Thesis no: BCS-2015-05

Performance analysis of multithreaded sorting algorithms

Kevin Jouper, Henrik Nordin

Dept. Computer Science & Engineering
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science. The thesis is equivalent to 10 weeks of full time studies.

Contact Information:
Author(s):
Kevin Jouper
E-mail: kevin.jouper@gmail.com

Henrik Nordin
E-mail: henknordin@gmail.com

University advisor:
Dr. Julia Sidorova
Dept. Computer Science & Engineering

Dept. Computer Science & Engineering
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context. Almost all of the modern computers today have a CPU with multiple cores, providing extra computational power. In the new age of big data, parallel execution is essential to improve the performance to an acceptable level. With parallelisation comes new challenges that needs to be considered.
Objectives. In this work, parallel algorithms are compared and analysed in relation to their sequential counterparts, using the Java platform. Through this, find the potential speedup for multithreading and what factors affects the performance. In addition, provide source code for multithreaded algorithms with proven time complexities.
Methods. A literature study was conducted to gain knowledge and deeper understanding into the aspects of sorting algorithms and the area of parallel computing. An experiment followed of implementing a set of algorithms from which data could be gather through benchmarking and testing. The data gathered was studied and analysed with its corresponding source code to prove the validity of parallelisation.
Results. Multithreading does improve performance, with two threads in average providing a speedup of up to 2x and four threads up to 3x. However, the potential speedup is bound to the available physical threads of the CPU and dependent of balancing the workload.
Conclusions. The importance of workload balancing and using the correct number of threads in relation to the problem to be solved, needs to be carefully considered in order to utilize the extra resources available to its full potential.

Keywords: Parallel algorithms, sorting, performance, time complexity

Acknowledgments

We would like to thank our supervisor ***Julia Sidorova*** for the support and energetic motivation.

A very special acknowledgement to ***Bengt Aspvall*** for the help he has provided in relation to algorithms.

Additional thanks ***Tomas Petersson*** and ***Mattias Eriksson*** for helping us with mathematical equations.

Contents

Chapter 1

Introduction

1.1 Introduction

Increasing performance is the basis for improving solutions for a given problem to its fullest potential by utilizing every tool and resource to our advantage. In computer science, the resources and tools could be seen as the hardware and software, while the performance would be the collaboration between the two. To fully utilize and increase performance, not only must the problem, resources and tools be understood, but also when and how to use them at appropriate situations.

The given problem in this work, is one of the biggest fundamental problems, namely sorting. Sorting have always been of interest and solutions have been extensively studied and documented. Numerous solution have been proposed and accepted, using different designs and techniques, each with its own advantage and disadvantage.

The aim in this work is to evaluate the performance of some of these approaches, as to what advantages and disadvantages they have and why they occur. In addition, this work also aims to evaluate an improvement technique of using extra raw computer power, namely multithreading. Improving the performance by utilizing multithreading, introduces new problems and factors, which needs to be considered. Knowing and understanding these factors, could prove useful when utilized correctly.

For this work, the given implementations are all in Java, and have been thoroughly tested and analysed. However, it needs to be emphasised that focus does not lay on optimization for the given implementations. Instead, the implementations shows what is possible in terms of increasing performance, through the results and analysis of this work.

1.2 Research questions

For this work, the following algorithms were chosen for the experiment:

- Rank sort
- Merge sort
- Quicksort
- Bitonic sort
- Bucket sort
- Radix sort

Although it is generally expected that multithreading does improve performance of algorithms, this work will focus on the following research questions:

- How much can an algorithm's efficiency and performance be improved by multithreading?
- What affects the performance and efficiency of algorithms, how are they different in multithreaded environments?
  - Is there a threshold for multithreaded algorithms in terms of performance, and what are the reasons for it?
  - How does unbalanced workload in multithreaded environments change the performance of an algorithm?
- How can time complexity be proven, by conversion of source code to mathematical statements?

Chapter 2

Related Work

In this chapter, previous studies related to areas of this work, are summarized and compared to illustrate the findings of this work in a broader context.

2.1 Related work

In [1], a quicksort algorithm was implemented to utilize the multithread technology in Java. An experiment was made, in which the multithreaded version was run through a data set, using a different number of threads for each run. The results were then compared to a traditional implementation of the algorithm. They find that multithreading the algorithm improves efficiency up to 40 percent, using two threads. In addition, it proved the existence of a threshold, after which the improvement in efficiency decreases until it becomes negligible. This is likely due to the fact that the testing platform was a CPU with two cores, making it most efficient when running with two threads.

A similar experiment was made by [2]. Instead of using multithreading in Java, POSIX Threads (or Pthreads) was utilize and implemented in C++. The same conclusion was reached: a parallel implementation of quicksort is more time-efficient than a traditional sequential one. Interestingly, in this experiment, the implementation that utilized four threads proved to be most efficient. This is due to the fact that the test environment being used, a dual-core processor, offered two threads to be run for each core, thus enables four threads to run simultaneously, increasing the resources available.

In another experiment [3], performance of the heap sort algorithm was optimized through Windows threads library API. The experiment is carried out on a dual-core machine and a single-core machine, later comparing the two. The conclusion was reached that via multithreading, the performance is improved significantly on the dual-core machine. However, there is very little difference between two threads and four threads, since the machine is limited to two cores, and can therefore only run two threads simultaneously. On a single-core machine, there is little improvement due to multithreading, since it only has one core and can only run one thread at the time. These two papers proves that it is possible to reach similar results, using a different environment and different methods.

Similar studies have been made, but in other areas. Using OpenMP [4], a simple program performing matrix multiplication is written and implemented in three different versions: a sequential, one using two cores and one using four logical processors. It concludes that parallel execution significantly outperforms the sequential, with speedup for two cores varying between $1.5 - 2.0$ and for four logical core varying between $1.4 - 2.8$ depending on the size of the data sets. Even though the results may be specific to the test environment, it still serves as a proof of concept and its validity seems to be in line with other studies.

In [5], a different idea was undertaken: namely different parallel sorting algorithms for GPUs are executed and compared using an eight-core CPU as test environment. The paper concludes that an implementation of radix sort is the fastest for GPUs, by a great margin. In addition, the experiment also executes algorithms with different number of threads, from two threads up to eight threads. For quicksort and radix sort, the speedup increases with the number of threads in relation to the size of the data sets. As the data sets grows larger, the speedup increases pending on the number of threads. Once again, the concept parallel algorithms and multithreading seems to be valid in many areas.

In [6], the performance of the most common sorting algorithms are compared. The algorithms were tested on three different data sets: sorted, reversely sorted and randomized, with ranges from 100 to 1,000,000 elements. This method is similar to the proposed testing method employed in this work and should provide a good point of reference. From the results, one can draw the conclusion that different algorithms have specific properties, and therefore have advantages and disadvantages related to the data set. Quicksort performs great on a randomized data set, but falls behind in comparison to merge sort and heap sort on the data sets that were already sorted in some way. Bubble sort, as simple of an algorithm as it may be, outperforms more complex algorithms on already sorted data sets. Its simple design actually helps it perform quite well under these conditions. However, on a random data set, it falls significantly behind other algorithms. This confirms the theory that there is no perfect algorithm that always should be used. The decision of which algorithm to utilize is dependent on the context and resources available on the proposed application.

Similarly in [7], a study was made of comparing some of the most widely used sorting algorithms. The focus point was made on the best average case. In addition, there are several test runs from which an average time is calculated. The results were similar to the previous studies. In this case however, the range was between 10-10,000 elements. While it shows the difference in the performance of different algorithms, the max-range could be considered to small. Larger data sets would be needed in order to see the real difference in performance of the algorithms, as the execution time for most algorithms grows exponentially in relation to the number of elements to be sorted.

In [8], the same type of experiment was made. It comes to a similar conclusion

as the previous ones. This one also includes improved versions of some algorithms in addition to the standard ones. This shows that sequential algorithms can be improved for the intended sorting purpose. It also provides test results in the form of number of comparisons made for each algorithm on different ranges of elements. In addition, the test contains different types of input for the data sets. While this is something that covered throughout this work, it provides another interesting point of difference in performance between sorting algorithms. Finally, the paper provides a list of algorithms that are suitable for certain type of problems, which provides a good point of reference when making a decision of which algorithm to utilize.

In [9], three parallel algorithms are compared in a homogeneous cluster of workstations, using MPI. The paper reaches the same conclusion as the above described papers, with parallel execution improving the performance in terms of execution time. However, this study shows an alternative method parallelisation in contrast to the previously mentioned ones. In this, the parallelisation is not limited to the number of cores in a CPU, but to 12 physical machines in the cluster. Interestingly, the performance for all algorithms decreases after eight CPUs. After this, the overhead for communication between the processors becomes a bigger factor then the actual time for sorting elements. No real explanation is given for this specific limit, but one could assume that eight is the magic number for the threshold in performance for the proposed data set.

Chapter 3

Background

In this chapter, an overview is given for used theories and techniques in this work.

The area ***time complexity*** is defined to give a further understanding into the analysis of algorithms.

The ***algorithms*** chosen for this work are described with examples, in addition to specific characteristics for sorting.

The aspects of ***parallel computing*** is also given, for a further insight to the workings and inherent challenges of parallelisation.

3.1 Asymptotic computational complexity

Algorithms are described and analysed with respect to asymptotic computational complexity. There are three major asymptotic notations that are used for analysis. These describe asymptotic boundaries for algorithm's upper and lower bounds as growth functions. With this notation it is possible to describe mathematically and prove algorithms as worst, best and average cases due to their asymptotic boundaries. However, computational complexity does not only cover time complexity analysis, but also space complexity, parallel computation and other probabilistic analysis of algorithms[10].

In this work, the asymptotic computational complexity will refer to the worst case and best case of algorithms time complexity and parallel computation. However, the analysis of average case could be defined by asymptotic tight boundaries. An algorithm's average case is defined in a probabilistic way, such that a tight bound can be found between the best and worst case. However, the probabilistics of an algorithm is defined with respect to several factors, such as the work balance, number of threads, execution time, space usage[10]. See further in section 3.1.5 ***Average case***.

3.1.1 Time complexity

Time complexity is the means to describe an algorithm's execution time for a problem with given input. The time complexity of an algorithm is defined with the previously mentioned cases and that they are dependent on the internal structure

and order of the given input. Consider that an algorithm sorts input that is already sorted. This case can be the best case of an algorithm since it only needs to check if the list is sorted. However, when an input is sorted in reversed order, some algorithms time complexity will spike, and be defined as their worst case. This means that the execution time to complete a problem can vary and that different cases are needed to describe when, why and how these inputs impact the algorithm execution time.

3.1.2 Growth functions

Growth functions are used as a common way to describe algorithms time complexity, space complexity, computational resources, and so forth. In time complexity, growth functions describe the runtime of an algorithm in relation to its purpose, including sorting, searching, insertion, deletion and so forth. All algorithms are different when it comes to implementation, which gives them different time complexities. For sorting algorithms, the difference does not lie in their intended purpose, but in the time it takes to execute it.

Knowing when an algorithm is suitable for implementation, where time might be a critical factor, growth function is a common way to describe them. However growth functions is just one way of doing that.

Comparison-based sorting algorithms are considered favourable when the determined growth function is closer to the linear growth function of n, which is the input size of elements given.

Function	**Growth rate**
c	constant
n	linear

Table 3.1: Constant growth rates.

Exponential growth functions

When a growth function is defined as exponential in the context of algorithms, it means that the exponential growth functions is proportional to the function's current value [11][12]. The current value could for example be the given input size of n elements. See further in figure 3.2

Function	Growth rate
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n^4	Fourth power

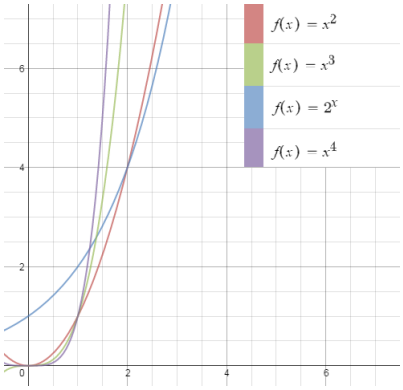


Table 3.2: Exponential growth.

Logarithm growth functions

When a growth function is defined as logarithmic, it means that the logarithm growth function is the inverse of the exponential growth function and is proportional to the inverse exponential function's value [11][12].

$$f^{-1}(n) = n^2 \Leftrightarrow f(n) = log_2\, n \tag{3.1}$$

The current value could for example be the given input size of n elements. See further in figure 3.3

Function	Growth rate
$log\ n$	Logarithmic
$log_2\ n$	Binary logarithm
$log_e\ n$	Natural logarithm
$n\ log\ n$	Log linear

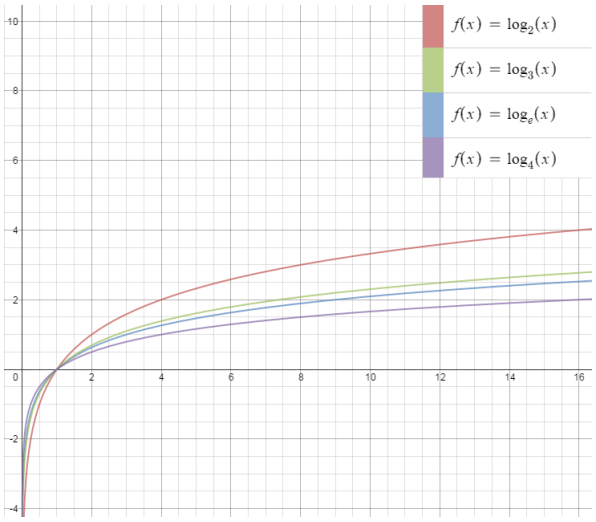


Table 3.3: Logarithmic growth.

3.1.3 Worst case

Worst-case analysis, covers the algorithm's longest time to complete any problem with n elements as input. It covers the case, where an algorithm's runtime for n is determined by its worst growth factor and that the growth factor is finite for input with n elements. A defined worst-case growth factor such as n^2, means that the algorithm can never have a growth rate greater than n^2 for any input with n elements. This is called the upper bound for a given algorithm. Analysing the upper bound, the asymptotic notations Big-O (O) is used, which is defined as following.

Big-O notation

Big-O, big oh, notation is defined as follows[10]:

for a given function g(n), we denote by O(g(n)) the set of functions.

$$O(g(n)) = \{f(n) : \textit{there exists positive constants } c \textit{ and } n_0 \textit{ such that} \\ 0 \leq f(n) \leq cg(n) \textit{ for all } n \geq n_0\} \tag{3.2}$$

3.1.4 Best case

Best-case analysis, covers algorithm's fastest time to complete any problem with n elements as input. It covers the case, that an algorithm's runtime for n is determined by its best growth factor and that the growth factor is finite for input with n elements. A defined best-case growth factor such as n^2, means that the algorithm can never have a growth rate less than n^2 for any sizes of n elements. This is called the lower bound for a given algorithm.

Analysing the lower bound, the asymptotic notations Omega is used, which is defined as following.

Big-Ω notation

Big-Ω, big omega, is defined as follows[10]:

for a given function g(n), we denote by (g(n))the set of functions

$$\Omega(g(n)) = \{f(n) : \textit{there exists positive constants } c \textit{ and } n_0 \textit{ such that} \\ 0 \leq cg(n) \leq f(n) \textit{ for all } n \geq n_0\} \tag{3.3}$$

3.1.5 Average case

Average case analysis covers the average time for an algorithm to complete any problem with n elements as input. It covers the case where an algorithm's runtime

for n lies in between its own upper and lower boundary growth factors for any sizes of n. The average case depends on a probability distribution for all input of a given size n. It is usually more difficult to obtain than the best case and the worst case bounds, where collections of data must be collected in terms of execution time, space usage to analysis and estimate an average execution time.

The asymptotic notations of Theta represents the average case, which is defined as following.

Θ notation

Theta is defined as follows[10]:

for a given function g(n), we denote by (g(n))the set of functions

$$\Theta(g(n)) = \{f(n) : \textit{there exists positive constants } c_1,\ c_2 \textit{ and } n_0 \textit{ such that} \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \textit{ for all } n \geq n_0\} \tag{3.4}$$

3.2 Sorting techniques and characteristics

Sorting algorithms sort a set of data in a specific order, such as numeric order or alphabetic order. There are different approaches in terms of design and techniques for sorting algorithms, however they can in general be classified as following below.

3.2.1 Computational complexity

Sorting algorithms are primarily evaluated with respect to the number of elements to be sorted. For this, best, worst and average cases are defined for each algorithm. See section 3.1 ***asymptotic computational complexity.***

3.2.2 Space complexity

Memory usage is another important aspect of sorting algorithms, which in some cases can be more important than the execution time. Ideally, one wants to keep the value for this parameter small, but extra memory can often improve execution times, which leads to a time-space trade-off. One can not have both, and therefore needs to find a reasonable compromise for what is best for the intended usage. For this reason, in-place sorting algorithms are often slower than its counterpart.

3.2.3 In-place sort

An in-place sorting algorithm modifies the data set as it being processed by overwriting the input with the output, therefore not needing any additional memory.

When finished, the sorted data set occupies the same storage as the original data set. Popular algorithms include bubble sort, insertion sort and quicksort.

A not-in-place sorting algorithm utilizes extra storage or data structures for sorting a data set. Examples of this includes merge sort, bucket sort and radix sort.

3.2.4 Comparison sort

A comparison based sorting algorithm compares elements in a data set with each other using a comparison operator, such as less than, more than or equal to. Some of well known algorithms include bubble sort, quick sort and merge sort.

3.2.5 Distribution sort

In contrast to comparison based sorting algorithms, distribution sort algorithms have the characteristic that they normally do not use comparisons in order to sort data. Instead, these algorithms rely on prior knowledge about the data to be sorted, e.g. size, nr of digits or radix. Based on this knowledge, the data can be distributed in a sorted sequence using different data structures or techniques. Examples of algorithms include radix sort, bucket sort and counting sort.

3.2.6 Stable sort

Stable sorting algorithms maintain the internal order of elements with the same value in a data set. Many sorting algorithms that utilizes partitioning are often unstable, as they reorder the elements within a partition. They often can be implemented as stable, but as result they suffer in performance. Stability is not a problem, when a data set contains unique elements only.

3.2.7 Recursion

Recursion is a method in which the result depends on smaller similar problems being solved. A recursive algorithm calls "itself" with a specific input, solving a subproblem for each call. Once all the subproblems have been solved, the result is normally obtained. Recursive sorting algorithms include quicksort and merge sort.

In contrast, many sorting algorithms are ***iterative***. This means that the data set is iterated over, processing one element at time.

3.2.8 Sequential vs parallel

A sequential sorting algorithm executes itself in sequential, meaning it executes all instructions from start to finish without any extra assistance. In contrast, a

parallel algorithm executes multiple instructions at the same time using different processing devices, such as multi-cores or threads, before combining it all back together in the end. What often distinguishes sequential from parallel algorithms are that parallel algorithms can break down a problem into smaller subproblems. ***Divide and conquer*** is an algorithm design paradigm that work this way, and is a natural fit for parallel implementation. Quicksort and merge sort are examples of divide and conquer algorithms, that both can be implemented in parallel.

3.2.9 Sorting methods

There are a wide range of sorting methods, but the more common ones include:

- Exchange, e.g. bubble sort
- Selection, e.g selection sort
- Insertion, e.g. insertion sort
- Merging, e.g. merge sort
- Partitioning, e.g. quicksort

Some algorithms are a combination of different methods. Quicksort, for example, utilizes both partitioning and exchange methods.

3.3 Sorting algorithms

In this section, a general explanation is given for the algorithms used in this work.

3.3.1 Rank sort

Definition

Iterate over a data set to find an element's relative rank, and insert it to a new set according to rank.

Algorithm

Rank sort is a comparison based algorithm, that can been seen as a mix between selection sort and insertion sort. It works by iterating over a data set, comparing an element with every other element in the set in order to find its rank. When the correct rank has been calculated, the element is moved to a new set and inserted in its proper position. As seen in table 3.4, rank sort has a time complexity of $O(n^2)$ for all cases, as it needs to compare each element with all other elements in order to ensure that a correct index has been calculated, no matter what order the list may be in[13][14].

Time complexity	
Worst case	$O(n^2)$
Best case	$O(n^2)$
Average case	$O(n^2)$
Space complexity	
Worst case	$O(n)$

Table 3.4: Rank sort - time complexity.

Example

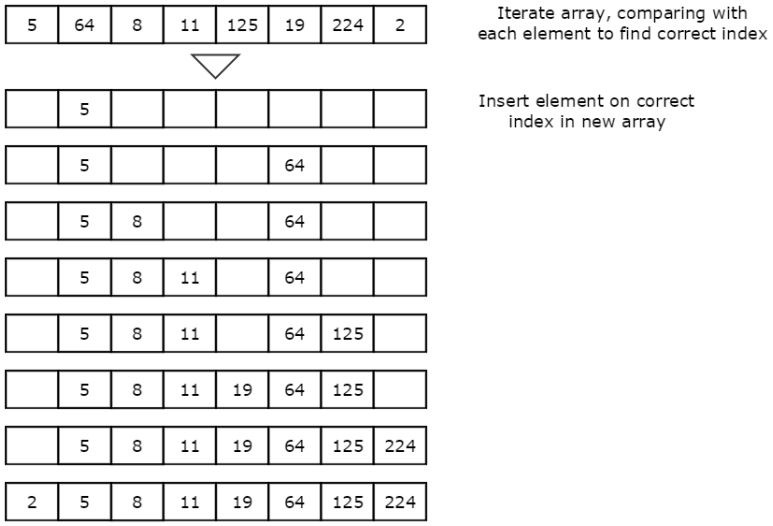


Figure 3.1: Rank sort - example.

In figure 3.1, each element in the original array is iterated over. Starting with element 5, it is compared to every other element in the array. After all the comparisons been made, it can be concluded that its proper rank relative to the data set is two, as the only smaller element in the array is element 2. Element 5 is then moved to a new data set, being inserted at its correct index.

The algorithm then moves to the next element in the data set, repeating the same routine until all elements been processed. The end result is a new data set in a sorted order.

3.3.2 Merge sort

Definition

Splits data set in half, sorts each half recursively, and merges them back together to a sorted set.

Time complexity	
Worst case	$O(n\ log\ n)$
Best case	$O(n\ log\ n)$
Average case	$O(n\ log\ n)$
Space complexity	
Worst case	$O(n)$ auxiliary

Table 3.5: Merge sort - time complexity.

Algorithm

Merge sort is a stable divide and conquer algorithm, and is considered to be one of the fastest. It recursively divides the data set into subarrays until each subarray consists of a single element. It then merges each subarray back, sorting each new subarray as it builds its way back to a single sorted array. Regardless of the shape of the data set to be sorted, merge sort performs the same number of steps, and will therefore have the same time complexity for all cases, $O(n\ log\ n)$, as seen in table 3.5. Even though it is an efficient algorithm in terms of sorting, it has a drawback in that it uses $O(n)$ extra memory when sorting. This makes it inefficient if memory usage is a key aspect when choosing a sorting algorithm to use. Due to it being a divide and conquer algorithm, it is possible to implement it in parallel[10].

Example

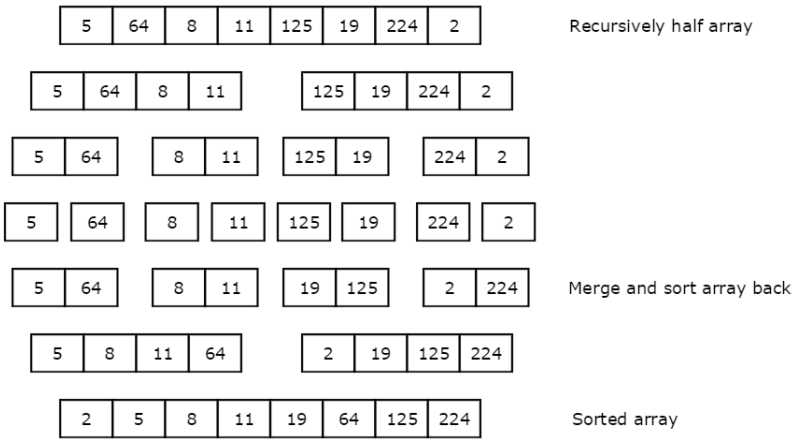


Figure 3.2: Merge sort - example.

In figure 3.2, the data set is recursively split in halves, until each element is on its own. Through the recursion, the subsets are merged back together, being sorted in the same instance. In this example, after the data set have been split down to single elements, element 5 and 64 are merged into a new subset. In this merge sequence, the elements are compared to each other and placed in a sorted order. In the next step, this subset merges with another one, in this case, a subset containing elements 8 and 11. Once again, the elements are compared and placed in a sorted order while merging. With only two subsets remaining, a final merge is done, resulting in sorted data set.

3.3.3 Quicksort

Definition

Pick an element (pivot) from the data set, partition the remaining elements into those greater than and less than pivot, and recursively sort the partitions.

Algorithm

Quicksort is an in-place divide and conquer algorithm. As its name suggest, it is one of the quickest sorting algorithms. The algorithm works in similar fashion as merge sort, by dividing a data set into smaller subsets. However, in difference to merge sort, a pivot element is chosen from which the data set is partitioned. When the partition occurs, all elements smaller than the pivot is moved to the

Time complexity	
Worst case	$O(n^2)$
Best case	$O(n\ log\ n)$
Average case	$O(n\ log\ n)$
Space complexity	
Worst case	$O(log\ n)$

Table 3.6: Quicksort - time complexity.

left hand side of the pivot and the greater ones to the right hand side. This routine is then repeated through recursion for each subset until the whole data set is sorted[10].

The key to a good performance for quicksort is to pick a proper pivot element. There are many different theories on how to pick the pivot. Best case scenario would be to pick the median value as pivot, as this would ensure that each partition would be of the same size, mimicking merge sort. However, this is generally hard task to achieve without actually sorting the data set. Should the pivot element be the min- or max element for each recursion, each new subset would only contain one element. This would make it very inefficient, only sorting one element at time, giving it a performance on par with the $O(n^2)$-algorithms. A randomly picked value as pivot would generally result in an average case $O(n\ log\ n)$, see table 3.6.

Due to it being a divide and conquer algorithm, it is suitable for implementation in parallel.

Example

According to figure 3.3, element 11 is picked as the pivot. Next, the data set is partitioned, placing all elements smaller then 11 in the lower half and all elements bigger then 11 in the upper half. Each subarray is recursively being partitioned until the data set is sorted.

In the lower half, element 5 is picked as pivot and the subarray is then partitioned again. This time, two new subarray are created. Since they only contain one element, each subarray is sorted. Through the recursion, all the smaller subarrays are put back together to the original subarray.

The upper half goes through the same procedure. In the end, the halfs are put together, resulting in a sorted data set.

