Evaluation of Parallelised Mergesort and Quicksort

Jacob Ekedahl
jaceke@kth.se

Alexander Nordlund
alnor@kth.se

February 2019

Abstract

The objective was to test performance of sorting data through parallel computing in an application running on a multicore processor. In modern day computing, with an immense flow of data that is to processed by hardware, subdivision of tasks between core processors can hasten the outcome. To test the performance of parallel computing, several sorting algorithms are used on large quantities of data. Java SE provides tools and a framework to employ parallelism and the "work stealing" algorithm. The result showed that by delicately choosing a threshold value, the overall performance of the application can be overall optimized.

Keywords— Parallelism, Quicksort, Mergesort, Arrays.parallelSort, Arrays.sort, Amdahl's laws, threshold

1 Introduction

The task was to implement the Quicksort and Mergesort sorting algorithms in a parallelised manner in which the processor cores used would be utilized. A further objective was to measure the performances of separate sorting algorithms during parallelised execution, which includes Arraysort and Parallelsort from Java's API. The numerical data would then have to be analysed to assess the benefits of subdivision of work through parallelism.

1.1 Problem Definition

A frequent problem of applications is neglecting the computing power of its host device which in many situations result in subjecting the CPU to a bottleneck hence causing delay and deterioration of the quality of the application. This problem can be contained through parallel computing. Since parallelism requires subdivision of labour to be done in parallel and to return the results of each subdivision, multiple issues will need to be addressed. The issues includes the approach of dividing labor among separate threads, determining a point as to when to cease dividing the problem and to identify the ways in which the performance has been improved through parallelism.

For this assignment, parallel computing of sorting tasks will be timed and measured to prove the benefit of parallelism. The measurements would include the time taken to complete the sorting of an array of floats with the length of 10e8 repeatedly starting from 1 core all the way to 8 cores for every sorting algorithm. The sorting algorithms that will be implemented is Quicksort, Mergesort, Arraysort and Parallelsort.

2 Theory and Background

In this section the tools used and theory with regards to our project are presented.

2.1 Background

According to a study, parallelism is a volatile approach to problem solving since its benefits are provided by the hardware specifications of the host device[1]. Furthermore, it is tasked by the programmer to discern the amount of computational work in which parallelism can lead to favorable cost-benefit outcome in terms of computing power [1]. This is due to a pivotal aspect of parallel computing, called parallel task granularity[1]. An experiment showed that parallelism, depending on the design can suffer heavy load imbalance due to the inefficient implementation parallelism or critical overhead from excessive specifications of parallelism[1].

2.2 Theory

In this section the theory behind parallelism in regards to performance are presented as well as Javas ForkJoin framework to handle thread pools.

2.2.1 Amdahl's Law

Amdahl's model provides a mathematical basis where a fixed amount of work can be processed faster through parallelism which is achieved from larger quantities of work being computed with a proportional number of core processors[2].

$$S(n) \leq n/(1 + f(n-1)).$$

Amdahl's first law defines how higher speed is achieved when n processors are handling f amount of work if f is processed sequentially [3]. S(n) corresponds to the speed-up value[3].

$$\frac{na}{n+a-1} \leq S(n) \leq min(n, a)$$

Amdalh's second law shows how parallelism can achieve higher speed[3]. The a corresponds to the working processors. This formula stipulates that overhead is independent from n[3].

The efficiency of Amdahl's model, especially in modern day computing, is subject to debate. The validity of the mathematical proof can be overridden by empirical circumstances[2]. As an example the rate of growth of work can harm the potential in parallel computing in terms of speed if the rate of the growth of the work is too slow or too fast[2].

2.2.2 Moore's Law

Moore's law speculates that speed of hardware computation increases as a result of doubling the amount of hardware components every 24 months[4]. The decrease of the size of the circuits are proportional to operational clockwork speed[4]. Additionally, added components to the hardware increased the operational capacity of the hardware's device and lowered the time needed to compute a problem[5]. The prime limitation for this advancement is essentially the laws of physics, which was proven as the internal components couldn't tolerate the heat originating from the increased clock speed[4].

