Multi-threaded Approach to Java's Collections.sort()

Nilan Larios
Undergrad Student
Dept. Computer Science
University of Central Florida
Orlando, FL, USA
nilanlarios@knights.ucf.edu

David Jones
Undergrad Student
Dept. Computer Science
University of Central Florida
Orlando, FL, USA
jones.david@knights.ucf.edu

Aaron Schwenke
Undergrad Student
Dept. Computer Science
University of Central Florida
Orlando, FL, USA
AaronSchwenke@knights.ucf.edu

Emilio Morales
Undergrad Student
Dept. Computer Science
University of Central Florida
Orlando, FL, USA
emiliomorales0228@knights.ucf.edu

Tien Pham
Undergrad Student
Dept. Computer Science
University of Central Florida
Orlando, FL, USA
tientpham@Knights.ucf.edu

***Abstract*—Java is a well established programming language released back in 1996 that is used to manage many of the API's, network systems, and software infrastructure that run our world today [1]. It was easy-to-use and well documented, but generally ran worse than the alternative C++. In its use with back-end systems, it is paramount that as much data processing performance should be extracted from the technology as possible. Especially when it comes to its utilization of modern multi-processor architectures. In this project, we will be focusing on attempting to bring, and research, the performance capability of multiprocessor hardware to Java's sorting function Collections.sort(). We will determine its current structure, research multi-threaded alternatives, and then implement and test a concurrent Collections.sort(), known as CustomCollections.sort(). We were successful in both parallelizing and improving Collections.sort() for large lists when using multiple threads with the use of a multi-threaded merge sort [2], however we are still in the process of parallelizing insertion sort to use it in lieu of merge sort when sorting smaller lists of N $\leq$ 10000.**

***Index Terms*—multi-threaded, parallel, concurrent, merge sort**

I. Introduction

Modern software applications are not only quite large, but they also deal with quite a substantial amount of data. As software deals with so much information, one of the core functionalities required in the technologies used to build them is data management, with the most common and expected form of data management being sorting. These sorting algorithms have generally been optimized as much as they can be, however their speeds can potentially be improved by the use of modern multi processor execution. In this project, we will be focusing on sorting, and it can be improved with concurrency.

Java is one of the most used programming languages in the world, being used by an estimated 33.27% of all developers on Stack Overflow [3]. It has been used with a multitude of applications, primarily in the back-end, being at the forefront of large data allocation, processing, and management. However despite this predominance in back-end technologies, Java's proprietary sorting operation, Collections.sort(), does not operate with multiple threads [4]. This is an issue, as it does not utilize the high-thread architecture that modern computers and software are oriented towards. Especially for enterprise-level systems that can have 8+ cores with 16+ threads.

A. Problem Statement

We are seeking to achieve the main goal of our project: To research and implement a parallelized/multi-threaded approach to Java's Collections.sort() algorithm. This is because Java's Collections library within the java.util package contains only a handful of classes that are thread-safe, notably only Collections.Vector and Collections.Hashtable [4]. Collections.sort() primarily utilizes an implementation of merge sort, along with the use of insertion and quick sorts for differently sized lists [5]. For our implementation we looked at the multithreaded implementation of merge sort described and shown in GeeksforGeeks [2]. Once we have a comparable implementation, we will be focusing on testing its sorting execution time based on different numbers of threads, list sizes, and data types.

B. Goals

Our initial goals for the milestone were to have a rough draft implementation of our CustomCollections.sort() algorithm, that would improve performance for large lists when using multiple threads compared to the singly threaded Collections.sort(). We also planned to allow the algorithm to take a list of any type (which implements Comparable) and sort it. Our longer term goals for the end of the semester will be to implement a combined sorting algorithm, with a multi-threaded insertion sort and possibly quick sort to handle smaller sized lists.

