EFFECT OF MULTI-CORE PROCESSORS ON SOME SORTING AND SEARCHING ALGORITHMS

BY

ABBAS MUHAMMAD RABIU

(SPS/13/MCS/00017)

SUPERVISED

BY

MAL. MANSUR BABAGANA

BEING A MASTERS DISSERTATION SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE, FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY, BAYERO UNIVERSITY, KANO.

OCTOBER, 2016

Approval

This dissertation has been examined and approved for the award of the Master of Science, M.Sc. Computer Science.

External Examiner

Name…….sign……date……

Internal Examiner

Name…….sign……date……

Supervisor

Name…….sign……date……

Representative of the Board of Postgraduate School, BUK

Name…….sign……date……

Head of department Computer Science (H.O.D)

Name…….sign……date……

Declaration

I hereby declare that this dissertation has been carried out by me (Abbas Muhammad Rabiu (SPS/13/MCS/00017) and therefore it is a product of my own efforts and determinations. This work has been carried out under the supervision of Mal. Mansur Babagana and has not been presented elsewhere for the award of any certificate. All sources have been duly cited.

Abbas Muhammad Rabiu (SPS/13/MCS/00017)

Sign……. Date……

Abstract

Information retrieval is the most fundamental requirement for any kind of computing application and which requires sorting and searching operations to be performed concurrently from massive databases implemented by various data structures. In addition to its main job, sorting is often required to facilitate some other operation such as binary searching, merging and normalization. Numbers of algorithms are developed for sorting and searching among which quick sort, linear search and binary search are the most popular algorithms. In this paper researcher has made efforts to compare the speed and run time of three different sorting algorithms; namely: QuickSortSequential, QuickSortParallelNaive, Fork/Joint and two searching algorithms: Linear and Binary search tested on single and multi-core machines using concurrency tools provided by Java. The results showed that Fork/Join emerged as the best when sorting larger array elements while QuickSortSequential exhibits best performance with smaller array size. Binary search has better performance than linear search and quick sort algorithms. The result also shows that dual-core improves the performance of these algorithms better than single-core machine.

Acknowledgement

All gratitude is due to Allah the creator of heaven and the earth, mankind and jinn; who enabled me to complete this work. Special thanks go to my supervisor Mal. Mansur Babagana who has been providing me with valuable feedback along the way and for being patient with me throughout the period of this dissertation. I also acknowledge the efforts of the external examiner; Dr. Aminu Mohammed for taking his time to go through this text to highlight all the necessary corrections so as to ensure the success of this work. Thanks also go to my internal examiner, Malama Hadiza Umar for her motherly supports and kind gesture towards me. My gratitude also goes to all my lecturers in the Faculty of Computer Science and Information Technology, Bayero University, Kano (BUK) and my fellow students in this great department. Finally, I must acknowledge the prayers and supports given to me by my parents and my uncle; Alhaji Mohammed Ibrahim mni, O.F.R. Makama of Ringim and District Head of Kanya-babba. Special thanks to my loving wife Amina Ishaq for her endurance and endless prayers throughout the period of my studies. May Allah make Aljannatul-firdaus our final home. Ameen.

Dedication

I have dedicated this work to my late father whom I lost recently; Mallam Muhammad Rabiu Ladan. May Allah forgive him his shortcomings and grant him Aljannatur Firdaus. Ameen.

Table of contents

List of Tables

List of Figures

CHAPTER ONE

(INTRODUCTION)

1.1 Background

A program can be called concurrent [62] if it is divided in such a way that two or more threads can progress at some time. This does not mean that they have to progress simultaneously, but that they can be swapped in and out by the operating system on a single core. [62] Parallel on the other hand is when two or more threads progress at the same time, requiring multiple cores and where each thread is assigned to a separate processor core. The Java class 'thread' is the whole foundation for all Java concurrency frameworks. This class makes it easy for the developer to create and execute threads. Initially a thread is identical to normal Java classes, but with the exception of having a runnable task which can be executed concurrently with the main program. When these threads are executed, the Java Virtual Machine (JVM) and the operating system (OS) decide which processor cores will be applied [63].

Before version 5 of Java was introduced the only way to synchronize threads was through the low-level concurrency control mechanisms such as synchronized, volatile, wait(), notify() and notifyAll(). They are all difficult to use correctly and the potential for common concurrent threats like deadlock and starvation is high

[63]. Java 5 (originally known as Java 1.5) changed this by including a new package named java.util.concurrent (including two sub-packages atomic and locks), which allows a more high-level synchronization on threads. These packages have been updated and improved by including new classes and interfaces in Java 6 and the current version 8, giving the developer more tools to use with concurrent programming [62].

With the continual growth in number of cores on the Central Processing Unit (CPU), developers will need to focus more and more on concurrent programming to get the desired performance boost that in the past have come logically with the increased clock-rate.

Today there are different libraries and mechanisms for synchronization and parallelization in Java, and this dissertation will attempt to test the speed and run time of some types of sorting and searching algorithms on machines with single multi-core processors using the concurrent tools provided by Java. The Central Processing Unit (CPU) is one of the main components we have in computer systems, and may be looked at as the brain in our computer. It is the CPU which performs the necessary processing and calculations of instructions for the computer.

Looking back at the years that has gone we see that the CPU originally consisted of only one single processing core, where the performance throughout the years have had an exponential growth because of a combination of Moore's law and the increasing of the clock frequency speed (i.e. more instructions can execute per second). But in the later years we have seen another trend rise, the multi-core CPU. This trend is due to the limitation on of how high the clock frequency could go before having a huge impact on energy consumption and heat production compared to the performance gain. And that is why the manufactures started producing CPUs with multi-core architecture, thus continue to increase the overall system performance. So while a single core CPU only could execute a single sequence of instructions, multi-core could execute many sequences [21, 22]. The introduction of Hyper-Threading was brought to the market appearing on the Pentium 4 processor in 2002. In short the Hyper- Threading is to physically duplicate certain sections of the processor (i.e. architectural state), but not the main execution resources. Resulting in giving a processor core the ability to schedule and assign resources to two threads at once, but only compute one at any given time, and with this theoretically increasing the performance. Intel claimed that they would get a performance boost around 15 to 30% compared to other non Hyper- Threaded CPUs and only increase the size of the die with 5% [62].

Therefore, with the introduction of multi-core the developers have had to change their mindset when writing program code. Not only would they have to divide the program in such a way that each part could run concurrently, they also had to think about synchronization when more than one core have access to shared data. Taking advantage of multi-core to get the desired performance increase, these concurrent parts will have to run in parallel and all necessary sequential fraction of the program needs to be kept to a minimum (Amdahl's law).

1.2 Problem Statement

Sorting is a problem that mostly arises in computer programming. Many sorting algorithms have been developed while some existing ones have been improved upon; all to make them run faster. According to [66] efficiency of an algorithm can be measured in terms of execution time (complexity) and the amount of memory required. Therefore concurrent programmers are faced with the challenges in choosing which algorithm and concurrent tools would be best used in developing their concurrent software. Therefore this dissertation uses single and multi-core processors to test the speed and running time of three types of quick sort algorithms: QuickSortSequential, QuickSortParallelNaive, QuickSortForkJoin and two searching algorithms: Linear and Binary search. The time it takes to sort variety of array will be compared to see how well they perform against each other.

1.3 Aim and objectives

The aim of this dissertation is to measure the speed and running time of quick sort and search algorithms using machines with different number of cores to achieve the following objectives.

1. To implement sequential and parallel versions of quick sort and search algorithms using java and to compare the results.
2. To determine whether sorting an integer array is faster or searching an integer array.
3. To determine if the increase in a number of core in a CPU has any effect on the performance of concurrent programs.

1.4 Significance of the study

1. This research will be useful to students and programmers because it will provide general introduction to currency particularly in java.
2. Students taking course in design and implementation of parallel algorithms will find this project helpful as it will provide a practical implementation of parallel algorithms, their speed up and running time.
3. To educate users especially programmers that machine with more number of CPU core is a better choice because it improves performance of a running programs.

1.5 Scope and limitations

This work used three quick sorts namely: QuickSortSequential, QuickSortParallelNaive, QuickSortForkJoin and two searching algorithms: Linear search, Binary search and the concurrency mechanisms in java to measure their speed and running time. The limitations are:

1. This dissertation uses Java as a programming language to implement the five algorithms of choice and therefore does not consider how these algorithms are implemented in other programming languages.
2. Also all the concurrency tools used are those provided by java only.
3. Only speed and running time were measured.

1.6 Dissertation Outline

CHAPTER 1: INTRODUCTION

A brief introduction and background of the dissertation is given here. Aim and objectives, problem statement, motivation, significance of the study, scope and limitations for this project are also given.

CHAPTER 2: LITERATURE REVIEW

This chapter will review some of the related literatures. The review is divided into three sections. Section 2.1 talks about sorting and searching algorithms, section 2.2 discusses multi-core processors and finally in section 2.3 literatures of concurrency are reviewed.

CHAPTER 3: METHODOLOGY

This chapter explains how the data used for testing are structured and generated. The method used (benchmarking) to implement different algorithms is also explained. A brief analysis of the two algorithms used is also given, followed by specification of the hardware and software that is used for producing and testing results in this dissertation.

CHAPTER 4: RESULTS AND DISCUSION.

To test out the various frameworks and some of its classes in java using single and multi-core machines, some sorting and searching algorithms with different characteristics are implemented in this chapter to be able to try out some varieties of concurrency tools. This would let us see the effect of different parallel implementations of the algorithms thereby getting some comparable results.

CHAPTER 5 – SUMMARY, CONCLUSSION AND RECOMMENDATION:

This chapter summarizes the whole work. General conclusion is also drawn. Precautions to be taken when constructing parallel implementation of sequential algorithms has also been discussed, this is followed by recommendations for further research on the topic.

CHAPTER TWO

(LITERATURE REVIEW)

This chapter reviews some of the related literatures. The review is divided into three sections. Section 2.1 talks about sorting and searching algorithms, section 2.2 discusses multi-core processors and finally in section 2.3 literatures of concurrency are reviewed.

2.1 Sorting and searching algorithms

The comparison of performance between selection sort, bubble sort, and insertion sort in terms of the time required to sort a list with random data is presented in [1] which represents the average case. Enhancement of the selection sort is achieved by avoiding unnecessary comparisons to find the maximum in each pass of the classical algorithm. A stack is used to store the locations of the past or local maximums to accelerate the process. The measurements are taken on a 2.4 GHz Intel Core 2 Duo processor with 2 GB 1067 MHz DDR3 memory machine with OS X version 10.6.8 platform. The substantial improvement of around 220-230% is achieved. However, the work did not consider other popular sorting algorithms such as the quick sort which can give better performance with increased number of processor-core.

Moreover, enhancement of two new sorting algorithms, Enhanced Selection Sort (ES) and Enhanced Bubble Sort (EBS) algorithm is presented in [2] performance enhancement is by extending the key so that comparisons between two objects with other equal keys are decided using the order of the entries in the original data order as a tie-breaker. The result showed that EBS is definitely faster than BS, since BS performs O(n₦2) operations but EBS performs O(nlogn) operations to sort n elements. However, another sorting algorithm called Bingo sort was proposed in [3]. It takes advantage of having many duplicates in a list. After finding the largest item, another pass is performed to move any other items equal to the maximum item to their final place. This was found to be faster than EBS. Another approach was taken in [4] Exact sort is another new variation, which locates the elements to their sorted positions at once [4]. To find the exact positions of elements it counts the smaller elements than the element which is indented to be located. It changes elements positions just once, to directly their destination. It is highly advantageous if changing positions of elements is costly in a system. It makes too many comparisons to find positions of elements and thus is highly disadvantageous if comparing two elements is costly, which the common case is. A new approach is presented in [5] that works on the principle rather than swapping two variables using third variable, a shift and replace procedure should be followed, which takes less time as compared to swapping.

The inner loop of insert sort can be simplified by using a sentinel value as discussed in [6]. A bidirectional approach like the binary insertion sort is presented in [7], which achieves time complexity of O(n₦1.585) for some average cases. When people run insertion sort in the physical world, they leave gaps between items to accelerate insertions. Gaps help in computers as well. This idea of gapped insertion sort discussed in [8] has insertion times of O(log n) with high probability, yielding a total running time of O(nlogn) with high probability. An end-to-end bi-directional sorting algorithm is proposed to address the shortcomings of the bubble sort, selection sort, and insertions sort algorithms [9]. Performance comparison of sorting algorithms on the basis of complexity is presented in [10]. However, every sorting algorithm can undergo a fine tuning with the intelligence aspects discovered so as to gain significant reduction in complexity values. It shows that the choice of sorting algorithm is not a straight forward matter, as a number of issues may be relevant. It may be the case that an O (n*n) algorithm is more suitable than an O(nlogn) algorithm. The efficiency gain will not go beyond O (nlogn), but hopeful enough to reduce complexities by using intelligent tactics there could be a smooth transition from quadratic complexity to linear one observed in comparative sorts due to intelligently using linked lists instead of arrays to hold data. This drastically reduces the space requirement since no need to

swap the data as we need to change the pointers only thereby keeping the contents of nodes the same.

[11] Re-evaluated the method for managing buckets held at leaves & shows better choice of data structures further improves the efficiency, at a small additional cost in memory. For sets of around 30,000,000 strings, the improved burst sort is nearly twice as fast as the previous best sorting algorithm. Discussed in [12] is a suggestion to the detailed implementation, combining the most effective improvements to Quick sort, along with a discussion of how to implement it in assembly language. It is wide applicability as an internal sorting method which requires minimal memory. Solution to the load imbalance problem present in parallel radix sort is discussed in [13]. Redistributing the keys in each round of radix, each processor has exactly the same number of keys, thereby reducing the overall sorting time to improved performance. Load balanced radix sort is currently the fastest internal sorting method for distributed-memory based multiprocessors. However, as the computation time is balanced, the communication time becomes the bottleneck of the overall sorting performance. The proposed algorithm pre-processes the key by redistribution to eliminate the communication time. Once the keys are localized to each processor, the sorting is confined within processor, eliminating the need for global redistribution of keys & enables well balanced communication and computation across processors.

Experimental results with various key distributions indicate significant improvements over balanced radix sort.

A new algorithm called Sequential Counting Split Radix sort is proposed in [14] (SCS-Radix sort). The three important features of the SCS-Radix are the dynamic detection of data skew, the exploitation of the memory hierarchy and the execution time stability when sorting data sets with different characteristics. They claim the algorithm to be 1:2 to 45 times faster compare to Radix sort or quick sort.

$\Omega(n^2)$ in the worst-case [15]. However, with proper precautions, worst-case behavior is very unlikely. In their work [16] stated that Quicksort being such a popular sorting algorithm, there have been a lot of different attempts to create an efficient parallelization of it. The obvious way is to take advantage of its inherent parallelism by just assigning a new processor to each new subsequence. This means, however, that there will be very little parallelization in the beginning, when the sequences are few. Presented and evaluated [17] several optimized and implemented techniques for string sorting. Forward radix sort has a good worst-case behavior. Experimental results indicated that radix sorting is considerably faster (often more than twice as fast) than comparison-based sorting. It is possible to implement a radix sort with good worst-case running time without sacrificing average-case performance. The implementations are competitive with the best previously published string sorting programs. Illustrated in [18] is the importance

of reducing misses in the standard implementation of least-significant bit first in (LSB) radix sort, these techniques simultaneously reduce cache and TLB misses for LSB radix sort, all the techniques proposed yield algorithms whose implementations of LSB Radix sort & comparison- based sorting algorithms.

Performance analysis of sorting and selection algorithms is presented in [19] by measuring their speed. The algorithm of Quick sort is rather complex and it will give very poor running time in case of large sets of data. But the performance of both algorithms is worse than the lower bound run time of comparison sort O(nlogn). However, with the divide and conquer nature of some quick sort such as Fork/Join when parallelized on multi-core machine, a large set of data of data can be handle without compromising the performance. This is the subject of this dissertation.

[56] Proposes a modification to the traditional binary search algorithm in which it checks the presence of the input element with the middle element of the given set of elements at each iteration. Modified binary search algorithm optimizes the worst case of the binary search algorithm by comparing the input element with the first & last element of the data set along with the middle element and also checks the input number belongs to the range of numbers present in the given data set at each iteration there by reducing the time taken by the worst cases of binary search algorithm. The result is that he modified binary search improves the execution time

vastly over traditional binary search. The algorithm is comparatively more efficient as it eliminates unnecessary comparisons at the preliminary stage itself. However, this algorithm can even be extended to include the String domain.

Presented in [57] the researcher has made efforts to compare linear and binary search algorithms to implement on various data structures and to find out the solution to implement binary search on linked linear list. This paper also analyzes both the algorithms at some extent for the applicability and execution efficiency. The paper also analyzes few data structures to implement these algorithms. At last based on the linear search and binary search algorithms, one algorithm is designed to function on linked linear list. The result is that the binary search is more efficient searching technique than linear search but insertion of an element is not efficient as it requires arranged elements in specific order. Further it is also possible to apply binary search on linked list by making necessary modifications to original binary search algorithm. It is concluded that the selection of searching algorithm can be based on the data structure on to which it is applied and which operations are required more. Balance is required between search and maintenance of a data structure.

Moreover, presented in [58] is an investigation of the effect of parameters n and p of a Binomial distribution input on the number of comparisons in linear search and binary search. Using factorial experiment, it is observed that both the main effect n

and p and the interaction effects n*p are highly significant for linear and significant but comparatively less for binary search. The result clearly suggests that apart from the size of the input, the parameters of the input distribution need also be taken into account to explain the behavior of certain search algorithms. In an earlier work on parameterized complexity, in [59] factorial experiment was used to explain software complexity for insertion sort. The same authors have used factorial experiments with a response surface design in [60] to examine the nature of the fast and popular quick sort.

Another different approach is described in [61]. This paper critically conducted experiments on the execution time of linear search, binary search without inclusion of sorting time, and binary search with the inclusion the sorting time. This paper compares the execution time of searching for an integer number among a list of integer numbers, using linear and binary search algorithms. Existing literatures have it that binary search, without considering the sorting time incurred in the binary searching. The emergence of this paper has establishes the fact that sorting time of the numbers before a binary search is conducted, is never considered. If sorting time is added to the binary search time then, execution time of binary search is not faster than linear search technique.

Therefore in 2014, another approach was taken to improve the performance of a binary such. Presented in [100] was a proposed modification to the traditional

binary search algorithm in which it checks the presence of the input element with the middle element of the given set of elements at each iteration. Modified binary search algorithm optimizes the worst case of the binary search algorithm by comparing the input element with the first & last element of the data set along with the middle element and also checks the input number belongs to the range of numbers present in the given data set at each iteration there by reducing the time taken by the worst cases of binary search algorithm. The modified binary search improves the execution time vastly over traditional binary search. The result shows that the algorithm is comparatively more efficient as it eliminates unnecessary comparisons at the preliminary stage itself. However, this improved binary algorithm did not consider string domain which might not give the same result.

Additionally in 2015, efforts were made to compare both linear and binary search algorithms to implement on various data structures and to find out the solution to implement binary search on linked linear list [101]. The paper also analyzed both the algorithms at some extent for the applicability and execution efficiency and also analyzed the few data structures to implement these algorithms. At last based on the linear search and binary search algorithms, one algorithm is designed to function on a linked linear list. It result shows that the binary search is more efficient searching technique than linear search but insertion of an element is not efficient as it requires arranged elements in specific order. It was concluded that

the selection of searching algorithm can be based on the data structure on to which it is applied and which operations are required more. However balance is required between search and maintenance of a data structure.

2.2 Single and Multi-core processor

A study of various factors affecting performance of multi-core processors was carried out by [20]. Since commercial applications have abundant thread-level parallelism, commercial servers were designed as multiprocessor systems or clusters of multiprocessors to provide sufficient throughput. While traditional symmetric multiprocessors (SMPs) can exploit thread-level parallelism, they also suffer from a performance penalty caused by memory stalls due to cache misses and cache-to-cache transfers, both of which require waiting for long off-chip delays. Several researchers have shown that the performance of commercial applications and data base applications in particular, is often dominated by sharing misses that require cache-to-cache transfers [21]. To avoid these overheads, architects proposed several schemes to integrate more resources on a single chip. Researchers have shown that chip-level integration of caches, memory controllers, cache coherence hardware and routers can improve performance of online transaction processing workloads by a factor of 1.5 [21]. Simultaneous multithreading designs allow the processor to execute several contexts (or threads) simultaneously by adding per-thread processor resources. This approach also

improves the performance of database applications compared to a superscalar processor with comparable resources [21, 22]. The trend towards more integration of resources on a single chip presented in [20] is becoming more apparent in CMP designs where multiprocessor systems are built on a single chip. Chip multiprocessor (CMP) systems can provide the increased throughput required by multi-threaded applications while reducing the overhead incurred due to sharing misses in traditional shared-memory multiprocessors. A chip multiprocessor design is typically composed of two or more processor cores (with private level-one caches) sharing a second- level cache. The results showed that performance increase is achieved but waste more power and consumed battery life due to higher computational speed and incompatibility with some software [21].

CMPs in various forms are becoming popular building blocks for many current and future commercial servers. The increasing number of processor cores on a single chip increases the demand on two critical resources: the shared cache capacity and the off-chip pin bandwidth.

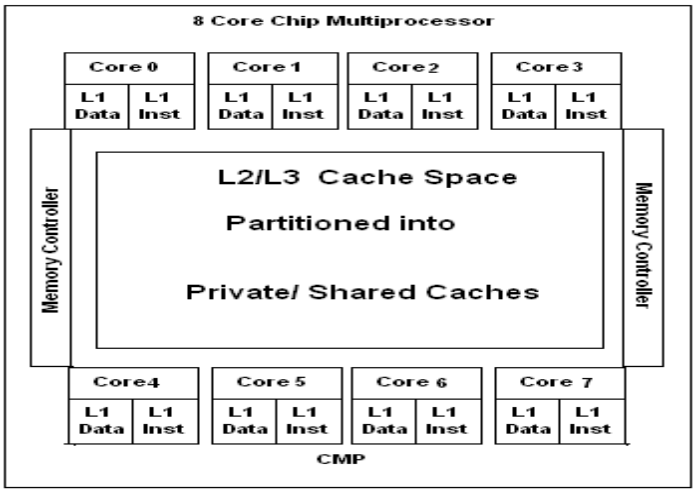


Figure 2.1 Single Chip Multiprocessor (CMP)

Performance evaluation studies carried out on AMD dual-core and Intel dual-core processor to know which of the processor has better execution time and throughput was performed in [22]. The architecture of AMD and Intel duo-core processor was studied SPEC CPU2006, benchmarks suite was used to measure the performance of AMD and Intel duo core processors. The overall execution and throughput time measurement of AMD and Intel duo core processors were reported and compared to each other. Results showed that the execution time of CQ56 Intel Pentium Dual-Core Processor is about 6.62% faster than AMD Turion II P520 Dual-Core Processor while the throughput of Intel Pentium Dual-Core Processor was found to be 1.06 times higher than AMD Turion (tm) II P520 Dual Core Processor. Therefore, Intel Pentium Dual-Core Processors exhibit better performance probably due to the following architectural features: faster core-to-core communication, dynamic cache sharing between cores and smaller size of level 2

cache. The disadvantage of this higher computation capacity is that more power is wasted and consumed battery life.

In spite of the disadvantages of multi-processor mentioned in [22, 23], description about some challenges and future prospects of multi-core technology was described in [23] using benchmarking method. The researchers have it that, in a decade or so from now when the full power of high performance computing and parallel processing is available to computer users everywhere and it might be possible to hold the power of a computer with hundreds of execution cores in the palm. The result obtained is that Multi-core processors are the future of computing. However, the researchers did not consider how Multi-core processors can enhance the performance of concurrent programs.

Moreover, another overview to multi-core processors, multi-core processor parallelism and performance measurement for multi-core Central Processing Units (CPUs) is presented in [24]. Factors affecting the performance of single and multi-core memory, I/O bandwidth, inter-core communications were measured, CPU clock speed and cache coherence. The result shows that numbers of cores affect the CPU performance as multi-core architecture as workload is divided between the cores. The conclusion is that different approaches used in multi-core CPU performance analysis were taken depending on the purpose of the study. We are expecting to see new approaches in multi-core CPU performance analysis as multi-

core CPU production increases to new levels. Therefore, this dissertation will take a new approach to measure the performance increase of concurrent programs using some algorithms on Multi-core CPUs.