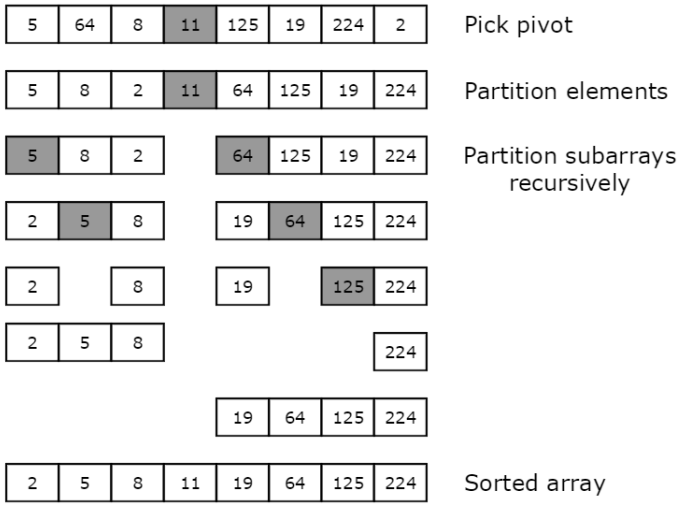


Figure 3.3: Quicksort - example.

3.3.4 Bitonic sort

Definition

Creates bitonic sequences of data set, until one increasing sequence remains.

Time complexity	
Worst case	$O(log(n)^2)$
Best case	$O(log(n)^2)$
Average case	$O(log(n)^2)$
Space complexity	
Worst case	$O(n\ log(n)^2)$

Table 3.7: Bitonic sort - time complexity.

Algorithm

Bitonic sort is a divide and conquer algorithm, specifically designed for running on parallel machines. It works by converting a data set into bitonic sequences, one that monotonically increases and one the monotonically decreases.

Example:
1 3 5 7 9 8 6 4 2
The sequence increases from 1 to 9, and then decreases. A sequence can also be considered as bitonic if it contains a cyclic shifting.

Example:
1 3 5 7 6 4 8 9
The sequence first increases from 1 to 7, then decreases from 7 to 4 before increasing once again from 4 to 9.

Bitonic sort works in a similar way as merge sort. It divides a data set into smaller subsets, creating bitonic sequences that alternates between increasing and decreasing order. The end result is one whole bitonic sequence that are either monotonically increasing or decreasing, depending on how the data set wishes to be sorted[15] [16].

As the algorithm is dependent on how many splits are made for a specific number of elements, the time complexity for each case is $O(log(n)^2)$ in parallel time, seen in table 3.7.

Example

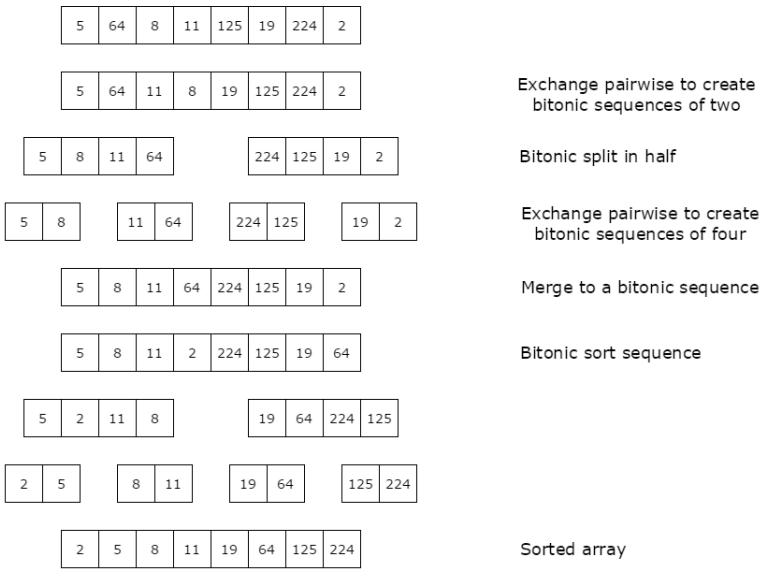


Figure 3.4: Bitonic sort - example.

According to the example in figure 3.4, bitonic sequences are created recursively. It works by breaking the data set down to smaller bitonic sequences that

are then merged together, forming longer bitonic sequences. This eventually results in a bitonic sequence that is monotonically increasing then decreasing. The final step is to create one long sequence that is monotonically increasing, which is equivalent to a sorted data set.

3.3.5 Bucket sort

Definition

Elements of a data set are distributed to several buckets based on part of the element's key. Each bucket is then sorted, before all buckets are merged together, keeping its relative order.

Time complexity	
Worst case	$O(n^2)$
Best case	$O(n+k)$
Average case	$O(n+k)$
Space complexity	
Worst case	$O(nk)$

Table 3.8: Bucket sort - time complexity.

Algorithm

Bucket sort is a stable distribution sorting algorithm, that can sort data in linear time. It is similar to radix sort, in that it works by dividing a data set into buckets, based on parts of the elements key. These buckets are then individually sorted, using either another sorting algorithm or recursively by bucket sort itself[10].

As seen in table 3.8, bucket sort has a time complexity for best and average case of $O(n+k)$, where k is the number of buckets. For worst case, the time complexity is dependent on the internal sorting algorithm for the buckets. In worst case, all elements would be placed in the same bucket. Using insertion sort for internal sorting, this would result in $O(n^2)$ in worst case. It also requires extra memory, $O(nk)$, in order to place elements in buckets while sorting.

Example

In the example of figure 3.5, buckets are created to store the elements of the data set. Each bucket is assigned a range, so that each bucket statistically could contain the same amount of elements.

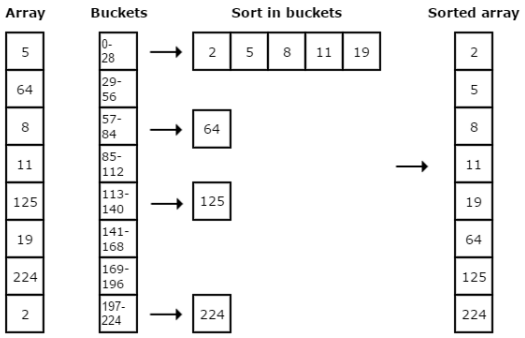


Figure 3.5: Bucket sort - example.

Elements of the data set is then distributed to its corresponding bucket. Each bucket sorts its content using an appropriate sorting algorithm, before extracting the elements in sorted order.

3.3.6 Radix sort

Definition

Distributes elements of a data set into buckets, based on the element's key, beginning with the least significant part of the key. After each pass, the elements are collected from the buckets, keeping the internal order, and then redistributed according to the next most significant part of the key.

Time complexity	
Worst case	$O(d(n+k))$
Best case	$O(d(n+k))$
Average case	$O(d(n+k))$
Space complexity	
Worst case	$O(n+k)$

Table 3.9: Radix sort - time complexity.

Algorithm

Radix sort is a stable distribution sorting algorithm, that can sort data in linear time. In mathematics, radix is the base of numeral system. For example, the decimal system have a radix of ten, as it include digits $0-9$. Radix sort works by placing elements into buckets based on parts of the key for the element, most

commonly starting with the least significant key. After each pass, the element are put back together again into a new set. The routine is then repeated, based on the following least significant key, until all possible keys been processed. For sorting the keys in each pass, an additional sorting algorithm is needed. Most common algorithms for this step are bucket sort or counting sort, both of which have ability to sort in linear time and are efficient on few digits. There are two different versions of radix sort. As previously mentioned, least significant key (LSD) sorts a data set using the least significant key as a starting point and moving upwards. In contrast, most significant key (MSD) start with the most significant key and moves downwards[17, 16].

Analysing the time complexity for radix sort is a bit complicated, as different assumptions can be made. But in general, the time complexity can be defined as $O(d(n + k))$, where d is the maximum number of digits and k is the base of numerical system, seen in table 3.9. It requires extra memory in order to place the elements in a temporary data set while sorting occurs for each pass.

Example

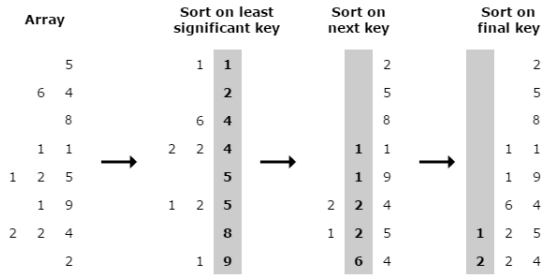


Figure 3.6: Radix sort - example.

In figure 3.6, the highest element is 224. Radix sort is bound to the number digits in the maximum value of the data set, in this case three. As result, radix sort goes through three steps in this instance.

In the first step, the data set is sorted based on the least significant digit. In the next step, sorting is based on the second least significant digit. The final step sorts the data set on the most significant digit. After all steps been completed, the result is a sorted data set.

3.4 Parallel computing

Most modern computers today consists of two or more cores within the CPU. This makes it possible to run several instructions simultaneously. The design of these multicore processor varies. A homogeneous multi-core system have cores

that are identical, while heterogeneous multi-core systems have cores that are not identical. Multi-core systems can share cache-memory or have individual memory. Some may implement message passing or have a shared memory intra-communication.

The performance of multi-core systems is still constricted to how algorithms utilizes the resources available. Without an efficient algorithm that take use of the extra resources a multi-core system provides, there will not be any substantial gain in performance.

There are many different aspects and characteristics of multi-core systems as further described below[18, 19, 20, 21, 22].

3.4.1 Workload balancing

A key aspect of an efficient parallel implementation is to distribute the workload evenly over the processors available. Consider an application where the workload is distributed in a 4:1 ratio between two processors. While the processor with the smaller part will finish its execution fast in relation to the second one, the overall execution time for the application will be dependant on processor with the greatest factor. Because of this, a parallel algorithm needs to ensure even balanced workload in order to maximize the performance.

3.4.2 Synchronization

To ensure that a parallel algorithm fulfils its intent, the processors needs to be synchronized. If not, two separate concurrent threads might access the same data and try to processes it, known as *race condition*. As a result, the data could end up corrupt or cause the application to either fail its purpose or provide false data. To avoid this, techniques like read- and write privileges, semaphores or mutual exclusion can be used.

3.4.3 Parallel slowdown

A phenomenon that can occur is parallel slowdown. In this, the parallelisation of an algorithm makes the execution of an application to run slower. This is most often related to the communication between processing threads. After a certain number threads have been created, in relation to the purpose of the application, threads will spend most of its time communicating with each other rather than processing data. With the creation of each processing thread, comes a cost in terms of an overhead. When the total cost of the overhead becomes a greater factor than the extra resources it provides, the parallel slowdown occurs.

3.4.4 Data movement

In distributed memory systems, data movement between processors can cause problems. The cost of moving data between processors can possibly become a greater factor than the execution time. Due to this, the task of moving data between processors should be kept to the minimum.

Chapter 4

Method

In this chapter, the chosen methodologies for this work are defined and explained as to how they were conducted.

4.1 Literature studies

During the initial literature studies, the area of parallel algorithms was researched and studied. The main focus was directed towards sorting algorithms, as it was chosen to be studied and experimented upon through the following experiment. From this, an understanding of how sorting algorithms works, what properties that makes for good or bad performance, and how to analyse them was gained, described in section 3.2. From this, a set of algorithms were identified, each with its own different properties and characteristics. With the main property of being highly able to utilize parallelisation, the set of algorithms chosen for the experiment was:

- Rank sort
- Merge sort
- Quicksort
- Bitonic sort
- Bucket sort
- Radix sort

The set of algorithms were chosen to get a variety of different designs, techniques, time complexities and characteristics of sorting algorithms.

Through the results from the experiments conducted, it would be expected to see the results differ pending on the specific characteristics for each algorithm. Both bucket sort and radix sort are distribution sorting algorithms, in contrast to the rest who could all be considered in some form to be comparison based algorithms. Rank sort has a time complexity of $O(n^2)$, while the distributions

algorithms in theory has a time complexity of $O(n)$. Bitonic sort is the only algorithm that by default can be considered as a parallel algorithm, while the rest by default are sequential algorithms.

In addition, during the literature studies, previous work in the area in regards to parallel execution and in comparisons to sequential algorithms, was studied to give an understanding of the differences between sequential and parallel execution and how such an experiment could be designed. The main interest lays in the potential speedup that can be achieved by parallelisation in relation to the additional resources used with subsequent costs.

It became apparent in this phase, that while there's been lot of work done in the chosen area, there was not a lot work done using Java as a platform. The original plan was to use optimized implementation of algorithms from previous work and use in the intended experiment. This plan had to be reworked due to the lack of available source code for Java. Due to this, all code had to be written and implemented for the experiment. As a result, no guarantees can be made for optimization of each algorithms.

4.2 Experiment

Java was chosen as the platform for this experiment. It is platform independent and has support for parallelisation with many different ways to approach and use multithreading.

In the experiment, all algorithms were executed to sort different data sets in ascending order. From this, data in form of execution times in nanoseconds, was gathered for each algorithm for both sequential and parallel execution. From this, a value representing the speedup achieved through multithreading was calculated to show difference between sequential and parallel execution.

All source code, including the test data and test results, can be found at `http://github.com/joupernordin/thesis` or is available by request to the authors.

4.2.1 Implementation

For the experiment, the ***Runnable*** interface of Java API[23] was used to implement all algorithms. Using this interface, an instance of a class can be executed by a thread. A specific ***Sorter*** class was created for this purpose for each algorithm. When running an algorithm in parallel, instances are created for each of the intended number of threads. The threads will then execute in parallel, with specific instructions for each algorithm. As a result, an overhead comes with the creation of each thread, including extra costs in comparison to executing in sequential.

Recursive algorithms

For merge sort, quicksort and bitonic sort, a recursive approach was used to implement the algorithms due their capacity of being divide-and-conquer algorithms. Using this technique, threads are created through the recursion, until the intended thread count have been reached. Merge sort and bitonic sort both have the property of self-balancing the workload between threads due to design of the algorithms that works by always dividing the data sets in halves. This way, threads will have approximately the same workload, with difference of one element if the data set is odd number in size. Quicksort however, provide no such guarantee. The workload of each thread is dependent on how the pivot element partition the data set. At best, there can be a similar balancing of workload as with merge sort. However, this is rarely the case.

For this experiment, a technique called median-of-three is used to pick a pivot element for quicksort. It works by deciding the median value of three elements in the data set: the first, the middle and the last one. This helps to mitigate the chance of hitting the worst case scenario when the min- or max element is picked as pivot.

For merge sort, [24] and bitonic sort, [25] was used as foundations for the implementation.

Iterative algorithms

For rank sort, bucket sort and radix sort, a linear approach was used to split the workload evenly between the available threads. This approach does not provide the same dynamic way of creating threads and dividing the workload. For rank sort, all threads work on the same data set, but are assigned a block of elements to sort.

For both bucket sort and radix sort, all elements are first divided into "buckets". These "buckets" are then divided evenly between the threads to be sorted independently. When all threads are finished with their execution, all elements are merged into a sorted order.

For radix sort, [26] was used as a foundation for implementation.

4.2.2 Test data

The data sets consisted of integers(4 bytes) that ranging from 2^4 to 2^{24} number of elements, with a total of seven test sets. Each set was created with three different structures:

- Random order - all integers in random order
- Increasing order - all integers sorted in increasing order
- Decreasing order - all integers sorted in decreasing order

Providing these three structures allows for testing specific algorithm traits, which can differ greatly depending on the data to be sorted. In addition, having a wide range between smallest and greatest number of elements allows for testing the relation between performance and efficiency to the number of threads for different sizes of data.

4.2.3 Test environment

The computer specification used for the experiment was:

CPU: Intel Core i5-4690K, 6MB cache, 4 cores, 4 threads
RAM: 12GB Dual-Channel DDR3 @ 666MHz (9-9-9-24)
OS: Windows 7 x64, installed on a SSD
Java: Version 1.8.0-40, x64

The algorithms was prioritized high within the operating system itself, such that the resources given from the operating system was utilized as much as possible.

4.2.4 Test execution and benchmark

For the test execution, all test were executed for different parallelism levels.

- Level 0 corresponds to sequential, where a single thread was used
- Level 1 corresponds to parallel execution with two threads
- Level 2 corresponds to parallel execution with four threads
- Level x corresponds to parallel execution with 2^x threads

Each parallelism level executed all data sets with subsequent structures. Thus the unique number of tests for each algorithm was

$$3\textit{(levels)} * 3\textit{(structures)} * 7\textit{(data sets, n elements)} = 63 \textit{ tests.}$$

A problem with benchmarking is to get correct and accurate results. The results could easily be affected in the case of an interruption by a process or other causes from the operating system. These external causes would impact the overall result of the execution time. In addition, for Java, there is an initialization phase when executing a program. This became noticeable when the first few results for each test case were always substantially slower than the following ones. This fluctuation led to misleading results in terms of the actual execution time. For this reason, it became necessary to run each test up to 100 times, with the first 10 test results being ignored as false messages. From this, an average result was

calculated, which in turns would represent execution time more accurately. The reasons for these fluctuations lies within the Java virtual machine, **JVM**, and the just-in-time compiler, **JIT**. The JVM switches from interpreted code to compiled code at will, and the same code path may already be compiled and recompile more than once during execution[27]. This puts big constraints in respect to getting results that are valid and reasonable, which in turn is one of the major reasons for executing tests more than once. However, since the focus of this work is not in benchmarking, the implemented test execution phase could be improved upon to get even more accurate results.

4.2.5 Parallel experimentation

With parallelisation of algorithms, comes new challenges and problems that needs to be addressed and carefully considered, mentioned in section 3.4.

To investigate some of these inherent problems, experiments were carried out to test and analyse what impact they have on the performance of execution.

Unbalanced workload

To test the impact of unbalanced workload between threads, quicksort was executed on a data set of size 2^{20} using two threads. The workload was divided in different ratios between the threads and executed.

Thread threshold

This test was designed to test if an eventual threshold exists for the number of threads in which parallel execution provides an increase in efficiency. For this purpose, merge sort was used. The same data sets was used as in previous experiments, running on different number of threads, ranging from 2^1 to 2^{10}, using sequential execution as baseline.

4.3 Analysis

From the results of the experiment, an analysis can made in regards to the execution times and parallel speedup. Parallel speedup is calculated by the following equation:

$$\frac{sequential\ time}{parallel\ time} = speedup\ x \tag{4.1}$$

The value of the potential speedup, as described in equation 4.1, refers to how much faster the parallel time is in relation to the sequential time. A speedup of

1.5 results in one and a half times faster execution, while a speedup of 0.5 is only half as fast as the baseline.

In addition, the source code of each algorithm will be analysed to mathematically prove that the algorithms does indeed perform at their expected time complexity. The analysis of algorithms and its time complexity was executed as following. The selected algorithms was converted from code to mathematical statements for each operation and how many times that operation is executed. In table 4.1, a representation of insertion sort is illustrated, and how it could be converted to mathematical statements using one thread, sequential execution [10].

Line	insertionSort(A)	Cost	Times
1	*for j = 2 to A.length*	c_1	n
2	*key = [j]*	c_2	$n-1$
3	*//comment*	0	$n-1$
4	*i = j - 1*	c_4	$n-1$
5	*while i > 0 and A[i] > key*	c_5	$\sum_{j=2}^{n} tj$
6	*A[i+1] = A[i]*	c_6	$\sum_{j=2}^{n}(tj-1)$
7	*i = i - 1*	c_7	$\sum_{j=2}^{n}(tj-1)$
8	*A[i + 1] = key*	c_8	$n-1$

Table 4.1: Example analysis - insertion sort

The mathematical statements are converted to equations for the given algorithm and its functions. The equations are formulated so the through analysis, the asymptotic boundaries can be found. The following format is used for every analysis.

$$\begin{cases} T(n) & = & equation \\ T(1) & = & constant \end{cases} \tag{4.2}$$

Where n is the number of elements such that $T(n) = equation$ for the given algorithm.

From table 4.1, an equation for insertion sort would be expressed with the following statements [10].

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) c_8(n-1) \tag{4.3}$$

From the equation, best case and worst case can be calculated. Using Θ, constants can be abstracted such that the determining growth factor of a single statement could be used in the overall equation.

The worst case of insertion sort would be expressed as a refined equation as follows [10].

$$\begin{aligned} T(n) &= (\frac{c_5}{2} + \frac{c_6}{2}) + \frac{c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8)n + (c_{2+}c_4 + c_5 + c_8) \\ &= an^2 + bn - c, \ for \ some \ constants \ a, b, c \Rightarrow O(n^2) \end{aligned} \tag{4.4}$$

Using the equation above, before abstraction through time complexity notations occurs, a new equation can be defined for p threads. This equation includes the constants times which will depend on the number of threads used due to a possible extra cost of overhead, but also a mitigation of overall execution time. Through this, a more interesting comparison can be made between the sequential and parallel version of the algorithm since several factors can be taken into consideration, and not just a comparison between the same growth function.

$$T(n) = (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + \Theta(n) - contstants \tag{4.5}$$

However, since insertion sort is not suitable for parallelisation, an assumption is made in this explanation as to how its parallel equation could look like.

$$T(n) = \frac{(Constants)n^2}{p} + \Theta(n) - constants \tag{4.6}$$

Chapter 5

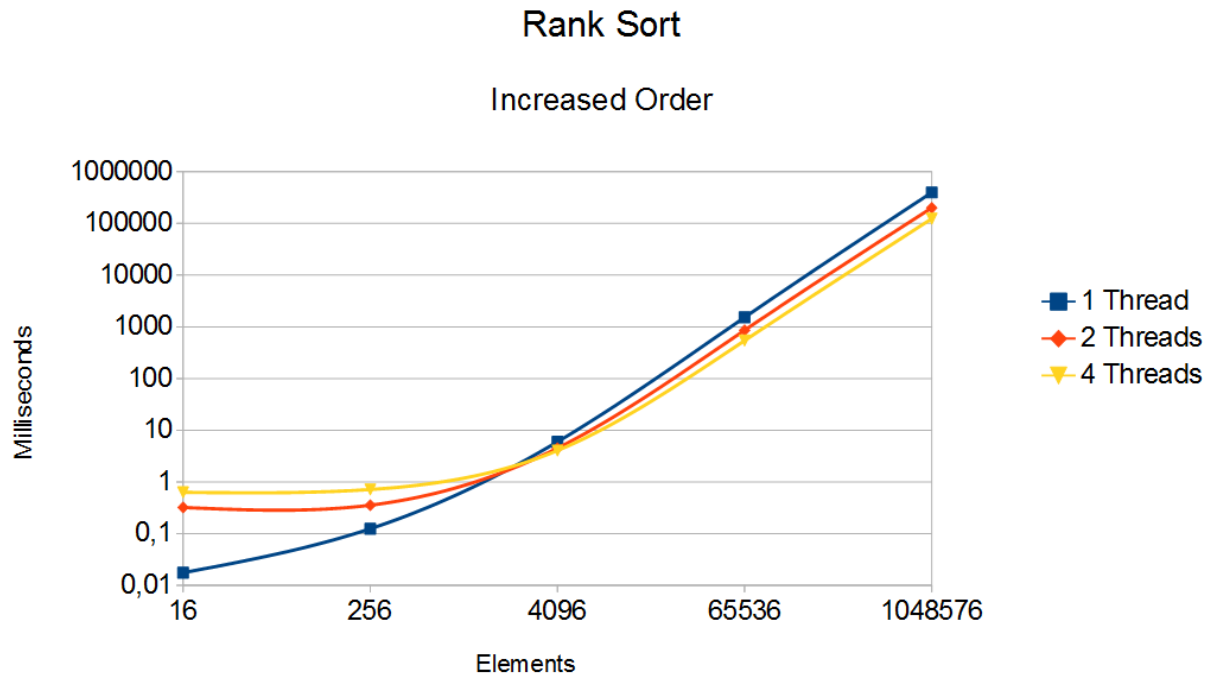
Results

In this chapter, the results from the experiment are illustrated in form of graphs and tables.

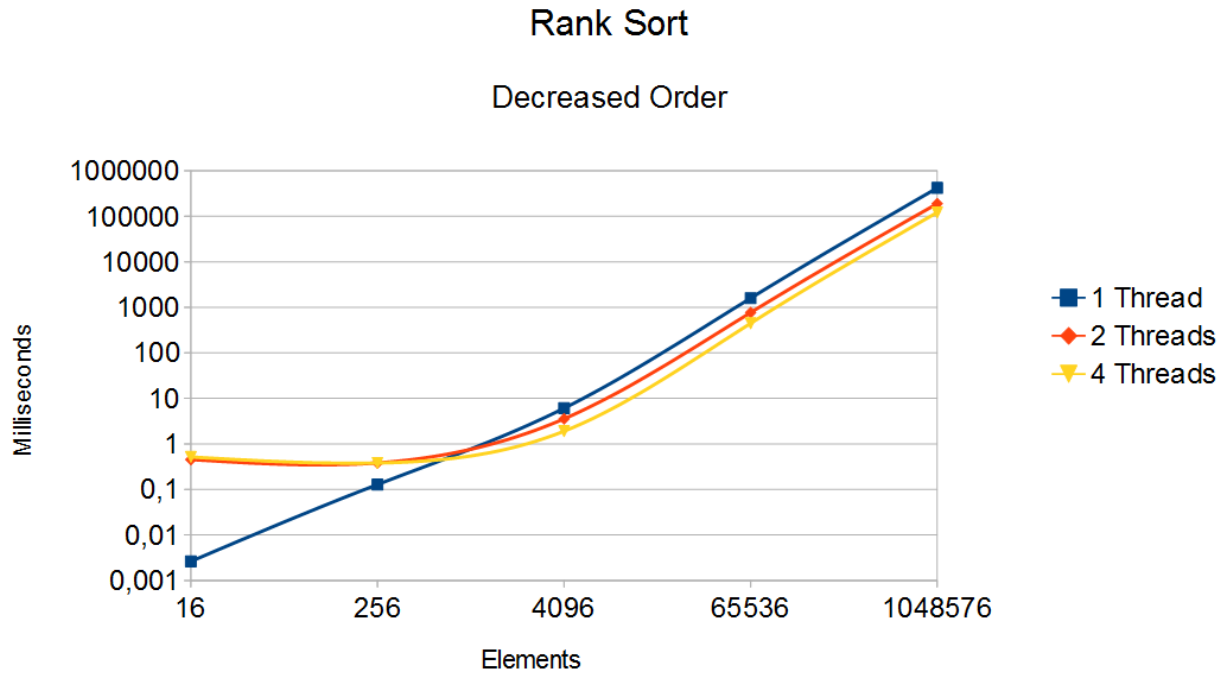
5.1 Rank sort

Following graphs 5.1 5.2 and table 5.1 shows the average execution times for rank sort on different data sets, where T represents the number of threads used in execution. Data sets larger than 2^{20} was opted not to execute due to extensive execution times. The data set containing 2^{20} took over 40 minutes to execute, and the expected time for larger data sets would go far beyond this.

In addition, it was opted not to run the test case for the data set of size 2^{20} up to a 100 times. This due to the actual execution time being such a great factor, with a sequential time over over 40 minutes. The benefit of multiple executions was therefore considered not to be of importance or beneficial in relation to the rest of the work.



(a) increasing order.



(b) decreasing order

Figure 5.1: Rank sort - sorted order.