2.2.3 Parallelism from Java SE

Java's ForkJoin framework implements the ExecutorService interface and is designed to facilitate the parallelisation of tasks and exploits multi-core hardware[6]. The ForkJoinTask is submitted to a ForkJoinPool, which keeps track of the ForkJoinTask processes and shares similar functionality to ExecutorService[7]. Conventionally the ForkJoinTask invokes a *fork()* operation which triggers the asynchronous execution of work and *join()* which allows the result of the work to be returned at the time of its completion[8]. The main difference between the ForkJoinPool and ExecutorService is that it employs work-stealing methods in which a idle thread assumes the task of a working thread[7]. The developer may define the level of parallelism for the ForkJoinPool[7].

3 Method and Result

The Strategy Design Pattern was used in order to facilitate the reuse of sorting tasks, as well as providing structure to the code. The Parallelisation of sorting tasks was implemented through subclasses of RecursiveAction and assigned to a ForkJoinPool, with each subclass instance focusing on Quicksort and Mergesort respectively. The program is aware of the number of core processors of that the running device possesses. Repeated testing was done in order to infer an appropriate threshold value through observation of time required to complete sorting. The Quicksort and Mergesort algorithms used were not embedded in Java's API , they were written and based on the existing algorithms, though Parallel Array Sort and Array Sort from Java API were used once the parallel task exceeded the threshold quota.

Only the implementation of the Quicksort algorithm used for in-array sorting to optimize performance in terms of memory usage. Reason why it is not implemented in mergesort is because the difficulty level was viewed as to high.

3.1 Finding a Reasonable Threshold

Knowing when to stop dividing the current problem into smaller problems is determined by the threshold value. The objective was to find the threshold which would result in the lowest average time to sort an randomized array of floats with the size of 10e8. The tests conducted to find this value were both in regards to the size of the array shown in Figure 1 and Figure 2 and the number of cores shown in Figure 3 and Figure 4. The threshold values tested are between 100 to 100 000. The array is randomized only once for each of the sizes. The average value to sort is recorded between iteration 5 to 15. In order to easily compare the result dependent on the size of the array, the time that the array got sorted is divided by (*size of array / 100 000*) and the scale of the y-axis is logarithmic.

The smaller sizes have higher variances and are thus disregarded. The data from Figure 4 and Figure 3 did not display and there was apparently no substantial gain if one was to adjust the threshold dependent on number of cores.
The threshold values chosen are the local minimum of the time it took to sort the array with the size of 10e7. For Mergesort it is 22100 and for Quicksort 55100.

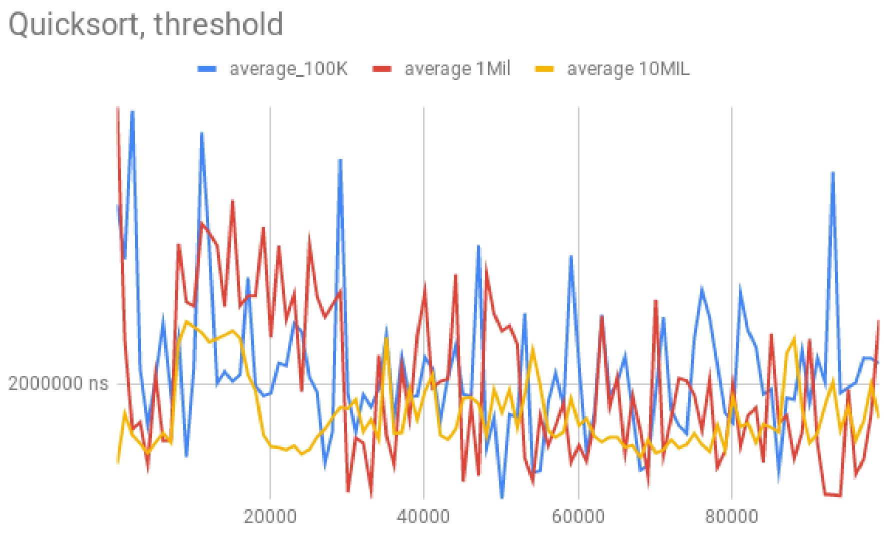

Figure 1: Time to sort a randomized array at different sizes, at 8 cores using parallelised Quicksort

