Republic of Yemen

Ministry of High Education

and Scientific Research

University of Modern Sciences

Deanship of Graduate Studies

Information Technology Program

جامعة العلـوم الحـديثة

UNIVERSITY OF MODERN SCIENCES

Time Complexity Enhancement for α-Stack Algorithm

By

Wasim Saeed E. Alaghbari

Supervisor

Prof. Ghaleb Al-Gaphari

A thesis submitted in partial fulfillment of the requirements for master’s degree of Information Technology.

2019

144١

(يَرۡفَعِ ٱللَّهُ ٱلَّذِينَ ءَامَنُواْ مِنكُمۡ وَٱلَّذِينَ أُوتُواْ ٱلۡعِلۡمَ دَرَجَٰتٖۚ وَٱللَّهُ بِمَا تَعۡمَلُونَ خَبِيرٞ ١١)

المجادلة

Dedication

I dedicate this thesis to two special women in my life who supported me to get to what I am. "For my mother and my wife."

Acknowledgment

In the first place I am thankful to God almighty praise be to him for fulfilling this humble effort for the sake of his satisfaction.

Then I’d like to thank my advisor, Prof. Ghaleb Al-Gaphari, for his help, support, and valuable remarks. In addition, I would like to thank my committee members for their efforts.

Also, I would like to thank the Department of Information Technology, at University of Modern Sciences, for all the support, cooperation, and guidance I received while pursuing my master degree.

Finally, I would like to thank my family for their unlimited patience and unconditional fondness.

الـمـلـخـص

تعتبر خوارزمية الدمج واحد من اشهر الخوارزميات المنتشرة في تقنية الترتيب، الإ أنها تعاني من بعض المشاكل اهمها الزمن المستغرق في تنفيذ الخوارزمية.وكنتيجة لهذه المشكلة، طورت العديد من الخوارزميات التي حسنت في زمن تنفيذ الخوارزمية وبالاخص عندما تكون البيانات مرتبة سابقاً (الحالة المثالية). من هذه التحسينات، خوارزمية ألفا–ستاك (α–stack algorithm) التي تعتمد على دمج التراتيب الجزئية الموجودة في البيانات والمسماة (Runs) . وعلى الرغم من أن خوارزمية ألفا–ستاك استطاعت أن تحسن في اداء خوارزمية الترتيب الأصلية وحل مشكلة الترتيب في الحالة المثالية الا أن القصور واضح في خوارزمية ألفا–ستاك عند تطبيقها في البرمجة المتوازية وكذلك عند تعاملها مع البيانات العشوائية (الحالة الاسوء).

في هذه الدراسة، قمنا بتطوير خوارزمية جديدة اسميناها تقسيم التراتيب (DRS) والتي اخذت مميزات خوارزمية ألفا–ستاك بالاضافة الى مميزات خوارزمية الدمج الاصلية. استطاعت خوارزمية تقسيم التراتيب أن تقلل من الزمن المستخدم في عملية الترتيب بالإضافة الى قدرتها على التعامل مع البرمجة المتوازية بكفائه عالية.

وقد اظهرت النتائج الأفضلية لخوارزمية تقسيم التراتيب مقارنة بالخوارزميات المختارة في المقارنة حيث وصلة نسبة التحسين في زمن التنفيذ –عندما كانت البيانات عشوائية بشكل كامل– إلى ٣٠% في البرمجة المتسلسلة و نسبة ٣٩% في البرمجة المتوازية.

Abstract

One of the most popular sort algorithms is Merge sort, although it suffers from some problems, the main problem of which is time complexity. For this reason, many algorithms were developed to enhance merge sort time complexity, especially in the best case. One of the natural merge sort algorithms enhancement is α-stack sort algorithm which has better performance than original merge sort and overcomes the merge sort best case problem. However, merge sort is still a powerful algorithm in parallel processing and external sort. In this study, a new version of merge sort called Divide-Runs Sort (DRS) algorithm is developed. The DRS algorithm takes advantage of original merge sort and α-stack sort algorithm. The DRS reduces time complexity of original merge sort and α-stack sort algorithm and it overcomes a parallel processing problem in α-stack sort algorithm and the best case problem in merge sort. The results show noticeable enhancement in time complexity -when dataset is random- comparing with benchmarks which reaches to 30% in sequential processing and 39% in parallel processing.

Keywords: Sorting, Merge sort, Adaptive sort, Natural merge sort, parallel sorting, α-stack sort.

List of abbreviations

Abbreviation	Meaning
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GNU	*Gnu's* Not Unix
RAM	Random Access Memory
MCSTL	Multi−Core Standard Template Library
TPL	Task Parallel Library
LSD	Least Significant Digit
MSD	Most Significant Digit
DRS	Divide−Runs Sort algorithm
2D	Two−Dimension Array

List of Figures

List of Tables

Contents

1. Introduction

This Chapter explores the motivation of study and problem statement. In addition, other sections will be described such as, objectives, methodology, benchmark selection, and others.

1.1.Motivation

Merge sort and its modifications are very important algorithms in sorting. They are found in many fields of science such as Database, distributed systems, operating systems, and others [1] [2] [3]. One of Merge sort versions has been Python's standard sorting algorithm, and it is also used to sort arrays of non-primitive type in Java SE 7, on the Android platform, and in GNU Octave [4]. Although there are many versions of merge algorithms, sorting problem has attracted a great deal of research because efficient sorting is important to optimize the use of other algorithms and to save wasted time [5].

Moreover, parallel computers are appearing on our desktops. The advent of multi-core causes a major change in our approach to software. Parallel sort algorithms are highly useful in processing huge volumes of data in quick time. For this reason, the need of parallel sort algorithms is very important, especially when parallel sort algorithm is difficult to implement [6] [7].

1.2.Problem Statement

Since the recursive sort algorithms cannot recognize the nature of input data, it cannot decide whether these input data are sorted or not. For this reason, the algorithm would have poor performance especially in the best case. Such drawbacks in the original merge sort motivate many researchers to start modification of the original merge sort such as [8] [9] [4].These modifications which are called natural merge sort algorithms take place based on the concept of the existing order of the input called Runs.

Unfortunately, these algorithms including α-stack sort algorithm suffer from some problem:

- **The way of merge Runs management.**

  α-stack sort algorithm uses stack to manage Runs and finds efficient ways to merge them. Stack effects badly on algorithms because of the time for managing stack.

- **Difficulty of α-stack sort algorithm parallelization.**

  Since the size of unsorted set is decreasing with every iteration, it is difficult to parallelize sort algorithms that use iteration [6].

1.3. Research Objectives

The main objectives of this study are :

1. To design a new sort algorithm called DRS algorithm based on original merge sort algorithm and α−stack sort algorithm.
2. To implement DRS algorithm in terms of sequential process as well as parallel process.

1.4.Research Methodology

Different methodologies have been used in this study as the following:

- **Literature Review**
- **Building a model**

  A new model designed based on existing merge sort algorithms and its modifications.
- **Setting up an experiment**
- **Analysis of experiment results and measuring performance.**

  This study was evaluated empirically using MS visual studio C#. Quantitative data has been measured in terms of execution time. Many different scenarios have been tested in different dataset size.

1.5. Scope of the Study

The scope of this study is to develop a new sort algorithm which reduces time complexity in α-stack sort algorithm rather than space complexity and save wasted time in managing Runs.

In parallel processing, we will choose a well-known model and apply our proposed algorithm on it. Chosen model will be modified according to needs on proposed algorithm.

1.6. Benchmark Selection

Timsort algorithm [9] is a popular and standard algorithm for many platform and programing languages. On the other hand, α-stack algorithm [8] is one of new versions of natural merge sort. Buss and Knop [4] introduce new stable natural merge sort algorithms and compared these algorithms with Timsort algorithm and α-stack algorithm. Therefore, Timsort and α-stack algorithms have been chosen to be the benchmark of this study.

On the other hand, Uyar claimed that [6] parallel processing is difficult to apply on iteration sort algorithm and recursive algorithm are better choices. For the best of our knowledge, Merge sort is one of the powerful algorithms in parallel sorting; therefore, it has been selected to be the benchmark in parallel processing. We applied the same model that is implemented in parallel Sort method of java.utils.Arrays class of Java Library on both algorithms.

DRS algorithm will be compared with benchmarks in terms of execution time.

1.7.Contributions

This study proposes a new natural merge sort algorithm, which was developed base on original and natural merge sort. In sequential processing, a new model has been designed to take advantage of divide and conquer technique as well as Runs. The main contribution is developing a new natural merge sort which reduces time complexity in natural merge sort and apply on sequential and parallel processing.

The new algorithm shows promising results compared to benchmarks –in random dataset− which reaches to 30% in sequential processing and 39% in parallel processing.

1.8.Thesis Organization

The rest of this study is organized in 4 chapters. Chapter 2 is dedicated to literature review and related work. Sequential and implementation of parallel proposed algorithm are explored in chapter 3. In Chapter 4, analysis and performance measure are described. Finally, summary and conclusion are in Chapter 5.

2. Literature Review and Related Work

In this chapter, main concepts of sorting such as classification of sort algorithms and well-known sort algorithms were discussed. This concepts help readers and beginners to gain overall views of sorting techniques. Some sort algorithms in terms of sequential and parallel processing were discussed. Merge sort and some of its modification were discussed intensively.

2.1. Sort Algorithms

One of the most popular and important techniques in computer science is sorting. It is a permutation function, which operates on elements [10]. Sorting algorithms are found in many places in computer science. We can find sort applications in operating systems [11], database systems [2], image processing applications [3], programming languages, data communications, pattern matching [12], business applications, and applications that use large databases may benefit from efficient sorting algorithms. For example, computational biology, and search engines are privileged fields that need sorting in the geographic information system [13].Also, it plays an important role in teaching of algorithm analysis, data structure and programming [14].

For consequences, many algorithms have been developed. There are many different sorting algorithms and even more ways in which they can be implemented [15].Some algorithms work perfectly on number. Some can be implemented in parallel processing whereas other work only on sequential process. Each algorithm has its advantages and disadvantages. For example, Merge sort is well-known to perform very well in most practical situations, regardless of the fact that many other sorting algorithms have a better best-case behavior. For Many years, Researchers show big interest in developed and enhanced sort algorithm. In addition, with evaluation of multi-core processes, researchers apply sort algorithms and design them to work in parallel.

2.2. Factors affecting the Classification of Sort Algorithms

Sorting algorithms can be classified with various factors [15]. These classifications end up being important factors for programmers when they are writing a sorting algorithm or choosing which one to implement. In this section we'll focus on most important factors.

2.2.1.Time Complexity

The main factor and easiest way that classifies the sorting algorithm is time complexity or computational complexity. In general, it related to how much time an algorithm need to sort dataset. The time complexity analysis is a theoretical process to categorize the algorithm into a relative order among function by predicting and calculating approximately the increase in running time of an algorithm as its input size increases. For instance a program can take seconds, hours or even years to complete the execution, usually this depends upon the particular algorithm used to construct the program [15] [16]. To ensure the execution time of an algorithm should anticipate the worst case, average case and best case performance of an algorithm.

- **Worse case:** The worst-case analysis is the greatest amount of running time that an algorithm needed to solve a problem for any input of size n. The worst-case running time of an algorithm gives us an upper bound on the computational complexity.
- **Best case:** The best-case analysis is the least amount of

running time that an algorithm needs to solve a problem for any input of size n. In this the running time of an algorithm gives us a lower bound on the computational complexity. In most algorithms' analysis of the best case not consider because it is not useful. However, in sort technique, it considers as advantage or disadvantage for sort algorithm.

- **Average case:** The average case analysis is the average amount of running time that an algorithm needed to solve a problem for any input of size n. It is difficult to determine average case for algorithm. In general, the average case running time is considered as bad as the worst case and analysis as same way as the worst case.

2.2.2. Space Complexity

Space complexity of an algorithm is another factor that considers seriously when selecting an algorithm. There are two types of classifications for the space complexity of an algorithm: *in-place or out-of-place.*

An in-place algorithm is one that operates directly on the input data. The danger with this is that the data is getting completely transformed in the process of transforming it, which means that the original dataset is effectively being destroyed! However, it is more space-efficient, because the algorithm only needs a tiny bit of extra space in memory—usually a constant amount of space, or O(1)—which can be helpful if you don't have enough memory to spare.

In contrast, out-of-place algorithms don't operate directly on the original dataset; instead, the make a new copy, and perform the sorting on the copied data. This can be safer, but the drawback is that the algorithm's memory usage grows with input size [17].

2.2.3.Stability

A stable algorithm is one that preserves the relative order of the elements; if the keys are the same, we can guarantee that the elements will be ordered in the same way in the list as they appeared before they were sorted [15]. For instance if there are two elements a[0] and a[1] with the same value and with a[0] show up before a[1] in the unsorted list, a[0] will also show up before a[1] in the sorted list.

2.2.4.Comparison and Non-Comparison Based Algorithms

It's possible to classify a sorting algorithm based on how it actually does the job of sorting elements. Any sorting algorithm that compares two items -or a pair- at a time in the process of sorting through a larger dataset is referred to as a comparison sort. This subset of algorithms use some type of comparator (for example: >=

or <=) to determine which of any two elements should be sorted first [17].

Sorting algorithms that do not use any type of comparators to do their sorting are referred to as *non-comparison sorts* [17].

2.2.5.Recursive and Non-Recursive

A recursive algorithm means it calls itself with smaller input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller input. Usually the problem can be solved utilizing solutions to smaller variants of the same problem, and the smaller variants reduce to easily solvable instance, then one can use a recursive algorithm to solve that problem. Quick sort and merge sort are examples for recursive algorithms while insertion sort and selection are non-recursive since it does not follow these steps [15].

2.2.6.Internal Sort Vs External Sort

Because our machines can sort through large datasets fairly easily, it's common to have some applications that have to sort through huge collections of data. In some cases, this can actually amount to more data than can be maintained in the machine's main memory (or RAM).The way that an algorithm has to store data while its sorting through records is yet another way that we can classify sorting algorithms.

If all of the data that needs to be sorted can be kept in main memory, the algorithm is an internal sorting algorithm. However, if the records have to be stored outside of main memory —in other words, stored in external memory, in either a disk or a tape—the algorithm is referred to as an external sorting algorithm [17].

2.2.7.Adaptability

Whether or not the pre-sorted of the input affects the running time. Adaptive sort takes advantage of the existing order of the input to try to achieve better times, so that the time taken by the algorithm to sort is a smoothly growing function of the size of the sequence and the disorder in the sequence. In other words, the more presorted the input is, the faster it should be sorted. Algorithms that take this into account are known to be adaptive [18].

The following table depict from [19] which shows some sort algorithms and its classification

		Time Complexity					
		Best	Worst	Avg.	Space	Recursive	adaptive
Comparison Sort	Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	No	Yes
	Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	No	Yes
	Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	No	Yes
	Quick Sort	O(n.lg(n))	O(n^2)	O(n.lg(n))	O(1)	Yes	No
	Randomized Quick Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	Yes	No
	Merge Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(n)	Yes	No
	Tim Sort	O(n)	O(n.lg(n))	O(n.lg(n))	O(n)	No	Yes
	Heap Sort	O(n.lg(n))	O(n.lg(n))	O(n.lg(n))	O(1)	No	No
Non-Comparison sort	Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(n+2^k)	No	No
	Radix Sort	O(n.k/s)	O(٢^s.n.k/s)	O(n.k/s)	O(n)	No	No
	Bucket Sort	O(n.k)	O(n^2.k)	O(n.k)	O(n.k)	No	No

Table 1: Sort algorithms and its classification

2.3. Survey of Sorting Algorithms

Because the importance of sort technique, many algorithms had been designed. Many literatures have been describe sorting algorithms like [12] [15] [20] [21] [22]. In this section, main and popular algorithms discuss.

2.3.1. Bubble sort

Bubble sort is the simplest and popular sort algorithm. It compares two consecutive elements and swaps them if needed. This process continue until no need for swapping [12] [21] [22].

2.3.2. Selection sort

Selection sort is another well-known sorting technique that scans the list/array to find the smallest item, puts it at the first location in the list/array, and then scans the list for the second smallest item, puts it in the second location, then third smallest and so forth until reaches the largest item in the list putting it at the last location of the list. It has O(n^2) complexity, inefficient for the larger lists or arrays [23].

2.3.3. Insertion sort

Insertion sort is a simple and efficient sorting algorithm useful for small lists and mostly sorted list. It works by inserting each element into its appropriate position in the final sorted list. For each insertion it takes one element and finds the appropriate position in the sorted list by comparing with neighboring elements and inserts it in that position. This operation is repeated until the list becomes sorted in the desired order. Insertion sort is an in- place algorithm and needed only a constant amount of additional memory space. It becomes more inefficient for the greater size of input data when compared to other algorithms. However, in general insertion sort is frequently used as a part of more sophisticated algorithms [15].

2.3.4. Merge Sort

Merge sort [24] [25] uses the divide and conquer approach to solve a given problem. It works by splitting the unsorted array into n sub array recursively until each sub array has 1 element. In general, an array with one element is considered to be sorted. Consequently, it merges each sub array to generate a final sorted array. The divide and conquer approach works by dividing the array into two halves such as sub array and follows the same step for each sub array recursively until each sub array has 1 element. Later it combines each sub array into a sorted array until there is only 1 sub array with desired order. Merge sort is a stable sort meaning that it preserves the relative order of elements with equal key [15]. The figure1 show how merge sort works.

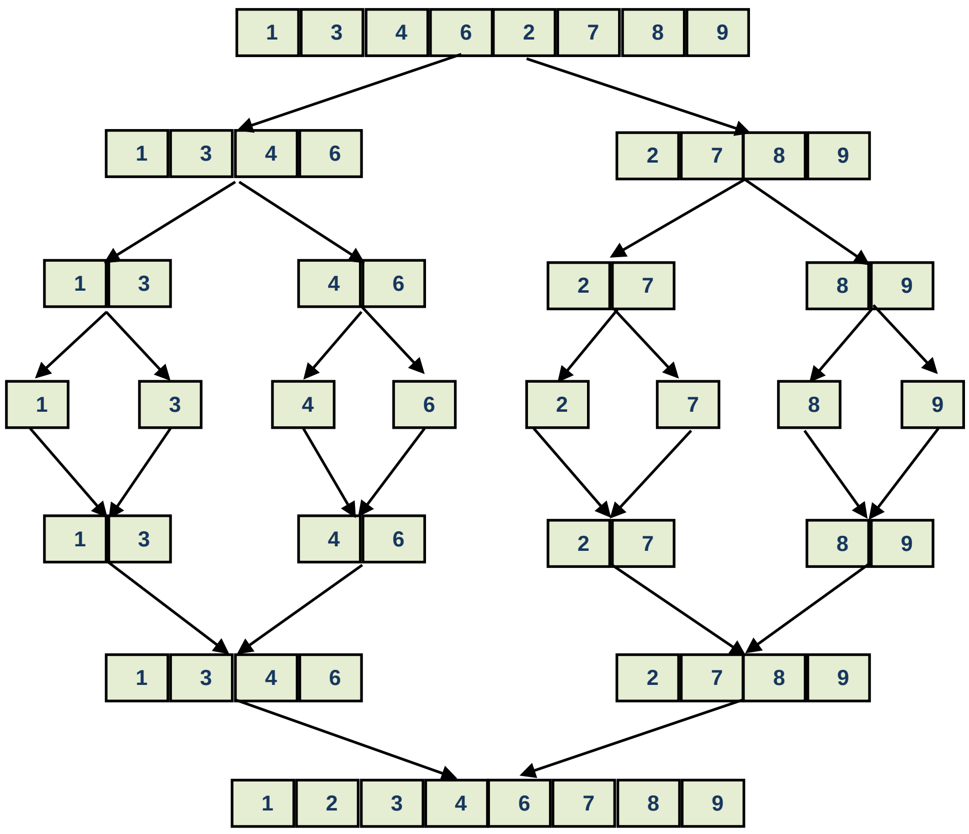


Figure 1: Merge sort Example

Mer

Algorithm 1: Merge−Sort (A ,p , r)

ge

sort

is

powe

rful

sort algorithm especially when working in parallel [26] [1], linked list, and external sort [27]. The algorithm for Merge sort is as follows.

```
If p < r then
    q = ( p + r ) / 2
    Merge−Sort ( A , p , q )
    Merge−Sort ( A , q+1 , r )
    Merge ( A , p , q , r)
End If
```

Merge sort suffers from two critical problems. First, it is not in-place algorithm, which need more space or auxiliary array to help in sort elements. These problem cost O(n) complexity in space whereas space complexity is O(1) for others sort algorithms. Nevertheless, in parallel and in external sort, auxiliary array give merge sort powerful in sort elements. Second problems, merge sort takes O(nlogn) complexity in all case even in best case. That mean, if elements already sorted merge sort cannot recognized that.

Many version of merge sort algorithms developed to solve space and time complexity. For Space Complexity, In-place Merge sort algorithm first published by Kronrod [28] showing that merging is possible without a workspace. After that, Trabb Pardo [29] presented the first stable in-place merging algorithm. Later Salowe and Steiger [30] observed an easy-to-correct error in the algorithm of Kronrod and made some simplifications to stable merging. According to the analysis of Pasanen [31], the algorithms developed by Huang and Langston [32] [33] have the lowest complexity with respect to the number of moves if a linear number of comparisons is approved.

For best case problem, the widely used solution is natural merge sort. It finds the sorted sub-lists by detecting consecutive runs of entries in the input which are already in sorted order. Natural merge sorts were first proposed by Knuth [12].There are many algorithms which merging strategies combined with decomposition into Runs such as TimSort [9],α-Stack sort [8], ShiversSort [34],AugmentedShiversSort [4], AdaptiveShiversSort [35], MinimalSort [36], PeekSort and PowerSort [37], NeatSort [38],Patience sorting [39], melsort [40], Splitsort [41], and 2-merge sort and α-merge sort [4].