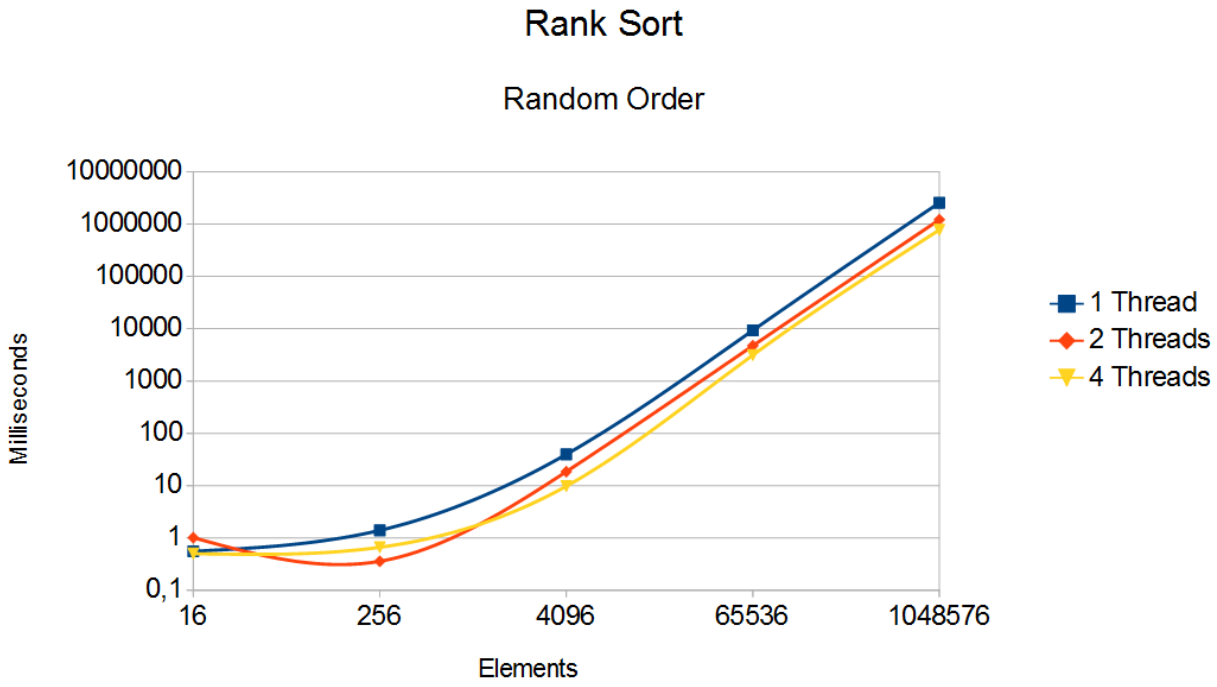


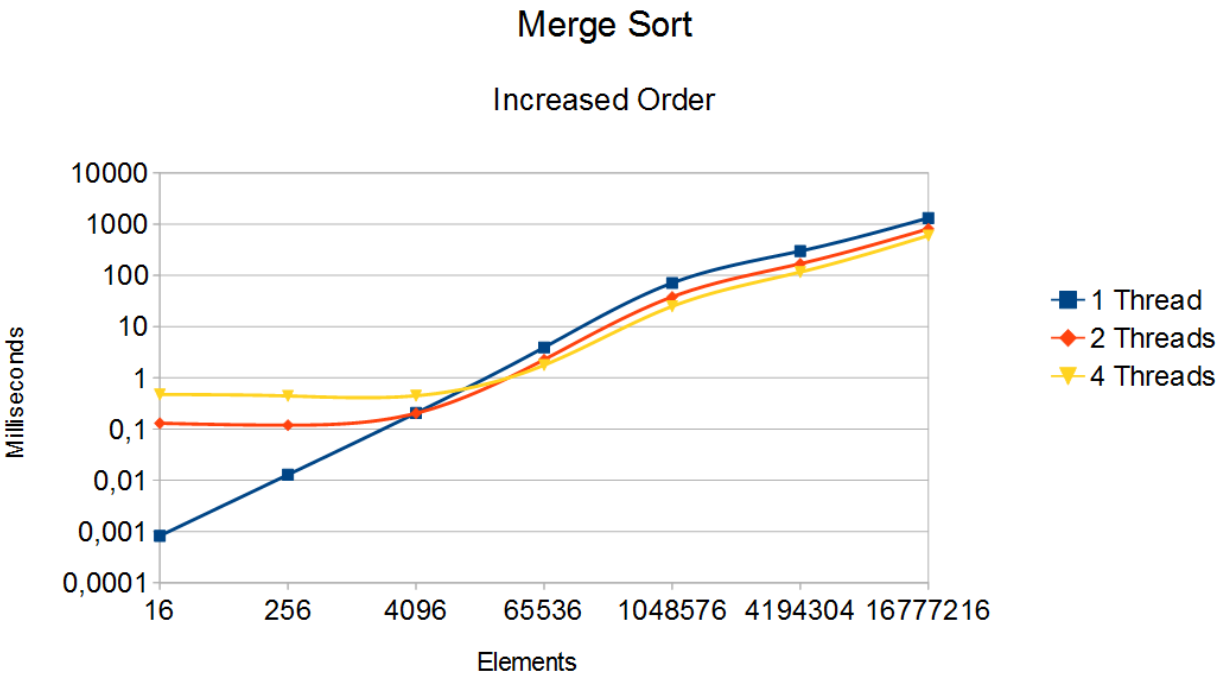
Figure 5.2: Rank sort - random order.

Rank sort: execution time (ms)									
Data	Random order			Increase order			Decrease order		
	T=1	T=2	T=4	T=1	T=2	T=4	T=1	T=2	T=4
2^4	0.6	1.0	0.5	0.02	0.3	0.6	0.003	0.5	0.5
2^8	1.4	0.4	0.7	0.1	0.4	0.7	0.1	0.4	0.4
2^{12}	40	18	9.8	6.0	4.5	4.0	6.1	3.6	1.9
2^{16}	9294	4722	3126	1524	859	542	1590	767	446
2^{20}	2563758	1214479	774736	397215	200546	123474	418925	189630	121173

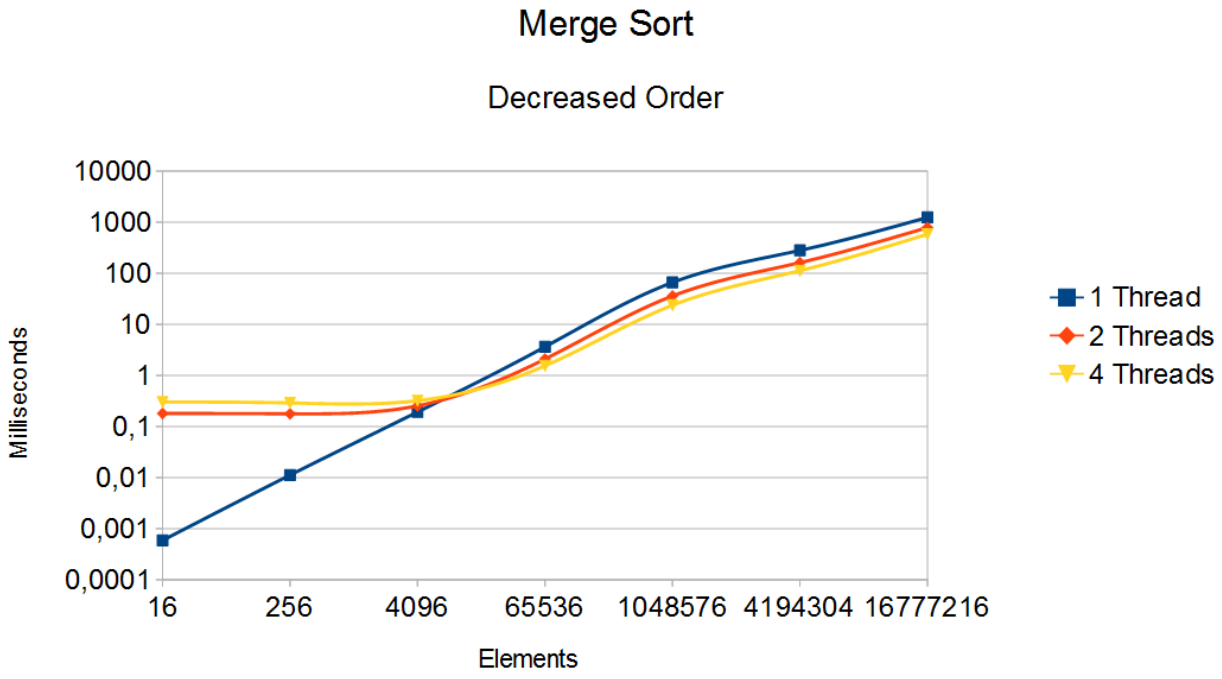
Table 5.1: Rank sort - execution times.

5.2 Merge sort

Following graphs 5.3 5.4 and table 5.2 shows the average execution times for merge sort on different data sets, where T represents the number of threads used in execution.



(a) increasing order.



(b) decreasing order.

Figure 5.3: Merge sort - sorted order.

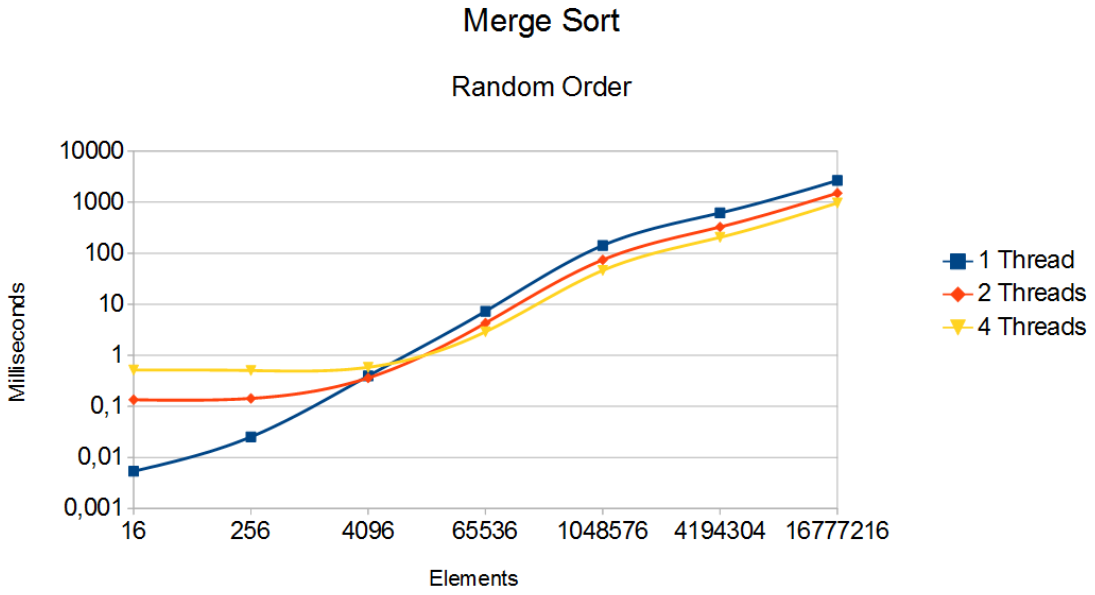


Figure 5.4: Merge sort - random order.

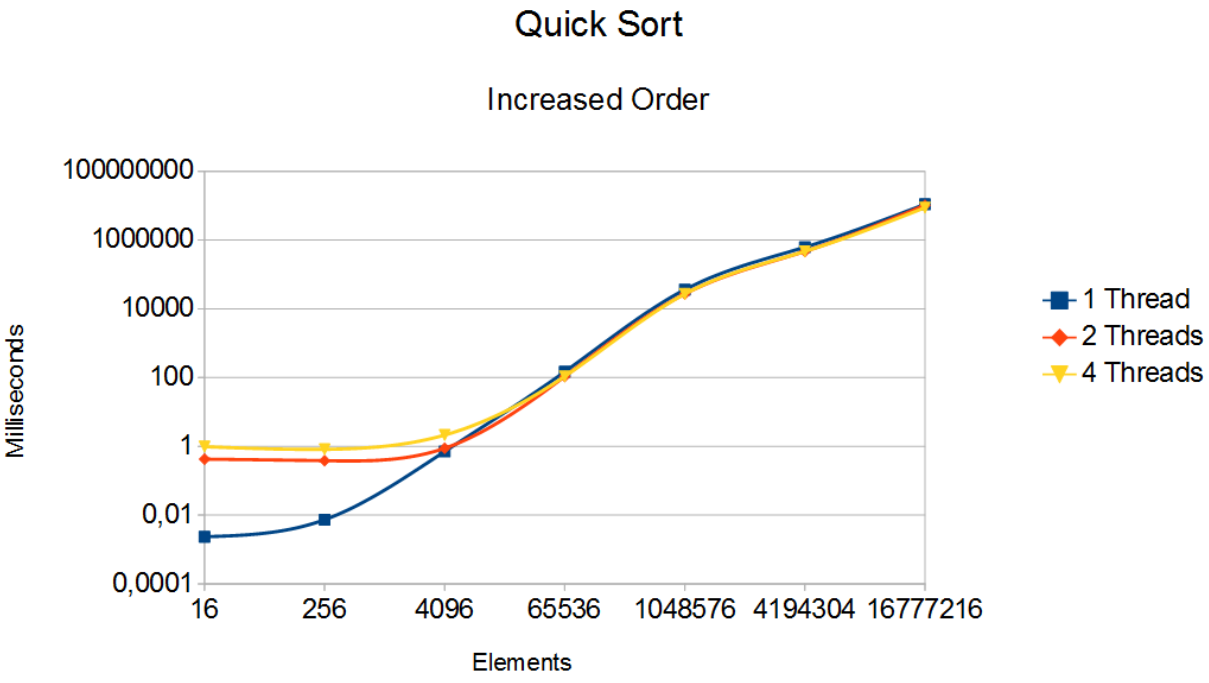
Merge sort: execution time (ms)									
Data	Random order			Increase order			Decrease order		
	T=1	T=2	T=4	T=1	T=2	T=4	T=1	T=2	T=4
2^4	0.005	0.1	0.5	0.0008	0.1	0.5	0.0006	0.2	0.3
2^8	0.03	0.1	0.5	0.01	0.1	0.4	0.01	0.2	0.3
2^{12}	0.4	0.4	0.6	0.2	0.2	0.4	0.2	0.2	0.3
2^{16}	7.3	4.3	2.9	3.9	2.3	1.8	3.6	2.1	1.5
2^{20}	142	74	46	71	38	25	66	36	24
2^{22}	616	327	205	299	169	116	280	162	112
2^{24}	2657	1508	965	1320	812	599	1239	786	581

Table 5.2: Merge sort - execution times.

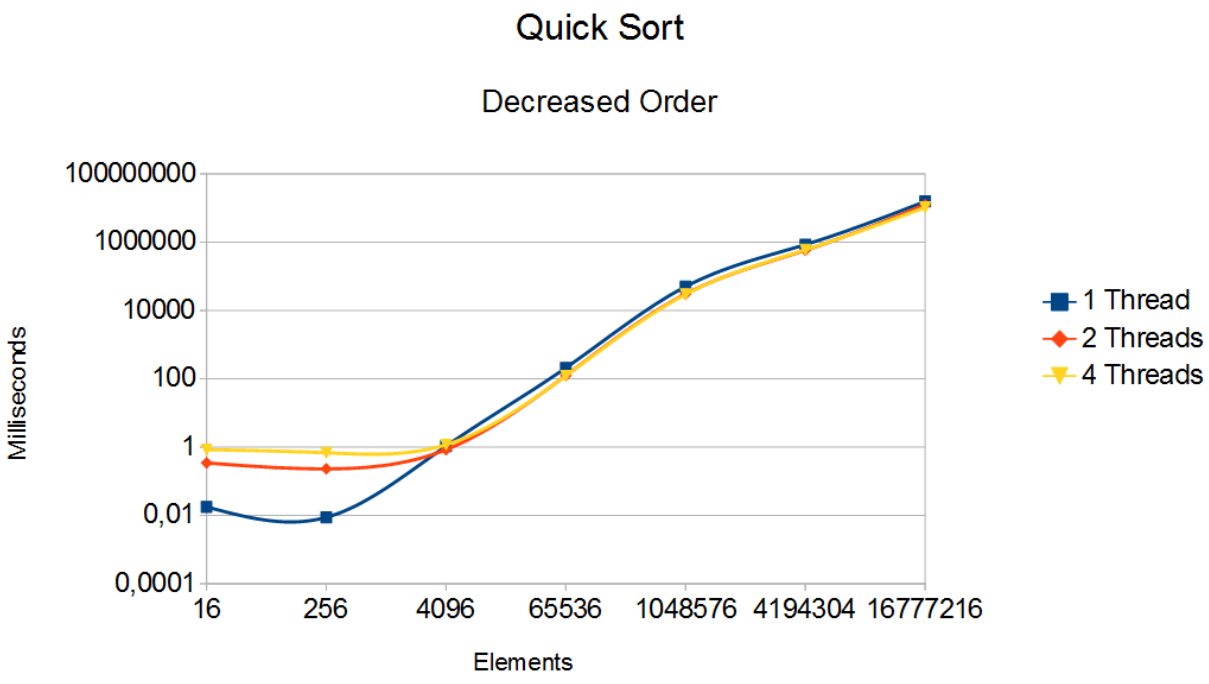
5.3 Quicksort

Following graphs 5.5 5.6 and table 5.3 shows the average execution times for quicksort on different data sets, where T represents the number of threads used in execution. For data set of 2^{24}, there is a dramatic decrease in performance. This is due the characteristics of quicksort, which relies on the pivot value to partition the workload between threads. The random data in this set, is not favourable for the quicksort algorithm as the initial pivot value partitions the data unevenly between threads, resulting in less increase in performance in relation to previous data sets.

In addition, it was opted not to run the test cases for the bigger data sets of increased and decreased order up to a 100 times. This due to the actual execution time being such a great factor, which in some cases would go over three hours for a single test case. The benefit of multiple executions was therefore considered not to be of importance or beneficial in relation to the rest of the work.



(a) increasing order.



(b) decreasing order.

Figure 5.5: Quicksort - sorted order.

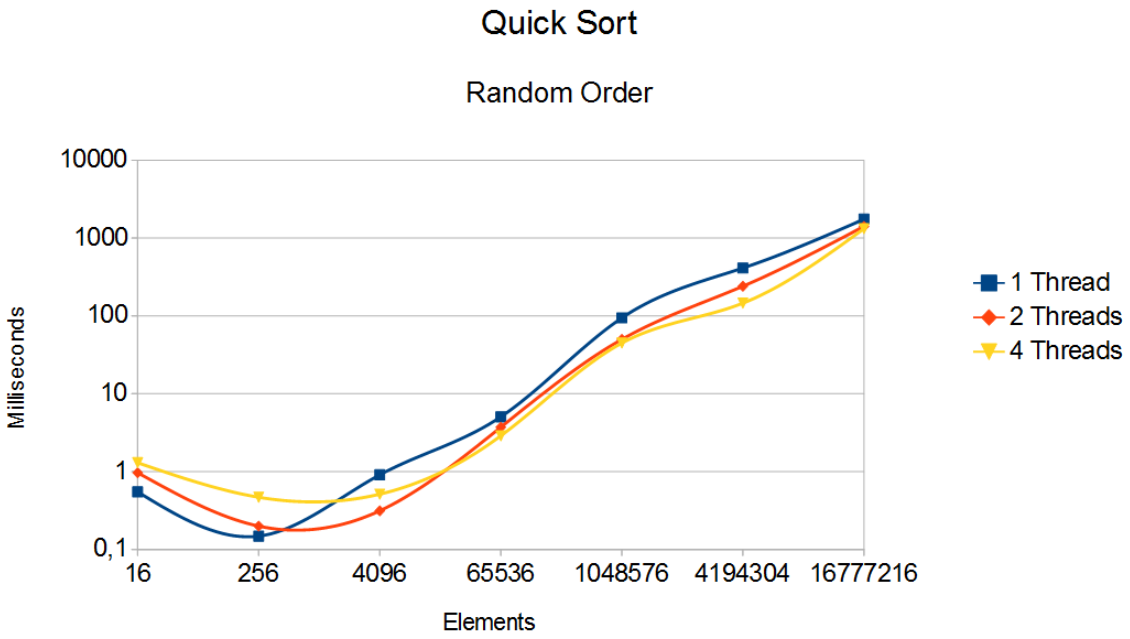


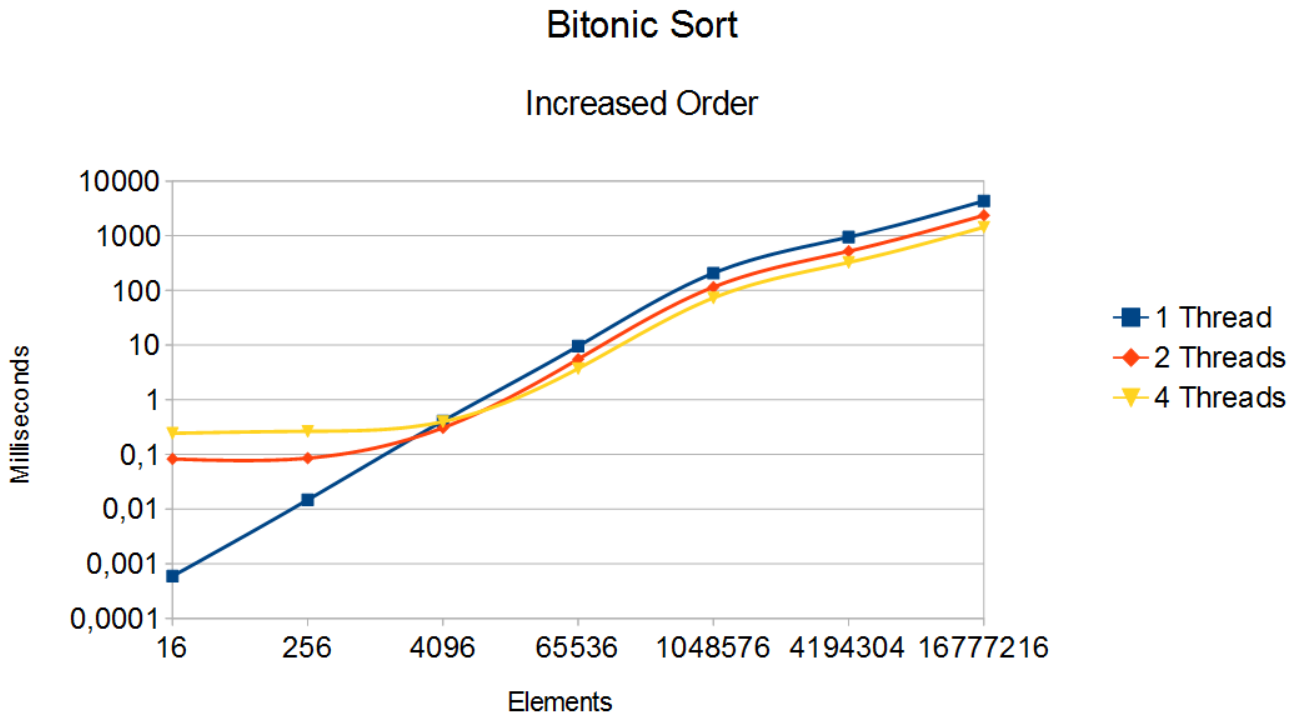
Figure 5.6: Quicksort - random order.

Quick sort: execution time (ms)									
Data	Random order			Increase order			Decrease order		
	T=1	T=2	T=4	T=1	T=2	T=4	T=1	T=2	T=4
2^4	0.5	1.0	1.3	0.002	0.4	1.0	0.02	0.3	0.8
2^8	0.1	0.2	0.5	0.007	0.4	0.8	0.009	0.2	0.7
2^{12}	0.9	0.3	0.5	0.7	0.9	2.1	1.0	0.8	1.2
2^{16}	5.0	3.7	2.9	148	108	108	210	125	122
2^{20}	94	50	45	36207	27303	27082	50252	30586	29698
2^{22}	412	240	146	624855	470400	471070	855164	581462	599282
2^{24}	1745	1412	1317	11019911	10312817	8800218	15628133	12550169	10457832

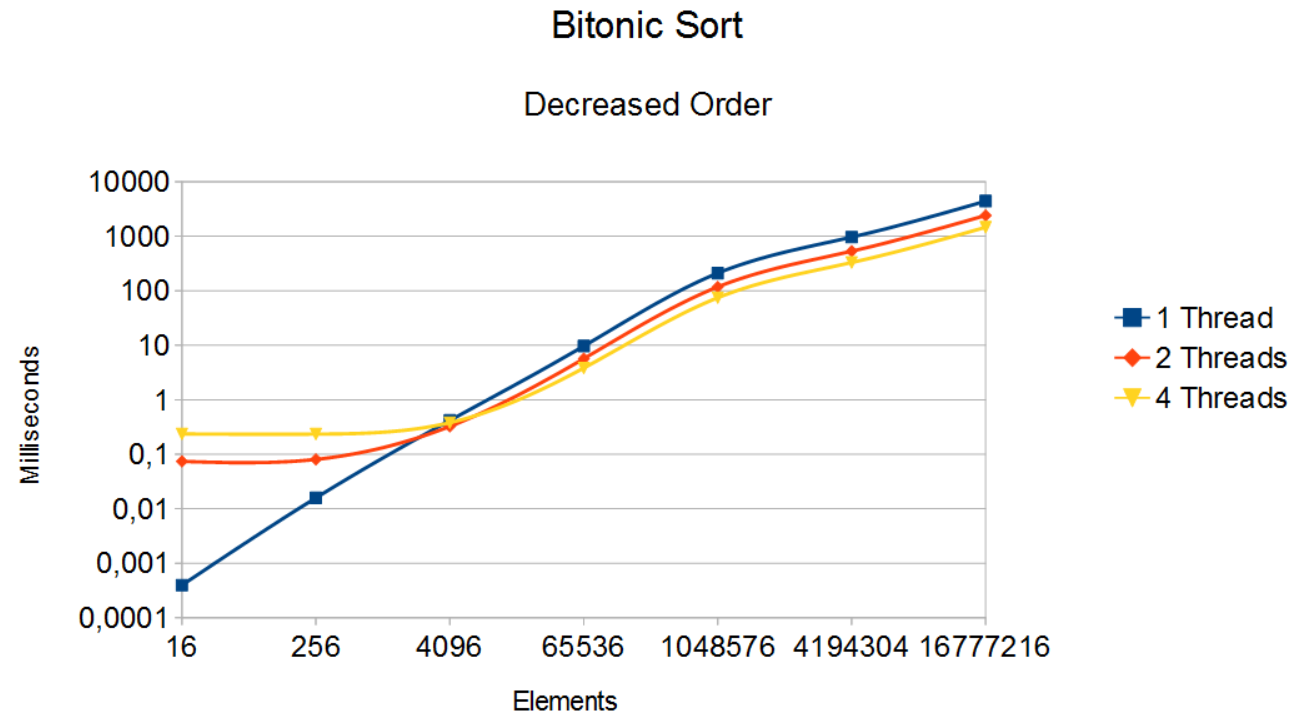
Table 5.3: Quicksort - execution times.