A review on trends in multi-core processor based on cache and power dissipation is presented in [45]. The review concentrated on two parameters such as cache coherence and heat dissipation implementation of robust cells and non-robust cells, a multi-core architecture mechanism is improved by 17% and reduced dynamic is achieved in cache about 50% by simple error detection and correction mechanism, which makes the system performing better. The Multi-core architecture mechanism is improved, with the implementation of robust cells and non-robust cells by simple error detection and correction mechanism, which makes the system performing better. Presented in [46] was a discussion about the high-level trends in computer architecture and the related drivers. However, a survey carried out microprocessor, graphics, memory, and I/O subsystems in emerging computing technology that will drive increased demands for power. As the industry improves from one process dimension to a smaller dimension, the results would be provided in several ways. Thereby, users can go to the same device with smaller die sizes, higher frequency, lower power, and higher yields. In some cases, the silicon developers in designing of improvement in die size of CMPs allow additional features into a device. This results in better functionality, even with the smaller

process geometry. The continuous process of integrating new features and technology, results in the current generation of computers having Multi-core with better performance. Hence this project will take advantage of these multi-core machines to measure the performance of concurrent programs on different machines having different number of cores.

Furthermore references [47, 48, 49] Presented a single-core and multi-core processor architecture for health monitoring systems where slow bio-signal events and highly parallel computations exist. The single-core architecture is composed of a processing core (PC), an instruction memory (IM) and a data memory (DM), while the multi-core architecture consists of PCs, individual IMs for each core, a shared DM and an interconnection crossbar between the cores and the DM. These architectures are compared with respect to power vs performance trade-offs for a multi-lead electrocardiogram signal conditioning application exploiting near threshold computing. The results show that the multi-core solution consumes 66% less power for high computation requirements (50.1 *MOps/s*), whereas10.4% more power for low computation needs (681 *kOps/s*). However, the researchers did not consider the effects of their performance gain in concurrency aspect.

[50] Presented a new ultra low energy processor with low voltage operations for wireless monitoring systems. They optimized the standby power consumption of the processor with the help of a new low leakage memory, memory size and

instruction set adjustments, and power gating. Leakage and dynamic power consumption comparison for various workload requirements were measured. However, the main issue with low-voltage operation is the performance loss, which, for a given processing requirement, can limit the degree of use of voltage-scaling. Parallel computing using multiple cores can alleviate this issue, provided that the algorithms to be executed can be parallelized (here, a sequential Quick sort and Linear search will be parallelized in this dissertation). To this end, [51] proposed a Near Threshold Computing (NTC), cluster-based multi-processor architecture with a shared cache that operates at a higher supply voltage to be able to serve multiple cores at the same time. In another study [52] introduced a sub/near threshold co-processor for low energy mobile image processing using architecture level parallelism to compensate the performance loss. Finally [53] proposed a massively parallel stream processor operating in NTC to achieve 1 Giga-operations per second with 1 mW of total power consumption. However the high power consumption is a disadvantage and weak battery life.

A more through research on Multi-core with a description of the basic concept of multiprocessor, advantages and a sample of Dual-core Processors in Intel and AMD is presented in [53]. The article also surveys the most important aspects of challenge in this method. However, before multi-core processors the performance increase from generation to generation was easy to see, an increase in frequency.

This model broke when the high frequencies caused processors to run at speeds that caused increased power consumption and heat dissipation at detrimental levels. Adding multiple cores within a processor gave the solution of running at lower frequencies, but added interesting new problems. Multi-core processors are architected to adhere to reasonable power consumption, heat dissipation, and cache coherence protocols. However, many issues remain unsolved. In order to use a multi-core processor at full capacity the applications run on the system must be multithreaded. There are relatively few applications (and more importantly few programmers with the know-how) written with any level of parallelism. And finally the memory systems and interconnection networks also need improvement. Therefore this dissertation will used multithreaded approach to divide the tasks amongst different threads to improve performance of concurrent program.

2.3 Concurrency in java

Java was created as a concurrent language right from the beginning. Its concurrency features, however, did not reflect the state of the art in concurrent object-oriented programming at that time [71]. A rather conservative approach was taken, featuring explicit threading and a version of monitors that was even less safe than previous monitor concepts. Concerning the interplay between concurrency and inheritance, Java is a prime example of a language for which the infamous inheritance anomaly is the rule rather than the exception. And regarding C#,

Microsoft's answer to Java, one is led to wonder why the designers of the language did not take a different road [71].

Java tries to alleviate the problems of concurrent programming by library support. For instance, a properly synchronized version of an (unsynchronized) java.util. TreeSet object is created by Collections.synchronized-SortedSet (new TreeSet(...)). The java.util.concurrent classes adopted for Java 1.5 support typical patterns and solutions for common concurrent programming problems [72, 73]. Several Java extensions meant to support higher-level concurrency models have been suggested in [74, 75, 76, 77, 78] and others. In addition, the aspect-oriented community, viewing synchronization just as an aspect to be added to business logic, has suggested weaving synchronization code into a given piece of sequential code [79]. A similar approach is encapsulating objects in what has been called composition filters [80], synchronization rings [81] or qualifying/ qualified types [82, 83,84].

However, different approach is taken in this dissertation using sorting and searching algorithms, by avoiding synchronization to investigate some of the concurrency tools initially introduced in Java 5 using multi-core machines to show the performance benefits of using multi-core in concurrent programming.

Discussed in [85] is the issue that needs to be addressed when designing a concurrency control techniques (CCT) technique for Mobile databases, analyses

the existing scheme of CCT and justify their performance limitations. A modified optimistic concurrency control scheme is proposed which is based on the number of data items cached, amount of execution time and current load of the database server. Experimental results show performance benefits, such as increase in average response time and decrease in waiting time of the transactions. However, with the presence of inadequate bandwidth, small processing capability, unreliable

Communication and mobility the method employed in his paper cannot work effectively. Therefore another approach is presented in [86]. In this paper an optimistic concurrency control strategy using on-demand multicasting is proposed for mobile database environments which guarantees consistency and introduces application-specific conflict detection and resolution strategies of concurrent program. The simulation results specify increase in system throughput by reducing the transaction abort rates as compared to the other optimistic strategies proposed in [85]. The weakness of this scheme is that there could be a possibility that the transaction which arrived quiet early might not get executed because the other mobile hosts are executing faster. To solve this problem, a new approach was described in [87, 88]. However, these papers showed a technique of locking with non-exclusive lock based on time out. In reference [88], a dynamic timer was used to solve the problem of transactions' blocking. In this method, transactions should be finished in particular time, otherwise they would abort. In [89] a method for

increasing the efficiency of [88] is provided; transactions which don't finish in specified time and are near to final of transaction wouldn't be aborted. They are allowed to continue execution concurrently for specified time. In methods basing on timer (time out) problems like long connection of mobile client with Server, estimated the amount of timer, and the length of transaction, do exist. Because of wrong estimation of the required time for completion of transactions, these problems could result in incorrect abortion of transactions. This suffers from the problem of frequent rollbacks due to regular expiry of the timer and wastage of computation.

Therefore to solve the problems in [89] a protocol based on AVI is proposed in [90]. This method has still the problem of transaction blocking and computational overhead. This problem has been mentioned in reference [91]. Multi-version concurrency control based on MV2PL protocol and timestamp being introduced in [92] are in fact an extension of method [93]. In [94, 95], combination of optimistic and pessimistic is performed according to the semantic of operators to solve problems in [91].

Additionally, a lasting solution to the problems mentioned in [91, 94, 95,] a thorough concurrent program testing is proposed by Bruening [26]. Bruening approaches testing concurrent programs to determine correctness and performance by using schedule-based execution in [26, 27]. However, different approach using

a lockset algorithm based on Eraser is presented in [28], this framework ensures that the program is race-free. In parallel, the framework then identifies the critical operations in the program where one thread can have an impact on the computation of other threads, and if necessary, the framework re-executes the program from that point on with a different scheduling choice. As long as the program is race-free, executing all possible arrangements of program blocks delimited by these critical operations is sufficient for covering all possible concurrent behaviors. The result is a consistent data usage and faster execution time.

Bruening's work focuses on testing entire applications to see the performance gain when they are nearly complete; in essence, the framework allows acceptance testing for concurrent programs. To do this, the framework has to have the ability to "roll back" to a previous point, since re-executing the entire concurrent program to that point may be too expensive or even impossible. The emergence of "Extreme Programming" [29, 30], however, has put a greater emphasis on testing smaller program units at an earlier stage of the development process. Since unit tests are short and predictable, it seems like the dynamic techniques of schedule-based execution developed by Bruening and the dynamic race detection by Savage et al may be easier to utilize with unit tests to determine the performance of concurrent programs. There are several frameworks available that support unit testing to determine the performance of a program, JUnit [31] and TestNG [32] for Java, but

none of these frameworks takes advantage of multi-core machines to test the performance of their concurrent programs. Without using multi-processor system to determine the performance of concurrent software, applying the research on schedule-based execution only is a compromised endeavor.

Other approaches to ensuring program correctness, even in concurrent situations, use specialized type systems [33, 34, 35, 36]. The benefits of a highly developed type system, though, are often paid for by greater difficulties in implementing the desired concurrent algorithms; complex types can become hurdles that the programmer needs to overcome. Furthermore, these types of systems are often experimental and not close to being integrated into a language commonly used by the industry, such as C, C++ or Java.

Another approach to simplifying concurrent software with an improved performance completely eschews locks: These techniques are typically grouped as lock-free or wait-free algorithms presented in [36, 37, 38]. At the core, these algorithms share an optimistic notion about the work they perform; each thread assumes it will succeed without interference from other threads. Without the help of locks, the thread performs all of its work, and in a final step uses a device like an atomic N-way compare-and-swap to produce either change at a large scale or a failure due to interference. In that latter case, the thread will redo its work [39, 40]. However, more time is spent comparing and re-doing of threads work, hence the

need for a faster approach using unsynchronized concurrency with no shared data and multi-core to improve the running time of concurrent applications; this is the subject of this dissertation.

Libraries providing some lock-free algorithms are available for many platforms (Java 5.0 introduced the java.util.concurrent package, for example) [25] to ensure synchronization and good performance of concurrent program. However, lock-free algorithms need to be re-developed for each standard data structure to which we have become accustomed, and not all data structures are supported yet.

While using lock-free algorithms may be easier than using one with locks, designing a lock-free algorithm is a complex task. We doubt locks can be completely eliminated in application programs if concurrency issues such as deadlock and starvation is to be eliminated. Therefore, the program developed for this dissertation does not used shared data thereby helping the developer of lock-free concurrent algorithms.

Tools for detecting and fixing bugs in concurrent program is presented in [41] using concurrency bugs pattern detector and fixers. However, the work in [42,43, 44] improves the state of the art in static analysis for detecting concurrency bug patterns by looking for instances of the bug patterns inter procedurally and

employing context sensitive points-to analysis. The result of all these error detection methods is an improved concurrent program that runs faster.

CHAPTER 3

(METHODOLOGY)

This chapter explains how the data used for testing are structured and generated. The method used (benchmarking) to implement different algorithms is also explained. A brief analysis of sorting and searching algorithms used is also given, followed by specification of the hardware and software that is used for producing and testing results in this dissertation.

3.1 Test Data Structure

Benchmarking is the main method used to measure time in this experiment. When benchmarking it is reasonable to do more than one test run on each dataset. This is mainly because other background processes on the computer may interrupt ongoing computation and may cause a relative huge impact on measurements, especially when sorting smaller dataset. When sorting 1000 elements sequentially, which only takes a few milliseconds on a modern computer, a small delay in the computation could cause the measured time to be many times higher than expected.

To get a wide range of test results, the test data structure consists of many different sizes. The results obtained in this dissertation will be used to generate different graphs by the structure that can be seen in Figure 3.1. This structure contains array-sizes ranging from 1000 to 10 million elements and also included the number of times each array is sorted to get a more precise result. The array-sizes used to test the speed and run time of QuickSort algorithms are defined using powers of ten *14. 10□□14*, 12<n<32 [99]. With n ranging from 12 to 32 we get a good collection of array-sizes between 1000 and 10 million to test. Another array size ranging from 1 to 2,400 is used to test the running time of searching algorithms.

```
static int[][]  dataArray = new int[][] {
/*{Array-size, N runs} */
  {      1 000, 2 500}, {      1 778, 2 500},
  {      5 623, 2 500}, {     10 000, 2 500},
  {     31 622, 1 250}, {     56 234, 1 250},
  {    177 827,   500}, {    316 227,   250},
  { 1 000 000,   100}, { 1 778 279,    75},
  { 5 623 413,    50}, {10 000 000,    40},
  {31 622 776,    16}, {56 234 132,     8},
};
```

Figure 3.1: Test Data Structure

For the smallest arrays-sizes a limit to 2500 test runs have been set, this should let the computation process “warm-up” and also giving a large collection of test data. As the array-sizes increase, the number of test runs is decreased. The main reason to do this is that as the array grows so does the computation time, and with an increased computation time the impact of other background processes will have a

smaller effect. The time for sorting up to 10 million elements will start to take a couple of seconds, so doing many tests runs start taking unnecessary amount of time. This effect has been explained and can be seen in Figure 3.3, Figure 3.4 and Figure 3.5 in this chapter.

3.2 Generating data

With defined sizes of the arrays containing test data, the next step would be to fill these arrays with actual data appropriate for sorting. The most common types of sorting would be to sort positive integers (i.e. 1, 2, 3, ...g), and in Java one can then use short, int or long as the variables. The choice of using 32-bit int as elements is best suited for the array-sizes chosen [99]; the signed int in Java may contain positive values from 1 to about 2.1 billion.

In Figure 3.2, a basic function is shown, this function will take two parameters; the wanted size of the array and a “seed” for the random data creation. After creating the random number generator rdm and constructing array with the desired size, a for-loop will fill each index of the array with random values ranging from 1 to array.length. By using the same seed when initializing an array with a specific size, this function will create an array which will contain identical values in each index every time. This is useful when wanting to benchmark the algorithms more than once and/or on other computers and still have the same starting point.

```
int[] initArray(int size, int seed) {
  Random rdm = new Random(seed);
  int[] array = new int[size];

  for (int i = 0; i < array.length; i++) {
    array[i] = rdm.nextInt(array.length) + 1;
  }

  return array;
}
```

Figure 3.2: Initialize an array with data

If one want to have completely random data each time, the Random(seed) can easily be changed to just Random() which then uses the systems clock rather than a given seed, ensuring randomized data with each and every use. But since we want to produce fair and comparable data to be used for all implementations in this dissertation, the same seed is used every time.

3.3 Benchmarking

To benchmark the algorithms in this dissertation, the main method used is practical measurement of run time. By measuring and comparing the time it takes to sort and search the variety of different array-sizes, as seen in Figure 3.1, we can see how well they perform compared to each other.

In Java there are currently two built-in functions which let the user retrieve time, and with a start and stop time we are then able to see how much time was spent performing the computation.

These functions are:

• System.currentTimeMillis()

• System.nanoTime() System.

System.currentTimeMillis() is based on the computers system-clock. This function actually returns the difference (in milliseconds) between the current time and midnight, January 1, 1970 UTC. This can be used to measure elapsed time but the function has some weaknesses; the system clock is in no way perfect, it may drift off and occasionally needs to be corrected. How often the system-clock ticks, increasing the unit of time also depends on the underlying operating system.

Included in Java 5 was the function System.nanoTime(), when this function is called it returns a number in nanoseconds from a fixed but arbitrarily chosen point in time, a time that may also be in the future. Since the purpose of this function is to measure elapsed time, it is unaffected by the small corrections done by currentTimeMills. Another possible measuring method would be profiling.

Since this dissertation is mainly focused on investigating the performance of different concurrent mechanisms in Java using single and dual-core machines, we are not really after implementing the “perfect” algorithm. But interested in measuring how the different mechanisms perform compared to each other on both machines. So profiling the different implementations to find CPU usage and

memory leaks is not really an issue. Comparing System.nanoTime() and System.currentTimeMillis(), we find that nanoTime() is the best alternative if one does not have a program that have to rely on system-date.

3.3.1 Median or average?

As mentioned in the previous Subsection 3.1 - Test Data Structure, we want to do many test runs on each dataset, but how should one treat these collection of test runs? Two of the easiest and best suited solutions here would be to either:

A. Sum all test results and get the average value.

B. Sort all the results and pick the median value.

Since the purpose is to summarize a set of test runs by a single typical value. The average would not be the best alternative in this case, as the average is more sensitive if non-typical values would occur. When measuring with only System.nanoTime() other background processes on the computer could cause the sorting process to get delayed and may be giving a much higher measured time than usual. In the following Figure 3.3, 3.4 and 3.5 the ExecutorService implementation of Quicksort was tested with an array-size of 50,000 and with three different numbers of runs; 10, 100 and 1000. The idea was to see how the average and median time differ from each other and to get a typical run times that can be

correctly compared and also to explain the effect of background process on the measured time.

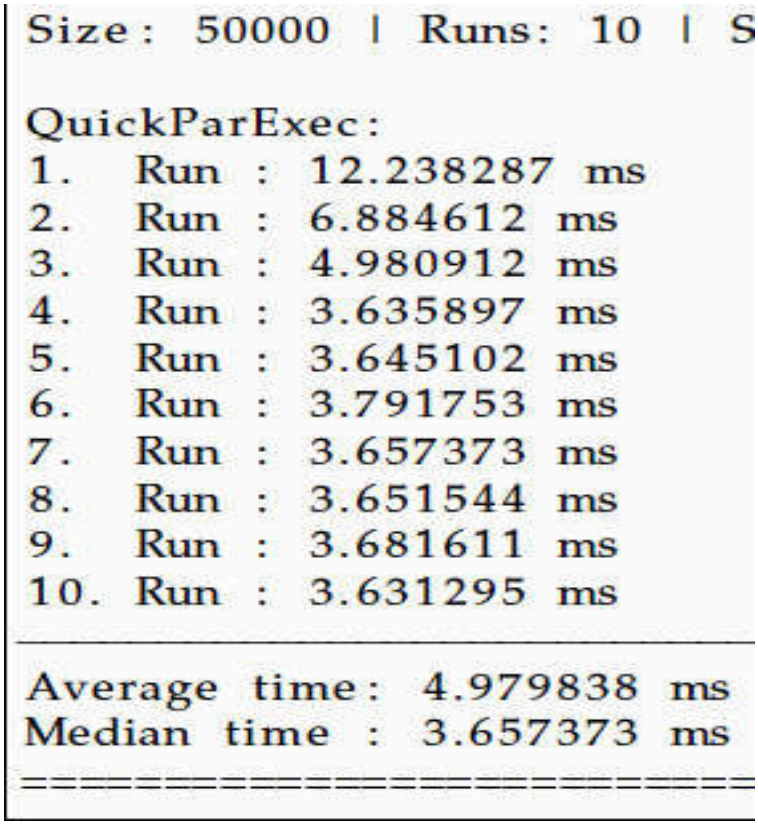


Figure 3.3: Quick Sort; 10 runs.

When increasing the number of runs we see much less impact of background process, but the average value still does not give out a typical run time.

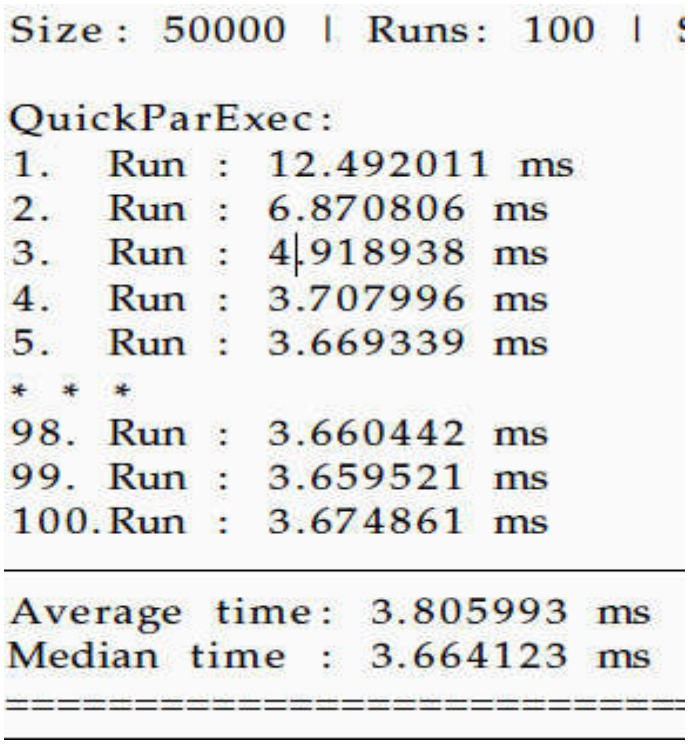


Figure 3.4: Quick sort; 100 runs.

When increasing the number of runs we see much less impact, but the average value still does not give out a typical run time.

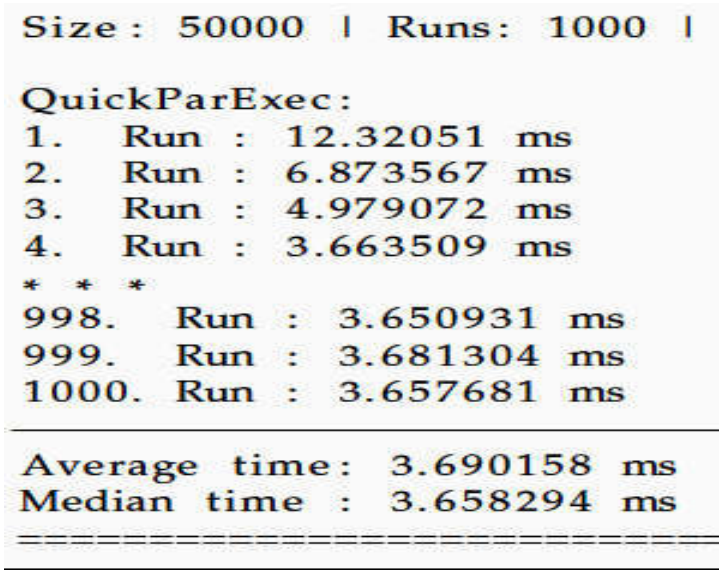


Figure 3.5: Quick sort; 1000 runs.

Even though one could increase the number of test runs and that the impact will be much less with the increased array-size. The conclusion is to use median time to get the typical value for a test run, and it is these values that are used for future drawing of graphs in the next chapters.

3.4 Algorithms analysis

Two most popular searching algorithms linear search and binary search followed by sorting algorithms are analyzed as follows:

3.4.1 Analysis of Binary search algorithm

In computer science, binary search, also known as half-interval search or logarithmic search is a search algorithm that finds the position of a target value within a sorted array. It compares the target value to the middle element of the array; if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful. Binary search runs in at worst logarithmic time, making O(logn) comparisons in the worst case, O(1) in the best case, where n is the number of elements in the array and log is the binary logarithm; and using only constant space. Although specialized data structures designed for fast searching such as hash tables can be searched more efficiently, binary search applies to a wider range of search problems. Binary searches require the collection to be sorted [96].

3.4.2 Pseudo-code for Binary search algorithm according to reference [98].

1: BinarySearch (S, k, low, high)

2: If low > high then

3: Return NO_SUCH_KEY

4: else

5: mid←(low+high) / 2

6: if k = key (mid) then

7: return key (mid)

8: else if

9: k < key (mid) then return BinarySearch

10: (S, k, low, mid−1)

11: else return

12: BinarySearch (S, k, mid+1, high).

3.4.3 Analysis of Linear search algorithm

Linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found. Linear search is the simplest search algorithm. For a list with n items, the best case is when the value is equal to the first element of the list, in which case only one comparison is needed. The worst case is when the value is not in the list (or occurs only once at the end of the list), in which case n comparisons are needed.

The worst case performance scenario for a linear search is that it has to loop through the entire collection, either because the item is the last one, or because the item is not found. In other words, if you have N items in your collection, the worst case scenario to find an item is N iterations. In Big O Notation it is O(N). The speed of search grows linearly with the number of items within your collection. Linear searches don't require the collection to be sorted [102].

3.4.4 Pseudo-code for Linear search algorithm according to reference [102]

Step 1: Linear search (a1,…,an: reals, b reals);

Step 2: I = 1;

Step 3: while (ai !b and b<n);

Step 4: i = i+1;

Step 5: if (ai = b) return I;

Step 5: else return zero;

3.4.5 Analysis of Quick Sort Algorithm.

Quick sort is a comparison sort developed by Tony Hoare [19]. Also, like merge sort, it is a divide and conquer algorithm, and just like merge sort, it uses recursion to sort the lists. It uses a pivot chosen by the programmer, and passes through the sorting list and on a certain condition, it sorts the data set. Quick sort algorithm can

be depicted as follows according to reference [19]. The choices of sorting algorithms have been based on trying to find types which let us use a wide range of different tools from the concurrency package in Java. Therefore we are after algorithms that achieve the following:

1. Sorting in parallel is feasible. The algorithms does not necessary have to be specifically constructed for parallel computing, as we want to see the effect of going from sequential to parallel. But we should be able to run at least some parts of the algorithms concurrently, so that we can expect some performance boost with multi-core.

