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]. We also implemented a parallel insertion sort which would be used for lists smaller than 750 elements, though the multi-threaded insertion sort was not as effective as Java's Collections.sort().**

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

I. Introduction

Modern software applications have grown in size and complexity, and as a result, they are now capable of managing vast amounts of data. Due to the abundance of information that software applications handle, data management has become a crucial functionality in the technologies used to build them. The most common and expected form of data management in software applications is sorting, which involves organizing data in a specific order for easier access and processing. Although sorting algorithms have been optimized to their fullest extent, their processing speeds can potentially be improved even further by utilizing modern multi-processor execution techniques. In this project, we will be placing our focus on sorting and exploring how concurrency can enhance its performance. By incorporating multi-threading and other concurrency techniques, we aim to improve the speed and efficiency of sorting algorithms, ultimately providing a better experience for users of modern software applications.

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 have achieved 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 multi-threaded implementation of merge sort described and shown in GeeksforGeeks [2]. We modified the source material greatly to ensure that the new sorting algorithm can take multiple different data types, as well as operate with a multitude of possible thread counts.

B. Goals

Our initial goals for the milestone were to have a rough draft implementation of our CustomCollections.sort() algo-

rithm, 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.

C. Background

The two sorting algorithms that we used to implement our multi-threaded sorting functionality is Merge sort and Insertion sort. Merge sort "is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array" according to GeeksForGeeks [6]. The implementation for merge sort used is largely unmodified from the example given in the "Merge Sort Algorithm" article in GeeksForGeeks [6], with each partition creating two more partitions based on a passed in 'beginning' and 'end' indices, and comparing the values of the two partitions, until the array can no longer be partitioned. Insertion sort "is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part." according to GeeksForGeeks [7]. The implementation of insertion sort used was also obtained, and largely unmodified, from the example given in the "Insertion Sort Algorithm" article in GeeksForGeeks [7], with one element being selected from the list, and being checked with every element below it until it finds one greater than itself, where the selected element is pushed into the list below the other element.

II. Related Works

The default Collections.sort() algorithm in java is a single-thread implementation of the Merge Sort algorithm. 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 [8] to convert our implementation into one that handles generics, as well as a data structure that belongs to collections, rather than an array.

In addition, we wanted to make sure we do not deviate from our original goal, which was not to just implement a multi-threaded version of Collections.sort(), but also to ensure it is efficient. Although our sort was efficient for large inputs, we noticed that for smaller quantities our performance could be improved. To optimize our implementation, we wanted to grasp a deeper understanding of how Java's Collections.sort() handles different input sizes. This led us to an article "Java Collections.sort( ) and arrays.sort( ) under the Hood" [5] by Surinder Kumar Mehra that goes into detail about how Java implements different sorting algorithms based on the number of elements and the data types in a given data structure.

The article goes into detail about a multitude of sorting algorithms that have been implemented and altered over the years in Java. It explains the reasoning behind why these sorting algorithms continued to be reworked, and how the data type of the elements being sorted, namely primitive types versus objects, affects the performance of certain sorts.

Essentially, we learned that Java used to rely on Quicksort. However, this algorithm is unstable as it will not sort duplicate values in a consistent manner, making this only useful for primitive data types. Due to this, Collections.sort() was updated to use a TimSort when handling objects. A TimSort is based off of both Merge Sort and Insertion Sort. In addition, due to the worst case performance of Quicksort, it was then updated so that a Timsort is used when dealing with larger arrays, regardless of whether we are dealing with objects or primitive data types. Thus, a Quicksort is now only used for small arrays containing primitive data types, and this implementation makes use of Insertion Sort for even smaller ranges. As a result, we can observe that Insertion Sort is taken advantage of throughout both sorting scenarios, since TimSort is a derivation of it, and the QuickSort being implemented is optimized by it.