5.4 Bitonic sort

Following graphs 5.7 5.8 and table 5.4 shows the average execution times for bitonic sort on different data sets, where T represents the number of threads used in execution.



(a) increasing order.



(b) decreasing order.

Figure 5.7: Bitonic sort - sorted order.

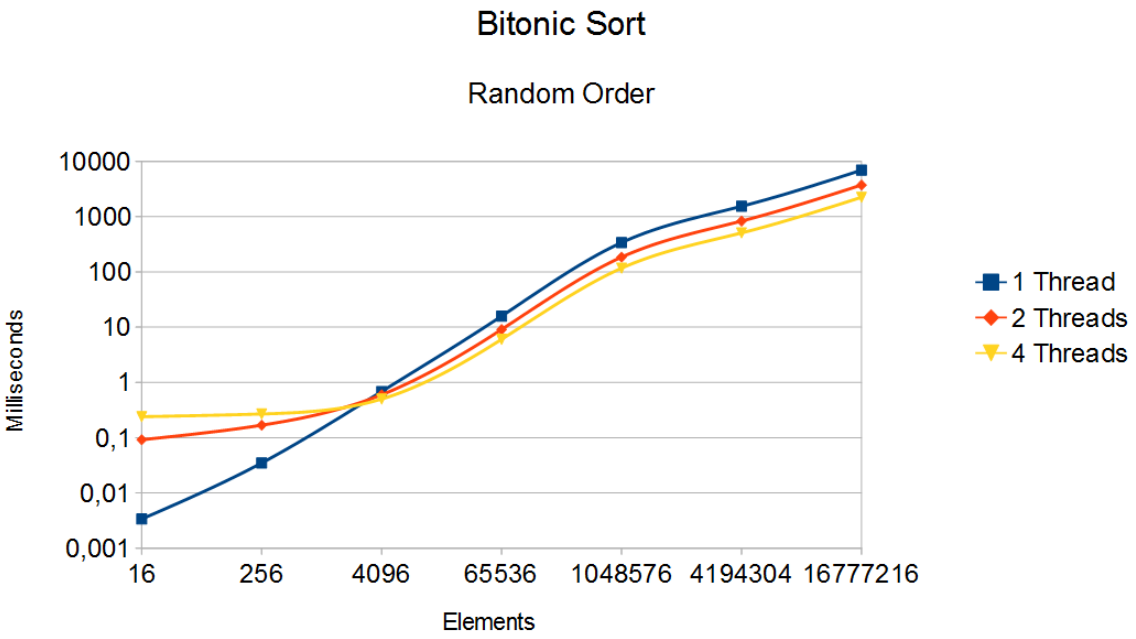


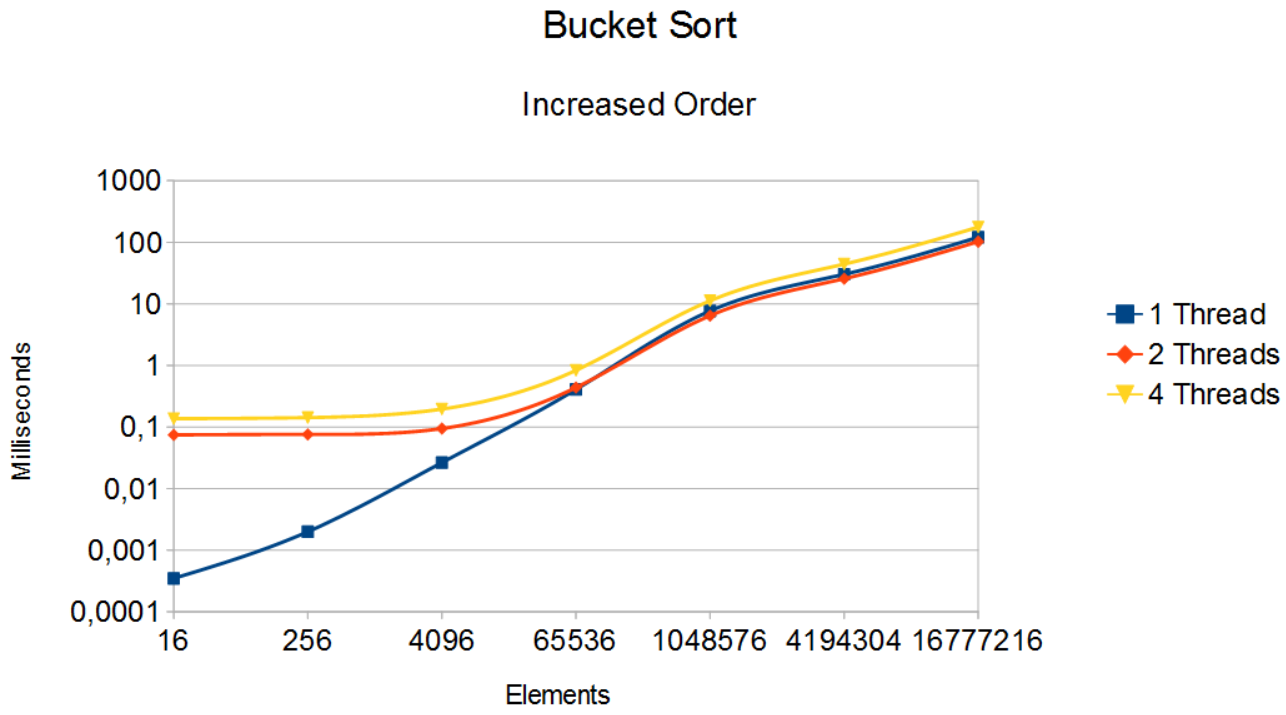
Figure 5.8: Bitonic sort - random order.

Bitonic sort: execution time (ms)									
Data	Random order			Increase order			Decrease order		
	T=1	T=2	T=4	T=1	T=2	T=4	T=1	T=2	T=4
2^4	0.004	0.1	0.2	0.0006	0.1	0.2	0.0004	0.1	0.2
2^8	0.03	0.2	0.3	0.01	0.1	0.3	0.02	0.1	0.2
2^{12}	0.7	0.6	0.5	0.4	0.3	0.4	0.4	0.3	0.4
2^{16}	16	9.1	6.0	10	5.5	3.7	9.6	5.7	3.8
2^{20}	339	185	117	207	115	73	210	117	74
2^{22}	1539	831	508	946	521	325	963	531	329
2^{24}	6940	3735	2239	4332	2363	1436	4406	2406	1458

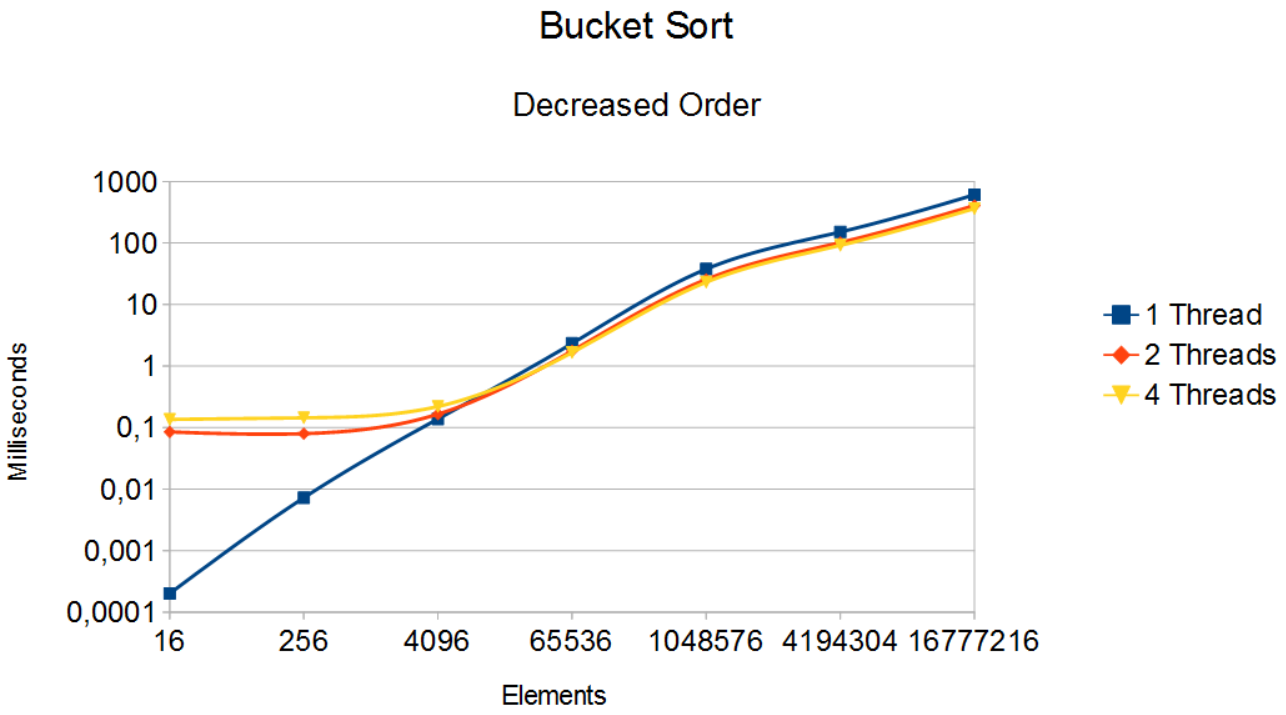
Table 5.4: Bitonic sort - execution times.

5.5 Bucket sort

Following graphs 5.9 5.10 and table 5.5 shows the average execution times for bucket sort on different data sets, where T represents the number of threads used in execution.



(a) increasing order.



(b) decreasing order.

Figure 5.9: Bucket sort - sorted order.

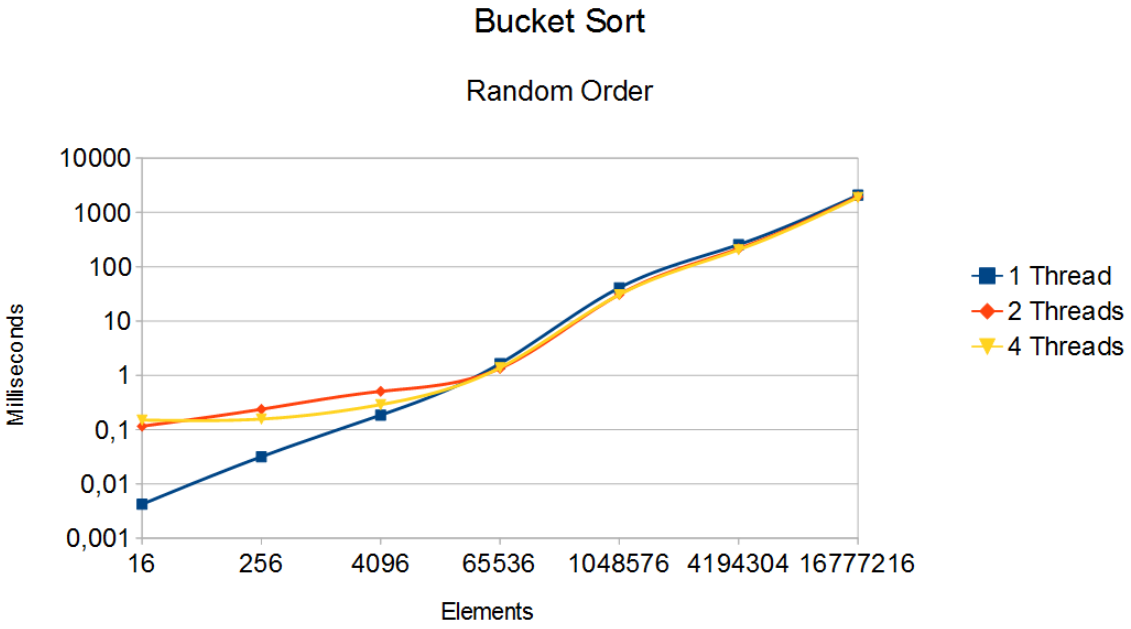


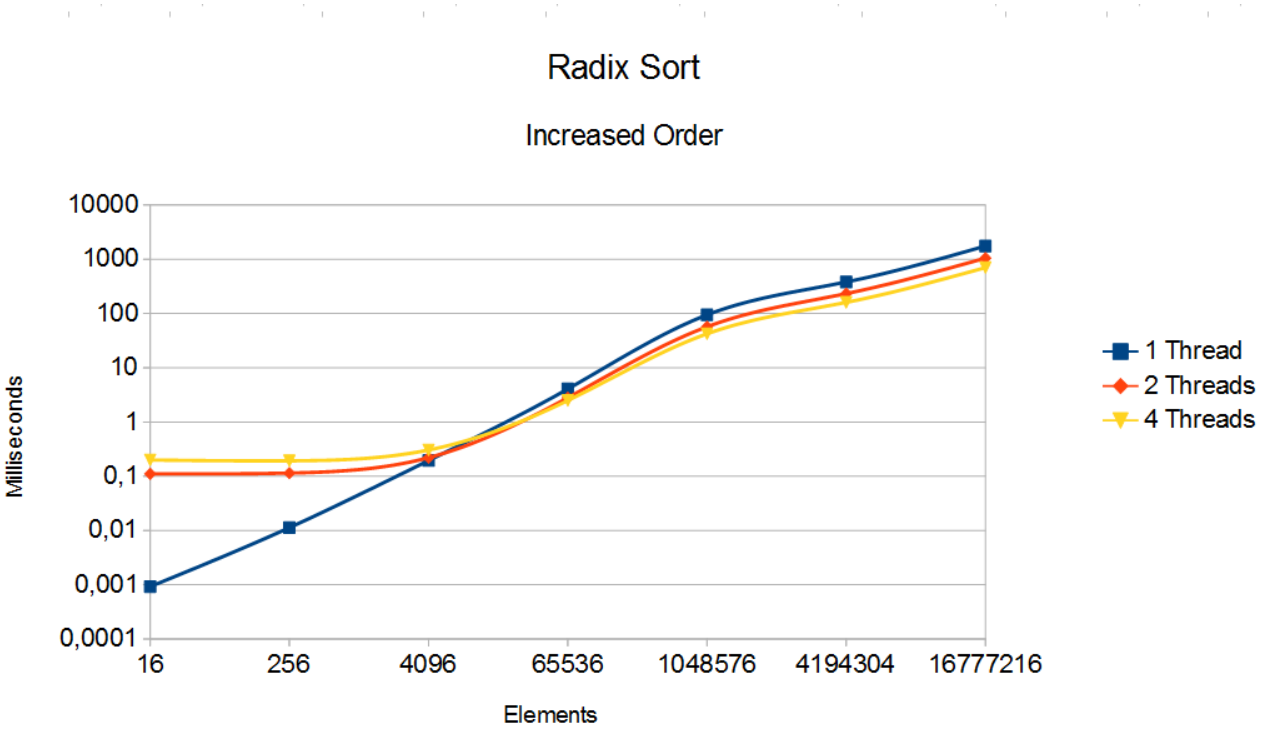
Figure 5.10: Bucket sort - random order.

Bucket sort: execution time (ms)									
Data	Random order			Increase order			Decrease order		
	T=1	T=2	T=4	T=1	T=2	T=4	T=1	T=2	T=4
2^4	0.004	0.1	0.1	0.0003	0.07	0.1	0.0002	0.08	0.1
2^8	0.03	0.2	0.2	0.002	0.08	0.1	0.007	0.08	0.1
2^{12}	0.2	0.5	0.3	0.02	0.09	0.2	0.1	0.2	0.2
2^{16}	1.6	1.3	1.4	0.4	0.4	0.8	2.3	1.8	1.7
2^{20}	41	31	31	7.7	6.4	11.2	38	26	23
2^{22}	255	215	205	30	26	44	151	102	91
2^{24}	2076	1948	1890	121	102	177	609	406	364

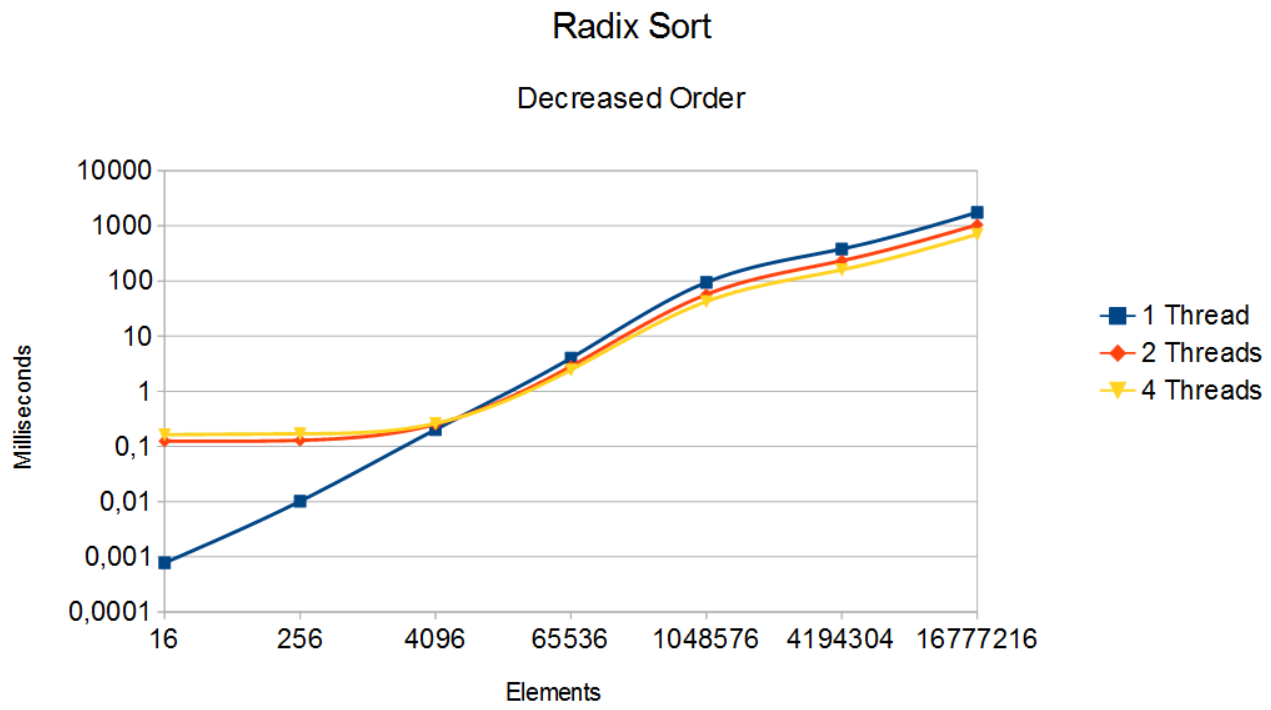
Table 5.5: Bucket sort - execution times.

5.6 Radix sort

Following graphs 5.11 5.12 and table 5.6 shows the average execution times for radix sort on different data sets, where T represents the number of threads used in execution.



(a) increasing order.



(b) decreasing order.

Figure 5.11: Radix sort - sorted order.

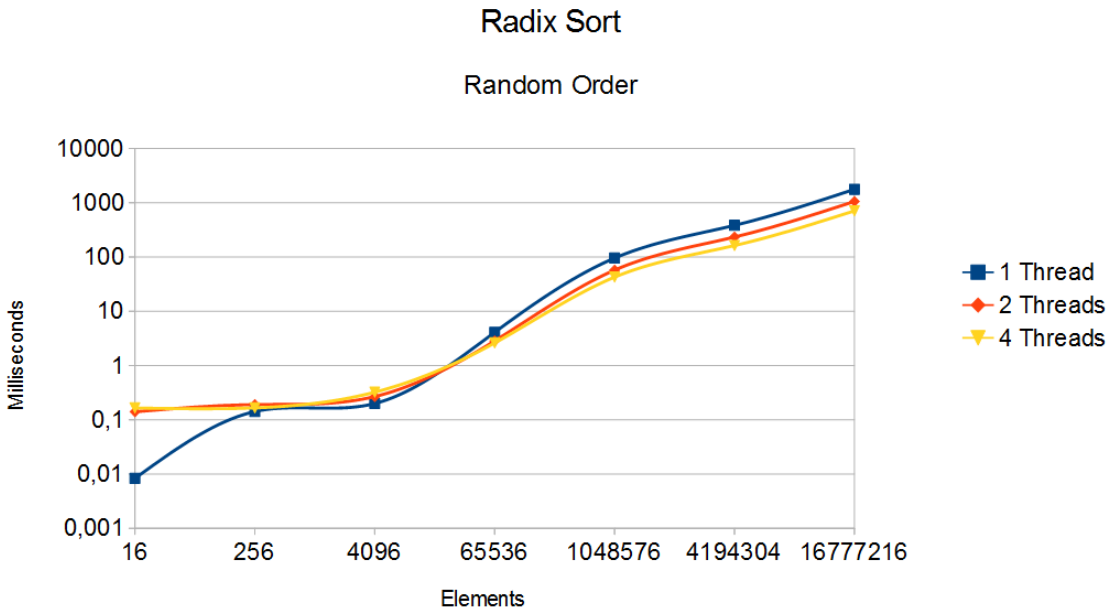


Figure 5.12: Radix sort - random order.

Radix sort: execution time (ms)									
Data	Random order			Increase order			Decrease order		
	T=1	T=2	T=4	T=1	T=2	T=4	T=1	T=2	T=4
2^4	0.008	0.1	0.2	0.0009	0.1	0.2	0.0008	0.1	0.2
2^8	0.1	0.2	0.2	0.01	0.1	0.2	0.01	0.1	0.2
2^{12}	0.2	0.3	0.3	0.2	0.2	0.3	0.2	0.3	0.3
2^{16}	4.1	2.8	2.6	4.1	2.8	2.5	4.0	2.8	2.4
2^{20}	95	57	42	94	57	42	94	57	42
2^{22}	382	233	162	381	233	160	382	233	160
2^{24}	1763	1053	707	1756	1049	702	1757	1047	704

Table 5.6: Radix sort - execution times.

Chapter 6

Analysis

In this chapter, the mathematical analysis of the algorithms and the analysis of the results from the experiments are presented.

6.1 Mathematical analysis

In the mathematical analysis, all the implemented algorithms have been converted from source code to mathematical statements. All statements reflects the cost for an operation and the number of times that operation is executed within a function call.

Function calls are denoted by T_x, which is the set of operation done within the specific function.

Constants are denoted by c_x, which defines a single operation that takes constant time to execute. Constants time are not defined with real execution time values, since each constant might differ depending on the used compiler and the time it take to execute the operation. Threads are denoted by p which defines a set of threads used.

The full translation for all mathematical statements and sets can be found in the appendix, where each algorithms are defined within a table with its corresponding code.

6.1.1 Rank sort

The following sets denotes the functions calls used by rank sort which are used to analyse and prove all cases for time complexity with its constants. Definition of T sets can be found in appendix A. Rank sort.

$T_1 = c_7 + c_{11} + p(c_8(T_2) + c_9 + c_{10})$
$T_2 = c_{21} + c_{22} + c_{23} + c_{24} + c_{25}$
$T_3 = c_{15}n(c_{16} + c_{17}n(c_{18} + c_{19}) + c_{20})/p$
$T_4 = c_1 + c_2 + c_3 + c_4(T_1) + T_3$

Each of the functions calls above defines a sub-process for sorting n elements. The full equation for sorting is denoted as following

$$\begin{aligned} T_n &= c_1 + c_2 + c_3 + c_4(c_7 + c_{11}p(c_8(T_2) + c_9 + c_{10})) \\ &\quad + c_{15}n(c_{16} + c_{17}n(c_{18} + c_{19}) + c_{20})/p \end{aligned} \tag{6.1}$$

From the equation above, sets of constants are extracted and defined as following

$$\begin{aligned} constants_1 &= c_1 + c_2 + c_3 \\ constants_2 &= c_7 + c_{11} \\ constants_3 &= c_9 + c_{10} \\ constants_4 &= c_{16} + c_{20} \\ constants_5 &= c_{18} + c_{19} \\ overhead &= T_2 = c_{21} + c_{22} + c_{23} + c_{24} + c_{25} \end{aligned}$$

Inserting the new sets of constants gives the following expression

$$\begin{aligned} T_n &= constants_1 + c_4(constants_2 + p(c_8(overhead) + constants_3)) \\ &\quad + c_{15}n(constants_4 + c_{17}n(constants_5))/p \\ &= constants_1 + c_4(constants_2 + p(c_8(overhead) + constants_3)) + O(n^2)/p \\ &= \Theta(n^2) \end{aligned} \tag{6.2}$$

The analysis shows that the expected time complexity of the implemented rank sort has a growth factor of $\Theta(n^2)$. Since rank sort executes the same way for all cases, best, worst and average cases can be defined as following:

$O(n^2)$ as upper bound, $\Omega(n^2)$ as lower bound, $\Theta(n^2)$ as tight bound

Since both the best and worst cases bounds are the same, the asymptotically tight bound for the average case will be the same.

A general equation for the implemented rank sort with p threads, where $p \geq 2^1$ can be concluded as

$$T_n = constants_1 + c_4(constants_2 + p(c_8(overhead) + constants_3)) + O(n^2)/p \tag{6.3}$$

And an equation for sequential rank sort can be concluded as

$$T_n = constants_1 + O(n^2) \tag{6.4}$$

6.1.2 Merge sort

The following sets denotes the functions calls used by merge sort which are used to analyse and prove the time complexity with its constants. Definition of T sets

can be found in appendix B. Merge sort.

$$T_1 = c_{14} + c_{15} + c_{16} + c_{17} + c_{18} + c_{19}$$

$$T_2 = c_{20} + c_{21} + c_{22}\sum_{i=1}^{n/2} t_i + c_{23}\sum_{i=1}^{n/2} t_i + c_{24} = c_{20} + c_{21} + (c_{22} + c_{23})n + c_{24}$$

$$T_3 = c_{25} + c_{26} + c_{27} + c_{28}\sum_{i=1}^{n/2} t_i + c_{29}\sum_{i=1}^{n/2} t_i + c_{30} = c_{25} + c_{26} + c_{27} + (c_{28} + c_{29})n + c_{30}$$

$$T_4 = c_{39} + c_{40} + c_{41}$$

$$\begin{aligned} T_5 &= c_{31} + c_{32} + c_{33}\sum_{i=1}^{n} t_i + c_{34}\sum_{i=1}^{n} t_i + c_{35}\sum_{i=1}^{n} t_i + c_{36}\sum_{i=1}^{n} t_i + c_{37}\sum_{i=1}^{n} t_i + c_{38}\sum_{i=1}^{n} t_i \\ &= c_{31} + c_{32} + (c_{33} + c_{34} + c_{35} + c_{36} + c_{37} + c_{38})n \end{aligned}$$

Each of the function calls above defines a sub-process for sorting n elements. The full equation for the sorting process is denoted as following

$$\begin{aligned} T(n) = {} & c_1(p+1) + c_2(p) + c_3(p+1) + c_4(p * T_1) + c_5(\frac{p}{2} * T_2) + c_6(\frac{p}{2} * T_3) \\ & + c_7(\frac{p}{2} * T_4) + c_8(\frac{p}{2} * T_4) + c_9(\frac{p}{2}) + c_{10}(\frac{p}{2}) + c_{11}(\frac{p}{2}) + c_{12}(\frac{p}{2}) + c_{13}(\frac{p}{2} * T_5) \end{aligned} \tag{6.5}$$

Suppose $p = 2$, which denotes that two threads are used, thus the sorting process above contains all function calls and the number of threads used, will define the equation as following

$$\begin{aligned} T(n) = {} & 3c_1 + 2c_2 + 3c_3 + c_4(2 * (c_{14} + c_{15} + c_{16} + c_{17} + c_{18} + c_{19})) \\ & + c_5(c_{20} + c_{21} + (c_{22} + c_{23})n + c_{24}) + c_6(c_{25} + c_{26} + c_{27} + (c_{28} + c_{29})n + c_{30}) \\ & + c_7(c_{39} + c_{40} + c_{41}) + c_8(c_{39} + c_{40} + c_{41}) + c_9 + c_{10} + c_{11} + c_{12} \\ & + c_{13}(c_{31} + c_{32} + (c_{33} + (c_{34} + c_{35} + (c_{36} + c_{37} + (c_{38})n)) \end{aligned} \tag{6.6}$$

The equation above shows every operation done for the merge sort implementation. However, since merge sort is executed through recursion, each recursive call breaks down the given problem into two subproblems containing $\frac{n}{2}$ elements. Once the downward recursion is done, all subproblems will be merged back together, represented by a constant factor n.

