# Parallel Quick Sort using Thread Pool Pattern

Somshubra Majumdar
D. J. Sanghvi College of Engineering
Mumbai, India

Ishaan Jain
D. J. Sanghvi College of Engineering
Mumbai, India

Aruna Gawade
D. J. Sanghvi College of Engineering
Mumbai, India

## ABSTRACT
Sorting algorithms, their implementations and their applications in modern computing necessitates improvements for sorting large data sets quickly and efficiently. This paper will analyze the performance of a multi-threaded quick sort implemented using the thread pool pattern. The analysis will be done by comparing the time required to sort various data sets and their memory constraints, against the native sorting implementations of the Dual Pivot Quicksort and Merge Sort using the Fork-Join framework in the Oracle Java 8 programming language. Analysis is done of the effect of different number of processor (cores) of the test machine, as well as the performance barrier due to the initial time taken to create "p" threads, p being the number of processors. This paper also analyzes the limitations of the inbuilt Java method Arrays.parallelSort() and how the proposed system overcomes this problem. Finally, it also discuss possible improvements to the proposed system to further improve its performance.

## Keywords
Sorting, Multithreading, Object oriented programming, Parallel algorithms

## 1. INTRODUCTION
Sorting is defined as the operation of arranging an unordered collection of elements into monotonically increasing (or decreasing) order. Specifically, S = {a1, a2 ……an} be a sequence of n elements in random order; sorting transforms S into monotonically increasing sequence S'= {a1 ', a2 '…… an '} such that ai '≤ aj' for 1≤ i ≤ j ≤ n, and S' is a permutation of S [1].

Sorting large data sets is a requirement for several modern applications. There are several different sorting algorithms which have been analyzed in great detail. Sorted data sets possess several important properties, enabling efficient searching and statistical analysis. Binary search is one of the fastest searching algorithms which operates only on sorted data sets.

Seeing how sorting in useful in many other domains, it is important for sorting algorithms to be fast and efficient. General purpose sorting algorithms must be designed, which offer efficient use of available processors, main memory as well as reduce the overall time required to sort various data sets.

In computer science, a thread is a small sequence of instructions that can be performed independently by a processor. Multi-threading is possible within a process, wherein they share resources such as instructions and context. [2, 3].

Multithreaded applications provide advantages such as [2, 4]:

- If the main thread of a single-threaded application blocks the flow of execution, the entire application can appear to freeze. Multi-threading prevents the operating system from freezing, as seen in the case of single threaded processes. By transferring a long task to a *worker thread* that runs parallel to the main thread, the application remains responsive to user input while executing background tasks.
- Multi-threaded programs work best on a multi-processor system, since they distribute executable tasks onto processors that execute each task in parallel.
- Multi-threading allows multiple clients to access an application concurrently.
- Multi-threaded applications can also utilize the system better. Multiple processors can execute in parallel and improve the throughput.

In version 7 of the Oracle Java Development Kit (JDK), "Dual-Pivot Quicksort" by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch has been implemented as the Arrays.sort() method [5]. Since this algorithm offers $O(n\,log(n))$ performance on many data sets, it is usually faster than a 1-pivot Quicksort implementation [5]. While this sorting method is fast and efficient for relatively small data sets (n < 1 million), its performance degrades at large data sets (n >10 million). One can make use of threads and Thread Pools introduced in version 5 of the JDK to perform a multithreaded variant of Quicksort to significantly reduce sort times.

Oracle has also implemented in version 8 of the JDK, a multithreaded Merge Sort algorithm which can be invoked by the method Arrays.parallelSort(). This algorithm is a parallel sort-merge that divides the array into sorted subsets that are subsequently merged. When the subset length reaches a lower bound threshold, the subset is sorted using an appropriate serial sorting algorithm. The algorithm requires a working space as large as the size of the original array. The Fork-Join thread pool is used to execute parallel tasks [5].

Quicksort is an unstable sorting algorithm, meaning it does not preserve the relative order of equal items. It is also an in place sorting algorithm requiring only small amounts of memory to sort data sets [6]. It has an average sort complexity of $O(n\,log(n))$, but degrades to $O(n^2)$ under rare circumstances. Generally, it out performs most of the other $O(n\,log(n))$ algorithms [7].

Quicksort is a divide and conquer algorithm. Initially, it splits the array into two sub-arrays: the lower items and the higher items respectively. It then recursively sorts these sub-arrays. Its algorithm can be described as [6]:

1. Pick a pivot element from the array.
2. Partition the array such that, all the elements to the left of the pivot have values less than the value of

the pivot, and all elements to the right of the pivot have values greater than the pivot value.

