An Exploration of Different Concurrent and Multi-threading Based Approaches For Existing Sorting Algorithms.

Yohan Hmaiti, Noah Seligson, Rebecca Baker, Melanie Ehrlich, Mohammad Abdulwahab
University of Central Florida
{yohanhmaiti, nseligson18, bakerrj, melanie6104, mohammadhawani8}@knights.ucf.edu

***Abstract*—Numerous computer processes and applications require the sorting and filtering of dissimilar amounts of data of different types. The sorting of data is one of the most prominent problems in computer programming. Several procedures and algorithms were designed to tackle this task, however, research effort is still being led to producing more efficient and optimized approaches than the currently existing ones due to the constant need for faster, cost-savvy, and correct approaches that are more adequate for larger amounts of unsorted data. In this paper, we present different multi-threading and parallel computing-based approaches for existing sorting algorithms: *Merge Sort, Bubble Sort, Insertion Sort, Selection Sort, and Quick Sort*. We give background and related work to our topic, then we proceed to explain the design of each algorithm and the evaluation procedure. Afterwards, we present our results and discuss them based on time efficiency, cost-effectiveness, and memory saving.**

***Index Terms*—Sorting algorithms, Parallelization, Multi-threading, Merge sort, Bubble sort, Insertion sort, Selection sort, Quick sort, Time complexity, Space complexity.**

I. Introduction

In the field of computer science, one of the most relevant technical operations is the ability to sort. Sorting a data structure allows for the time optimization of other algorithms on a set of data.

For example, one of the most common algorithms for finding if an item exists in a list is binary search. Using time complexity notation, a linear search runs in *O(n)* time as the requested item can be anywhere in the list with no concise means of determining which locale of the list it resides in. However, binary search fixes this setback by traversing through the list logarithmic-ally, skipping over sections of the list that we know cannot contain the item. As a result, this algorithm runs in *O(log(n))* time.

However, binary search can only function if the list is sorted, or else it may turn out invalid results. The overhead of this, for a fast sorting algorithm, is an *O(nlog(n))* runtime added on. However, if a program intends on doing multiple searches, it is optimal to use a fast sorting algorithm nonetheless. Mathematically, this is because not sorting would lead to a *O(n*k)* runtime, while sorting would produce a *O(nlog(n) + klog(n))* runtime, where *k* is the number of times an item is searched for in the list. Thus, sorting can be a crucial component in making fast and efficient software algorithms, and it is important in knowing each of the different types of sorts and ways to product them. Each sorting algorithm has its own uses, and efficiencies, like how in the example, a sort such as **merge sort** may be preferred [6].

One way to implement a sorting algorithm is through the use of **concurrency**, also referred to as **parallelization**. Multiple **threads** can run the same program at the same time. This can be done in different ways. Some concurrent applications operate in that threads run the same segments of code concurrently, which can be done to get more results quicker, for example. Another technique with concurrency is for each thread to work on a different division of code. The purpose of this is to get different tasks performing, without having to fear about potential problems such as incorrectly modifying shared data structures.

Sorting algorithms have had parallel interpretations conducted before. These have come with varying results, but with an almost consistent indication of one relevant piece of information. This is, the use of concurrency with threads improves the runtime of sorting algorithms, almost dramatically. Even though there is a memory overhead cost with the use of threads, the amount of time saved can make parallelization an important factor in determining what type and implementation of a sorting algorithm should be used in a software project.

This paper seeks to develop parallel applications of five popular sorting algorithms: **Merge Sort, Quick Sort, Bubble Sort, Insertion Sort** and **Selection Sort**. Each version will include the use of a number of threads, with every thread having some amount of usage in the parallelized sort. We will evaluate the results of each sorting algorithm in different metrics. Several consistent test cases will be utilized among every sorting file to obtain each outcome. These findings will be compared to a non-concurrent, traditional version of the particular sorting algorithm.

Most importantly, our final results will be assessed and compared to the findings of several concurrent sorting algorithms in established literature. This information will allow for the assessment of what are the prominent upsides and downsides of introducing concurrency to a sort and if any improvements can be made to bring about better, optimized results.

II. Related Work and Background

We present in this section of the paper prior work related to the design of parallel and concurrent implementations of our algorithms of choice. We discuss the results found in the

literature, the advantages and disadvantages found along with the design recommendations given by the authors.

