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 produce 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, Concurrency, 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.

One of the most common algorithms for finding if an item exists in a list is binary search. 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 resulting in *O(log(n))* time.

However, binary search can only function if the list is sorted. 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, as in the example, where **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 performed simultaneously, without having to fear 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, because the use of concurrency with threads improves the runtime of sorting algorithms, almost dramatically. Even though there can be 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. We evaluate the runtime and memory consumption across different test cases chosen based on the literature. We compare our findings to other sorts and to a sequential version of the particular sorting algorithm. Additionally, we compare our findings to several concurrent sorting algorithms in the established literature. This information will allow for the assessment of 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

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].

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. Sample sort was in most cases 15 percent slower than their quick sort algorithm. With further hardware advances, the algorithm could easily be improved upon by improving synchronization between threads [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. Hyperquicksort performed better than a standard parallel sort, which was better than a sequential one [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(log(N)) 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 [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 was found 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. Their results show 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 afforded 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 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 is relevant to our investigation.

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. Sorting Algorithms Implementation

In this section, we present our implementation of each sorting algorithm [1].

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. This method takes the array of unsorted integers for the current test case along with the number of threads. After execution, the array is returned to the driver method and we log the end time and the quantity of memory used. The output is then printed to an Output.txt file, and the results of *execution time and memory consumption* 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

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

note that in the processes described, we rely on the use of helper 2D arrays, however, a 1D array is returned at the end.

Algorithm 1: Parallel Merge Sort

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(() → 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 $results[0]$;

*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.

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.

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

*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 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 works with half of the indices, 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.

*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 overlap or are equal.

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
```

V. Evaluation Methodology

To evaluate our parallel sorting algorithms and provide design recommendations, we designed our own testing method-

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
```

ology 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 of memory utilized once thread execution has been 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

Algorithm 5: Parallel Selection Sort - Input array (arr)

```
midPoint ← arr.length/2;
bound ← Boundry(0, arr.lengh);
while bound.low < bound.high do
    util ← new Utility();
    Thread[] threads ← new Thread[2];
     threads[0] ← new Thread(() -¿ startT ←
     bound.start;
    for i = startT to midPoint do
        if arr[i] < util.min1 then
            util.min1 ← arr[i];
            util.min1Index ← i;
        end
        if arr[i] > util.max1 then
            util.max1 ← arr[i];
            util.max1Index ← i;
        end
    end
    );
    threads[1] ← new Thread(() -¿
    for j = midPoint to bound.end do
        if arr[j] < util.min2 then
            util.min2 ← arr[j];
            util.min2Index ← j;
        end
        if arr[j] > util.max2 then
            util.max2 ← arr[j];
            util.max2Index ← 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 ← util.min1Index;
        end
        swap(arr, util.min1Index, bound.start);
    end
    else
        if util.max1Index == bound.start then
            util.max1Index ← 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 − −;
end
```

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

In this section, we present the results of our investigation based on the discussed evaluation methodology. We report the ***execution time (ms) and memory consumption (bytes)*** for each of our suggested parallel sorting algorithms per evaluation test case (see Table I and Table III). Additionally, for comparison purposes, we also report the sequential algorithm performance across all the test cases (see Table II and Table IV). Afterward, we discuss the implication of our findings and present design recommendations.

Note: Results classified under "Indeterminate" imply that the particular test case could not complete execution in a timely manner.

A. Discussion and Design Recommendations

1) Merge Sort: Based on our findings, our suggested parallel Merge Sort implementation has shown an improved execution time faster than the sequential implementation of the sort. Additionally, the parallel implementation had a faster runtime across all the data set sizes used in our evaluation compared to the sequential and parallel implementations of *Bubble Sort, Selection Sort, and Insertion Sort*. However, a minimal under-performance was recorded in terms of runtime compared to *Quick Sort*. According to our literature, it was found that for small data set sizes, the single-threaded merge sort implementation was more advantageous [1], however, our findings contradict that, such that even for small data set sizes of 10^4 and $5*10^4$ we had a better runtime through the parallel concurrent implementation compared to the sequential with optimization of around (42%).

For bigger data set sizes, our findings align with prior work, yet our implementation resulted in better performance by (68%). We note that in our experiment we used 6 threads along with the Java language based *Future and ExecutorService* interfaces. Consequently, to generalize our results and also our comparison, a follow-up investigation should replicate our suggested implementation using other languages such as: *C++ or Rust* to conclude more concrete implications of findings in terms of execution time.