2. Popular or at least commonly known sorting algorithms.

3.4.6 Pseudo-code for Quick sort algorithm according to reference [19]

```
QUICKSORT (A)
1: step←m;
2: while step > 0
3: for (i←0 to n with increment 1)
4: do temp←0;
5: do j←i;
6: for (k←j+step to n with increment step)
7: do temp←A[k];
 8: do j←k-step;
 9: while (j>=0 && A[j]>temp)
```

10: do A [j+step] = A[j];

11: do j←j-step;

12: do Array[j]+step←temp;

13: do step←step/2.

Run Time of Quick Sort is O(nlogn) on the average, and the worst case O(n₦2) is according to [19].

3.5 Software and hardware analysis

With the idea of making many different implementations of the sorting and Searching algorithms, it is necessary to keep using the same hardware with every test run. This we do in order to ensure that the measured times can be correctly compared to each other, giving us the information on how well each implementation perform on a single and Multi-core machines.

3.5.1 Hardware and Software Specifications

For the development and benchmarking of algorithms used in this dissertation the hardware specifications in the next page in Table 3.1 and Table 3.2 will be used.

Table 3.1: Dual-core-core processor specification

CPU	Intel(R) Pentium® Dual Core CPU T2390 @ 1.86GHz, 1.87GHz
RAM	1.00 GB
OS	Windows 7 32-bit

Table 3.2: Single-core processor specification

CPU	Mobile AMD Sempron™ Single Core Processor 3200+1.60GHz
RAM	1.50 GB
OS	Windows 7 64-bit

While the main platforms used for this dissertation are Windows 7 64 and 32 bits, the development environment is Net-beans IDE 8.0.2

CHAPTER FOUR

(RESULTS AND DISCUSSIONS)

To test out the various frameworks and some of its classes in java using single and multi-core machines, some sorting and searching algorithms with different characteristics are implemented in this chapter to be able to try out some varieties of concurrency tools. This would let us see how increase in the number of cores in a CPU improves performance of concurrent program.

4.1 Sorting Algorithms

Analysis of sorting algorithm was carried out earlier in section 3.4.5 Chapter three.

Here, each of the three sorting algorithms of choice will be separately implemented.

4.1.1: Quicksort pseudocode

Simpler pseudo-code for a Quick sort algorithm is shown in Figure 4.1 in the next page, and is based on the code from the book Introduction to Algorithms by Thomas H [21].

```
QUICKSORT(A, p, r)
  if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)


PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
    if A[j] <= x
      i = i + 1
      exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

Figure 4.1: Quicksort pseudocode

The fundamental steps for the sorting are rather simple:

1. Pick an element q, the pivot, from the array A.

2. Partitions the remaining elements into those greater than, less than the pivot q.

3. And recursively repeat the process on the partitions, until the array is completely sorted.

There are at least two things that make Quick sort a suitable algorithm for parallelism. First it does not have any shared data, meaning no need for synchronization, as the pivot split the array in two each iteration. Second each part of the array would recursively call Quick sort dividing it in such a way that they can be assigned for the available resources on the system.

4.1.2 Implementation of QuickSortSequential on a dual-core machine

Based on the previous mentioned pseudocode of Quicksort, the code in Figure 4.2 is a rather straight forward implementation of Quicksort in Java.

```
class QuicksortSequential {
  void quicksort(int[] array, int left, int right) {
    if (left < right) {
      int pivotIndex = partition(array, left, right);
      quicksort(array, left, pivotIndex - 1);
      quicksort(array, pivotIndex + 1, right);
    }
  }

  int partition(int[] array, int left, int right) {
    pivotValue = array[right];
    int index = left;

    for (int i = left; i < right; i++) {
      if (array[i] <= pivotValue) {
        swap(array, i, index);
        index++;
      }
    }

    swap(int[] array, index, right);
    return index;
  }

  void swap(int[] array, int left, int right) {
    int temp = array[left];
    array[left] = array[right];
    array[right] = temp;
  }
}
```

Figure 4.2 Quick sort sequential

With only some variable declaration, name differences and the included swap function, one will easily notice the similarity in the code with Figure 4.1. But one problem with this sequential implementation is when the worst-case scenario happens. When the pivotValue is set by the rightmost (or could also be leftmost) integer in the array, worst-case would actually be to sort an already sorted array and the same goes if it is in reverse order.

There is more than one solution to help dealing with this problem, we can:

A. Choose the integer in middle.

B. Choose the median of three integers.

C. Choose a random integer from the array.

Figure 4.3 shows the usage with option A, which is used for the Quick sorts while option C is used for Linear and binary search implementation in this dissertation.

int pivotValue = array [(left right)/2] ;

2 swap (array, (left+right)/2 , right) ;

Figure 4.3: Pivot Value.

To make the Quicksort algorithms more like the built-in Arrays.sort() in Java, insertion sort is included for small arrays. The current threshold for when to use insertion sort is set to **47**, the same threshold currently used in Arrays.sort().

```
void insertionSort(int[] array, int left, int right) {
  for (int i = left + 1; i <= right; i++) {
    int a = array[i];
    int j;
    for (j = i - 1; j >= left && a < array[j]; j--) {
      array[j + 1] = array[j];
    }
    array[j + 1] = a;
  }
}
```

Figure 4.4: Insertion Sort

The Figure 4.4 includes the code used in all Quicksort implementation in this dissertation, and is called when an array (or part of an array while sorting) contains 47 or less elements. By including this, a performance boost can be achieved as insertion sort is very efficient for small arrays.

4.1.3 Test run of Quick sort sequential on dual-core machine

To compare how well this implementation of Quicksort are to the Arrays.sort(), a test run has been done. What size and type of data that is used to create the graph are described in greater detail in Chapter 3-methodology.

Figure 4.5 and Figure 4.6 were plotted by comparing the speed and running time of this algorithm as seen in Table 4.1 with that of Array.Sort(), the default Sequential Sorting algorithms in Java.

Table 4.1: Performance of Quick sort sequential on dual core machine

QuickSortSequential		
Array Size	Speed (%)	Running time(ms)
1000	91.8	0.074
31622	94.6	3.526
100000	91.8	12.6
177827	91	24.12
316227	94.3	44.18
5623413	92.7	81.84
1000000	94.4	151.9
3162277	90.2	537.4
562341	91.4	977.2
10000000	92.3	1808

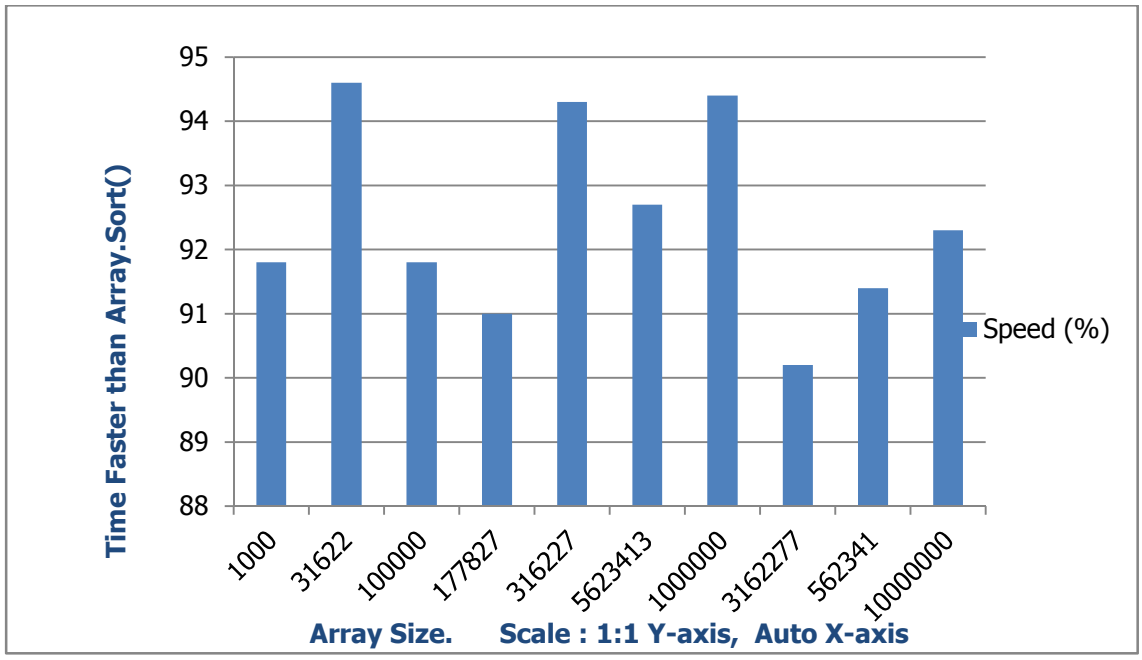


Figure 4.5 Speed of QuickSortSequential.

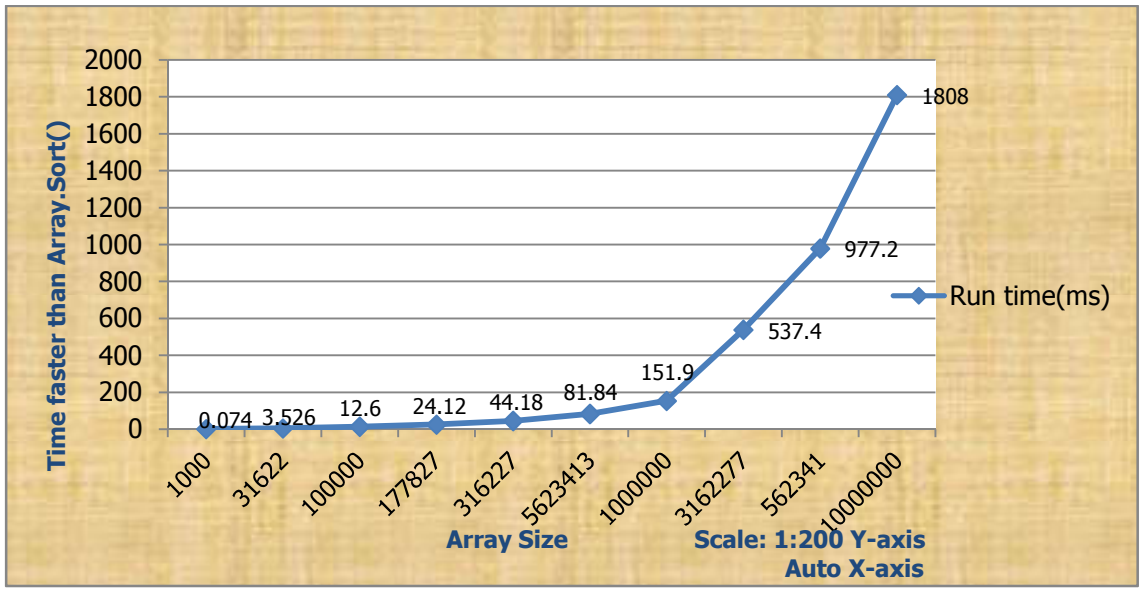


Figure 4.6 Running time of QuickSortSequential

Thus, it can be clearly from Table 4.1 and figure 4.1 and 4.2 that QuickSortSequential performs better on small array elements as it takes 0.074ms to sort 1000 array size with the corresponding speed of 91.8% but as the array grew

larger from 1000 to 10,000000 it started to perform poorly as compared to other parallel algorithms with the running time of 1808ms.

4.1.4 Performance of Parallel Quicksort Implementation on a dual-core.

With the divide-and-conquer pattern that Quick sort is built on, there is no need for synchronization as the array that we are going to sort will be divided for each recursive call and which will result in that we have no shared data. With no shared data and an algorithm that split into smaller tasks for us, we only need to have tools for executing these tasks in parallel. Going through Java API documentation, we ended up with three different tools to create parallel implementations of Quicksort. The parallelization will be achieved with the following scenarios:

1. A “naive” attempt by only creating new Threads.
2. Using the new Fork/Join-framework.

This will let us see how well each of the implementation performs compared to each other. We will also see if the new framework Fork/Join does have any advantages compared with the older Naïve implementation.

4.1.5 Performance of QuickSortParallelNaive on dual-core machine

The naive attempt differs from the sequential Quicksort in that way that each time Quicksort recursively calls itself; it instead creates new threads which then call. And by creating these new threads, each core (included Hyper-Threads) on the

current system can then be assigned the available threads and do the computation in parallel, which then should result in increasing the overall performance. The Figure 4.1 contains code for this implementation. But with large and larger arrays to sort, even more threads will be created in the naive attempt of QuickSort. As creating threads causes overhead and with a new thread for each recursive call, the algorithm would result in poor performance, we therefore need to limit the creation process somehow.

```
class QuicksortParallel implements Runnable {
  private final int[] array;
  private final int left, right;
  final static int INSERTION_SORT_THRESHOLD = 47;

  public QuicksortParallel(int[] arr, int l, int r) {
    // Constructor
  }

  public void run() {
    quicksort(array, left, right);
  }

  void quicksort(int[] array, int left, int right) {
    if (right-left <= INSERTION_SORT_THRESHOLD) {
      insertionSort(array, left, right);
    }

    else {
      int pivotIndex = partition(array, left, right);
      Thread t1 = new Thread(
          new QuicksortParallel(array, left,  pivotIndex - 1));
      Thread t2 = new Thread(
          new QuicksortParallel(array, pivotIndex + 1, right));
```

```
    t1.start();
    t2.start();
    try {
      t1.join(); t2.join();
    } catch (InterruptedException e) {}
```

```
  }

  int partition(int[] array, int left, int right) {
    int pivotValue = array[(left + right) / 2];
    swap(array, (left + right) / 2, right);
    int index = left;

    for (int i = left; i < right; i++) {
      if (array[i] <= pivotValue) {
        swap(array, i, index);
        index++;
      }
    }

    swap(array, index, right);
    return index;
  }


  void swap(int[] array, int left, int right) {
    int temp = array[left];
    array[left] = array[right];
    array[right] = temp;
  }
}
```

Figure 4.7 Parallel Naive Quicksort.

In Figure 4.8 a hardcoded threshold is used to limit the creation of threads so that a better performance is achieved as creating threads causes overhead.

```
final static int LIMIT = 50000;

void quicksort() {
...
  if (right-left <= LIMIT) {
    /** If current part of the array-size is less than the set
    LIMIT, we call quicksort() recursive with current thread */
  }
  else {
    /** Else we create two new threads to call quicksort() */
  }
}
```

Figure 4.8: Granularity; limit the creation of threads for Quicksort.

The result of using this IF-check is that it will also reduce the number of additional threads to create, and on this naive implementation only one thread is created when the array-sizes is less or equal to 50,000 elements.

4.1.6 Test run on QuicksortParallelNaive on dual-core machine

Having including the hardcoded threshold to the code in Figure 4.8, the next step is doing a test run and comparing it to Arrays.sort(). The test result can be seen in Table 4.2 in the next page.

Table 4.2 Performance of QuickSortParallelNaïve

QuicSortParallelNaïve		
Array Size	Run time(ms)	Speed(%)
1000	0.344	19.7
31622	3.876	86
100000	10.89	106.2
177827	19.16	114.5
316227	32.77	127.1
5623413	57.6	131.7
1000000	105.3	136.2
3162277	389.3	124.6
562341	672.1	133
10000000	1270	131.4

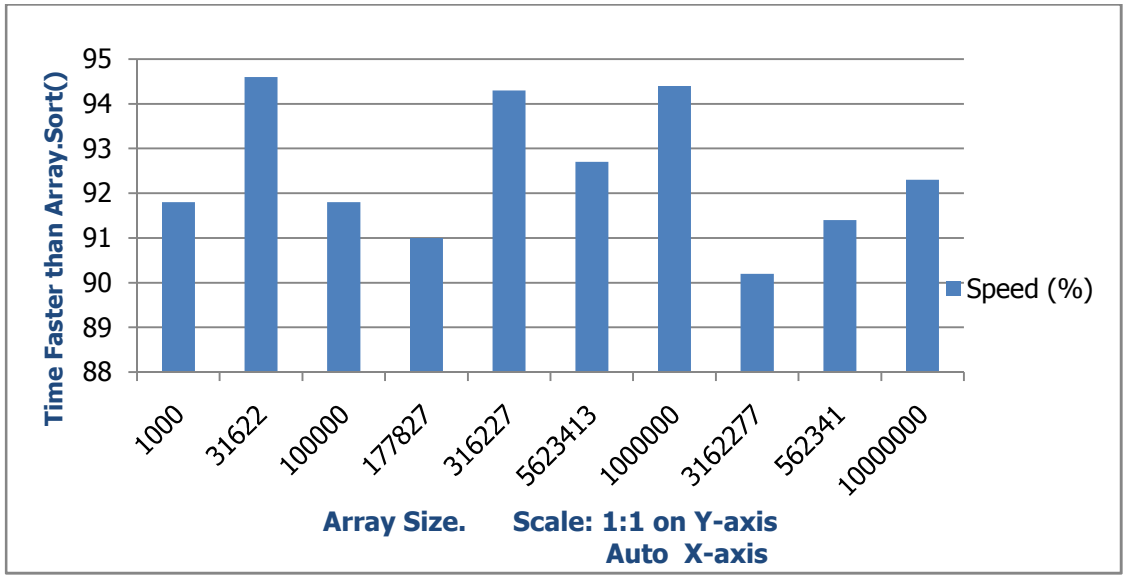


Figure 4.9: Speed of Parallel Naive Quick sort.

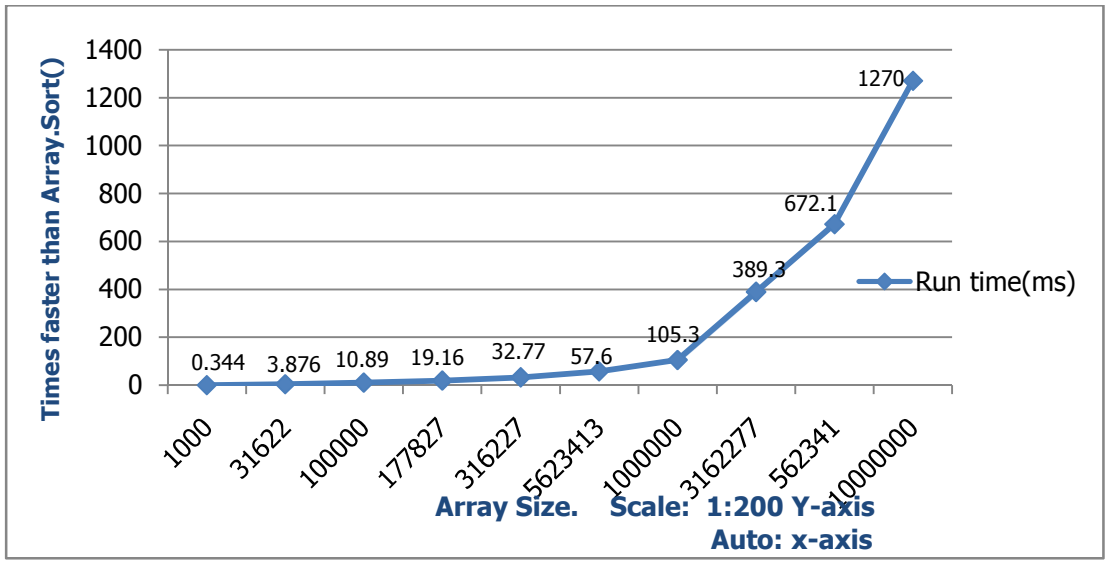


Figure 4.10: Running time of QuickSortParallelNaive.

Because the sorting started by allocating a new thread for the QuicksortParallel class, the overhead will cause such an impact on the measured times that it performs worse when sorting less than 10,00000 elements. A solution to this problem could have been to do add an array length check before deciding to create and run the parallel implementation, and rather call a sequential Quicksort implementation to get the result from Figure 4.2.

As the number of elements goes above 1,000000, the algorithm starts to create new threads for each recursive calls. Threads get assigned to different cores on the CPU by the Java Virtual Machine and Operating System. The outcome of this is the performance increase seen in Figure, with the highest at 10 million elements being 92.3% times faster than Arrays.sort().

From Figure 4.9 and Figure 4.10, it can be seen that QuickSortParallelNaive sorted 1000 array elements in just 0.34ms when compared with the Array.Sort(), comparing this results with one obtained in Figure 4.2, the QuickSortSequential performs better with a running time of just 0.074ms. This is due to the fact that QuickSortParallelNaive creates thread for each recessive call and if the array to be sorted is small more number of threads is created leading to large overheads in the measured time. This delay has been minimized by introducing the insertion sort when the array size is less than or equal to 50,000. But it can still be seen that the effect of the overhead is still significant when the array size is small even with the used of insertion Sort.

Considering the largest array size, 10,000000 elements, QuickSortParallelNaive performs better than QuickSortSequential with the speed of 131.3% faster than Array.sort() when compared with 92.3% obtained from Table 4.1**.** Thus, it can be seen that the parallel behaviour of QuickSortParallelNaive improves its performance since the created threads are divided amongst the different cores in the CPU and the result is combined to get the array completely sorted.

4.2 Implementation of QuickSortFork/Join on a dual-core machine

One of the new and very interesting features introduced in Java concurrency framework is the **Fork/Join**-framework. A fixed pool is created to contain the available worker threads, and then a task is submitted to this pool. The Fork/Join uses invoke on tasks, this performs the given task and returning its result upon completion. This means that there is no need to use future to keep track of the tasks. Creating the Fork/Join-framework only requires specifying the number of desired threads for the pool (You can actually just call ForkJoinPool(), which uses Runtime.getRuntime().availableProcessors() as default), example:

ForkJoinPool QParFJ = new ForkJoinPool(CORES); Rest of the implementation can be seen in Figure 4.6.

