# Project report (DS 8001)

# "Performance evaluation of sequential and parallel approach to optimize quick sort"

**Submitted to:**

**Dr. Andriy Miranskyy**

**Submitted by:**

**Shailendra Khadka Yadav**

**December 28, 2016**

## Introduction

Sorting is the arrangement of object of interest in either ascending order or descending order (alphabetically or numerically) [8]. It is one of the most fundamental computational task that is required in various areas of computer science such as databases and file indexing [8]. Sorting algorithms can be used in two ways: internal sorting and external sorting [2]. In internal sorting, data are sorted from memory while in external sorting, data are sorted from auxiliary devices [2]. There are many sorting algorithms such as bubble sort, insertion sort, selection sort, quick sort, merge sort, heap sort and a lot more [8]. Among these various sorting techniques, quick sort is one of the most widely used [1]. This is basically because it's time complexity is, quicker among most sorting algorithms, O(n log n) [1]. Although the worst-case time complexity for quick sort is $O(n^2)$, the situation normally does not occur in practical scenarios [2].

Parallelization is one of the most hyped technology of this modern era. Parallelization means running a task into multiple processors by sub-dividing the task and assigning each sub-task to a different processor in order to perform the whole task in less time [6]. Though speedup can be gained by parallelizing tasks, there are some concerns related to the idea of parallelization such as finding independent sub-tasks, load balancing, etc. which must be addressed [6].

Today's computers contain multi-core processors, which can significantly increase computational speed if computational tasks could be properly parallelized [1]. In this project, I am implementing Hoare's version of quick sort technique to try to find out the efficiency gained by parallelizing it [7]. I believe the parallel approach to quick sort technique would significantly decrease sorting time. This approach would be beneficial to applications that require fast sorting.

## Goal

There are two major goals of this project:

1. To find out an easy to implement yet efficient approach to parallelize quick sort.
2. To perform an empirical analysis on the performance of sequential and parallel approach to quick sort in terms of CPU time.

## Related Literature

There are several works that have been done on sequential and parallel quick sort and ways to optimize them [1, 2]. The sequential approach works on a divide-and-conquer strategy [8]. It does so by choosing a pivot element in an array first, then finding its pivot position and dividing the array into two sub-arrays recursively such that values less than pivot are in one sub-array and values greater than pivot are in next sub-array [8]. Since quick sort is an in-place sort, the entire array is sorted after the recursion terminates [8].

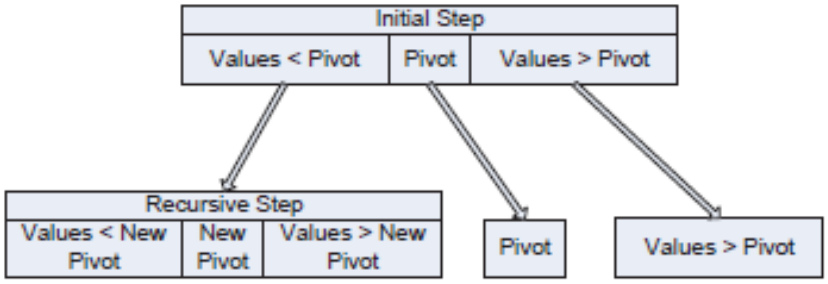


**Figure 1:** Simple graphical representation of the quick sort algorithm [2]

There are various approaches to implementing parallel quick sort [1, 2]. One of the approaches is to work the same way as in sequential sort, up to finding a pivot position for the first element of the array. But, then instead of subdividing the array into two halves recursively as in sequential sort, two threads are created once the pivot position is found [2]. Between these two threads, one thread will contain elements that have values less than pivot and another thread will contain elements that have values greater than pivot [2]. This approach works for small data sets, however when the data size is bigger, this approach becomes impracticable. This is because there is a limit on the number of threads that a process can have [3]. Thus, the approach that has been taken in this project is to limit the number of concurrently executing threads so that they may be implemented on any multi-core machine [4].