2.3.5. TimSort

Timsort [9] is a hybrid stable sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data. It uses techniques from Peter McIlroy [42] and implemented by Tim Peters in 2002 for use in the Python programming language. Timsort has been Python's standard sorting algorithm since version 2.3, and it is also used to sort arrays of non-primitive type in Java SE 7, on the Android platform, and in GNU Octave [4]. Timsort has worst-case runtime O(n log n), but is designed to run substantially faster on inputs which are partially pre–sorted by using intelligent strategies to determine the order in which merges are performed. It is quite a strongly engineered algorithm, but its high–level principle is rather simple. The sequence S to be sorted is decomposed into monotonic Runs (i.e., non–increasing or non–decreasing subsequences of S), which are merged pairwise according to some specific rules [8]. Concurrently with the search for runs, the runs are merged with merge technique. Except where Timsort tries to optimize for merging disjoint runs in galloping mode, Runs are repeatedly merged two at a time, with the only concerns being to maintain stability and merge balance [43].

Looking for balanced merges, Timsort considers three runs on the top of the stack, X, Y, Z, and maintains the invariants as shown in figure 2:

i. $|Z| > |Y| + |X|$

ii. $|Y| > |X|$

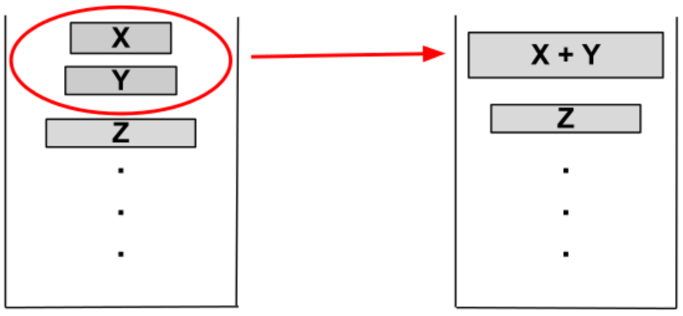


Figure 2: Stack and Merge Runs in Timsort

The runs are inserted in a stack. If $|Z| \leq |Y| + |X|$, then X and Y are merged and replaced on the stack. In this way, merging is continued until all runs satisfy i. $|Z| > |Y| + |X|$ and ii. $|Y| > |X|$.

Timsort performs an almost in-place merge sort, as actual in-place merge sort implementations have a high overhead. First Timsort performs a binary search to find the location in the first run of the first element in the second run, and the location in the second run of the last element in the first run.this call Individual merges [43].

An individual merge keeps a count of consecutive elements selected from the same input set. The algorithm switches to galloping mode when this reaches the minimum galloping threshold (min_gallop) in an attempt to capitalize on sub-runs in the data. The success or failure of galloping is used to adjust min_gallop, as an indication of whether the data does or does not contain sufficient sub-runs [43].

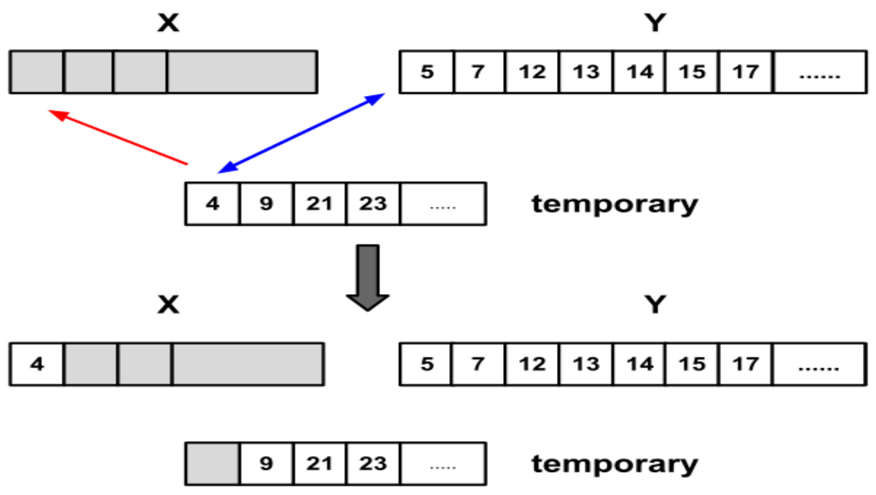


Figure 3: Elements (pointed to by blue arrow) are compared and the smaller element is moved to its final position (pointed to by red arrow).

When merging is done right-to-left, galloping starts from the right end of the data, that is, the last element. Galloping from the beginning also gives the required results, but makes more comparisons [43].

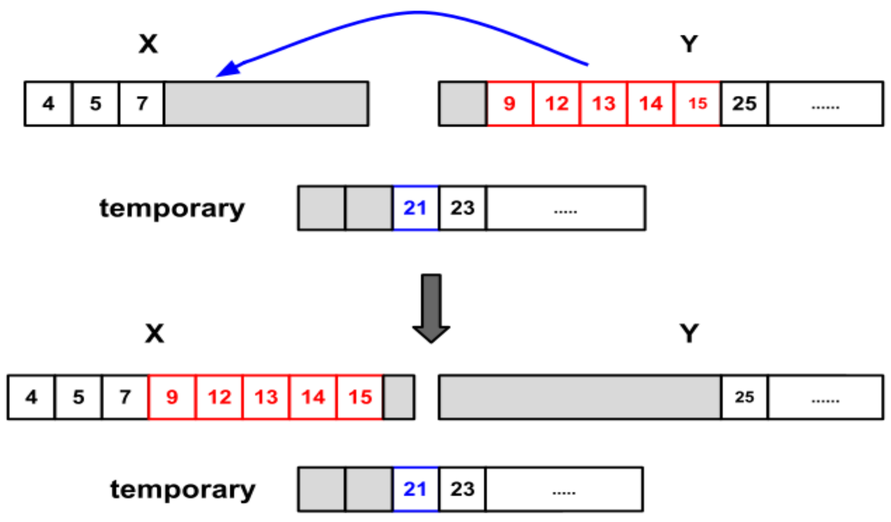


Figure 4: All red elements are smaller than blue (here, 21). Thus they can be moved in a chunk to the final array.

Galloping is not always efficient. In some cases galloping mode requires more comparisons than a simple linear search. While for the first few cases both modes may require the same number of comparisons, over time galloping mode requires 33% more comparisons than linear search to arrive at the same results [43].

2.3.6. α−Stack sort

One of the newest enhance on merge sort is an α−stack algorithm [8] which enhanced TimSort by improving stack that hold Runs. It consists in adding the runs one by one in a stack, and in performing merges on the go according some rules. This rules are always local as they only involve the runs at the top of the stack. A stack strategy relies on a stack X of runs that is initially empty. During the first stage, at each step, a run is extracted from R and added to the stack. The stack is then updated, by merging runs, in order to assure that some conditions on the top of the stack are satisfied. These conditions and the way runs are merged when they are not satisfied define the strategy. The second stage occurs when there is no more run in R: the runs in X are then merged pairwise until only one remains [8] .

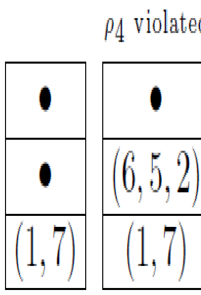


Figure 5: Statues of stack

Algorithm 2: Main Loop of α−StackSort

```
X=0
While R!=0 do
    R=pop(R)
    Append R to X
    While X violates the rule |Y|>=α|Z| do
        Merge Y and Z
    End While
End While
```

α−Stack Sort can be seen as a stack−merge algorithm of degree 2. It depends on a fixed parameter α > 1, and consists only in one rule which is |Y | > α |Z|. If it is violated, α consists in merging Y and Z [8] .

The table2 depicts from [8] to show average of execution time between α−stack and TimSort.

n	k	TimSort	α-Stack
10,000	-	129,271.65	129,178.79
10,000	10	33,581.81	33,479.05
10,000	100	68251.65	67,154.79
100,000	100	678,449.70	669,692.66

Table 2: Comparing between TimSort and α−Stack algorithms

2.3.7. Quick Sort

Quick sort [21] [22] [44] is the fastest general purpose internal sorting algorithm on the average among other sophisticated algorithms. Unlike merge sort it does not require any additional memory space for sorting an array. For the reason that it is widely used in most real time application with large data sets. Quick sort uses divide and conquer approach for solving problems. Quick sort is quite similar to merge sort. It works by selecting elements from unsorted array named as a pivot and split the array into two parts called sub arrays and reconstruct the former part with the elements smaller than the pivot and the latter with elements larger than the pivot. This operation is called as partitioning. The algorithm repeats this operation recursively for both the sub arrays. In general, the leftmost or the rightmost element is selected as a pivot. Selecting the left most and right most element as pivot was practiced in the early version of quick sort and this causes the worst case behavior, if the array is already sorted. Later it was solved by various practices such as selecting a random pivot and taking the median of first, middle and last elements. Quick sort is an in-place algorithm and it works very well, even in a virtual memory environment [15].

2.3.8. Radix Sort

Radix sort [45] is a linear sorting algorithm and works without comparing any element unlike other sorting methods such as insertion sort and quick sort. Radix sort works by sorting data elements with keys. Keys are usually represented in integers mostly binary digits and sometimes it considers an alphabet as keys for strings. Radix sort works by sorting each digit on the input element and for each of the digits in that element. In general, it might start with least significant digit and then followed by next significant digit till the most significant digit. This process somewhat considered to be unreasonable most of the time. Radix sort is a stable sort for the reason that it preserves the relative order of element with equal keys [15].

There are two classification of radix sort such as LSD and MSD. The Least significant digit method works by processing the integer representation starting from the least digit and shift in order to obtain the most significant digit. Likewise the Most significant digit works the opposite way [15].

2.4. Parallel sorting

Parallel sorting has been studied extensively during the past years. Sorting is a difficult problem to parallelize. Since the size of unsorted set is decreasing with every iteration, it is difficult to parallelize it. The recursive sorting algorithms are better suited for parallelization. They divide the unsorted data set into multiple segments and work on them independently [6] .Parallel sorting algorithms can be divided into two categories [46]:

- Partition-based Sorting: First, use partition keys to split the data into disjoint buckets. Second, sort each bucket independently, and then concatenate the sorted buckets.
- Merge-based Sorting: First, partition the input data into data chunks of approximately equal size and sort these data chunks in different processors. Second, merge the data across all of the processors.

With evolution of Multi-core processing, Many sorting algorithms have been develop to implement on parallel in GPU [13] [46] [26] and CPU [1] [6].In this section, the most popular parallel sorting algorithms discuss.

2.4.1. Parallel Merge Sort

A parallel merge sort algorithm proposed by Varman et al. [47] and popularized by the developers of the GNU Multi-Core Standard Template Library (MCSTL) [48]. In this algorithm, first the unsorted array is divided by the number of threads and each partition is sorted by one thread. Then all threads take part in merging the sorted partitions. Parallel merging is a complex process. The parallel version of the merge sort is shown at Figure 6 for four cores as implemented in parallel Sort method of java.utils.Arrays class of Java Library [6].

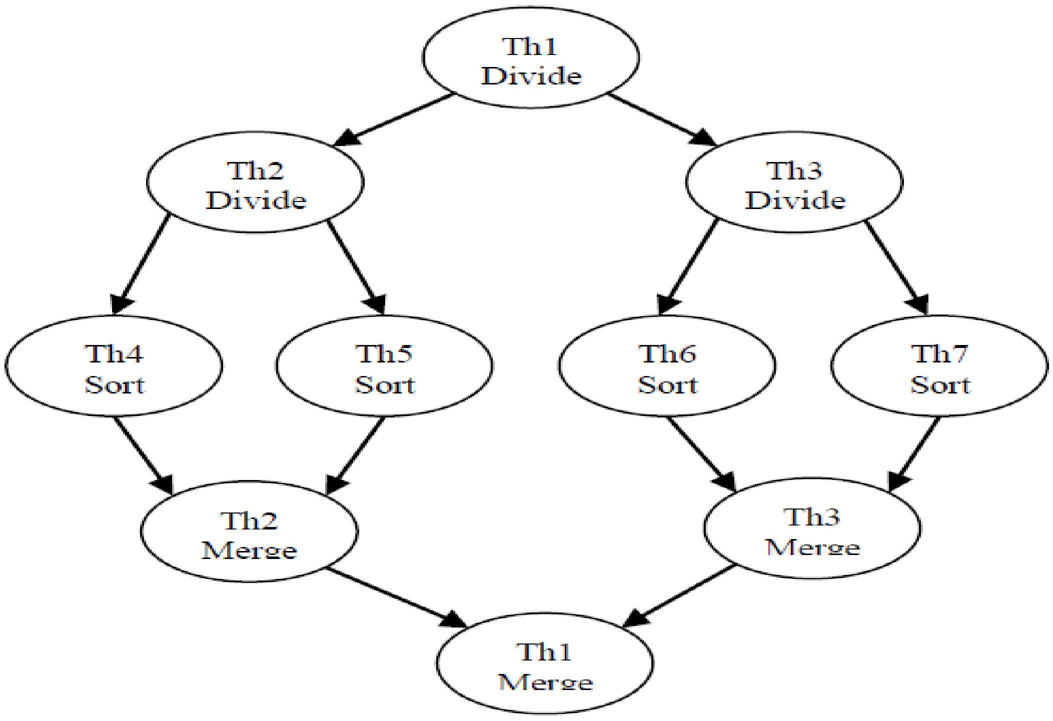


Figure 6: Parallel merge sort with 8 threads

It first divides the unsorted dataset recursively into two. This process continues until the number of unsorted subsets reaches to the number of cores in the system. Then, each core sorts one unsorted subset independently in parallel. They may use any single CPU sorting algorithm to sort their segments. Once, each thread is done sorting their parts, the process of merging starts. Each parent thread merges the two sorted subsections from its children threads. As the final step, the root thread merges two subsections from its children threads and produces the sorted dataset. In this algorithm, all four cores are utilized fully when sorting their subsections. However, when merging is performed, system utilization is reduced significantly. Only two cores are used at the first round of merge operations and the other two cores sit idle. In the final stage of the merge operation, only one core is used and the other three cores sit idle. As the number of cores increases in a system, the utilization of cores is reduced even more during the merge operations. Therefore, the primary objective of parallel merge sort algorithms has been to try to distribute the load of merging among more cores [6].

2.4.2. Bitonic Sort

Bitonic Sort, a merge-based algorithm, was one of the earliest procedures for parallel sorting. It was introduced in 1968 by Batcher [49]. The basic idea behind Bitonic Sort is to sort the data by merging bitonic sequences. A bitonic sequence increases monotonically then decreases monotonically. Bitonic Sort can be generalized for $n/p > 1$, with a complexity of $\Theta(n \lg^2 n)$. Adaptive Bitonic Sorting, a modification of Bitonic Sort, avoids unnecessary comparisons, which results in an improved, optimal complexity of $\Theta(n \lg n)$ [50].

The algorithm consists of two parts. First, the unsorted sequence is built into a bitonic sequence; then, the series is split multiple times into smaller sequences until the input is in sorted order. The bitonic split is a procedure that cuts one bitonic sequence into two smaller ones, where all the elements of the first sequence are less than or equal to the ones in the second. Looking at the example below, a bitonic sequence is divided between its two halves, and the *n th* element in each part is compared with each other. If they are out of order, they are swapped. Applying this procedure repeatedly onto the smaller lists, the result is a sorted sequence in ascending order [51].

Before the sorting can occur, the original sequence must first be transformed into a bitonic one. Note that two numbers by themselves are a bitonic sequence; from that, the sequence can be partitioned into smaller bitonic ones and then merged together.

The building algorithm is a variation of the bitonic split: two adjacent bitonic sequences are split and sorted in ascending order, the next two in descending order, and so on. The original two sequences are now a single bitonic sequence. This procedure continues until the entirety of the input has been converted.

2.4.3. Sample Sort

Sample Sort is a popular and widely analyzed splitter-based method for parallel sorting [52], [53]. This algorithm acquires a sample of data of size s from each processor, then combines the samples on a single processor. This processor then produces p−1 splitters from the sp-sized combined sample and broadcasts them to all other processors. The splitters allow each processor to send each key to the correct final destination immediately [54]. The algorithm is simple and executes as follows.

i. Each processor sorts its local data.

ii. Each processor selects a sample vector of size p−1 from its local data.

iii. The samples are sent to and merged on processor 0, producing a combined sorted sample

iv. Processor 0 defines and broadcasts a vector of p−1 splitters of the combined sorted sample.

v. Each processor sends its local data to the appropriate destination processors, as defined by the splitters, in one round of all-to-all communication.

vi. Each processor merges the data chunks it receives.

2.4.4. Radix Sort

Radix Sort is not a comparison-based sort. However, it can be parallelized simply by assigning some subset of buckets to each processor [55] [56].In addition, it can deal with uneven distributions efficiently by assigning a varying number of buckets to all processors every step. This number can be determined by having each processor count how many of its keys will go to each bucket, then summing up these histograms with a reduction. Once a processor receives the combined histogram, it can adaptively assign buckets to processors [54].

3. Proposed DRS Algorithm

This chapter describes different phases of the proposed algorithm called DRS such as design algorithm and implementation in terms of sequential and parallel processing. The proposed algorithm named Divide-Runs sort algorithm (DRS) is developed based on Divide and conquer technique as well as Runs. DRS takes advantage of original merge sort as well as natural merge sort. With the evaluation of multi-core processes, the advent of multi-core caused a major change in our approach to software. DRS is one of rare sort algorithm that can be applied on parallel processing, which is highly useful in processing huge volumes of data in quick time.

3.1. Design Sequential DRS Algorithm

3.1.1. Divide and Conquer technique

Divide and conquer (D&C) is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem [57]. A typical Divide and Conquer algorithm solves a problem using the following three steps:

1. **Divide**: Break the problem into sub-problems of the same type.
2. **Conquer**: Recursively solve these sub-problems.
3. **Combine**: Combine the solution of sub-problems.

3.1.2. Runs

Run is sub-order elements in the input array. At the same time, the order is non-descending or strictly descending, i.e. "$a0 \leq a1 \leq a2 \leq \ldots$» or «$a0 > a1 > a2 > \ldots$".

In some sort algorithms [8] [9] [4], sorting starts by looking for Runs which gives advantages to sort algorithm to reduce time complexity for sorting, especially if array is sorted or semi-sorted.

For example, for the following array

1	2	3	4	5	3	4	9	7	6	5	4	3	2	1

sorting algorithm starts by determining runs as follows

1	2	3	4	5	3	4	9	7	6	5	4	3	2	1
Run 1					*Run 2*			*Run 3*						

N=15

Then, sorting operation works based on Runs which use merge two sorted arrays.

3.1.3. Divide-Runs Model

As mentioned before, DRS is designed to take advantage of two algorithms: first, Natural merge sort algorithms which decomposes elements to Runs to solve the best case problem , second, original Merge sort algorithm which uses divide-conquer technique. For consequence, a new model was designed to combine advantages of merge sort and natural merge sort. The figure 7 shows the proposed model:

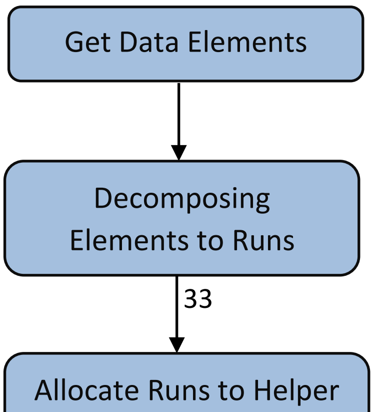


Figure ٧ : Divide-Runs Model

DRS model consists of five steps follows:

1. Get data elements to main array.
2. The elements are split into a run decomposition.
3. Runs allocate to Helper array to prepare for divide process.
4. Runs divide according Helper array.
5. Runs merge using merge technique.

3.1.4. Divide-Runs sort algorithm Design

First, Divide-Runs algorithm looks for runs in given array . Then first and last index in runs are stored in Helper array. Helper array contains two columns. The first column stores the first index of Run and the second column stores for last index of Run. For example, if we have an array with following elements:

1	2	3	4	5	3	4	9	7	6	5	4	3	2	1
Run1 (1-5)					*Run2(6-8)*			*Run3 (9-15)*						

N=15

This Given array contains 3 Runs ,The first one starts with index 1 and ends with index 5.The second starts with index 6 and ends with index 8.The third run starts with index 9 and ends with index 15.

All found Runs will be stored in 2-D array with 2 columns. The first column contains first index and the second column contains last index of the Run. This 2-D array is called Helper array.

First index	Last index
1	5
6	8
9	15

If any Run is a reversed order, Reverse function will be called, For example, Run 3 is a reversed array ,so it will be reversed to be

1	2	3	4	5	6	7
Run3 (9-15)						

Reverse Runs can be done at merge time.

Figure 8 presents the flowchart of looking for runs process and store Runs to Helper array.