Thus, the recursive formula for merge sort is defined as $2T(\frac{n}{2}) + cn$ for a

constant c.

$$\begin{aligned}T(n) &= 3c_1 + 2c_2 + 3c_3 + c_4(2 * (c_{14} + c_{15} + c_{16} + c_{17} + c_{18} + c_{19})) \\ &+ c_5c_6 2T\left(\frac{c_{20} + c_{21} + (c_{22} + c_{23})n + c_{24}}{2} + \frac{c_{25} + c_{26} + c_{27}(c_{28} + c_{29})n + c_{30}}{2}\right) \\ &+ c_7(c_{39} + c_{40} + c_{41}) + c_8(c_{39} + c_{40} + c_{41}) + c_9 + c_{10} + c_{11} + c_{12} \\ &+ c_{13}(c_{31} + c_{32} + (c_{33} + c_{34} + c_{35} + c_{36} + c_{37} + c_{38})n) \end{aligned} \tag{6.7}$$

The constants can be abstracted into sets of constants. $constants_1$ represents the initialization of the sort method. c_5 and c_6 are called recursive, which are calls to T_5 and T_6 functions. The internal operation of T_5 and T_6 are summarized within $constants_2 n$ and $constants_3 n$, while the n factor is shown outside the scope. $Overhead$ is the set of operation it takes to create two threads, while $O(n)$ is used to abstract the merging process.

$$\begin{aligned} constants_1 &= 3c_1 + 2c_2 + 3c_3 + c_4(2 * (c_{14} + c_{15} + c_{16} + c_{17} + c_{18} + c_{19})) \\ constants_2 n &= c_{20} + c_{21} + (c_{22} + c_{23})n + c_{24} \\ constants_3 n &= c_{25} + c_{26} + c_{27}(c_{28} + c_{29})n + c_{30} \\ overhead &= c_7(c_{39} + c_{40} + c_{41}) + c_8(c_{39} + c_{40} + c_{41}) + c_9 + c_{10} + c_{11} + c_{12} \\ O(n) &= c_{13}(c_{31} + c_{32} + (c_{33} + c_{34} + c_{35} + c_{36} + c_{37} + c_{38})n) \end{aligned}$$

The defined variables can now be used with the recursion formula $2T(\frac{n}{2}) + cn$, to prove the time complexity of merge sort.

$$\begin{aligned} T(n) &= constants_1 + c_5c_6 2T\left[\frac{constants_2\, n}{2} + \frac{constants_3\, n}{2}\right] + overhead + O(n) \\ &= constants_1 + c_5c_6 2\left[2\left(\frac{constants_2\, n}{2} + \frac{constants_3\, n}{2}\right) + O(n)\right] + overhead + O(n) \\ &= constants_1 + c_5c_6 2^2 T\left[\frac{constants_2\, n}{2^2} + \frac{constants_3\, n}{2^2}\right] + overhead + 2O(n) \\ &= constants_1 + c_5c_6 2^2\left[2\left(\frac{constants_2\, n}{2^2} + \frac{constants_3\, n}{2^2}\right) + O(n)\right] + overhead + 2O(n) \\ &= constants_1 + c_5c_6 2^3 T\left[\frac{constants_2\, n}{2^3} + \frac{constants_3\, n}{2^3}\right] + overhead + 3O(n) \\ &= ... = \\ &= constants_1 + c_5c_6 2^k T\left[\left(\frac{constants_2 n}{2^k} + \frac{constants_3 n}{2^k}\right) + \right] + overhead + kO(n) \end{aligned} \tag{6.8}$$

Let $k = log_2\, n$ thus each $2^k = 2^{log_2\, n}$

$$
\begin{aligned}
&constants_1 + c_5 c_6 2^{log_2\, n} \left[\frac{constants_2\, n}{2^{log_2\, n}} + \frac{constants_3\, n}{2^{log_2\, n}}\right] + overhead + log_2\, n\, O(n) \\
&= constants_1 + c_5 c_6 n \left[\frac{constants_2\, n}{n} + \frac{constants_3\, n}{n}\right] + overhead + log_2\, n\, O(n) \\
&= constants_1 + c_5 c_6 n \left[constants_2\, n + constants_3\, n\right] + overhead + log_2\, n\, O(n) \\
&= \Theta(n log_2\, n) \qquad (6.9)
\end{aligned}
$$

The analysis shows that the time complexity of the implemented merge sort has a growth factor of $\Theta(n\, log_2\, n)$. Since the execution of merge sort is the same, regardless of input, a best, worst and average case can be defined as

$O(n\, log_2\, n)$ as upper bound, $\Omega(n\, log_2\, n)$ as lower bound, $\Theta(n\, log_2\, n)$ as tight bound

Since the bounds of best and worst cases are the same, the asymptotically tight bound for the average case will be the same.

A general equation for the implemented merge sort with p threads, where $p \geq 2^1$ and $p = 2^k$, in which the result become $p = 2^{k+1}$, can be concluded as

$$T(n) = Constants_1 + (c_5 c_6 (Constants_2 n + Constants_3 n)) + Overhead + n log_2\, n$$

$$= Constants_1 + \frac{c_5 c_6 (Constants_2 n + Constants_3 n) + n log_2\, n}{p} + (p * Overhead) \qquad (6.10)$$

An equation for sequential merge sort can be concluded as

$$T(n) = Constants_1 + c_5 c_6 (Constants_2 n + Constants_3 n) + log_2\, n\, O(n) \qquad (6.11)$$

6.1.3 Quicksort - best case

The following sets denotes the functions calls used by quicksort which are used to analyse and prove the best case time complexity with its constants. Definition

of T sets can be found in appendix C. Quicksort - best case.

$$
\begin{aligned}
T_1 &= c_{31}\sum_{i=1}^{n/2} t_i + c_{31}\sum_{i=1}^{n/2} t_i(c_{32}\alpha + c_{33}\alpha) + c_{31}\sum_{i=1}^{n/2} t_i(c_{34}\beta + c_{35}\beta) + c_{36}\sum_{i=1}^{n/2} t_i \\
&\quad + c_{37}\sum_{i=1}^{n/2} t_i + c_{38}\sum_{i=1}^{n/2} t_i + c_{39}\sum_{i=1}^{n/2} t_i + c_{40} \\
&= c_{31}(\frac{n}{2}) + c_{31}(\frac{n}{2}(c_{32}\alpha + c_{33}\alpha)) + c_{31}(\frac{n}{2}(c_{34}\beta + c_{35}\beta)) \\
&\quad + c_{36}(\frac{n}{2}) + c_{37}(\frac{n}{2}) + c_{38}(\frac{n}{2}) + c_{39}(\frac{n}{2}) + c_{40} \\
T_2 &= c_{41} + c_{42} + c_{43} + c_{44} + c_{45} + c_{46} + c_{47} + c_{48} + c_{49} + c_{50} + c_{51} + c_{52} + c_{53} \\
&\quad + c_{54} + c_{55} + c_{56} + c_{57} + c_{58} \\
T_3 &= c_{27} + c_{28} + c_{29} + c_{30} \\
T_4 &= c_{17} + c_{18} + c_{19} + c_{20} + c_{21}(T_2) + c_{22}(T_3) + c_{23}(T_1) + c_{24}(T_3) + c_{25}(T_4) + c_{26}(T_4) \\
T_5 &= c_{58} + c_{59} + c_{60} + c_{61} + c_{62}
\end{aligned}
$$

Each of the function calls above defines a subprocess for sorting n elements. The full equation for sorting process is denoted as following

$$
\begin{aligned}
T_n &= c_1(p+1) + c_2(p) + c_3(p+1) + c_4(pT_4) + c_5(\frac{p}{2}) + c_6(\frac{p}{2}) + c_7(\frac{p}{2}T_2) + c_8(\frac{p}{2}T_3) \\
&\quad + c_9(\frac{p}{2}T_1) + c_{10}(\frac{p}{2}T_3) + c_{11}(\frac{p}{2}T_5) + c_{12}(\frac{p}{2}T_5) + c_{13}(\frac{p}{2}) + c_{14}(\frac{p}{2}) + c_{15}(\frac{p}{2}) + c_{16}(\frac{p}{2})
\end{aligned}
\tag{6.12}
$$

Suppose $p = 2$ which denotes that two threads are used, thus the sorting process above will define the equation as following.

$$
\begin{aligned}
T_n &= c_1(2) + c_2(1) + c_3(2) + c_4(2T_4) + c_5(\frac{2}{2}) + c_6(\frac{2}{2}) + c_7(\frac{2}{2}T_2) + c_8(\frac{2}{2}T_3) \\
&\quad + c_9(\frac{2}{2}T_1) + c_{10}(\frac{2}{2}T_3) + c_{11}(\frac{2}{2}T_5) + c_{12}(\frac{2}{2}T_5) + c_{13}(\frac{2}{2}) + c_{14}(\frac{2}{2}) + c_{15}(\frac{2}{2}) + c_{16}(\frac{2}{2})
\end{aligned}
\tag{6.13}
$$

By abstracting the constants and overhead, the equation can be rewritten as following

$$
\begin{aligned}
constants_1 &= c_1(2) + c_2(1) + c_3(2) \\
overhead &= c_5(\frac{2}{2}) + c_6(\frac{2}{2}) + c_7(\frac{2}{2}T_2) + c_8(\frac{2}{2}T_3) + c_9(\frac{2}{2}T_1) + c_{10}(\frac{2}{2}T_3) \\
&\quad + c_{11}(\frac{2}{2}T_5) + c_{12}(\frac{2}{2}T_5) + c_{13}(\frac{2}{2}) + c_{14}(\frac{2}{2}) + c_{15}(\frac{2}{2}) + c_{16}(\frac{2}{2})
\end{aligned}
$$

Which gives the following equation

$$T(n) = constants_1 + c_4(2T_4) + overhead \tag{6.14}$$

However, since *overhead* and t_4 have a function call to T_1,the partition function, the following equation would be expressed by abstracting T_1 from T_4 and *overhead*.

$$\begin{aligned} &constants_1 + c_4(2T_4 + c_{23}(T_1)) + Overhead + c_9(T_1) \\ &= constants_1 + c_4(2T_4 + c_{23}(\frac{5}{2}n + \alpha n + \beta n + c_{40})) + overhead \\ &+ c_9(2T_4 + c_{23}(\frac{5}{2}n + \alpha n + \beta n + c_{40})) \end{aligned} \tag{6.15}$$

The final abstracting phase, uses big oh notation, to rewrite the statements of c_{23} and c_9 such that the partition factors can be used in a recursion formula and remove the constants

$$\begin{aligned} &\frac{5}{2}n = n \,\text{and}\, c_{40} \\ &O(n)(1 + \alpha + \beta) = n + \alpha n + \beta n \end{aligned} \tag{6.16}$$

Thus the equation is defined as following.

$$\begin{aligned} constants_1 &= c_1(2) + c_2(1) + c_3(2) \\ constants_2 &= 2T_4 + c_{23}(T_1) \\ overhead &= overhead - c_9(T_1) \\ T(n) &= constants_1 + c_4(constants_2 + n + \alpha n + \beta n) + overhead + n + \alpha n + \beta n \end{aligned}$$

The defined equation can now be used with the recursion formula $2T(\frac{n}{2}) + cn$, to prove quicksort's time complexity for best case. However, for this, an assumption is made that the pivot value always partitions the data such that each recursive call creates two subproblems with $\frac{n}{2}$ elements.

$$
\begin{aligned}
T(n) &= constants_1 + c_4 2T\left[\frac{constants_2 + n + \alpha n + \beta n}{2}\right] + overhead + n + \alpha n + \beta n \\
&= constants_1 + c_4 2\left[2\left(\frac{constants_2 + n + \alpha n + \beta n}{2}\right) + n + \alpha n + \beta n\right] \\
&+ overhead + n + \alpha n + \beta n \\
&= constants_1 + c_4 2^2 T\left[\frac{constants_2 + n + \alpha n + \beta n}{2^2}\right] + overhead + 2(n + \alpha n + \beta n) \\
&= constants_1 + c_4 2^2\left[2\left(\frac{constants_2 + n + \alpha n + \beta n}{2^2}\right) + n + \alpha n + \beta n\right] \\
&+ overhead + 2(n + \alpha n + \beta n) \\
&= constants_1 + c_4 2^3 T\left[\frac{constants_2 + n + \alpha n + \beta n}{2^3}\right] + overhead + 3(n + \alpha n + \beta n) \\
&= ... = \\
&= constants_1 + c_4 2^{k-1} T\left[\frac{constants_2 + n + \alpha n + \beta n}{2^{k-1}}\right] + overhead + (k-1)(n + \alpha n + \beta n) \\
&= constants_1 + c_4 2^k T\left[\frac{constants_2 + n + \alpha n + \beta n}{2^k}\right] + overhead + (k)(n + \alpha n + \beta n)
\end{aligned}
\tag{6.17}
$$

Let $k = log_2\, n$ thus each $2^k = 2^{log_2\, n}$

$$
\begin{aligned}
&constants_1 + c_4 2^{log_2\, n} T\left[\frac{constants_2 + n + \alpha n + \beta n}{2^{log_2\, n}}\right] + overhead + log_2\, n(n + \alpha n + \beta n) \\
&= constants_1 + c_4 T\left[constants_2 + n + \alpha n + \beta n\right] + overhead + n(1 + \alpha + \beta) log_2 n \\
&= \Theta(n(1 + \alpha + \beta) log_2\, n) \\
&= \Omega(n(1 + \alpha + \beta) log_2\, n)
\end{aligned}
\tag{6.18}
$$

The analysis shows that the time complexity for best case of the implemented quicksort has a growth factor of $\Omega((n + \alpha n + \beta n) log_2\, n)$ which can be defined as $\Omega(n log_2\, n)$ lower bound.

A general equation for best case of the implemented quicksort with p threads, where $p \geq 2^1$ and $p = 2^k$, in which the result become $p = 2^{k+1}$, can be concluded as

$$
\begin{aligned}
&T(n) = constants_1 + c_4(constants_2 + n + \alpha n + \beta n) + overhead + n(1 + \alpha + \beta) log_2\, n \\
&= constants_1 + \frac{c_4(constants_2 + n + \alpha n + \beta n) + n(1 + \alpha + \beta) log_2\, n}{p} + (p * overhead)
\end{aligned}
\tag{6.19}
$$

An equation for the best case of sequential quicksort can be concluded as

$$T(n) = constants_1 + c_4(constants_2 + n + \alpha n + \beta n) + n(1 + \alpha + \beta)log_2\, n \tag{6.20}$$

6.1.4 Quicksort - worst case

The following sets denotes the functions calls used by quicksort which are used to analyse and prove the best case time complexity with its constants. Definition of T sets can be found in appendix D. Quicksort - worst case.

$$\begin{aligned}
T_1 &= c_{31}\sum_{i=1}^{n-1} t_i + c_{31}\sum_{i=1}^{n-1} t_i(c_{32}\alpha + c_{33}\alpha) + c_{31}(c_{34}\beta + c_{35}\beta) + c_{36}\sum_{i=1}^{n-1} t_i \\
&\quad + c_{37}\sum_{i=1}^{n-1} t_i + c_{38}\sum_{i=1}^{n/2} t_i + c_{39}\sum_{i=1}^{n/2} t_i + c_{40} \\
&= c_{31}(n-1) + c_{31}(n - 1(c_{32}\alpha + c_{33}\alpha)) + c_{31}(c_{34}\beta + c_{35}\beta) \\
&\quad + c_{36}(n-1) + c_{37}(n-1) + c_{38}(n-1) + c_{39}(n-1) + c_{40} \\
T_2 &= c_{41} + c_{42} + c_{43} + c_{44} + c_{45} + c_{46} + c_{47} + c_{48} + c_{49} + c_{50} + c_{51} + c_{52} + c_{53} \\
&\quad + c_{54} + c_{55} + c_{56} + c_{57} + c_{58} \\
T_3 &= c_{27} + c_{28} + c_{29} + c_{30} \\
T_4 &= c_{17} + c_{18} + c_{19} + c_{20} + c_{21}(T_2) + c_{22}(T_3) + c_{23}(T_1) + c_{24}(T_3) + c_{25}(T_4) + c_{26}(T_4) \\
T_5 &= c_{58} + c_{59} + c_{60} + c_{61} + c_{62}
\end{aligned}$$

Each of the function calls above defines a subprocess for sorting n elements. The full equation for sorting process is denoted as following.

$$\begin{aligned}
T_n &= c_1(p+1) + c_2(p) + c_3(p+1) + c_4(pT_4) + c_5(\frac{p}{2}) + c_6(\frac{p}{2}) + c_7(\frac{p}{2}T_2) + c_8(\frac{p}{2}T_3) \\
&\quad + c_9(\frac{p}{2}T_1) + c_{10}(\frac{p}{2}T_3) + c_{11}(\frac{p}{2}T_5) + c_{12}(\frac{p}{2}T_5) + c_{13}(\frac{p}{2}) + c_{14}(\frac{p}{2}) + c_{15}(\frac{p}{2}) + c_{16}(\frac{p}{2})
\end{aligned} \tag{6.21}$$

Suppose $p = 2$ which denotes that two threads are used, thus the sorting process above will define the equation as following.

$$\begin{aligned}
T_n &= c_1(2) + c_2(1) + c_3(2) + c_4(2T_4) + c_5(\frac{2}{2}) + c_6(\frac{2}{2}) + c_7(\frac{2}{2}T_2) + c_8(\frac{2}{2}T_3) \\
&\quad + c_9(\frac{2}{2}T_1) + c_{10}(\frac{2}{2}T_3) + c_{11}(\frac{2}{2}T_5) + c_{12}(\frac{2}{2}T_5) + c_{13}(\frac{2}{2}) + c_{14}(\frac{2}{2}) + c_{15}(\frac{2}{2}) + c_{16}(\frac{2}{2})
\end{aligned} \tag{6.22}$$

By abstracting the constants and overhead, the equation can be rewritten as following

$$\begin{aligned} constants_1 &= c_1(2) + c_2(1) + c_3(2) \\ overhead &= c_5(\frac{2}{2}) + c_6(\frac{2}{2}) + c_7(\frac{2}{2}T_2) + c_8(\frac{2}{2}T_3) + c_9(\frac{2}{2}T_1) + c_{10}(\frac{2}{2}T_3) \\ &\quad + c_{11}(\frac{2}{2}T_5) + c_{12}(\frac{2}{2}T_5) + c_{13}(\frac{2}{2}) + c_{14}(\frac{2}{2}) + c_{15}(\frac{2}{2}) + c_{16}(\frac{2}{2}) \end{aligned}$$

Which gives the following equation

$$T(n) = constants_1 + c_4(2T_4) + overhead \tag{6.23}$$

However, since *overhead* and t_4 have a function call to T_1,the partition function, the following equation would be expressed by abstracting T_1 from T_4

$$\begin{aligned} &constants_1 + c_4(2T_4 + c_{23}(T_1)) + overhead \\ &= constants_1 + c_4(2T_4 + c_{23}(5(n-1) + \alpha(n-1) + \beta + c_{40})) + overhead \end{aligned} \tag{6.24}$$

Thus, the equation will be defined as following

$$\begin{aligned} constants_1 &= c_1(2) + c_2(1) + c_3(2) \\ constants_2 &= 2T_4 + c_{23}(T_1) \\ overhead &= c_5 + c_6 + c_7(T_2) + c_8(T_3) + c_9(T_1) + c_{10}(T_3) + c_{11}(T_5) \\ &\quad + c_{12}(T_5) + c_{13} + +c_{14} + c_{15} + +c_{16} \\ T(n) &= constants_1 + c_4(constants_2 + \alpha(n-1) + \beta + 5(n-1) + c_{40}) + overhead \end{aligned} \tag{6.25}$$

The defined equation can now be used to prove the time complexity for worst case of quicksort. However, for this, an assumption is made that the pivot value partitions the data, such that each recursive call creates two subproblems where the left half has (n-1) elements and right half have 1 element. Thus the following recursion is done n times.

$c_4T(constants_2 + \alpha(n-1) + \beta + 5(n-1))$ will equal the following sum

$$\begin{aligned} &\sum_{j=1}^{n} \alpha(n-j) + \beta + 5(n-j) \\ &= (\alpha(n-1) + \beta + 5(n-1)) + (\alpha(n-2) + \beta + 5(n-2)) \\ &+ ... + (\alpha(n-n) + n\beta + 5(n-n)) \\ &= \alpha\left(\frac{n(n-1)}{2}\right) + n\beta + 5\left(\frac{n(n-1)}{2}\right) \end{aligned} \tag{6.26}$$

Thus the equation states

$$T(n) = constants_1 + c_4(constants_2 + \alpha\left(\frac{n(n-1)}{2}\right) + n\beta + 5\left(\frac{n(n-1)}{2}\right)) + overhead$$
$$= \Theta(n^2) \tag{6.27}$$

The analysis shows that the worst case time complexity for the implemented quicksort has a growth factor of $\Theta(n^2) \rightarrow O(n^2)$ upper bound.

A general equation for worst case of the implemented quicksort with p threads, where $p \geq 2^1$ and $p = 2^k$, in which the result become $p = 2^{k+1}$, can be concluded as

$$T(n) = constants_1 + c_4(constants_2 + \alpha\left(\frac{n(n-1)}{2}\right) + n\beta + 5\left(\frac{n(n-1)}{2}\right)) + overhead$$

$$= constants_1 + \frac{c_4(constants_2 + \alpha\left(\frac{n(n-1)}{2}\right) + n\beta + 5\left(\frac{n(n-1)}{2}\right))}{p} + (p * overhead) \tag{6.28}$$

And an equation for the worst case of sequential quicksort can be concluded as

$$T(n) = constants_1 + c_4(constants_2 + \alpha\left(\frac{n(n-1)}{2}\right) + n\beta + 5\left(\frac{n(n-1)}{2}\right)) \tag{6.29}$$

6.1.5 Quick sort - average case

Quicksort's average case defines the tight bound between the upper bound of $O(n^2)$ and the lower bound of $\Omega(n\,log_2\,n)$. Since the bounds of the best and worst case differs, a mathematical approach is taken to determine the average case. This approach differs from previously analysed algorithms, e.g. rank sort, which had the same upper bound as lower bound. When lower and upper bounds are the same, the average case will have the same growth function, as there is no growth function in between the bounds. The average case will therefore be tightly bound to the same growth function, for rank sort $\Theta(n^2)$.

For quicksort this cannot be done in the same manner. It must be shown that the average case is closer to one growth function than the other, and that the growth rate moves towards a bound in some cases. When a function grows towards either bound, probabilistics are often used to describe the of this behaviour. However, in this work a more general approach will be used to show the average case for quicksort.

As previously stated, best case for quicksort is $\Omega(n\,log_2\,n)$, where each recursion divides the problem into halves which contains $\frac{n}{2}$ elements. For worst case, it

was shown to be $O(n^2)$, where each recursion divides the problem into two halves, but with one of half containing $n-1$ elements and the other one element.

Now, suppose that each recursion divides the problem into two halves, with one half containing $n-2$ elements and the other two elements. This would lead to the following

$$\sum_{j=1}^{\frac{n}{2}}(n-2j+1) = 2\frac{\frac{n}{2}(\frac{n}{2})+1}{2} + \frac{2n}{2} = \frac{n^2}{4} + n$$

$$n\,log_2\,n < \frac{n^2}{4} + n < n^2 \tag{6.30}$$

Lets use the same sum, but instead divide the two subproblems such that one half get $n-3$ elements and that the other half gets 3 elements.

$$\sum_{j=1}^{\frac{n}{3}}(n-3j+1) = 3\frac{\frac{n}{3}(\frac{n}{3})+1}{2} + \frac{3n}{2} = \frac{n^2}{6} + \frac{3n}{2}$$

$$n\,log_2\,n < \frac{n^2}{6} + \frac{3n}{2} < \frac{n^2}{4} + n < n^2 \tag{6.31}$$

Consider using the same sum, but instead divide the two subproblems such that one half get $n-m$ elements and the other gets m elements, where $m =$ number of elements greater than $pivot+1$. Thus the following sum is used

$$\sum_{j=1}^{\frac{n}{m}}(n-mj+1) = m\frac{\frac{n}{m}(\frac{n}{m})+1}{2} + \frac{mn}{2} = \frac{n^2}{2m} + \frac{nm}{2}$$

$$n\,log_2\,n < \frac{n^2}{2m} + \frac{nm}{2} < ... < \frac{n^2}{6} + \frac{3n}{2} < \frac{n^2}{4} + n < n^2 \tag{6.32}$$

By increasing the pivot value, such that it moves closer to the middle and by balancing each recursions partitions, the growth rate moves from the worst case $O(n^2)$ towards best case $\Omega(n\,log_2\,n)$. However, in order to actually prove that the average case is $\Theta(n\,log_2\,n)$, and where the break point changes from n^2 to $n\,log_2\,n$, further studies with the inclusion of probability is needed [28].

