Parallelized Sorting Algorithm

vak Parallel Computing

jaar 2017

Authors: Stefan Schenk
500600679

Remco de Leeuw
500692449

Hogeschool van Amsterdam
19-04-2017

Parallel Computing
Karthik Srinivasan

Table Of Contents

Introduction

For the subject Parallel Computing, students are required to parallelize an algorithm, in order to understand and be able to apply multicore software development techniques on a small scale. The assignments in this subject have been made separately, and have been merged together in this document, that is handed in as a final assignment. It is very important to notice that different tests were performed on different machines, and therefore assignment benchmarks can't be cross-compared.

Implementations of the mergesort algorithm are handwritten and unique. For further reviewing of these implementations visit Gitlab, ("GitLab," n.d.).

1. Case Study Selection

We will focus on the following algorithms in this assignment: Shellsort, Mergesort and Quicksort. The reason why we've picked these three algorithms is because they are relatively easy to understand. So, the reader, but also the writers of this document can focus on the parallelisation of the algorithms instead of trying to understand the complexity of the algorithms used.

The goal of this document is to give a clear picture about the algorithms, how they can benefit from the parallelisation and how the correctness can be tested.

1.1. Shellsort

1.1.1. Serial solution

The shellsort algorithm is based on the insertion algorithm, but avoids large shifts. The shellsort algorithm creates sub arrays of the array based on a chosen increment. After creating different sub arrays, the algorithm sorts these arrays by making use of the insertion algorithm.

After this step the sub arrays will be merged back into one array. The array is not completely sorted, but the different values are closer to their final index. The final step is doing a insertion sort on the new created array. This way the total number of swapping is reduced.

The worst case scenario of the algorithm is O(n) where n is the total number of elements.

```
Step 1 - Initialize the value of h.
Step 2 - divide the list into smaller sub-list of equal interval h
Step 3 - Sort these sublist using insertion sort
Step 4 - Repeat until complete list is sorted
```

Suppose we have the following array [3, 7, 8, 5, 2, 1, 9, 5, 4]. The increment will be three.

- The sublists will be:
  [3, 5, 9] [7, 2, 5] [8, 1, 4]
- After insertion sort:
  [3, 5, 9] [2, 5, 7] [1, 4, 8]

- After merging back:
  [3, 5, 9, 7, 2, 5, 8, 1, 4]
- Last insertion sort:
  [1, 2, 3, 4, 5, 5, 7, 8, 9]

1.1.2. Parallelised solution

The shellsort algorithm can be parallelised by running every sublist on a different thread or core and let the increment be determined by the available threads or cores. The last step can be done on a single thread or core.

1.1.3. Concurrency gains

In the best case scenario we can keep dividing arrays into smaller subs arrays with a increment of 2 or 3, but even with a unlimited amount of cores and or threads we still need to combine the smaller arrays and perform a insertion sort on a bigger array. So, even in the best case scenario the task will grow to a larger function. So, this parallelization can be called “coarse grained parallelism” because the arrays still need to be combined and sorted with a insertion sort.

1.2. Mergesort

1.2.1. Serial Solution

The mergesort algorithm is based on breaking up an array into smaller sorted arrays that can easily be merged because the subarrays are sorted. The mergesort algorithm has a worst case complexity of O(n log(n)), where n is the number of elements. The following steps describe the outline of the algorithm:

```
Step 1 - if it is only one element in the list it is already sorted, return.
Step 2 - divide the list recursively into two halves until it can no more be
divided.
Step 3 - merge the smaller lists into new list in sorted order.
```

When we take the following unsorted array [3, 7, 8, 5, 2, 1, 9, 5, 4], the algorithm will divide the array into subarrays. It will then recursively call mergesort on both subarrays, causing the entire array to be split up until only single elements remain in the subarrays
[3], [7], [8], [5], [2], [1], [9], [5], [4].

It will then merge them back together, positioning increasingly larger elements next to the smallest element, until the entire array is put back together:

- [3, 7], [5, 8], [1, 2], [5, 9], [4]
- [3, 5, 7, 8], [1, 2, 5, 9], [4]
- [1, 2, 3, 5, 5, 7, 8, 9], [4]
- [1, 2, 3, 4, 5, 5, 7, 8, 9]

1.2.2. Parallelised solution

The mergesort algorithm can easily be parallelised because the algorithm starts off with lots of tiny arrays that can be sorted in parallel, resulting into bigger and less arrays until the full array is sorted.