II. Related Works

As you know, the default Collections.sort() algorithm in java is a single-thread implementation of the Merge Sort. Taking into consideration that the Collections.sort() is largely not thread safe, we took inspiration from an open source multi-threaded implementation of Merge Sort by S. Rana uploaded to Geeks-for-Geeks [2]. However, this alone was not enough. Although this implementation correctly applies multi-threading to Merge Sorts, it is limited to only being able to work with integer arrays. There are two main issues we wanted to circumvent with this: being limited to a single data type (integers) and the fact that we want to improve Collections.sort(), which does not apply to the array data structure as they have their own library consisting of Arrays.sort().

Knowing this, we studied an article by N.H Minh, "Understanding Java Collections and Thread Safety" [4] to grasp a better understanding on the limitations of collections and threads, and analyzed a multi-threaded implementation by CarrsCode [7] to convert our implementation into one that handles generics, as well as a data structure that belongs to collections, rather than an array.

III. Methodology

Our goal is to implement a parallel merge sort using Java because merge sort already runs quickly (nlog(n) time) especially for larger datasets, merge sort also uses a divide-and-conquer approach that we believe correlates with parallelization. We use a SortThread object that extends the Thread class to act as our threading. Furthermore we write a sort() method to act as our parallel merge sort algorithm. We hope to show a performance improvement upon a single threaded merge sort implementation, and upon the default Collections.sort() method. Optimally, our implementation will lack the use of locks or parallel queues, as our strategy is to simply split the original data equally between the threads, and then perform a regular merge sort on each thread before merging them together at the end. We debated the use of a thread queue and other more advanced solutions, but determined that simplicity would be easier and likely perform better.

A. Challenges and Redesigns

There were a number of overhauls and small design tweaks that we ran into while writing the code. The first issue we ran into was that our sort would only work with the integer data type, and although it would still be comparable to other sorting algorithms performance wise, we wanted to be able to test any form of data. Our solution to this issue was to use generic types in all the methods, as long as the type implements comparable for the sort. The second notable issue we ran into was being outperformed by the default Collections.sort() when using sorting small quantities, our original solution is to only split the work between threads when it makes sense to use more than one thread. This solution was relatively poor and only managed to optimize the sort for a small number of cases.

a) Rough draft only: We plan to add functionality to the sort that will use a non-merge sort, potentially insertion, when the amount of entries is below a certain threshold

IV. Testing

A. Evaluation

In order to test the custom sort we've created, we have created a testing harness with multiple functions. It creates arrays of multiple different lengths and of varying data types. The data types currently being tested are Characters, Strings, Shorts, Integers, Longs, Floats, and Doubles. The array lengths being tested are powers of 10 in the interval [10, 100000000].

For each array created, we test our custom sort with a differing number of active threads from [1, 8] and then we run java's collection.sort on the same array. The execution times for these sorting algorithms is stored and shows how our sorting algorithm performs compared to Java's collection sort with different parameters.

B. Challenges

Because we are testing the sorting algorithms on large array lengths, testing can take a long time to complete. Because testing is done on our personal devices and we had to use our devices for other reasons during testing, there is a chance that the productivity of the threads used in sorting may not be as high as they should be, and so our data may provide some inaccuracies.

C. Results

Depending on the parameters given to the sorting algorithms, our custom sort could perform better or worse than Java's Collection.sort. When the thread count is 1, the performance of our sort will always be worse as the sorting algorithm is no longer utilizing the benefits of the thread safe sort. Improvements can then be seen as the thread count increases, but the benefit from increasing the thread count diminishes quickly. Also, on small array lengths, our sort performs significantly worse. This could be explained by Java utilizing different sorting algorithms for arrays of different sizes whereas our sorting algorithm currently only uses merge sort [6].

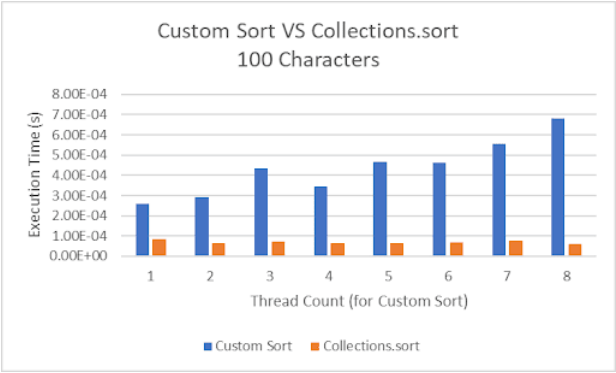

Fig. 1. Comparison of our concurrent sort versus Java's Collections.sort on 100 Characters.