6.1.6 Bitonic sort

The following sets denotes the functions calls used by bitonic sort which are used to analyse and prove the time complexity with its constants [29]. Definition of T

sets can be found in appendix E. Bitonic sort.

$$T_1 = c_1 + c_2(T_2) + c_3$$
$$T_2 = c_4 + c_5 + c_6 + c_7(T_2) + c_8(T_2) + c_9(T_3) + c_{10}(\frac{p}{2}(T_6)) + c_{11}(\frac{p}{2}(T_6)) + c_{12}(\frac{p}{2}) + c_{13}(\frac{p}{2}) + c_{14}(\frac{p}{2}) + c_{15}(\frac{p}{2}) + c_{16}(\frac{p}{2}(T_3))$$
$$T_3 = c_{17} + c_{18} + c_{19}(\frac{n}{2}) + c_{20}(\frac{n}{2}(T_4)) + c_{21}(T_3) + c_{22}(T_3)$$
$$T_4 = c_{23} + c_{24}(T_3)$$
$$T_5 = c_{25} + c_{26} + c_{27}$$
$$T_6 = c_{28} + c_{29} + c_{30} + c_{31} + c_{32}$$

Each of the function calls above defines a sub-process for sorting n elements. The full equation for the sorting process is denoted as following.

$$T(n) = c_1 + c_2(c_4 + c_5 + c_6 + c_7(T_2) + c_8(T_2) + c_9(T_3) + c_{10}(\frac{p}{2}(T_6)) + c_{11}(\frac{p}{2}(T_6)) + c_{12}(\frac{p}{2}) + c_{13}(\frac{p}{2}) + c_{14}(\frac{p}{2}) + c_{15}(\frac{p}{2}) + c_{16}(\frac{p}{2}(T_3)) + c_3 \tag{6.33}$$

Suppose that p = 2 which denotes that two threads are used, thus the sorting process above contains all function calls and the number of threads used, will define the equation as following

$$T(n) = c_1 + c_2(c_4 + c_5 + c_6 + c_7(T_2) + c_8(T_2) + c_9(T_3) + c_{10}(T_6) + c_{11}(T_6) + c_{12} + c_{13} + c_{14} + c_{15} + c_{16}(T_3)) + c_3 \tag{6.34}$$

The equation uses the recursion formula $2T(\frac{n}{2}) + cn$, to prove time complexity for bitonic sort. $T_2$is defined as the outer recursive calls shown below, but T_2 also contains has a recursive call to T_3 internally. Thus, the inner recursive call is defined as following

$$T(n) = c_1 + c_2(c_4 + c_5 + c_6 + 2T\left[\frac{c_7(T_2) + c_8(T_2)}{2}\right] + c_9(T_3) + c_{10}(T_6) + c_{11}(T_6) + c_{12} + c_{13} + c_{14} + c_{15} + c_{16}(T_3)) + c_3 \tag{6.35}$$

Inner recursion

Since T_2 contains a recursive call of T_3, this inner recursion needs to solved first. Thus T_3 is defined as following

$$T_3 = c_{17} + c_{18} + c_{19}(\frac{n}{2}) + c_{20}(\frac{n}{2}(T_4)) + c_{21}(T_3) + c_{22}(T_3)$$
$$= constants_1 + \frac{n}{2}(c_{19} + c_{20}(T_4)) + c_{21}c_{22}(T_3)$$

Where $constants_1$ is defined as following.

$constants_1 = c_{17} + c_{18}$

Below, $T(n_s)$ is defined as the number of steps for merging each recursion. An additional cost of returning each recursion is defined as $O(1)$. Using the recursion formula $2T(\frac{n}{2}) + cn$ with the equation, the following expression is defined

$$T(n_s) = 2T\left[\frac{constants_1 + \frac{n}{2}(c_{19} + c_{20}(T_4)) + c_{21}c_{22}(T_3)}{2}\right] + O(1) \tag{6.36}$$

Let $T(n_s)$ be the depth of the bitonic sorting

$$\begin{aligned} T(1) &= 0 \\ T(n_s) &= T\left(\frac{n_s}{2}\right) + 1, for\, n > 1 \end{aligned} \tag{6.37}$$

$$\begin{aligned} T(n_s) &= 2T\left[\frac{Constants_1 + \frac{n}{2}(c_{19} + c_{20}(T_4)) + c_{21}C_{22}(T_3)}{2}\right] + O(1) \\ &= 2\left[2\left(\frac{Constants_1 + \frac{n}{2}(c_{19} + c_{20}(T_4)) + c_{21}C_{22}(T_3)}{2} + \frac{O(1)}{2}\right)\right] + O(1) \\ &= 2^2T\left[\frac{Constants_1 + \frac{n}{2}(c_{19} + c_{20}(T_4)) + c_{21}C_{22}(T_3)}{2^2}\right] + O(2) \\ &= 2^2\left[2\left(\frac{Constants_1 + \frac{n}{2}(c_{19} + c_{20}(T_4)) + c_{21}C_{22}(T_3)}{2^2} + \frac{O(1)}{2^2}\right)\right] + O(2) \\ &= 2^3T\left[\frac{Constants_1 + \frac{n}{2}(c_{19} + c_{20}(T_4)) + c_{21}c_{22}(T_3)}{2^3}\right] + O(3) \\ &= 2^3\left[2\left(\frac{Constants_1 + \frac{n}{2}(c_{19} + c_{20}(T_4)) + c_{21}c_{22}(T_3)}{2^3} + \frac{O(1)}{2^3}\right)\right] + O(3) \\ &= .. = \\ &= 2^{k-1}T\left[\frac{Constants_1 + \frac{n}{2}(c_{19} + c_{20}(T_4)) + c_{21}c_{22}(T_3)}{2^{k-1}}\right] + O(k-1) \\ &= 2^{k-1}\left[2\left(\frac{Constants_1 + \frac{n}{2}(c_{19} + c_{20}(T_4)) + c_{21}c_{22}(T_3)}{2^{k-1}} + \frac{O(1)}{2^{k-1}}\right)\right] + O(k-1) \\ &= 2T\left[\frac{Constants_1 + \frac{n}{2}(c_{19} + c_{20}(T_4)) + c_{21}c_{22}(T_3)}{2^k}\right] + O(k) \end{aligned} \tag{6.38}$$

Let $k = log_2\, n$ thus each $2^k = 2^{log_2\, n}$

$$
\begin{aligned}
&2^{log_2\, n}\left[\frac{Constants_1 + \frac{n}{2}(c_{19} + c_{20}(T_4)) + c_{21}c_{22}(T_3)}{2^{log_2\, n}}\right] + O(log_2\, n) \\
&= Constants_1 + \frac{n}{2}(c_{19} + c_{20}(T_4)) + c_{21}c_{22}(T_3) + O(log_2\, n) \\
&= \Theta(log_2\, n)
\end{aligned} \tag{6.39}
$$

Thus, it is shown that merging takes $\Theta(log_2\, n)$

Outer recursion

Now that $T_3 = log_2\, n$ has been defined, let $T(n_s)$ be the number of steps in each recursion. The inner recursion 6.1.6 is defined as the time it takes to merge given subproblems for the recursion formula $2T(\frac{n}{2}) + cn$.

$$T(n_s) = 2T\left[\frac{c_7(T_2) + c_8(T_2)}{2}\right] + O(log_2\, n)$$

Let $T(n_s)$ be the depth of the bitonic sorting

$$
\begin{aligned}
&T(1) = 0 \\
&T(n_s) = T(\frac{n_s}{2}) + O(log_2\, n),\ for\, n > 1
\end{aligned} \tag{6.40}
$$

$$
\begin{aligned}
T(n_s) &= 2T\left[\frac{c_7c_8(T_2)}{2}\right] + O(log_2\, n) \\
&= 2\left[2\left(\frac{c_7c_8(T_2)}{2} + \frac{O(log_2\, n)}{2}\right)\right] + O(log_2\, n) \\
&= 2^2T\left[\frac{c_7c_8(T_2)}{2^2}\right] + 2O(log_2\, n) \\
&= 2^2\left[2\left(\frac{c_7c_8(T_2)}{2^2} + \frac{O(log_2\, n)}{2^2}\right)\right] + 2O(log_2\, n) \\
&= 2^3T\left[\frac{c_7c_8(T_2)}{2^3}\right] + 3O(log_2\, n) \\
&= 2^3\left[2\left(\frac{c_7c_8(T_2)}{2^3} + \frac{O(log_2\, n)}{2^3}\right)\right] + 3O(log_2\, n) \\
&= ... = \\
&= 2^{k-1}T\left[\frac{c_7c_8(T_2)}{2^{k-1}}\right] + (k-1)O(log_2\, n) \\
&= 2^{k-1}\left[2\left(\frac{c_7c_8(T_2)}{2^{k-1}} + \frac{O(log_2\, n)}{2^{k-1}}\right)\right] + (k-1)O(log_2\, n) \\
&= 2^kT\left[\frac{c_7c_8(T_2)}{2^k}\right] + kO(log_2\, n) \qquad (6.41)
\end{aligned}
$$

Let $k = log_2\, n$ thus each $2^k = 2^{log_2\, n}$

$$
\begin{aligned}
&2^{log_2\, n}\left[\frac{c_7c_8(T_2)}{2^{log_2\, n}}\right] + log_2\, nO(log_2\, n) \\
&= c_7c_8(T_2) + log_2\, n\, O(log_2\, n) \\
&= O(log_2\,(n)^2) \qquad (6.42)
\end{aligned}
$$

Using the original equation for bitonic sort

$$
\begin{aligned}
T(n) = c_1 + c_2(c_4 + c_5 + c_6 + 2T\left[\frac{c_7(T_2) + c_8(T_2)}{2}\right] + c_9(T_3) + c_{10}(T_6) + c_{11}(T_6) \\
+ c_{12} + c_{13} + c_{14} + c_{15} + c_{16}(T_3)) + c_3 \qquad (6.43)
\end{aligned}
$$

With the sets of constants

$$
\begin{aligned}
constants_1 &= c_1 + c_3 \\
constants_2 &= c_4 + c_5 + c_6 \\
overhead &= c_{10}(T_6) + c_{11}(T_6) + c_{12} + c_{13} + c_{14} + c_{15} + c_{16}(T_3)
\end{aligned}
$$

Gives the new expression

$$T(n) = constants_1 + c_2(constants_2 + 2T\left[\frac{c_7 c_8(T_2)}{2}\right] + O(log_2\, n) + overhead \tag{6.44}$$

By inserting the result from the outer recursion, we get that bitonic sort have a time complexity of

$$\begin{aligned} T(n) &= constants_1 + c_2(constants_2 + O(log_2\,(n)^2)) + log_2\, n + overhead \\ &= \Theta(log_2\,(n)^2) \end{aligned} \tag{6.45}$$

Since bitonic sort executes the same way regardless of input, best, worst, and average cases can be defined as following:

$O(log_2\,(n)^2)$ as upper bound, $\Omega(log_2\,(n)^2)$ as lower bound, $\Theta(log_2\,(n)^2)$ as tight bound

Since both the best and worst cases bounds are the same, the asymptotically tight bound for the average case will be the same.

A general equation for the implemented bitonic sort with p threads, where $p \geq 2^1$ and $p = 2^k$, in which the result become $p = 2^{k+1}$, can be concluded as

$$T(n) = constants_1 + \left(\frac{c_2(constants_2 + O(log_2\,(n)^2)) + log_2\, n}{p}\right) + (overhead * p) \tag{6.46}$$

And an equation for sequential bitonic sort can be concluded as

$$T(n) = constants_1 + c_2(constants_2 + O(log_2\,(n)^2)) + log_2\, n \tag{6.47}$$

6.1.7 Bucket sort

The following sets denotes the functions calls used by bucket sort which are used to analysed and prove the time complexity with its constants. Definition of T sets can be found in appendix F. Bucket sort.

$$T_1 = c_1 + c_2 + c_4(T_3) + c_9 k(c_6 m) + c_7(T_4)$$
$$T_2 = c_{35} + c_{36} + c_{37} + \frac{c_{39} k(c_{40} + c_{41})}{p}$$
$$T_3 = c_{25} n(c_{26} + c_{27})$$
$$T_4 = c_{21} + c_{22} k(c_{23} m(c_{24})) = c_{21} + c_{22} k + n(c_{23} + c_{24})$$

Each of the above function calls defines a subprocess for sorting n elements. The full equation for the sorting process is denoted as following.

$$\begin{aligned} T_n &= c_8 + c_{12} + c_{16} + c_{11}(T_3) + p(c_{13}(T_2) + c_{14} + c_{15}) + c_{20}(T_4) \\ &= c_8 + c_{12} + c_{16} + c_{11}(c_{25}n(c_{26} + c_{27})) + p(c_{13}(c_{35} + c_{36} + c_{37} + c_{39}k(c_{40} + c_{41}))) \\ &+ c_{20}(c_{21} + c_{22}k + n(c_{23} + c_{24})) \end{aligned} \tag{6.48}$$

From the equation above, sets of constants are defined

$$\begin{aligned} constants1 &= c_8 + c_{12} + c_{16} \\ constants2n &= c_{25}n(c_{26} + c_{27}) \\ constants3n &= n(c_{23} + c_{24}) \\ overhead &= c_{35} + c_{36} + c_{37} \end{aligned}$$

Best case

For this implementation, c_{41} refers to insertion sort which is used for internal sorting of buckets. Insertion sort has a best case time complexity of $\Theta(n)$. Used in this case, the algorithm can be considered running close to linear time when the size of each bucket is small and partial sorted. Using insertion sort on each bucket, gives n/k sorted elements.

$$c_{41} = \frac{n}{k}$$

$$\begin{aligned} T(n) &= constants_1 + c_{11}(constants_2 n) + p(c_{13}(overhead + c_{39}k(c_{40} + c_{41}\frac{n}{k}))) \\ &+ c_{20}(c_{21} + c_{22}k + constants_3 n) \\ &= c_{39}k(c_{40} + c_{41}\frac{n}{k}) = c_{39}(c_{40}k + c_{41}n) \\ &= \Theta(n + k) \rightarrow \Omega(n + k) \end{aligned} \tag{6.49}$$

Thus the analysis shows that the implemented bucket sort best case time complexity has a growth factor of $\Theta(n + k) \rightarrow \Omega(n + k)$ lower bound.

A general equation for best case of the implemented bucket sort with p threads, where $p \geq 2^1$, can be concluded as

$$\begin{aligned} T(n) = constants_1 + c_{11}(constants_2 n) + p(c_{13}(overhead + \frac{O(n + k)}{p})) \\ + c_{20}(c_{21} + c_{22}k + constants_3 n) \end{aligned} \tag{6.50}$$

An equation for best case of the sequential bucket sort can be concluded as

$$T(n) = constants_1 + O(n + k) \tag{6.51}$$

Worst case

If all elements are located in one bucket, the time to sort all elements are dependent on the sorting algorithm used for internal sort of the bucket. For insertion sort, worst case gives $O(n^2)$

$$c_{41} = n^2$$

$$\begin{aligned} T(n) &= constants_1 + c_{11}(constants_2 n) + p(c_{13}(overhead + c_{39}k(c_{40} + c_{41}n^2))) \\ &\quad + c_{20}(c_{21} + c_{22}k + constants_3 n) \\ &\quad = c_{39}k(c_{40} + c_{41}n^2) = c_{39}(c_{40}k + c_{41}n^2) \\ &\quad \Theta(n^2) \rightarrow O(n^2) \end{aligned} \tag{6.52}$$

Thus the analysis shows that the implemented bucket sort worst case time complexity have a growth factor of $\Theta(n^2) \rightarrow O(n^2)$ upper bound

A general equation for worst case of the implemented bucket sort with p threads, where $p \geq 2^1$, can be concluded as

$$\begin{aligned} T(n) = constants_1 + c_{11}(constants_2 n) + p(c_{13}(overhead + \frac{O(n^2)}{p})) \\ + c_{20}(c_{21} + c_{22}k + constants_3 n) \end{aligned} \tag{6.53}$$

An equation for worst case of the sequential bucket sort can be concluded as

$$T(n) = constants_1 + O(n^2) \tag{6.54}$$

Average case

To prove the average case of bucket sort, an assumption can be made that [0,1] elements are evenly distributed over [0,1] buckets. Therefore, there will be O(1) elements in each bucket, which in turn takes O(1) in expected time to sort. If n elements are distributed over n buckets in O(n), and removed from the buckets and put back together in O(n+k), the expected time for the algorithm would be $O(n + k) \rightarrow \Theta(n + k)$.[30]

Even if the elements are not evenly distributed, the expected time for the algorithm could still be in linear time if the sum of the squares of bucket sizes is linear to the total number of elements.[31]

6.1.8 Radix sort

The following sets denotes the functions calls used by radix sort which are used to analysed and prove the time complexity with its constants. Definition of T

sets can be found in appendix G. Radix sort.

$$T_1 = c_{18} + c_{19} + c_{20}n(c_{21} + c_{22}) + c_{23}k(c_{24}) + c_{25}n(c_{26} + c_{27}) + c_{28}n(c_29)$$
$$T_2 = c_{30} + c_{31} + c_{32} + c_{33}n(c_{34} + c_{35}) + c_{36}k(c_{37}) + c_{38}n(c_{39} + c_{40}) + c_{41}n(c_{42})$$
$$T_3 = c_7 + c_{10} + c_{14} + p(c_{11}(T_4) + c_{12} + c_{13}) + T_5$$
$$T_4 = c_{44} + c_{45} + c_{46} + c_{47} + c_{48} + c_{49} + c_{52}\frac{k}{p}$$
$$T_5 = c_{51}d(c_{53} + c_{54}(T_2))$$

Each of the function calls above defines a subprocess for sorting n elements. The full equation for the sorting process is denoted as following.

$$\begin{aligned} T(n) &= c_1 + c_2 + c_3(T_1) + c_6(T_3) \\ &= c_1 + c_2 + c_3(c_{18} + c_{19} + c_{20}n(c_{21} + c_{22}) + c_{23}k(c_{24}) + c_{25}n(c_{26} + c_{27}) \\ &\quad + c_{28}n(c_{29}) + c_6(c_7 + c_{10} + c_{14} + p(c_{11}(T_4) + c_{12} + c_{13}) + T_5) \end{aligned} \tag{6.55}$$

From the equation above, sets of constants are defined and inserted to T(n)

$$constants_1 = c_1 + c_2$$
$$constants_2 = c_{18} + c_{19}$$
$$constants_3 = c_7 + c_{10} + c_{14}$$
$$constants_4 = c_{12} + c_{13}$$

Which gives the following equation.

$$\begin{aligned} T(n) &= constants_1 + c_3(constants_2 + c_{20}n(c_{21} + c_{22}) + c_{23}k(c_{24}) \\ &\quad + c_{25}n(c_{26} + c_{27}) + c_{28}n(c_{29})) + c_6(constants_3 + p(c_{11}(T_4) + constants_4)T_5) \end{aligned} \tag{6.56}$$

Using of Big-O notation, abstracts c_3 such that the expression is rewritten as following

$$T(n) = constants_1 + O(n + k) + c_6(constants_3 + p(c_{11}(T_4 + constants_4) + T_5) \tag{6.57}$$

T_4 is now defined with its constants as $overhead = T_4$

Such that

$$\begin{aligned} T(n) &= constants_1 + O(n + k) + c_6(constants_3 + p(c_{11}(overhead + constants_4) \\ &\quad + c_{51}d(c_{53} + c_{54}(t_2))) \end{aligned} \tag{6.58}$$

By abstracting $c_{54}(T_2)$ with Big-O notation gives following

$$\begin{aligned} T_2 &= c_{30} + c_{31} + c_{32} + c_{33}n(c_{34} + c_{35}) + c_{36}k(c_{37}) + c_{38}n(c_{39} + c_{40}) + c_{41}n(c_{42}) \\ &= O(n + k) \end{aligned} \tag{6.59}$$

Inserting T_2 and abstracting with Big-O notation

$$\begin{aligned} T(n) &= constants_1 + O_1(n + k) + c_6(constants_3 + p(c_{11}(overhead + constants_4) \\ &\quad + c_{51}d(c_{53} + O_2(n + k))) \\ &= \Theta(d(n + k)) \end{aligned} \tag{6.60}$$

Since radix sort executes the same way no matter the input, worst, best and average cases can be defined as following

$O(d(n + k))$ as upper bound, $\Omega(d(n + k))$ as lower bound, $\Theta(d(n + k))$ as tight bound

Since the bounds of best and worst cases are the same, the asymptotically tight bound for the average case will be the same.

A general equation for the implemented radix sort with p threads, where $p \geq 2^1$, can be concluded as

$$T(n) = constants_1 + O_1(n + k) + c_6(constants_3 + (p * c_{11}(overhead + constants_4))$$

$$+ O_2(\frac{d(n + k)}{p})) \tag{6.61}$$

And the equation for the sequential radix sort can be concluded as

$$T(n) = constants_1 + O_1(n + k) + c_6(constants_3 + O_2(d(n + k))) \tag{6.62}$$

6.2 Experiments

6.2.1 Rank sort

Rank sort: speedup						
Data	Random order		Increase order		Decrease order	
	T=2	T=4	T=2	T=4	T=2	T=4
2^4	0.5	1.1	0.05	0.03	0.006	0.005
2^8	3.9	2.1	0.35	0.17	0.3	0.3
2^{12}	2.1	4.1	1.3	1.5	1.7	3.2
2^{16}	2.0	3.0	1.8	2.8	2.1	3.6
2^{20}	2.1	3.3	2.0	3.2	2.2	3.5

Table 6.1: Rank sort - speedup.

As seen in table 6.1, as the only n^2 algorithm, rank sort provides the greatest speedup when executing in parallel. Parallel execution does not become efficient until the size of the data set reaches 2^{12}, with exception for random order. In random order there is a significant increase in performance at 2^8 elements. However, at this level, using two threads is more efficient than four threads, likely due to the extra cost in more overhead.

According to table 5.1, when running in rank sort in sequential, 2^{20} elements takes over 40 minutes to sort. Using two threads brings down the time to around 20 minutes, and four threads mitigates the time even further to around 13 minutes. A difference noticeable even to us humans. Sorting in increasing order and decreasing is substantially faster execution times. Even though the algorithm uses the same routine to calculate rank elements indifference to the structure of the data set, the operation of comparing an element to other elements in a sorted order executes faster than elements in random order.

6.2.2 Merge sort

For merge sort, according to table 6.2, running the algorithm in parallel does not become efficient until the size of the data sets reach 2^{16}. There is a difference in execution time between randomized data set, and sorted data sets. This likely due to the step of merging all subsets back together. In already sorted order, elements do not need to be moved over long distances as could be the case in a randomized data set.

When the speedup steadies, the results are quite similar between the different data sets. Two threads results in a speedup just below two, and four threads results in speedup of just below three.

Merge sort: speedup						
Data	Random order		Increase order		Decrease order	
	T=2	T=4	T=2	T=4	T=2	T=4
2^4	0.04	0.01	0.006	0.002	0.003	0.002
2^8	0.2	0.05	0.1	0.03	0.06	0.04
2^{12}	1.1	0.7	1.0	0.5	0.8	0.6
2^{16}	1.7	2.5	1.7	2.2	1.7	2.4
2^{20}	1.9	3.1	1.9	2.9	1.8	2.8
2^{22}	1.9	3.0	1.8	2.6	1.7	2.5
2^{24}	1.8	2.8	1.6	2.2	1.6	2.1

Table 6.2: Merge sort - speedup.

6.2.3 Quicksort

Quick sort: speedup						
Data	Random order		Increase order		Decrease order	
	T=2	T=4	T=2	T=4	T=2	T=4
2^4	0.6	0.4	0.006	0.002	0.05	0.02
2^8	0.7	0.3	0.02	0.009	0.03	0.01
2^{12}	2.9	1.8	0.8	0.3	1.2	0.9
2^{16}	1.4	1.8	1.4	1.4	1.7	1.7
2^{20}	1.8	2.0	1.3	1.3	1.6	1.7
2^{22}	1.7	2.8	1.3	1.3	1.5	1.4
2^{24}	1.2	1.3	1.0	1.3	1.2	1.5

Table 6.3: Quicksort - speedup.

From table 6.3, running quicksort in parallel does not provide any efficiency until the data set reaches 2^{12} in size for random order, and 2^{16} for increased- and decreased order. The results are quite interesting, since the algorithm is generally considered as favourable when discussing sorting of data. For randomized data set, a speedup can be achieved by parallelisation. But compared to already sorted data sets, the outcome is different. Even though the technique used for choosing pivot help in mitigating worst case scenario, the execution time still moves towards the worst case. For a randomized data set of size 2^{24}, the execution time is just under two seconds. In contrast, a sorted data set of same size give a execution time of about three hours.

In addition, while there is some speedup to be achieved from parallelisation, there isn't much difference between using two and four threads. The reason for this, is that through the partition in each recursion, only small number of elements are sorted each time. As the total execution time is dependent on the thread with

the largest part of the data set to sort, the extra resources does provide much assistance to improve the performance.

6.2.4 Bitonic sort

Bitonc sort: speedup						
Data	Random order		Increase order		Decrease order	
	T=2	T=4	T=2	T=4	T=2	T=4
2^4	0.04	0.01	0.007	0.002	0.005	0.002
2^8	0.2	0.1	0.17	0.06	0.2	0.07
2^{12}	1.1	1.4	1.3	1.0	1.3	1.1
2^{16}	1.7	2.6	1.7	2.6	1.7	2.5
2^{20}	1.8	2.9	1.8	2.8	1.8	2.8
2^{22}	1.9	3.0	1.8	2.9	1.8	2.9
2^{24}	1.9	3.1	1.8	3.0	1.8	3.0

Table 6.4: Bitonic sort - speedup.

From table 6.4, running bitonic sort in parallel does not become efficient until data sets of size 2^{12}. Prior to this, the overhead for the threads slows down the total execution time. The results of bitonic sort are similar to the ones of merge sort, as they work using similar techniques.

The results in table 5.4, shows a distinct difference in execution time between data sets in random order and data sets sorted in increasing and decreasing order. This is most likely due to the fact that bitonic sort works by creating bitonic sequences that are either monotonically increasing or decreasing. If the data is already sorted in these orders, the algorithm can speed up the process of creating sequences, hence sorting the elements more efficiently. As the data sets grows larger, the speedup becomes more steady for respective threads. Running two threads gives a speedup of just under two, while executing four threads provides a speedup of just below three.

6.2.5 Bucket sort

Bucket sort: speedup						
Data	Random order		Increase order		Decrease order	
	T=2	T=4	T=2	T=4	T=2	T=4
2^4	0.04	0.03	0.005	0.003	0.002	0.001
2^8	0.1	0.2	0.03	0.01	0.09	0.05
2^{12}	0.4	0.6	0.3	0.1	0.8	0.6
2^{16}	1.2	1.2	0.9	0.5	1.3	1.4
2^{20}	1.3	1.3	1.2	0.7	1.5	1.6
2^{22}	1.2	1.2	1.2	0.7	1.5	1.7
2^{24}	1.1	1.1	1.2	0.7	1.5	1.7

Table 6.5: Bucket sort - speedup.

As seen in table 6.5, running bucket sort in parallel does not become efficient until the data set reaches 2^{16} in size. In addition, the speedup remains about the same for all data sets.

The results from bucket sort is quite interesting to analyse. The first noticeable aspect, is the results from execution of data sets in increasing order. As insertion sort is used to sort each bucket internally, an already sorted data set gives insertion sort its best case. As a result, the execution times are very fast running in sequential. Using two threads provides some increase in performance, but it could be considered as negligible in relation to the additional cost of parallelisation. Using four threads, the extra costs in overhead by creating and using multiple threads becomes a greater factor than the increase in performance, resulting in parallel slowdown.