Prior work evaluated different parallel merge sort implementations to determine the optimal number of threads through a comparison of dissimilar multi-threading layers with different thread counts. The efficiency was assessed through an experiment where different devices executed sequential and concurrent implementations of the sort with similar data sets across all the machines available in the study. The findings in the literature show that the addition of more threads doesn't always bring about an improvement in execution runtime, such that spawning more threads than the cores available threatens optimal performance. It was found that the suitable thread count to be used by the algorithm is dependent on the number of cores available in the device. A single thread was found to be more advantageous (53%) than parallel execution for small data sets ($n < 10^5$). However, using 4 or 8 threads was found to be more efficient for larger data sets ($n > 10^5$) with a better performance reaching an optimization of (72% and 28%) for each [1].

Other work that compared different merge sort implementations: *bitonic merge sort, parallel merge sort, odd-even merge sort, serial merge sort, and modified merge sort* found that the suggested parallel merge sort implementation has a better runtime than serial merge sort, but with more extensive usage of resources. Design recommendations were made based on runtime, resource consumption, and area such that the study ranked the implementations based on the tradeoffs and needs in each case. Overall the suggested work presents the potential possibility to modify merge sort with the use of parallelization to minimize the execution time and reduce device resource usage, such that the authors also suggest future work that can combine different merge sort implementations with other dissimilar sorts to assess potential improvements and novel design and usage recommendations for data processing and sorting [8]. In addition, it was also found that parallel merge sorting can achieve a computational time complexity of O(n), where n represents the number of processes with one element for each process, such that the execution runtime changes logarithmically whenever the number of processes changes, and it changes linearly when its the number of elements that changes [9].

The following three papers describe related work on implementing quicksort as a parallelized function.

Prior work involved the implementation of a parallel quicksort and a comparison to other common sorting algorithms of the time. The algorithm is split into 4 phases: parallel partition, sequential partition, process partition, and sequential sorting. Diagrams and pseudocode are included to help describe these steps. The algorithm has the same time complexity of a sequential quicksort (O(nlog(n)) on average and O(n squared) worst case), but it performs faster by splitting up the work into blocks and having each block be processed by a thread simultaneously. Tests were performed on the Sun Enterprise 10000, a 32 processor machine. The algorithm's speed was compared to sample sort, the most popular sort at the time, as well as other common sorts for larger sets of data. It was found that sample sort was in most cases 15 percent slower than their quick sort algorithm. The paper concludes that with further hardware advances, the algorithm could easily be improved upon by improving synchronization between threads. The paper was written in 2003, so we should be able to improve upon their work with our faster machines [15].

Another group of researchers analyzed the respective efficiencies of three different implementations of quicksort: sequential quicksort, parallel quicksort, and hyperquicksort. Specifically, researchers wanted to see how much the parallel implementation sped up the algorithm. The programs were evaluated based on runtime and number of comparisons made. The paper contains flow charts demonstrating how each program works as well as datasets demonstrating the efficiency of each program. Hyperquicksort performed better than a standard parallel sort, which was better than a sequential one. This paper will be useful to look at for different implementations of the same general idea of quicksort. However, it does not compare quicksort to any other sorting algorithms, so we can do that in our paper [13].

A different group demonstrated a new algorithm for a parallel quicksort that uses a fetch-and-add method. This is used to efficiently do the partition phase of the quicksort. The adaptive scheduling algorithms included generate low overhead and ensure that each processor is doing roughly the same amount of work. The program executes at an average of O(logN) time on a computer with N processors assuming a constant time fetch-and-add. This may not be the case, however, so we will need to be careful with how we store and access our data to use this implementation. We can use this paper as a more in-depth example of a quicksort implementation that the other papers did not touch on [5].

There has been prior work related to creating a parallel implementation of insertion sort. In one such implementation, there is a merge function that calls itself three times in parallel and a recurrence relation is solved to get the runtime. The researchers then run their algorithms compared to their sequential counterparts for experiments, and calculate the speedup given.The parallel implementations are fast and portable, and the speedup from the sequential version was 1.1x to 280x [4].

Another algorithm was in place and was found to be faster than the quicksort. The experimental results for the new algorithm indicated less time complexity up to 70%–10% within the data size range of 32–1500 elements, compared to quicksort. The new algorithm minimizes the shifting operations within the standard insertion sort. After the experimental test, it was found that the new algorithm, BCIS (bidirectional insertion sort) had a faster runtime and made less comparisons between elements than the original insertion sort [10].