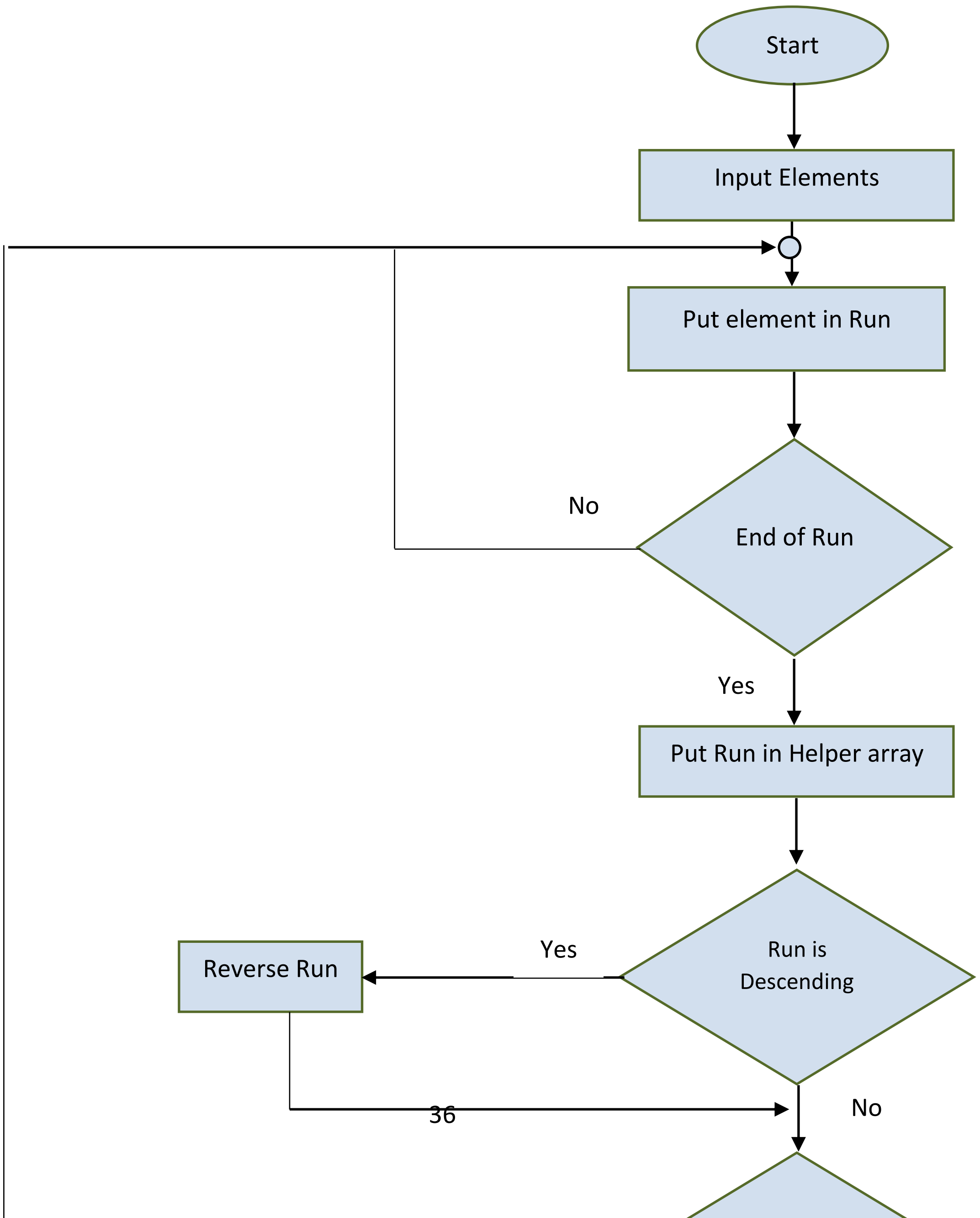
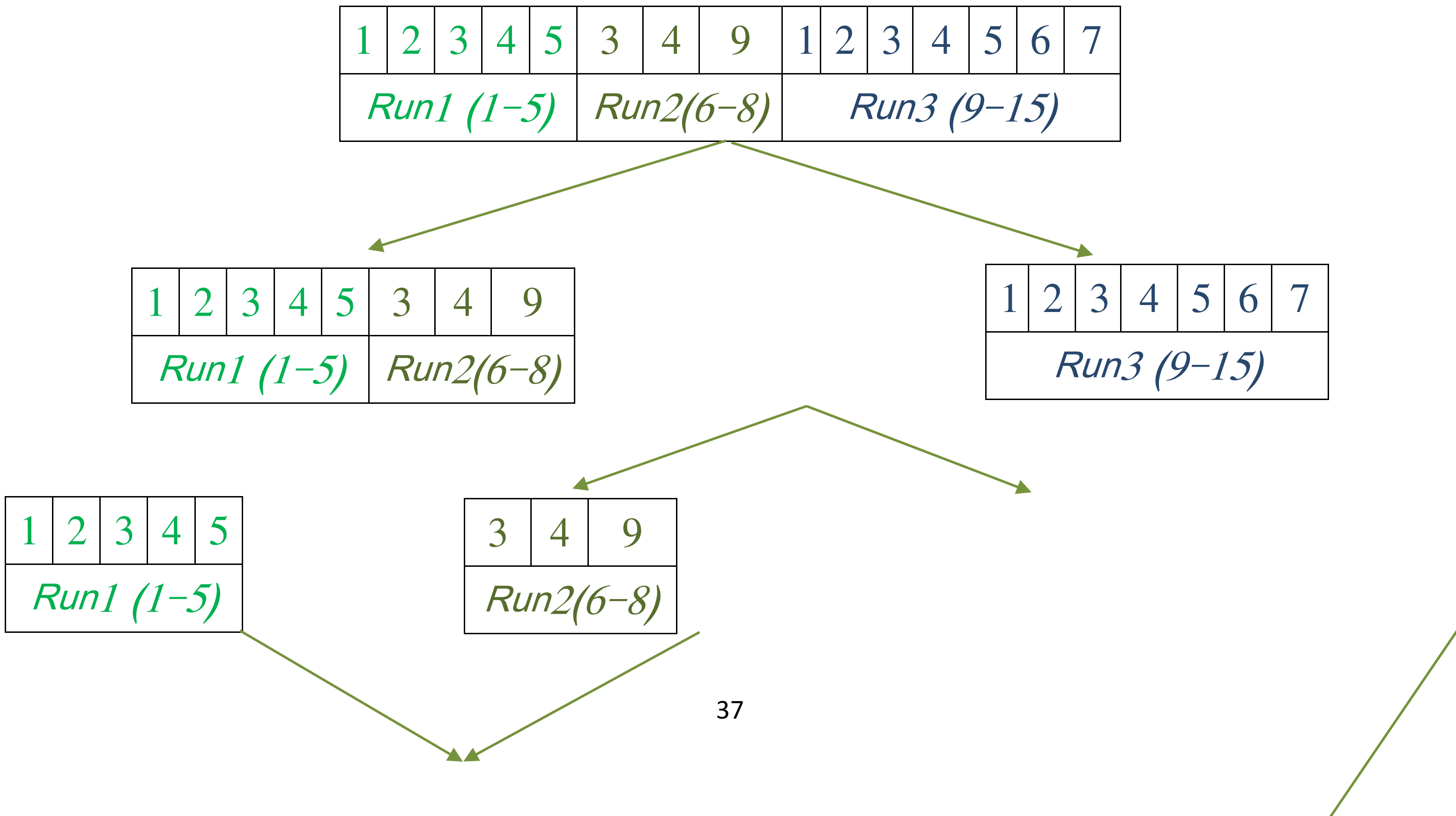

Start
Input Elements
Put element in Run
End of Run
No
Yes
Put Run in Helper array
Run is Descending
Yes
Reverse Run
No

Figure 8: Find Runs Flowchart

After finding all Runs in the array and storing them in Helper array, Divide and Conquer technique will apply on Helper array not like merge sort algorithm which applies divide and conquer on all given array.

Figure 9 shows how previous array will divide and merge



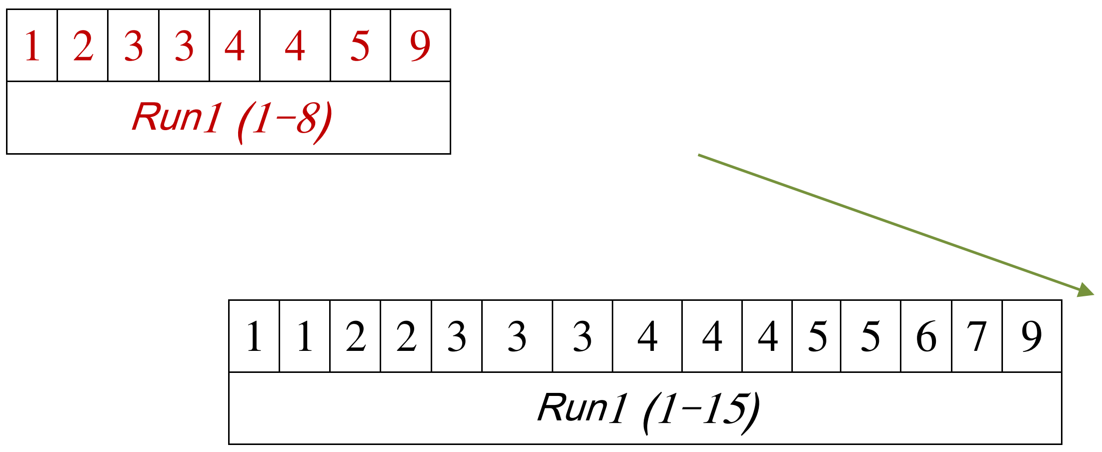


Figure 9: Divide-Runs algorithm Example

The following figure presents the flowchart for divide Runs using Helper array.

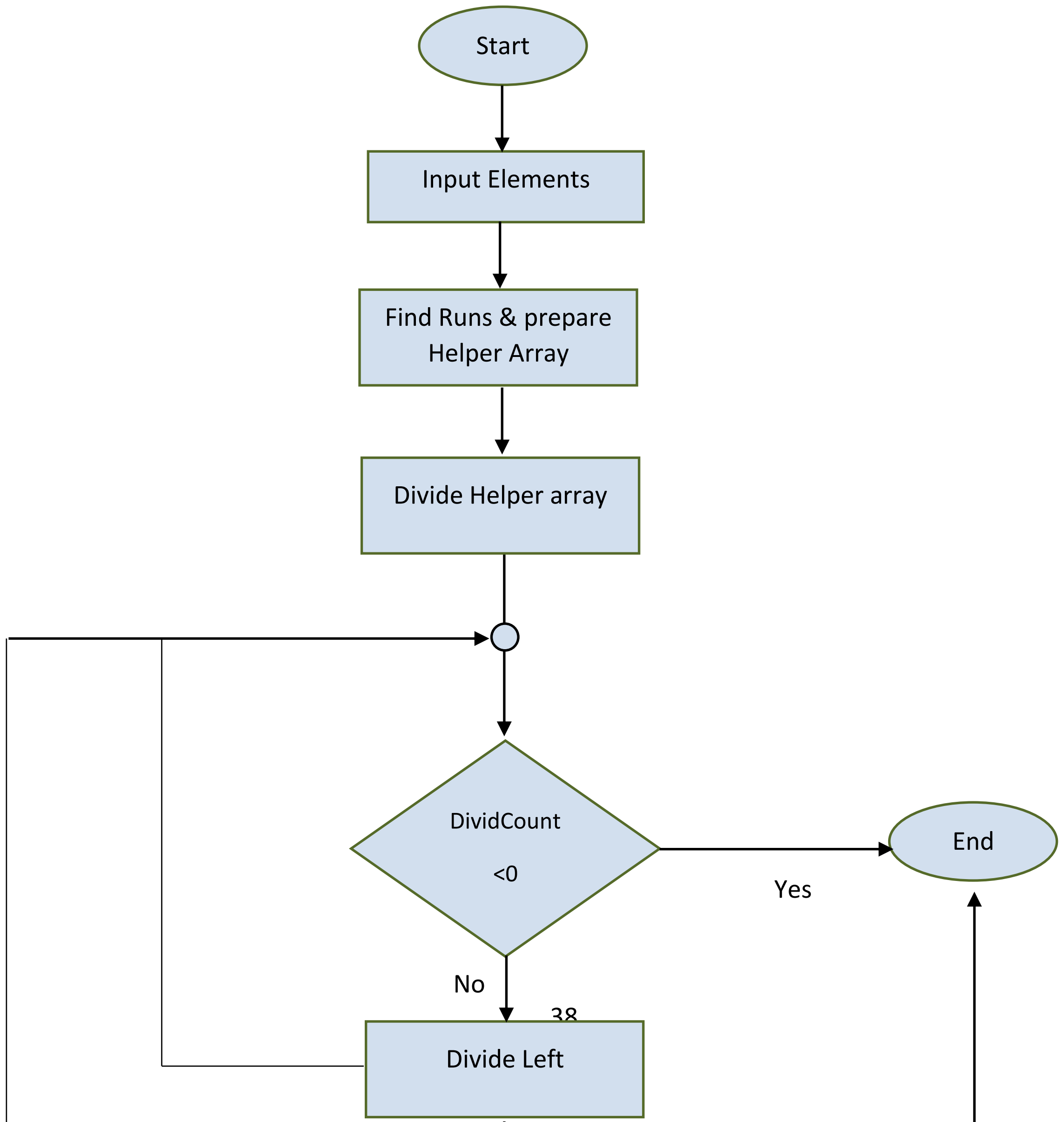
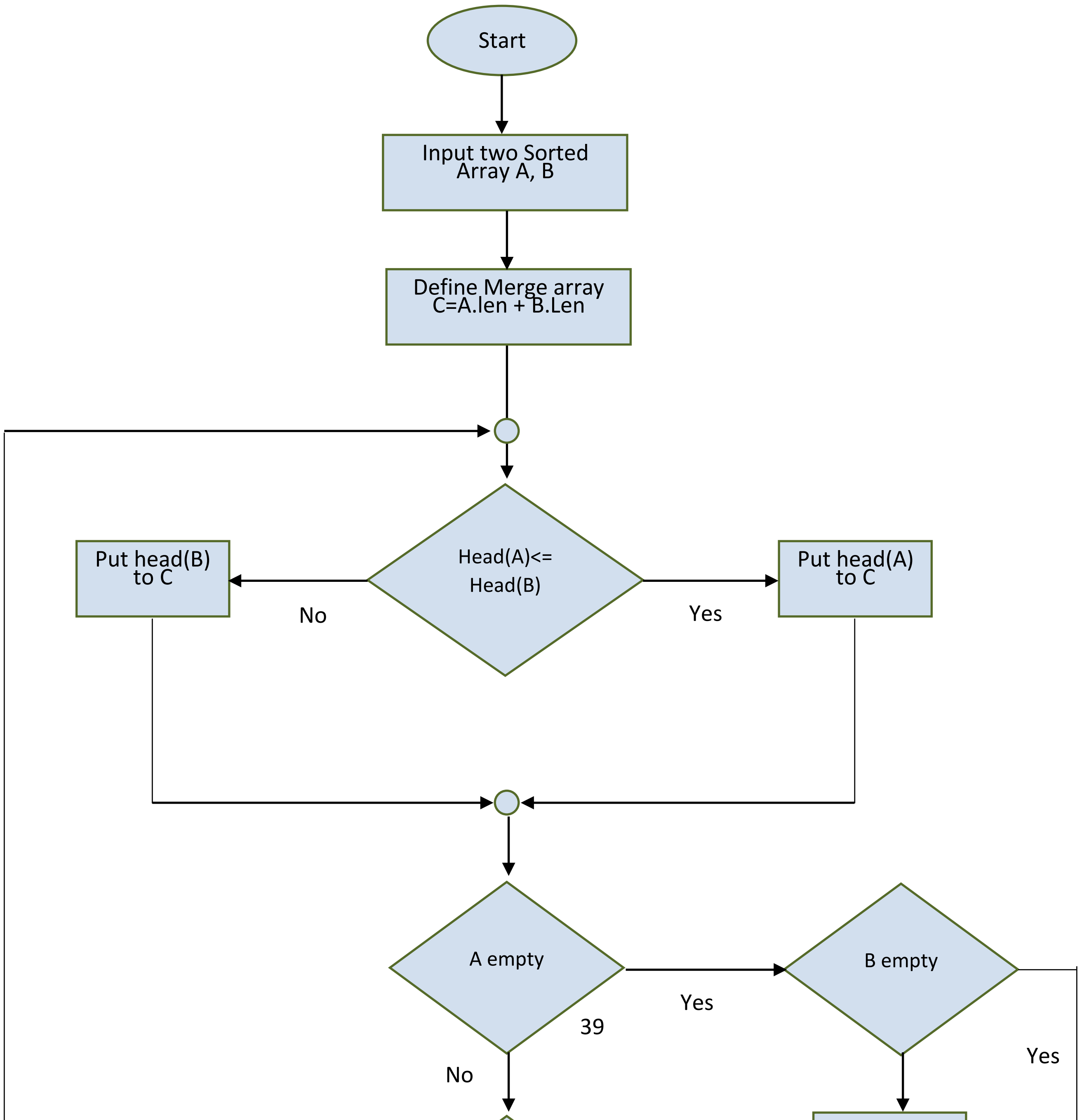


Figure 10: Divide Runs Flowchart

When divide Runs reaches to 1 run in divide left and divide right process, merge two sorted array works.

The following figure presents the flowchart for merge algorithm.



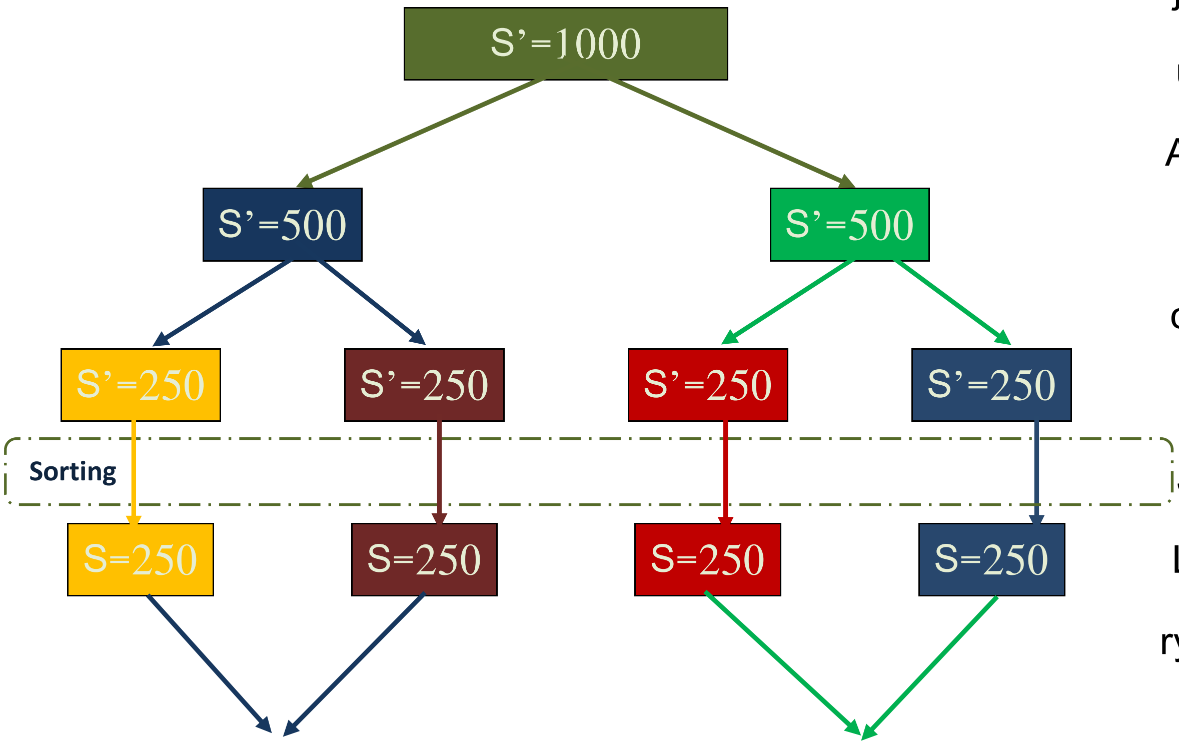
3.2.Design Parallel DRS Algorithm

Sorting is a difficult problem to parallelize [6].Since the size of unsorted set is decreasing with every iteration, it is difficult to parallelize it. For example, it is difficult to parallelize insertion sort, selection sort, and bubble sort. However, the better suited for parallelization is a recursive sort algorithms.

Divide-Runs sort algorithm is a recursive algorithm that takes advantage of divide and conquer recursively. It can apply on parallel as efficient as merge sort.

3.2.1. Design Parallel Algorithm

The parallel version of the merge sort is shown in Figure 12 for four cores as implemented in parallel Sort method of



java.
utils.
Array
s
class
of
Java
Libra
ry [6].