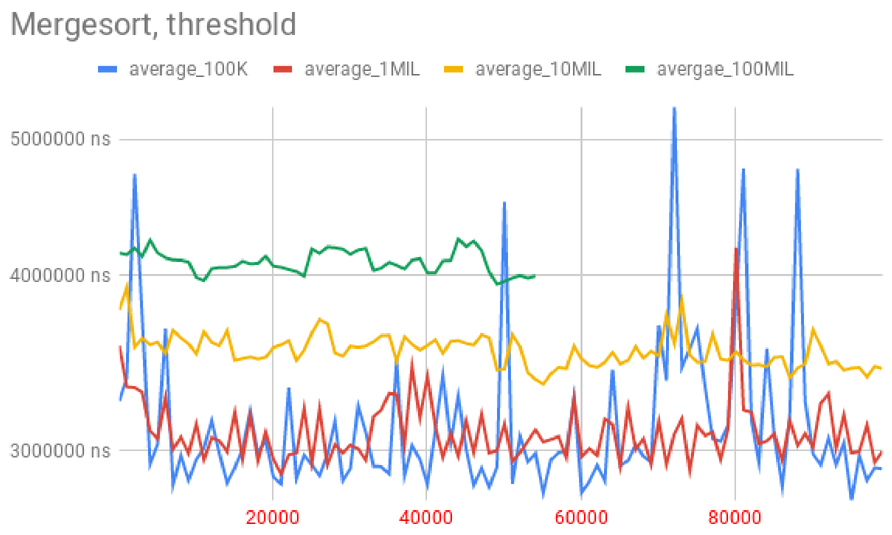

Figure 2: Time to sort a randomized array at different sizes, at 8 cores using parallelised mergesort

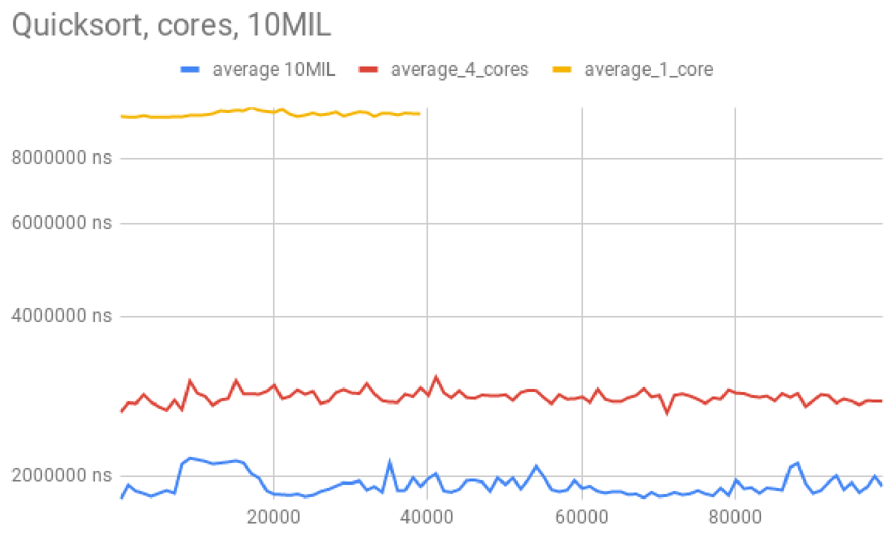

Figure 3: Performance of parallelised Quicksort at different core sizes and threshold values