The implementation of both the sequential and parallel merge sort requires the use of different objects and auxiliary arrays. Thus, the memory consumption is higher and more apparent compared to the other sorting algorithms. However, we note that there is an overall similitude between the memory consumption for both sequential and parallel Merge Sort implementations. Consequently, the parallelization process doesn't impact memory consumption, yet only optimizes runtime for the Merge Sort process. Moreover, we notice a positive correlation between the data set size and the memory consumption, such that with the increase of the data set size, the memory consumption also increases.

Our recommendation is that the parallel implementation should be used instead of the sequential one. Nevertheless, additional work is needed to generalize our results across different data types considering that only integers were used in our work, and also investigating a more dynamic approach to the choice of the number of threads might bring about valuable insights on a more optimized execution of the Merge Sort.

2) Quick Sort: At first, the sequential and parallel quick sort implementations seemed to work at about the same efficiency in runtime. In fact, at lower data sizes up to 10^6, the sequential sort often outperformed the parallel sort. However, there seems to be an exponential increase to the runtime of the sequential sort as it encounters even larger data sets. This makes sense, as quick sort has an average runtime of O(n log n). The parallel sort overtakes the sequential sort in efficiency at this point. This is likely because both implementations are following a similar procedure of recursing the same function on smaller subsections of the array, but the parallel implementation is simply able to handle multiple calls at once and keep its call stack smaller. The parallel sort also has the advantage in memory usage for this reason when sorting larger data sets. Its memory usage is less efficient for smaller data sets because of the amount of overhead required to handle multithreading. Compared to the other sorts implemented, quick sort has the fastest runtime but uses more memory than an n squared sort such as insertion sort.

3) Insertion Sort: The parallel implementation of insertion sort had a minimally faster execution time than the sequential version of the sort, and the memory consumption was higher for the parallel version due to the creation of multiple threads while the sequential version does not consume any memory. The performance time increase becomes higher when there are more elements to be sorted, as the speedup was only 1 ms for the 10^4 test case. As for memory usage, the memory consumption scales up with the increase in the input size, but like the execution time, can vary based on how many elements in the starting array will need to be rearranged. The sequential version does not consume any memory because it does not implement and start multiple threads.

Some design recommendations include using a bidirectional insertion sort (BCIS), recursively implementing the insertion sort, and utilizing a merge function. These recommendations would be used to optimize the execution time, but would likely require more memory usage. A bidirectional insertion sort implemented in [10] was found to have a faster execution time than quick sort in some cases, but it requires using insertion sort from both sides of the array which is similar to how quick sort uses a pivot. To implement the concurrent insertion sort recursively or to use a merge function would induce a performance speedup as less of the array would be sorted at each iteration. An implementation of the merge function was discussed in [2] which had more optimization of performance time. This incorporates an element from Merge Sort. In addition, a recommendation to optimize memory consumption would be to utilize libraries or data structures that could include better memory management.

Input Size	Merge Sort	Quick Sort	Selection Sort	Insertion Sort	Bubble Sort
10^4	17,519,760	6,752,328	7,003,800	503,336	188,760
$5 * 10^4$	94,371,840	3,994,288	31,457,280	703,352	200,016
10^5	197,305,784	16,616,936	88,253,024	1,006,672	522,240
$5 * 10^5$	203,963,016	20,969,472	31,415,952	4,180,544	2,099,200
10^6	91,854,648	28,311,552	Indeterminate	4,194,304	4,424,280
$5 * 10^6$	319,838,912	94,371,840	Indeterminate	42,446,376	Indeterminate

TABLE I: Parallel Sorting Algorithms Memory Consumption Across Different Test Cases (bytes)

Input Size	Merge Sort	Quick Sort	Selection Sort	Insertion Sort	Bubble Sort
10^4	17,334,120	249,472	0	0	0
$5 * 10^4$	96,115,864	1,887,208	1,314,080	0	0
10^5	200,973,752	4,192,256	2,097,152	0	0
$5 * 10^5$	254,944,616	17,825,792	139,668,688	0	0
10^6	330,119,640	17,827,840	Indeterminate	0	Indeterminate
$5 * 10^6$	385,170,760	187,695,104	Indeterminate	0	Indeterminate

TABLE II: Sequential Sorting Algorithms Memory Consumption Across Different Test Cases (bytes)

Input Size	Merge Sort	Quick Sort	Selection Sort	Insertion Sort	Bubble Sort
10^4	4	10	886	10	75
$5 * 10^4$	29	36	4,501	129	1,365
10^5	51	52	10,269	526	6,978
$5 * 10^5$	257	105	74,966	12,334	107,191
10^6	766	171	Indeterminate	48,485	3,461,898
$5 * 10^6$	3346	315	Indeterminate	1,235,674	Indeterminate