3. Recursively perform the above steps over the generated sub-arrays, until completely sorted sub-arrays are obtained.

After performing the above steps, the array has been partitioned into three sub-arrays:

1. Values less than the pivot
2. The pivot value
3. Values greater than the pivot

However, post-partition the sorting of the new sub-sequences can be performed in parallel as there is no collision. Thus, the algorithm is trivial to parallelize [8].

The choice of pivot elements affects the performance of the algorithm since Quicksort is data dependent. Sorting algorithms are said to be data dependent if their performance depends on the type of data sets provided to them. Some standard types are nearly sorted data set, completely sorted data set and reverse sorted data set. Usually selecting the middle value of the list/sub-list as the pivot value prevents most degradations in performance. Calculation of the middle value index is usually done using $(low + high)/2$ [4]. However, this may cause integer overflow, and yield a negative index. To avoid this, the middle value index can be obtained using $(low + ((high - low) / 2))$, or its equivalent $((low + high) >>> 1)$ [4]. Also, to reduce the recursive component, one may sort the data directly using some other algorithm such as Insertion Sort for very small subsets of the data set. This can greatly reduce the space requirement of stack memory as well as reduce computation time.

The structure of the paper is as follows: In Section 3 analysis of Java's inbuilt functions, namely Arrays.sort() and Arrays.parallelSort() is performed. Section 4 describes the theory behind the proposed algorithm as well as the system itself. Section 5 compare the performance of the proposed system with respect to the Java functions mentioned in Section 3. The paper concludes with Section 6.

## 2. LITERATURE SURVEY

The concept of multi-threaded Quicksort has been well studied and analyzed over several decades. Ranging from Oracle's official documentation on Java SE 8 to papers by other researchers, there have been multiple improvements in the efficiency of the implementations of the Quicksort algorithm as well as its parallel variants.

Originally Quicksort (developed by C.A.R. Hoare) was implemented using recursion to divide the given array of unsorted values into smaller, more easily sortable sub-lists [1]. These sub-lists are either further divided, or deemed to be sorted and requiring no further splitting. A parallel implementation of Quicksort was a good choice for two reasons. Firstly, it is one of the fastest sorting algorithms for an average case. Secondly, Quicksort has a natural concurrency, that is, when it calls itself recursively, the two recursive calls can be executed simultaneously since the subsets are disjoint.

Studies have been conducted on parallel quick sort implementation in the SUN Enterprise 10000 systems [9]. It is a cache efficient implementation using overlapping fine grain parallelism to improve efficiency. Their research indicates that their implementation is 50% faster than parallel Sample Sort implementation. Generally it is observed that sample sort outperforms Quicksort in most data sets, however the parallel implementations of Quicksort can be made to sort even very large data sets quickly.

Modern computers have multiple processors which act as a single logical unit. Utilizing multiple processors to asynchronously execute independent tasks relevant to a single program is referred to as multithreading, which is often managed by the overlying operating system. However there is an overhead involved in creating, managing, executing and destroying threads by the operating system. Thus, the thread pool pattern was introduced [10]. A thread pool creates a cache of running threads on startup which wait to execute tasks supplied to them via a work queue. As tasks are submitted, each task is allocated to a thread in the pool ready to accept and execute a task. If there are too many tasks, and no available threads to execute them, then the tasks remain in the work queue until they are assigned their thread for execution.

Sorting has multiple applications [11] such as removing duplicate values, median and order statistics calculations. It is used in algorithms such as Prim's and Kruskals to calculate the shortest sequence between two points. It is also used in Djikstras algorithm to compute the shortest distance between two points in a graph. Sorting is also used before performing Binary Search in order to rapidly find the required item. Huffman Compression algorithm also utilizes sorting algorithms to quickly find the two smallest weighted items and to produce the concatenated value. It is also used in String processing algorithms.

## 3. ANALYSIS OF JAVA'S INBUILT FUNCTIONS

To compare the performance between Oracle Java sorting methods [5] and the proposed system, random data sets of varying size were selected for analysis. The data sets are generated at run time using the inbuilt Random class, thus generating pseudo-random data values. Due to main memory limitations, data set size has an upper bound of 800 million integer type data items. Size of each integer type data item is 4 bytes on Windows operating systems.