The same can be said for the randomized and decreasing data sets. While using four threads does not result in a parallel slowdown, the extra overhead in relation to using only two threads does not provide any additional speedup.

Decreased order proved to be faster than random order. This caused some confusion, and further investigation was needed. After some testing, it was discovered that the deciding factor for these results was the operation of distributing elements into buckets. The reason for this is somewhat unclear, but one could assume that in average, elements have to move a shorter distances when they appear in a sorted order in relation to randomized order. Therefore, the cost of moving elements in a sorted order into buckets is lower than moving elements in a randomized order.

In general, the results shows lower speedup compared to other algorithms. This could be narrowed down to the implementation not being optimized. As it uses and is dependent on insertion sort for internal sorting of each bucket, the performance will in general not be optimized for parallel execution. As a

generic implementation of bucket sort was made, tests showed that a reasonable performance could be achieved when each bucket contained approximately 128 elements. Fine tuning this parameter might give different results.

6.2.6 Radix sort

Radix sort: speedup						
Data	Random order		Increase order		Decrease order	
	T=2	T=4	T=2	T=4	T=2	T=4
2^4	0.06	0.05	0.008	0.005	0.006	0.005
2^8	0.7	0.9	0.1	0.06	0.08	0.06
2^{12}	0.7	0.6	0.9	0.6	0.8	0.8
2^{16}	1.4	1.6	1.4	1.6	1.4	1.6
2^{20}	1.7	2.2	1.7	2.2	1.7	2.2
2^{22}	1.6	2.4	1.6	2.4	1.6	2.4
2^{24}	1.7	2.5	1.7	2.5	1.7	2.5

Table 6.6: Radix sort - speedup.

As seen in table 6.6, running radix sort in parallel does not become efficient until the data set reaches 2^{16} in size. After this point, there is an average speedup of 1.6 for two threads and an average speedup around 2.2 when running four threads. Noticeably, there is very little difference in the results between the different orders. Due to design and workings of the algorithm, the order in which the elements appear in the data sets only a has negligible effect on the performance. As it evaluates the significant digits for each element, the order in which the element appear doesn't matter as they range for each pass is equivalent to the radix, which is the same for all elements.

6.2.7 Thread threshold

When using multiple threads, an increase in performance is to be expected, but it is bound to its max potential. As shown in the result of the experiment of thread threshold, the max performance potential of the threads are bound to the number of physical cores available, which in the work is four cores, four threads.

Using more threads than physical available, virtual threads, will not increase the performance. Instead it increases the parallelity with a cost, such that more code can be executed simultaneously in different states. Each virtual thread will be bound to a physical core and will be given a window of available execution time before it gets interrupted by another thread or process. These interruption causes the performance to be bound to the physical available cores, since the max concurrency is bound to physical cores. As shown in the results, increasing the

number of threads will never outperform four threads in this case, even if the parallelity is increased.

As shown in figures 6.1 6.2 6.3, creating threads increases the overall execution time, but in some instances a sequential approach can be better than a parallel. When an input is very small, the thread overhead takes more time than solving the actual problem. However, when the input grows bigger, the overhead is compensated by the performance of the executing thread, such that the overall execution time is faster. When the overhead is compensated, the performance increases. At this point, it is shown that multithreading is faster than its sequential counterpart and its parallel counterpart, if the thread is increased within the bound of physical available threads. Thus in a performance aspect, using threads are only suitable when the data is big enough such that the thread overhead is compensated for and that the number of threads used are within the number of physical available threads.

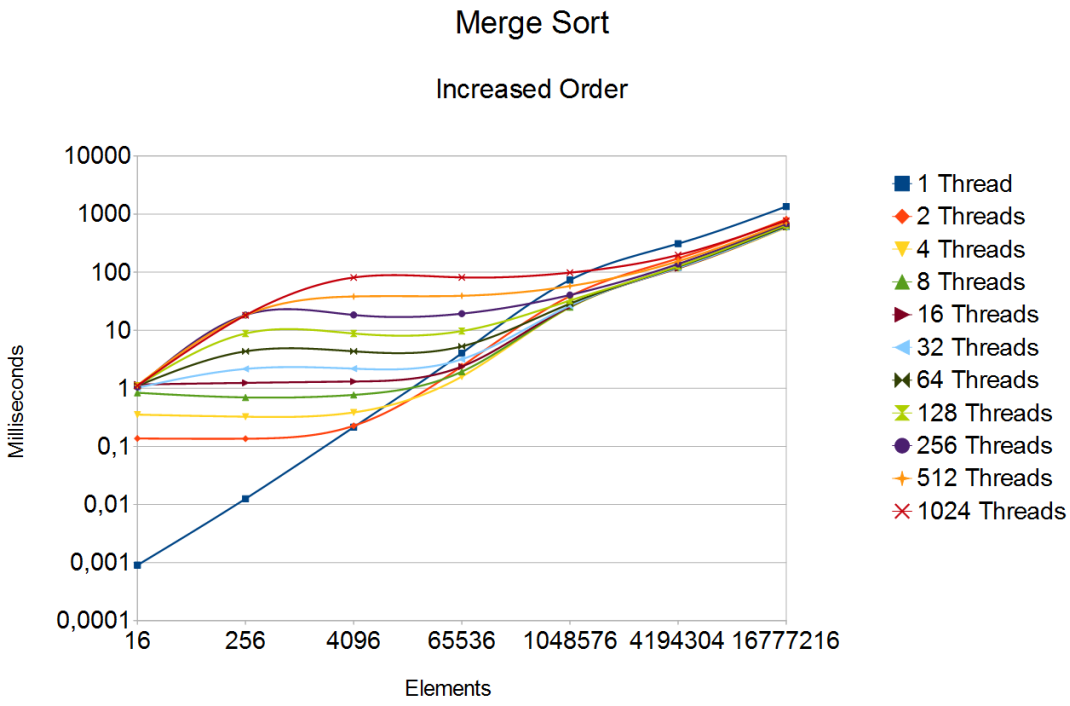


Figure 6.1: Thread threshold - increasing order.

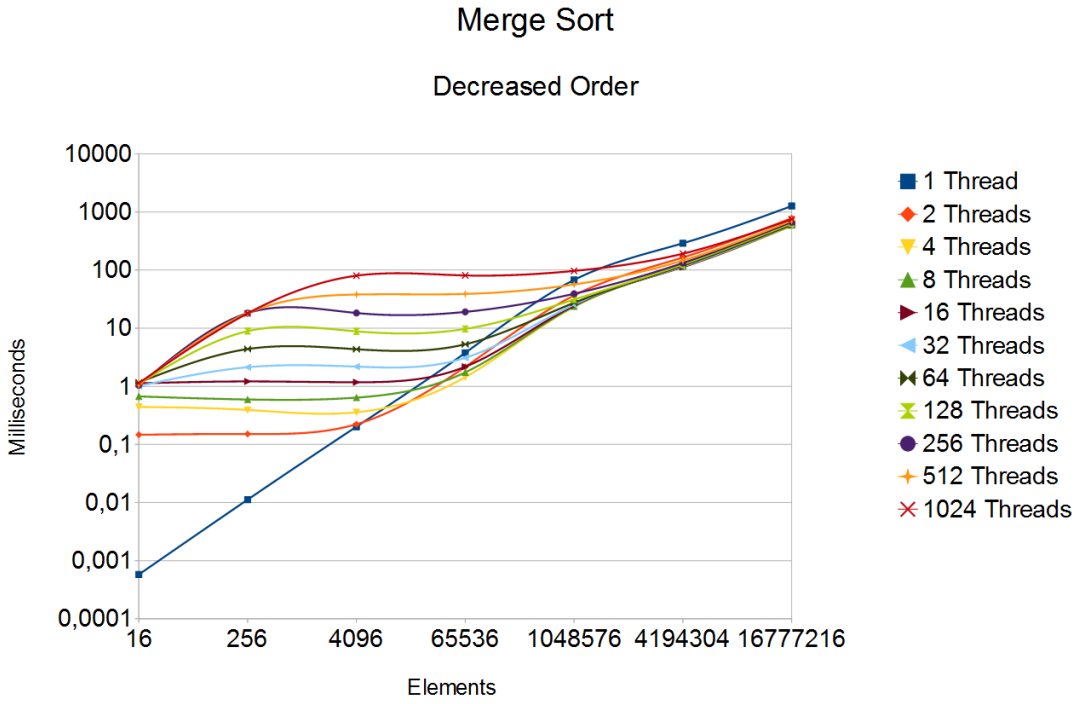


Figure 6.2: Thread threshold - decreasing order.

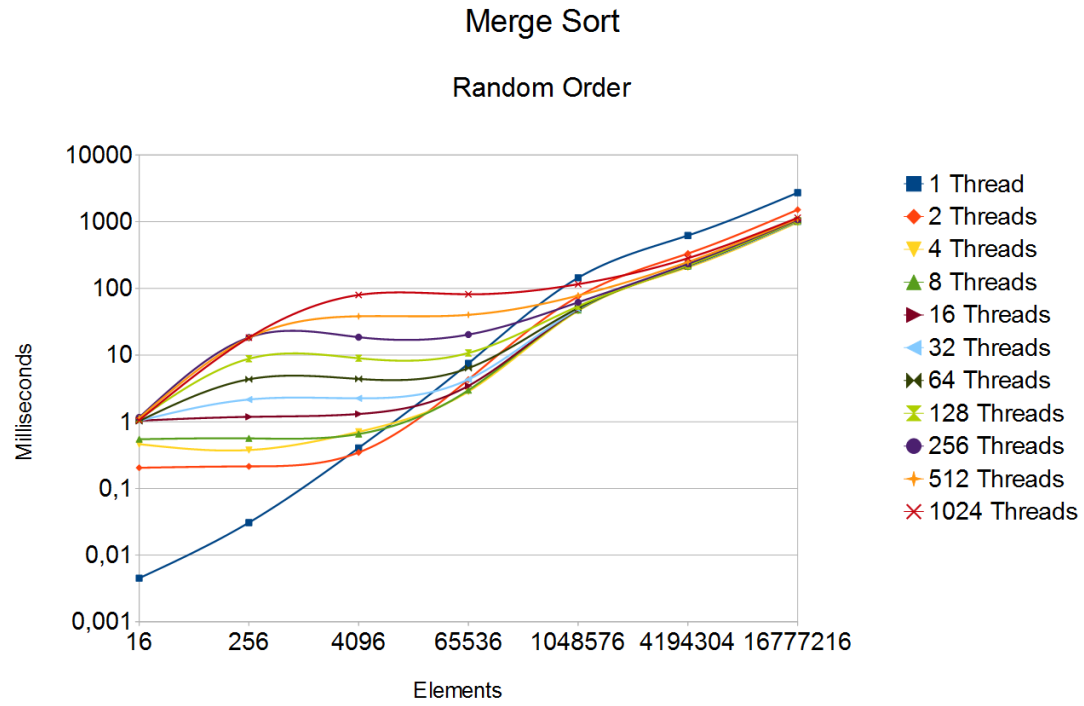


Figure 6.3: Thread threshold - random order.

6.2.8 Unbalanced workload

As mentioned in section 3.4.1, a key factor to consider in order to take full advantage of the extra resources available in a multi-core system is workload balancing. The execution time for parallel algorithms is dependent on the time for the processor with the largest workload. If the workload is evenly distributed between the processors available, they will approximately do the same amount of work in the same amount of time. However, balancing the workload is not always an easy task, and can't always be achieved.

As a part of the experiment, 6.4 shows the difference in execution and speedup for different workload ratios between two threads. In this case, quicksort is used with a data set of size 2^{20} elements.

Unbalanced workload: ms		
Ratio	Time	Speedup
1:1	55	1
1:2	67	0, 82
1:3	74	0, 74
1:4	81	0, 68

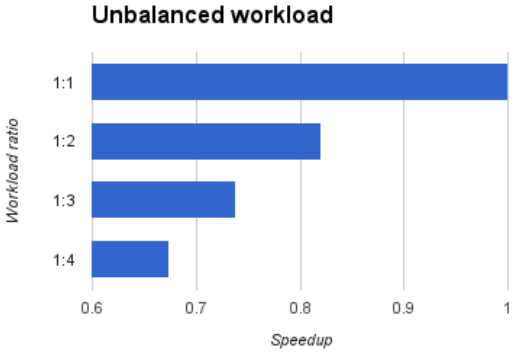


Figure 6.4: Unbalanced workload, quicksort 2^{20} elements

From figure 6.4, it can be concluded that an imbalance in workload of the available processors does effect the efficiency of execution. An even distribution between the threads sets a benchmark, from which different ratios are compared to. A 1:2 ratio, makes the execution time 12 milliseconds slower. It may not seem all the much, but it results in a decrease in speedup to 0.82. The decrease continues to grow as the workload ration grows further apart. The final result of a 1:4 ratio shows an execution time of 81 milliseconds, and a speedup of 0.68. This means that an even distribution of data between two threads are 33 percent faster then a 1:4 workload ratio between threads. With larger data sizes, the difference would become even more apparent. For a system or application that is time critical in nature, clearly this would not be sustainable.

Chapter 7

Summary

This work has evaluated the performance of six different sorting algorithms as sequential and parallel.

- Rank sort
- Merge sort
- Quicksort
- Bitonic sort
- Bucket sort
- Radix sort

All the algorithms have been implemented in Java, using the *Runnable* interface, such that each algorithm have support for parallelisation. The implemented algorithms was benchmarked and tested with different parallelisation levels, on data sets ranging from 2^4 to 2^{24} elements. As a result, execution times and parallel speedup could studied and analysed.

The implemented algorithms have also been analysed mathematically, by converting each operations into mathematical statements. The conversion have been used to calculate time complexity for best, worst and average cases. The mathematical analysis gives a general equation for parallelisation, such that calculations for an algorithm can be done, depending on the number of threads used. The analysed time complexities can be seen in table 7.1.

Algorithm	Worst case	Average case	Best case
Rank sort	$O(n^2)$	$\Theta(n^2)$	$\Omega(n^2)$
Quicksort	$O(n^2)$	$\Theta(n\,log_2\,n)$	$\Omega(n\,log_2\,n)$
Merge sort	$O(n\,log_2\,n)$	$\Theta(n\,log_2\,n)$	$\Omega(n\,log_2\,n)$
Bitonic sort	$O(log_2(n)^2)$	$\Theta(log_2(n)^2)$	$\Omega(log_2(n)^2)$
Radix sort	$O(d(n+k))$	$\Theta(d(n+k))$	$\Omega(d(n+k))$
Bucket sort	$O(n^2)$	$\Theta(n+k)$	$\Omega(n+k)$

Table 7.1: Time complexity - implemented algorithms.

Chapter 8

Conclusions and Future Work

8.1 Conclusion

In this work it is shown that utilizing multithreading as a source for more raw computer power, indeed increases the performance of an algorithm. However, the increase in starts off when the data being processed is sufficiently big enough that the overhead for thread creation is a smaller factor than the overall execution time.

To answer the research questions, stated in the beginning of this work, the following conclusions can be made.

- How much can an algorithm's efficiency and performance be improved by multithreading?

Overall, the results shows that in general, executing two threads gives an increase in performance of up to $2x$ the original execution time. Using four threads, provides an increase of $2.5x$ to $3x$ the original execution time. However, there are some exceptions. Algorithms like quicksort, that can not guarantee a stable execution and balanced workload, the overall speedup may vary pending on the partition of data. Also, an algorithm like bucket sort, who relies on an additional sorting algorithm, the potential speedup is dependant on the execution for said algorithm.

With exception of rank sort, most algorithms did not reach a speedup of $2x$ when executing with two threads. This is likely due to the implementation, and lack of optimization of source code. With a better and more efficient source code, the results should be closer to the expected parallel speedup of $2x$.

The same can be said for executing with four threads, in which the results were nowhere near a speedup of $4x$. However, in this case, Java might be a partial reason for it as well. As mentioned previously in this work, the JVM and the JIT caused some problems and challenges. After completing this experiment, we have come to learn that Java might not be a perfect choice for this kind of experiment, which also seems be the general consensus among people who work in the field. Perhaps the results would have been better and closer to a $4x$ speedup for another platform.

Another reason for not fully reaching the potential speedup might be the hardware used in the experiment. The computer on which the tests were executed had a quad-core CPU. Using all cores at once may not give full effect as other processes might be running concurrently, stealing some processing power. Had more cores been available, a $4x$ speedup for four threads might have been reached. But same problem would have most likely occurred when reaching the maximum number of threads, no matter what hardware had been used.

- What affects the performance and efficiency of algorithms, how are they different in multithreaded environments?
  - Is there a threshold for multithreaded algorithms in terms of performance, and what are the reasons for it?
  - How does unbalanced workload in multithreaded environments change the performance of an algorithm?

As shown in the tables in section 6.2, the start off point for an increase in performance differs depending on the algorithm and its design and structure. For the slowest algorithm, table 6.1, the starting point comes earlier then for the fastest one, table 6.5. In general, using multithreading on data below 2^{16} elements, generally does not provide much increase in performance, since the overhead of the threads are greater than solving the sorting problem in sequential.

A shown in figure, 6.3, the limit for increase in performance is bound to the physical available threads for the CPU. Using more threads than physical available will result in longer execution time due to an extra cost in overhead. To compensate for this, the size of data must be big enough before the same maximum performance will be bound to the physical available threads.

When using the appropriate number of threads correctly, there is a factor as to how the work balance is distributed between each thread. The results shows, figure 6.4, that dividing the workload, such that there is an imbalance, will result in a decrease in performance. This is due to the fact the overall execution time is always dependent on the longest executing thread. If there is an imbalance in workload, no substantial gain is made for the threads that completes their work quickly, as they would halt in order to await the slowest thread before synchronization can occur between the threads.

- How can time complexity be proven, by conversion of source code to mathematical statements?

The mathematical analysis, in section 6.1, shows that the implemented algorithms time complexity agrees with the expected outcomes. Conversion from source code to mathematical statements can be used to prove time complexities and be used to analyse characteristics of an algorithm.

8.2 Future work

This work has focused on parallel execution using sorting algorithms. However, there are many other types algorithms that could be studied further, such as searching, dynamic programming, mathematical calculations, or other types of data structures.

There are many aspects of the Java platform that could be further researched. Benchmarking is an example, how to do this accurately while mitigating the fluctuations of the results. Finding a reliable method and compare it to already existing tools for this concept.

Another possible aspect is to execute algorithms using different techniques for multithreading within Java, and compare the result to see if there is a favourable method. In the latest Java release, 1.8, a fork/join framework was included with the potential of simplifying the implementation of multithreading. A further analysis of this framework could be done, to see if it actually provides any advantages towards other techniques for multithreading.

The results of this work is specific to the implemented algorithms. As no optimization have been made regarding the algorithms, a further study could be done in optimizing the algorithms using the results from this work as a baseline for potential improvements.

In addition, creating a method to make the analysis of time complexity, in particular the average case, for algorithms that are dependant on probability factors could be a potential for future work. In some way define a method for input of a random variable that simplifies and shortens the calculations for analysing time complexities. As a prime example, proving the average case for quicksort, which requires complex calculations.

References

[1] Wang, D., Zhang, X., Men, T., Wang, M., & Qin, H. (2012, March). An implementation of sorting algorithm based on Java multi-thread technology. In *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*(Vol. 1, pp. 629-632). IEEE.

[2] Mahafzah, B. A. (2013). Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture. *The Journal of Supercomputing*, 66(1), 339-363.

[3] Qian, X. J., & Xu, J. B. (2011, May). Optimization and implementation of sorting algorithm based on multi-core and multi-thread. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on* (pp. 29-32). IEEE.

[4] Kathavate, S., & Srinath, N. K. (2014). Efficiency of Parallel Algorithms on Multi Core Systems Using OpenMP. *International Journal of Advanced Research in Computer and Communication Engineering,* 3(10), 8237-8241.

[5] Satish, N., Harris, M., & Garland, M. (2009, May). Designing efficient sorting algorithms for manycore GPUs. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (pp. 1-10). IEEE.

[6] Damien DeVille. (2010, Oct). Sorting algorithms comparison. http://ddeville.me/2010/10/sorting-algorithms-comparison (accessed 2015, June)

[7] Sareen, P. (2013). Comparison of sorting algorithms (on the basis of average case). *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(3), 522-532.

[8] Karunanithi, A. K. (2014). A Survey, Discussion and Comparison of Sorting Algorithms.

[9] Qureshi, K. (2006). A Practical Performance Comparison of Parallel Sorting Algorithms on Homogeneous Network of Workstations. In *PDPTA* (pp. 615-619).

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to algorithms, third edition, ISBN: 9780262533058, 2009-08

[11] Gunnar Sparr. Linjär algebra (1997). ISBN: 9789144197524

[12] Kimmo Eriksson, Hillevi Gavel. Diskret matematik och diskreta modeller (2013). ISBN: 9789144089997

[13] ALECU, A. L. F. (2005). Parallel rank sort. *Economy Informatics. v1-4*, 49-51.

[14] Kaur, M. Comparative Study of Parallel Odd Even Transposition and Rank Sort Algorithm.

[15] Anastasio, T. (2003). Bitonic Sorting. Dec, 4, 1-2.

[16] Amato, N. M., Iyer, R., Sundaresan, S., & Wu, Y. (1996). *A comparison of parallel sorting algorithms on different architectures.* Technical Report TR98-029, Department of Computer Science, Texas A&M University.

[17] Dewanchand, M., & Blankendaal, R. (1999). *The usablility of Java for developing parallel applications* (Doctoral dissertation, Master's thesis, Vrije Universiteit Amsterdam).

[18] Kale, V., & Solomonik, E. (2010, March). Parallel sorting pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns* (p. 10). ACM.

[19] Żurek, D., Pietroń, M., Wielgosz, M., & Wiatr, K. (2013). The comparison of parallel sorting algorithms implemented on different hardware platforms. *Computer Science*, 14(4)), 679-691.

[20] Aater Suleman. (2011) What makes parallel programming hard? http://www.futurechips.org/tips-for-power-coders/parallel-programming.html (accessed 2015, June)

[21] Aater Suleman. (2011) Writing and Optimizing Parallel Programs — A complete example. http://www.futurechips.org/tips-for-power-coders/writing-optimizing-parallel-programs-complete.html (accessed 2015, June)

[22] Aater Suleman. (2011) Parallel programming: How to choose the best task-size? http://www.futurechips.org/software-for-hardware-guys/parallel-programming-choose-task-size.html (accessed 2015, June)

[23] Java Runnable interface.
https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html (accessed 2015, June)

[24] University of Washington Computer Science and Engineering. CSE 373: Data Structures and Algorithms, Winter 2013
https://courses.cs.washington.edu/courses/cse373/13wi/lectures/03-13/MergeSort.java (accessed 2015, June)

[25] Hans Werner Lang. FH Flensburg.
http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm (accessed 2015, June)

[26] Robert Sedgewick, Kevin Wayne. Department of Computer Science Princeton University, Lecture notes - Radix sort, 2007
https://www.cs.princeton.edu/r̃s/AlgsDS07/18RadixSort.pdf (accessed 2015, June)

[27] Brian Goetz. (2004, Dec) Java theory and practice: Dynamic compilation and performance measurements.
http://www.ibm.com/developerworks/library/j-jtp12214/j-jtp12214-pdf.pdf (accessed 2015, June)

[28] Hanan Ayad. SYDE 423 - Computer Algorithm Design and Analysis. Lecture - average-case analysis of quicksort
http://www.hananayad.com/teaching/syde423/quickSortAvgCase.pdf (accessed 2015, June)

[29] Donald E. Knuth. The Art of Computer Programming. Volume 3, Second Edition Updated and Revised, ISBN: 9780321751041, 2011-03

[30] Ben Gurion of University of the Negev. Data Structures - 2014/Spring. Lecture: Sorting in Linear Time
http://www.cs.bgu.ac.il/d̃s142/wiki.files/ds142_ps11_updated.pdf (accessed 2015, June)

[31] Horsmalahti, P. (2012). Comparison of Bucket Sort and RADIX Sort. *arXiv preprint arXiv:1206.3511.*

Appendix

A. Rank sort

B. Merge sort

C. Quicksort - best case

D. Quicksort - worst case

E. Bitonic sort

F. Bucket sort

G. Radix sort

A. Rank sort

n = number of elements
p = number of processes

RankSort.sort(int[] a, int[] b, int level)

Line	RankSort.sort()	Cost	Times
1	if level == 0	c1	1
2	rankSort(a, step, start, b)	c2	1
3	else if level == 1	c3	1
4	parallelSort(a, b, nrOfThreads)	c4	1
5	else if level == 2	c5	1
6	parallelSort(a, b, nrOfThreads)	c4	1

RankSort.parallelSort(int[] a, int[] b, int nrOfThreads)

Line	RankSort.parallelSort()	Cost	Times
7	if nrOfThreads == 2	c7	1
8	new thread()	c8	p
9	thread.start()	c9	p
10	thread.join()	c10	p
11	else if nrOfThreads == 4	c11	1
12	new thread()	c12	p
13	thread.start()	c13	p
14	thread.join()	c14	p

RankSort.rankSort(int[] a, int[] b, int start, int step)

Line	RankSort.rankSort	Cost	Times
15	for i = start, i < a.length, i+=step	c15	n/p
16	index = 0	c16	1
17	for j = 0, j < a.length, j++	c17	n
18	if(a[i] > a[j] || (a[i] == a[j] && (j < i)))	c18	1

19	index++	c19	1
20	b[index] = a[i]	c20	n/p

RankSortSorter(int[] a, int[] b, int start, int step)

Line	RankSortSorter	Cost	Times
21	RankSortSorter()		
22	this.a = a	c21	1
23	this.step = step	c22	1
24	this.start = start	c23	1
25	this.b = b	c24	1
26	run()		
27	RankSortT1.rankSort(a, step, start, b)	c25	1

ParallelSort

$$T_1 = c_7 + c_{11} + p(c_8(T_2) + c_9 + c_{10})$$

Sorter

$$T_2 = c_{21} + c_{22} + c_{23} + c_{24} + c_{25}$$

RankSort

$$T_3 = c_{15}n(c_{16} + c_{17}n(c_{18} + c_{19}) + c_{20})/p$$

Sort

$$T_4 = c_1 + c_2 + c_3 + c_4(T_1) + T_3$$

B. Merge sort

n = number of elements
p = number of processes

MergeSort function - sort(int[] a, int level)

Line	Merge-sort(A).sort()	Cost	Times
1	if a.length < 2	c1	p+1
2	return	c2	p
3	if level == 0	c3	p+1
4	mergeSort(a)	c4	p
5	else	0	p
6	leftHalf(a)	c5	p/2
7	rightHalf(a)	c6	p/2
8	leftThread.Sorter(left)	c7	p/2
9	rightThread.Sorter(right)	c8	p/2
10	leftThread.start()	c9	p/2
11	rightThread.start()	c10	p/2
12	rightThread,join()	c11	p/2
13	leftThread.join()	c12	p/2
14	merge(left, right, a)	c13	p/2