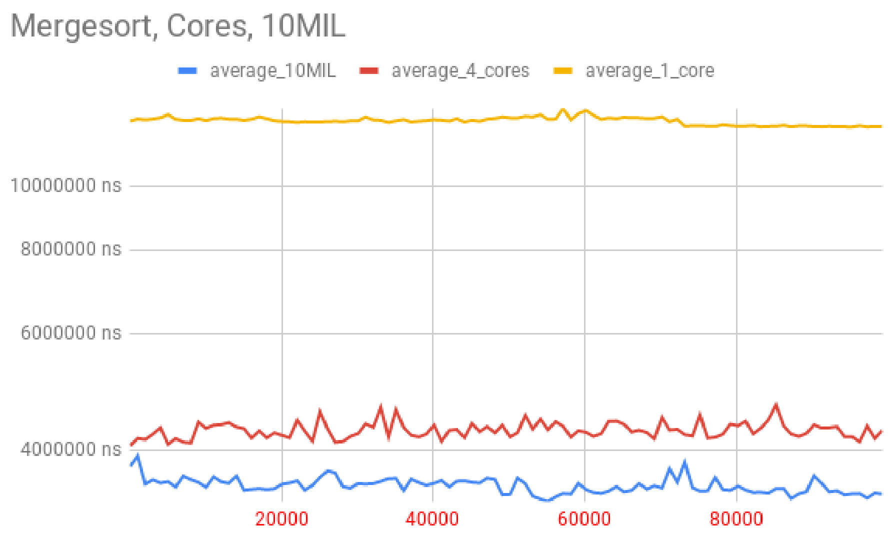

Figure 4: Performance of parallelised mergesort at different core sizes and threshold values

3.2 Implementation and result of Quicksort

In this section the result and methods used for implementing the parallelised version of Quicksort are presented.

3.2.1 Pivot value

The pivot value used is the last value in an arbitrary thread partition. Other pivots tried were aimed at getting equally sized partitions by picking a pivot close to the median or by randomizing the pivot to improve the average time complexity to O(N log N). These solutions added time complexity and proved to perform worse than our final solution.

3.2.2 Division of the original problem into sub-problems

To parallise Quicksort, the original problem of sorting the entire array has to be divided into coherent, separate sub-problems. The technique used is called divide and conquer. *Algorithm 1* explains how we solved this as well to when a thread stops dividing the problem and start sorting its partition. The sorting algorithm implemented on this step is Javas Array.parallelSort. The reason being was that it performed better than our own implementation of Quicksort as well as a combination of Quicksort and insertionsort for partitions at smaller sizes.

Algorithm 1 Quicksort - Divide and Conquer

```
procedure COMPUTE
    $arr \leftarrow array\ to\ sort$
    $low \leftarrow startIndex$
    $high \leftarrow endIndex$
    $threshold \leftarrow 55100$
    $compute$:
    if $high - low < threshold$ then
        $pi = partition(arr, low, high)$
        $invokeAll(newQuicksortTask(arr, low, pi - 1), newQuicksortTask(arr, pi + 1, high))$
    if $high - low >= threshold$ then
        $sort(arr, low, high)$. return
```

3.2.3 Performance of the Implemented Parallelised Quicksort

In Figure 5, the results illustrates how fast the parallelised implementation of Quicksort can sort an randomized array with the size of 10e8.

For every iteration a new randomized is constructed and the iterations 5 to 15 are recorded. The graph displays the median value as well and the the error bars are showing the confidence interval of 95%.

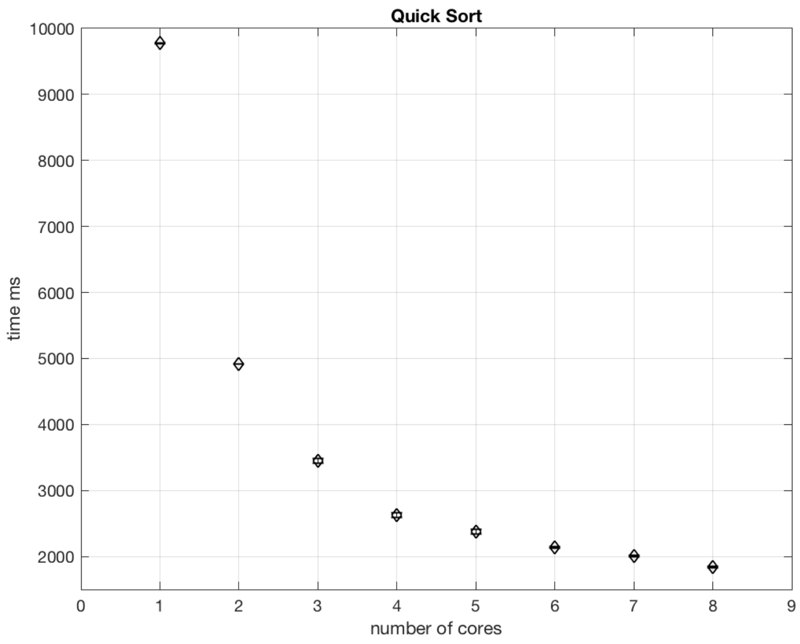

Figure 5: Time for Quicksort to sort an randomized array with the size of 10e8

3.3 Implementation and Result of Mergesort

In this section the methods used and the result of our implementation of parallelised Mergesort are presented.