```
class QuicksortExecutor implements Runnable {
  ExecutorService pool;
  List<Future> futures;

  ...

  void quicksort(int[] array, int left, int right) {
    if (right-left <= INSERTION_SORT_THRESHOLD) {
      insertionSort(array, left, right);
    }

    else {
      int pivotIndex = partition(array, left, right);

      if (right-left <= LIMIT) {
        quicksort(array, left, pivotIndex - 1);
        quicksort(array, pivotIndex + 1, right);
      }

      else {
        futures.add(pool.submit(new QuicksortExecutor
            (array, left, pivotIndex - 1, futures, pool)));
        futures.add(pool.submit(new QuicksortExecutor
            (array, pivotIndex + 1, right, futures, pool)));
      }
    }
  }
  ...
}
```

Figure 4.11 Parallel Quick sort using Fork/Join

4.2.1 Test Runs of QuickSortFork/join on dual-core machine

Table 4.3, Figure 4.12 and Figure 4.13 show the result of the test run with the Fork/Join implementation. Comparing the different graphs, Fork/Join-framework is actually faster compared to Parallel naive. This is not surprising as we expect the work-Stealing aspect would increase the performance as none of the worker threads would idle (i.e. they got dealt a smaller portion of the array that is sorted and finished earlier).

Table 4.3 Performance of QuickSortForkJoin on dual-core machine

QuickSortForkJoin		
Array Size	**Speed (%)**	**Runtime (ms)**
1000	65.5	0.104
31622	93.1	3.583
100000	129.4	8.934
177827	142.9	15.36
316227	156.1	26.687
5623413	164.3	46.16
1000000	168.7	84.95
3162277	157.5	307.8
562341	164.4	543.6
10000000	165.6	1008

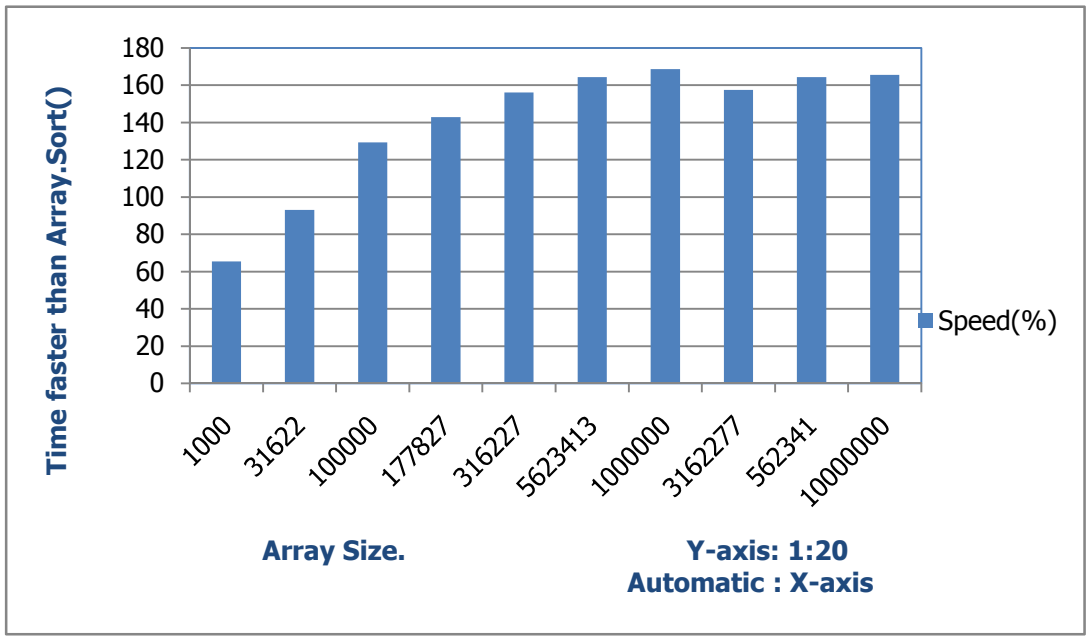


Figure 4.12: Speed of QuickSortForkJoin

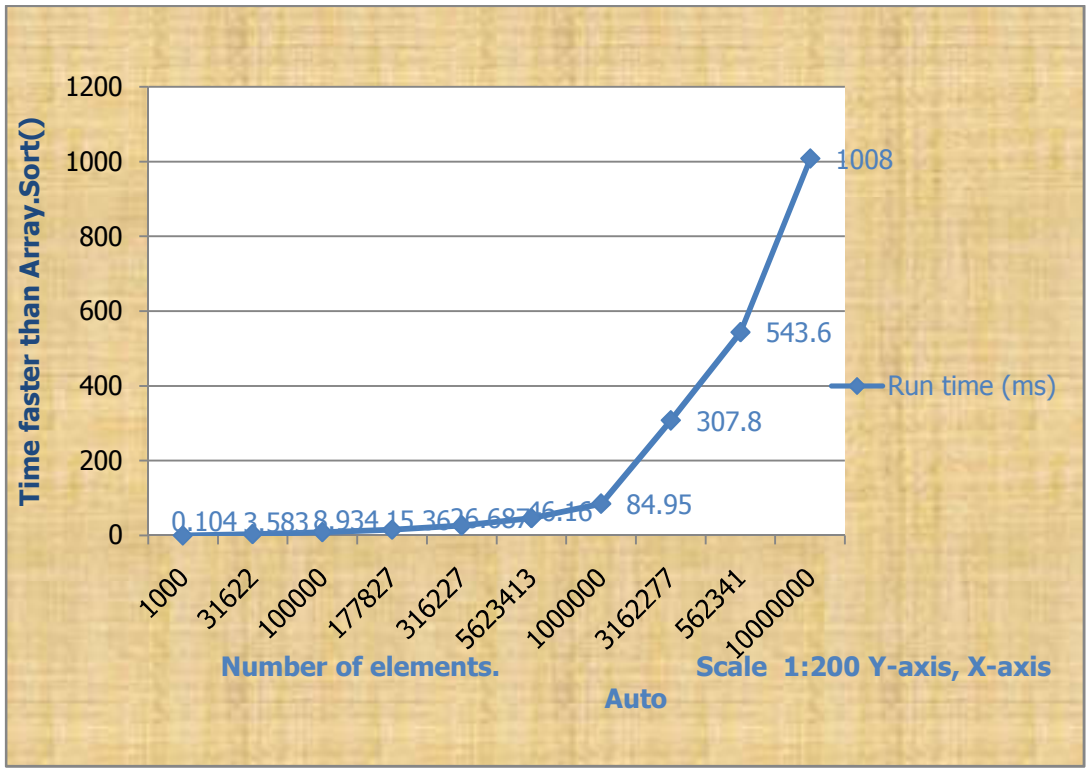


Figure 4.13: Running Time of ParallelQuickSort Fork/Join.

From Figure 4.13, it can be seen that QuickSortForkJoin took 0.104ms with the corresponding speed of 65.5% only to sort 1000 array elements when compared with Array.Sort(). When compared with other two algorithms however, QuickSortForkJoin performs worse with smaller array size. But with an increased array size, its good performance can be seen. As the array size increased to 10,000000 elements, it took 1008ms only with the corresponding speed of 165.6 % when compared with Array.Sort() to sort the array completely.

Table 4.4 General speed of all the three sorting algorithms on dual-core machine

Array size	QuickSortParallelNaïve Speed (%)	QuickSortForkJoint Speed (%)	QuickSortSequential Speed (%)
1000	19.7	65.5	91.8
31622	86	93.1	94.6
100000	106.2	129.4	91.8
177827	114.5	142.9	91
316227	127.1	156.1	94.3
5623413	131.7	164.3	92.7
1000000	136.2	168.7	94.4
3162277	124.6	157.5	90.2
562341	133	164.4	91.4
10000000	131.4	165.6	92.3

Table 4.4.1: General running time of all the three Sorting algorithms on dual-core machine

QuickSortSequential		QuickSortParallelNaïve	Fork/Join
Array Size	Run time(ms)	Run time(ms)	Run time (ms)
1000	0.074	0.344	0.104
31622	3.526	3.876	3.583
100000	12.6	10.89	8.934
177827	24.12	19.16	15.36
316227	44.18	32.77	26.687
5623413	81.84	57.6	46.16
1000000	151.9	105.3	84.95
3162277	537.4	389.3	307.8
562341	977.2	672.1	543.6
10000000	1808	1270	1008

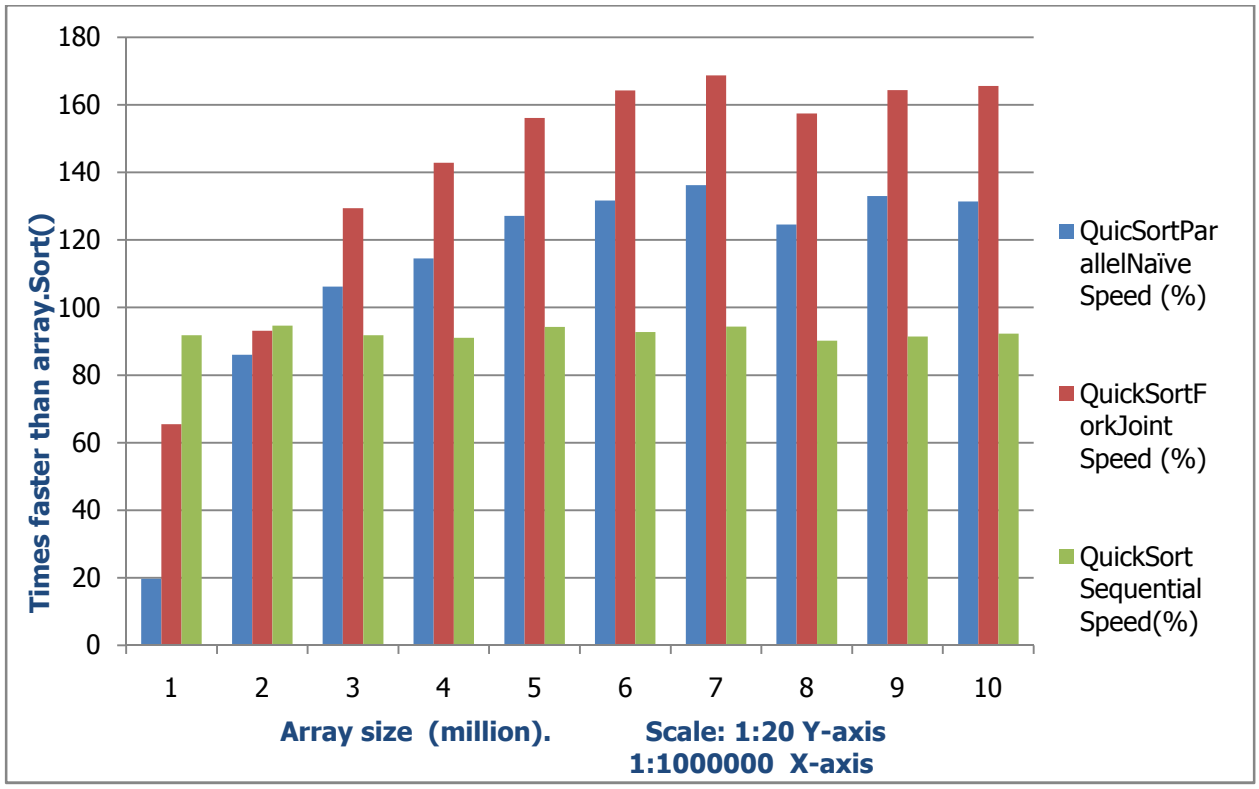


Figure 4.14: Speed of all the three quick sort algorithms.

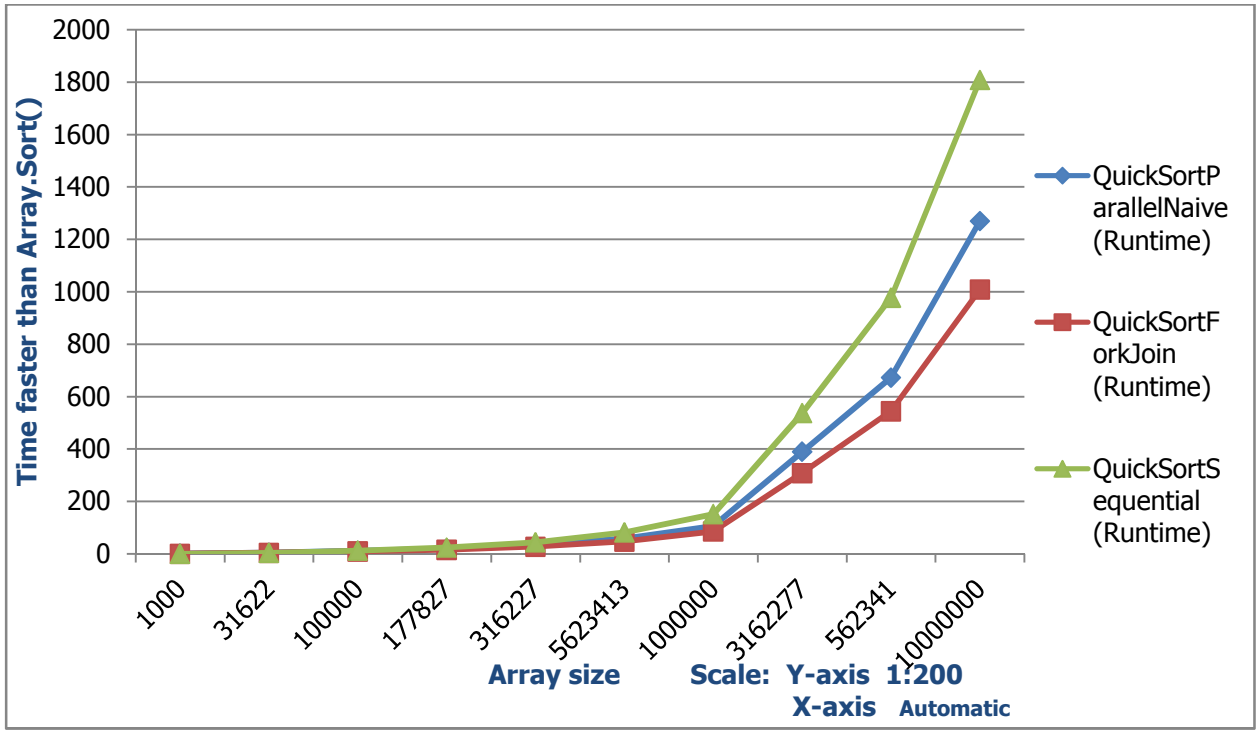


Figure 4.15: Running time of all the three sorting algorithms.

Figure 4.14 and Figure 4.15 shows that QuickSortForkJoint took 0.104ms with the corresponding speed of 65.5% to sort 1000 array elements when compared with Array.Sort(). Comparing it with other three algorithms however, QuickSortForkJoint performs worse with small array size. But with an increased array size, its good performance was elevated as the array size increases to 10,000000 elements. QuickSortForkJoin took 1008ms with the corresponding speed of 165.6 % when compared with Array.Sort() to sort this array completely.

Therefore, Comparing QuickSortForkJoin with the other two algorithms: QuickSortSequential, and QuickSortParallelNaive plotted in Figure 4.9 and Figure 4.9.1, it could be concluded that QuickSortForkJoin emerged as the best Sorting algorithms when the array size is very large but QuickSortSequential has a better performance with the small array size. The best performance of QuickSortForkJoin is not surprising because the Fork/Join uses invoke on tasks, this performs the given task and returning its result upon completion. This means that there is no need to use future to keep track of the tasks as used in older concurrency tools. Secondly, the divide and conquer approached used by Fork/Join is also another reason for its best performance because with the program divided into smaller tasks, a work stealing action is used by each of the created threads if it has no task in its queue. The outcome of this is an increased performance seen from the result obtained in this experiment.

4.3 Test runs on both single and dual core-core machines

To fully investigate the effect of increasing the number of cores in the performance of a given program, test runs is carried out on a single core machine to compare it with the one carried out on a dual core machine described earlier in section 4.1 in this chapter. The specification of the hardware and software that is used has been described in section 3.4.1, chapter three.

4.3.1 Test Run of QuickSortSequential on both single and dual core machine.

As expected when using a machine with less number of cores to run the same program designed on a Multi-core machine, a performance decreased has been recorded in Table 4.5 This can be seen in Figure 4.16.

Table 4.5: Speed and running time of QuickSortSequential on both machines

QuickSortSequential (Dual-core)			**QuickSortSequential (Single-core)**	
Array Size	Speed (%)	Running time(ms)	Speed (%)	Running time(ms)
1000	91.8	0.074	90.5	0.086
31622	94.6	3.526	93.8	4.227
100000	91.8	12.6	86.3	16.27
177827	91	24.12	86.3	31.25
316227	94.3	44.18	90.1	56.16
562341	92.7	81.84	88.9	104.7
1000000	94.4	151.9	88.9	197.2
3162277	90.2	537.4	87.2	681.3
5623413	91.4	977.2	87.4	1212
10000000	92.3	1808	88.1	2399

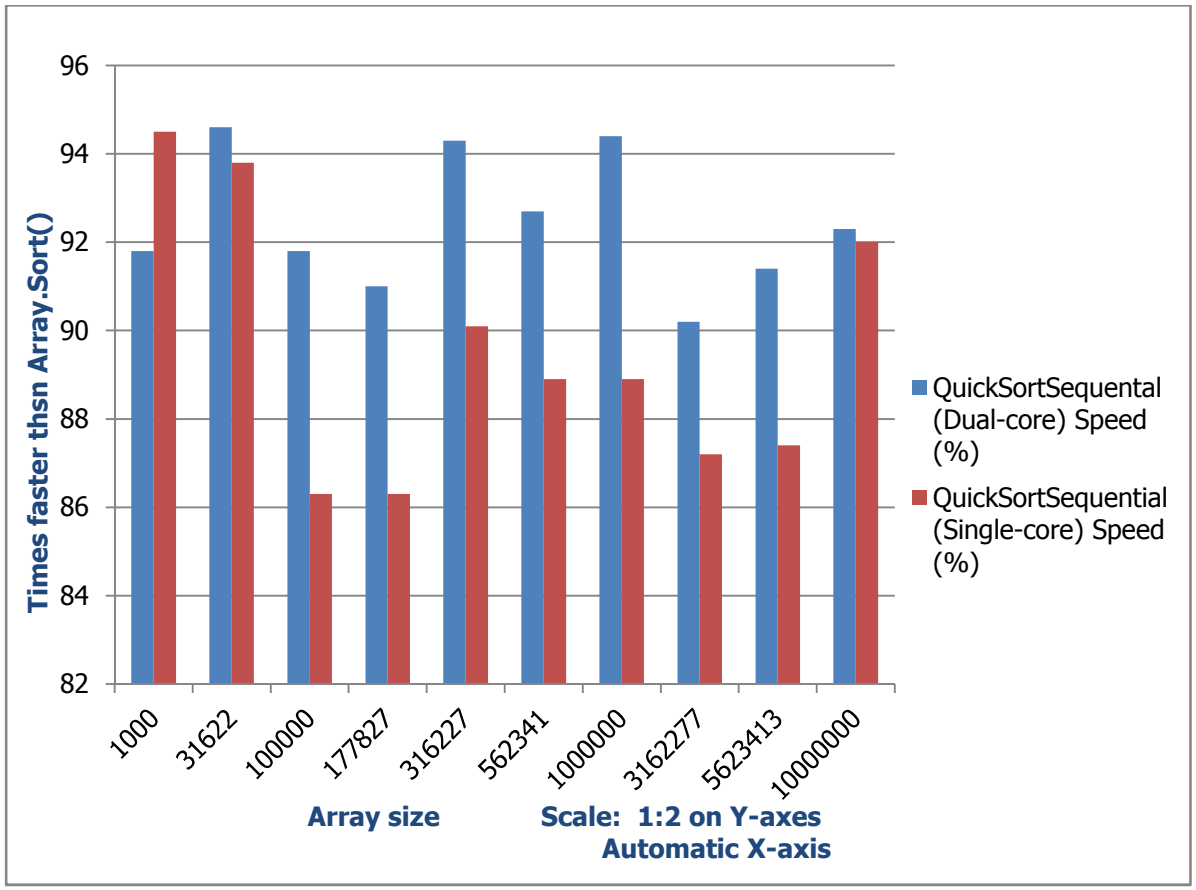


Figure 4.16: Speed of QuickSortSequential on both dual and single core machine.

From Figure 4.16: As the size of the array increases, speed of the QuickSortSequential is also increased. Considering the smallest array size 1000 elements the speed on a single-core machine is measured as 90.5% against 91.8% recorded on a dual-core machine. Conversely, when sorting 10,000000 elements, a speed of 88.1% was obtained on single core as against 92.3% recorded on a dual-core machine. Therefore, a program run on a dual-core machine recorded 4.2% speed increased compared with the same program run on a single core machine with QuickSortSequential algorithm.

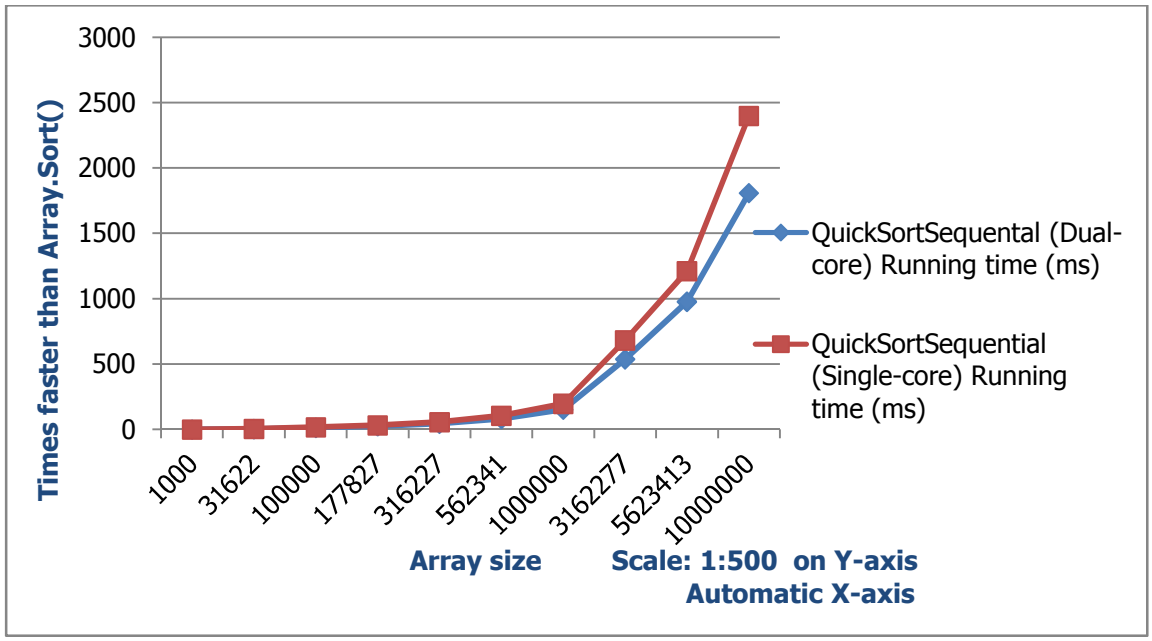


Figure 4.17: Running time of QuickSortSequential on both dual and single core machine.

From Figure 4.17, performance of both algorithms on both single seems to converge with small difference up to the time when 562341elements were sorted. However, as the number of elements increased to 1,000000 elements the performance of a dual-core machine became noticeable throughout the sorting process. Running time recorded on a single core machine at the end of the sorting process was 2399ms while 1808ms was recorded on a dual core machine. Thus, 591ms difference in the running time was recorded. Hence, dual-core machine runs the same program much faster than single core machine.

4.3.2 Test Run of QuickSortParallelNaive on both single and dual core.

Table 4.6 Speed and running time of QuickSortParallelNaive on both machines

Array Size	QuickSortParallelNaive(Dual-core)		QuickSortParallelNaive (SingleCore)	
	Speed (%)	Running time(ms)	Speed (%)	Running time(ms)
1000	91.7	0.344	20.3	20.3
31622	94.3	3.876	79.9	4.965
100000	106.2	10.89	67.7	20.73
177827	114.5	19.16	68.7	40.43
316227	127.1	32.77	71.4	70.85
562341	131.7	57.6	72.7	128
1000000	136.2	105.3	72.8	240.6
3162277	124.6	389.3	73.7	806.2
5623413	133	672.1	75.6	1480
10000000	131.4	1270	80.1	2920

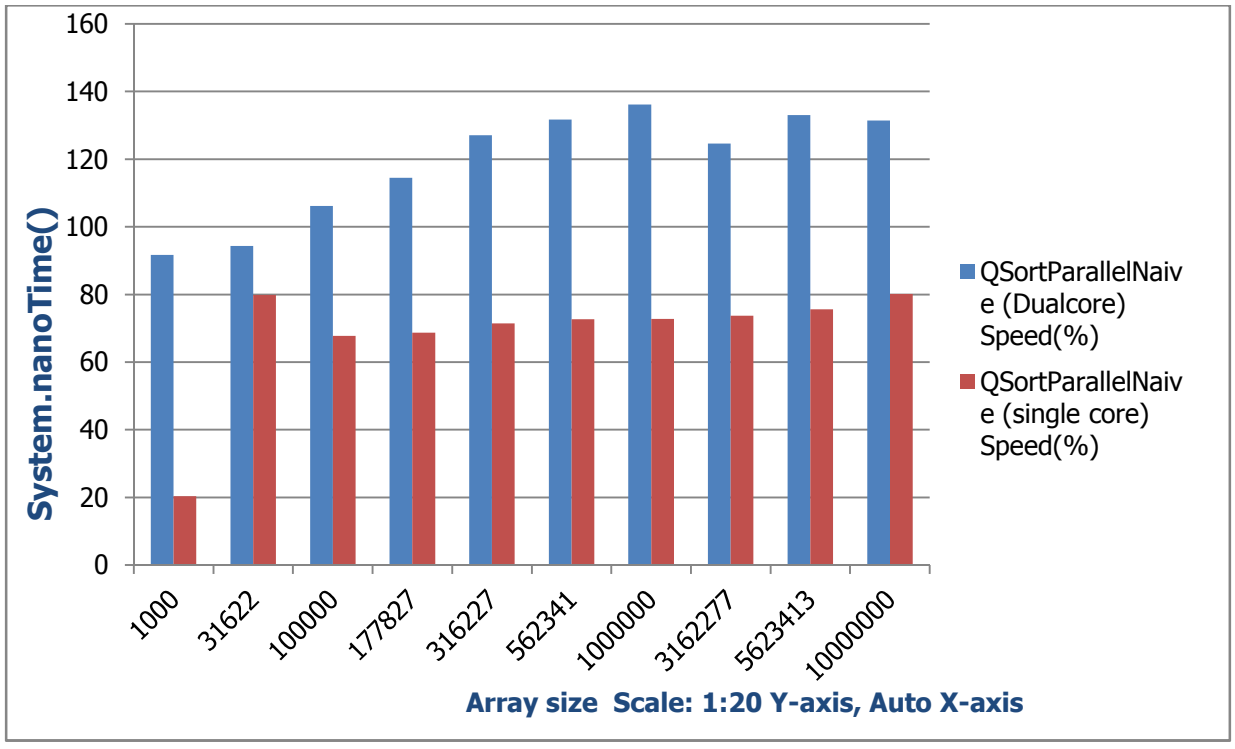


Figure 4.18: Speed of QuickSortParallelNaive on both dual and single core machine.

It is clear from Table 4.6 and Figure 4.18 that dual-core machine exhibits better performance, recording the highest speed than single-core machine throughout the sorting process. Considering the smallest array dual-core recorded 91.3% speed while single core recorded just 20.3%. Hence, dual-core is found to be 71% faster than single core. While sorting 10,00000 dual core recorded a speed of 131.4% as against 80.1 recorded on the single core, making it 51.3% faster at the end of the sorting process.

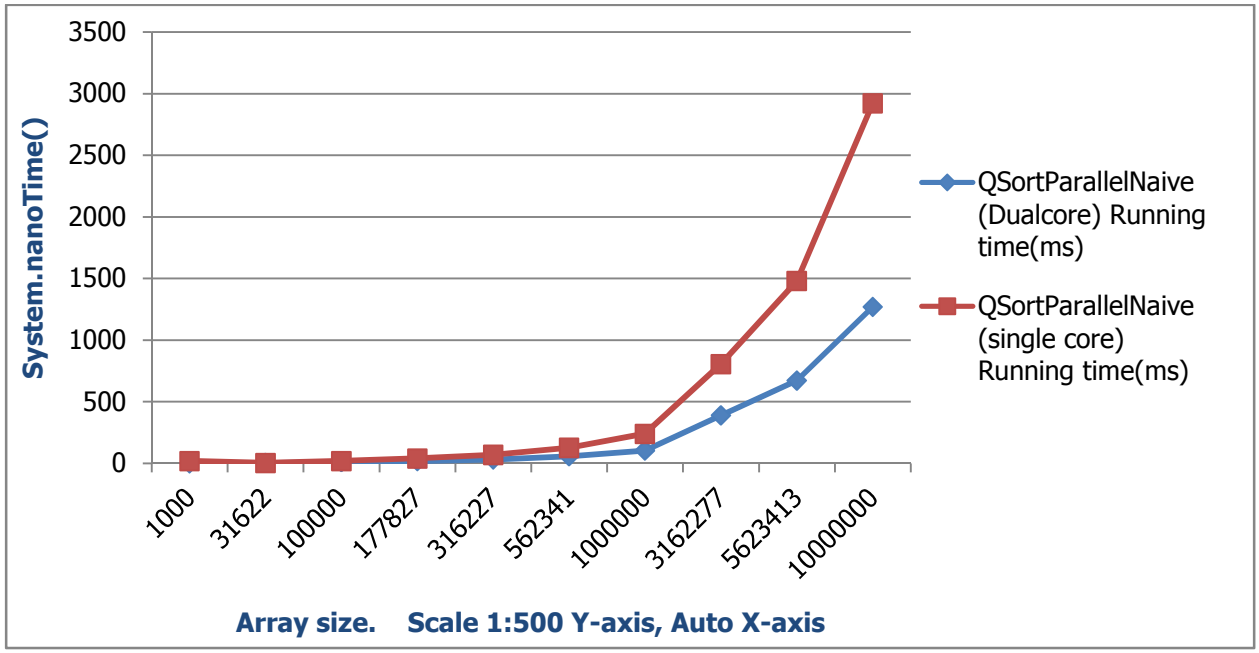


Figure 4.19: Running times of QuickSortParallelNaive on both dual and single core machines.

From Figure 4.19, Naïve attempt of Quick sort performs better on a dual-core with less running time throughout the sorting processes. Running time of 1270ms was obtained on dual-core while 2920ms was recorded on a single-core machine when the array reached 10,000000 elements. An improved speed of 1650ms is achieved.

Table 4.7: Speed and running time of QuickSortForkJoin on single and dual-core machines

Array Size	QuickSortParallelNaive (Dual-core)		QuickSortParallelNaive (Singlecore)	
	Speed (%)	Running time(ms)	Speed (%)	Running time(ms)
1000	95.5	0.104	76.9	0.106
31622	93.1	3.583	80.3	4.392
100000	129.4	8.934	86.2	16.2
177827	142.9	15.36	93.7	35.43
316227	156.1	26.687	97.5	57.84
562341	164.3	46.16	99.2	108.2
1000000	168.7	84.95	1004.9	204.6
3162277	157.5	307.8	1008.9	699.9
5623413	164.4	543.6	110.2	1248
10000000	165.6	1008	120.5	2495

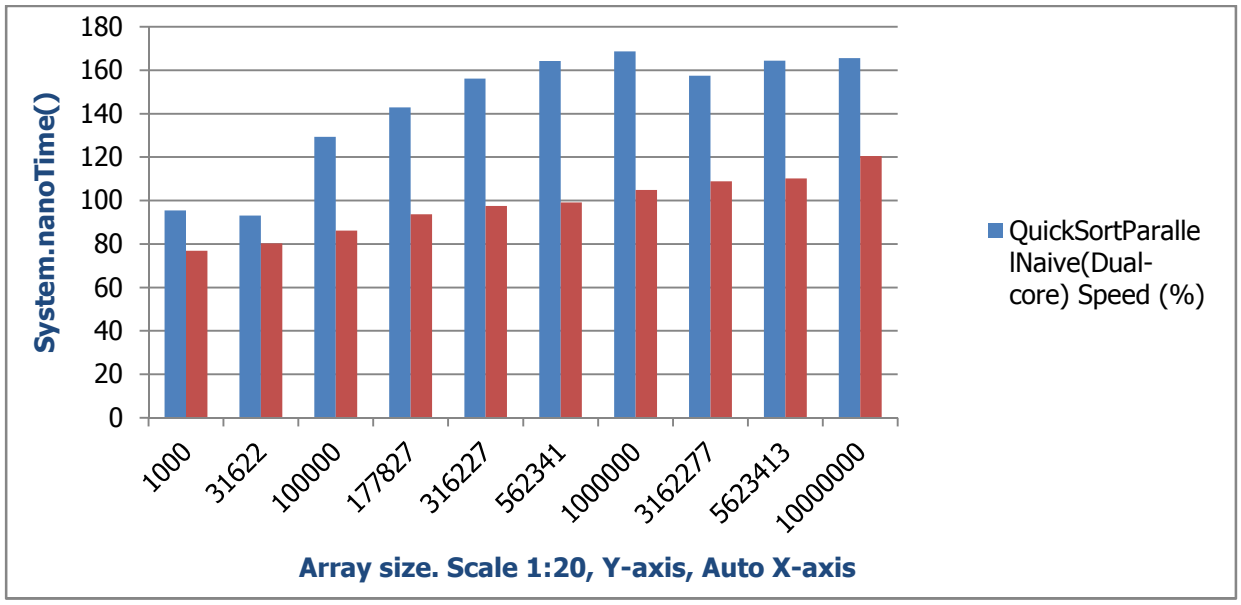


Figure 4.20: Speed of QuickSortFork/Join on both dual and single core machine.

Same conclusion can be drawn as in QuickSortParallelNaive. From Figure 4.20, Starting from the smallest array size, dual-core machine exhibits better performance when compared with the single-core throughout the sorting process with a speed of 95% and 76.9% respectively. With the largest array size, the highest speed of 165.6% and 120.5% were obtained on both dual-core and single-

core respectively. An improved speed of 45.1% was obtained on dual-core machine.

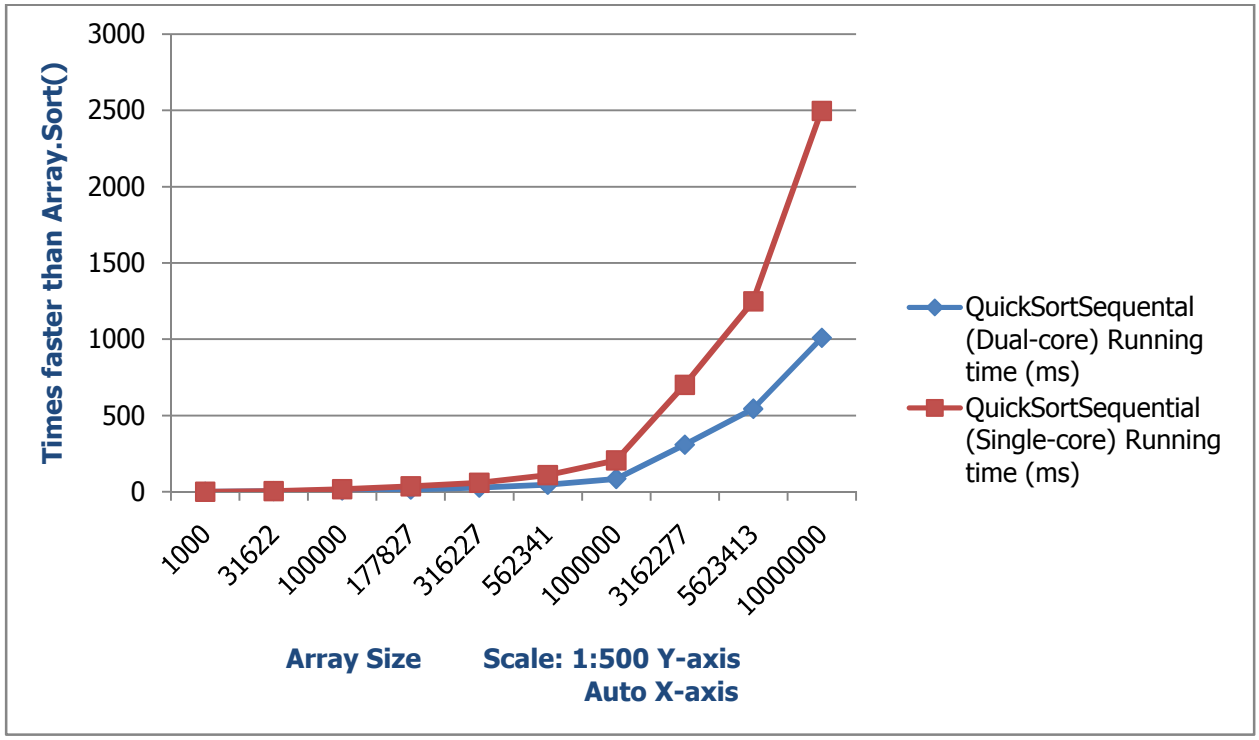


Figure 4.21: Running time of QuickSortFork/Join on both dual and single core machine.

It can be seen From Figure 4.21 that Fork/Join performs better on a dual-core with less running time of 1008ms when compared with 2495ms obtained on a single-core when sorting 10,000000 array elements. Speed increased of 1487ms was obtained on a dual-core machine. The results of all these is that running a program on single-core machines takes more time to execute than on a dual-core machine.

4.4 Search algorithm implementations

As done during Quicksort implementations, here, two searching algorithms are developed; one sequential and the other parallel implementation of the same

algorithms. This is to be able to see the effect of going from serial to parallel when running the same program on both single and dual-core machines.

4.4.1 Linear search algorithm implementation.

Linear search implementation presented in Figure 4.4.1 is based on the Pseudo-code described in Section 3.3.2, Chapter Three (Methodology).