For a another insertion sort implementation, it is stated that it is known that insertion sort runs faster when the list is 'nearly' sorted but it runs slow when the list is in reverse order. The new algorithm proposed uses a while loop and recursive calls for the insertion sort. In the results, the proposed algorithm gives better performance in terms of number of

comparisons [2].

Another implementation sorts a one dimensional subbus machine which is an array of processors connected by a single communication bus. The researchers wanted to determine the best way to sort in place along one dimension. They use a parallel insertion model for the sorting and discuss a simple insertion step, and greedy sorting strategies for the algorithm to use. In the parallel insertion model, the data to be sorted is represented by a permutation p of 1, 2, ..., n, where n is the number of processors. After testing their implementations in a simulation, they find that left greedy and simple insertion sorting are optimal rather than one-way sorting [3].

There have been several implementations of a concurrent bubble sort. Typically, the eventual findings are that parallelizing bubble sort leads to faster runtimes, with the overhead cost of higher memory usage [11].

One version of a parallel bubble sort in particular uses that OpenMP (Open Multi-Processing) API to create and run threads. In this implementation, each of the *n* threads runs one loop of a standard bubble sort algorithm. The execution time of this was compared to a serial, non-parallel bubble sort algorithm with the same data sets over many different sizes. Ultimately, the parallel bubble sort dealed great time savings consistent throughout all but the smallest data set, among two trials. Excluding the outlier, the time savings ranged from [*52.54%, 99.84%*] [12].

Selection sort is an algorithm that is straightforward and simple, nevertheless there has been some work in the past that attempted to come up with a parallel selection sort. One of the algorithms that tried this was a a hybrid parallel sorting algorithm for GPU which combines parallel Radix sort and parallel selection sort. The name of the algorithm is "Split and Parallel selection" and first splits the data sequence into smaller sub sequences that are sorted concurrently using a fast sequential Radix sort. After sorting the smaller sequences, a parallel selection sort is performed using binary search to find the correct position of each element in the sorted data sequence. [7].

Another version that is intuitive and uses two threads is the Min-Max Bidirectional Parallel Selection Sort (MMBPSS). The MMBPSS is an improved version of traditional Bidirectional Selection Sort and Friend Sort Algorithms. MMBPSS reduces the number of loops required for sorting by positioning two elements in each round parallel. The sorting process divides the list into two parts, selects the minimum and maximum values from each part, compares both values to determine the minimum and maximum of the whole array, and places them at their proper locations. [14].

III. Sorting Algorithms

Throughout the literature, researchers categorize sorting algorithms based on their effectiveness or non-effectiveness relative to several factors [6]. In this section of the paper, we present the multiple categories of sorting algorithms based on the literature that are relevant to our investigation only. Then, we proceed to discuss the implementation of our novel sorting algorithms that use multi-threading and parallelization: ***Merge Sort, Bubble Sort, Insertion Sort, Selection Sort and Quick Sort***.

A. Different Types of Sorting Algorithms

Sorting algorithms can be described as either ***stable or unstable*** sorting algorithms. A stable sort refers to the case when two data entities remain in the same order even after the sort has occurred, wheres an unstable sort consists in the case where no element in the original data set remains in the same order in relation to another after the sort occurs. Moreover, some sorting processes use the memory as the only storage component during the sorting process, which brings about qualifying this process as an ***internal sort***. Usually a sort tends to be internal when the amount of unsorted data is relatively small or medium in size. Nevertheless, when additional storage space is required during the sorting process, the sort is then referred to as ***external sort***. This external sort usually occurs when the size of data is large and the process needs to break it down to smaller groups or chunks.

In addition, this also sheds light over ***in-place sorts and out-of-place sorts***, as during an in-place sort process, no extra memory is needed, and the original chunk of memory that was taken by the input data is the same that is overwritten by the output data. However, for an out-of-place sort more memory space is required to handle the changes in the data, which might require a size higher than the one of the input just for the helper variables and helper data structures. Another categorization of sorting algorithms consists in adaptivity, such that an ***adaptive sort*** reflects a sorting process where if an unsorted data set has some entities that are already sorted, those sorted components are directly disregarded and remain unchanged by the sorting algorithm. On the other hand, a ***non-adaptive sort*** sorts the full data set without exceptions and ensures the quality of the sorted data without exceptions made or pre-selectivity being done.

