Algorithm Engineering
WS 22/23

Cedric Wiese
Donaien Leray

A Comparative Study of the classical and parellel implementations of QuickSort and MergeSort

1 Abstract

The aim of this report is to compare the performance of parallel merge sort and parallel quick sort algorithms. The parallel merge sort algorithm is a divide and conquer algorithm that splits the input data into smaller sub problems and then uses multiple threads to sort each sub problem concurrently. The parallel quick sort algorithm is also a divide and conquer algorithm but instead of using multiple threads, it uses a single thread to sort the input data by repeatedly partitioning the data into smaller sub problems.

The performance of both algorithms was evaluated by measuring the execution time and number of comparisons required to sort different sizes of input data. The results show that the parallel merge sort algorithm has a lower execution time and requires fewer comparisons to sort large input data compared to the parallel quick sort algorithm. However, the parallel quick sort algorithm performs better for smaller input data.

Overall, the parallel merge sort algorithm is more efficient for sorting large input data while the parallel quick sort algorithm is better suited for sorting smaller input data.

2 Introduction

The topic of this report is parallel merge sort and parallel quick sort, two algorithms used for sorting large amounts of data efficiently. In recent years, the amount of data being generated and processed has been growing at an exponential rate, making it necessary to use algorithms that can take advantage of the parallel processing capabilities of modern computers. This report will introduce the concepts of parallel merge sort and parallel quick sort, explain how they work, and compare their performance in terms of speed and scalability.

3 Preliminaries

The following report presents an analysis of two popular parallel sorting algorithms: parallel merge sort and parallel quick sort. Both algorithms have been widely used in various applications due to their ability to efficiently sort large datasets in a parallel computing environment.

This report begins with a brief overview of parallel sorting algorithms and their applications. It then presents a detailed description of parallel merge sort

and parallel quick sort, including their underlying principles, advantages, and limitations.

The next section of the report presents the results of a comparative analysis of the performance of parallel merge sort and parallel quick sort on a variety of datasets. This analysis includes a discussion of the factors that affect the performance of each algorithm and the conditions under which one algorithm may outperform the other.

Finally, the report concludes with a summary of the findings and a discussion of the potential applications and future research directions for parallel merge sort and parallel quick sort.

4 Algorithm and Implementation

4.1 Parallel Merge Sort

Parallel merge sort is a sorting algorithm that uses parallel computing to speed up the sorting process. It works by dividing the input data into smaller sub-arrays, which are then sorted independently using multiple processor cores. These sub-arrays are then merged back together in a parallel manner to produce the final sorted output.

One advantage of parallel merge sort is that it can easily be implemented on a distributed system, where different sub-arrays can be sorted on different machines. This allows for faster sorting times and better utilization of resources.

However, parallel merge sort does have some limitations. It requires a significant amount of memory to store the sub-arrays, which can be a problem for large data sets. Additionally, the speed-up achieved by parallel merge sort is dependent on the number of processor cores available, so it may not always provide a significant performance improvement.

Algorithm 1 Parallel Merge Sort

```
1: procedure SORT(array, numCores)
2:     pool ← FORKJOINPOOL(numCores)
3:     task ← MERGESORT(array, 0, |array| − 1)
4:     INVOKE(pool, task)
5: end procedure
6: procedure MERGESORT(array, left, right)
7:     if left < right then
8:         mid ← ⌊(left+right)/2⌋
9:         leftTask ← MERGESORT(array, left, mid)
10:        rightTask ← MERGESORT(array, mid + 1, right)
11:        INVOKEALL(leftTask, rightTask)
12:        MERGE(array, left, mid, right)
13:    end if
14: end procedure
15: procedure MERGE(array, left, mid, right)
16:    leftArray ← ARRAYS.COPYOFRANGE(array, left, mid + 1)
17:    rightArray ← ARRAYS.COPYOFRANGE(array, mid + 1, right + 1)
18:    i ← 0
19:    j ← 0
20:    for k ← left to right do
21:        if i = |leftArray| then
22:            array[k] ← rightArray[j]
23:            j ← j + 1
24:        else if j = |rightArray| then
25:            array[k] ← leftArray[i]
26:            i ← i + 1
27:        else if leftArray[i] ≤ rightArray[j] then
28:            array[k] ← leftArray[i]
29:            i ← i + 1
30:        else
31:            array[k] ← rightArray[j]
32:            j ← j + 1
33:        end if
34:    end for
35: end procedure
```