```
7   package binarysearch;
8   // ... the code being measured ...
9
10  public class MyLinearSearch {
11
12      public static int linerSearch(int[] arr, int key){
13          int size = arr.length;
14          for(int i=0;i<size;i++){
15              if(arr[i] == key){
16                  return i;
17              }
18          }
19          return -1;
20
21      }
22
23      public static void main(String a[]){
24          int runs=2000;
25          long startTime = System.nanoTime();
26          int cores= 2;
27          int searchKey = 1234;
28          int numberOfElement=10;
29          for (int i = 0; i < runs; i++) {
30          int[] arr1= {23,45,21,55,234,1,34,90,1,1234 };
31           System.out.println( " Number of Cores: " + cores);
32          System.out.println("Number of Element: "+numberOfElement);
33          System.out.println("Key "+searchKey+" found at index: "+linerSearch
34          System.out.println( "Total Time: " + (System.nanoTime() - startTime
35          }
```

Figure 4.4.1: Linear Search implementation.

With only variable declarations and name differences, a similarity to the code in section 3.3.2 can be noticed.

The fundamental steps involved in the implementation of linear search also known as sequential search are rather simple:

1. Call linear Search during each iteration, to look for an item within the array one at a time and in sequence.
2. If the item is present, then key is returned and its index.
3. If the item is not present then a -1 is returned and the index is zero.

4.4.2 Linear search test runs on both single and dual core machines:

Table 4.4.2: Running Time of Linear search algorithm on both single and dual-core machine

Array Size	Linear Search Running Time (Single-core) ms	Linear Search Running Time (dual-core) ms
10	0.011	0.007
20	0.031	0.011
50	0.032	0.014
100	0.034	0.021
200	0.041	0.032
400	0.056	0.046
800	0.072	0.058
1200	0.085	0.064
2400	0.112	0.101

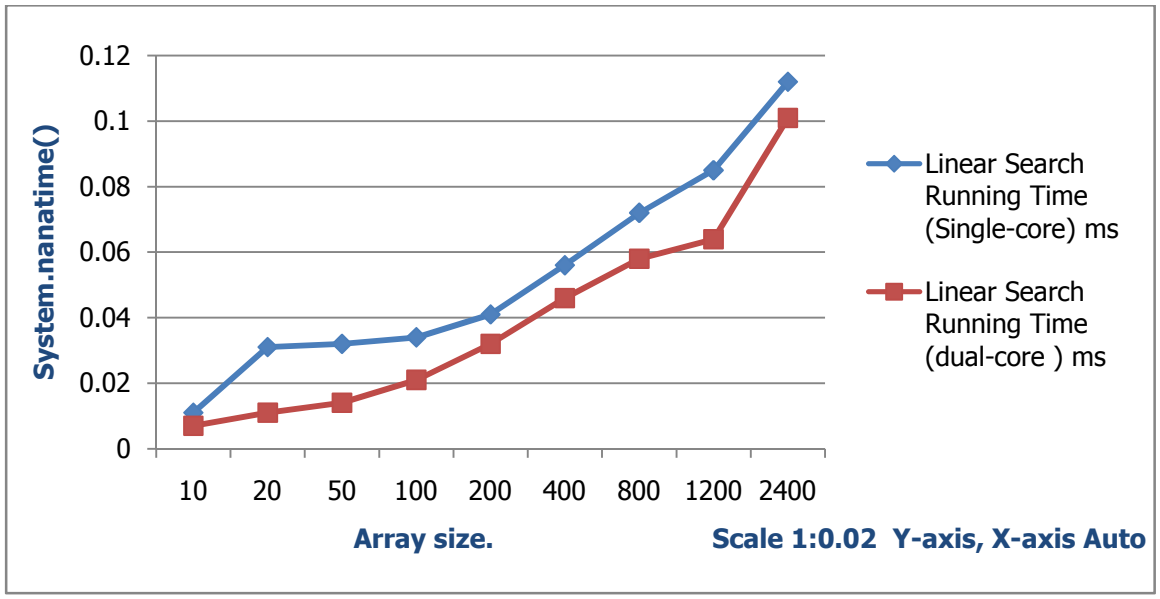
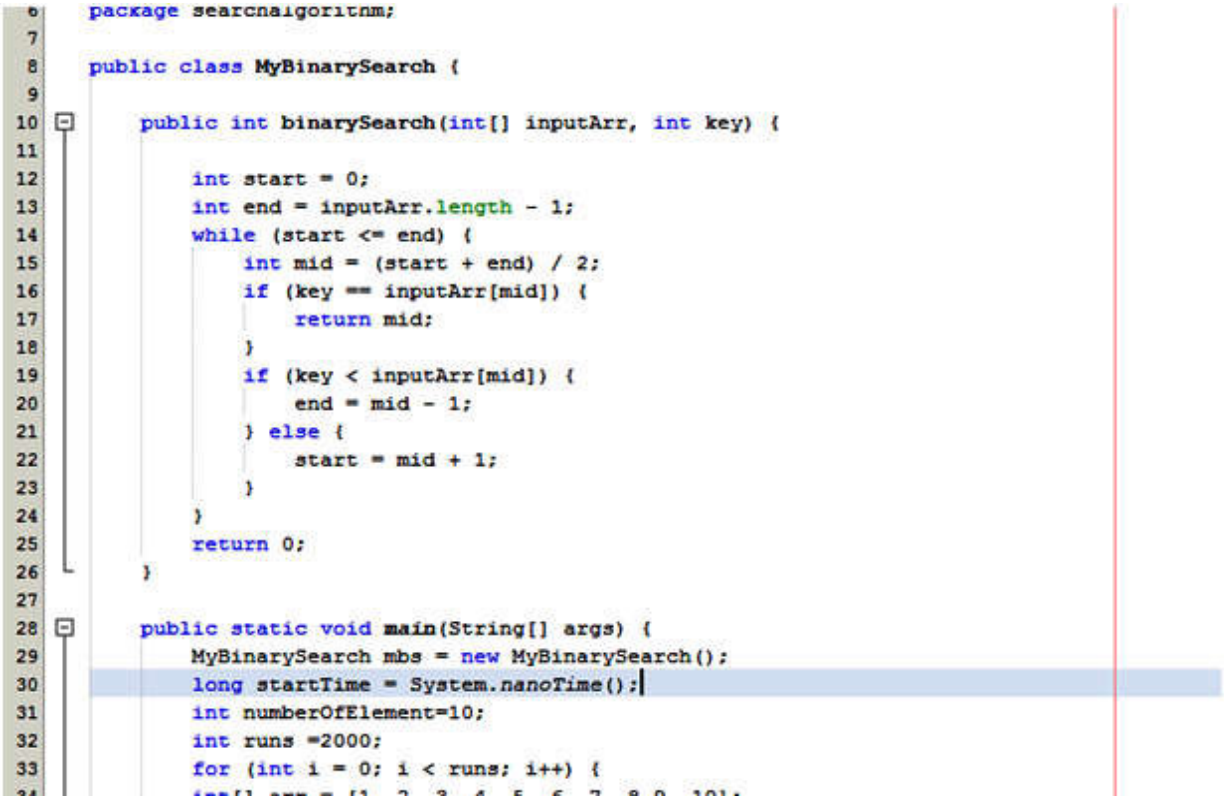


Figure 4.4.2: Running time of Linear search on both dual and single core machine.

From Figure 4.4.2, test was run 2000 times to warm up the computation process thereby reducing the effect of background process on the subsequent results (this effect has been explained in chapter three, methodology). When the array size changes from 10 to 20 the number of tests runs was decreased to 1750 as the computation process started to stabilize. In the subsequent readings, number of test runs keeps decreasing because as the array grew larger so did the computation time, therefore doing many tests run starts to take unnecessary amount of time. From the beginning to the end of the searching process when the array elements reached 2400, Linear search on a dual-core machine recorded 0.101ms while 0.112ms was obtained on a single-core machine. Decreased running time of 0.011ms is achieved on a dual-core machine, thereby exhibiting better performance.

4.4.3 Binary search algorithm implementation.

Binary search implementation presented in Figure 4.4.3 is also based on the Pseudo-code described in section 3.4.2, Chapter Three (Methodology).



```
package searchalgorithm;

public class MyBinarySearch {

    public int binarySearch(int[] inputArr, int key) {

        int start = 0;
        int end = inputArr.length - 1;
        while (start <= end) {
            int mid = (start + end) / 2;
            if (key == inputArr[mid]) {
                return mid;
            }
            if (key < inputArr[mid]) {
                end = mid - 1;
            } else {
                start = mid + 1;
            }
        }
        return 0;
    }