IV. Parallel Sorting Algorithms Implementation

In this section, we present our implementation of each sorting algorithm using multi-threading and parallelization [1]. Initially, we describe the basic and one-threaded approach of the sorting algorithm. Afterward, we present our own implementation, its characteristics, and the pseudo-code associated with each implementation.

1) ***Merge Sort****:* Initially, we load all the test files in the directory into a string array. Then for every test case available (*6 total*), we read the input of the current test case into an integer array, then the number of threads is set to 6 and we log the start time and initial amount of memory used before calling the merge sort parallel method that takes the array of unsorted integers for the current test case along with the number of threads. Once the execution of the parallel merge sort is done, the array is returned to the driver method and we log the end time and the quantity of memory used. The output is then

[1]https://github.com/YHmaiti/COP-4520—Project

printed to an Output.txt file, the results of *execution time and memory consumption* for each test case and for each input size are then written to a results txt file and to the console. Before going to the next test case, we first check that the array returned by the parallel merge sort implementation was sorted correctly otherwise we execute the process again for the current test case and report the new results.

The parallel merge method takes in an integer array of unsorted data and the number of threads to be used. If the array length is less than 2, we simply return the array directly. Otherwise, we first create an *ExecutorService* variable to instantiate a thread pool using the number of threads passed to the method. Next, we declare an array using the *Future interface* with a size equal to the number of threads used. By looping through the number of threads, we set at each iteration the start to be: $\frac{(arraylength)\times(i)}{numThreads}$ and the end to be: $\frac{(arraylength)\times(i+1)}{numThreads}$. Then we declare a helper sub-array to hold a copy of the original array from start to end. Then, we execute a call of the sequential merge sort method with the sub-array and we set this call to be submitted through the executor used and then stored in the future array at the matching index of the current iteration (i).

Once the looping process is complete, we build a results array by getting all the thread-submitted calls stored in the future array. Then, we loop as long as the number of threads is more than one, and we divide the results array into smaller arrays and then we merge them in parallel using different threads simultaneously. The while loop repeats this division and merge process in parallel as long as there is more than one subarray that needs to be merged. In the end, we shut down the executor of the threads and the final array is returned. We note that in the processes described, we rely on the use of helper 2D arrays, however, a 1D array is returned at the end.

2) ***Quick Sort****:* Main.java handles input and output as well as tracking runtime and memory usage. It reads all of the numbers in input.txt into an array, starts time, and sends the array to be sorted. Once all threads have shut down and the sort is complete, it stops time and prints the output to output.txt and the runtime and memory used to the standard output.

Quicksort is a class that implements the Runnable interface. When it is created, it is given the array to be sorted, the low and high indicies to start and stop sorting at, and the minimum size of the array to be sorted.

When Quicksort is run, the sort function is called. First, the function checks if the designated length is below the minimum length. If it is, it returns. A pivot value within the selected portion of the array is randomly chosen. Its position is swapped with the value of the given high index. A pointer variable is created that points the the current index for the next part. Next, the function iterates through the selected portion of the array. If the value of the current index is less than or equal to the pivot value, its position is switched with the value of the current index and the current index pointer is incremented. After the loop, the current index value is swapped with the value at the high index.

Algorithm 1: Parallel Merge Sort Using the Future and Executor Frameworks in Java

Input: Array (arr) of integers, number of threads $(numThreads)$
Output: Array (arr) sorted through the parallel merge sort
if $arr.length < 2$ **then**
  return arr;
end
$executor =$ newFixedThreadPool($numThreads$);
$futures =$ new Future[$numThreads$];
for $i = 0$ **to** $numThreads - 1$ **do**
  $start = i \times arr$.length / $numThreads$;
  $end = (i + 1) \times arr$.length / $numThreads$;
  $subArray =$ copyOfRange(arr, $start$, end);
  $futures[i] = executor$.submit(() $\rightarrow$ SimpleMergeSortWithoutParallelization ($subArray$));
end
$IntStream.range$(0, numThreads).mapToObj: (i -¿ try: return results[i] = futures[i].get() catch (InterruptedException — ExecutionException e): e.printStackTrace()
return null).$toArray(int[][] :: new)$
while $numThreads > 1$ **do**
  $midThreadCount = (numThreads + 1)/2$;
  $finalResults = results$;
  $results = IntStream.range$(0, numThreads / 2).mapToObj: ((i -¿ merge(finalResults[i], finalResults[i + getMidThreadCount])).toArray($int[][] :: new$)
  $numThreads = midThreadCount$;
end
$executor$.shutdown();
return $results[0]$;

After all of this, the function has to decide how to split the remaining work. If the length of the current selection of the array is greater than the minimum size, a new instance of Quicksort will be created and called to run on the lower half of the current selection. This quicksort will then run a sort on the higher half of the current selection. If the array is equal to the minimum length, the current quicksort will handle sorting both of its halves.

