Algorithm Engineering
WS 22/23

Jakob Sanowski
Fiona Riesterer

Comparison of classical and parrallel Merge- and Quicksort

Abstract

1 Introduction

We implement and compare the practical performance of two well known sorting algorithms when parallelised, namely, merge-sort and quick-sort. We first explain the basic algorithms, then describe methods to parallelise the two algorithms, and finally discuss the actual implementation. In the experiments, we test and compare our implementations on different amout of cores with instances of different sizes. The results suggest quick-sort is faster on bigger instances.

2 Preliminaries

We want to solve the following problem: Given an unsorted sequence of integers, we want to generate a sorted sequence of those integers.

Both algorithms, Mergesort and Quicksort are recursive algorithms and are using the divide-and-conquer principle. Further explanations are given in the following.

2.1 Mergesort

Classical Mergesort divides a given sequence of integer into sequences half the length of the initial one until the sequence has length one. Then these small sequences are merged recursively together by comparing the contents and creating a sequence the size of the initial two combined. This is done until all subsequences were merged together, which will then be the sorted result. Since the input sequence is splitted in half, the running time of classical Mergesort is in $O(n \cdot \log n)$.Algorithm 1 provides the pseudocode of classical Mergesort.

Algorithm 1: Classical Mergesort

```
Input: unsorted integer Array A
Output: sorted integer Array A
mergesort(Array A) begin
    if $A.length == 1$ then
        return A
    else
        sortedA1 ← mergesort($A.firstHalf$)
        sortedA2 ← mergesort($A.secondHalf$)
        return merge(sortedA1, sortedA2)

merge(Array A1,Array A2) begin
    Array A
      ← $newArray[A1.length + A2.length]$
    indexA1 ←0
    indexA2 ←0
    indexA ←0
    while $indexA1 < A1.length \wedge$
      $indexA2 < A2.length$ do
        if $A1[indexA1] \leq A2[indexA2]$ then
            A[indexA]← $A1[indexA1]$
            indexA1++
        else
            A[indexA]← $A2[indexA2]$
            indexA2++
    while $indexA1 < A1.length$ do
        A[indexA]← $A1[indexA1]$
        indexA1++
    while $indexA2 < A2.length$ do
        A[indexA]← $A2[indexA2]$
        indexA2++
    return A
```

Algorithm 2: Classical quick-sort

```
Input: unsorted integer Array A
Output: sorted integer Array A
quicksort(Array A) begin
    if $A.length == 1$ then
        return
    pivot ← $random(0, A.length)$
    partition(A, pivot)
    quicksort(A.smaller)
    quicksort(A.bigger)
    return
partition(Array A, p) begin
    j ←0
    for $i \leftarrow 0$ to $A.length$ do
        if $A[i] < p$ then
            A.swap(i,j)
            j++
    A.swap(j,p)
```

2.2 Quicksort (Algorithm 2)

Quicksort chooses one random element of the input, called pivot. This pivot is then comapred to all other elements to create two lists; one with all the elements smaller than the pivot and one with all the elements bigger than the pivot. This can be done in place, so that the pivot is now at the position it would be in the sorted sequence. Quicksort is then applied recursively on the two lists. The running time of Quicksort is in $O(n \cdot \log n)$.

3 Algorithm & Implementation

We paralleized both, Mergesort and Quicksort. How they were implemented and constructed is described in the following.

3.1 Details of the Parallel Algorithms

3.1.1 Parallel Mergesort

Parallel Mergesort is similar to Mergesort, only that everytime Mergesort is doing a recursive split, Parallel Mergesort creates a new Thread that can be executed in parallel. To also merge in parallel, the merging step has to be slightly changed. Now, both Threads compare their first element in their sorted sequence via binary search to the sorted sequence of the other thread to find the position in which the element would have been. Adding the indexes together yields to the index of the sorted sequence. You just have to make sure, that no elements occur more than once. Since the Threads do not write on the same indexes and only accesses the other threads sequence to read, this won't result in any problems or false states.