## Method

- **Software environment:** Java programming language is used for implementation of this project as it is easy to create threads in Java by using either the built-in Thread class or the Runnable interface. The version of Java used is 1.8.0. NetBeans IDE 8.0.2. is used for creating classes that implement sorting task as it supports both coding and testing and is an easy to use IDE.
- **Data size:** In this project, data sizes for performance evaluation are taken starting from one million to five million in steps of one million. The data sizes have been taken keeping in mind the available heap space tradeoff of Java [5]. The data sets are generated in an array using a loop starting from an integer one to required range such as one million, which gives linear ordering of data items. They are then shuffled using Collections feature available in Java to provide us random data sets.
- **Technique used for sequential approach:** For the sequential quick sort implementation, a random data set is provided to the function performing sorting task. The function takes the first element of the array as pivot element and finds out its pivot position by the use of two pointers down and up (given as p and r in [8]). The up and down start with last and first position of the array which are moved according as given in [8]. After the pivot position is discovered, the pivot element is in its proper position in the array [8]. The function then splits the original array into two sub-arrays in such a way that elements smaller than the pivot are in one sub-array while elements larger than the pivot are in another sub-array. This whole process is recursively executed until the recursion terminates.

- **Technique used for parallel approach:** For the parallel quick sort implementation, the same random data set is provided to the function performing sorting task. The function computes the pivot position as in sequential sorting task, however after finding the pivot position it creates two threads: one having elements smaller than the pivot and another having elements larger than the pivot. To avoid the sorting function from creating multiple threads such that they consume all the available resources, we restrict the function to creating only a limited number of threads. However, no guarantee can be given of equal load balancing to all cores as data sizes of various threads may be different and thus loads can be different.
- **Time capture:** The time taken by both versions of quick sort has been taken by using the nanoTime() method of System class which returns current time in nanoseconds as a long integer. The difference between the time when the quick sort function starts and ends its execution is the actual time consumed in the sorting task and this difference has been taken and formulated as a table for results analysis.
- **Hardware environment:** The results of the algorithms used for sequential and parallel approaches will be run on my Apple MacBook Pro laptop machine having 16GB RAM, quad-core i5 processor having CPU speed of 2.3 GHz and macOS Sierra version 10.12.1 operating system.

## Results

The following table and graph summarize the outcome of the experiments carried out during this study. The average of ten runs on varying data sets using both sequential and parallel approach is provided in the table. The corresponding graph built from the table shows the time variation found on different data sets using both versions of quick sort.

| Data set in millions | Average time taken in nanoseconds of ten runs using sequential approach | Average time taken in nanoseconds of ten runs using parallel approach |
|---|---|---|
| 1 | 251,931,818 | 15,587,380 |
| 2 | 355,496,994 | 18,483,522 |
| 3 | 495,968,030 | 24,013,624 |
| 4 | 715,652,044 | 32,080,419 |
| 5 | 886,984,078 | 35,319,101 |

**Table 1:** Result of time capture in nanoseconds of ten runs using sequential and parallel approach on various data sets in millions