After analyzing that documentation and considering that we wanted our sort to be able to handle generics, we decided it would be best to create and add a multi-threaded Insertion Sort to our implementation in order to deal with smaller inputs.

III. Methodology

Our main objective is to create a parallel merge sort algorithm in Java that will run faster than the single threaded version, and even outperform the default Collections.sort() method. The reason we chose to work with merge sort is because it has a fast runtime of nlog(n) especially for larger datasets, and its divide-and-conquer approach can easily be parallelized. To achieve parallelism, we created a SortThread object that extends the Thread class to act as our threading mechanism.

Our parallel merge sort algorithm involves splitting the original data equally between threads and then performing a regular merge sort on each thread. After each thread has sorted its assigned data, we merge all the sorted data together at the end. To ensure thread safety, we decided not to use locks or parallel queues in our implementation. Instead, we

focused on keeping the algorithm simple to avoid any potential bottlenecks or complications that might arise from using more advanced techniques.

Our performance evaluations suggest that our parallel merge sort algorithm is best suited for large datasets, even when only a few threads are available. On the other hand, for relatively small datasets, the parallel merge sort may actually be slower than the single threaded version, and Collections.sort() is still the better option. Therefore, our implementation is likely to be more useful in applications that deal with large amounts of data, such as database management systems, where the use of additional threads for a short period of time is acceptable.

A. Challenges and Redesigns

Throughout the process of writing the code, we encountered a few notable issues that required several modifications and/or minor adjustments that needed to be made.

1) One of the primary challenges was that our sorting algorithm could only operate on integers. While it demonstrated comparable performance to other algorithms in this capacity, we sought to ensure that it could handle any form of data for proper testing. incorporate generic types in all of our methods. This approach guarantees that the sorting algorithm will work as long as the type implements the Comparable interface, which provides the ability to compare objects.
2) We encountered a second notable issue, which arose when we compared the performance of our algorithm to that of the default Collections.sort() function, we discovered that our algorithm was being outperformed by the Collections.sort() function when sorting small quantities of data. To address this issue, we initially attempted to optimize our algorithm by splitting the work between threads only when it made sense to use more than one thread. Unfortunately, this solution proved to be relatively poor, as it was only able to optimize the sorting process for a limited number of cases.

We investigated numerous sorting algorithms and looked into the documentation for Collections.sort(), in which three sorts are actually used based on sample size. Collections.sort() implements merge sort, insertion sort, and quicksort to achieve maximum efficiency, and with this information we decided to implement an insertion sort in situations when the number of elements is below a specific limit, in the case of our implementation 750 elements, at which merge sort would operate just as well.

B. Implementation

For our multi-threaded sorting algorithm We utilized Java's built-in "Thread" class, which we extended to create a new class called "SortThreads". These threads were then implemented to operate independently, in parallel with the main thread, with the objective of sorting data more efficiently.

To enable the program to handle any type of data, we made sure that our sorting algorithm accepts a generic Java "Object" that extends "Comparable". This ensures that the algorithm is capable of comparing the same types of data as Java's Collection.sort().

Our multi-threaded sorting algorithm works by dividing the input array into several smaller sub arrays, with the number of sub arrays being determined by the number of allocated threads. Each sub array is then worked upon by its own dedicated "SortThreads" thread, operating independently to speed up the sorting process. Once the threads finish their respective operations, they are joined using the "Thread.join()" method to ensure that the program waits for all the threads to finish their operations.

Finally, the sorted sub arrays are merged into a final sorted array, concluding the sorting operation. This merging process is critical to ensure that the final output of the sorting algorithm is in the correct order, with all of the smaller sorted sub arrays combined to form one sorted output.

In order to make comparisons between Java's Collection.sort() and our multi-threaded sorting algorithm, we created a testing harness that would compare the execution time of each sort with differing values for certain parameters. Each of the lists created for comparison would be lists of Java's "Object" that extend the "Comparable" class. These lists would then be populated with random values (within certain ranges e.g. Strings only contain alphabetic characters) before being copied so that each sorting algorithm can operate on the same lists. The length of Strings would also be randomly generated (within a certain bound).