    public static void main(String[] args) {
        MyBinarySearch mbs = new MyBinarySearch();
        long startTime = System.nanoTime();
        int numberOfElement=10;
        int runs =2000;
        for (int i = 0; i < runs; i++) {
```

Figure 4.4.3: Binary search implementation.

Without variable declarations and name differences, a similarity to the code in Section 3.4.2 can also be noticed.

The fundamental steps involved in the implementation of Binary search are different from that of sequential search. Binary Search is based on divide and conquers approach, thus making it parallel algorithm. Each iteration eliminates half

of the remaining possibilities. This makes binary searches very efficient - even for large collections.

The steps are as follows:

Step 1: Finds the position of a specified value (the input "key") within a sorted array

Step 2: In each step, the algorithm compares the input key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and so its index, or position, is returned.

Step 3: Otherwise, if the sought key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the input key is greater, on the sub-array to the right.

Step 4: If the remaining array to be searched is reduced to zero, then the key cannot be found in the array and a special "Not found" indication is returned.

4.4.4 Test runs of Binary search on both single and dual core machine.

Table 4.4.4: Running time of Binary search on both single and dual-core

Array Size	Running Time of Binary Search (Single-core) ms	Running Time of Binary Search (Dual-core) ms
10	0.005	0.002
20	0.011	0.005
50	0.016	0.009
100	0.021	0.013
200	0.031	0.016
400	0.038	0.022
800	0.052	0.031
1200	0.059	0.041
2400	0.087	0.064

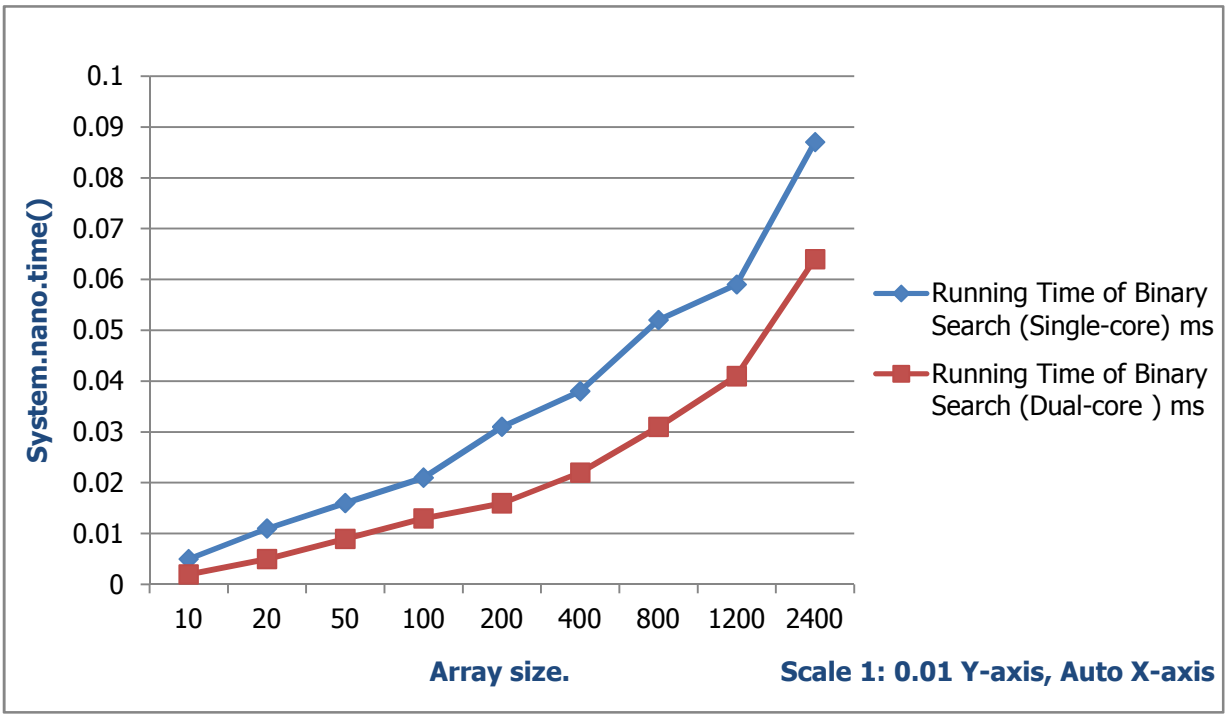


Figure 4.4.4: Running time of binary search on both single and dual-core machine

From Figure 4.4.4, Starting from the smallest array element (Array of size 10), 0.005ms running time was recorded on a single core machine while 0.002ms was recorded on dual-core machine. As the number of elements increases the performance of the dual-core becomes more noticeable throughout the searching process. The highest running time recorded on the dual-core is 0.064ms while 0.087.ms was recorded on the single core. Therefore, a decreased running time of 0.023ms was recorded on the dual-core. This makes it faster than single core machine.

4.4.5 Test runs of Binary and Linear search algorithms on a single core.

Here, binary and linear search algorithms are compared on a single core machine. The results obtained from the experiment presented in Table 4.4.5 is used to generate the graph seen in Figure 4.4.5.

Table 4.4.5: Running time of Binary and Linear search algorithms on single-core machine.

array size	Running Time of Linear Search (Single-core) ms	Running Time of Binary Search (Single-core) ms
10	0.011	0.005
20	0.031	0.011
50	0.032	0.016
100	0.034	0.021
200	0.041	0.031
400	0.056	0.038
800	0.072	0.052
1200	0.085	0.059
2400	0.104	0.087

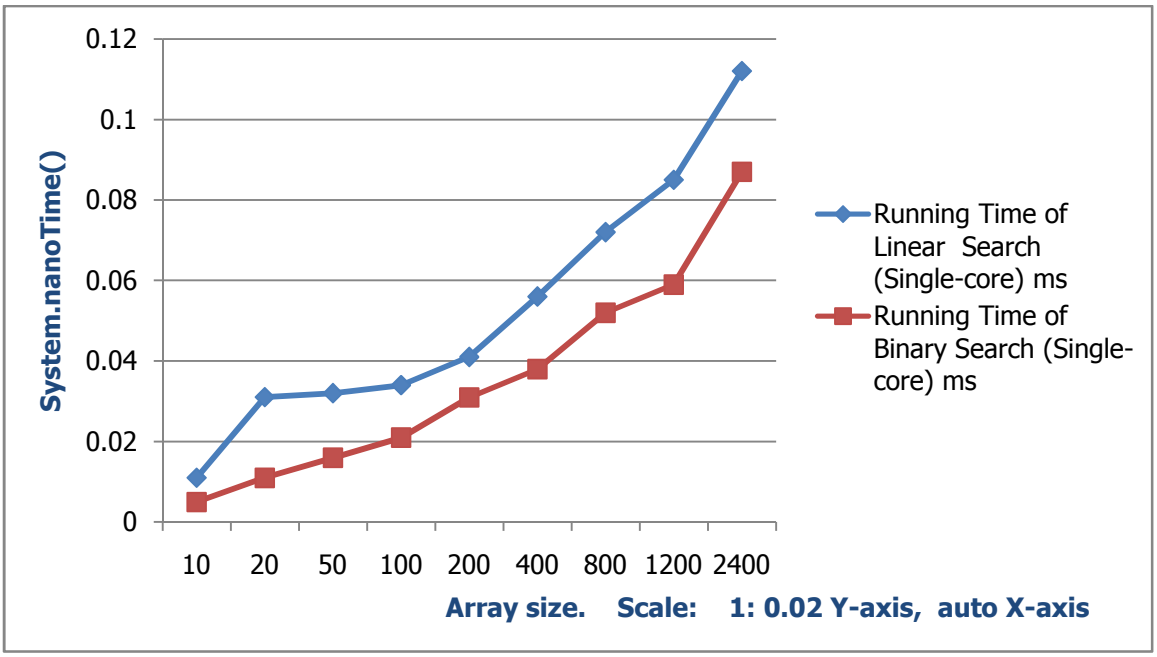


Figure 4.4.5: Running time of binary and linear search on single-core machines.

It can be clearly seen from Figure 4.4.5 that the binary search has better running time than linear search. Considering the smallest array elements, the running time recorded by linear search is 0.011ms while 0.005ms was recorded by binary search. A decreased running time of 0.006ms was achieved by binary search. At the end of the processes, linear search recorded 0.104ms while binary search recorded 0.087ms, having 0.017ms less than linear search. Therefore, binary search exhibits better performance throughout the searching process.

4.4.6 Test runs of Binary and Linear search algorithms on dual-core.

In this section, binary and linear search performances on a dual-core machine are compared. The result obtained is tabulated in Table 4.4.6 and the graph plotted in Figure 4.4.6.

Table 4.4.6: running time of linear and binary search on a dual-core machine

Array size	Linear Search Running Time (dual-core) ms	Running time of Binary search (dual-core)
10	0.007	0.002
20	0.011	0.005
50	0.014	0.009
100	0.021	0.013
200	0.032	0.016
400	0.046	0.022
800	0.058	0.031
1200	0.064	0.041
2400	0.112	0.064

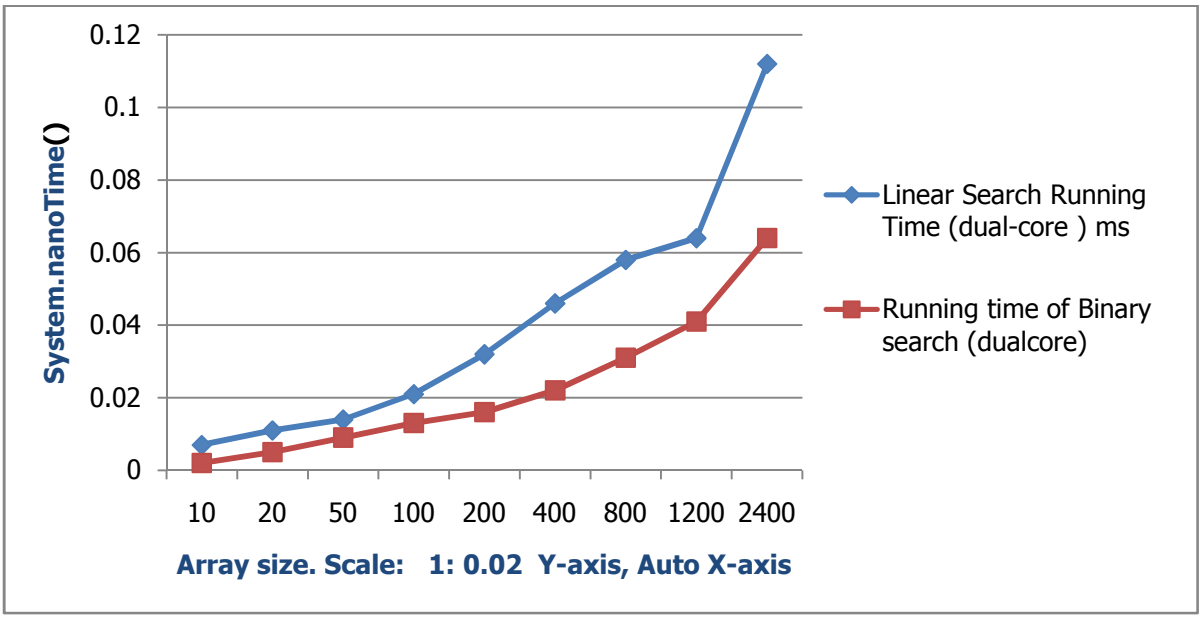


Figure 4.4.6: Running time of linear and binary search on dual-core machine

Same conclusion on the result obtained here can be drawn as with the case of the single core machine plotted in Figure 4.4.5. Therefore, From Figure 4.4.6, it can also be seen that the performance of binary search is much better than linear search. Considering the smallest array elements, the running time recorded by linear search is 0.007ms while 0.002ms was recorded by binary search. A decreased running time of 0.005ms was achieved by using binary search. At end of the processes, linear search recorded 0.112ms while binary search recorded 0.064ms. Thus, binary search recorded 0.048ms less than linear search. Therefore, binary search exhibits much better performance throughout the searching process.

4.4.7 Performance of Quick sort and Linear search algorithms on a single-core.

To correctly compare the times it takes to sort and search for an element of a given array, a sequential sort and linear search algorithms are compared. Parallel sorting and searching algorithms are also compared for convenience because they work in a similar ways. The result is presented in Table 4.4.7 and the graph is shown in figure 4.4.7

Table 4.4.7: Running time of Quick sort sequential and Linear search on a single-core

Array size	Running time of QuicksortSequential (single-core)ms	Running time of Linear Search (single-core)ms
10	0.005	0.011
20	0.015	0.031
50	0.019	0.032
100	0.023	0.034
200	0.028	0.041
400	0.034	0.056
800	0.052	0.072
1200	0.064	0.085
1400	0.084	0.104

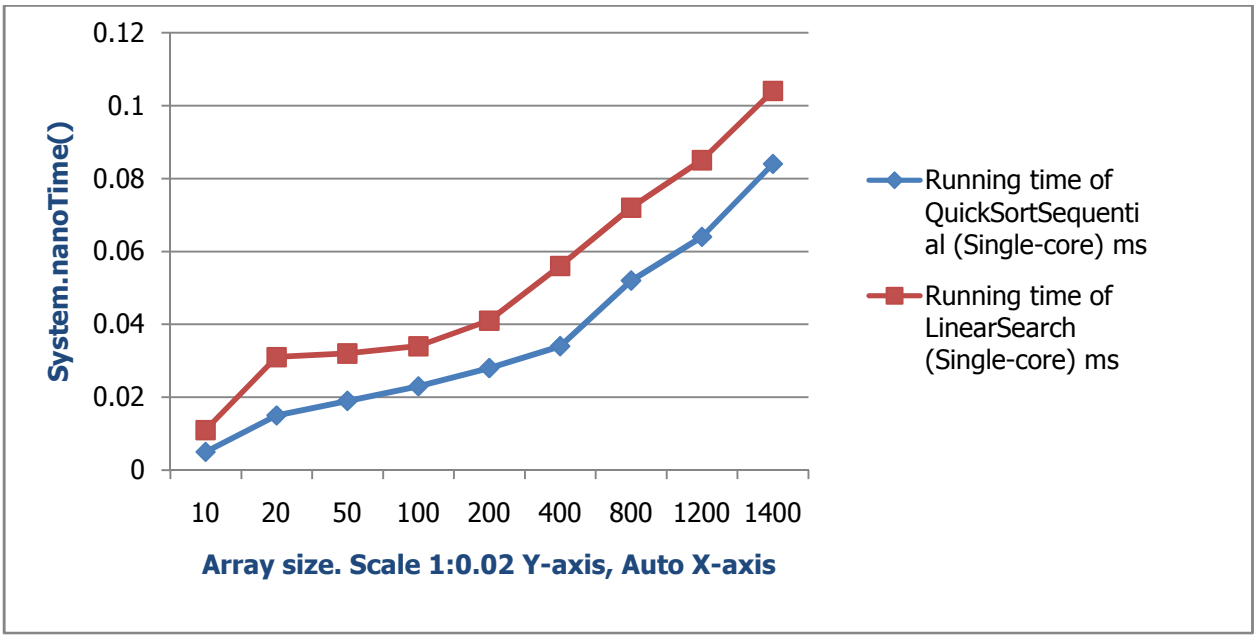


Figure 4.4.7: Running time of Quick sort sequential and Linear search on single-core.

It can be observed from Figure 4.4.7 Quick sort is faster than Linear search algorithm. The running time obtained with Quick sort is 0.005ms for the smallest element while Linear search recorded 0.011ms. Therefore sorting operation takes

less time to perform than searching. This process continues until the number of elements reaches 2400 when the running time recorded by Quicksort sequential increased to 0.084ms while 0.104ms was obtained with Linear search. A reduced running time of 0.02ms was recorded by Quick sort sequential. Therefore sorting process is performed faster than searching on a single-core machine.

4.4.8 Performance of Quick sort and Linear search algorithms on dual-core.

Test runs to determine the best running time between Quick sort and search algorithms is repeated on a dual-core machine with the specification of hardware and software described in section 3.5.1, chapter three. Figure 4.4.8 and Table 4.4.8 present the results obtained from the experiment.

Table 4.4.8: Running time of Quick sort sequential and Linear search algorithms on a dual-core.

Array size	Running time of QuickSort Sequential (dual-core) ms	Running time of Linear Search (dual-core) ms
10	0.002	0.007
20	0.003	0.011
50	0.004	0.014
100	0.007	0.021
200	0.011	0.032
400	0.016	0.046
800	0.021	0.058
1200	0.031	0.064
2400	0.042	0.101

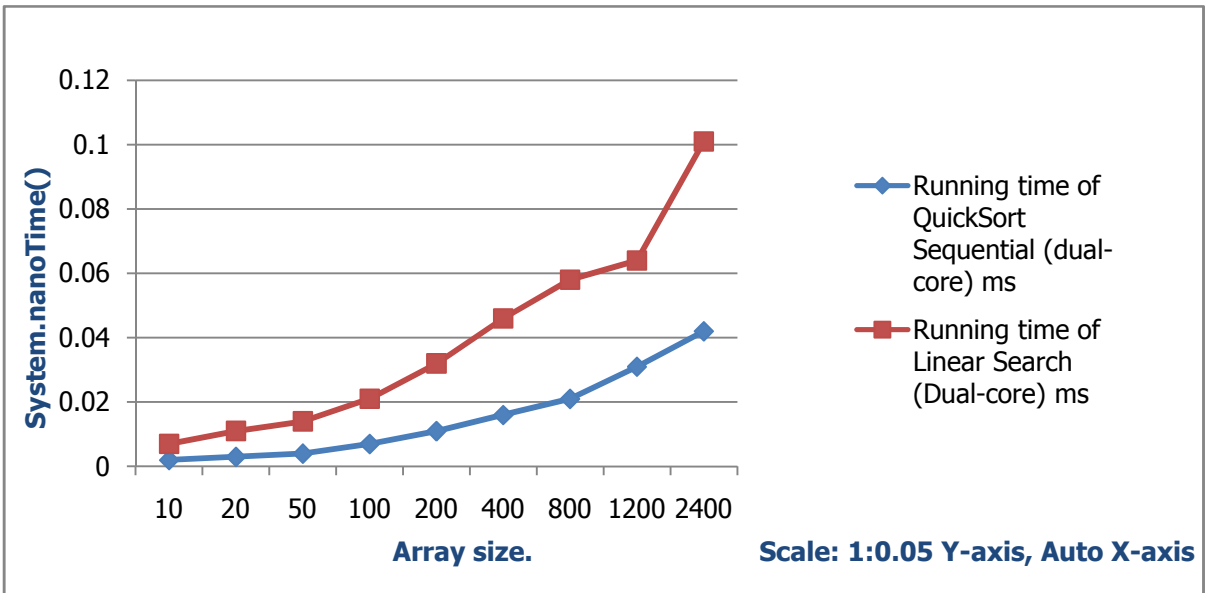


Figure 4.4.8: Running time of Quick sort sequential and Linear search on Dual-core.

The same conclusion can be drawn as with the test carried out on a single-core in section 4.4.7. It can be seen in Table 4.4.8 and graph in Figure 4.4.8 that Quick sort is faster than Linear search algorithm with the smallest array size. A running time of 0.002ms and 0.007ms were recorded by Quick sort and Linear search respectively. A difference of 0.005ms was recorded by Quick sort. This makes it much faster than Linear search algorithm on a dual-core machine. The performance of quick sort becomes more noticeable throughout the experimental process. At the end, Quick sort recorded 0.042ms while Linear search recorded 0.101ms respectively. Therefore, a decreased running time of 0.059ms was achieved by Quick sort thereby making it a better algorithm on both single and dual-core machine.

4.4.9: Performance of Quick sort parallel and Binary search algorithms on a single-core machine.

Here, for convenience, a running time of parallel implementations of quick sort and binary search algorithms are compared using single core machine. Both algorithms work based on divide and conquer approach to solving a problem. The reason for this is that comparing two algorithms that solve problem in the same way will enable us get results that can be correctly compared.

Table 4.4.9: Running time of Quick sort and Search algorithms on single-core machine.

Array Size	Running time of QuickSortForkJoin (Single-core)ms	Running time of BinarySearch (Single-core)ms
10	0.009	0.005
20	0.013	0.011
50	0.019	0.016
100	0.025	0.021
200	0.035	0.031
400	0.042	0.038
800	0.061	0.052
1200	0.078	0.059
2400	0.102	0.087

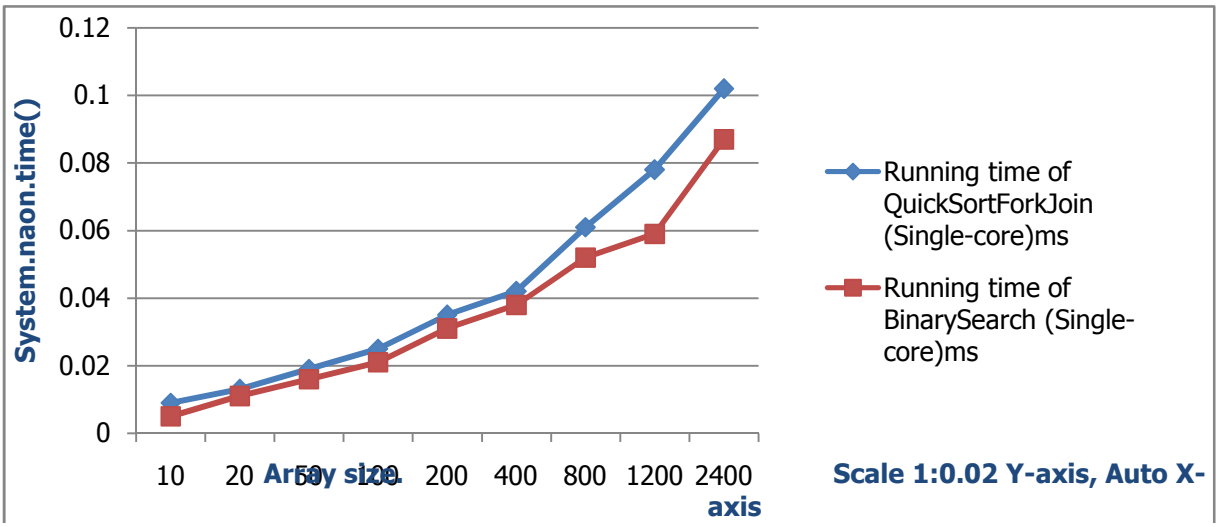


Figure 4.4.9: Running time of QuickSortForkjoin and Binary search on a single-core.

From Figure 4.4.9, unlike linear search, Binary search is more efficient with less running time. Running time of 0.005ms was obtained using Binary search while 0.009ms was obtained with Quick sort fork join respectively. Binary such continues to show better performance throughout the processes recording 0.087ms while Quick sort fork join recorded 0.102ms with the largest number of element respectively. A difference of 0.004ms and 0.015ms was obtained using binary search algorithm to search for an element within the smallest and largest array. The better performance of a binary search is not surprising, because for a binary search to be performed effectively, the array must be sorted first. However, if the sorting time is to be added to the running time of binary search, it will be less efficient than linear search and Quick sort.

4.5: Performance of parallel Quick sort and Binary search algorithms on a dual-core machine.

Test runs carried out in section 4.4.9 is repeated on a dual core machine with the same hardware and software specification given in section 3.5.1, Chapter three. The results of all these is presented in Table 4.5 and the graph in Figure 4.5.

Table 4.5.1: Running time of Quick sort and Search algorithms on a dual-core machine.

Array Size	Running time of QuickSortForkJoin (Dual-core)ms	Running time of BinarySearch (Dual-core) ms
10	0.012	0.002
20	0.018	0.005
50	0.019	0.009
100	0.021	0.013
200	0.034	0.016
400	0.042	0.022
800	0.068	0.031
1200	0.111	0.041
2400	0.225	0.064

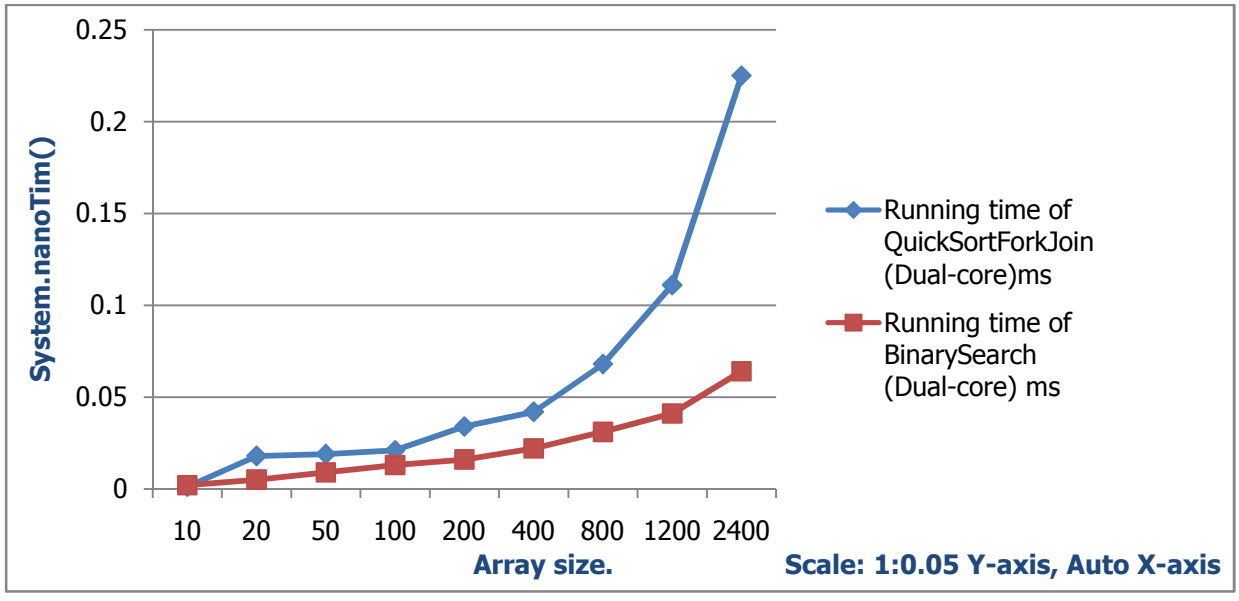


Figure 4.5: Running time of QuickSortForkJoin and Binary search on a dual-core machine.

From Figure 4.5, like its binary counterpart tested on the single-core machines, binary search performances become more noticeable on a dual-core machine. Binary search recorded 0.002ms at the beginning while QuickSortForkJoin recorded 0.012ms respectively. Thus, a decreased running time of 0.010ms was achieved with binary search tested on the dual-core machine. At the end of the sorting and searching processes, Binary search recorded 0.064ms whereas 0.225ms was recorded by QuickSortForkJoin respectively. Therefore, 0.161ms decreased in the running time was achieved using binary search. The reason for the best performance of binary search exhibited here can be divided into two. One is that, the running time of the sorted array is not considered in binary searching as the array to be searched must be sorted first, and the other is actually due to the improved performance of the dual-core machine.

CHAPTER FIVE

(SUMMARY CONCLUSION ANDRECOMMENDATION)

This chapter summarizes the whole work. General conclusion is also drawn. Precautions to be taken when constructing parallel implementation of sequential algorithms has also been discussed, this is followed by recommendations for further research on the topic**.**

6.1 Summary

General background and outline of all the chapter were given in chapter one. Aim and objectives, problems statement, importance of the research, scope and limitations were discussed. In chapter two, related literatures were reviewed. In chapter three, the method to be followed to carry out the experiment was described. Algorithms analysis, data size, specifications of the hardware and software used in benchmarking as the method used to measure the speed and time of sorting and searching algorithms have been given. Experiment is carried out in chapter four; Sequential and parallel version of searching and sorting algorithms were implemented. Results were tabulated and discussed. The results showed that machine with more number of cores gives less running time (higher speed) and Quick sort sequential works faster than Linear searching algorithm on a single core machine whereas Binary searching algorithm is faster on a dual-core machine. Summary, conclusion and recommendations were given in chapter five.

6.2 Conclusion

From the test carried out on all the three sorting algorithms and both linear and binary search, it can be concluded that the binary search is more efficient searching technique than linear search and its performance on dual-core was found to be twice that of a single-core machine. It can also be concluded that Quick sort fork/join is the best sorting algorithms than Quick sort sequential and Quick sort parallel naïve on larger array due to its divide and conquer and work stealing nature when measured on both single and dual-core machine. Quick sort sequential does better with smaller array size. However, Binary search shows better performance than Quick sort fork/join which was found to be the best sorting algorithm. The best performance of binary search may be due to the fact that sorting time is not considered in binary searching. It can also be concluded that dual-core machine enhances performance of concurrent program more than single-core machine. Other popular sorting algorithms such as the heap sort and shell sort from these categories have the potentiality to show improved performance by using the same approaches. These methods and the mentioned algorithms deserve future research.

6.3 Precautions

With the research and implementation that has been done for this dissertation, following precautions must be taking when constructing parallel versions of already existing sequential algorithms using the Java concurrency library:

1. Avoid old concurrent tools - At least for Java, the only tools available before version 5 were a couple of low-level concurrent primitives. These are all difficult to use correctly which potentially could induce dead-lock, starvation or other safety issues. The more modern concurrency tools found in the standard java.util.concurrent package help create a more easy to use and much safer environment when working with concurrent programming.
2. Use specific tools for the job - Different problems may require different tools to achieve the best performance. Certain problems like divide-and-conquer may find the Fork/Join framework, which was specific constructed for these problems, to be the best tool as shown in the result in this research.
3. Know when to parallelize - Parallelized CPU intense parts of a program, but also takes notice of the computation time for these parts. If it takes more time to set-up, spilt into tasks and initiates the threads then it takes to just compute the part sequential, the choice is obvious.
4. Try to reduce or remove synchronization - Shared data can be a problem with parallelization, as the number of threads that needs access to the same

data increases so does the number of synchronizations. This will cause a huge effect on the performance.

5. Educate for concurrency – Developing traditional software varies a lot from implementing concurrent software. Developers need to be aware of the tools available for concurrency on the programming language they use, and get to know various design patterns suitable for concurrency. The ability to write concurrent software that scales well on multi-core architectures is very helpful, but also be cautious on the different safety issues with parallel computations.

6.4 Future work

From the work that has been carried out in this dissertation; the following are recommended for further research on the topic:

1. It would be interesting to test the Fork/Join framework on other algorithms to find out if it would conclude the same results we got with Quicksort and Binary Search. Because of our conclusion we have reason to believe that Fork/Join is a better quick sort while binary is a better search algorithm.
2. Using algorithms that involved synchronizations of the shared data is also an area of another research as the algorithms used in this dissertation involved no shared data. Designing such algorithms and comparing it with the one used here will be an interesting work to perform.

3. Testing this program on a machine with more number of cores such as Quad-core and Octa-core to see if we can conclude the same results as the one obtained here will also be an interesting work to perform. Because of our conclusion, we have reason to believe that Octa-core will have better performance than Quad-core.

Reference

[1] Md. Khairullah, “Enhancing Worst Sorting Algorithms” International Journal of Advanced Science and Technology, Vol. 56, (2013), pp. 1-14.

[2] Alnihoud, J., and Mansi, R., An Enhancement of Major Sorting Algorithms, The International Arab Journal of Information Technology, vol.7 **(2010)** pp. 55-62.

[3] Bingo Sort, http://xlinux.nist.gov/dads/HTML/bingosort.html, retrieved on 23/06/2016.

[4] Exact-Sort, http://www.geocities.ws/p356spt/, retrieved on 23/06/2016.

[5] Lipscz, S., “Theory and Problems of Data Structure”, McGraw Hill Book Company, **(2013)**, pp 30-40.

[6] H. W. Thimbleby, “Using Sentinels in Insert Sort”, Software-Practice and Experience, vol. 19, no. 3, **(1989)**, pp. 303-307.

[7] T. S. Sodhi, S. Kaur and S. Kaur, “Enhanced Insertion Sort Algorithm”, International Journal of Computer Applications, vol. 64, no. 21, **(2013)**, pp. 35-39.

[8] M. A. Bender, M. Farach-Colton and M. A. Mosteiro, “Insertion Sort is O(n log n)”, Proceedings of the Third International Conference on Fun With Algorithms (FUN), **(2004)**, pp. 16-23.

[9] E. Kapur, P. Kumar and S. Gupta, “Proposal of a two way sorting algorithm and performance comparison with existing algorithms”, International Journal of Computer Science, Engineering and Applications (IJCSEA), vol. 2, no. 3, **(2012)**, pp. 61-78.

[10] Mr. Niraj Kumar, Mr. Rajesh Singh, “Performance comparison of sorting algorithms on the basis of complexity”, International Journal of Computer Science and Information Technology Research, Vol. 2, **(2014)** pp. 394-398

[11] Nilsson, S., “Radix sorting & searching”, Ph.D. thesis, Department of Computer Science, Lund University, Lund, Sweden, **(1996)**, pp. 4-56.

[12] Jon L. Bentley, Robert Sedgwick, “Fast algorithms for sorting and searching strings”, Proceedings of the eighth annual ACM-SIAM symposium on discrete algorithms, New Orleans, Louisiana, United States, **(1997)**, pp. 360-369,

[13] Shin-Jae, “Partitioned Parallel sort 1”, IDEA**L** Journal of Parallel & Distributed, Computing, **(2002)**, vol. 3, pp. 4-10.

[14] Santorn, “Adapting Radix Sort to the Memory Hierarchy”, Journal of Experimental Algorithmic (JEA), vol. 2, **(2007)**, pp.7-9.

[16] Macllory, Norton, A., & John, T. R. “Parallel Quicksort Using Fetch-And Add”. *IEEE Trans. Computing*., **(1996)**, pp. 133-138.

[17] Santorn “Adapting Radix Sort to the Memory Hierarchy”, Journal of Experimental Algorithmic (JEA), **(2007)**, pp.7-9.

[18] Naila Rahman, Rajeev Raman, “Adapting Radix Sort to the Memory Hierarchy”, Journal of Experimental Algorithmic (JEA), **(2001)**, pp. 6-9.

[19] Ahmed M. Aliyu, Dr. P. B. Zirra, “A Comparative Analysis of Sorting Algorithms on Integer and Character Arrays” Computer Science Department, Adamawa State University, Mubi, International Journal of Engineering and Science (IJES), **(2013)**, Vol. 2, pp. 25-30

[20] A. S. Leon, etel, “The UltraSPARC T1 processor: CMT reliability,” *in Proc. IEEE* Custom Integrated Circuits Conf., Sep. **(2006)**, pp. 555–562.

[21] Nitin Chaturvedi, S. Gurunarayanan “Study of Various Factors Affecting Performance of Multi-Core Processor” International Journal of Distributed and Parallel Systems (IJDPS) Vol.4, No.4, **(July 2013)**, pp. 3-13

[22] Fasiku, Oyinloye, Falaki, Adewale, “A Review of Architectures - Intel Single Core, Intel Dual Core and AMD Dual Core Processors and the Benefits”, International Journal of Engineering and Technology, Vol. 2, **(May 2012)**, pp. 2-9

[23] Nikhil Srivastava, Vineet Pandey, Ruchi Pathak, Abhishek Pandey "Multicore: Move to the Future" International Journal of Engineering Trends and Technology, Vol. 4 Issue2, **(2013)**, pp. 1-3.

[24] IRakhee Chhibber, II Dr. R.B. Garg, "Multicore Processor, Parallelism and Their Performance Analysis" International Journal of Advanced Research in Computer Science & Technology (IJARCST 2014), Vol. 2, Issue 3 **(July - Sept. 2014)**, pp. 1-7.

[25] Java SE 7 API, 2012. http://docs.oracle.com/javase/7/docs/api/. Retrieved on 27/07/2015.

[26] Derek L. Bruening, "Systematic Testing for Multithreaded Programs" master's thesis, MIT, Cambridge, MA, USA, **(1999)**, pp. 1-104.

[27] Derek Bruening and John Chapin, "Systematic Testing for Multithreaded Programs Technical Report", MIT/LCS, Cambridge, MA, USA, **(2000)**, No. LCS-TM-607, pp. 3-14.

[28] Stefan Savage "A Dynamic Data Race Detector for Multithreaded Programs in ACM Trans. Computer System 15", ACM Press, New York, NY, USA, **(1997)**, No. 4, pp. 391–411.

[29] Ron Jefferies, “Extreme Programming Installed” Addison-Wesley, Boston, MA, USA, **(2001)**, pp. 67-109

[30] Ron Jefferies, XProgramming.com, http://www.xprogramming.com retrieved on 28/07/2016

[31] JUnit Project, JUnit Website, http://www.junit.org, retrieved on 24/06/2016

[32] TestNG Project, TestNG Website, http://testng.org, retrieved on 24/06/2016

[32] Cormac Flanagan, “Types for Safe Locking in Proceedings of the Eighth European Symposium on Programming, Springer, Berlin” **(1999)**, pp. 91–108.

[33] Cormac Flanagan and Stephen N. Freund, “Type-Based Race Detection for Java in ACM SIGPLAN Notices”, ACM Press, New York, NY, USA, **(2000),** Vol. 35, pp. 219–232.

[34] Christian Skalka and Scott Smith, “Static Enforcement of Security with Types in ACM SIGPLAN Notices”, ACM Press, New York, NY, USA, **(2000**), pp. 34-45.

[35] Robert DeLine and Manuel, “Enforcing high-level protocols in low-level software in PLDI Proceedings of the ACM SIGPLAN” Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, **(2001),** pp. 59–69.

[36] Maurice Herlihy, “Wait-free synchronization in ACM Transactions on Programming Languages and Systems”, ACM Press, New-York, NY, USA, **(1991),** No. 1, pp. 124–149

[37] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir, Dynamicsized “lock-free data structures in PODC” 02: Proceedings of the 21st Annual Symposium on Principles of Distributed Computing”, ACM Press, New York, NY, USA, **(2002),** pp. 131–131.

[38] Hakan Sundell, “Efficient and Practical Non-Blocking Data Structures” Ph.D. thesis, Chalmers University of Technology and Goteburg University, Goteburg, Sweden, **(2004),** pp. 70-160.

[39] Keir Fraser and Tim Harris, “Concurrent programming without locks in ACM Trans. Computer System”, 25 ACM Press, New York, NY, USA, (2007), No. 2, pp.4-56

[40] University of Cambridge Systems Research Group, “Practical lock-free data structures”, http://www.cl.cam.ac.uk/research/srg/netos/lock-free/, retrieved on 28/09/2016.

[41] FindBugs. URL http://findbugs.sf.net/. Retrieved on 2/6/2015

[42] T.J. Watson, "Libraries for Analysis (WALA)" URL http://wala.sf.net/. Retrieved on 29/4/2015

[43] F. Long, D. Mohindra, R. C. Seacord, and D. Svoboda, "Java Concurrency Guidelines Technical report", **2010.** URL http://www.sei.cmu.edu/reports/10tr015.pdf, Retrieved on 3/7/2015

[44] Z. D. Luo, L. Hillis, "Effective Static Analysis to Find Concurrency Bugs in Java", 10th IEEE Working Conference on Source Code Analysis and Manipulation, **(2010)**, pp. 4-21

[45] K.Aruli, B.Nivetha, G.N.Jayabhavani, Dr. M.Saravanan, "A review on trends in multi-core processor based on cache and power dissipation" International Research Journal of Engineering and Technology (IRJET), Volume: 02, Issue: (**01, Apr-2015**), pp. 1-4

[46] Kevin Kettler, "Technology Trends in Computer Architecture and Graphics Chip Set their Impact on Power Subsystems", *0-7803-8975-1/0511620.00,* IEEE, **(2005)**, pp.1-23

[47] Ahmed Yasir Dogan, David Atienz1, Andreas Burg "Power/Performance Exploration of Single-core and Multi-core Processor Approaches for Biomedical Signal Processing", Lausanne - 1015, Switzerland, **(2015)**, pp. 1-102.

[48] Hanson, S., et al.: "A Low-Voltage Processor for Sensing Applications With Picowatt Standby Mode", IEEE J. Solid-State Circuits 44, **(2009)**, pp. 1145–1155

[49] Dreslinkski, R.G., et al. "An Energy Efficient Parallel Architecture Using Near Threshold Operation", 16th International Conference on Parallel Architecture and Compilation Techniques, Brasov, **(2007)**, pp. 175–188

[50] Yu, P., "An Ultra-Low-Energy Multi-Standard JPEG Co-Processor in 65 nm CMOS with Sub/Near Threshold Supply Voltage", IEEE J. Solid-State Circuits 45, **(2010),** pp. 668–680

[51] Krimer, E., "Synctium: a Near-Threshold Stream Processor for Energy-

Constrained Parallel Applications", Computer Architecture Letters 9, **(2010),** pp. 21–24

[52] Yaser et al., "The Challenges of Multi-Core Processor", International Journal of Advancements in Research & Tech., Volume 2, Issue 6, **(June-2013)**, pp. 1-4

[53] John Darlinton, Moustafa Ghanem, Yike Guo, Hing Wing, "Guided Resource Organisation in Heterogeneous Parallel Computing", Journal of High Performance Computing, Vol. 4, **(2012),** pp 1-5.

[54] Robert Atkey and Donald Sannella, "THREADSAFE: Static Analysis for Java Concurrency" Journal of Electronics Communications (EASST), Proceedings of the 15th International Workshop on Automated Verification of Critical Systems, **(2015),** Vol. 72, pp. 1-16.

[55] Y. Eytani, Havelund, D. Stoller, "Towards a framework and a benchmark for testing tools for multi-threaded programs Concurrency and Computation: Practice and Experience" **(2007)** pp. 267–279

[56] Ankit R. Chadha "Modified Binary Search Algorithm" International Journal of Applied Information Systems (IJAIS), Foundation of Computer Science FCS, New York, USA, Vol.7, **(April 2014)**, pp.1-4

[57] CK. Kumbharana and Vimal P. Parmar, "Comparing Linear Search and Binary Search Algorithms", International Journal of Computer Applications, Volume 121, No.3, **(July 2015),** pp. 1-16.

[58] Kumari , A. and Chakraborty , S., Software Complexity: A Statistical Case Study Through Insertion Sort, Applied Math. and Computer., vol. 190(1), **(2007**), pp. 40-50

[59] Kumari, A. and Chakraborty, S., "A Simulation Study on Quick Sort Parameterized Complexity Using Response Surface Design", International Journal of Mathematical Modeling, Simulation and Applications, Vol. 1, No. 4, **(2008),** pp. 448-458

[60] Knuth, D. E. "The Art of Computer Programming, Sorting and Searching", Vol.3, Addison Wesley, (1997), 3rd ed., pp. 396-408

[61] Deepak Kumar and Manish Sharma, "Binary search is faster than the linear search", international journal of innovative research in technology, **(2014)**,|Vol.1, Issue 5, pp.1-4

[62] Clay Brashear, "The Art of Concurrency", 3rd Edition, Graham, **(2009)**, pp.130-145

[63] Intel Corp. Desktop Products Group., "Hyper-threading technology architecture and micro-architecture", Intel Technology Journal, Vol. 3, **(1980)**, pp. 4–17

[64] Gordon E. Moore, "Cramming more components onto integrated circuits" Journal of Electronics, vol. 1, **(April 1965)**, pp. 8-10.

[65] Macllory, Norton, A., & John, T. R. "Parallel Quicksort Using Fetch-And Add". IEEE Trans. Computing., **(1996)**, pp. 133-138.

27-2015

[66] Ananth, Anshul, George & Vipin “Introduction to Parallel Computing,” 2nd Ed.,Addison-Wesley, **(2007)**, pp. 150-162

[67] Brian Goetz, “Java Concurrency in Practice” 2nd edition, **(2006)**, pp.120-150.

[68] Stallings, W., “Operating Systems Internal And Design Principles” London: Pearson Education Ltd., London, **(2009)**, pp.chp5.

[69] C. A. R. Hoare, “Quicksort in Java”, The Computer Journal, **(2012)**, pp.10–16.

[70] Jon L. Bentley, Robert Sedgwick, “Fast algorithms for sorting and searching strings”, Proceedings of the eighth annual ACM-SIAM symposium on discrete algorithms, New Orleans, Louisiana, United States, **(1997)**, pp.360-369.

[71] J.-P. Briot, R. Guerraoui, K.-P. Löhr: “Concurrency and distribution in object-oriented programming”, *ACM Computing Surveys,* **(1998)** vol. 30, no. 3, pp. 291-329.

[72] D. Lea: “The java.util.concurrent synchronizer framework”. Proc. Workshop on Concurrency and Synchronization in Java Programs (CSJP ’04), **July 2004**, pp. 1-9.

[73] URL: http://www.podc.org/podc2004/csjp-proceed.pdf, retrieved on September, 4/8/2015

[74] D.F. Bacon, R.E. Strom, A. Tarafdar: "Guava: a dialect of Java without data races", Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '00), (**October 2000)**, pp. 382-400.

[75] P. Felber, M.K. Reiter: "Advanced concurrency control in Java", Concurrency and Computation: Practice and Experience, **(April 2012)**, vol. 14, no. 4, pp. 261-285.

[76] J.L. Keedy, G. Menger, C. Heinlein, and F. Henskens: "Qualifying Types illustrated by synchronisation examples", Proc. *Objects, Compo-nents,* Architectures, Services and Applications for a Networked World, Int. Conf. NetObjectDays (NODe 2002), LNCS 2591, **(2013)**, pp. 330-344.

[77] G. Milicia, V. Sassone: "Jeeg: a programming language for con-current objects synchronization", Proc. Joint ACM-ISCOPE Conf. on Java Grande (JGI-02), ACM Press, **(2002)**, pp. 1-5.

[78] R.A. Olsson, A.W. Keen: "The JR Programming Language", Kluwer, **(2004)**, pp.7-40.

[79] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold: “An overview of Aspect”, Proc. 15. European Conf. on Object-Oriented Programming (ECOOP ‘01), LNCS 2072, **(2011)**, pp. 327-353.

[80] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa: “Ab-stracting object interactions using Composition Filters”, Proc. ECOOP ’93 Workshop on Object-Based Distributed Programming, LNCS 791, **(2014)**. pp. 152-184.

[81] D. Holmes: “Synchronization Rings: Composable Synchronization for Object-Oriented Systems”, PhD thesis, Mcquarie University, Sydney, **(1999)**, pp. 1-5.

[82] J.L. Keedy, G. Menger, C. Heinlein, and F. Henskens: “Qualifying Types illustrated by synchronisation examples”, Proc. Objects, Compo-nents, Architectures, Services and Applications for a Networked World, Int. Conf. NetObjectDays (NODe 2002), LNCS 2591, **(2013)**, pp. 330-344.

[83] J.L. Keedy, K. Espenlaub, Ch. Heinlein, G. Menger, M. Evered: “Statically qualified types in Timor”, Journal of Object Technology, vol. 4, no. 7, **(September/October 2005)**, pp. 115-137,

[84] URL http://www.jot.fm/issues/ issue_2005_09/article5, Retrieved on March, 2015.

[85] Md. Anisur Rahman, “An Efficient Concurrency Control Technique for Mobile Database Environment”, Global Journal of Computer Science and Technology, vol.4, no.22, **(2013)** pp.1-5.

[86] Salman Abdul Moiz and Dr. Lakshmi Rajamani “A real time optimistic Strategy to achieve concurrency control in mobile environment using on demand multicasting” International Journal of Mobile wireless network, Vol.2, Issue 2, No. 2, **(2010),** pp.1-6.

[87] K Vijay, "TCOT -A Timeout-Based Mobile Transaction Commitment Protocol," IEEETransactions on Computers, vol. 51, **(2002)** pp. 1212-1218

[88] S.Moiz and L. Rajamani, "Single Lock Manager Approach for Achieving Concurrency Control in Mobile Environments," in 14th International Conference on High-Performance Computing, Goa, India, **(2007)**, pp. 650-660.

[89] M. Salman Abdul, "Concurrency Control Strategy to Reduce Frequent Rollbacks in Mobile Environments," in International Conference on Computational Science and Engineering, **(2009)**, pp. 709-714

[90] S. A. Moiz and M. K. Nizamuddin, “Concurrency Control without Locking in Mobile Environments," in Emerging Trends in Engineering and TechnologyICETET '08, First International Conference, **(2008)**, pp. 1336-1339

[91] D. L. R. Salman Abdul Moiz, "A Real Time Optimistic Strategy to achieve Concurrency control in Mobile Environments using on demand multicasting," International Journal of Wireless & Mobile Networks (IJWMN), vol. 2, **(2010)**, pp. 172-185.

[92] S. Madria, M. Baseer, V. Kumar, and S. Bhowmick, "A transaction model and multiversion concurrency control for mobile database systems," Distributed and Parallel Databases, vol. 22, **(2015)** pp. 165-196

[93] S. K. Madria, M. Baseer, and S. S. Bhowmick, "A Multi-version Transaction Model to Improve Data Availability in Mobile Computing," in International Conferences DOA, CoopIS and ODBASE, **(2002)**, pp. 322-338

[94] A. C. Carmine Cesarano, Vincenzo Moscato, Antonio Picariello, Antonio d'Acierno, "A Hybrid Approach for Improving Concurrency of Frequently Disconnecting Transactions," International Journal of Computer Science and Network Security (IJCSNS), vol. 7, pp. (**2007**), 205-215.

[95] A. Chianese, A. d'Acierno, V. Moscato, and A. Picariello, "Pre serialization of long running transactions to improve concurrency in mobile environments," in Data Engineering Workshop, 2008, ICDEW 2008, IEEE 24th International Conference, **(2008)**, pp. 129-136.

[96] URL: https://en.wikipedia.org/wiki/Binary_search_algorithm, Retrieved on 02/07/2016.

[97] URL: https://http://www.ibm.com/developerworks/library/j-benchmark1/, Retrieved on 02/07/2016.

[98] URL:https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap7a.pdf, Retrieved on 02/07/2016.

[99] URL: http://javadevnotes.com/java-int-array-examples, Retrieved on 09/07/2016.

[100] Ankit R., Rishikesh Misal and Tanaya Mokashi "Modified Binary Search" Algorithm, International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868, Foundation of Computer Science FCS, New York, USA Vol. 7– No. 2, **(April 2014)**, pp. 1-4.

[101] VimalP.Parmar and CK. Kumbharana, "Comparing Linear Search and Binary Search Algorithms to Search an Element from a Linear List Implemented through Static Array,Dynamic ArrayandLinkedList" International Journal of Computer Applications (0975 8887), Vol. 121 ,No.3, **(July 2015)**, pp. 1-17.

[102]https://courses.engr.illinois.edu/cs173/fa2010/lectures/algorithms.pdf, retrieved on 8/18/2016.

Appendix