3) ***Bubble Sort****:* After obtaining the current test file and converting its content into an integer array, the algorithm begins with the creation of two threads. One thread is mapped onto the number 0, the other is mapped onto 1. Both threads are then executed where the Run function is called.

Run() starts by getting the value mapped onto by the currently used thread, representing a start index. The stop indices for the outer and inner for loops are then calculated. The bulk of the algorithm is a nested for loop. This falls under the normal bubble sort algorithm where two are compared and

Algorithm 2: Parallel Quicksort

```
Input: Array (arr) of integers, low and high indicies
length = high − low + 1;
if length <= 1 then
    return
end
pivot = randomarrelementbetweenlowandhigh;
swap(pivot, high);
curIndex = low;
for i = low, i < high, i++ do
    if arr[i] <= pivotVal then
        swap(i, curIndex);
        curIndex + +;
    end
end
swap(curIndex, high);
if length > minSize then
    newQuicksortquick(arr, low, curIndex − 1);
    executorservice.submit(quick);
    sort(arr, curIndex + 1, high);
end
else
    sort(arr, low, curIndex − 1);
    sort(arr, curIndex + 1, high);
end
return
```

Algorithm 3: Parallel Bubble Sort

```
Run():
start = map.get(currentThread)
if arr.length % 2 == 0 then
    stopOuter = testData.length/2
end
else
    stopOuter = (testData.length/2) + (1 − start)
end
stopInner = arr.length − 2
for i = 0 to stopOuter do
    for j = start to stopInner , j += 2 do
        if testData[j] > testData[j + 2] then
            swap(testData, j, j + 2)
        end
    end
end
return

Combine():
even = 0
odd = 1
len = testData.length
newArray = new int[len]
i = 0
while odd < len && even < len do
    if testData[even] < testData[odd] then
        newArray[i] = testData[even]
        even+ = 2
    end
    else
        newArray[i] = testData[odd]
        odd+ = 2
    end
    i += 1
end
if odd >= len then
    for j = i to len do
        newArray[j] = testData[even]
        even += 2
    end
end
else
    for j = i to len do
        newArray[j] = testData[odd]
        odd += 2
    end
end
testData = newArray
return
```

if the lesser-indexed element is greater, the values are swapped. The difference, however, is that rather than comparing the array's j'th index to the (j + 1)'th index, it's compared to the (j + 2)'th index. This is because each thread covers half of the indexes, effectively making each thread sort a unique half of the array. The previously mentioned start index is used to begin the inner for loop. One thread covers odd indices, the other even indices.

After both threads have completed execution, Combine() is called. This function starts by creating a new array equal to the length of the current test data array. There are two pointers, an odd pointer starting at 1 and an even pointer starting at 0. While both pointers are less than the length of the array, the minimum of testData[even] and testData[odd] is found. If testData[even] is less, testData[even] will be appended to the new array, and even will be incremented by two. Otherwise, testData[odd] will be appended and odd will be incremented by two. Once this while loop finishes, if odd or even is still less than the length of the array, the remaining odd or even index values are appended successively to the new array. The test data array is then set equal to the new array to fully complete the sorting process.

*4) **Insertion Sort**:* Insertion sort is implemented in the parallel function, which takes in the array of integers passed in from the main function, and an integer *max*, which is the maximum index of the array that will be sorted. That is, each call of the parallel function will sort the array up to array[max].

Within the function, there is a for loop that iterates through the array from index zero to the index at *max*, and inside this, the array is iterated over with the variable *i* used to track the current index of the array, and *target* is the value at this index. The variable *j* is used to hold the value of the index right before *i*.