To ensure that the sorting algorithms worked correctly, we made sure to loop through each list created by the testing harness and confirm that the values were stored in the correct order.

After gathering the data, that data would be stored in .csv files which would then be used in Microsoft Excel which would be used to create the graphs displaying the difference in execution times for each sorting algorithm.

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 varying data types. The data types currently being tested are

- Characters
- Strings
- Shorts
- Integers
- Longs
- Floats
- 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

Due to the nature of testing sorting algorithms on large arrays, testing can be a time-consuming process. In addition, the testing was performed on personal devices that were also used for other purposes during the testing period. This could potentially impact the productivity of the threads used in sorting and may result in some inaccuracies in the collected data. For example, if the sorting threads had to compete with other processes running on the device for CPU and memory resources, this could slow down the sorting process and potentially impact the performance of the sorting algorithms. As a result, the collected data may not accurately reflect the true performance of the sorting algorithms in ideal conditions, and it is important to consider this when interpreting the results of the testing.

C. Results

In our experiments comparing our custom sorting algorithm with Java's Collection.sort, we observed that the performance of both sorting algorithms depends on several factors, including the thread count and the length of the input array.

Specifically, we found that when the thread count is set to 1, our custom sort algorithm always performs worse than Java's Collection.sort because it fails to take advantage of parallelism. However, as we increase the thread count, the performance of our custom sort improves, although the benefits from increasing the thread count quickly diminish. Furthermore, on small input arrays, our sort algorithm performs significantly worse than Java's Collection.sort.

It's important to note that some of our experimental results deviate from this general pattern and are outliers. We suspect that these outliers may be due to external variables beyond our control, such as fluctuations in the operating system's scheduler that cause the threads to experience unexpected slowdowns. In our data analysis, we plan to ignore these outliers as they do not appear to represent the typical performance of the sorting algorithms.

Interestingly, we observed that our sorting algorithm performs best when its input consists of "Character" types. This could be due to the fact that "Character" types contain fewer bits than other types, leading to faster comparisons and thus faster sorting times. On the other hand, we found that our sorting algorithm performs worst on input arrays consisting of "String" types. This is likely because sorting a string requires comparing each character within the string until a difference is found, leading to a more time-consuming sorting process.

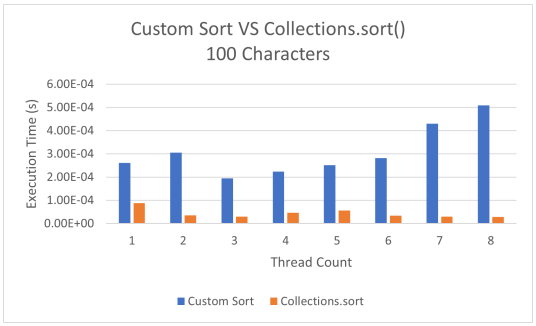

Fig. 1. Comparison of our concurrent sort versus Java's Collections.sort on 100 Characters.

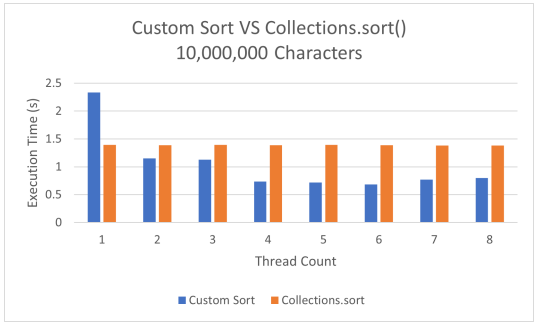

Fig. 2. Comparison of our concurrent sort versus Java's Collections.sort on 10,000,000 Characters.