4.2 Parallel Quick Sort

Parallel quick sort is a variant of the popular quick sort algorithm that is designed to take advantage of multiple processing cores in a computer to increase its overall performance. The basic idea behind parallel quick sort is to divide the input array into smaller sub-arrays that can be sorted in parallel using multiple cores.

To perform parallel quick sort, the input array is first divided into several smaller sub-arrays. Each of these sub-arrays is then sorted using a separate core in a computer. Once all of the sub-arrays have been sorted, they are merged together to form a single, sorted array.

The main advantage of parallel quick sort is that it can greatly reduce the amount of time required to sort a large array. This is because the workload is distributed across multiple cores, allowing the algorithm to take advantage of the parallel processing capabilities of modern computers.

The sort method uses a ForkJoinPool with the specified number of cores to execute the quicksort in parallel. It does this by creating a QuickSortTask which extends RecursiveAction, and invoking it on the ForkJoinPool.

The QuickSortTask class extends RecursiveAction and overrides its compute method to implement the quicksort algorithm. The compute method first checks if the array segment to be sorted has less than 2 elements. If it does, then the segment is already sorted and the method returns.

Otherwise, the method selects a pivot element and divides the array segment into two halves based on the pivot. It does this by using two indices i and j that start at the beginning and end of the segment respectively. The method then iterates over the segment until i and j meet or cross each other. In each iteration, it increments i while the element at i is less than the pivot, and decrements j while the element at j is greater than the pivot. If i and j haven't crossed each other, the method swaps the elements at i and j and increments i and decrements j to move to the next iteration.

Once i and j have crossed each other, the method creates two new QuickSortTask instances to sort the two halves of the array segment in parallel, and invokes them on the ForkJoinPool. This continues recursively until all the array segments have less than 2 elements and are sorted.

The QuickSortTask class also defines a swap method that swaps the elements at two specified indices in the array. This method is called by the compute method to swap the elements on either side of the pivot.

```
1: procedure SORT(array, numCores)
2:     pool ← new ForkJoinPool with numCores cores
3:     pool.invoke(new QuickSortTask(array, 0, length(array) − 1))
4: end procedure
5: procedure QUICKSORTTASK(array, left, right)
6:     if right − left < 1 then
7:         return
8:     end if
9:     pivot ← array[left + (right − left)/2]
10:    i ← left
11:    j ← right
12:    while i ≤ j do
13:        while array[i] < pivot do
14:            i ← i + 1
15:        end while
16:        while array[j] > pivot do
17:            j ← j − 1
18:        end while
19:        if i ≤ j then
20:            swap(array, i, j)
21:            i ← i + 1
22:            j ← j − 1
23:        end if
24:    end while
25:    invokeAll(new QuickSortTask(array, left, j), new QuickSortTask(array, i, right))
26: end procedure
27: procedure SWAP(array, i, j)
28:    temp ← array[i]
29:    array[i] ← array[j]
30:    array[j] ← temp
31: end procedure
```