As a baseline, identical data sets were sorted using the inbuilt Java methods (Arrays.sort() and Arrays.parallelSort()) to compare the time required to sort each data set. To improve accuracy of comparison between the methods, multiple data sets were generated for each size quantum. As the size of the data sets increased exponentially, the time required to sort the sets increased proportionally. Due to this, for larger data set sizes, fewer numbers of data sets were compared. Due to the Java 8 Arrays.parallelSort() algorithm utilizing Merge Sort as its base algorithm, it is limited to 100 million data items on the proposed system. This is because Merge Sort is an external sort, requiring an auxiliary working array of same size 'n' as the data set given as input.

The following figures describe the time in milliseconds required to sort 'n' data items by either Arrays.sort() or Arrays.parallelSort(). The blue line represents the time required by Arrays.sort(), while the yellow line represents the time required by Arrays.parallelSort(). The computation time was calculated on an Intel i5 processor 4 core machine (2 physical, 4 logical) and 8 GB of RAM.

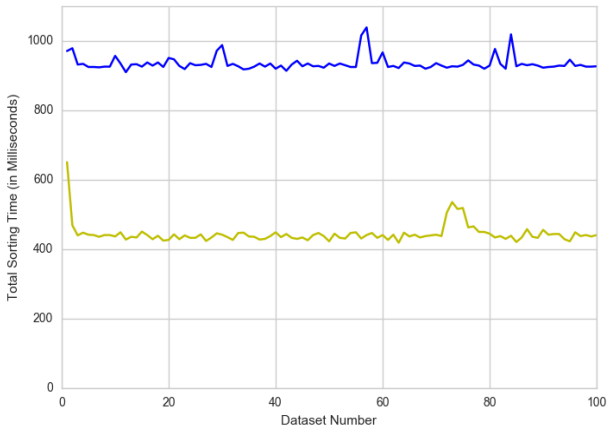


**Figure 3.1 Time (in milliseconds) required to sort 100 data sets each having 10 million items**

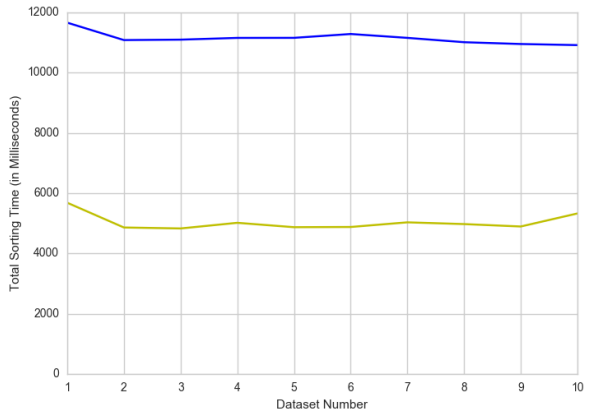


**Figure 3.2 Time (in milliseconds) required to sort 10 data sets each having 100 million items**

As the figures suggest, the sorting time for the single threaded Dual Pivot Quicksort used in Arrays.sort() requires at least twice the time to sort the same data set as the multi-threaded Arrays.parallelSort(). On a machine with 4 processors, the parallel execution of Arrays.parallelSort() provides nearly 100% gain in performance.

## 4. PROPOSED SYSTEM: MULTI-THREADED QUICK SORT WITH THREAD POOL

In computer programming, a thread pool pattern is a collection of threads in a synchronized blocking queue designed to perform multiple tasks in parallel. The output of the finished tasks can be queued, or they may return no output [10]. In most cases, the number of tasks outnumber that of threads. On completion of a task, the thread requisitions the next queued task until all tasks are completed. The thread can then end, or wait until the next task arrives [12].

The degree of parallelism can be altered to provide optimum performance. Additionally, the degree of parallelism can be dynamically determined based on the number of tasks awaiting execution. The algorithm used to administrate the creation and destruction of threads impacts overall performance [8, 9]:

- Creation of a large number of threads wastes computational resources
- Destruction of a large number of threads causes the system to waste time recreating them
- Sudden increase in the number of required threads results in poor client performance
- Sudden destruction of threads may starve other processes of resources

This method is a combination of the aforementioned 3 techniques of Quicksort, multithreading and thread pool pattern to make the sorting process faster for larger data sets compared to the Java implementation of Arrays.sort() and Arrays.parallelSort() while also caching the threads to avoid recreation of additional threads.

The algorithm is backed by the task queue held by the Thread Pool, which caches "p" threads for as long as the sorting task is not completed, "p" being the maximum number of available processors. A task can be defined as a set of executable statements that are passed to the Thread Pool to be executed [10]. The task queue must be a synchronized blocking queue. Also, "n" is considered to be the size of the data set being sorted.