```
package quicksorttest;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;
import java.math.BigDecimal;
import java.util.Arrays;
import java.util.List;
import java.util.Random;
import java.util.Vector;
//import java.util.Vector;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;
public class QuicksortTesting {
    // Variable Declarations
static int median, size, runs,
seed = 3141592,
cores = Runtime.getRuntime().availableProcessors();
static int[] orginArray, ReferArray, SrtedArray;
// Test Data Structure: { size , runs }
```

```
static int[][] datasArray = new int[][] {
{ 1000,2500}, { 1778,2500}, { 3162,2500}, { 5623,2500},
{ 10000,2500}, { 17782,2500}, { 31622,1250}, { 56234,1250},
{ 100000, 750}, { 177827, 500}, { 316227, 250}, { 562341, 150},
{ 1000000, 100}, { 1778279, 75}, { 3162277, 60}, { 5623413, 50},
{ 10000000, 40}, {17782794, 24}, {31622776, 16}, {56234132, 8},
{100000000, 4}
};
static long startTimer, totalTimer;
static long[][] timerArray;
static String[] names = new String[] {
"Arrays.Sort()", "QuicksortSequential", "QuicksortParallelNaive",
"QuicksortForkJoin", "QuicksortExecutor"
};
static boolean firstRun = true;
public static void main(String[] args) {
totalTimer = System.nanoTime();
    for (int[] datasArray1 : datasArray) {
        size = datasArray1[0];
        runs = datasArray1[1];
        orginArray = new int[size];
        timerArray = new long[names.length][runs];
        initArrays(seed);
        median = (int)runs/2;
        if (median > 0) median--;
```

```
        System.out.println("Size: " + size + " | Runs: " + runs +
                " | Seed: " + seed + " | Cores: " + cores);
        computeArrays();
    }
System.out.println("Done in: " + ((System.nanoTime()-totalTimer)/1000000000.0 + " s\n"));
}
static void computeArrays() {
//-- Built-in Arrays.sort() computation, used as referanse --
for (int i = 0; i < runs; i++) {
ReferArray = orginArray.clone();
startTimer = System.nanoTime();
Arrays.sort(ReferArray);
timerArray[0][i] = System.nanoTime()-startTimer;
//System.out.println((timerArray[0][i]/1000000.0) + " ms");
}
try {
sortAndWrite(0);
} catch (IOException e) {}
//-- Sequential Quicksort computation --
for (int i = 0; i < runs; i++) {
SrtedArray = orginArray.clone();
startTimer = System.nanoTime();
QuickSeq QSeq = new QuickSeq();
QSeq.quicksort(SrtedArray, 0, SrtedArray.length-1);
timerArray[1][i] = System.nanoTime()-startTimer;
```

```
//System.out.println((timerArray[1][i]/1000000.0) + " ms");
}
try {
sortAndWrite(1);
} catch (IOException e) {}
//-- Parallel Naive Quicksort computation --
for (int i = 0; i < runs; i++) {
SrtedArray = orginArray.clone();
startTimer = System.nanoTime();
Thread QParNaive = new Thread(new QuickParNaive(SrtedArray, 0, SrtedArray.length-1));
QParNaive.start();
try {
QParNaive.join();
} catch (InterruptedException e) {}
timerArray[2][i] = System.nanoTime()-startTimer;
//System.out.println((timerArray[2][i]/1000000.0) + " ms");
}
try {
sortAndWrite(2);
} catch (IOException e) {}
//-- Parallel Fork/Join Quicksort computation --
ForkJoinPool QParFJ = new ForkJoinPool(cores);
for (int i = 0; i < runs; i++) {
SrtedArray = orginArray.clone();
startTimer = System.nanoTime();
```