TABLE III: Parallel Sorting Algorithms Execution Times Across Different Test Cases (ms)

Input Size	Merge Sort	Quick Sort	Selection Sort	Insertion Sort	Bubble Sort
10^4	7	6	117	11	102
$5 * 10^4$	45	10	2,553	162	5,847
10^5	93	16	7,945	616	16,333
$5 * 10^5$	440	49	125,582	14,458	486,701
10^6	1,510	97	Indeterminate	57,245	Indeterminate
$5 * 10^6$	10,278	522	Indeterminate	1,443,860	Indeterminate

TABLE IV: Sequential Sorting Algorithms Execution Times Across Different Test Cases (ms)

Overall, the implementation of parallel insertion sort that was produced uses an approach that does not utilize approaches from other types of sorting algorithms, which leads to less memory consumption but a higher execution time compared to some of the other algorithms. The use of this implementation provides more of a speedup over the sequential version the higher the input size becomes. Whether to optimize for execution time or memory usage depends on the case under which the sort will run.

4) Bubble Sort: The parallel implementation of bubble sort completed every test case in less time compared to the sequential implementation. Specifically, and only including test cases that finished in a reasonable time for both versions, the time savings ranged anywhere from $[26.47\%, 77.98\%]$, with a mean time save of 59.59%. Compared to the findings of Susanto et al., the implementation of bubble sort included in this paper assembles similar results. Their findings presented an average time improvement of 62.85%. Compared to the average time saved by our parallel implementation, this is an adequate improvement. This could be a result of the study utilizing a total of 8 threads, while our version is limited to only 2 threads [12].

Memory usage analysis differs from the results found from time execution for the bubble sort implementations. For the parallel implementation, there is a positive correlation between additional memory usage and test case size. However, there is no such memory supplement at all for the sequential implementation. This discrepancy can likely be explained by object creation in the parallel version. Specifically, the parallel implementation involves creating two thread objects for the run function and a completely new array in the combine function. However, the standard sequential implementation never creates any new arrays or objects.

Overall, the amalgamation of these findings demonstrate

that the two-threaded parallel implementation of bubble sort as discussed can provide notable time-savings against a standard, sequential bubble sort application in a consistent matter. Consequentially, due to the creation of thread objects, there is an overhead in memory usage not seen in any capacity in a sequential implementation of bubble sort. With these factors in mind, recommendations about what particular version of bubble sort, sequential or parallel, should be utilized depending on if an improvement in runtime outweighs the negative effects of additional memory consumption.

5) Selection Sort: The performance of the parallel implementation of the selection sort algorithm was found to differ from the sequential implementation across varying input sizes. Initial test cases showed the sequential implementation outperformed the parallel version. However, for larger input sizes, the situation changed (see Tables III, IV) for an input size of $5 * 10^5$. Here, the parallel implementation of selection sort demonstrated improved performance compared to the sequential version. These findings suggest that the parallel algorithm is better suited for larger input sizes, while the sequential algorithm is more efficient for smaller input sizes. Based on these results, a suitable approach for implementing a selection sort algorithm in an application would be to choose the best-suited algorithm based on the input size.

The observed execution time differences can be attributed to the additional logic required by the parallel MMBPSS algorithm. In each iteration of the array loop, the parallel algorithm executes four if-statements, while the sequential version only executes one. Furthermore, the process of setting up and starting threads incurs additional overhead, as it involves allocating memory and initializing various data structures. Context switching and lock acquisition and release also add to the cost of dealing with threads. These factors are particularly relevant in the MMBPSS algorithm, which uses two threads, compared to the sequential selection sort that uses none.

VII. Limitations and Future Work

A. Merge Sort

Based on our findings, our suggested parallel and concurrent implementation of *Merge Sort* has shown an efficient performance across all the test cases used in our evaluation. Additionally, a better performance compared to most of the other sorting algorithms was recorded, except for Quick Sort which was slightly faster, along with a faster and optimized performance compared to the single-threaded sequential *Merge Sort* implementation. However, some limitations in our work consist in the presence of potential resource overhead, such that the use of our algorithm can have limited application scenarios. This limitation in applicability can occur when more resources are needed to allow a good stable execution of threads and smooth use of thread pools along with asynchronous computations.