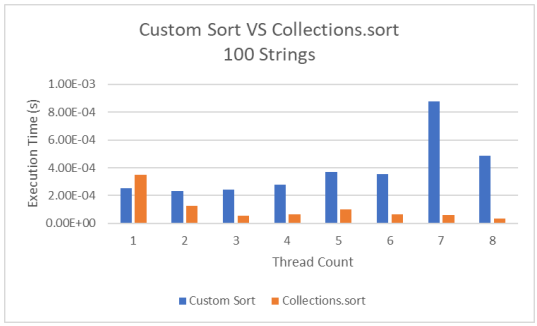

Fig. 3. Comparison of our concurrent sort versus Java's Collections.sort on 100 Strings.

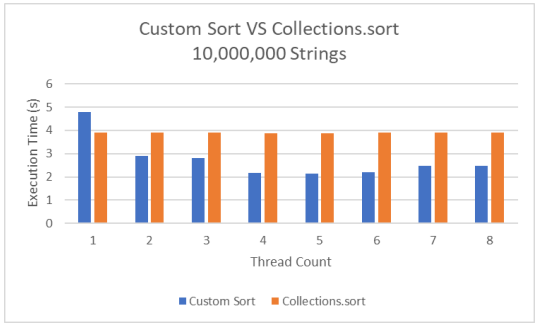

Fig. 4. Comparison of our concurrent sort versus Java's Collections.sort on 10,000,000 Strings.

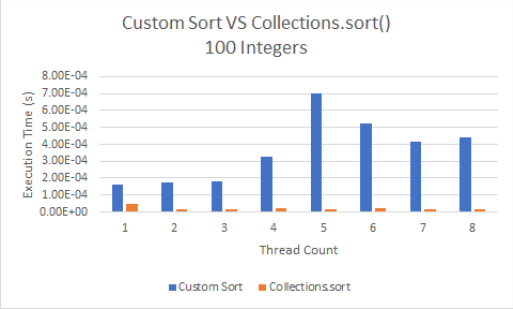

Fig. 5. Comparison of our concurrent sort versus Java's Collections.sort on 100 Integers.

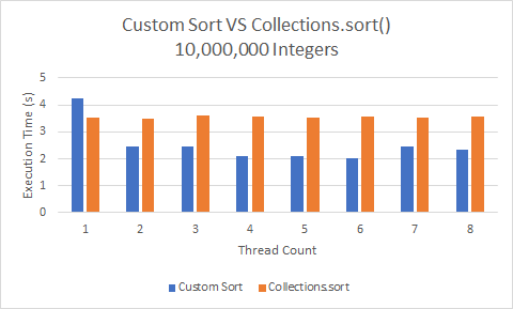

Fig. 6. Comparison of our concurrent sort versus Java's Collections.sort on 10,000,000 Integers.

D. Testing Environment

All of the code was written, compiled, and tested using the standard Java build tools in a Windows environment. The primary tool used was Visual Studio Code, with the Java extension and the latest Java SDK. Compilation and execution commands were ran within Windows Powershell in a terminal opened inside of Visual Studio Code. The hardware that the tests were conducted on were:

- Intel Core i7 13700k (8 Performance Cores + 8 Efficiency Cores / 24 Threads)
  - Performance Cores: 3.40 GHz base, 5.30 GHz boost
  - Efficiency Cores: 2.50 GHz base, 4.20 GHz boost
- 32 GB DDR4 3600 MHz Memory
- Windows 10 Home 22H2

V. 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 multit-hreaded 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 that Java's Collection.sort() uses for lower numbered elements other than the multi-threaded merge/insertion sort that we used. Also, the overhead for creating multiple threads and then combining their individual slices of data could outweigh the benefits of having multiple threads sort the array at once when the number of elements in the array is small.