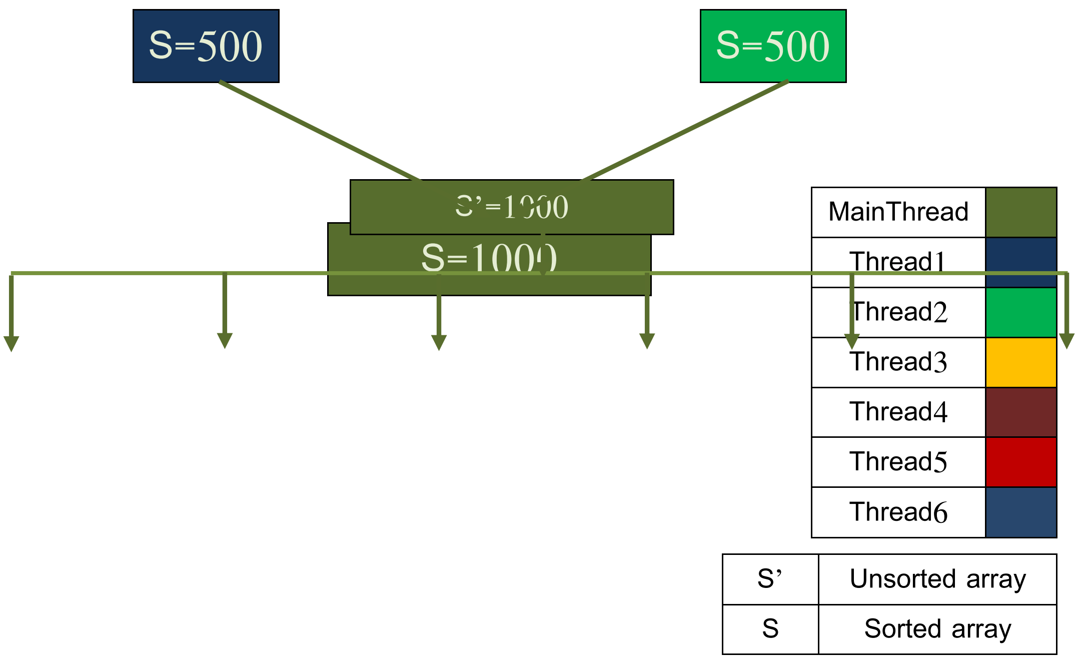


Figure 12: Parallel merge sort with 8 threads

The same model will be use to design Parallel DRS algorithm with change in finding Runs and creating Helper Array as shown in Figure 13.

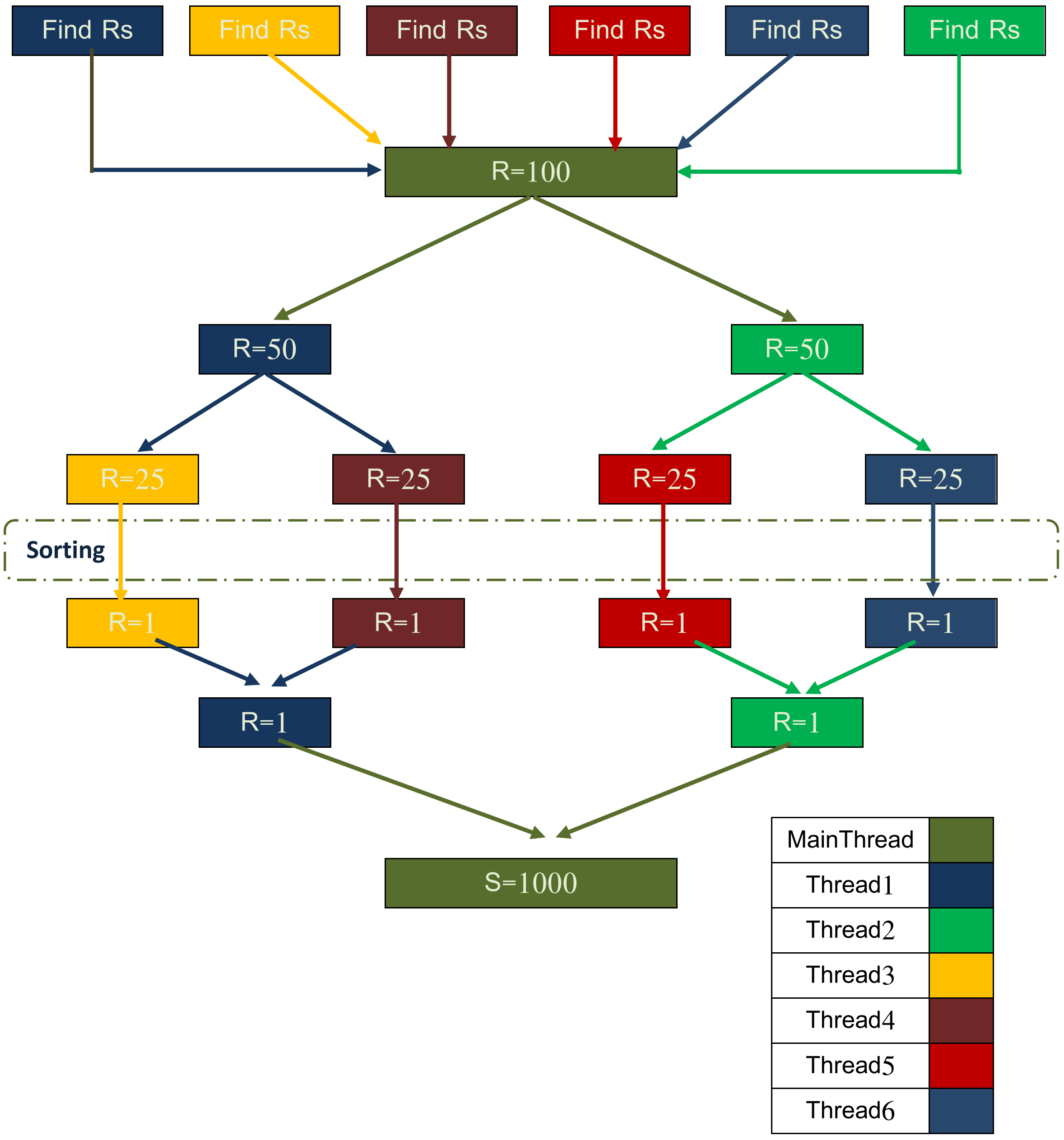


Figure 13: Parallel Divide-Runs sort with 8 threads

In this model, Main CPU uses to call available CPUs to find Runs Using parallel Processing. Then it merges founded Runs in one helper array. After find Runs and create Helper array, Root CPU divides the unsorted array into two sub-array than distributed them to two CPUs. Every CPU divides sub-array to two sub-array. This process continues until the number of unsorted sub-array reaches to desire number of parallel. Then, each CPU sorts one unsorted sub-array independently in parallel. Once, each CPU is done sorting their parts, the process of merging starts. Each parent CPU merges the two sorted subsections from its children threads. As the final step, the Root CPU merges two subsections from its children threads and produces the sorted array.

3.3. Analysis Algorithm

To ensure the execution time of an algorithm, three cases should be concerned.

3.3.1.Worse case

In DRS, the worst case appears when number of Runs reach Maximum number in array. In Array with 10 elements, number of Runs may be 1,2,3,4 ,or 5.No more Runs in array with 10 elements than 5.This happened when all Runs have only two elements.

When Run have 2 elements the helper array contains n/2 Runs

For that, the length of Helper array m=n/2.

Divide–Runs Algorithm contains two parts FindRuns which cost O(n) and recursion Divide–Runs.

Algorithm 3: Divide–Runs (A[] ,Aux[], RunsDivid[,],left , right)

```
If  left <  right
      middle = ( left + right ) / 2                                        — 1
      Divide–Runs ( A , aux, RunsDivid,  left  ,  middle  )                — T(m/2)
      Divide–Runs ( A , aux, RunsDivid,  middle +1 ,  right )              — T(m/2)
      Merge ( A  aux, RunsDivid [left, 0], RunsDivid [middle, 1],
          RunsDivid [right, 1])                                            — O(n)
End
```

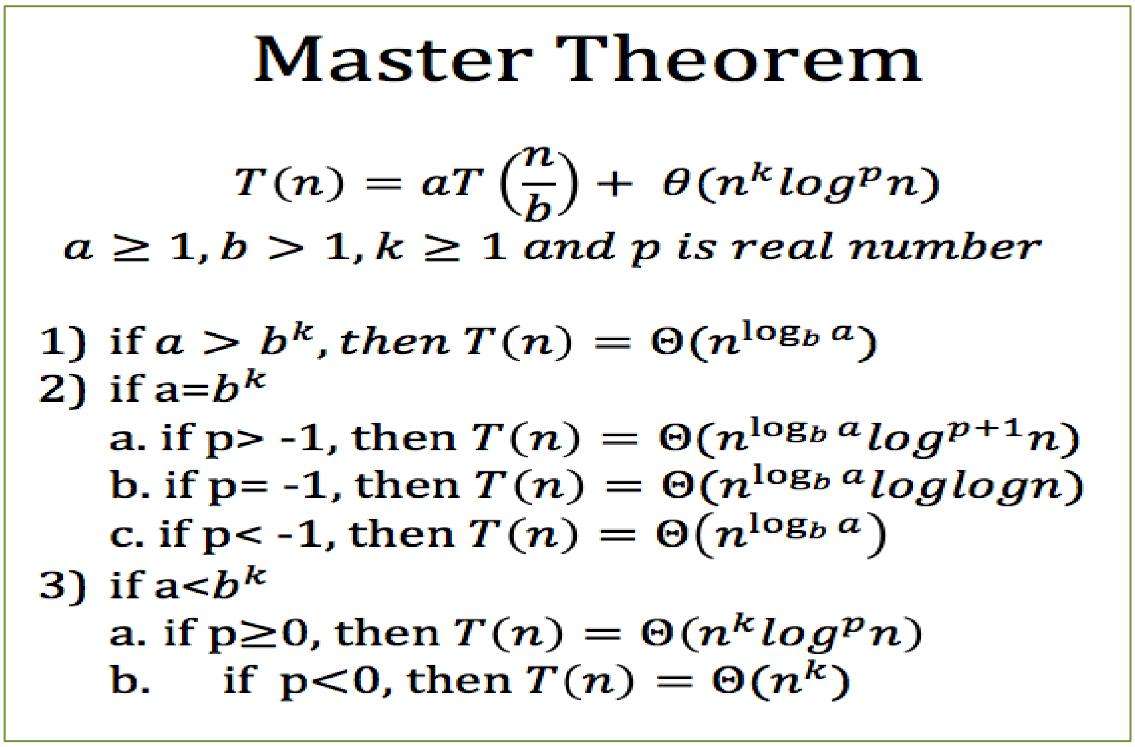
To find the general recursion form, we have to calculate cost for every line as follows:

$$T(n) = T(m/2) + T(m/2) + n$$
$$= 2T(m/2) + n$$
$$= 2T(n/4) + n$$

Rewrite merge sort recurrence as:

$$T(n) = \begin{cases} O(n) & if\ \ R = 1 \\ 2T\left(\frac{n}{4}\right) + cn & if\ \ R > 1 \end{cases}$$

By using the extended master theorem in case 3:



Master Theorem

$$T(n) = aT\left(\frac{n}{b}\right) + \theta(n^k log^p n)$$
$$a \geq 1, b > 1, k \geq 1 \ and \ p \ is \ real \ number$$

1) if $a > b^k, then \ T(n) = \Theta(n^{\log_b a})$
2) if a=b^k
   a. if p> -1, then $T(n) = \Theta(n^{\log_b a} log^{p+1} n)$
   b. if p= -1, then $T(n) = \Theta(n^{\log_b a} loglogn)$
   c. if p< -1, then $T(n) = \Theta\left(n^{\log_b a}\right)$
3) if a<b^k
   a. if p≥0, then $T(n) = \Theta(n^k log^p n)$
   b. if p<0, then $T(n) = \Theta(n^k)$

when a = 2; b = 4; k = 1; p = 0

we get:

T(n)=O(nlog n)

3.3.2. Best case

In DRS algorithm, if Runs is 1, it considers to be the best case because there is no need to use divide and merge functions. Two scenarios appear when number of Runs is 1. First scenario, if target array is already sorted decreasing .for example:

1	2	3	4	5	6	7	7	8	9

In this case, DRS algorithm looking for Runs which is one in n time which is the same number of target elements . Either the second scenario, if the Runs is 1 and it sorted Ascending.

9	8	7	7	6	5	4	3	2	1

In this case, Reverse Function is called which cost n/2 to reverse array.so that, total time complexity for find Runs and reverse array as following :

$O(n) = n + n/2 = 3n/2$

Which is $O(n) = n$.

3.3.3. Average case

In general, the average case running time is considered as bad as the worst case and analysis as same way as the worst case.

3.4. Sequential DRS Algorithm Implementation

Our code is implemented in c#. It contains the following variables:

I. MinRuns which determines the minimum Run size .Algorithm can be implemented without determine it ,However it enhance performance of algorithm. The chosen minimum size is 8 elements.

II. Divide which is a helper array. It is 2D array with 2 columns. Numbers of row in Divide-Runs determine by divide numbers of element by MinRuns.

Algorithm 4: FindRuns (A ,Aux , RunsDivid[,])

III. D
ivi
dC
ou
nt
wh
ich
hel
ps
to
det
er
mine Divide-Runs current row and length of Helper array.

IV. Axillary array which has the same size of main array to help to rearrange data in merge function.

The implementation also contains three main functions as following:

I. FindRuns function: This function token from Natural merge sort algorithms. However, it modified to store first index and last index of Run to Divide-Runs array. This function calls two other function to help in find and format Runs. If the Runs length less than MinRun, elements added to meet minimum length and Insertion sort uses to sort this Run. In addition, if Run sorted opposite of demand, Reverse function uses to reverse Run.

```
int acendOrder = 0;
int descOrder = 0;
int right = 0, mid = 0;
int DividCount = 0;
for (int i = 0; i <= numbers.Length - 1; i++)
   {
       if (i == numbers.Length - 1)
           {

               if (right < numbers.Length && descOrder == 1)
                 {

                     Array.Reverse(numbers, mid, right - mid + 1);
                     Divid[DividCount,0] = mid;
                     Divid[DividCount,1] = right;
                     DividCount++;
                  }
                else if (right < numbers.Length)
                  {
                     Divid[DividCount, 0] = mid;
                     Divid[DividCount, 1] = right;
                     DividCount++;
                  }
                  break;
            }
          else if (acendOrder == 0 && descOrder == 0)
            {

                 if (numbers[i] <= numbers[i + 1] && i + 1 >= numbers.Length -
1)
                  {
                    right++;
                    Divid[DividCount, 0] = mid;
                    Divid[DividCount, 1] = right;
                    DividCount++;
                    mid = right + 1;
                    right++;
                    acendOrder = 0;
                    descOrder = 0;
                  }
                else if (numbers[i] <= numbers[i + 1])
                  {
                    acendOrder = 1;
                    right++;
                  }
                 else if (numbers[i] >= numbers[i+1] && i+1 >=
numbers.Length-1)
                    {
                        right++;
                        Array.Reverse(numbers, mid, right - mid + 1);
                        Divid[DividCount, 0] = mid;
                        Divid[DividCount, 1] = right;
                        DividCount++;
                        mid = right + 1;
                        right++;
                        acendOrder = 0;
                        descOrder = 0;
                    }
                    else
                    {
                        descOrder = 1;
                        right++;
                    }
                }
                else if (acendOrder == 1)
                {
                  if (numbers[i] <= numbers[i + 1] && i + 1 >= numbers.Length
- 1)
                    {
                        right++;
                        Divid[DividCount, 0] = mid;
                        Divid[DividCount, 1] = right;
                        DividCount++;
                        mid = right + 1;
                        right++;
                        acendOrder = 0;
                        descOrder = 0;
                    }

                    else if (numbers[i] <= numbers[i + 1])
                        right++;
                    else
                    {
                        //**********
                        if (right - mid == 1)
                        {
                            int tmp = numbers[i];
                            numbers[i] = numbers[i + 1];
                            numbers[i + 1] = tmp;
                            if (numbers[i - 1] > numbers[i])
                            {
                                tmp = numbers[i - 1];
                                numbers[i - 1] = numbers[i];
                        numbers[i] = tmp;
                    }
                    right++;
                }

                //************
                else
                {
                    Divid[DividCount, 0] = mid;
                    Divid[DividCount, 1] = right;
                    DividCount++;
                    mid = right + 1;
                    right++;
                    acendOrder = 0;
                    descOrder = 0;
                }
            }
        }




        else
        {

         if (numbers[i] >= numbers[i + 1] && i + 1 >= numbers.Length
- 1)
            {
                right++;
                Array.Reverse(numbers, mid, right - mid + 1);
                Divid[DividCount, 0] = mid;
```

```
                Divid[DividCount, 1] = right;
                DividCount++;
                mid = right + 1;
                right++;
                acendOrder = 0;
                descOrder = 0;
            }
            else if (numbers[i] >= numbers[i + 1])
            {
                right++;

            }
            else
            {
                if (right - mid == 1)
                {
                    int tmp = numbers[i];
                    numbers[i] = numbers[i + 1];
                    numbers[i + 1] = tmp;
                    if (numbers[i - 1] < numbers[i])
                    {
                        tmp = numbers[i - 1];
                        numbers[i - 1] = numbers[i];
                        numbers[i] = tmp;
                    }
                    right++;
                }
                else
                {
                    Array.Reverse(numbers, mid, right - mid + 1);
                    Divid[DividCount, 0] = mid;
                    Divid[DividCount, 1] = right;
                    DividCount++;
                    mid = right + 1;
                    right++;
                    acendOrder = 0;
                    descOrder = 0;
                }
            }
        }


    }


    if (DividCount > 1)
        DividRuns(numbers, aux, Divid, 0, DividCount - 1);
```

II. Divide_Runs Algorithm: This function as same as classic merge sort algorithm which uses Divide and conquer technique .However, it divides helper array rather than main array .In our proposed algorithm we add one more parameter which represent Helper array and other parameters are the same. Moreover, all code as same as merge sort algorithm accept call Merge algorithm which calls data in helper array. Data in Helper array represent indexes in Main array .Our technique assumes to save indexes in array and divides elements according to Runs not like merge sort algorithm which divides array according to elements.

Algorithm 6: Merge (A[] ,B[])

III. M

erge algorithm: The merge algorithm plays a critical role in

t **Algorithm 5: Divide−Runs (A[] ,Aux[], RunsDivid[,],left , right)**

```
If  left <  right
h    middle = ( left + right ) / 2
     Divide−Runs ( A , aux, RunsDivid,  left  ,  middle  )
e    Divide−Runs ( A , aux, RunsDivid,  middle +1 ,  right )
     Merge ( A  aux, RunsDivid [left, 0], RunsDivid [middle, 1],
         RunsDivid [right, 1])
mEnd
```

erge sort algorithm as well as Divide−Runs sort algorithm. This function merges two sorted array into one in linear time and linear space. There is no change on Merge technique .The idea in calling Merge by index in Runs.

```
C []=new empty
  While A is not empty and B is not empty
      If head(A) ≤ head(B)
          append head(A) to C
          drop the head of A
      End
      Else
          append head(B) to C
          drop the head of B
      End
  End

    While A is not empty
      append head(A) to C
      drop the head of A
    End

    While B is not empty
      append head(B) to C
      drop the head of B
    End

   Return C[]
```

3.5.Parallel DRS Algorithm Implementation

Parallel Divide−Runs sort algorithm will implement using TPL in C#.TPL is based on the concept of a task, which represents an asynchronous operation. In some ways, a task resembles a thread or ThreadPool work item, but at a higher level of abstraction. The term task parallelism refers to one or more independent tasks running concurrently.

Parallel Divide-Runs algorithm have same variables as sequential Divide-Runs with adding some other variables which help in parallel processing. One of these variables is **_maxParallelDepth** . This variable is dynamically determined depth of parallel processing. The **Algorithm 7** returns number of maximum parallel depth which uses in specific PC.

Algorithm 7: DetermineMaxParallelDepth()

```
const int MaxTasksPerProcessor = 1;
 int maxTaskCount = Environment.ProcessorCount *
MaxTaskPerProcessor;
 int icheck = (int)Math.Log(maxTaskCount, 2);

Return (int) Math.Log(maxTaskCount, 2);
```

After determine maximum parallel depth, we are going to create as many threads as the number of processors (or cores). So we keep making the recursive method calls. It is very similar to the standard implementation except adding parallel depth. We use a parallel depth for every recursive method call until there is no more available processor cores. After that, we use the sequential algorithm. Algorithm 8 Shows how parallel Divide-Runs algorithm implemented.

Algorithm 8: Parallel_Divide-Runs(int[] array, int[] aux, int[,] Divid, int iStart, int iEnd, int recursionDepth)

```
         if (iStart >= iEnd)
         return;
  int middle = (iStart + iEnd) / 2;
  if (recursionDepth <= _maxParallelDepth)
      {
        Parallel.Invoke(
          () => DivideRuns (array, aux, Divid, iStart, middle, recursionDepth +
1),
          () =>  DivideRuns (array,aux, Divid, middle+1, iEnd,recursionDepth +
1)
            );
      }
     else
      {
          DivideRuns (array, aux, Divid, iStart, middle, recursionDepth + 1);
          DivideRuns (array, aux, Divid, middle + 1, iEnd, recursionDepth + 1);
       }
MergeTechnique.Merge(array , aux , Divid [ iStart, 0 ], Divid[ middle , 1 ],
                                             Divid[iEnd, 1]);
```

3.5.1. Advantages and disadvantages

Advantages of DRS

- DRS algorithm has better performance than other algorithms that use natural sorted technique because it does not use stack .In addition it takes advantage of natural sorted sub array which not need to operate more sort on array. It takes only O(n) when array sorted.
- Reversed array is a worst case in most sort algorithms however in DRS reversed array takes only O(n) to sort.
- It takes O(nlogn) in average and worse case and his Performance is better compared to other algorithms.
- The divide-and-conquer nature of DRS algorithm makes it very convenient for parallel processing. By using Divide technique, it can act to any type of parallel architecture or distributed system.
- Merge sort usually uses in external sort because of merge technique which can apply on separated elements. Like Merge sort, DRS algorithm can apply perfectly on external sort because it takes divide and merge techniques from merge sort.
- Divide-Runs is a stable, comparison, recursive, and adaptive sort algorithm.

3.5.2. Disadvantages of DRS

As in most Merge algorithms , DRS algorithm need more space to apply Merge technique .It takes O(n) complexity in space whereas, most algorithms take only O(1) space complexity.