The algorithm can be simply described as:

```
Algorithm ParallelQuicksort(data_set, n, p)
{
threshold : = (p > 1)? (1 + n / (p << 3)): n
submitToThreadPool (PQuicksort (data_set, 0, n - 1))
wait for all threads to complete execution
}

Algorithm PQuicksort (data_set, low, high)
{
if ((high - low) < threshold)
        sortDirectly (data_set, low, high)
else
{
i := low, j := high
pivot:= data_set[(low + (high-low)/2)

while i <= j {
        while data_set[i] < pivot {
                increment i
        }

        while data_set[j] > pivot {
                decrement j
        }

        if i <= j {
                swap data_set[i] with data_set[j]
                increment i
                decrement j
        }
}
```

```
if low < j
        submitToThreadPool(PQuicksort(data_set, low, j))
if i < high
        submitToThreadPool (PQuicksort(data_set, i, high)
}
}
```

In the above algorithm, to limit the number of threads utilized by the executor, first check if the dataset size if smaller than the threshold defined. If so, then it is directly sorted using another sorting algorithm, such as Insertion sort (if n < 35) or with a non-parallel Quick Sort (n > 1000). Otherwise, partition the data set into 2 sets as described above, as well as swap values if necessary.

Recursion is not permitted as long as the data set is larger than the threshold, that is, other than in sortDirectly(). Instead, the algorithm submits the two possible recursive components of PQuicksort using different parameters for the low and high values, to the task queue in the Thread Pool. Then, the new tasks are atomically removed from the task queue and subsequently executed by the assigned thread. However, if no thread is ready to accept new tasks, then the task must wait in the task queue

This implies that the cost of recursion is now replaced with the initial cost of creating "p" threads for the Thread Pool. Since threads are cached, thus all "p" threads are active throughout the execution of the sorting algorithm. The cost of recursion is dependent on the stack, the number of recursive calls and the time required to return to the calling function, which reduces performance. Instead, one may parallelize the recursive components and replace the recurring cost of recursion with the one time cost of creating the "p" threads.

## 5. ANALYSIS OF PERFORMANCE OF PROPOSED SYSTEM

While the performance of the Arrays.parallelSort() is far better than the single threaded Arrays.sort(), it outperforms the latter algorithm only in large data sets (n > 1 million). Another major drawback of Arrays.parallelSort() is the fact that, due to its base algorithm being Merge Sort, it requires a working array of size "n" to execute the sorting algorithm. The proposed method for using Thread Pool pattern to implement Quicksort seeks to improve upon the existing Arrays.parallelSort() function in terms of speed and memory requirement.

While the proposed method may improve the aforementioned performance factors, one must keep in mind that as a multithreaded sorting mechanism, the proposed method is highly dependent on the maximum number of processors as well as the maximum available physical memory (RAM) available. The performance of this system has been analyzed using a machine with a maximum of 8GB of RAM and an Intel i3 processor (processor clock frequency is 2.8 GHz) with 2 physical (4 logical) cores and a second machine having a maximum of 8 GB of RAM and an Intel i5 processor (processor clock frequency is 2.9 GHz) with 2 physical (4 logical) cores. Performance of the proposed method will vary on systems with varying measure of available RAM or processor clock speeds or number of available processors. Hence the proposed method has been tested on 2 different machines to keep a broader view of the comparison between the proposed and the existing functions.

Note that Java Virtual Machine (JVM) is allocate at most 2048 megabytes for the application heap, using -Xmx2048m as the command line JVM argument. Using these constraints, one can create and sort a data set of maximum size close to 350 million integer data items. However, upon testing Arrays.parallelSort() using n > 160 million data items, it is found that the program quits execution due to an OutOfMemoryError. This is due to the $O(n)$ memory space required as a working array for Merge Sort. Thus, test the proposed algorithm and Arrays.parallelSort() at n <= 100 million. Having seen no such error due to the proposed algorithm or Arrays.sort(), even at the system limit of 350 million data items, one may consider a few tests at n = 350 million (approximately 1335 Megabytes of RAM).

The following figures are the results of comparing the proposed system with the Java sorting functions Arrays.sort() and Arrays.parallelSort(). The results are shown as percentage gain or loss when the execution time (in milliseconds) of the proposed system is compared to execution time of Arrays.sort() or Arrays.parallelSort(). Note that the blue lines represent the percentage gain/loss when comparing the proposed system with Arrays.sort(), while the yellow lines represent the same when comparing the proposed system with Arrays.parallelSort().