Based on our experimental results, it can be inferred that our multi-threaded sorting algorithm is highly advantageous for programs that handle a vast amount of data, even if they have access to only a limited number of threads, but more than one. However, it should be noted that programs that deal with relatively small amounts of data could potentially experience negative impacts on their performance if they opt for the multi-threaded sorting algorithm over Java's Collection.sort(). This implies that the multi-threaded sorting algorithm may be best suited for systems that are specifically designed to manage databases since such systems typically handle large amounts of data and it is generally acceptable to temporarily use their additional threads for a short period of time. Overall, the suitability of the multi-threaded sorting algorithm largely depends on the specific use case and the available resources of the system.

VI. Conclusion

Our implementation of a multi-threaded approach to Java's Collections.sort() was designed to enhance the efficiency of sorting operations in large datasets. We based our implementation on a Geeks-For-Geeks thread-safe merge sort, which divides each array into two halves and assigns a thread to each half to sort them. Our implementation, called CustomCollection.sort(), is highly versatile as it supports two different types of sorting algorithms, depending on the sample size, and is designed to work with any type of data that extends the Comparable interface, including Integers, Strings, Characters, and Floats.

To assess the performance of our implementation, we created a comprehensive testing suite to compare the efficiency of CustomCollection.sort() with that of Java's Collection.sort(). We used this suite to evaluate the impact of different factors, such as the number of threads used and the size of the sample, on the performance of our algorithm. Our testing results showed a substantial improvement in the runtime of CustomCollection.sort() in larger sample sizes, outperforming Java's Collection.sort(). However, we observed that Java's Collection.sort() was able to achieve better performance in smaller sample sizes. To address this limitation, we plan to

make further improvements to the runtime of CustomCollection.sort() for smaller sample sizes. By doing so, we hope to achieve an overall better performance compared to Java's Collection.sort() and provide users with a more efficient and versatile solution for sorting their data.

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] "Merge sort algorithm," GeeksforGeeks, 31-Mar-2023. [Online]. Available: https://www.geeksforgeeks.org/merge-sort/#. [Accessed: 18-Apr-2023].

[7] "Insertion sort," GeeksforGeeks, 10-Apr-2023. [Online]. Available: https://www.geeksforgeeks.org/insertion-sort/#. [Accessed: 14-Apr-2023].

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

[9] "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].

APPENDIX

Algorithm 1 Parallel Merge Sort

function SORT($array$)
  $length \leftarrow$ length of $array$
  $exact \leftarrow length \bmod max\_threads = 0$
  if $exact$ **then**
    $lenPerThread \leftarrow length/max\_threads$
  else
    $lenPerThread \leftarrow length/(max\_threads - 1)$
  end if
  $lenPerThread \leftarrow \max(lenPerThread, max\_threads)$
  $threads \leftarrow$ empty list
  for $i \leftarrow 0$ to $length$ in steps of $lenPerThread$ **do**
    $beg \leftarrow i$
    $remain \leftarrow length - i$
    if $remain < lenPerThread$ **then**
      $end \leftarrow i + (remain - 1)$
    else
      $end \leftarrow i + (lenPerThread - 1)$
    end if
    $t \leftarrow$ new $SortThreads(array, beg, end)$
    add t to $threads$
  end for
  for all $t \in threads$ **do**
    t.start()
  end for
  for all $t \in threads$ **do**
    t.join()
  end for
  for $i \leftarrow 0$ to $length$ in steps of $lenPerThread$ **do**
    $mid \leftarrow i = 0 \ ? \ 0 : i - 1$
    $remain \leftarrow length - i$
    if $remain < lenPerThread$ **then**
      $end \leftarrow i + (remain - 1)$
    else
      $end \leftarrow i + (lenPerThread - 1)$
    end if
    merge($array, 0, mid, end$)
  end for
end function

Algorithm 2 SortThreads

procedure SORTTHREADS($array$, $begin$, end)
  super(
    () $\rightarrow$ **if** $(array.length > 750)$
      CustomCollections.mergeSort$(array, begin, end)$;
    else
      CustomCollections.insertionSort$(array)$;
)
  this.start()
end procedure