4. Experiment and Result Analysis

This chapter describes performance metrics and the evaluation of experiment results. The goal of the evaluation is to show the efficiency of the proposed algorithm compared to those available in the literature. Our focus was on the measuring of the algorithm in term of Time complexity.

4.1. Experiment Setting

Benchmark algorithms and proposed algorithm was implemented in C# MS Visual studio 2017 using Console App (.Net Framework 4.6) and The Task Parallel Library (TPL) for parallel processing. The machine used for performance evaluation is Lenovo laptop with 256MB of SSD disk storage. It has an Intel(R) Core i7-3632QM CPU which works at 2.20GHz. The CPU has four cores that can deliver 8 threads via Intel hyper-Threading Technology.The Chip on mainboard has one DDR3 memory controllers which provides 800MHz memory clock frequency. The capacity of main memory is 8GB.

4.2. Performance Metrics

The main measured metric to evaluate the proposed algorithm is Execution time (ET).This metric was used in literature to evaluate the efficiency of algorithms, such as [15] [26] [6] [8].

Integer numbers have been generated randomly and used for the experiments. Statistical analysis for 100 samples generated randomly was conducted for each dimension of input, starting from the input set of 1 million elements and increasing the size of the task 100,000 elements for each test to 2 million elements. The number of samples was chosen as 100 since it is a standard statistical number to examine proposed methods in benchmark tests [58]. Average of execution for 100 samples calculate by equation:

$$\text{Average of ET} = \frac{Sum\ of\ execution\ time}{number\ of\ execution\ time}$$

The experiment is conducted in sequential with three scenarios, which are

(1) When elements are Sorted (one Run),

(2) When elements are random and

(3) Elements with 10,100 and 1000 Runs.

These Scenarios will compare with different dataset size. For evaluating parallel Divide-Runs algorithm, it will be compared with Merge sort in different scenarios and different dataset size too.

4.3.Result Evaluation

4.3.1 Sequential experiment

Proposed algorithm is compared sequentially with Timsort and α-stack algorithms in different situation as shown in table 3:

K	TimSort	α-Stack	DRSA
1	7.24	7.32	7.34
10	122.46	122.53	108.91
100	232.62	232.01	198.16
1000	347.52	346.58	289.74
Random	839.95	837.16	644.00

Table 3: COMPARING BETWEEN TIMSORT, Α-STACK, AND DRS ALGORITHMS

i. When all elements are random Figure 14 shows the enhancement and the preference of proposed algorithm. The results shows decreasing in the execution time with 29.99% comparing with α-stack and 30.43% comparing with Timsort.

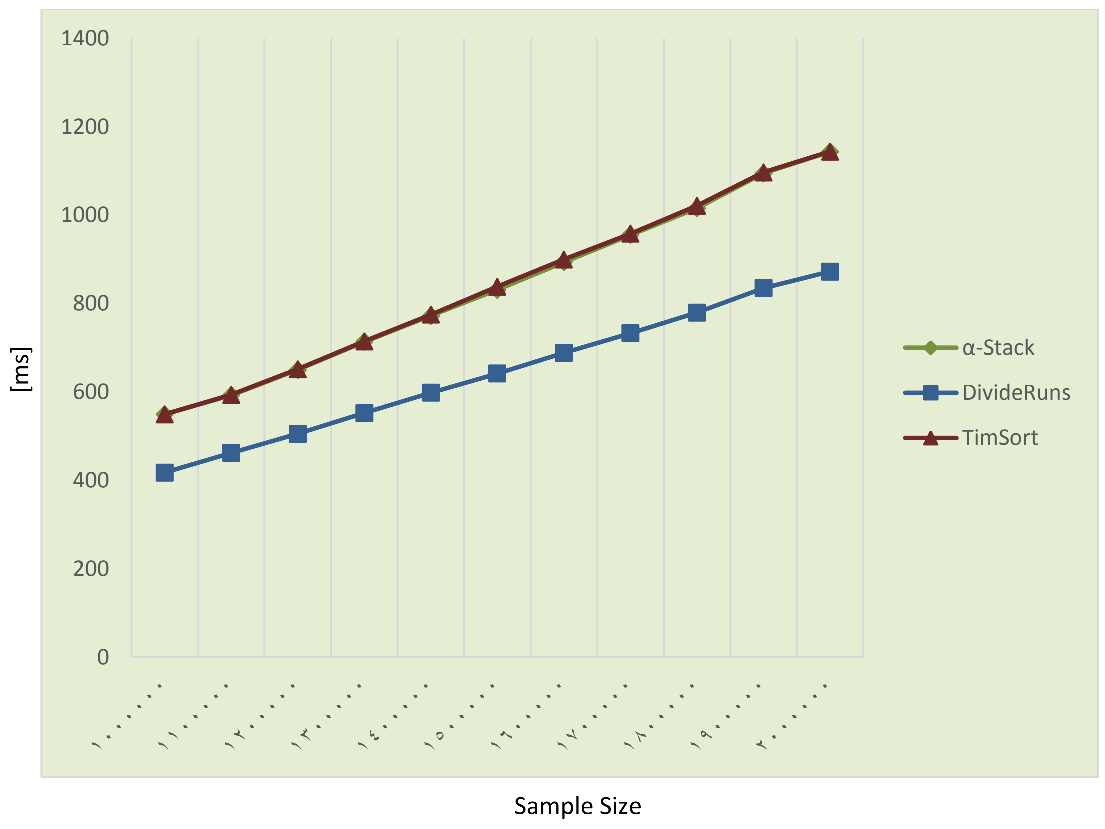


Figure 14: Sorting time comparisons (Random Elements)

ii. When elements are sorted .Figure 15 shows convergence between proposed algorithm and benchmark algorithms because all of them use the same technique to find Runs and all of them have one Run when elements are sorted.

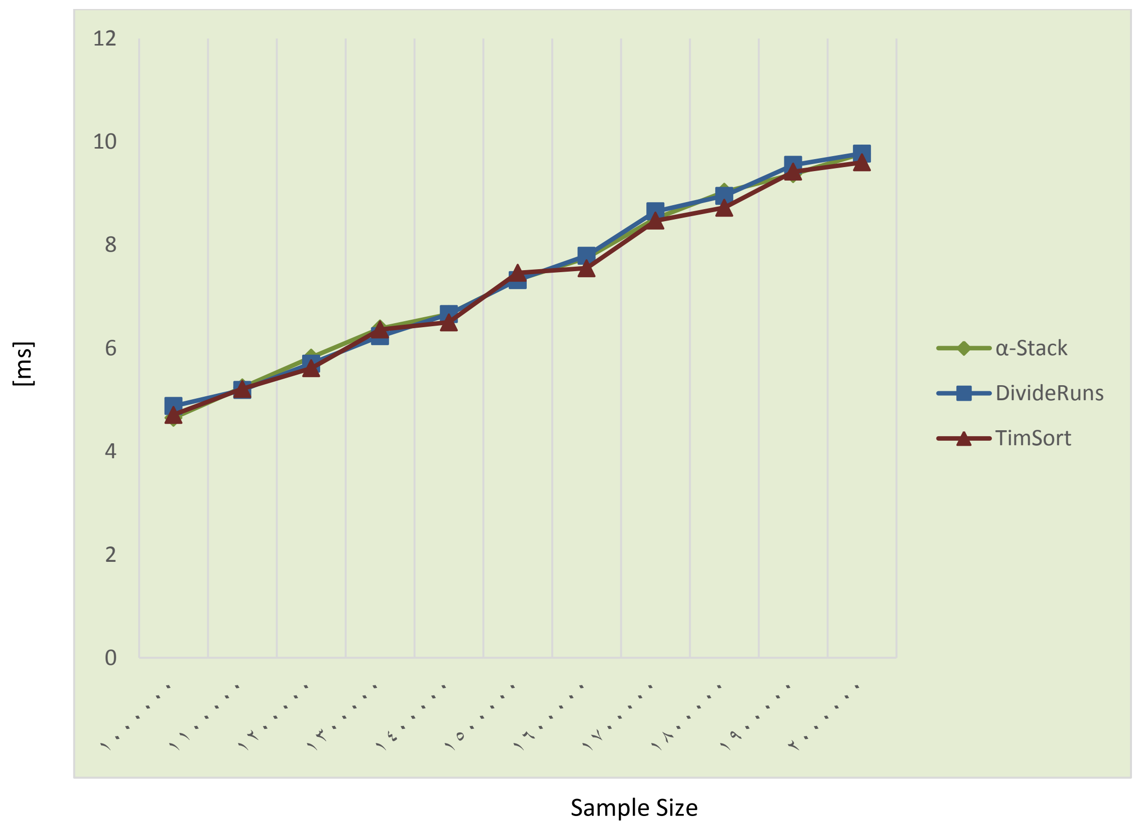


Figure 15: Sorting time comparisons (Sorted Elements)

iii. When dataset used sequences of 10,100,1000 Runs, Table 3 shows the enhancement of DRS algorithm according to numbers of Runs comparing with Benchmark algorithms.

Also, Figure 16,17, and 18 show the preference of proposed algorithm

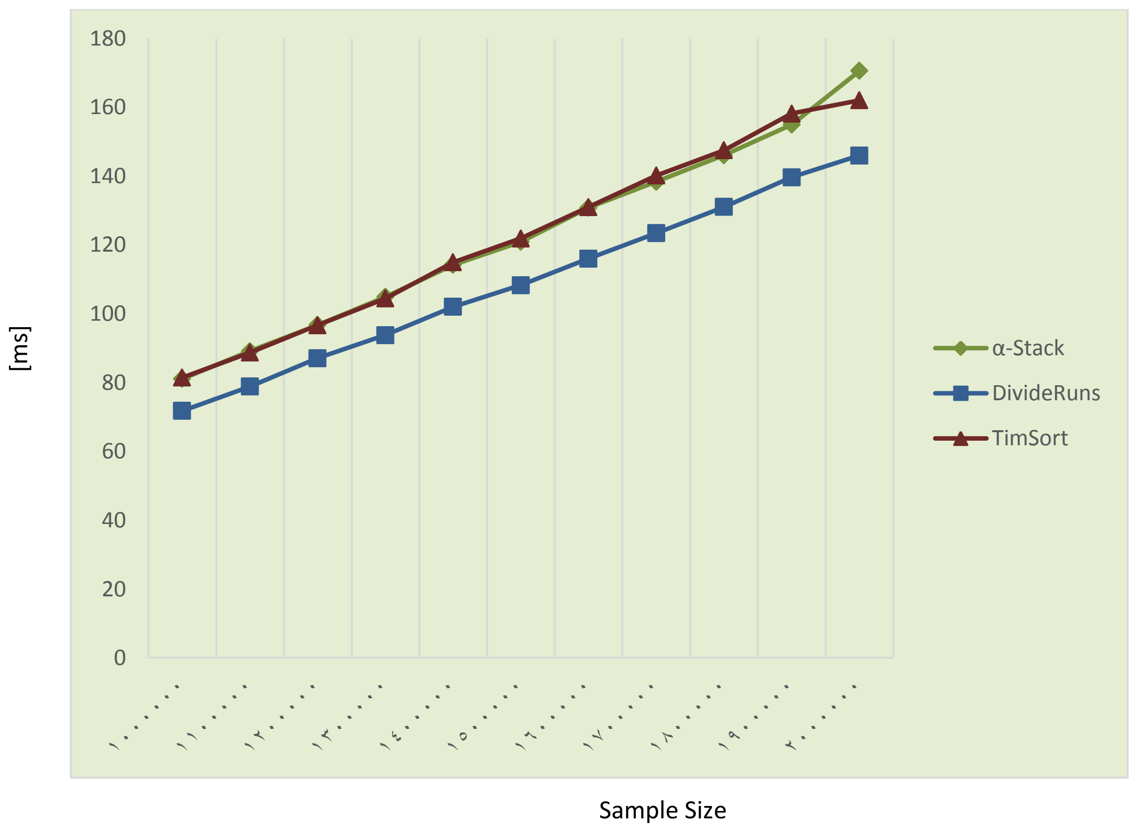


Figure 16: Sorting time comparisons (10 Runs)

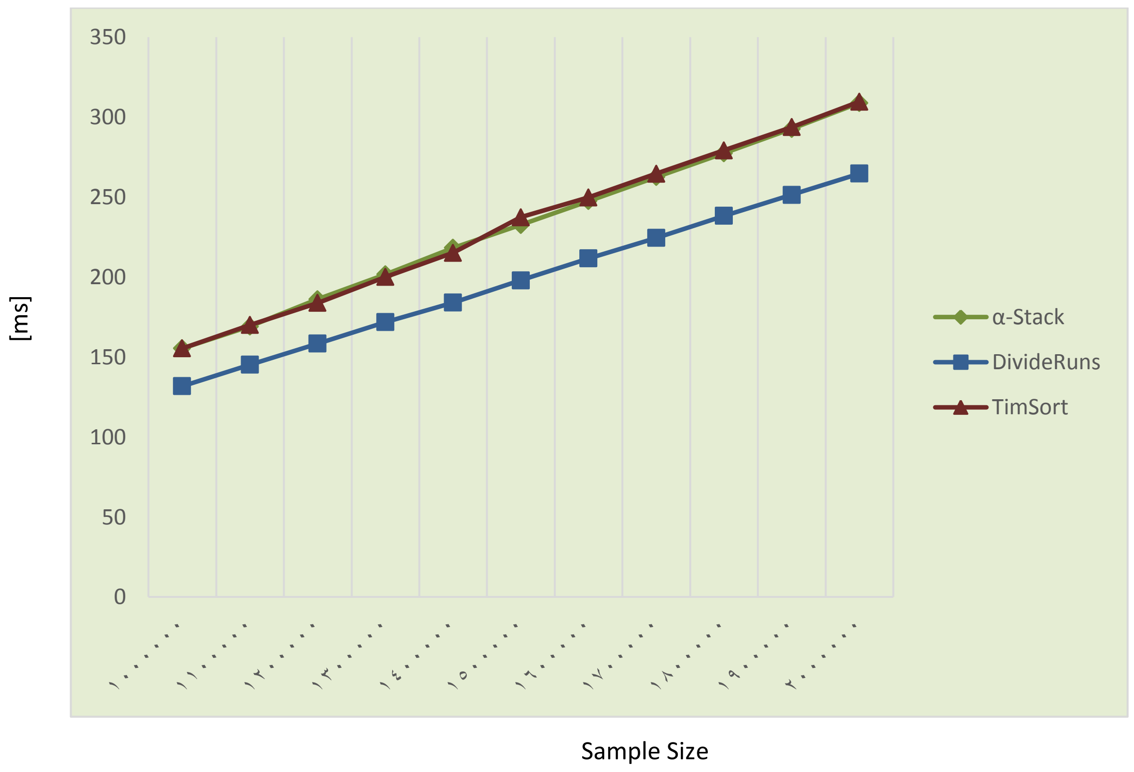


Figure 17: Sorting time comparisons (100 Runs)

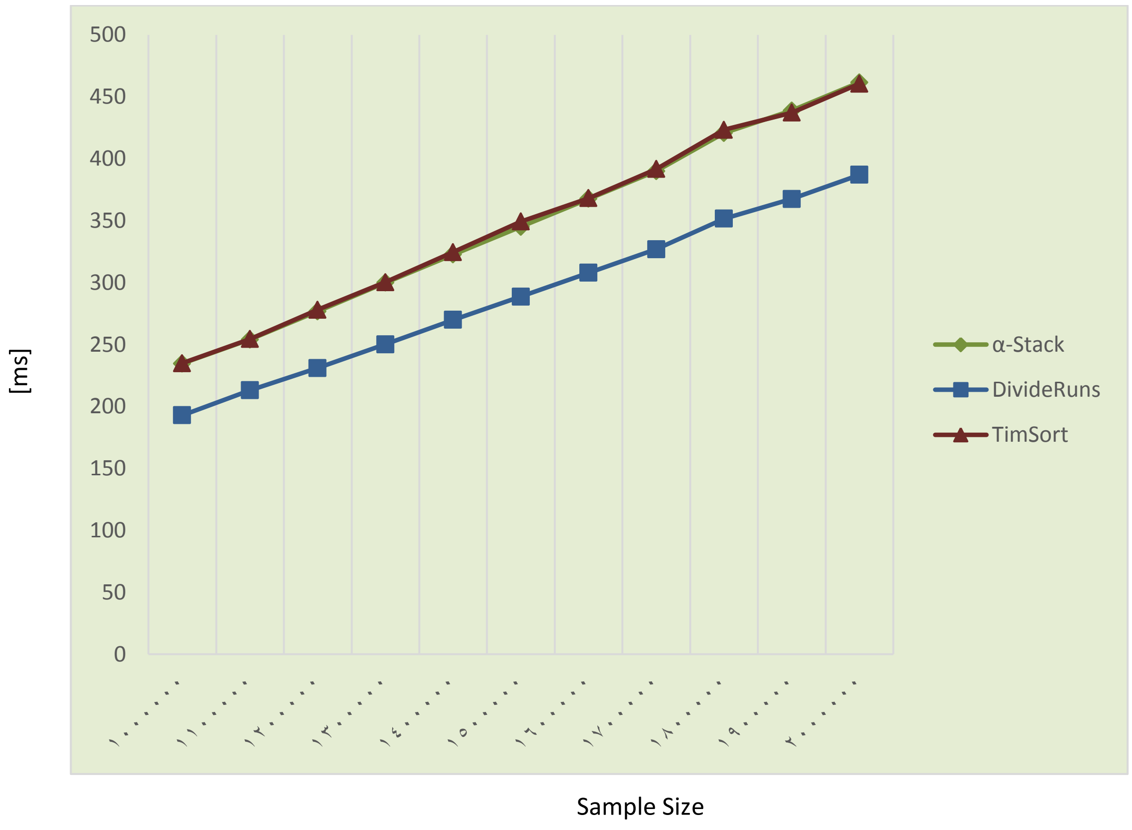


Figure 18: Sorting time comparisons (1000 Runs)

4.3.2 Parallel Experiment

Proposed algorithm is compared parallel with Merge Sort when inputs are sorted and random as shown in table 4.

	P-DRS	**P-MergeSort**
Sorted	20.70	140.14
Random	173.38	241.18

Table 4: Compare between Merge sort and DRS

1. When all elements are random Figure 19 shows the enhancement and the preference of proposed algorithm. The results shows decreasing in the execution time with 39.1 % comparing with merge sort. In addition, we notice that when Runs is less , the decreasing of execution time became better.

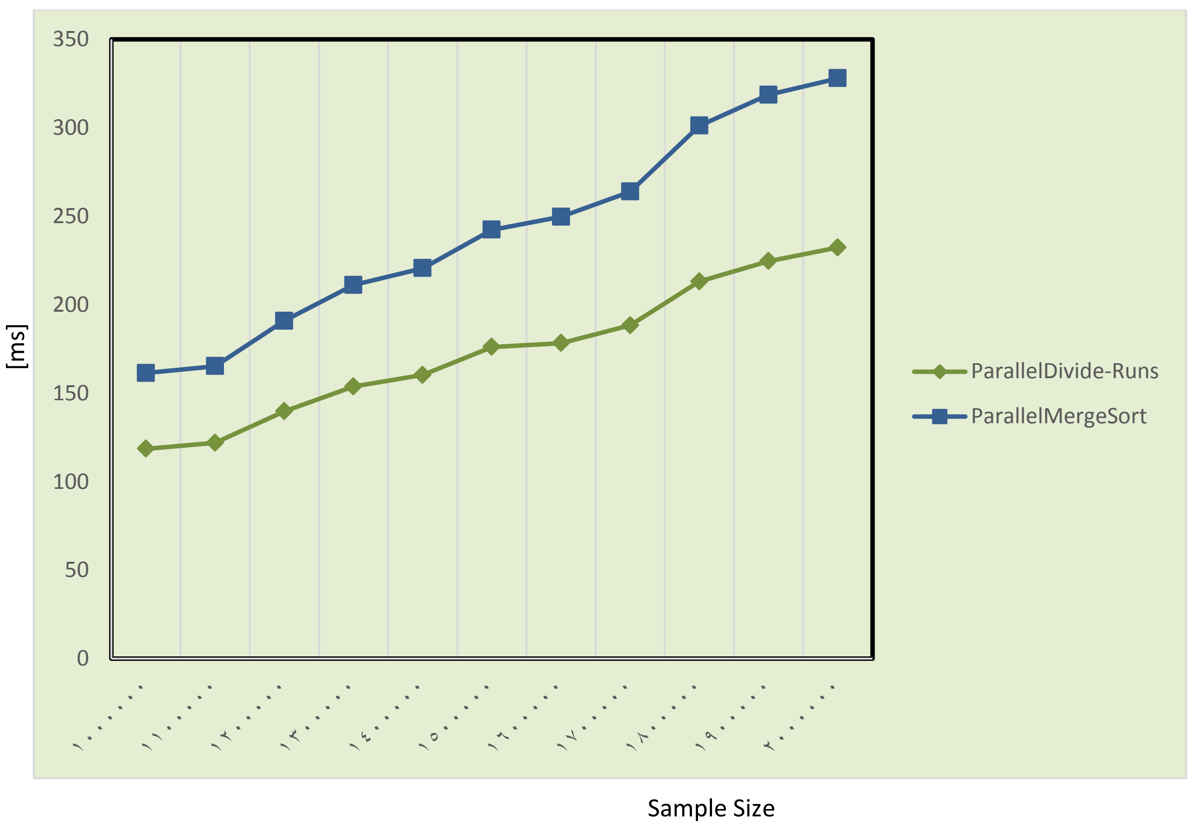


Figure 19: Parallel Sorting time comparisons (Random Elements)

2. When elements are sorted .Figure 20 shows the proposed algorithm is much better than merge sort because proposed algorithm is adaptive algorithm and it takes only O(n) time whereas merge sort takes O(nlogn).