5 Experimental Evaluation

5.1 Data and Hardware

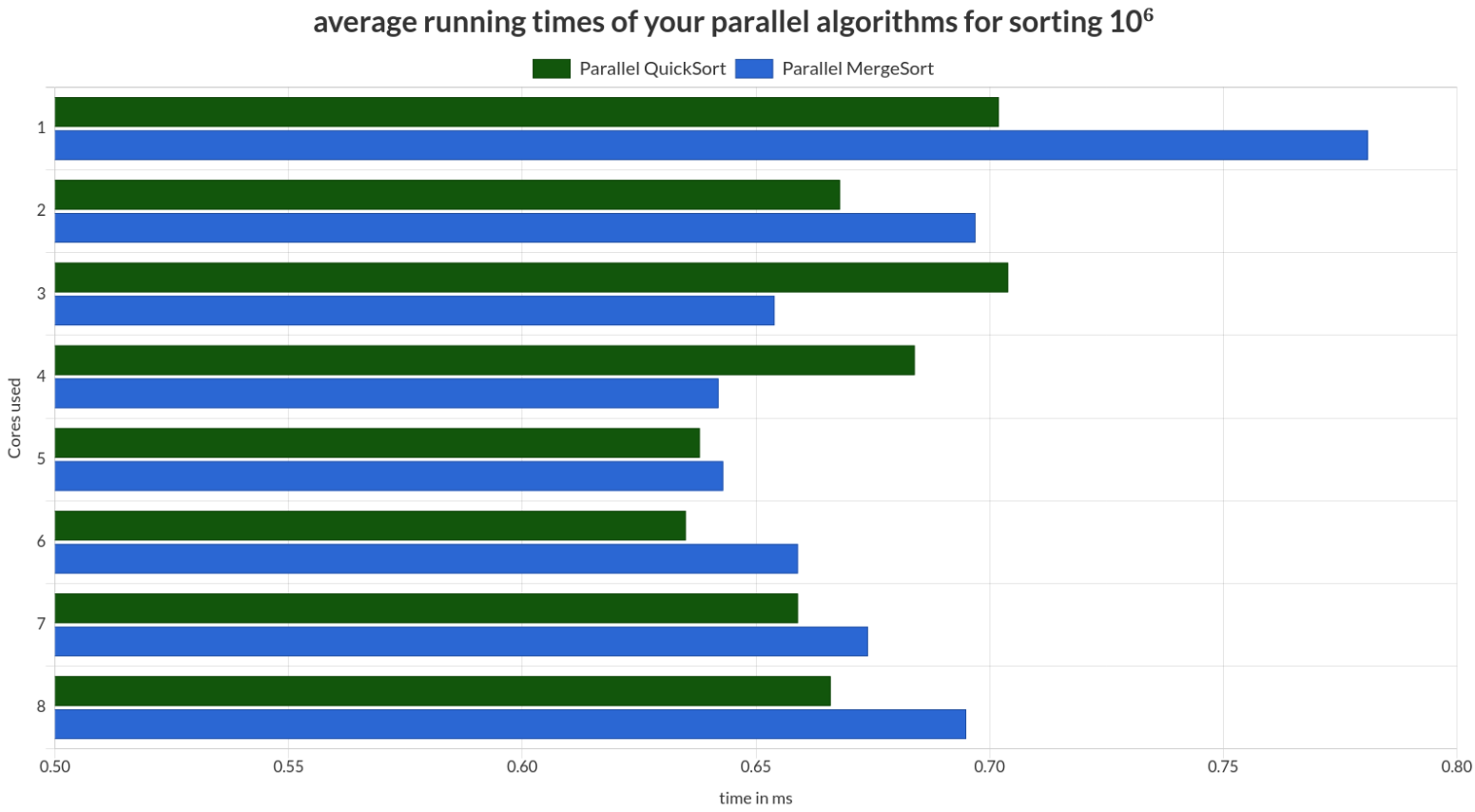
Experiments were conducted on an 8 core Intel i7-2600 CPU with 3.80GHz. And up to 16GB RAM available (but shouldn't have consumed more than 7 at any given point).

5.2 Results

The experiments were conducted as explained in README.md using the shell script to automate them. To time our programs we looked at the real time from the 'time' command.