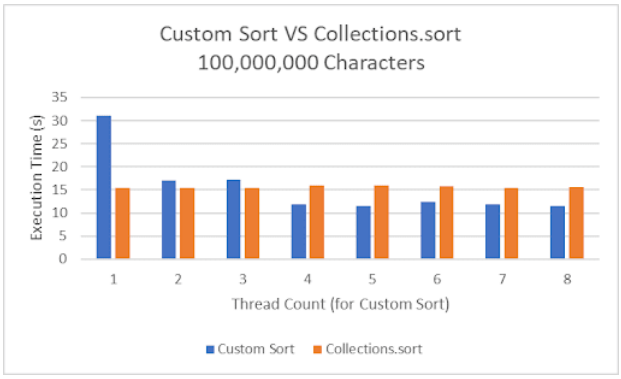

Fig. 2. Comparison of our concurrent sort versus Java's Collections.sort on 100,000,000 Characters.

Characters provided the best sorting performance with an increase in performance from our algorithm when using 4 or more threads for 100,000,000 Characters.

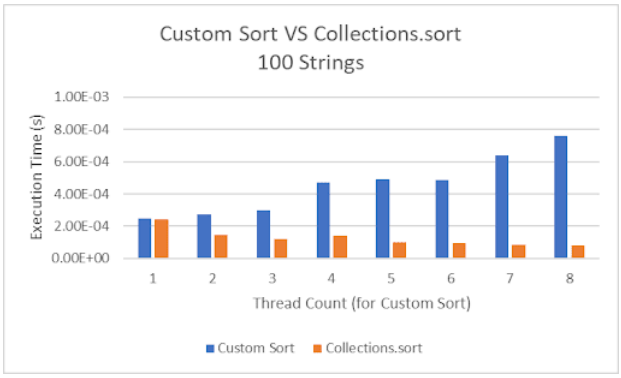

Fig. 3. Comparison of our concurrent sort versus Java's Collections.sort on 100 Strings.

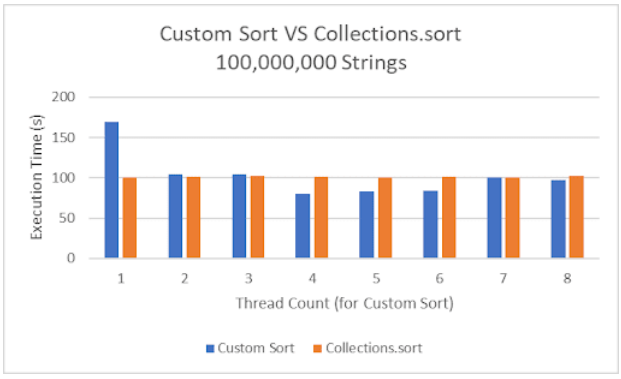

Fig. 4. Comparison of our concurrent sort versus Java's Collections.sort on 100,000,000 Strings.

Strings provided the worst sorting performance and our sorting algorithm, while still showing improvements with 4 or more threads at 100,000,000 Strings, showed only marginal improvements when compared to other data types.

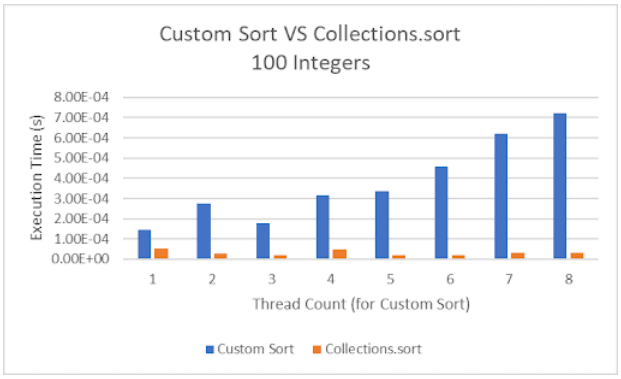

Fig. 5. Comparison of our concurrent sort versus Java's Collections.sort on 100 Integers.