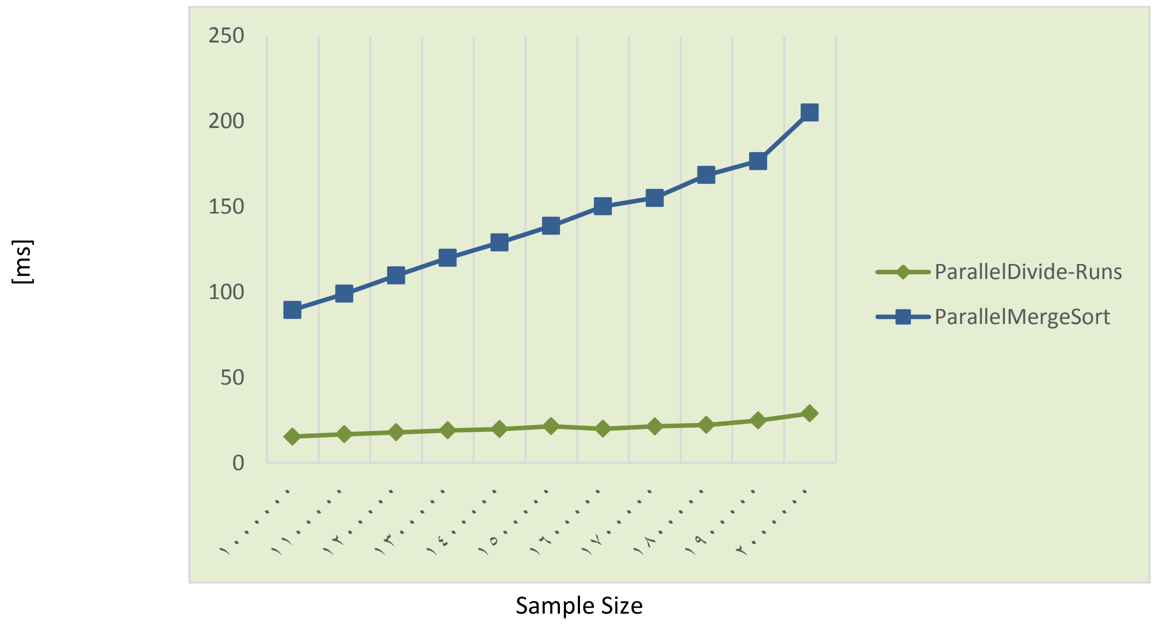


Figure 20: Parallel sorting time comparisons (Sorted Elements)

4.3.3 Additional Experiments

For another way of evaluation, testing were used 100 samples generated at random for the task size from 100 to 10,000 000 elements, increasing the size of sorted array ten times in the following experiments.

Average of ET				
# of elements	α−Stack	DivideRuns	TimSort	Grand Total
10	0	0	0	0
100	0	0	0	0
1000	0	0	0	0
10000	2.05	2.02	2.05	2.04
100000	33.78	25.15	33.33	30.75333333
1000000	396.32	297.71	393.66	362.5633333
10000000	4443.04	3422.53	4456.76	4107.443333
Grand Total	**696.455714**	**535.344285**	**697.971428**	**643.257142**

Table 5: Sequential Sorting time comparisons (Random 10M Elements)

Average of ET				
# of elements	α−Stack	DivideRuns	TimSort	Grand Total
10	0	0	0	0
100	0	0	0	0
1000	0	0	0	0
10000	0	0	0	0
100000	0	1	0	0.333333333
1000000	3.03	10.42	3.04	5.496666667
10000000	36.75	108.41	36.8	60.65333333
Grand Total	**5.682857143**	**17.1185714**	**5.69142857**	**9.49761904**

Table 6: Sequential Sorting time comparisons (Sorted 10M Elements)

Average of ET				
# of elements	α−Stack	DivideRuns	TimSort	Grand Total
10	0.01	0.01	0.01	0.01
100	0.04	0	0.04	0.026666667
1000	0	0	0	0
10000	0	0	0.01	0.003333333
100000	5.95	5.1	5.26	5.436666667
1000000	61.79	57.89	59.29	59.65666667
10000000	632.38	598.07	592.95	607.8
Grand Total	**100.0242857**	**94.43857143**	**93.93714286**	**96.13333333**

Table 7: Sequential Sorting time comparisons (10Runs with 10M Elements)

Average of ET				
# of elements	α−Stack	DivideRuns	TimSort	Grand Total
10	0	0	0	0
100	0	0	0	0
1000	0	0	0	0
10000	1.01	0.84	1.02	0.956666667
100000	11.01	9.26	11.05	10.44
1000000	113.44	99.49	111.05	107.9933333
10000000	1121.06	1014.89	1134.95	1090.3
Grand Total	**178.0742857**	**160.64**	**179.7242857**	**172.8128571**

Table 8: Sequential Sorting time comparisons (100Runs with 10M Elements)

Average of ET				
# of elements	α−Stack	DivideRuns	TimSort	Grand Total
10	0	0	0	0
100	0	0	0	0
1000	0	0.02	0.02	0.013333333
10000	2.02	1.01	2.01	1.68
100000	20.47	13.97	19.72	18.05333333
1000000	170.79	141.18	167.51	159.8266667
10000000	1646.94	1431.5	1660.71	1579.716667
Grand Total	**262.8885714**	**226.8114286**	**264.2814286**	**251.3271429**

Table 9: Sequential Sorting time comparisons (1000Runs with 10M Elements)

Average of ET			
# of elements	ParallelDivide−Runs	ParallelMergeSort	Grand Total
10	0.07	0.05	0.06
100	0.02	0.01	0.015
1000	0.02	0.03	0.025
10000	1.05	1.2	1.125
100000	9.66	13.32	11.49
1000000	118.82	163.79	141.305
10000000	1190.58	1656.82	1423.7
Grand Total	**188.6028571**	**262.1742857**	**225.3885714**

Table 10: Parallel Sorting time comparisons (Random 10M Elements)

Average of ET			
# of elements	ParallelDivide-Runs	ParallelMergeSort	Grand Total
10	0.67	0.14	0.405
100	0.06	0	0.03
1000	0.01	0.03	0.02
10000	0	0.53	0.265
100000	2.15	9.61	5.88
1000000	15.18	91.36	53.27
10000000	137.51	1010.58	574.045
Grand Total	**22.22571429**	**158.8928571**	**90.55928571**

Table 11: Parallel Sorting time comparisons (Sorted 10M Elements)

The following graphs show statistical analysis for 100 samples generated randomly was conducted for each dimension of input, starting from the input set of 1million elements and increasing the size of the task 1,000,000 elements for each test to 10 million elements.

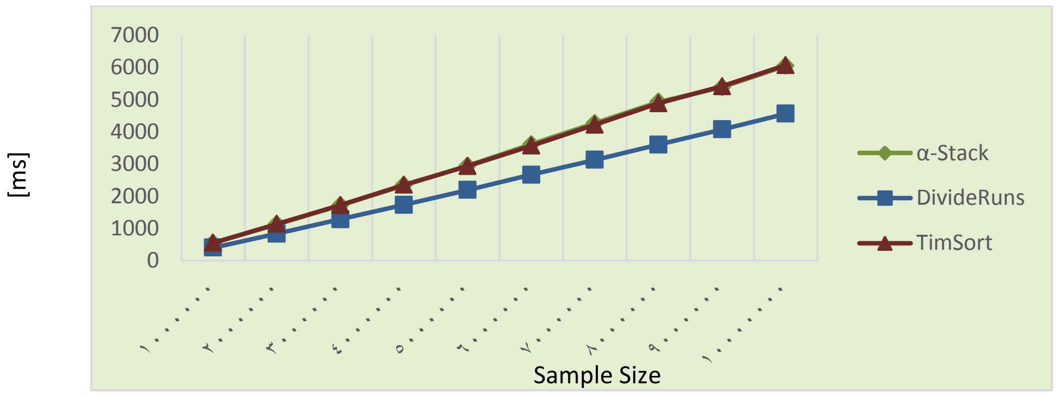


Figure 21: Sequential Sorting time comparisons (Random 1M to 10M Elements)

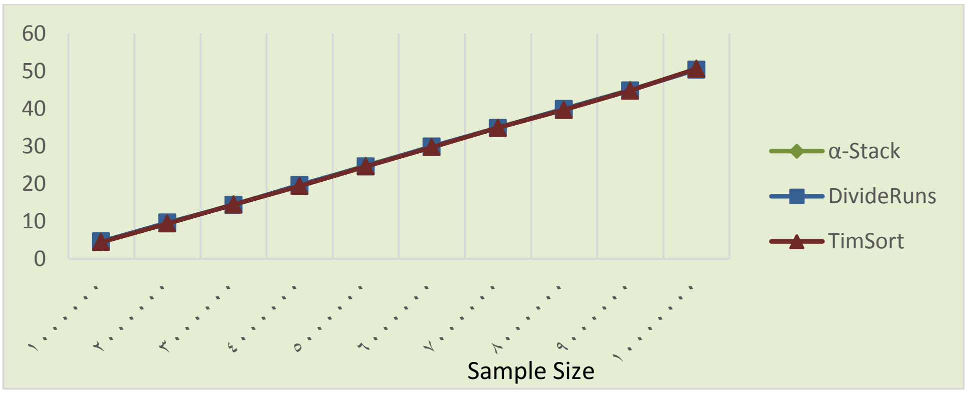


Figure 22: Sequential Sorting time comparisons (Sorted 1M to 10M Elements)

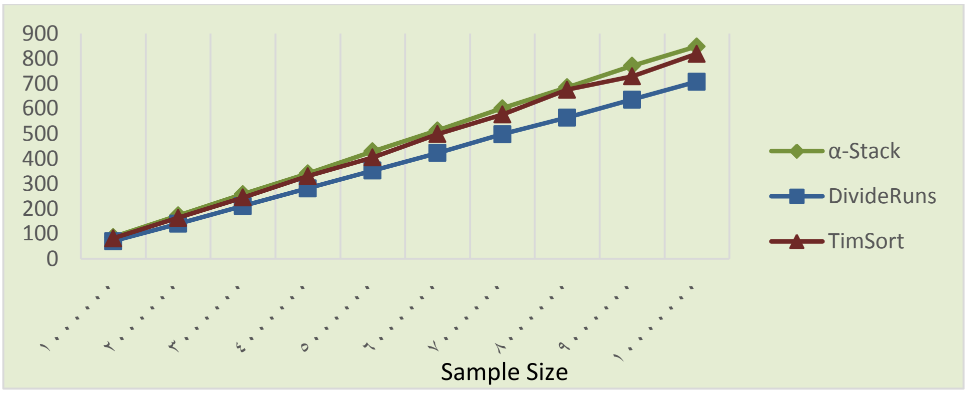


Figure 23: Sequential Sorting time comparisons (10Runs 1M to 10M Elements)

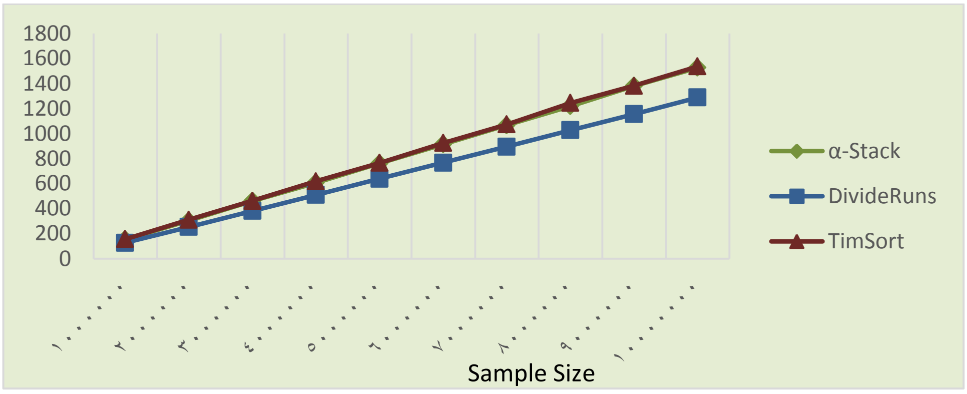


Figure 24: Sequential Sorting time comparisons (100Runs 1M to 10M Elements)

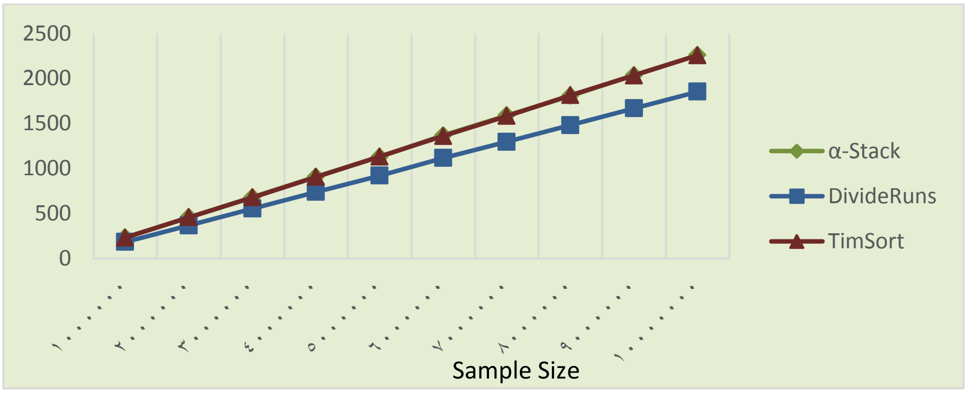


Figure 25: Sequential Sorting time comparisons (1000Runs 1M to 10M Elements)

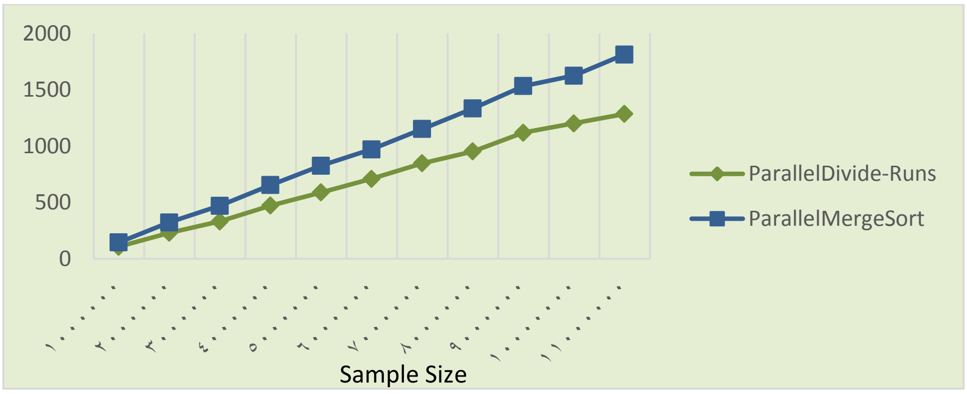


Figure 26: Parallel Sorting time comparisons (Random 1M to 10M Elements)

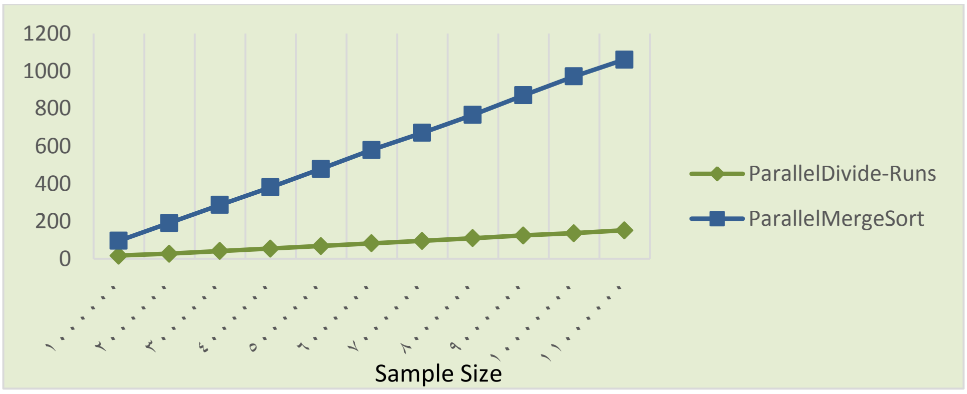


Figure 27: Parallel Sorting time comparisons (Sorted 1M to 10M Elements)

The following graphs show statistical analysis for 50 samples generated randomly was conducted for each dimension of input, starting from the input set of 1million elements and increasing the size of the task 100,000 elements for each test to 2 million elements.

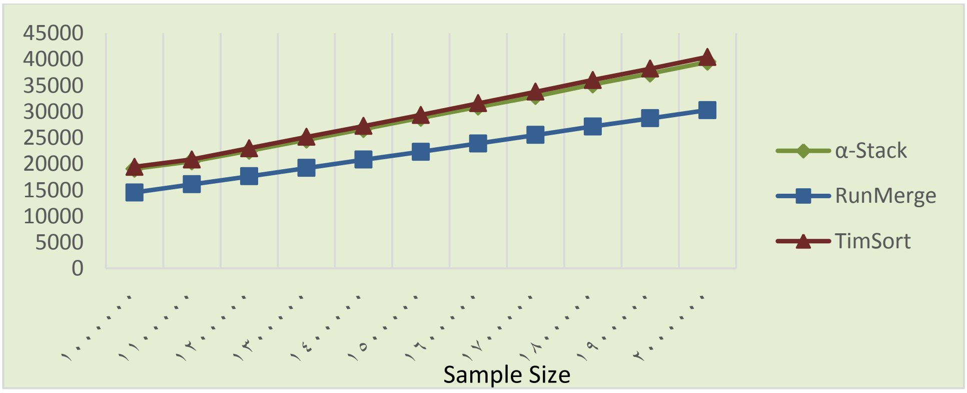


Figure 28: Sequential Sorting time comparisons (Random Elements 50 Samples)

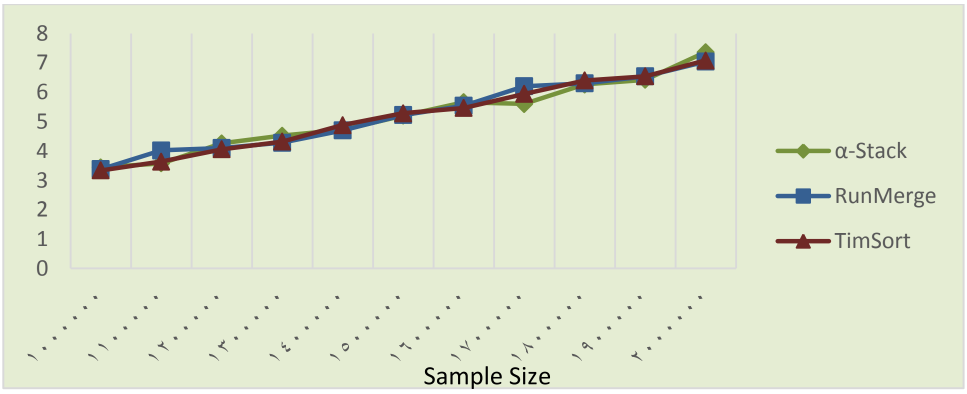


Figure 29: Sequential Sorting time comparisons (Sorted Elements 50 Samples)

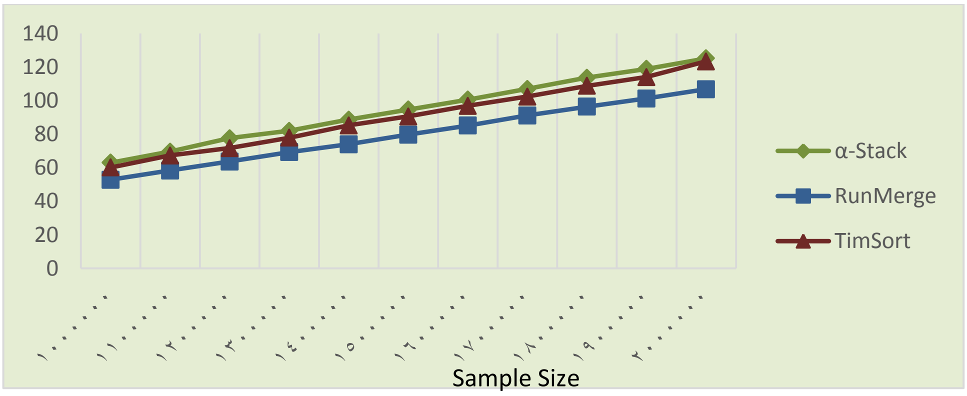


Figure 30: Sequential Sorting time comparisons (10Runs Elements 50 Samples)

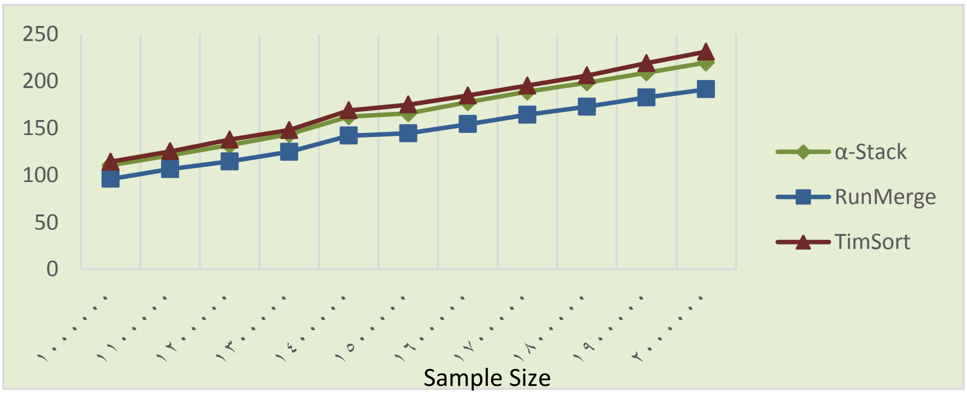


Figure 31: Sequential Sorting time comparisons (100Runs Elements 50 Samples)