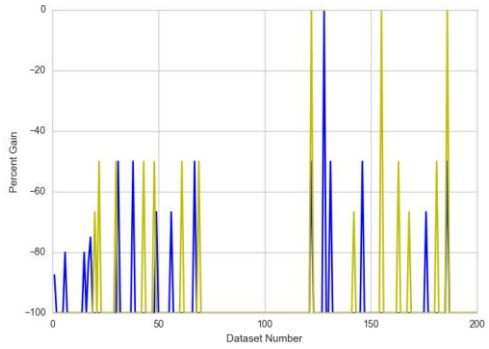


**Figure 5.1 Percentage loss on 200 data sets of 1000 items each on first 4-core machine**

Note that the time required to sort small data sets (n < 1000) by Arrays.sort() or Arrays.parallelSort() is calculated as 0 milliseconds (actual execution time is approximately 200-250 microseconds). Proposed system however requires some time in creating the thread pool itself, thus the execution time is greater than 1 millisecond. Due to this, the comparison figures at n = 1000 produces a large loss percentage with spikes due to thread pool creation time as shown in above figure.

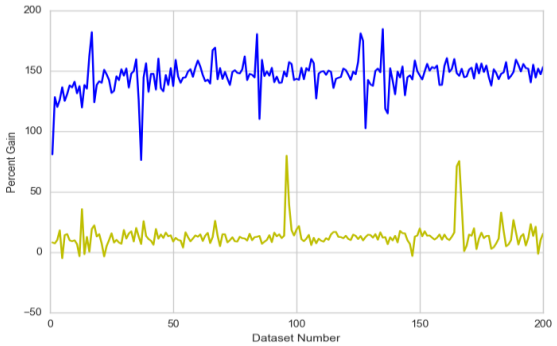


**Figure 5.2 Percentage gain on 200 data sets of 10 million items each on second 4-core machine**

As seen in the above image, the blue line indicates an average performance gain of 150% over Arrays.sort(), while the yellow line indicates a more modest performance gain of 20% over Arrays.parallelSort().

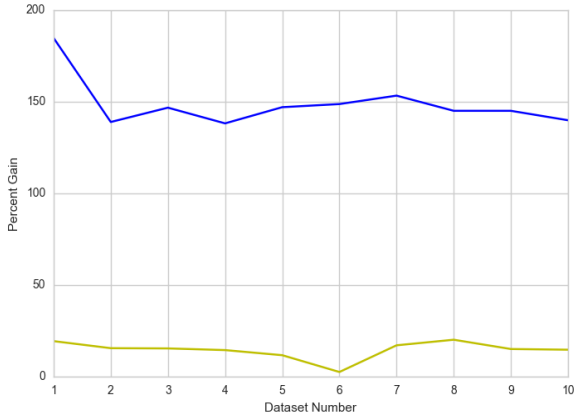


**Figure 5.3 Percentage gain on 10 data sets of 100 million items each on second 4-core machine**

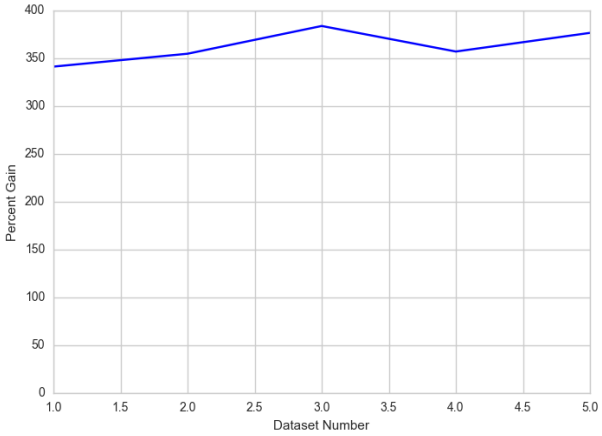


**Figure 5.4 Percentage gain on 5 data sets of 350 million items each on an 8-core machine**

As seen, the figures comparing the proposed system and Arrays.parallelSort() for data sets of very large size (greater than 100 million items) have not been shown, as explained above. Above are a few figures comparing the proposed system to Arrays.sort() at data set sizes of over 100 million data items, since only Arrays.sort() does not crash during execution.

Note that the last two data set (n = 200 and 350 million data items) is sorted on a different machine, having an i7 processor with 8 cores (4 physical, 8 logical) having 8 GB of RAM. This was done to indicate that the performance of the proposed system will improve with the increase in number of available processors, as seen by the nearly 350% gain in performance compared to the 125-150% gain as seen on the other machines.