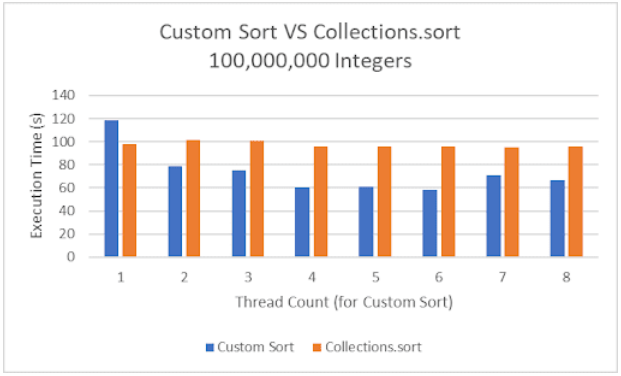

Fig. 6. Comparison of our concurrent sort versus Java's Collections.sort on 100,000,000 Integers.

Running our sorting algorithm on Integers seems to provide a better ratio of sorting speeds in favor of our sorting algorithm than it would when ran on other data types.

Discussion

Our testing of the multi-threaded approach to Java's Collection.sort() showed that the number of threads we used decreased the time it took to sort our data. This decrease in runtime follows what we expected where the time will decrease linearly as the number of threads increases but, plateaus out to the point where we do not see significant improvements in times at 5-8 threads. This is consistent with the works that we reference. Even when we are not using integers to test (because our CustomCollections.sort() can use generics) the runtime of the sort is still consistent so we can confirm that having a multithreaded approach will decrease the runtime of algorithms that use Java's Collection.sort(). We got to the point where we were cutting the runtime in half compared to Java's Collection.sort(). This does have its drawbacks as for smaller sizes that we wanted to sort, Java's Collection.sort() would beat out our program. When sorting 10000+ elements our CustomCollections.sort() would beat out Java's every single time but when it was below 10000+ elements it would be 3-5 ms slower compared to Java. Our reasoning for why this might be the case is that we believe

that there is a different sort than Java's Collection.sort() uses lower elements other than the multi-threaded merge sort that we used. We talked about maybe implementing a different multithreaded sorting algorithm for a low amount of elements as a solution to this problem.

Conclusion

TBD

References

[1] A. Trukhanov, "History of java. A full story of java development, from 1991 to 2021," CodeGym, 07-Sep-2021. [Online]. Available: https://codegym.cc/groups/posts/594-history-of-java-a-full-story-of-java-development-from-1991-to-2021. [Accessed: 10-Mar-2023].

[2] S. Rana, "Merge sort using multi-threading," GeeksforGeeks, 24-Feb-2023. [Online]. Available: https://www.geeksforgeeks.org/merge-sort-using-multi-threading/. [Accessed: 07-Mar-2023].

[3] "Stack Overflow Survey 2022," Stack Overflow, Jun-2022. [Online]. Available: https://survey.stackoverflow.co/2022/#most-popular-technologies-language. [Accessed: 10-Mar-2023].

[4] N. H. Minh, "Understand Java Collections and Thread Safety," Understand java collections and Thread Safety, 17-Jun-2019. [Online]. Available: https://www.codejava.net/java-core/collections/understanding-collections-and-thread-safety-in-java. [Accessed: 07-Mar-2023].

[5] S. K. Mehra, "Java Collections.sort( ) and arrays.sort( ) under the Hood," LinkedIn, 24-Jun-2020. [Online]. Available: https://www.linkedin.com/pulse/java-collectionssort-arrayssort-under-hood-surinder-kumar-mehra/. [Accessed: 06-Mar-2023].

[6] "Source for java.util.collections," Source for java.util.Collections (GNU Classpath 0.95 Documentation). [Online]. Available: https://developer.classpath.org/doc/java/util/Collections-source.html. [Accessed: 10-Mar-2023].

[7] Carrcodes's gists. Gist. (n.d.). Retrieved March 10, 2023, from https://gist.github.com/CarrCodes