperformed its parallel counterpart. Such results are due to overhead associated with creating new threads and managing subtasks during parallel processing. For smaller grouping of data, this overhead slows down the performance of parallel execution enough for it to fall behind. For array sizes above 10 million, parallel sorting showed significant improvement in times. Thus, it is very beneficial to use parallel merge sort when dealing with particularly large arrays of data. To further improve the performance of the considered merge sort algorithm, is to find, through practical experimentation, a certain threshold. When this threshold is reached, a sequential execution is used without further dividing the array and creating new sub-tasks. Perhaps to use a slower sorting algorithm like Insertion sort or similar. In this case, it will be faster than continuing with the merge sort. Such a solution comes from a single-threaded experience where a large number of recursive calls make merge sort not beneficial or even slower for sorting small arrays compared to Insertion sort. This kind of approach would eliminate a vast amount of sub-tasks that would consist of sorting only a couple of elements. It would further decrease the amount of work-stealing between threads and lead to an overall increase in performance.

7.2 Observation of Quicksort

n	10	10^2	10^3	10^4	10^5	10^6	10^7	10^8
P, (ms)	2	1	3	11	21	32	192	1863
S, (ms)	0	0	0	1	6	75	862	9581

TABLE 2
Comparison between parallel(P) and sequential(S) for Quicksort
CPU: 6 cores, 12 threads

As with merge sort, our quick sort algorithm was implemented using Java's Fork/Join Framework. Just as with single-threaded quick sort, our parallel implementation divides the main array to be sorted into sub-arrays. However, instead of recursively proceeding further, new sub-arrays are submitted into the common pool of tasks to be worked on by the threads in a parallel fashion. Since all the threads in this algorithm are meant to be working on the same array without using additional space to copy sub-arrays into, no merging is needed at the end as all the sorting happens in place. Although this creates a concern of data being corrupted or lost by overwrites, this problem is solved by the fact that each thread works on a separate part of the array. It happens because of the quick sort's pivot point. After the current pass of the array is complete, the pivot is brought into a proper place of the array where it is now in a sorted position, and then it serves as a mark where to split the array for further sorting. Thus, new tasks that are submitted into the pool never happen to have overlapping intervals that need to be sorted. Our experimental results showed that the parallel quick sort algorithm is much more beneficial for sorting very large arrays. We observed that the faster computational times started to become noticeable with arrays of 1 million elements. After that, with an increase of elements, the difference with single-threaded algorithms would continue to grow. Particularly, for the 10 million elements array, the speed up was 1.97 times and for the size of 100 million elements, it performed 2.58 times faster. The slow down of parallel quick sort when processing arrays of sizes less than 1 million could also be due to the longer time that it takes to create the new threads. During our research of quick sort, we faced an issue with the algorithm. It was that it would consistently give stack overflow errors for array sizes of 100k elements and above. That was an odd fact because the algorithm worked for arrays of smaller sizes, so the sorting part was performing correctly but somehow too many sub-tasks were being created. Such a thing was not observed with the merge sort even though it used a similar strategy. After some further research, we suspected that the root of the problem was in the quick sort algorithm itself and specifically in its performance with the array that is already sorted. In this case, single-threaded quick sort run-time could become quadratic. But arrays that we were using were not sorted and nevertheless we would have a stack overflow problem. This fact led us to find the fault in the way arrays were randomly populated. There was a limit set on the range of random values to be produced and in the very large arrays that created too many duplicate values. Further, too many duplicates would lead closer to the worst-case run time and a very large number of tasks to be submitted in the pool, and eventually cause the stack overflow problem. Besides increasing the range of random numbers in the array, another solution to this problem could be to implement a certain threshold beyond which a different sorting algorithm would be utilized; also use a different partitioning method that would pick a pivot as a median of randomly selected elements from the part of the array to be sorted. To conclude our results, even with straightforward implementations through using Fork/Join Framework, merge sort and quick sort algorithms showed a significant increase in computational speed and demonstrated the usefulness of parallel processing techniques applied to sorting tasks, especially on vast collections of data.

References

[1] "Introduction," Tony Hoare Contributions CSP. [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2008-09/tony-hoare/csp.html.

[2] Marcelo M De Barros (2015). "Quick sort Interview with Sir Tony Hoare, the Inventory of Quick sort

[3] Knuth, D (1998), "Section 5.2.5: Sorting by Merging". Sorting and Searching, The Art of Computer Programming, Vol.3 (2nd ed.) Addison-Wesley. pp. 158-168.

[4] "Quick sort vs merge sort," GeeksforGeeks, 29-Apr-2021. [Online]. Available: https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/.

[5] "Forkjoinpool class in java with examples," GeeksforGeeks, 07-Jun-2021. [Online]. Available: https://www.geeksforgeeks.org/forkjoinpool-class-in-java-with-examples/.

[6] Powers, David M. W. (1991). Parallelized Quicksort and Radixsort with Optimal Speedup. Proc. Int'l Conf. on Parallel Computing Technologies

[7] StackOverflowError (java platform SE 7), 24-Jun-2020. [Online] https://docs.oracle.com/javase/7/docs/api/java/lang/StackOverflowError.htm