If the value of the array at index *j* is higher than the one at index *i*, then the value at the index higher than *j* holds the value currently at index *j* and then the value of the *j* variable decrements by one. This simulates that all the elements at the array before *i* are iterated over so that any value greater than one to the left of it is moved down. After the iteration is completed, the index above the current value of *j* gets set to the original value of *i*, because all values to the left of this index should now be less than this value. This iteration is done for every index of the array so that the array is completely sorted.

Algorithm 4: Parallel Insertion Sort

Input: Array (arr) of integers, maximum index to sort (max)
if max <=$arr.length$ **then**
  $max = arr.length - 1$;
end
for $i = 0$ **to** max **do**
  $target = arr[i]$; $j = i - 1$;
  while $j < 0$ && $arr[i]$ >*target* **do**
    $arr[j + 1] = arr[j]$; $j = j - 1$;
  end
  $arr[j + 1] = target$;
end
return

*5) **Selection Sort**:* The way the parallel selection sort was implemented was by using the Min-Max Bidirectional Parallel Selection Sort technique. This technique uses 2 threads that will be running concurrently. One thread will select the min and max elements of the left sub-array, from start of the array to the midpoint. The second thread will obtain the min and max elements of the right sub-array, from the midpoint to the end of the array. The true minimum and maximum elements of the array are obtained by comparing the minimum and maximum elements of the left and right sub-arrays. Then the min element will be swapped with the starting element of the array and the max element will be swapped with the ending element of the array. The start of the array will be incremented and the end of the array will be decremented since we have found the elements that belong to these locations. The next iteration will work on the smaller array using the same procedure as described previously. The process will repeat until the start and the end indices of the array overlab or are equal.

V. Evaluation Methodology

To evaluate our parallel sorting algorithms and provide design recommendations, we designed our own testing methodology inspired from the literature. The evaluation consists in the use of three levels of unsorted data sizes ranging from *small* to *large*. The data nature is of *Integer* type and each testing level has two data sets, where the second in each level is $5 \times (currentLevelFirstDataSetSize)$.

A. Test Case Creation

A program was developed that generates an input file with $5 \cdot 10^6$ random integers. This file was used as a model to create 6 test case files of txt type that each of the programs used. Each generated file was a subset of the generated input file, reading the first *x* integers, where *x* was one of the following values:

Small size test cases	10^4	$5 \times (10^4)$
Mid size test cases	10^5	$5 \times (10^5)$
Large size test cases	10^6	$5 \times (10^6)$

Every algorithm incorporated the same testing files in order to maintain consistency as much as possible to prevent any disparities unrelated to the sorting algorithm and avoid any confounding variables.

B. Evaluation and Result Collection

Every program parses all the integers of the current testing file, converting them into elements of an array used to execute a particular sorting algorithm. This process is done 6 times per program, one for each test file. After the execution of the algorithm was completed, two metrics were recorded throughout the scope of the algorithm's execution: ***Execution time*** and ***Memory usage***.

*1) **Execution Time (ms)**:* In our context *execution time* refers to the exact time between the process of thread creation and the moment once all threads have successfully finished their execution and were joined.

*2) **Memory Usage (bytes)**:* In our context *memory usage* consists in the difference between the amount memory utilized once thread execution has completed and the amount used at the very beginning of the program.

We note that in the process of obtaining both instances of the value of total memory, the free memory was deducted from the total. This is because free memory is a component of total memory, making total memory a constant value. Thus, by deducting free memory we were able to calculate the actual amount of free memory that turned into used memory through the parallel sorting algorithm under testing.

VI. Results and Discussion

We have the results on Github, we already analyzed them, we just need to report here and write the future work.

VII. Limitations and Future Work

VIII. Conclusion

References

[1] M. Altarawneh, U. Inan, and B. Elshqeirat. Empirical analysis measuring the performance of multi-threading in parallel merge sort. *International Journal of Advanced Computer Science and Applications*, 13(1), 2022.

[2] P. S. Dutta. An approach to improve the performance of insertion sort algorithm. *International Journal of Computer Science & Engineering Technology (IJCSET)*, 4(05):503–505, 2013.

[3] J. Fix and R. Ladner. Sorting by parallel insertion on a one-dimensional subbus array. *IEEE Transactions on Computers*, 47(11):1267–1281, 1998.

[4] P. Ganapathi and R. Chowdhury. Parallel divide-and-conquer algorithms for bubble sort, selection sort and insertion sort. *The Computer Journal*, 65(10):2709–2719, 2022.