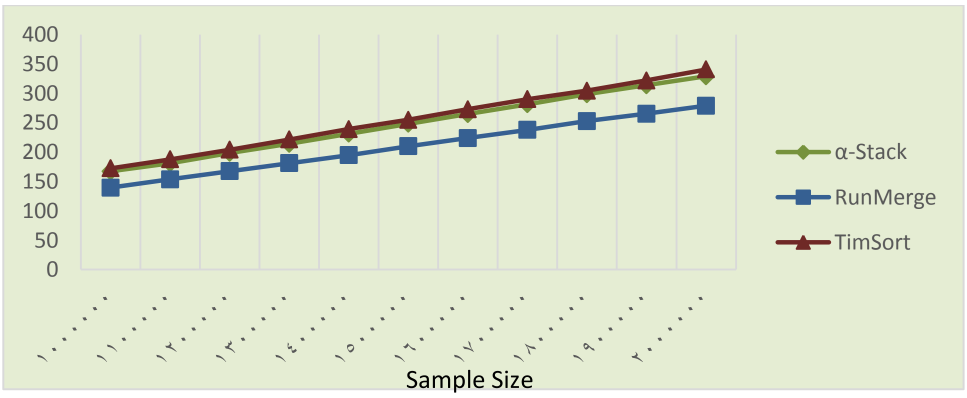


Figure 32: Sequential Sorting time comparisons (1000Runs Elements 50 Samples)

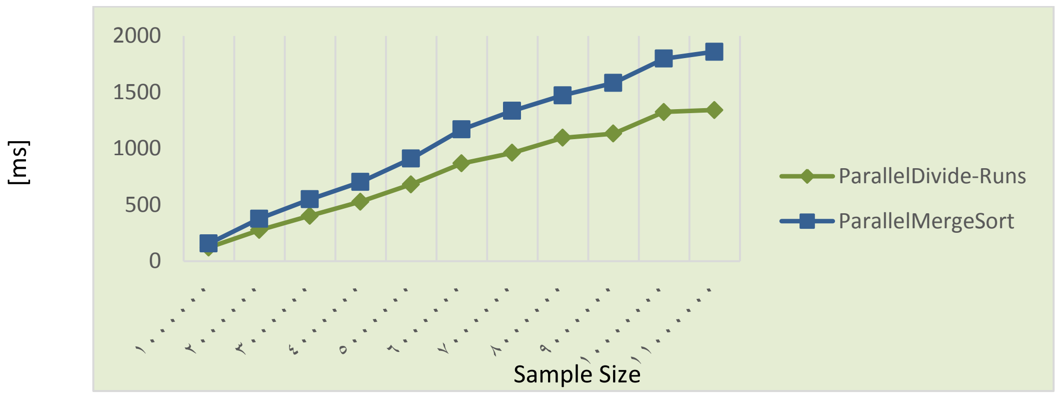


Figure 33: Parallel Sorting time comparisons (Random Elements 50 Samples)

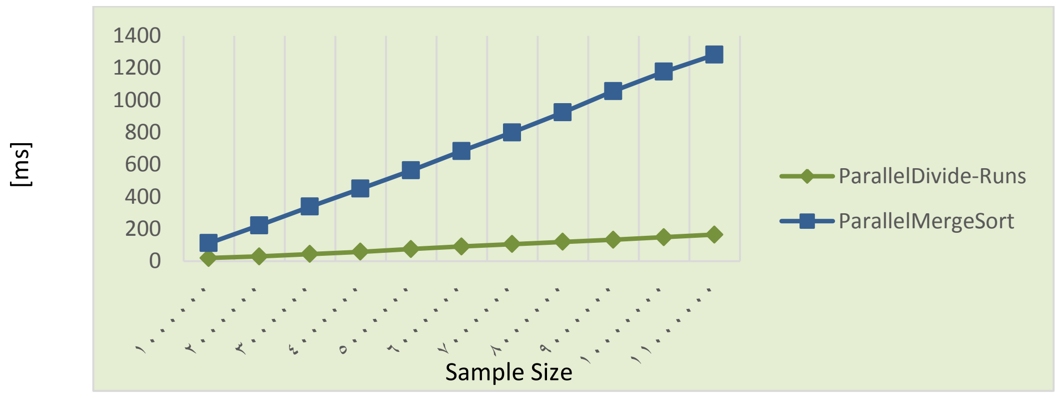


Figure 34: Parallel Sorting time comparisons (Sorted Elements 50 Samples)

5. Conclusion and Future work

5.1. Conclusion

In this study, a new natural merge sort algorithm called Divide-Runs Sort algorithm (DRS) proposed. DRS algorithm is a comparison, stable, and adaptive algorithm which works recursively. DRS takes benefits from natural sub-ordered list call Runs as well as Divide and Conquer technique. DRS solves the best-case problem in merge sort ,and it solves parallel processing in natural merge sort algorithms.

The results evaluation indicates that DRS is an efficient algorithm and decreasing time complexity in execution time to 30% comparing with benchmarks in sequential processing and decreasing execution time to 39% in parallel processing. Thus, the proposed algorithm is more efficient than the previous merge sort algorithms because it takes advantage of two parts of Merge (original merge sorting and Natural Merge sorting)

5.2. Future work

For future work, we are planning to implement DRS parallel sort algorithm in GPU and Distributed system. The implementation will compare with [26] and it will take the same model .Nowadays, Merge sort is usually used in external sort .we are planning to implement DRS on external sort model that shows in [27].We expect DRS algorithm will be more efficient in parallel, distributed and external sorting

References

[1] F. Zheng, M. Yin, X. Xu and M. Xu, "Optimized Merge Sort on Modern Commodity Multi-core CPUs," *TELKOMNIKA Telecommunication, Computing, Electronics and Control,* Vols. Vol 14, No 1: March 2016, 2016.

[2] H. D. M. V. de Wiel, "Sort Performance Improvements in Oracle Database 10g Release2," 2005, An Oracle White Paper.

[3] A. Djajadi, F. Laoda, R. Rusyadi, T. Prajogo and r. Sinaga, "A Model Vision of Sorting System Application Using Robotic Manipulator," *TELKOMNIKA Telecommunication, Computing, Electronics and Control,* Vols. Vol 8, No 2: 2010, 2010.

[4] S. Buss and A. Knop, "Strategies for stable merge sorting," in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 2019, pp. 1272--1290.

[5] G. a. A. N. Kocher, "Analysis and Review of Sorting Algorithms," *International Journal of Scientific Engineering and Research (IJSER),* vol. 2, no. 3, pp. 81--84, 2014.

[6] A. Uyar, "Parallel merge sort with double merging," in *2014 IEEE 8th International Conference on Application of Information and Communication Technologies (AICT)*, IEEE, 2014, pp. 1--5.

[7] D. a. A. A. Pasetto, "A comparative study of parallel sort algorithms," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ACM, 2011, pp. 203--204.

[8] Auger, Nicolas, Nicaud, C. a. Pivoteau and Carine, "Merge Strategies: from Merge Sort to TimSort," *HAL,* 2015.

[9] T. Peters, "Timsort," 2002. [Online]. Available: http://svn.python.org/projects/python/trunk/Objects/listsort.txt.

[10] R. Rashidy, S. Yousefpour and M. Koohi, "Parallel bubble sort using stream program," *International Conference on Application of Information and Communication Technologies (AICT),* pp. 1-5, 2011.

[11] X. X. M. Y. F. Z. Ming Xu, "Optimized Merge Sort on Modern Commodity Multi-core CPUs," *TELKOMNIKA (Telecommunication Computing Electronics and Control),* vol. 14, 2016.

[12] D. E. Knuth, The art of computer programming: sorting and searching, Pearson Education, 1998.

[13] Z. Yildiz, M. Aydin and G. Yilmaz, "Parallelization of bitonic sort and radix sort algorithms on many core GPUs," 2013.

[14] S. Y. Wang, "A new sort algorithm: self-indexed sort," *ACM SIGPLAN Notices,* vol. 31, pp. 28--36, 1996.

[15] A. K. Karunanithi, "A Survey , Discussion and Comparison of Sorting Algorithms," Department of Computing Science, Umea University, 2014.

[16] Z. A. Abbas, "Comparison Study of Sorting Techniques in Dynamic Data Structure," Universiti Tun Hussein Onn Malaysia, 2016.

[17] V. Joshi, "https://medium.com," 8 5 2017. [Online]. Available: https://medium.com/basecs/sorting-out-the-basics-behind-sorting-algorithms-b0a032873add.

[18] "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Adaptive_sort.

[19] "EECS Instructional and Electronics Support," University of California,Berkeley, [Online]. Available: https://inst.eecs.berkeley.edu/~cs61bl/r//cur/sorting/algorithm-comparisons.html?topic=lab26.topic&step=5&course=.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, Third Edition ed., 3th, Ed., The MIT Press, 2009.

[21] R. L. Kruse and A. J. Ryba, Data Structures and Program Design in C++, Prentice-Hall, Inc., 1999.

[22] M. A. Weiss, Data Structures and Algorithm Analysis in C++, 2nd ed., Addison-Wesley Longman Publishing Co., Inc., 1998.

[23] K. a. B. A. Thabit, "A Novel Approach of Selection Sort Algorithm with Parallel Computing and Dynamic Programing Concepts," *Computing and Information Technology Sciences,* p. 27.

[24] C. Bron, "Merge Sort Algorithm," *Commun. ACM,* vol. 15, no. May 1972, pp. 357--358, 1972.

[25] T. H. a. L. C. E. a. R. R. L. a. S. C. Cormen, Introduction to algorithms, 3th, Ed., MIT press, 2009.

[26] H. Shamoto, K. Shirahata, A. Drozd, H. Sato and S. Matsuoka, "GPU-Accelerated Large-Scale Distributed Sorting Coping with Device Memory Capacity," *IEEE Transactions on Big Data,* vol. 2, pp. 57-69, 2016.

[27] J. Lee, H. Roh and S. Park, "External Mergesort for Flash-Based Solid State Drives," *IEEE Transactions on Computers,* vol. 65, pp. 1518-1527, 2016.

[28] M. Kronrod, "Optimal ordering algorithm without operational field," *DOKLADY AKADEMII NAUK SSSR,* vol. 186, p. 1256, 1969.

[29] L. T. Pardo, "Stable sorting and merging with optimal space and time bounds," *SIAM Journal on Computing,* vol. 6, pp. 351--372, 1977.

[30] J. Salowe and W. Steiger, "Simplified stable merging tasks," *Journal of Algorithms,* vol. 8, pp. 557--571, 1987.

[31] T. Pasanen, "Lajittelu minimitilassa," M. Sc. Thesis T-93-3, Department of Computer Science, University of Turku, Turku, 1993.

[32] B.-C. Huang and M. A. Langston, "Practical in-place merging," *Communications of the ACM,* vol. 31, pp. 348--353, 1988.

[33] B.-C. Huang and M. A. Langston, "Fast stable merging and sorting in constant extra space," *The Computer Journal,* vol. 35, pp. 643--650, 1992.

[34] O. Shivers, "A simple and efficient natural merge sort," Georgia Institute of Technology, 2002.

[35] V. Jugé, "Adaptive Shivers Sort: An Alternative Sorting Algorithm," *arXiv preprint arXiv:1809.08411,* 2018.

[36] T. Takaoka, "Partial Solution and Entropy," in *Mathematical Foundations of Computer Science 2009*, Berlin, Heidelberg, Springer Berlin Heidelberg, 2009, pp. 700--711.

[37] J. I. Munro and S. Wild, "Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs," *arXiv preprint arXiv:1805.04154,* 2018.

[38] M. a. C. D. La Rocca, "NeatSort-A practical adaptive algorithm," *arXiv preprint arXiv:1407.6183,* 2014.

[39] B. Chandramouli and J. Goldstein, "Patience is a virtue: Revisiting merge and sort on modern processors," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, ACM, 2014, pp. 731--742.

[40] S. S. Skiena, "Encroaching lists as a measure of presortedness," *BIT Numerical Mathematics,* vol. 28, pp. 775--784, 1988.

[41] C. Levcopoulos and O. Petersson, "Splitsort—an adaptive sorting algorithm," *Information Processing Letters,* vol. 39, pp. 205--211, 1991.

[42] P. McIlroy, Optimistic Sorting and Information Theoretic Complexity, Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1993.

[43] "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Timsort.

[44] C. A. Hoare, "Quicksort," *The Computer Journal,* vol. 5, no. 1, pp. 10--16, 1962.

[45] H. E. Seward, *Information sorting in the application of electronic digital computers to business operations. Master's thesis,* Stanford University, 1954.

[46] N. a. G. J. a. K. R. a. M. D. Govindaraju, "GPUTeraSort: high performance graphics co-processor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, ACM, 2006, pp. 325--336.

[47] P. J. a. S. S. D. a. I. B. R. a. R. G. R. Varman, "Merging multiple lists on hierarchical-memory multiprocessors," *Journal of Parallel and Distributed Computing,* vol. 12, pp. 171--177, 1991.

[48] J. a. S. P. a. P. F. Singler, "MCSTL: The multi-core standard template library," in *European Conference on Parallel Processing*, Springer, 2007, pp. 682--694.

[49] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30--May 2, 1968, spring joint computer conference*, ACM, 1968, pp. 307--314.

[50] G. a. N. A. Bilardi, "Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines," *SIAM Journal on Computing,* vol. 18, pp. 216--228, 1989.

[51] L. Liu, "Acceleration of k-Nearest Neighbor and SRAD Algorithms Using Intel FPGA SDK for OpenCL," 2018.

[52] W. D. a. M. A. Frazer, "Samplesort: A sampling approach to minimal storage tree sorting," *Journal of the ACM (JACM),* vol. 17, pp. 496--507, 1970.

[53] J. a. C. Y. Huang, "Parallel sorting and data partitioning by sampling," 1983.

[54] E. a. K. L. V. Solomonik, "Highly scalable parallel sorting," 2010.

[55] M. a. B. G. E. Zagha, "Radix sort for vector multiprocessors," in *Conference on High Performance Networking and Computing: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, 1991, pp. 712--721.

[56] A. a. K. Y. Sohn, "Load balanced parallel radix sort," in *Proceedings of the 12th international conference on Supercomputing*, ACM, 1998, pp. 305--312.

[57] V. Choudhary, "developerinsider," [Online]. Available: https://developerinsider.co/introduction-to-divide-and-conquer-algorithm-design-paradigm/.

[58] Z. a. W. M. a. P. D. Marsza{\l}ek, "Fully Flexible Parallel Merge Sort for Multicore Architectures," *Complexity,* vol. 2018, 2018.

[59] A. Aslam, M. S. Ansari and S. Varshney, "Non-Partitioning Merge-Sort: Performance Enhancement by Elimination of Division in Divide-and-Conquer Algorithm," 2016.

[60] M. Wozniak, Z. Marszaek, M. Gabryel and R. K. Nowicki, "Modified merge sort algorithm for large scale data sets," 2013.

[61] T. Sutikno and I. M. I. S. D. Subroto, "The architecture of indonesian publication index:A major indonesian academic database," *TELKOMNIKA Telecommunication, Computing,Electronics and Control,* vol. 12, 2014.

Appendices

Appendix A

Additional Tables and Graphs

The following tables show experiment result between DRS algorithm and benchmarks which represent in graphs in chapter 4.

Average of ET				
# of elements	**α−Stack**	**DivideRuns**	**TimSort**	**Grand Total**
1000000	549.51	417.48	549.51	505.5
1100000	592.35	462.03	593.47	549.2833333
1200000	649.89	505.23	651.33	602.15
1300000	713.17	552.31	714.22	659.9
1400000	772.16	598.21	774.61	714.9933333
1500000	830.74	641.43	837.97	770.0466667
1600000	893.41	688.14	899.64	827.0633333
1700000	955.05	732.85	957.93	881.9433333
1800000	1015.16	779.59	1021.01	938.5866667
1900000	1094.32	834.81	1096.32	1008.483333
2000000	1142.96	871.93	1143.49	1052.793333
Grand Total	**837.16**	**644.00**	**839.95**	**773.7039394**

Table 12 : Sequential Sorting time comparisons (Random Elements)

Average of ET				
# of elements	α−Stack	DivideRuns	TimSort	Grand Total
1000000	4.65	4.88	4.71	4.746666667
1100000	5.24	5.19	5.21	5.213333333
1200000	5.82	5.7	5.61	5.71
1300000	6.38	6.23	6.36	6.323333333
1400000	6.66	6.66	6.5	6.606666667
1500000	7.33	7.32	7.46	7.37
1600000	7.75	7.79	7.55	7.696666667
1700000	8.51	8.65	8.47	8.543333333
1800000	9.03	8.95	8.72	8.9
1900000	9.37	9.55	9.42	9.446666667
2000000	9.77	9.77	9.6	9.713333333
Grand Total	**7.32**	**7.34**	**7.24**	**7.30**

Table 13 : Sequential Sorting time comparisons (Sorted Elements)

Average of ET				
# of elements	α−Stack	DivideRuns	TimSort	Grand Total
1000000	81.1	71.79	81.45	78.11
1100000	89.15	78.84	88.72	85.57
1200000	96.71	87.1	96.67	93.49
1300000	104.92	93.8	104.46	101.06
1400000	114.21	102.03	114.94	110.39
1500000	120.92	108.3	121.9	117.04
1600000	130.83	116.03	131.03	125.96
1700000	138.37	123.41	140.17	133.98
1800000	146.08	131.04	147.5	141.54
1900000	154.98	139.67	158.19	150.95
2000000	170.58	146	162.04	159.54
Grand Total	**122.53**	**108.91**	**122.46**	**117.97**

Table 14: Sequential Sorting time comparisons (10 Runs)

Average of ET				
# of elements	**α−Stack**	**DivideRuns**	**TimSort**	**Grand Total**
1000000	155.38	131.81	155.38	147.5233333
1100000	169.1	145.2	170.05	161.45
1200000	186.16	158.45	183.81	176.14
1300000	201.69	171.83	200.07	191.1966667
1400000	218.31	184.01	215.12	205.8133333
1500000	232.57	197.99	237.36	222.64
1600000	247.5	211.66	249.83	236.33
1700000	262.49	224.51	264.61	250.5366667
1800000	277.45	238.28	279.21	264.98
1900000	292.62	251.37	293.71	279.2333333
2000000	308.88	264.67	309.69	294.4133333
Grand Total	**232.01**	**198.16**	**232.62**	**220.9324242**

Table 15: Sequential Sorting time comparisons (100 Runs)

Average of ET				
# of elements	**α−Stack**	**DivideRuns**	**TimSort**	**Grand Total**
1000000	234.64	192.98	234.92	220.8466667
1100000	254.14	213.09	254.43	240.5533333
1200000	276.84	231.09	278.07	262
1300000	300.01	250.18	300.39	283.5266667
1400000	322.69	269.99	324.69	305.79
1500000	345.01	288.68	349.31	327.6666667
1600000	367.71	307.97	368.15	347.9433333
1700000	390.04	327.01	391.71	369.5866667
1800000	420.82	351.64	423.35	398.6033333
1900000	438.92	367.48	437.15	414.5166667
2000000	461.59	387.06	460.52	436.39
Grand Total	**346.58**	**289.74**	**347.52**	**327.9475758**

Table 16: Sequential Sorting time comparisons (1000 Runs)

Average of ET			
# of elements	ParallelDivide-Runs	ParallelMergeSort	Grand Total
1000000	118.63	161.35	139.99
1100000	121.96	165.24	143.6
1200000	139.74	190.9	165.32
1300000	153.84	211.16	182.5
1400000	160.27	220.62	190.445
1500000	176.14	242.34	209.24
1600000	178.27	249.71	213.99
1700000	188.31	263.9	226.105
1800000	213.09	301.17	257.13
1900000	224.66	318.62	271.64
2000000	232.32	327.93	280.125
Grand Total	**173.3845455**	**241.1763636**	**207.2804545**

Table 17: Parallel Sorting time comparisons (Random Elements)

Average of ET			
# of elements	ParallelDivide-Runs	ParallelMergeSort	Grand Total
1000000	15.35	89.52	52.435
1100000	16.81	99.02	57.915
1200000	17.87	109.7	63.785
1300000	19.07	119.98	69.525
1400000	19.78	129	74.39
1500000	21.46	138.73	80.095
1600000	19.98	150.18	85.08
1700000	21.41	155.13	88.27
1800000	22.23	168.52	95.375
1900000	24.82	176.65	100.735
2000000	28.96	205.08	117.02
Grand Total	**20.70363636**	**140.1372727**	**80.42045455**

Table 18: Parallel Sorting time comparisons (Sorted Elements)

The following graphs show experiment with change of α in α−stack algorithm (α=2)