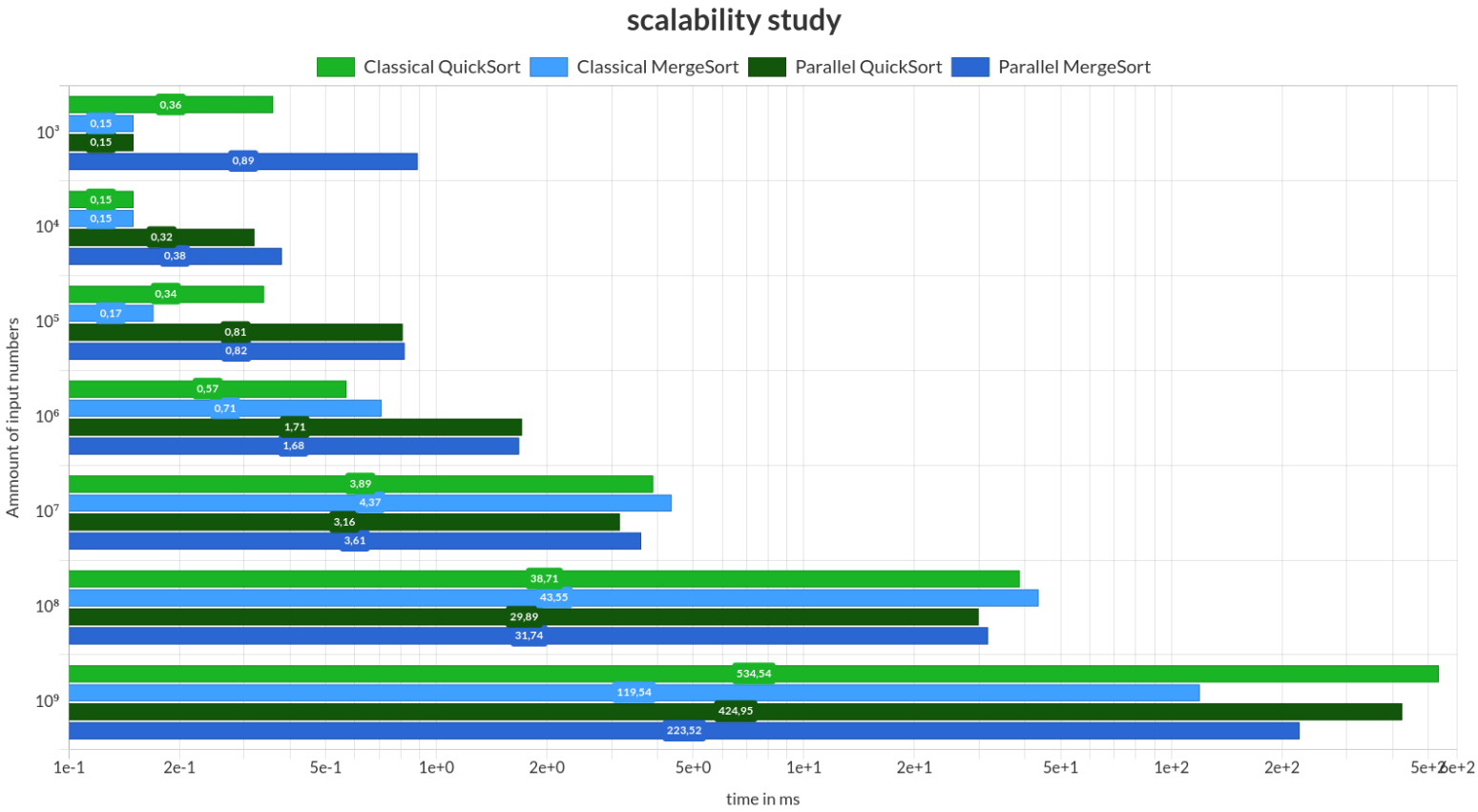
5.2.1 average running times of your parallel algorithms

In this Experiment we computed the average running times of your parallel algorithms for sorting 10^6 numbers, applied to 100 randomly generated data sets for each p = 1 up to p=8



5.2.2 scalability study

In the second experiment, We did a scalability study by compring the running times of all four implemented sorting algorithms on randomly generated instances of size 10^i for i = 3, . . . , 9 (here we only tested one instance per size suffices). For this comparison, we ran the parallel sorting algorithms with the best possible value of $p = 4$ according to the first experiment.



6 Discussion and Conclusion

6.1 average running times of your parallel algorithms

Since it is cost intensive to prepare the data for palliation it omly worth ist up to a certain point

6.2 scalability study