3.1.2 Parallel Quicksort

Parallel quick-sort also creates a new task for each recursive call which can be executed in parallel. The partitioning phase also can be parallelised by spliting the array into smaller arrays which in turn can be executed in parallel. In order to merge the result we calculate the prefix-sums for the results and then write the results to their correct location. The writing also can be done in parallel. In practice it is not sensible to create a new task for each recursive call because it isn't particular cheap. It makes much more sense to create exaclty so much tasks so all processors are similarly used. Because the tasks can take different amounts of time, it makes sense to create a few more than the available number of processors. This way, if a task takes only a short amout of time, another task can be picked up to maximise utilisation.

3.2 Implementation Details

Both algorithms were implemented in Java 18. We used the ForkJoinPool provided by java.util.concurrent for parallelising. The ForkJoinPool-framework provides the crucial benefit of organising all of the threads and having a queue for executing tasks. This reduces the creation of threads to a minimum which is desirable because tread creation is expensive. It also provides worksteeling so waiting threads can pick up other tasks in order to avoid idling threads.

4 Experimental Evaluation

In this section, the experimental setup is described and the results are presented.

4.1 Data and Hardware

The experiments were executed on a machine with a Intel i7 6700K which has 4 real and 8 logical cores with a clock speed of 4.2GHz per core and 16GB of RAM and on a ... with 8 real and 16 logical cores. The data used for testing was created by filling an array with numbers from 0 to the desired number of elements and then shuffling them around. This way we avoid tow elements being the same number. To test on which number of processors our implementations perform the best, we ran 100 test per processor count with 10^6 and 10^7 elements and took the average over all test runs. The second experiment was conducted with instances of size 10^i with $i = 3, \dots, 9$ using the optimal number of processors from the first experiment.

4.2 Results

The first experiment showed merge-sort getting faster the more processors it has available (Figure 1), tho the increase in performance for more than two processors is very small. Qick-sort on the other hand did not show an reduction in running time. Not just that it's running time increased slightly when parallelized. The reason for this behavior probably is that, first parallel quick-sort requires coping the results of the partitioning sub-tasks and second thread management is a quite expensive task.

For this reason we repeated the experiment with 10^7 elements (Figure 2). As we can see, quick-sort now also gets significantly faster when parallelized, tho also does not really benefit from more processors. One feasible explanation could be, the sub tasks get to small to quickly and don't really benefit from the extra processors.

Using the results from the first experiment we used 8 processors to do the second experiment because merge-sort clearly benefits from more processors and quick-sort did only slightly worse than with 2 processors.

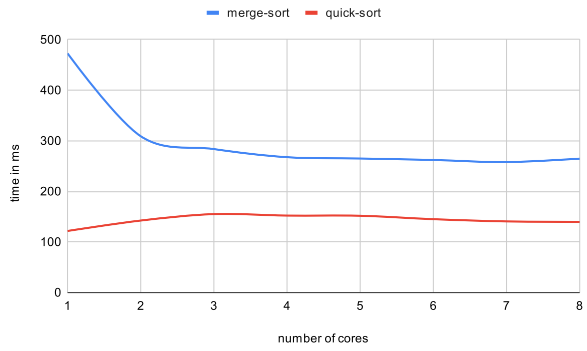

Figure 1: Running times of parallel merge and quick sort with 10^6 elements.

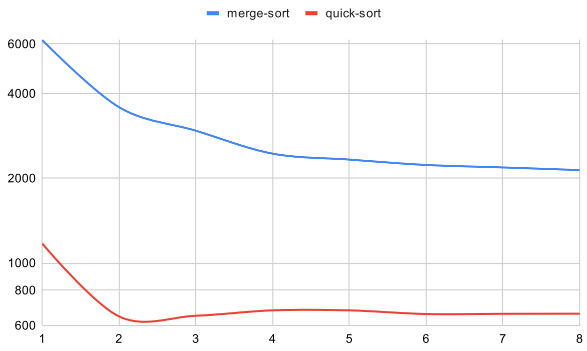

Figure 2: Running times of parallel merge and quick sort with 10^7 elements. The time is in logarithmic scale.