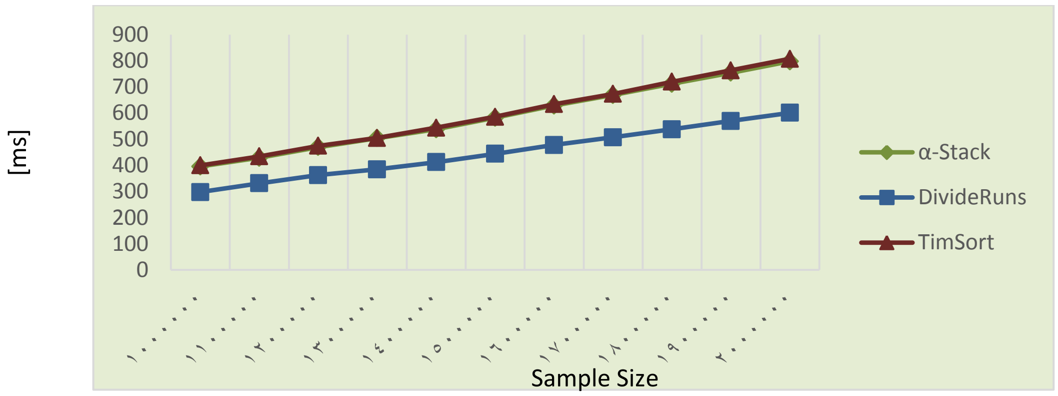


Figure 35: Sequential Sorting time comparisons (Random Elements with α=2)

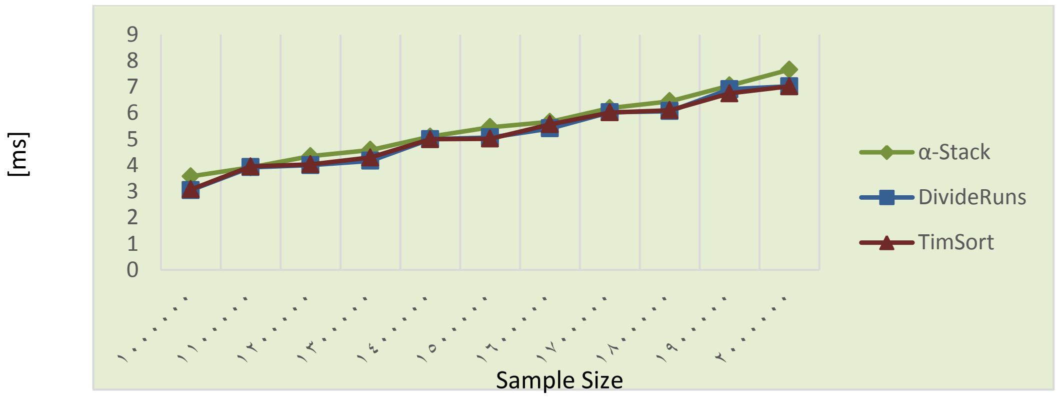


Figure 36: Sequential Sorting time comparisons (Sorted Elements with α=2)

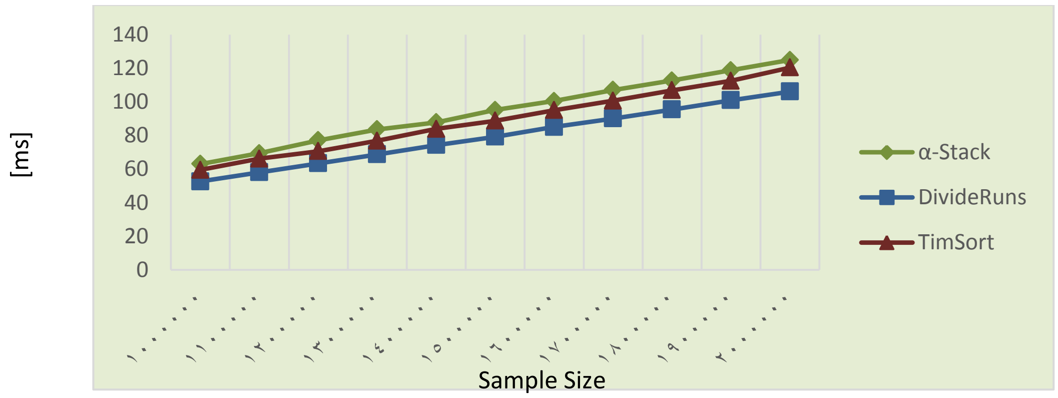


Figure 37: Sequential Sorting time comparisons (10Runs Elements with α=2)

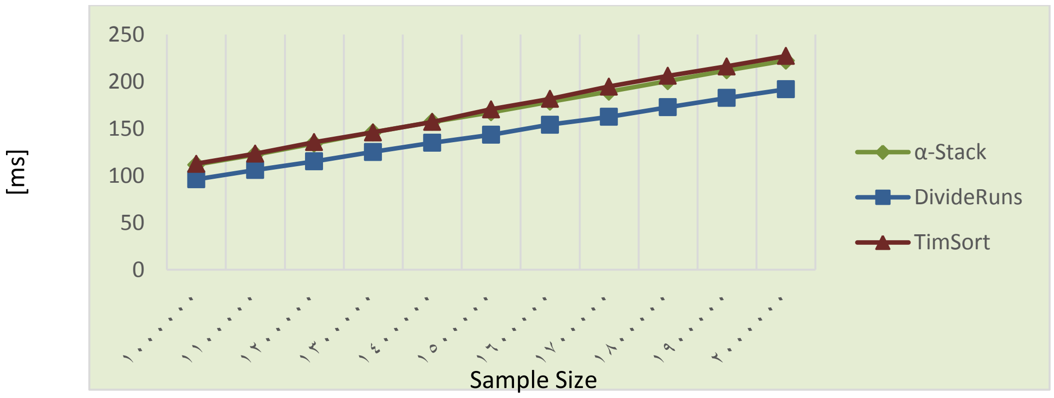


Figure 38: Sequential Sorting time comparisons (100Runs Elements with α=2)

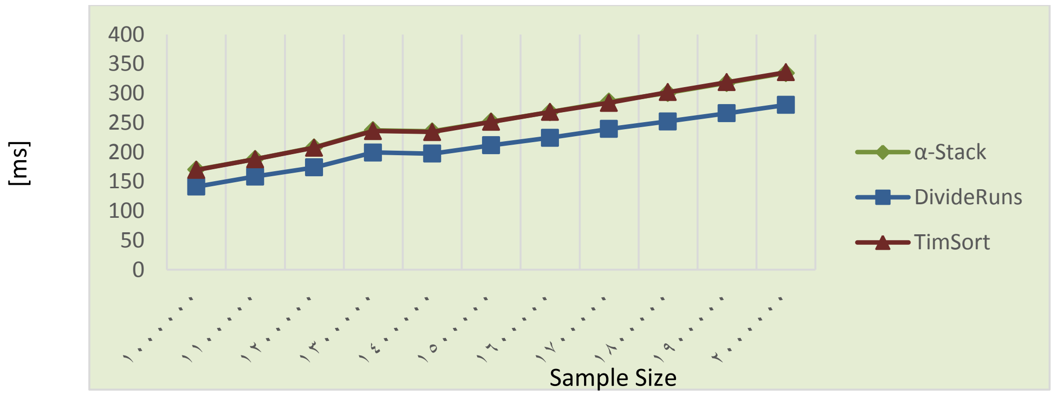


Figure 39: Sequential Sorting time comparisons (1000Runs Elements with α=2)

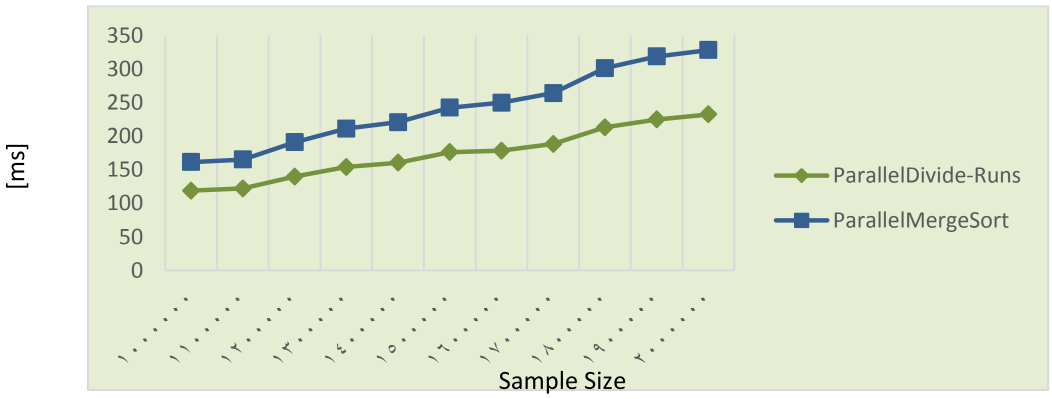


Figure 40: Parallel Sorting time comparisons (Random Elements with α=2)

Appendix B

Sequential Divide−Runs Algorithm Code

DRS.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace RunsMerge
{
    class DRS
    {
      static public void FindRuns(int[] numbers, int[] aux, int[,] Divid)
        {
            int acendOrder = 0;
            int descOrder = 0;
            int right = 0, mid = 0;
            int DividCount = 0;

            for (int i = 0; i <= numbers.Length − 1; i++)
            {
```

```
if (i == numbers.Length - 1)
{
    if (right < numbers.Length && descOrder == 1)
    {
     Array.Reverse(numbers, mid, right - mid + 1);
     Divid[DividCount,0] = mid;
     Divid[DividCount,1] = right;
     DividCount++;
    }
    else
        if (right < numbers.Length)
    {
        Divid[DividCount, 0] = mid;
        Divid[DividCount, 1] = right;
        DividCount++;
    }
    break;
}
else if (acendOrder == 0 && descOrder == 0)
{
    if (numbers[i] <= numbers[i + 1] && i + 1 >= numbers.Length - 1)
    {
        right++;
        Divid[DividCount, 0] = mid;
        Divid[DividCount, 1] = right;
        DividCount++;
        mid = right + 1;
        right++;
        acendOrder = 0;
        descOrder = 0;
    }
```

```
                    else if (numbers[i] <= numbers[i + 1])
                    {
                        acendOrder = 1;
                        right++;
                    }
                    else if (numbers[i] >= numbers[i + 1] && i + 1 >= numbers.Length
        - 1)
                    {
                        right++;
                        Array.Reverse(numbers, mid, right - mid + 1);
                        Divid[DividCount, 0] = mid;
                        Divid[DividCount, 1] = right;
                        DividCount++;
                        mid = right + 1;
                        right++;
                        acendOrder = 0;
                        descOrder = 0;
                    }
                    else
                    {
                        descOrder = 1;
                        right++;
                    }
                }
                else if (acendOrder == 1)
                {
                    if (numbers[i] <= numbers[i + 1] && i + 1 >= numbers.Length - 1)
                    {
                        right++;
                        Divid[DividCount, 0] = mid;
                        Divid[DividCount, 1] = right;
```

```
    DividCount++;
    mid = right + 1;
    right++;
    acendOrder = 0;
    descOrder = 0;
}
else if (numbers[i] <= numbers[i + 1])
    right++;
else
{
    //**********
    if (right - mid == 1)
    {
        int tmp = numbers[i];
        numbers[i] = numbers[i + 1];
        numbers[i + 1] = tmp;

        if (numbers[i - 1] > numbers[i])
        {
            tmp = numbers[i - 1];
            numbers[i - 1] = numbers[i];
            numbers[i] = tmp;
        }
        right++;



    }

    //************
    else
    {
```

```
                Divid[DividCount, 0] = mid;
                Divid[DividCount, 1] = right;
                DividCount++;
                mid = right + 1;
                right++;
                acendOrder = 0;
                descOrder = 0;
            }
        }
    }
    else
    {
        if (numbers[i] >= numbers[i + 1] && i + 1 >= numbers.Length - 1)
        {
            right++;
            Array.Reverse(numbers, mid, right - mid + 1);
            Divid[DividCount, 0] = mid;
            Divid[DividCount, 1] = right;
            DividCount++;
            mid = right + 1;
            right++;
            acendOrder = 0;
            descOrder = 0;
        }
        else if (numbers[i] >= numbers[i + 1])
        {
            right++;
        }
        else
        {
            if (right - mid == 1)
```

```
                    {
                        int tmp = numbers[i];
                        numbers[i] = numbers[i + 1];
                        numbers[i + 1] = tmp;
                        if (numbers[i - 1] < numbers[i])
                        {
                            tmp = numbers[i - 1];
                            numbers[i - 1] = numbers[i];
                            numbers[i] = tmp;
                        }
                        right++;
                    }
                    else
                    {
                        Array.Reverse(numbers, mid, right - mid + 1);
                        Divid[DividCount, 0] = mid;
                        Divid[DividCount, 1] = right;
                        DividCount++;
                        mid = right + 1;
                        right++;
                        acendOrder = 0;
                        descOrder = 0;
                    }
                }
            }
        }
    if (DividCount > 1)
            DividRuns(numbers, aux, Divid, 0, DividCount - 1);
}
public static void DividRuns(int[] array, int[] aux, int[,] Divid, int left, int right)
{
```

```
            if (left < right)
            {
                int middleIndex = (left + right) / 2;
                DividRuns(array, aux, Divid, left, middleIndex);
                DividRuns(array, aux, Divid, middleIndex + 1, right);
                Merge(array, aux, Divid[left, 0], Divid[middleIndex, 1], Divid[right, 1]);
            }
        }
        private static void Merge(int[] array, int[] aux, int leftIndex, int middleIndex,
int right)
        {
            int rightIndex = middleIndex + 1;
            int auxIndex = leftIndex;
            int start = leftIndex;
            int num_elements = (right − leftIndex + 1);
            while (leftIndex <= middleIndex && rightIndex <= right)
            {
                if (array[leftIndex] <= array[rightIndex])
                {
                    leftIndex++;
                    start++;
                    num_elements−−;
                    auxIndex++;
                }
                else
                    break;
            }
            while (leftIndex <= middleIndex && rightIndex <= right)
            {
                if (array[leftIndex] <= array[rightIndex])
                {
                    aux[auxIndex] = array[leftIndex++];
                }
                else
                {
                    aux[auxIndex] = array[rightIndex++];
                }
                auxIndex++;
            }
            while (leftIndex <= middleIndex)
            {
                aux[auxIndex] = array[leftIndex++];
                auxIndex++;
            }
            while (rightIndex <= right)
            {
                aux[auxIndex] = array[rightIndex++];
                auxIndex++;
            }
            Array.Copy(aux, start, array, start, num_elements);
        }
    }
}
```

Appendix C

Parallel Divide-Runs Algorithm Code

ParallelRunsMerge.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace ParallelRunsMerge
{
    class ParallelRunMerge2
    {
        private static int _maxParallelDepth;
        protected static int DetermineMaxParallelDepth()
        {
            // const int MaxTasksPerProcessor = 8;
            const int MaxTasksPerProcessor = 1;
            int maxTaskCount = Environment.ProcessorCount *
MaxTasksPerProcessor;
            int icheck = (int)Math.Log(maxTaskCount, 2);
            return (int)Math.Log(maxTaskCount, 2);
        }
        public static void MainTask(int[] numbers)
        {
            //Find Runs First RunMerge
            int[] aux = new int[numbers.Length];
            int[,] Divid = new int[numbers.Length / 8 + 10, 2];
            int dividCountNo = numbers.Length / 8;
            dividCountNo = dividCountNo / 6 + 2;
            int ParallelNumber = (numbers.Length - 1) / 6;
            int[,] Divid1 = new int[dividCountNo, 2];
            int[,] Divid2 = new int[dividCountNo, 2];
            int[,] Divid3 = new int[dividCountNo, 2];
            int[,] Divid4 = new int[dividCountNo, 2];
            int[,] Divid5 = new int[dividCountNo, 2];
            int[,] Divid6 = new int[dividCountNo, 2];

            Task<int> task1F = Task<int>.Factory.StartNew(() => { return
FindRuns(numbers, Divid1, 0, ParallelNumber + 1); });
            Task<int> task2F = Task<int>.Factory.StartNew(() => { return
FindRuns(numbers, Divid2, (1 * ParallelNumber) + 1, (2 * ParallelNumber) + 1);
});
            Task<int> task3F = Task<int>.Factory.StartNew(() => { return
FindRuns(numbers, Divid3, (2 * ParallelNumber) + 1, (3 * ParallelNumber) + 1);
});
            Task<int> task4F = Task<int>.Factory.StartNew(() => { return
FindRuns(numbers, Divid4, (3 * ParallelNumber) + 1, (4 * ParallelNumber) + 1);
});
            Task<int> task5F = Task<int>.Factory.StartNew(() => { return
FindRuns(numbers, Divid5, (4 * ParallelNumber) + 1, (5 * ParallelNumber) + 1);
});
```

```
            Task<int> task6F = Task<int>.Factory.StartNew(() => { return
FindRuns(numbers, Divid6, (5 * ParallelNumber) + 1, numbers.Length); });
            int DividCount1 = task1F.Result;
            int DividCount2 = task2F.Result;
            int DividCount3 = task3F.Result;
            int DividCount4 = task4F.Result;
            int DividCount5 = task5F.Result;
            int DividCount6 = task6F.Result;
            for (int i = 0; i < DividCount1; i++)
            {
                Divid[i, 0] = Divid1[i, 0];
                Divid[i, 1] = Divid1[i, 1];
            }
            int iCount = DividCount1;
            for (int i = 0; i < DividCount2; i++)
            {
                Divid[i + iCount, 0] = Divid2[i, 0];
                Divid[i + iCount, 1] = Divid2[i, 1];
            }
            iCount += DividCount2;
            for (int i = 0; i < DividCount3; i++)
            {
                Divid[i + iCount, 0] = Divid3[i, 0];
                Divid[i + iCount, 1] = Divid3[i, 1];
            }
            iCount += DividCount3;
            for (int i = 0; i < DividCount4; i++)
            {
                Divid[i + iCount, 0] = Divid4[i, 0];
                Divid[i + iCount, 1] = Divid4[i, 1];
            }
            iCount += DividCount4;
            for (int i = 0; i < DividCount5; i++)
            {
                Divid[i + iCount, 0] = Divid5[i, 0];
                Divid[i + iCount, 1] = Divid5[i, 1];
            }
            iCount += DividCount5;
            for (int i = 0; i < DividCount6; i++)
            {
                Divid[i + iCount, 0] = Divid6[i, 0];
                Divid[i + iCount, 1] = Divid6[i, 1];
            }
            int DividCount = 0;
            DividCount = DividCount1 + DividCount2 + DividCount3 + DividCount4 +
DividCount5 + DividCount6;
            int iDividArr = (DividCount - 1) / 2;
            _maxParallelDepth = DetermineMaxParallelDepth();
            MergeSort(numbers, aux, Divid, 0, DividCount - 1, 1);

        }
        public static int FindRuns(int[] a, int[,] Divid, int lo, int hi)
        {
            Debug.Assert(lo < hi);
            int DividCount = 0;
            int minRun = 8;
            int nRemaining = hi - lo;
            while (lo < hi)
            {
                int runHi = lo + 1;
                if (runHi == hi)
                {
```

```
        Divid[DividCount, 0] = lo;
        Divid[DividCount, 1] = (hi - 1);
        DividCount++;
        return DividCount;
    }
    // Find end of run, and reverse range if descending
    if ((a[runHi++] < a[lo]))
    { // Descending
        while (runHi < hi && (a[runHi] <= a[runHi - 1]))
            runHi++;
        Array.Reverse(a, lo, runHi - lo);
        // reverseRange(a, lo, runHi);
    }
    else
    {                              // Ascending
        while (runHi < hi && (a[runHi] >= a[runHi - 1]))
            runHi++;
    }

    int runLen = runHi - lo;

    // If run is short, extend to min(minRun, nRemaining)
    if (runLen < minRun)
    {
        int force = nRemaining <= minRun ? nRemaining : minRun;
        binarySort(a, lo, lo + force, lo + runLen);
        runLen = force;
    }



   Divid[DividCount, 0] = lo;
```

```
                Divid[DividCount, 1] = runLen + lo − 1;
                DividCount++;
                lo += runLen;
                nRemaining −= runLen;
            }
            return DividCount;
        }

        public static void MergeSort(int[] array, int[] aux, int[,] Divid, int iStart, int
iEnd, int recursionDepth)
        {

            if (iStart >= iEnd)
                return;
            int middle = (iStart + iEnd) / 2;
            if (recursionDepth <= _maxParallelDepth)
            {
             Task task1 = new Task(() => MergeSort(array, aux, Divid, iStart,
middle, recursionDepth + 1));
                Task task2 = new Task(() => MergeSort(array, aux, Divid, middle +
1, iEnd, recursionDepth + 1));

                task1.Start();
                task2.Start();
                task1.Wait();
                task2.Wait();
            }
            else
            {
                MergeSort(array, aux, Divid, iStart, middle, recursionDepth + 1);
                MergeSort(array, aux, Divid, middle + 1, iEnd, recursionDepth + 1);
            }
            MergeTechnique.Merge(array, aux, Divid[iStart, 0], Divid[middle, 1],
Divid[iEnd, 1]);

        }
        public static void binarySort(int[] a, int lo, int hi, int start)
        {
            Debug.Assert(lo <= start && start <= hi);
            if (start == lo)
                start++;
            for (; start < hi; start++)
            {
                int pivot = a[start];

                // Set left (and right) to the index where a[start] (pivot) belongs
                int left = lo;
                int right = start;
                Debug.Assert(left <= right);
                /*
                 * Invariants:
                 *   pivot >= all in [lo, left).
                 *   pivot <  all in [right, start).
                 */
                while (left < right)
                {
                    int mid = (left + right) / 2;
                    if (pivot < a[mid])
                        right = mid;
                    else
                        left = mid + 1;
                }
                int n = start - left;
                switch (n)
                {
                    case 2:
                        a[left + 2] = a[left + 1];
                        break;
                    case 1:
                        a[left + 1] = a[left];
                        break;
                    default: Array.Copy(a, left, a, left + 1, n); break;
                }
                a[left] = pivot;
            }
        }
    }
}
```