[5] P. Heidelberger, A. Norton, and J. Robinson. Parallel quicksort using fetch-and-add. *IEEE Transactions on Computers*, 39(1):133–138, 1990.

[6] Z. Inayat, R. Sajjad, M. Anam, A. Younas, and M. Hussain. Analysis of comparison-based sorting algorithms. In *2021 International Conference on Innovative Computing (ICIC)*, pages 1–8, 2021.

[7] S. Kumari and D. P. Singh. A parallel selection sorting algorithm on gpus using binary search. In *2014 International Conference on Advances in Engineering Technology Research (ICAETR - 2014)*, pages 1–6, 2014.

[8] J. Lobo and S. Kuwelkar. Performance analysis of merge sort algorithms. In *2020 International Conference on Electronics and Sustainable Communication Systems (ICESC)*, pages 110–115, 2020.

[9] K. Manwade. Analysis of parallel merge sort algorithm. *International Journal of Computer Applications*, 1(19):66–69, 2010.

[10] A. S. Mohammed, Şahin Emrah Amrahov, and F. V. Çelebi. Bidirectional conditional insertion sort algorithm; an efficient progress on the classical insertion sort. *Future Generation Computer Systems*, 71:102–112, 2017.

[11] D. Purnomo, A. Arinaldi, D. Priyantini, A. Wibisono, and A. Febrian. Implementation of serial and parallel bubble sort on fpga. *Jurnal Ilmu Komputer dan Informasi*, 9:113, 06 2016.

[12] R. Rihartanto, A. Susanto, and A. Rizal. Performance of parallel computing in bubble sort algorithm. *Indonesian Journal of Electrical Engineering and Computer Science*, 7:861–866, 09 2017.

[13] I. Singh, B. Kumar, and T. Singh. Performance comparison of sequential quick sort and parallel quick sort algorithms. *International Journal of Computer Applications*, 57:975–8887, 11 2012.

[14] K. Thabit and A. Bawazir. A novel approach of selection sort algorithm with parallel computing and dynamic programing concepts@@@ "" . (14), 2013.

[15] P. Tsigas and Y. Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2003. Proceedings.*, pages 372–381, 2003.

Algorithm 5: Parallel Selection Sort using Threads and the MMBPSS techneque

Input: Array (arr) of integers
$midPoint \leftarrow arr.length/2$;
$bound \leftarrow Boundry(0, arr.lengh)$;
while $bound.low < bound.high$ **do**
  $util \leftarrow$ new Utility();
  $Thread[]\ threads \leftarrow$ new Thread[2];
  $threads[0] \leftarrow$ new Thread(() -¿ $startT \leftarrow$ bound.start;
  for $i = startT$ **to** $midPoint$ **do**
    if $arr[i] < util.min1$ **then**
      $util.min1 \leftarrow arr[i]$;
      $util.min1Index \leftarrow i$;
    end
    if $arr[i] > util.max1$ **then**
      $util.max1 \leftarrow arr[i]$;
      $util.max1Index \leftarrow i$;
    end
  end
  );
  $threads[1] \leftarrow$ new Thread(() -¿
  for $j = midPoint$ **to** $bound.end$ **do**
    if $arr[j] < util.min2$ **then**
      $util.min2 \leftarrow arr[j]$;
      $util.min2Index \leftarrow j$;
    end
    if $arr[j] > util.max2$ **then**
      $util.max2 \leftarrow arr[j]$;
      $util.max2Index \leftarrow j$;
    end
  end
  );
  $threads[0].start()$;
  $threads[1].start()$;
  for $Thread\ thread$ ***in*** $threads$ **do**
    Try $thread.join()$;
    Catch$(InterruptedException\ e)$
     $e.printStackTrace()$;
  end
  if $util.min1 < util.min2$ **then**
    if $util.max1Index == bound.start$ **then**
      $util.max1Index \leftarrow util.min1Index$;
    end
    swap$(arr, util.min1Index, bound.start)$;
  end
  else
    if $util.max1Index == bound.start$ **then**
      $util.max1Index \leftarrow util.min2Index$;
    end
    swap$(arr, util.min2Index, bound.start)$;
  end
  if $bound.end - bound.start > 1$ **then**
    if $util.max1 > util.max2$ **then**
      swap$(arr, util.max1Index, bound.end)$;
    end
    else
      swap$(arr, util.max2Index, bound.end)$;
    end
  end
  s $bound.start++$;
  $bound.end--$;