We noticed that the performance of the algorithm was fast for the different test cases used that ranged from a small size 10^4 to a large size $5 * 10^6$. However, we still cannot generalize our performance finding for very large data sets, future work should consider testing the suggested parallel *Merge Sort* implementation with bigger data sets. Additionally, we have used a fixed size of threads, which can be considered a limitation to the scalability of the algorithm's performance. Thus, future work should investigate the use of a higher amount of threads, or even apply a dynamic thread allocation strategy for better scalability. Also, an interesting idea would be to explore new memory management maneuvers to optimize memory consumption, considering that *Merge Sort* relies on the use of auxiliary data structures that can heavily consume memory along with the possibly large size of the data, such that in our investigation we found that as the data set size increased the memory consumption increased excessively.

B. Quick Sort

A major limitation of quick sorts in general is the amount of memory used. The algorithm relies on recursion, which requires the computer to store each function call in the stack. This problem cannot be solved with multithreading. We can only mitigate its effects by using multiple threads to get to the base case and return function calls more quickly. The algorithm's efficiency can also vary based on the choice of a pivot. This implementation chooses a random pivot index in the specified area. This results in a better average runtime, but it also often leads to differing runtimes for the same data. Future work on parallel implementations of quick sort should investigate both larger data sets and larger amounts of threads. This implementation was tested on a 16-thread processor, so it would be a good idea to experiment with smaller and larger amounts of threads to see the impact on both runtime and performance. We were also constrained to creating data sizes that could be generated in a reasonable amount of time on our own hardware, so it would be interesting to see the function of runtime dependent on data size on a larger scale.

C. Insertion Sort

Some limitations that occurred with the production of the concurrent insertion sort implementation include processing power, performance, and the memory consumption. The processor available to use could take a maximum of 12 threads, so future work could discuss how insertion sort could be implemented on a machine with more processing power to see how performance would be impacted and if there would be better scalability for the larger test cases.

In addition, the algorithm currently sorts up to a certain index of the array on each thread, starting from the beginning, but future work could discuss having the sort be performed recursively, or using a merge function so that each thread only sorts a small portion of the array, and each smaller array is merged together after the sorting has been complete. This would take in elements of other sorts such as Merge Sort while the current implementation is highly based on the original sequential Insertion Sort, and this method may consume more memory, however there can be a decrease in execution time which could be discussed.

D. Bubble Sort

The main limitation that came with the production of the concurrent bubble sort program utilized was the strict constraint on the number of threads. The algorithm, in how it is right now, can only work with 2 threads. This can be explained by the implementation for each function. In the run function, the inner for loop increments *j* by 2 every iteration. This is so the current thread in execution only covers one set of indices (odd or even). This concept follows suit for the combine function, which includes odd and even pointers meant to set the values for the corresponding indices.

With this limitation, one way it can be improved in future works can be to modify the function algorithms to be obtainable with more threads. One such expanded implementation can include 4 threads. Essentially, this example can be constructed by having each thread, with a unique starting point each, cover every fourth index. This would be in a similar vein to how the current implementation has one thread skip every other index to cover the odd indices, the other covering every even index.

E. Selection Sort

The implementation of the parallel selection sort algorithm is technically vulnerable due to its implementation complexity. The algorithm involves keeping track of two minimum and maximum values simultaneously, which requires careful handling of many moving parts. For example, in Algorithm 5, the maximum element is swapped only if the size of the array is greater than one. However, an error occurred initially where the algorithm swapped the elements regardless of the sub-array size, which could have resulted in incorrect sorting.

Another limitation of this approach is the use of only two threads. This decision was made because implementing the algorithm with more than two threads would increase its complexity, as the algorithm would need to keep track of more than two minimum and maximum values. While this could be done, it would pose additional challenges and may require a more sophisticated implementation approach.

VIII. Conclusion

To sum up, sorting data is an essential task that can be tedious and costly to execute especially with large quantities of data. Research work presented an array of different sorting algorithms that differ in their best usage case, efficiency, and disadvantages based on different factors. In our investigation, we present various approaches to parallel-based sorting by suggesting new implementations to known sorting algorithms: ***Merge Sort, Bubble Sort, Insertion Sort, Selection Sort, and Quick Sort***. We described the implementation of each algorithm and discussed its characteristics. We executed an evaluation approach where we tested the single-threaded version of each algorithm against our novel parallel and multi-threaded one, then we compared all the algorithms against each other to present design recommendations and a discussion about the advantages and disadvantages of each implementation. Our implementation of the chosen sorting algorithms showed a time and space complexity that makes them competitive with others present in the literature, however, their usage is dependent on several factors: data size, hardware characteristics, memory and time affordances, etc. Thus, the choice of adequate implementation is case dependent.

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.