3.3.1 Division of the Original Problem into Sub-problems

The divide and conquer technique is used to divide the problem into smaller sub-problems until the size of the array which the thread is responsible for is below the chosen threshold at 22100.

When the problem can be divided into smaller sub-problems the original thread waits for its children to complete before it starts merging those sorted arrays. Algorithm 2 displays our implementation of the parallelised Mergesort. The sorting algorithm implemented in conjunction with our implementation is Javas Arrays.sort.

Algorithm 2 Mergesort - Divide and Conquer

```
procedure COMPUTE
    $arr \leftarrow$ array to sort
    $low \leftarrow$ startIndex
    $high \leftarrow$ endIndex
    $threshold \leftarrow 22100$
    compute:
    if $low > high$ then return
    $size \leftarrow high - low$
    if $size < threshold$ then
        $sort(arr, low, high)$.
    if $size >= threshold$ then
        $m \leftarrow (low + high) / 2$
        $invokeAll(newMergeTask(arr, low, m), newMergeTask(arr, m + 1, high))$
        $merge(m)$
```

3.3.2 Performance of the Implemented Parallelised Mergesort

Figure 6 shows how fast the parallelised implementation of Mergesort can sort an randomized array with the size of 10e8.

For every iteration a new randomized is constructed and the iterations 5 to 15 are recorded. The graph displays the median value as well and the the error bars are showing the confidence interval of 95%.

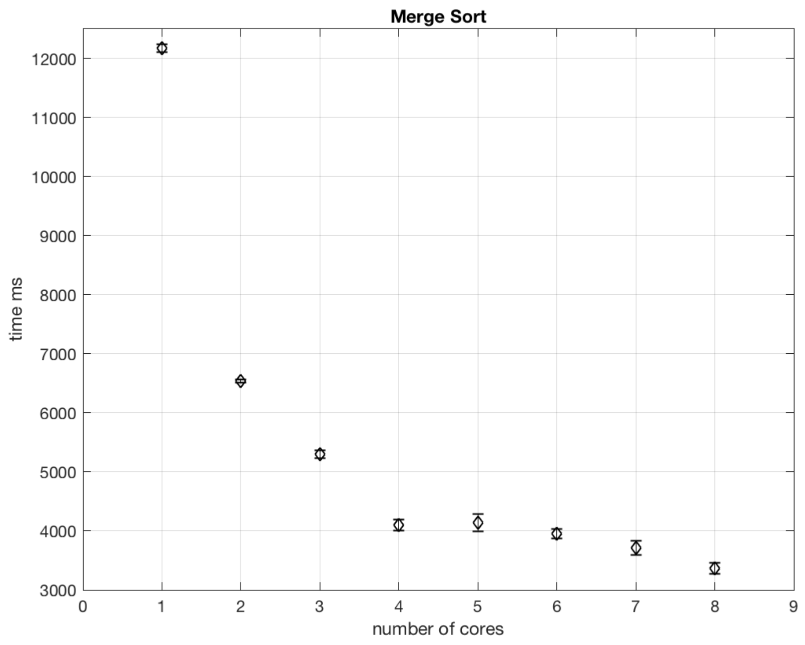

Figure 6: Time for mergesort to sort an randomized array with the size of 10e8

3.4 Overall Result In Relation to Javas Sorting Algorithms

Figure 7 shows the average time it took for each sorting algorithm to sort an array with the size of 10e8. Javas standardized Arrays.parallelSort always use the maximum amount of cores which is 8 cores in our case and Javas Arrays.sort use only one. For every iteration they sort a copy of the same generated array. Our result show that our implementation of the parallelised version of Quicksort overperforms Javas Arrays.parallelSort at 8 cores.