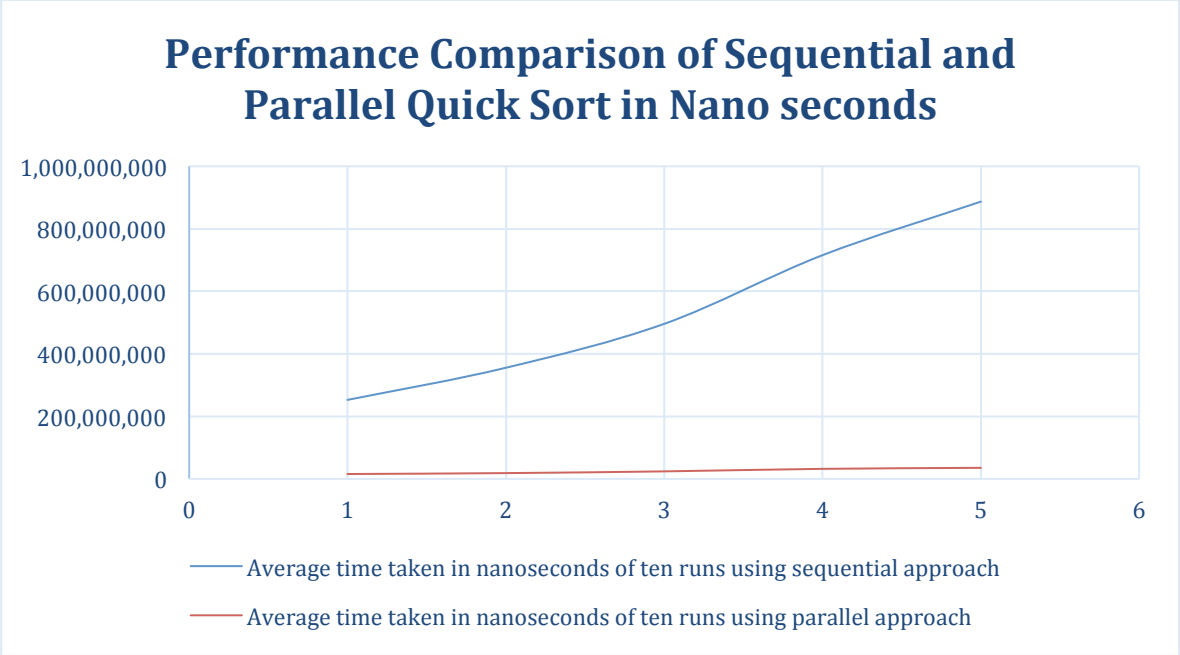


**Fig 2:** Graph of performance comparison between sequential and parallel quick sort

From the graph, it can be seen that there is a rapid increase in sorting time in case of sequential quick sort when the data size is increased from one million to five million. It comes out that sequential quick sort increases its sorting time by:
((886,984,078 - 715,652,044) + (715,652,044 - 495,968,030) + (495,968,030 - 355,496,994) + (355,496,994-251,931,818)) / 4 = 158,763,065 nanoseconds when the data size is stepped by one million.

In case of parallel quick sort, there is a comparative very slow increase in sorting time when the same data size is taken. It comes out that parallel quick sort increases its sorting time by:
((35,319,101 - 32,080,419) + (32,080,419 - 24,013,624) + (24,013,624 - 18,483,522) + (18,483,522 - 15,587,380)) / 4 = 4,932,930 nanoseconds when the data size is stepped by one million.

From the results, we could imply that parallel quick sort will perform way better than sequential quick sort when the data size will be massive. This is because the step increase in time with step increase in data size by one million is comparatively quite less in parallel quick sort.

It can also be inferred from the table and the graph that the efficiency of parallel quicksort increases with increase in data size. For example, when the data size is one million, the speedup is 251,931,818 / 15,587,380 ≈ 16 times, when the data size is two million, the speedup is 355,496,994 / 18,483,522 ≈ 19 times, when the data size is three million, the speedup is 495,968,030 / 24,013,624 ≈ 21 times, when the data size is four million, the speedup is 715,652,044 / 32,080,419 ≈ 22 times and when the data size is five million, the speedup is 886,984,078 / 35,319,101 ≈ 25 times. Thus, the parallel version of quick sort is best suited if scalability is a concern.

## Summary

The results show that significant speedup can be obtained by parallelizing quick sort because it utilizes the power of multi-core processors. However, uniform load balancing to each core could not be provided because there is no guarantee of where the pivot position will fall. There are various other approaches to implement parallel quick sort as well such as those given on [1, 2], however I have not tested those as I was performing a comparative analysis on sequential versus parallel version.

The performance results actually fluctuated up to 20-50% while time capturing was being done. This may be due to processes running in the background. There may be other hidden reasons as well. However, it was never the case that parallel quick sort had not won over sequential quick sort and that too with a large factor. I believe this study could be beneficial to others to study the performance of various other parallel versions of quick sort [2]. This study can also be extended to study the behavior, i.e. the time analysis, of the parallel version of quick sort as the number of concurrently executing threads increase.

## References


1. A.H. Almutairi and A.H. Alruwaili. "Improving of Quicksort Algorithm Performance by Sequential Thread or Parallel Algorithms." Global Journal of Computer Science and Technology, Vol. 12, Issue 10 Version 1.0, 2012.
2. H I.S. Rajput, B. Kumar and T. Singh. "Performance Comparison of Sequential Quick Sort and Parallel Quick Sort Algorithms." International Journal of Computer Applications, Vol. 57, No. 9, November 2012.
3. http://stackoverflow.com/questions/763579/how-many-threads-can-a-java-vm-support
4. http://www.deftitlenil.com/2011/04/blog-post_05.html
5. https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/memleaks002.html
6. https://en.wikipedia.org/wiki/Parallel_computing
7. https://en.wikipedia.org/wiki/Quicksort
8. T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein. "Introduction to Algorithms." The MIT Press, 1990, Second Edition.