The experiment showed quick-sort starting to benefit from parallelism for a instance size between 10^6 and 10^7. This corresponds with our findings from earlier. Merge-sort on the other hand shows no improvement to sequential merge-sort. The reason for this seems to be Arrays.binarySearch which takes up a lot of the total computation time. We also tested merge-sort on a machine with 16 cores and could observe parallel merge-sort getting faster than sequential merge-sort with more than 8 available cores on very large instances.

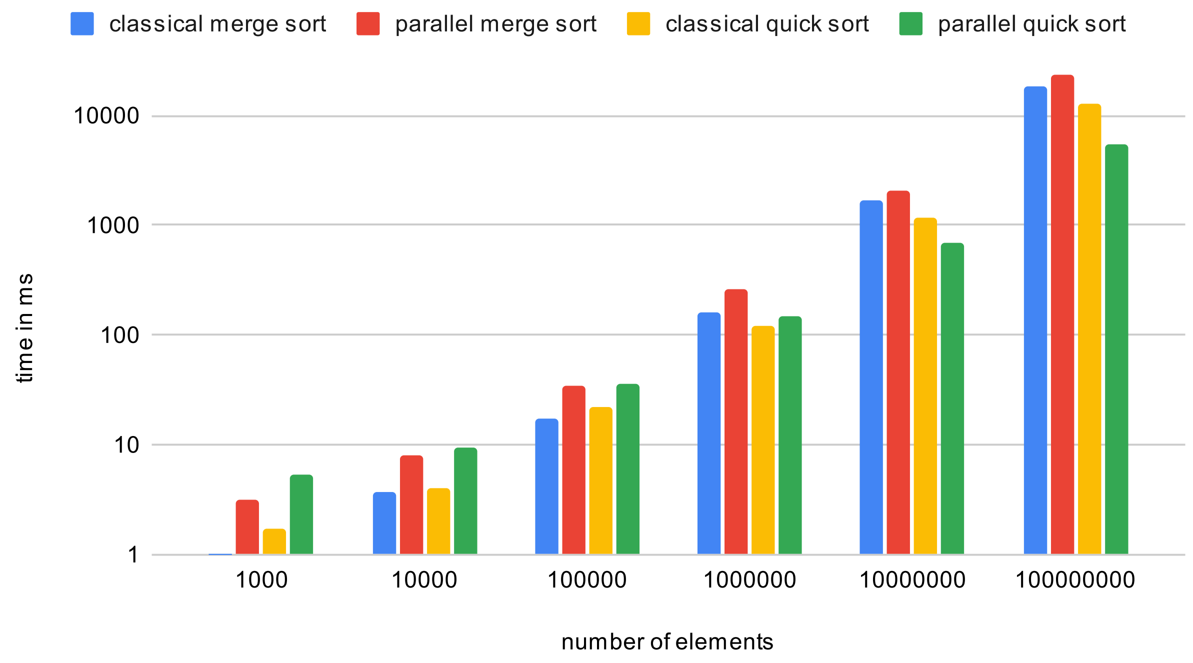

Figure 3: Running times of the different algorithms with 8 processors. Time is in logarithmic scale.

5 Discussion and Conclusion

After reviewing the two algorithms, we can say parallel quick-sort performs the best overall but didn't seem to profit from more than 2 cores too much. Even after heavily analysing parallel quick-sort we could not figure out why quick-sort didn't show a performance increase with more than two cores. Our guess is due to the uneven splitting into subtasks the cores can't be utilized evenly. It would be interesting to test if this behavior persist with instances of size 10^9 or bigger but sadly our machines don't have enough RAM for that.

6 References

The references list the external resources used in the work at hand. LaTeX offers special ways to list those resources.