**Table 5.1 Tabularization of results**

| **Array size** | **Arrays. sort() (ms)** | **Arrays.parallelSort() (ms)** | **Proposed System (ms)** |
|---|---|---|---|
| 1,000 | 0 | 0 | 2 |
| 1 Million | 90 | 43 | 40 |
| 5 Million | 446 | 206 | 172 |
| 10 Million | 947 | 437 | 386 |
| 20 Million | 1954 | 845 | 771 |
| 50 Million | 5063 | 2328 | 2013 |
| 100 Million | 11379 | 4844 | 4523 |
| 200 Million | 21961 | - | 8614 |
| 350 Million | 40941 | - | 9034 |

The above table summarizes the results empirically obtained in this section. The values that are shown are the average execution times for sorting data sets using the respective methods.

For the dataset with a size of 1000 items, it is seen that Arrays.sort() and Arrays.parallelSort() have an execution time of 0 milliseconds, while internally the time taken is on average is less than 1 millisecond, and can be considered negligible.

For the datasets with a size of greater than 160 million data elements, Arrays.parallelSort() fails due to an OutOfMemoryError, giving no output.

## 6. CONCLUSION

Quicksort is widely considered to be one of the fastest, general purpose sorting algorithm. Due to its divide and conquer strategy, it can be implemented to execute on multiple processors to speed up sorting. The proposed algorithm implements a multithreaded Quicksort using a thread pool and have analyzed its performance in comparison with native Oracle Java implementations of the single threaded Dual Pivot Quicksort used in Arrays.sort() and the multithreaded Merge Sort using the Fork-Join Pool in Arrays.parallelSort(). Computational experiments have shown us that our algorithm outperforms Arrays.sort() when the data set is not trivial (over 1 million data items) and, on average, it outperforms Arrays.parallelSort().

## 7. FUTURE WORK

Our system provides reasonable performance benefits over the inbuilt functions of Arrays.sort() and Arrays.parallelSort(), however there are still improvements which can be made. Quicksort is inherently an unstable sort, and does not maintain the relative order of data items. It is possible to utilize the same method to parallelize other sorting algorithms which include Merge Sort, Sample Sort and Heap Sort. Also, currently the calculation of the partition pointers (i and j) is done sequentially. However, it is possible to also parallelize the calculation of these pointers, thus one can hope to achieve a minor gain in the execution speed of the algorithm to improve the performance of this system.

## 8. ACKNOWLEDGEMENTS

The authors humbly acknowledge the invaluable contributions of Professor Aruna Gawade for her help and guidance in developing this document.

## 9. REFERENCES

[1] "*Parallel Programming in C with MPI and OpenMP*". M. J. Quinn, Tata McGraw Hill Publications, 2003, p. 338

[2] "*Java Thread Programming*", Paul Hyde, ISBN 0-672-31585-8.

[3] "*Multi-Threaded Programming in C++*", Mark Walmsley, Springer, ISBN 1-85233-146-1.

[4] "*Algorithms In C: Fundamentals, Data Structures, Sorting, Searching, Parts 1-4 (3 ed.)*".Sedgewick, Robet (1 September 1998).

[5] *"Arrays (Java Platform SE 8)*."Arrays (Java Platform SE 8). Web. 28 Mar. 2015.

[6] <http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html>. "*The Art of Computer Programming*, Volume 3: *Sorting and Searching".* Donald Knuth. Third Edition. Addison-Wesley, 1997.

[7] "*Popular sorting algorithms*", C. Canaan, M. S. Garai, M. Daya <http://waprogramming.com/papers/50ae468b07bca6.46839978.pdf>.

[8] "*Introduction to Algorithms*". Cambridg. Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

[9] *"A Simple, Fast Parallel Implementation of Quicksort and Its Performance Evaluation on SUN Enterprise 10000."* Tsigas, P., and Yi Zhang. Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2003. Proceedings. (2003)

[10] "*Java Theory and Practice: Thread Pools and Work Queues."* Web. 26 Mar. 2015 <http://www.ibm.com/developerworks/library/j-jtp0730/>.

[11] "2.5 *Sorting Applications*." *Sorting Applications*. Web. 26 Mar. 2015. <http://algs4.cs.princeton.edu/25applications/>.

[12] "*Techniques for Optimizing Applications - High Performance Computing"*, Garg, Rajat P. & Sharapov, Ilya. Prentice-Hall 2002,