```
QParFJ.invoke(new QuickParFJ(SrtedArray, 0, SrtedArray.length-1));
timerArray[3][i] = System.nanoTime()-startTimer;
//System.out.println((timerArray[3][i]/1000000.0) + " ms");
}
try {
sortAndWrite(3);
} catch (IOException e) {}
//-- Parallel ExecutorService Quicksort computation --
for (int i = 0; i < runs; i++) {
SrtedArray = orginArray.clone();
startTimer = System.nanoTime();
final ExecutorService pool = Executors.newFixedThreadPool(cores);
List<Future> futures = new Vector<>();
QuickParExec QParExec = new QuickParExec(SrtedArray, 0, SrtedArray.length-1, futures,
pool);
futures.add(pool.submit(QParExec));
while(!futures.isEmpty()) {
Future topFeature = futures.remove(0);
try {
if(topFeature!=null)topFeature.get();
} catch(InterruptedException | ExecutionException ie) {}
}
pool.shutdown();
timerArray[4][i] = startTimer-System.nanoTime();
//System.out.println((timerArray[4][i]/1000000.0) 6+ " ms");
```

```
}
try {
sortAndWrite(4);
} catch (IOException e) {}
System.out.println();
firstRun = false;
}
static void initArrays(int seed) {
Random r = new Random(seed);
for (int i = 0; i < orginArray.length; i++) {
orginArray[i] = r.nextInt(orginArray.length);
}
}
static void sortAndWrite(int id) throws IOException {
String fileName = names[id] + ".txt";
Writer out = new OutputStreamWriter(new FileOutputStream(fileName, !firstRun));
Arrays.sort(timerArray[id]);
try {
String s = Long.toString(timerArray[id][median]);
out.write(size + "\t" + runs + "\t" + s + "\n");
} finally { out.close(); }
compareAndPrint(id);
}
static void compareAndPrint(int id) {
double time = timerArray[id][median]/1000000.0;
```

```
BigDecimal bd = new BigDecimal(time);
if (time >= 1000) bd = bd.setScale(0, BigDecimal.ROUND_UP);
else if (time >= 100) bd = bd.setScale(1, BigDecimal.ROUND_UP);
else if (time >= 10) bd = bd.setScale(2, BigDecimal.ROUND_UP);
else bd = bd.setScale(3, BigDecimal.ROUND_UP);
if (id == 0) System.out.print(names[id] + "\t\t: " + bd.toString() + " ms");
else System.out.print(names[id] + "\t: " + bd.toString() + " ms");
if (id > 0) {
bd = new BigDecimal((timerArray[0][median]*1.0/timerArray[id][median]*1.0) * 100);
bd = bd.setScale(1, BigDecimal.ROUND_UP);
System.out.print("\tspeed: " + bd.toString() + " %");
System.out.print((Arrays.equals(SrtedArray, ReferArray) ? "\n" : "\t!Error: not identical to
referance!\n"));
}
else System.out.print("\n");
}
}
//Binary Search codes:
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package searchalgorithm;

public class MyBinarySearch {
```

```
public int binarySearch(int[] inputArr, int key) {

    int start = 0;
    int end = inputArr.length - 1;
    while (start <= end) {
        int mid = (start + end) / 2;
        if (key == inputArr[mid]) {
            return mid;
        }
        if (key < inputArr[mid]) {
            end = mid - 1;
        } else {
            start = mid + 1;
        }
    }
    return 0;
}

public static void main(String[] args) {
    MyBinarySearch mbs = new MyBinarySearch();
    long startTime = System.nanoTime();
    int numberOfElement=10;
    int runs =2000;
    for (int i = 0; i < runs; i++) {
    int[] arr = {1, 2, 3, 4, 5, 6, 7, 8,9, 10};
    System.out.println("Number of Element: "+numberOfElement);
    System.out.println("Key 7's position: "+mbs.binarySearch(arr, 7));
    System.out.println( "Total Time: " + (System.nanoTime() - startTime) / 1000000+ " ms\n");
    }
```

```
numberOfElement=20;
runs=1750;
for (int i = 0; i < runs; i++) {
startTime = System.nanoTime();
int[] arr1 = {1, 2,3, 4, 5, 6,7, 8,9, 10, 11, 12, 13, 14, 15, 16, 17,18,19,20};
System.out.println("Number of Element: "+numberOfElement);
System.out.println("Key 20's position: "+mbs.binarySearch(arr1, 20));
 System.out.println( "Total Time: " + (System.nanoTime() - startTime) / 1000000+ " ms\n");
}
 numberOfElement=50;
 runs=1500;
for (int i = 0; i < runs; i++) {
 startTime = System.nanoTime();
 int[] arr2 = {1,2, 3, 4, 5,34,78,123,432,900,
     100,101,102,103,105,105,106, 108,109, 120,
     122,125,126, 129,130,131,135,137,139,140,
     141,145,147,148,148,149,150,151,152,153,
     154,155,156,157,158,159,160,161,162,163};
 System.out.println("Number of Element: "+numberOfElement);
System.out.println("Key 162's position: "+mbs.binarySearch(arr2, 162));
 System.out.println( "Total Time: " + (System.nanoTime() - startTime) / 1000000+ " ms\n");
}
 numberOfElement=100;
 runs =1250;
 for (int i = 0; i < runs; i++) {
 startTime = System.nanoTime();
 int[] arr3 = {1,2,3,4,5,34,48,50,90,98,
     100,101,102,103,105,105,106,108,109,120,
     122,125,126, 129,130,131,135,137,139,140,
     141,145,147,148,148,149,150,151,152,153,
```

```
    154,155,156,157,158,159,160,161,162,163,
    164,165,166,167,168,169,170,172,173,175,
    179,180,182,183,185,186,188,189,190,191,
    192,193,194,195,196,197,198,199,200,201,
    102,203,204,205,206,207,208,209,210,211};
 System.out.println("Number of Element: "+numberOfElement);
System.out.println("Key 162's position: "+mbs.binarySearch(arr3, 162));
 System.out.println( "Total Time: " + (System.nanoTime() - startTime) / 1000000+ " ms\n");
 }

 numberOfElement=300;
 runs =1000;
 for (int i = 0; i < runs; i++) {
 startTime = System.nanoTime();
 int[] arr4 = {1,2,3,4,5,34,48,50,90,98,
    100,101,102,103,105,105,106,108,109,120,
    122,125,126, 129,130,131,135,137,139,140,
    141,145,147,148,148,149,150,151,152,153,
    154,155,156,157,158,159,160,161,162,163,
    164,165,166,167,168,169,170,172,173,175,
    179,180,182,183,185,186,188,189,190,191,
    192,193,194,195,196,197,198,199,200,201,
    102,203,204,205,206,207,208,209,210,211,
    212,213,214,215,216,217,218,219,220,221,
    222,223,224,225,226,227,228,228,230,233,
    235,236,238,239,239,300,302,303,304,305,
    306,307,308,308,309,400,401,402,403,404,
    405,405,406,407,408,409,410,411,412,413,
    414,415,416,417,418,419,500,501,502,503,
    504,505,506,507,508,509,510,511,512,513,
```

```
    514,516,517,518,519,520,521,522,523,524,
    525,526,527,528,529,530,531,532,533,534,
    535,536,537,538,539,540,542,543,544,545,
    546,547,548,549,550,552,553,553,554,555,
};
System.out.println("Number of Element: "+numberOfElement);
System.out.println("Key 162's position: "+mbs.binarySearch(arr4, 162));
System.out.println( "Total Time: " + (System.nanoTime() - startTime) / 1000000+ " ms\n");

}
numberOfElement=400;
runs =750;
 for (int i = 0; i < runs; i++) {
startTime = System.nanoTime();
int[] arr5 = {1,2,3,4,5,34,48,50,90,98,
    100,101,102,103,105,105,106,108,109,120,
    122,125,126, 129,130,131,135,137,139,140,
    141,145,147,148,148,149,150,151,152,153,
    154,155,156,157,158,159,160,161,162,163,
    164,165,166,167,168,169,170,172,173,175,
    179,180,182,183,185,186,188,189,190,191,
    192,193,194,195,196,197,198,199,200,201,
    102,203,204,205,206,207,208,209,210,211,
    212,213,214,215,216,217,218,219,220,221,
    222,223,224,225,226,227,228,228,230,233,
    235,236,238,239,239,300,302,303,304,305,
    306,307,308,308,309,400,401,402,403,404,
    405,405,406,407,408,409,410,411,412,413,
    414,415,416,417,418,419,500,501,502,503,
    504,505,506,507,508,509,510,511,512,513,
```

```
    514,516,517,518,519,520,521,522,523,524,
    525,526,527,528,529,530,531,532,533,534,
    535,536,537,538,539,540,542,543,544,545,
    546,547,548,549,550,552,553,553,554,555,
    556,557,558,559,560,561,562,563,564,567,
    568,569,570,571,572,573,574,575,576,577,
    578,579,580,581,582,583,584,585,586,587,
    589,590,591,592,593,594,595,596,597,598,
    600,601,602,603,604,605,606,607,608,609,
    610,611,612,613,614,615,616,617,618,619,
    620,621,622,623,624,625,625,627,628,629,
    630,631,632,633,634,635,636,637,638,639,
    640,641,642,643,644,645,646,647,648,649,
    650,651,652,653,654,655,656,657,658,659,
    670,671,672,673,674,675,676,677,678,679,
    680,681,682,684,685,686,687,688,689,690,
    691,692,693,694,695,696,697,698,699,700,
    701,702,703,705,706,707,708,709,710,711,
    712,713,715,716,717,718,719,720,721,722,
    723,724,725,726,727,728,729,730,731,732,
    733,734,735,736,737,738,739,740,741,742,
    743,744,745,746,747,748,749,750,751,752,
    753,754,755,756,757,758,759,760,761,762,
    763,764,765,766,767,768,768,769,770,771};

System.out.println("Number of Element: "+numberOfElement);
System.out.println("Key 771's position: "+mbs.binarySearch(arr5, 771));
System.out.println( "Total Time: " + (System.nanoTime() - startTime) / 1000000+ " ms\n");
}
```

```
numberOfElement=800;
 runs =500;
  for (int i = 0; i < runs; i++) {
 startTime = System.nanoTime();
 int[] arr6 = {1,2,3,4,5,34,48,50,90,98,
     100,101,102,103,105,105,106,108,109,120,
     122,125,126, 129,130,131,135,137,139,140,
     141,145,147,148,148,149,150,151,152,153,
     154,155,156,157,158,159,160,161,162,163,
     164,165,166,167,168,169,170,172,173,175,
     179,180,182,183,185,186,188,189,190,191,
     192,193,194,195,196,197,198,199,200,201,
     102,203,204,205,206,207,208,209,210,211,
     212,213,214,215,216,217,218,219,220,221,
     222,223,224,225,226,227,228,228,230,233,
     235,236,238,239,239,300,302,303,304,305,
     306,307,308,308,309,400,401,402,403,404,
     405,405,406,407,408,409,410,411,412,413,
     414,415,416,417,418,419,500,501,502,503,
     504,505,506,507,508,509,510,511,512,513,
     514,516,517,518,519,520,521,522,523,524,
     525,526,527,528,529,530,531,532,533,534,
     535,536,537,538,539,540,542,543,544,545,
     546,547,548,549,550,552,553,553,554,555,
     556,557,558,559,560,561,562,563,564,567,
     568,569,570,571,572,573,574,575,576,577,
     578,579,580,581,582,583,584,585,586,587,
     589,590,591,592,593,594,595,596,597,598,
     600,601,602,603,604,605,606,607,608,609,
```

610,611,612,613,614,615,616,617,618,619,

620,621,622,623,624,625,625,627,628,629,

630,631,632,633,634,635,636,637,638,639,

640,641,642,643,644,645,646,647,648,649,

650,651,652,653,654,655,656,657,658,659,

670,671,672,673,674,675,676,677,678,679,

680,681,682,684,685,686,687,688,689,690,

691,692,693,694,695,696,697,698,699,700,

701,702,703,705,706,707,708,709,710,711,

712,713,715,716,717,718,719,720,721,722,

723,724,725,726,727,728,729,730,731,732,

733,734,735,736,737,738,739,740,741,742,

743,744,745,746,747,748,749,750,751,752,

753,754,755,756,757,758,759,760,761,762,

763,764,765,766,767,768,768,769,770,771,

1111,1112,1113,1114,1115,11134,11148,11150,11190,11198,

111100,111101,111102,111103,111105,111105,111106,111108,111109,111120,

111122,111125,111126, 111129,111130,111131,111135,111137,111139,111140,

111141,111145,111147,111148,111148,111149,111150,111151,111152,111153,

111154,111155,111156,111157,111158,111159,111160,111161,111162,111163,

111164,111165,111166,111167,111168,111169,111170,111172,111173,111175,

111179,111180,111182,111183,111185,111186,111188,111189,111190,111191,

111192,111193,111194,111195,111196,111197,11198,111199,111200,111201,

111102,111203,111204,111205,111206,111207,111208,111209,111210,111211,

111212,111213,111214,111215,111216,111217,111218,111219,111220,111221,

111222,111223,111224,111225,111226,111227,111228,111228,111230,111233,

111235,111236,111238,111239,111239,111300,111302,111303,111304,111305,

111306,111307,111308,111308,111309,111400,111401,111402,111403,111404,

111405,411105,111406,111407,111408,111409,111410,111411,111412,111413,

111414,111415,111416,111417,111418,111419,111500,111501,111502,111503,

```
    111504,111505,111506,111507,111508,111509,111510,111511,111512,111513,
    111514,111516,111517,111518,111519,111520,111521,111522,111523,111524,
    111525,111526,111527,111528,111529,111530,111531,111532,111533,111534,
    111535,111536,111537,111538,111539,111540,111542,111543,111544,111545,
    111546,111547,111548,111549,111550,111552,111553,111553,111554,111555,
    111556,111557,111558,111559,111560,111561,111562,111563,111564,111567,
    111568,111569,111570,111571,111572,111573,111574,111575,111576,111577,
    111578,111579,111580,111581,111582,111583,111584,111585,111586,111587,
    111589,111590,111591,111592,111593,111594,111595,111596,111597,111598,
    111600,111601,111602,111603,111604,111605,111606,111607,111608,111609,
    111610,111611,111612,111613,111614,111615,111616,111617,111618,111619,
    111620,111621,111622,111623,111624,111625,111625,111627,111628,111629,
    111630,111631,111632,111633,111634,111635,111636,111637,111638,111639,
    111640,111641,111642,111643,111644,111645,111646,111647,111648,111649,
    111650,111651,111652,111653,111654,111655,111656,111657,111658,111659,
    111670,111671,111672,111673,111674,111675,111676,111677,111678,111679,
    111680,111681,111682,111684,111685,111686,111687,111688,111689,111690,
    111691,111692,111693,111694,111695,111696,111697,111698,111699,111700,
    111701,111702,111703,111705,111706,111707,111708,111709,111710,111711,
    111712,111713,111715,111716,111717,111718,111719,111720,111721,111722,
    111723,111724,111725,111726,111727,111728,111729,111730,111731,111732,
    111733,111734,111735,111736,111737,111738,111739,111740,111741,111742,
    111743,111744,111745,111746,111747,111748,111749,111750,111751,111752,
    111753,111754,111755,111756,111757,111758,111759,111760,111761,111762,
    111763,111764,111765,111766,111767,111768,111768,111769,111770,111771};
System.out.println("Number of Element: "+numberOfElement);
System.out.println("Key 111771's position: "+mbs.binarySearch(arr6, 111771));
System.out.println( "Total Time: " + (System.nanoTime() - startTime) / 1000000+ " ms\n");
 }
```

```
 numberOfElement=1200;
runs =250;
 for (int i = 0; i < runs; i++) {
startTime = System.nanoTime();
int[] arr7 = {1,2,3,4,5,34,48,50,90,98,
    100,101,102,103,105,105,106,108,109,120,
    122,125,126, 129,130,131,135,137,139,140,
    141,145,147,148,148,149,150,151,152,153,
    154,155,156,157,158,159,160,161,162,163,
    164,165,166,167,168,169,170,172,173,175,
    179,180,182,183,185,186,188,189,190,191,
    192,193,194,195,196,197,198,199,200,201,
    102,203,204,205,206,207,208,209,210,211,
    212,213,214,215,216,217,218,219,220,221,
    222,223,224,225,226,227,228,228,230,233,
    235,236,238,239,239,300,302,303,304,305,
    306,307,308,308,309,400,401,402,403,404,
    405,405,406,407,408,409,410,411,412,413,
    414,415,416,417,418,419,500,501,502,503,
    504,505,506,507,508,509,510,511,512,513,
    514,516,517,518,519,520,521,522,523,524,
    525,526,527,528,529,530,531,532,533,534,
    535,536,537,538,539,540,542,543,544,545,
    546,547,548,549,550,552,553,553,554,555,
    556,557,558,559,560,561,562,563,564,567,
    568,569,570,571,572,573,574,575,576,577,
    578,579,580,581,582,583,584,585,586,587,
    589,590,591,592,593,594,595,596,597,598,
    600,601,602,603,604,605,606,607,608,609,
    610,611,612,613,614,615,616,617,618,619,
```

620,621,622,623,624,625,625,627,628,629,

630,631,632,633,634,635,636,637,638,639,

640,641,642,643,644,645,646,647,648,649,

650,651,652,653,654,655,656,657,658,659,

670,671,672,673,674,675,676,677,678,679,

680,681,682,684,685,686,687,688,689,690,

691,692,693,694,695,696,697,698,699,700,

701,702,703,705,706,707,708,709,710,711,

712,713,715,716,717,718,719,720,721,722,

723,724,725,726,727,728,729,730,731,732,

733,734,735,736,737,738,739,740,741,742,

743,744,745,746,747,748,749,750,751,752,

753,754,755,756,757,758,759,760,761,762,

763,764,765,766,767,768,768,769,770,771,

1111,1112,1113,1114,1115,11134,11148,11150,11190,11198,

111100,111101,111102,111103,111105,111105,111106,111108,111109,111120,

111122,111125,111126, 111129,111130,111131,111135,111137,111139,111140,

111141,111145,111147,111148,111148,111149,111150,111151,111152,111153,

111154,111155,111156,111157,111158,111159,111160,111161,111162,111163,

111164,111165,111166,111167,111168,111169,111170,111172,111173,111175,

111179,111180,111182,111183,111185,111186,111188,111189,111190,111191,

111192,111193,111194,111195,111196,111197,11198,111199,111200,111201,

111102,111203,111204,111205,111206,111207,111208,111209,111210,111211,

111212,111213,111214,111215,111216,111217,111218,111219,111220,111221,

111222,111223,111224,111225,111226,111227,111228,111228,111230,111233,

111235,111236,111238,111239,111239,111300,111302,111303,111304,111305,

111306,111307,111308,111308,111309,111400,111401,111402,111403,111404,

111405,411105,111406,111407,111408,111409,111410,111411,111412,111413,

111414,111415,111416,111417,111418,111419,111500,111501,111502,111503,

111504,111505,111506,111507,111508,111509,111510,111511,111512,111513,

111514,111516,111517,111518,111519,111520,111521,111522,111523,111524,

111525,111526,111527,111528,111529,111530,111531,111532,111533,111534,

111535,111536,111537,111538,111539,111540,111542,111543,111544,111545,

111546,111547,111548,111549,111550,111552,111553,111553,111554,111555,

111556,111557,111558,111559,111560,111561,111562,111563,111564,111567,

111568,111569,111570,111571,111572,111573,111574,111575,111576,111577,

111578,111579,111580,111581,111582,111583,111584,111585,111586,111587,

111589,111590,111591,111592,111593,111594,111595,111596,111597,111598,

111600,111601,111602,111603,111604,111605,111606,111607,111608,111609,

111610,111611,111612,111613,111614,111615,111616,111617,111618,111619,

111620,111621,111622,111623,111624,111625,111625,111627,111628,111629,

111630,111631,111632,111633,111634,111635,111636,111637,111638,111639,

111640,111641,111642,111643,111644,111645,111646,111647,111648,111649,

111650,111651,111652,111653,111654,111655,111656,111657,111658,111659,

111670,111671,111672,111673,111674,111675,111676,111677,111678,111679,

111680,111681,111682,111684,111685,111686,111687,111688,111689,111690,

111691,111692,111693,111694,111695,111696,111697,111698,111699,111700,

111701,111702,111703,111705,111706,111707,111708,111709,111710,111711,

111712,111713,111715,111716,111717,111718,111719,111720,111721,111722,

111723,111724,111725,111726,111727,111728,111729,111730,111731,111732,

111733,111734,111735,111736,111737,111738,111739,111740,111741,111742,

111743,111744,111745,111746,111747,111748,111749,111750,111751,111752,

111753,111754,111755,111756,111757,111758,111759,111760,111761,111762,

111763,111764,111765,111766,111767,111768,111768,111769,111770,11177,

221111,221112,221113,221114,221115,2211134,2211148,2211150,2211190,2211198,

22111100,22111101,22111102,22111103,22111105,22111105,22111106,22111108,22111109,22111120,

22111122,22111125,22111126, 22111129,22111130,22111131,22111135,22111137,22111139,22111140,

22111141,22111145,22111147,22111148,22111148,22111149,22111150,22111151,22111152,22111153,

22111154,22111155,22111156,22111157,22111158,22111159,22111160,22111161,22111162,22111163,

22111164,22111165,22111166,22111167,22111168,22111169,22111170,222111172,22111173,22111175,

22111179,22111180,22111182,22111183,22111185,22111186,22111188,22111189,22111190,22111191,
22111192,22111193,22111194,22111195,22111196,22111197,2211198,22111199,22111200,22111201,
22111102,22111203,22111204,22111205,22111206,22111207,22111208,22111209,22111210,22111211,
22111212,22111213,22111214,22111215,22111216,22111217,22111218,22111219,22111220,22111221,
22111222,22111223,22111224,22111225,22111226,22111227,22111228,22111228,22111230,22111233,
22111235,22111236,22111238,22111239,22111239,22111300,22111302,22111303,22111304,22111305,
22111306,22111307,22111308,22111308,22111309,22111400,22111401,22111402,22111403,22111404,
22111405,22411105,22111406,22111407,22111408,22111409,22111410,22111411,22111412,22111413,
22111414,22111415,22111416,22111417,22111418,22111419,22111500,22111501,22111502,22111503,
22111504,22111505,22111506,22111507,22111508,22111509,22111510,22111511,22111512,22111513,
22111514,22111516,22111517,22111518,22111519,22111520,22111521,22111522,22111523,22111524,
22111525,22111526,22111527,22111528,22111529,22111530,22111531,22111532,22111533,22111534,
22111535,22111536,22111537,22111538,22111539,22111540,22111542,22111543,22111544,222111545,
22111546,22111547,22111548,22111549,222111550,22111552,22111553,22111553,22111554,22111555,
22111556,22111557,22111558,22111559,22111560,22111561,22111562,22111563,22111564,22111567,
22111568,22111569,22111570,22111571,22111572,22111573,22111574,22111575,22111576,22111577,
22111578,22111579,22111580,22111581,22111582,22111583,22111584,22111585,22111586,22111587,
22111589,22111590,22111591,22111592,22111593,22111594,22111595,22111596,22111597,22111598,
22111600,22111601,22111602,22111603,22111604,22111605,22111606,22111607,22111608,22111609,
22111610,22111611,22111612,22111613,22111614,22111615,22111616,22111617,22111618,22111619,
22111620,22111621,22111622,22111623,22111624,22111625,22111625,22111627,22111628,22111629,
22111630,22111631,22111632,22111633,22111634,22111635,22111636,22111637,22111638,22111639,
22111640,22111641,22111642,22111643,22111644,22111645,22111646,22111647,22111648,22111649,
22111650,22111651,22111652,22111653,22111654,22111655,22111656,22111657,22111658,22111659,
22111670,22111671,22111672,22111673,22111674,22111675,22111676,22111677,22111678,22111679,
22111680,22111681,22111682,22111684,22111685,22111686,22111687,22111688,22111689,22111690,
22111691,22111692,22111693,22111694,22111695,22111696,22111697,22111698,22111699,22111700,
22111701,22111702,22111703,22111705,22111706,22111707,22111708,22111709,22111710,22111711,
22111712,22111713,22111715,22111716,22111717,22111718,22111719,22111720,22111721,22111722,

```
    22111723,22111724,22111725,22111726,22111727,22111728,22111729,22111730,22111731,22111732,
    22111733,22111734,22111735,22111736,22111737,22111738,22111739,22111740,22111741,22111742,
    22111743,22111744,22111745,22111746,22111747,22111748,22111749,22111750,22111751,22111752,
    22111753,22111754,22111755,22111756,22111757,22111758,22111759,22111760,22111761,22111762,
    22111763,22111764,22111765,22111766,22111767,22111768,22111768,22111769,22111770,22111771
};
System.out.println("Number of Element: "+numberOfElement);
System.out.println("Key 22111771's position: "+mbs.binarySearch(arr7, 22111771));
System.out.println( "Total Time: " + (System.nanoTime() - startTime) / 1000000+ " ms\n");
 }
 //note only some part of the codes appears here.
```

Table of contents