## APPENDIX

**Java code to perform sequential quick sort with one million random data**

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class SequentialQuickSort {
   static void quickSort(int list[], int lb, int ub) {
          int pivot, down, up;
          int temp;
          if(lb>=ub)
                return;
      pivot=list[lb];
      down=lb;
      up=ub;
          while(down<up) {
                while(list[down]<=pivot  &&  down<ub)
                      down++;                    //move up the array
                while(list[up]>pivot)
                      up--;                      //move down the array
                if(down<up) {                    //interchange
                      temp=list[down];
                      list[down]=list[up];
                      list[up]=temp;
                      }
                }
          list[lb]=list[up];
          list[up]=pivot;

          quickSort(list, lb, up-1);
          quickSort(list, up+1, ub);
   }

//For creating random dataset
   static int [] createArray(int n) {
      // Create an ordered list
      List<Integer> list = new ArrayList<>();
      for (int i = 1; i <= n; i++) {
```

```
            list.add(i);
        }
        // Shuffle it
        Collections.shuffle(list);

        // Get an int[] array
        int[] array = new int[list.size()];
        for (int i = 0; i < list.size(); i++) {
            array[i] = list.get(i);
        }
        return array;
    }

    public static void main(String[] args) {
        int a[] = createArray(1000000);
        long start = System.nanoTime();
        quickSort(a,0,a.length-1);
        long end = System.nanoTime();
        for (int i = 0; i < a.length; i++) {
         System.out.print(a[i]+" ");
        }
        System.out.println("\nTotal time taken in ns:"+(end-start));
        try {
            Thread.sleep(10000);
        } catch (InterruptedException ex) {

        }
    }
}
```

**Java code to perform parallel quick sort with one million random data**

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class ParallelQuickSort extends Thread {
   int a[];
   int lb;
   int ub;

   static int NO_OF_THREADS = Runtime.getRuntime().availableProcessors();
   static int count;

   public ParallelQuickSort(int a[], int lb, int ub) {
   this.a=a;
   this.lb=lb;
   this.ub=ub;
   }

  //For creating random dataset
   static int [] createArray(int n) {
      // Create an ordered list
      List<Integer> list = new ArrayList<>();
      for (int i = 1; i <= n; i++) {
         list.add(i);
      }
      // Shuffle it
      Collections.shuffle(list);

      // Get an int[] array
      int[] array = new int[list.size()];
      for (int i = 0; i < list.size(); i++) {
         array[i] = list.get(i);
      }
      return array;
   }
```

```
static void quickSort(int list[], int lb, int ub)
{
        int pivot, down, up;
        int temp;
        if(lb>=ub)
                return;
   pivot=list[lb];
   down=lb;
   up=ub;

        while(down<up)
                {
                while(list[down]<=pivot  &&  down<ub)
                        down++;                 //move up the array
                while(list[up]>pivot)
                        up--;                   //move down the array
                if(down<up)                     //interchange
                        {
                        temp=list[down];
                        list[down]=list[up];
                        list[up]=temp;
                        }
                }
        list[lb]=list[up];
        list[up]=pivot;

   if(count <= NO_OF_THREADS){
        new Thread (new ParallelQuickSort(list, lb, up-1)).start();
        new Thread(new ParallelQuickSort(list, up+1, ub)).start();
   count += 2;
   }
   else
   {
      quickSort(list, lb, up-1);
      quickSort(list, up+1, ub);
   }
}
```

```
    public static void main(String[] args) {
        int x[] = createArray(1000000);
        long start = System.nanoTime();
        quickSort(x, 0, x.length-1);
        long end = System.nanoTime();

        try {
            Thread.sleep(10000);
        } catch (InterruptedException ex) {
        }
        for (int i = 0; i < x.length; i++) {
         System.out.print(x[i]+" ");
        }

        try {
            Thread.sleep(10000);
        } catch (InterruptedException ex) {
        }
        System.out.println("\nTotal time taken in ns:"+(end-start));
        try {
            Thread.sleep(10000);
        } catch (InterruptedException ex) {
        }
    }

    @Override
    public void run() {
        quickSort(a,lb,ub);
    }
}
```