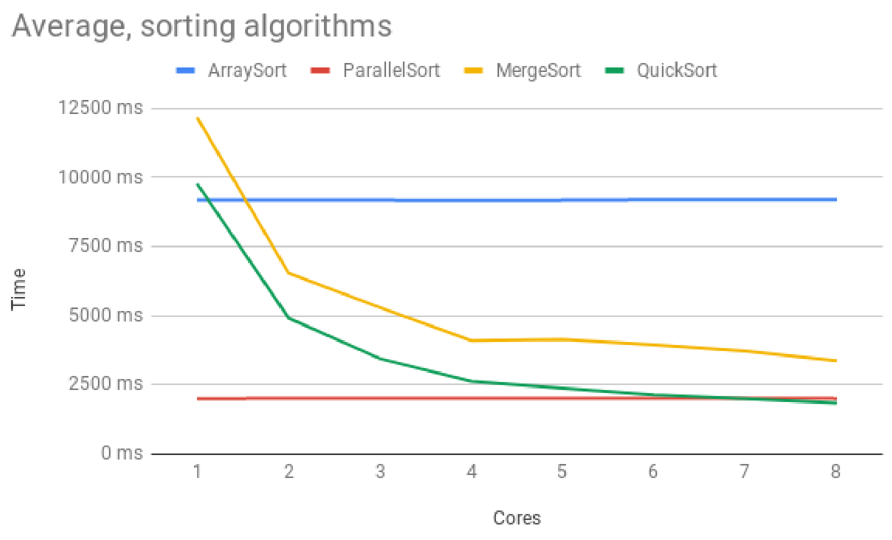

Figure 7: Comparisons between the sorting algorithms

4 Discussion

The reason why our implementation of the parallelised Quicksort overperformed Javas Arrays.parallelSort is most likely due to how we chosen to designate the pivot value and/or the threshold value. It otherwise works very similarly to Arrays.parallelSort as it also implements ForkJoinPool and allocates a threshold value for when to stop making partitions.

The reason why the implementation of Quicksort outperforms the implemention of Mergesort is most likely due to one thread waiting for its childthread to complete before merging and thus adding overhead when the ForkJoin pool has to reassign it for other tasks until it can start merging. There might also be benefits to implement Arrays.parallelSort instead of Arrays.sort at the latter part of the sorting algorithm. However, not enough tests were conducted to validate this speculation.

A future improvement for finding a better value for the threshold is to train a neural network which is dependent of a large set of generated data such as the one generated for Figure 1 and Figure 3. Examples of input data might be number of cores, current load of the processor, size of the array and possibly which types of objects are to be sorted.

Other improvements to validate this result is to use larger arrays to negate the variation caused by some arrays being preemptively more sorted than others and do more than 10 measurements per core.

References

[1] Albert Noll and Thomas Gross. Online Feedback-Directed Optimizations for Parallel Java Code. *ACM SIGPLAN Notices - OOPSLA '13*, 48:713–728, 2013.

[2] Michael T. Heath. A Tale of Two Laws. *The International Journal of High Performance Computing Applications*, 29:320–330, 2015.

[3] Shikharesh Majumdar. On the energy-performance tradeoff for parallel applications. In *EPEW'10 Proceedings of the 7th European performance engineering conference on Computer performance engineering*, pages 67–82, 2010.

[4] Peter J. Denning and Ted G. Lewis. Exponential Laws of Computing Growth. *Communications of the ACM*, 60:54–65, 2016.

[5] Juan R. Cebral Fernando F. Camelli Fumiya Togashi Joeseph D. Baum Hong Luo Eric L. Mestreau Orlando A. Soto Rainald Löhner, Chi Yang. *Moore's Law, the Life Cycle of Scientific Computing Codes and the Diminishing Importance of Parallel Computing.* Elsevier Science, 2006.

[6] Oracle. Parallelism, 2017.

[7] Oracle. Forkjoinpool (java platform se 8), 2018.

[8] Oracle. Forkjointask (java platform se 8), 2018.