MergeSort function - mergeSort(int[] a)

Line	Merge-sort(a).mergeSort()	Cost	Times
1	if a.length >= 2	c14	1
2	leftHalf(a)	c15	1
3	rightHalf(a)	c16	1
4	mergeSort(left)	c17	1
5	mergeSort(right)	c18	1
6	merge(left, right, a)	c19	1

MergeSort function - leftHalf(int[] a)

Line	Merge-sort(a).leftHalf()	Costs	Times
1	size1 = a.length / 2	c20	1
2	left = new array [size 1]	c21	1
3	for i=0, i<size1, i++	c22	$\sum_{i=1}^{n/2} t_i$
4	left[i] = a[i]	c23	$\sum_{i=1}^{n/2} t_i$
5	return left	c24	1

MergeSort function - rightHalf(int[] a)

Line	Merge-sort(a).rightHalf()	Costs	Times
1	size1 = a.length / 2	c25	1
2	size2 = a.length - size1	c26	1
3	right = new array[size2]	c27	1
4	for i=0, i<size2, i++	c28	$\sum_{i=1}^{n/2} t_i$
5	right[i] = a[i + size1]	c29	$\sum_{i=1}^{n/2} t_i$
6	return right	c30	1

MergeSort function - merge(int[] left, int[] right, int[] a)

Line	Merge-sort(a).merge()	Costs	Times
1	i1 = 0	c31	1
2	i2 = 0	c32	1

3	for i=0, i<a.length, i++	c33	$\sum_{i=1}^{n} t_i$
4	if i2 >= right.length || (i1 < left.length && left[i1] < right[i2])	c34	$\sum_{i=1}^{n} t_i$
5	a[i] = left[i1]	c35	$\sum_{i=1}^{n} t_i$
6	i1++	c36	$\sum_{i=1}^{n} t_i$
7	else	0	0
8	a[i] = right[i2]	c37	$\sum_{i=1}^{n} t_i$
9	i2++	c38	$\sum_{i=1}^{n} t_i$

MergeSort function - Sorter(int[] a, int level)

Line	Merge-sort().Sorter()	Costs	Times
1	this.a = a	c39	1
2	this.level = level	c40	1

MergeSort function - run()

Line	Merge-sort().run()	Costs	Times
1	MergeSort.sort()	c41	1

Merge sort

$$T_1 = C_{14} + C_{15} + C_{16} + C_{17} + C_{18} + C_{19}$$

Left Half

$$T_2 = C_{20} + C_{21} + C_{22}\sum_{i=1}^{n/2} t_i + C_{23}\sum_{i=1}^{n/2} t_i + C_{24} = C_{20} + C_{21} + C_{22}\tfrac{n}{2} + C_{23}\tfrac{n}{2} + C_{24}$$

$$= C_{20} + C_{21} + (C_{22} + C_{23})n + C_{24}$$

Right Half

$$T_3 = C_{25} + C_{26} + C_{27} + C_{28}\sum_{i=1}^{n/2} t_i + C_{29}\sum_{i=1}^{n/2} t_i + C_{30} = C_{25} + C_{26} + C_{27} + C_{28}\tfrac{n}{2} + C_{29}\tfrac{n}{2} + C_{30}$$

$$= C_{25} + C_{26} + C_{27} + (C_{28} + C_{29})n + C_{30}$$

Run & Sorter

$$T_4 = C_{39} + C_{40} + C_{41}$$

Merge

$$T_5 = C_{31} + C_{32} + C_{33}\sum_{i=1}^{n} t_i + C_{34}\sum_{i=1}^{n} t_i + C_{35}\sum_{i=1}^{n} t_i + C_{36}\sum_{i=1}^{n} t_i + C_{37}\sum_{i=1}^{n} t_i + C_{38}\sum_{i=1}^{n} t_i$$

$$= C_{31} + C_{32} + C_{33}n + C_{34}n + C_{35}n + C_{36}n + C_{37}n + C_{38}n$$

$$= C_{31} + C_{32} + (C_{33} + C_{34} + C_{35} + C_{36} + C_{37} + C_{38})\, n$$

C. Quicksort - Best Case

n = number of elements
p = number of processes

QuickSort function - sort(int[] a, int min, int max, int level)

Line	Quick-sort(A).sort()	Cost	Times
1	if min >= max	C_1	$p+1$
2	return	C_2	p
3	if level == 0	C_3	$p+1$
4	quickSort(a, mind, max)	C_4	p
5	else	0	p
6	pivot = -1	C_5	$\frac{p}{2}$
7	pivotIndex = -1	C_6	$\frac{p}{2}$
8	getPivot(a, pivot, pivotIndex)	C_7	$\frac{p}{2}$
9	swap(a, pivotIndex, max)	C_8	$\frac{p}{2}$
10	middle = partition(a, min, max-1, pivot)	C_9	$\frac{p}{2}$
11	swap(a, middle, max)	C_{10}	$\frac{p}{2}$
12	leftThread.Sorter(a, min, middle-1, level-1)	C_{11}	$\frac{p}{2}$
13	rightThread.Sorter(a, middle+1, max, level-1)	C_{12}	$\frac{p}{2}$
14	leftThread.start()	C_{13}	$\frac{p}{2}$
15	rightThread.start()	C_{14}	$\frac{p}{2}$
16	leftThread.join()	C_{15}	$\frac{p}{2}$
17	rightThread.join()	C_{16}	$\frac{p}{2}$

QuickSort function - quickSort(int[] a, int min, int max)

Line	Quick-sort(A).getPivot()	Cost	Times
1	if min >= max	C_{17}	1
2	return	C_{18}	1

3	pivot = -1	C_{19}	1
4	pivotIndex = -1	C_{20}	1
5	getPivot(a, pivot, pivotIndex)	C_{21}	1
6	swap(a,pivotIndex, max)	C_{22}	1
7	middle = partition(a, min, max-1, pivot)	C_{23}	1
8	swap(a, middle, max)	C_{24}	1
9	quickSort(a, min, middle-1)	C_{25}	1
10	quickSort(a, middle+1, max)	C_{26}	1

QuickSort function - swap(int[] a, int i, int j)

Line	Quick-sort(A).swap()	Cost	Times
1	if i != j	C_{27}	1
2	temp = a[i]	C_{28}	1
3	a[i] = a[j]	C_{29}	1
4	a[j] = temp	C_{30}	1

QuickSort function - partition(int[] a, int i, int j, int pivot)

Line	Quick-sort(A).partition()	Cost	Times
1	while i <= j	C_{31}	$\sum_{i=1}^{n/2} t_i$
2	while i<=j && a[i] < pivot	C_{32}	α
3	i++	C_{33}	α
4	while i<=j && a[j] > pivot	C_{34}	β
5	j--	C_{35}	β
6	if i <= j	C_{36}	$\sum_{i=1}^{n/2} t_i$
7	swap(a, i, j)	C_{37}	$\sum_{i=1}^{n/2} t_i$

8	i++	C_{38}	$\sum_{i=1}^{n/2} t_i$
9	j--	C_{39}	$\sum_{i=1}^{n/2} t_i$
10	return i	C_{40}	1

QuickSort function - getPivot(int[] a, int pivot, int pivotIndex)

Line	Quick-sort(A).getPivot()	Cost	Times
1	if a[min] < a[max]	C_{41}	1
2	if a[min] < a[max/2]	C_{42}	1
3	if a[max] < a[max/2]	C_{43}	1
4	pivot = a[max]	C_{44}	1
5	pivotIndex = max	C_{45}	1
6	else	0	
7	pivot = a[max/2]	C_{46}	1
8	pivotIndex = max/2	C_{47}	1
9	else	0	
10	pivot = a[min]	C_{48}	1
11	pivotIndex = min	C_{49}	1
12	else	0	
13	if a[min] < a[max/2]	C_{50}	1
14	pivot = a[min]	C_{51}	1
15	pivotIndex = min	C_{52}	1
16	else	0	
17	if a[max] < a[max/2]	C_{53}	1
18	pivot = a[max/2]	C_{54}	1
19	pivotIndex = max/2	C_{55}	1
20	else	0	

21	pivot = a[max]	C_{56}	1
22	pivotIndex = max	C_{57}	1

QuickSort function - run()

Line	Quick-sort(A).run()	Cost	Times
1	QuickSort.sort(a, min, max, level)	C_{58}	1

QuickSort function - QuickSortSorter(int[] a, int min, int max, int level)

Line	Quick-sort(A).run()	Cost	Times
1	this.a = a	C_{59}	1
2	this.level = level	C_{60}	1
3	this.min = min	C_{61}	1
4	this.max = max	C_{62}	1

Partition

$$T_1 = C_{31}\sum_{i=1}^{n/2} t_i + C_{31}\sum_{i=1}^{n/2} t_i(C_{32}\alpha + C_{33}\alpha) + C_{31}\sum_{i=1}^{n/2} t_i(C_{34}\beta + C_{35}\beta) + C_{36}\sum_{i=1}^{n/2} t_i + C_{37}\sum_{i=1}^{n/2} t_i$$
$$+ C_{38}\sum_{i=1}^{n/2} t_i + C_{39}\sum_{i=1}^{n/2} t_i + C_{40}$$
$$= C_{31}(\tfrac{n}{2}) + C_{31}(\tfrac{n}{2}(C_{32}\alpha + C_{33}\alpha)) + C_{31}(\tfrac{n}{2}(C_{34}\beta + C_{35}\beta)) + C_{36}(\tfrac{n}{2}) + C_{37}(\tfrac{n}{2}) + C_{38}(\tfrac{n}{2}) + C_{39}(\tfrac{n}{2}) + C_{40}$$

getPivot

$$T_2 = C_{41} + C_{42} + C_{43} + C_{44} + C_{45} + C_{46} + C_{47} + C_{48} + C_{49} + C_{50} + C_{51} + C_{52} + C_{53} + C_{54} + C_{55} + C_{56} + C_{57} + C_{58}$$

swap

$$T_3 = C_{27} + C_{28} + C_{29} + C_{30}$$

quickSort

$$T_4 = C_{17} + C_{18} + C_{19} + C_{20} + C_{21}(T_2) + C_{22}(T_3) + C_{23}(T_1) + C_{24}(T_3) + C_{25}(T_4) + C_{26}(T_4)$$

Run&Sorter

$$T_5 = C_{58} + C_{59} + C_{60} + C_{61} + C_{62}$$

D. Quicksort - Worst Case

n = number of elements
p = number of processes

QuickSort function - sort(int[] a, int min, int max, int level)

Line	Quick-sort(A).sort()	Cost	Times
1	if min >= max	C_1	$p+1$
2	return	C_2	p
3	if level == 0	C_3	$p+1$
4	quickSort(a, mind, max)	C_4	p
5	else	0	p
6	pivot = -1	C_5	$\frac{p}{2}$
7	pivotIndex = -1	C_6	$\frac{p}{2}$
8	getPivot(a, pivot, pivotIndex)	C_7	$\frac{p}{2}$
9	swap(a, pivotIndex, max)	C_8	$\frac{p}{2}$
10	middle = partition(a, min, max-1, pivot)	C_9	$\frac{p}{2}$
11	swap(a, middle, max)	C_{10}	$\frac{p}{2}$
12	leftThread.Sorter(a, min, middle-1, level-1)	C_{11}	$\frac{p}{2}$
13	rightThread.Sorter(a, middle+1, max, level-1)	C_{12}	$\frac{p}{2}$
14	leftThread.start()	C_{13}	$\frac{p}{2}$
15	rightThread.start()	C_{14}	$\frac{p}{2}$
16	leftThread.join()	C_{15}	$\frac{p}{2}$
17	rightThread.join()	C_{16}	$\frac{p}{2}$

QuickSort function - quickSort(int[] a, int min, int max)

Line	Quick-sort(A).getPivot()	Cost	Times
1	if min >= max	C_{17}	1
2	return	C_{18}	1

3	pivot = -1	C_{19}	1
4	pivotIndex = -1	C_{20}	1
5	getPivot(a, pivot, pivotIndex)	C_{21}	1
6	swap(a,pivotIndex, max)	C_{22}	1
7	middle = partition(a, min, max-1, pivot)	C_{23}	1
8	swap(a, middle, max)	C_{24}	1
9	quickSort(a, min, middle-1)	C_{25}	1
10	quickSort(a, middle+1, max)	C_{26}	1

QuickSort function - swap(int[] a, int i, int j)

Line	Quick-sort(A).swap()	Cost	Times
1	if i != j	C_{27}	1
2	temp = a[i]	C_{28}	1
3	a[i] = a[j]	C_{29}	1
4	a[j] = temp	C_{30}	1

QuickSort function - partition(int[] a, int i, int j, int pivot)

Line	Quick-sort(A).partition()	Cost	Times
1	while i <= j	C_{31}	$\sum_{i=1}^{n-1} t_i$
2	while i<=j && a[i] < pivot	C_{32}	α
3	i++	C_{33}	α
4	while i<=j && a[j] > pivot	C_{34}	β
5	j--	C_{35}	β
6	if i <= j	C_{36}	$\sum_{i=1}^{n-1} t_i$
7	swap(a, i, j)	C_{37}	$\sum_{i=1}^{n-1} t_i$

8	i++	C_{38}	$\sum_{i=1}^{n-1} t_i$
9	j--	C_{39}	$\sum_{i=1}^{n-1} t_i$
10	return i	C_{40}	1

QuickSort function - getPivot(int[] a, int pivot, int pivotIndex)

Line	Quick-sort(A).getPivot()	Cost	Times
1	if a[min] < a[max]	C_{41}	1
2	if a[min] < a[max/2]	C_{42}	1
3	if a[max] < a[max/2]	C_{43}	1
4	pivot = a[max]	C_{44}	1
5	pivotIndex = max	C_{45}	1
6	else	0	
7	pivot = a[max/2]	C_{46}	1
8	pivotIndex = max/2	C_{47}	1
9	else	0	
10	pivot = a[min]	C_{48}	1
11	pivotIndex = min	C_{49}	1
12	else	0	
13	if a[min] < a[max/2]	C_{50}	1
14	pivot = a[min]	C_{51}	1
15	pivotIndex = min	C_{52}	1
16	else	0	
17	if a[max] < a[max/2]	C_{53}	1
18	pivot = a[max/2]	C_{54}	1
19	pivotIndex = max/2	C_{55}	1
20	else	0	

21	pivot = a[max]	C_{56}	1
22	pivotIndex = max	C_{57}	1

QuickSort function - run()

Line	Quick-sort(A).run()	Cost	Times
1	QuickSort.sort(a, min, max, level)	C_{58}	1

QuickSort function - QuickSortSorter(int[] a, int min, int max, int level)

Line	Quick-sort(A).run()	Cost	Times
1	this.a = a	C_{59}	1
2	this.level = level	C_{60}	1
3	this.min = min	C_{61}	1
4	this.max = max	C_{62}	1

Partition

$$T_1 = C_{31}\sum_{i=1}^{n-1} t_i + C_{31}\sum_{i=1}^{n-1} t_i(C_{32}\alpha + C_{33}\alpha) + C_{31}(C_{34}\beta + C_{35}\beta) + C_{36}\sum_{i=1}^{n-1} t_i + C_{37}\sum_{i=1}^{n-1} t_i$$
$$+ C_{38}\sum_{i=1}^{n-1} t_i + C_{39}\sum_{i=1}^{n-1} t_i + C_{40}$$
$$= C_{31}(n-1) + C_{31}(n-1(C_{32}\alpha + C_{33}\alpha)) + C_{31}(C_{34}\beta + C_{35}\beta) + C_{36}(n-1) + C_{37}(n-1) + C_{38}(n-1)$$
$$+ C_{39}(n-1) + C_{40}$$

getPivot

$$T_2 = C_{41} + C_{42} + C_{43} + C_{44} + C_{45} + C_{46} + C_{47} + C_{48} + C_{49} + C_{50} + C_{51} + C_{52} + C_{53} + C_{54} + C_{55} + C_{56} + C_{57} + C_{58}$$

swap

$$T_3 = C_{27} + C_{28} + C_{29} + C_{30}$$

quickSort

$$T_4 = C_{17} + C_{18} + C_{19} + C_{20} + C_{21}(T_2) + C_{22}(T_3) + C_{23}(T_1) + C_{24}(T_3) + C_{25}(T_4) + C_{26}(T_4)$$

Run&Sorter

$$T_5 = C_{58} + C_{59} + C_{60} + C_{61} + C_{62}$$

E. Bitonic sort

n = number of elements
p = number of processes

BitonicSort function - sort(int[] externalData, int level)

Line	BitonicSort.sort()	Cost	Times
1	internalData = externalData	C_1	1
2	bitonicSort(0, internalData.length, ASCENDING, level)	C_2	1
3	return internalData	C_3	1

BitonicSort function - bitonicSort(int lo, int n, boolean dir, int level)

Line	BitonicSort.bitonicSort()	Cost	Times
1	if n > 1	C_4	1
2	m = n / 2	C_5	1
3	if level == 0	C_6	1
4	bitonicSort(lo, m, ASCENDING, level)	C_7	1
5	bitonicSort(lo+m, m, DESCENDING, level)	C_8	1
6	bitonicMerge(lo, n, dir)	C_9	1
7	else	0	0
8	Thread_a.sorter()	C_{10}	$\frac{p}{2}$
9	Thread_b.sorter()	C_{11}	$\frac{p}{2}$
10	Thread_a.start()	C_{12}	$\frac{p}{2}$
11	Thread_b.start()	C_{13}	$\frac{p}{2}$
12	Thread_a.join()	C_{14}	$\frac{p}{2}$
13	Thread_b.join()	C_{15}	$\frac{p}{2}$
14	bitonicMerge(lo, n, dir)	C_{16}	$\frac{p}{2}$

BitonicSort function - bitonicMerge(int lo, int n, boolean dir)

Line	BitonicSort.bitonicMerge()	Cost	Times

1	if n > 1	C_{17}	1
2	int m = n / 2	C_{18}	1
3	for(i = lo; i < lo+m; i++)	C_{19}	$\frac{n}{2}$
4	compare(i, i+m, dir)	C_{20}	$\frac{n}{2}$
5	bitonicMerge(lo, m, dir)	C_{21}	1
6	bitonicMerge(lo+m, m, dir)	C_{22}	1

BitonicSort function - compare(int i, int j, boolean dir)

Line	BitonicSort.compare()	Cost	Times
1	if dir == (internalData[i] > internalData[j])	C_{23}	1
2	exchange(i, j)	C_{24}	1

BitonicSort function - exchange(int i, int j)

Line	Bitonic-Sort(A).exchange()	Cost	Times
1	t = internalData[i]	C_{25}	1
2	internalData[i] = internalData[j]	C_{26}	1
3	internalData[j] = t	C_{27}	1

BitonicSort function - Sorter(int min, int max, boolean dir, int level)

Line	Bitonic-Sort(a).Sorter	Cost	Times
1	this.dir = dir	C_{28}	1
2	this.level = level	C_{29}	1
3	this.min = min	C_{30}	1
4	this.max = max	C_{31}	1

BitonicSort function - Run()

Line	Bitonic-Sort(A).Run()	Cost	Times
1	BitonicSort.bitonicSort(min, max, dir, level)	C_{32}	1

Sort

$$T_1 = C_1 + C_2(T_2) + C_3$$

BitonicSort

$$T_2 = C_4 + C_5 + C_6 + C_7(T_2) + C_8(T_2) + C_9(T_3) + C_{10}(\frac{p}{2}(T_6) + C_{11}(\frac{p}{2}(T_6)) + C_{12}(\frac{p}{2}) + C_{13}(\frac{p}{2}) + C_{14}(\frac{p}{2}) + C_{15}(\frac{p}{2}) + C_{16}(\frac{p}{2}(T_3))$$

BitonicMerge

$$T_3 = C_{17} + C_{18} + C_{19}(\frac{n}{2}) + C_{20}(\frac{n}{2}(T_4)) + C_{21}(T_3) + C_{22}(T_3)$$

Compare

$$T_4 = C_{23} + C_{24}(T_5)$$

Exchange

$$T_5 = C_{25} + C_{26} + C_{27}$$

Run&Sorter

$$T_6 = C_{28} + C_{29} + C_{30} + C_{31} + C_{32}$$

F. Bucket sort

n = number of elements
k = number of buckets
m= elements in bucket
p = number of processes

BucketSort.bucketSort(int[] a)

Line	BucketSort.bucketSort()	Cost	Times
5	min = a[0]	c1	1
6	max = a[a.length-1]	c2	1
8	addToBuckets()	c4	1
9	for(i = 0; i < NR_OF_BUCKETS; i++)	c5	k
10	insertionSort(bucket)	c6	1
11	removeFromBuckets()	c7	1

BucketSort.sort(int[] a, int level)

Line	BucketSort.sort()	Cost	Times
12	if(level == 0)	c8	1
13	bucketSort(a)	c9	1
15	addToBuckets()	c11	1
16	if(level == 1)	c12	1
17	new thread()	c13	p
18	thread.start()	c14	p
19	thread.join()	c15	p
20	else if(level == 2)	c16	1
21	new thread()	c17	p
22	thread.start()	c18	p
23	thread.join()	c19	p
24	removeFromBuckets()	c20	1

BucketSort.removeFromBuckets(int[] a)

Line	BucketSort.removeFromBuckets()	Cost	Times
25	index = 0;	c21	1
26	for(i = 0; i < NR_OF_BUCKETS; i++)	c22	k
27	for(j = 0; j < bucket.size; j++)	c23	m
28	a[index++] = bucket[i].getElement[j]	c24	1

BucketSort.addToBuckets(int[] a)

Line	BucketSort.addToBuckets()	Cost	Times
29	for (key : a)	c25	n
30	int temp = key / (LENGTH/NR_OF_BUCKETS)	c26	1
31	buckets[temp].addToBucket(key)	c27	1

BucketSortSorter(int[] a, int start, int end)

Line	BucketSortSorter	Cost	Times
38	BucketSortSorter()	0	1
39	this.a = a	c35	1
40	this.start = start	c36	1
41	this.end = end	c37	1
42	run()	0	1
43	for(i = start; i <= end; i++)	c39	k
44	if(a[i].size > 0)	c40	1
45	insertionSort(a[i].getBucket(), a[i].getCounter())	c41	1

Bucket sort

$$T_1 = c_1 + c_2 + c_4(T_3) + c_5 k(c_6 m) + c_7(T_4)$$

Sorter

$$T_2 = c_{35} + c_{36} + c_{37} + \frac{c_{39}k(c_{40}+c_{41})}{p}$$

Add

$$T_3 = c_{25}n(c_{26} + c_{27})$$

Remove

$$T_4 = c_{21} + c_{22}k(c_{23}m(c_{24})) \;=\; c_{21} + c_{22}k + n(c_{23} + c_{24})$$

G. Radix sort

n = number of elements
p = number of processes
k = radix(base 10)
d = number of digits

RadixSort.sort(int[] a, int[] b, int maxValue, int level, int length, int mod)

Line	RadixSort.sort()	Cost	Times
37	temp = new int[length]	c1	1
38	if mod != 100	c2	1
39	radixMSD(maxValue-1, a, temp, 0, a.length, length, mod)	c3	1
40	else	0	1
41	radixMSD(maxValue-2, a, temp, 0, a.length, length, mod)	c5	1
42	sortParallel(level, a, b, temp, maxValue, length)	c6	1

RadixSort.sortParallel(int[] a, int[] b, int[] c, int level, int nrOfDigits)

Line	RadixSort.sortParallel()	Cost	Times
1	if level == 0	c7	1
2	for i = 0, i < nrOfDigits, i++	c8	d
3	radixLSD(i, a, b, 0, a.length)	c9	1
4	else if level == 1	c10	1
5	new thread()	c11	p
6	thread.start()	c12	p
7	thread.join()	c13	p
8	else if level == 2	c14	1
9	new thread()	c15	p
10	thread.start()	c16	p
11	thread.join()	c17	p

RadixSort.radixMSD(int[] a, int[] b, int radix, int length, int mod, int start, int end)

Line	BucketSort.radixMSD()	Cost	Times
12	count = new int[length];	c18	1
13	emp = new int[a.length]	c19	1
14	for i = start, i < end, i++	c20	n
15	test = (int) (a[i]/ Math.pow(10, radix)) % mod	c21	1
16	count[test]++	c22	1
17	for k = 1, k < length, k++	c23	k
18	count[k] += count[k-1]	c24	1
19	for i = end-1, i >= start, i--	c25	n
20	test = (int) (a[i]/ Math.pow(10, radix)) % mod;	c26	1
21	temp[--count[test]] = a[i]	c27	1
22	for i = start, i < end, i++	c28	n
23	a[i] = temp[i]	c29	1

RadixSort.radixLSD(int radix, int[] a, int[] b, int start, int end)

Line	BucketSort.radixLSD()	Cost	Times
24	if end <= (start+1)	c30	1
25	return	c31	1
26	count = new int[length]	c32	1
27	for i = start, i < end, i++	c33	n
28	int test = (int) (a[i]/ Math.pow(10, radix)) % 10	c34	1
29	count[test]++	c35	1
30	for k = 1, k < 10, k++	c36	k-1
31	count[k] += count[k-1]	c37	1
32	for i = end-1, i >= start, i--	c38	n

33	int test = (int) (a[i]/ Math.pow(10, radix)) % 10	c39	1
34	b[--count[test]+start] = a[i]	c40	1
35	for i = start, i < end, i++	c41	n
36	a[i] = b[i]	c42	1

RadixSortSorter(int[] a, int[] b, int[] c, int nrOfDigits, int start, int end)

Line	RadixSortT1Sorter	Cost	Times
43	RadixSortT1Sorter()	c43	
44	this.a = a	c44	1
45	this.b = b	c45	1
46	this.c = c	c46	1
47	this.nrOfDigits = nrOfDigits	c47	1
48	this.start = start	c48	1
49	this.end = end	c49	1
50	run()	c50	1
51	for i = 0 i < nrOfDigits-1, i++	c51	d
52	for j = start, j < end, j++	c52	k/p
53	if j != 0	c53	1
54	RadixSort.radixLSD(i, a, b, 0, c[j])	c54	1
55	else	0	1
56	RadixSort.radixLSD(i, a, b, c[j-1], c[j])	c56	1

MSD

$$T_1 = c_{18} + c_{19} + c_{20}n(c_{21} + c_{22}) + c_{23}k(c_{24}) + c_{25}n(c_{26} + c_{27}) + c_{28}n(c_{29})$$

LSD

$$T_2 = c_{30} + c_{31} + c_{32} + c_{33}n(c_{34} + c_{35}) + c_{36}k(c_{37}) + c_{38}n(c_{39} + c_{40}) + c_{41}n(c_{42})$$

Parallel

$$T_3 = c_7 + c_{10} + c_{14} + p(c_{11}(T_4) + c_{12} + c_{13})$$

Sorter

$$T_4 = c_{44} + c_{45} + c_{46} + c_{47} + c_{48} + c_{49} + c_{52}(\frac{k}{p})$$

$$T_5 = c_{51}d(c_{53} + c_{54}(T_2))$$