1.2.3. Concurrency Gains

The first stage of sorting many small arrays can be handled by using a different algorithm until sizable chunks of sorted arrays are created. These fine-grained parallel tasks can be run in lots of different threads sharing memory. In the last stage, larger chunked arrays can be merged with mergesort by individual processors to reduce the overhead, resulting in many but decreasing amount of concurrent arrays to be merged.

1.3. Quicksort

1.3.1. Serial Solution

The quicksort algorithm is based on partitioning an array into smaller arrays. The first part will choose a pivot element from the array that is to be sorted, so that other elements can be compared against this pivot element, resulting into two arrays that are then recursively sorted. The algorithm has a worst case complexity of O(n^2), where n is the number of elements in the unsorted array. There are different variations of executing the quicksort, but they all result in two sections of the array which either contain lower or higher values than the pivot. The following steps describe how the pivot element is picked, and how the array is partitioned and sorted in place:

```
Step 1 - Choose the highest index value has pivot
Step 2 - Take two variables to point left and right of the list excluding pivot
Step 3 - left points to the low index
Step 4 - right points to the high
Step 5 - while value at left is less than pivot move right
```

```
Step 6 - while value at right is greater than pivot move left
Step 7 - if both step 5 and step 6 does not match swap left and right
Step 8 - if left ≥ right, the point where they met is new pivot
```

When we take an unsorted array [3, 7, 8, 5, 2, 1, 9, 5, **4**], the last element, for example, will be used as the pivot. The leftmost element and the element left to the pivot will be compared to the pivot. If the left element is smaller than the pivot, the next element is selected, until an element is found that is bigger than the pivot. The same is done for the right element, but this time it has to be bigger than the pivot for the next element to be selected. When the left element is bigger than the pivot, and the right element is smaller than the pivot, the elements are swapped, like so: [3, 5, 8, 5, 2, 1, 9, **4**, 7].

The point where left and right meet, when left is bigger or equal to right, the first element that is bigger than the pivot is swapped with the pivot, finally giving the output: [3, 1, 2, **4**, 5, 8, 9, 5, 7].

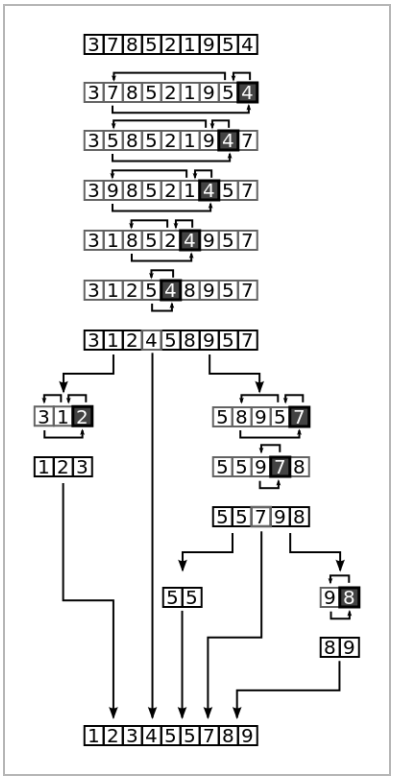

Image 1.3.1.1 - Visualization Of Quicksort

Now we are left with two partitions that are to be sorted using the quicksort algorithm all over again. So to summarize the steps taken using the following steps, we can say that step 1 and step 2 have been taken. Now proceed by following step 3 and 4 to finish sorting the array. The entire sorting of the array can be seen in image 1.3.1.

```
Step 1 - Make the right-most index value pivot
Step 2 - partition the array using pivot value
Step 3 - quicksort left partition recursively
Step 4 - quicksort right partition recursively
```

1.3.2. Parallelised Solution

The quicksort algorithm can either be parallelised by delegating each partitioning step to it's own thread, meaning that the low and high list can sort themselves concurrently. This strategy can be achieved by simply delegating each quicksort execution to its own thread.

The algorithm can also be parallelised by separating individual tasks. The quicksort algorithm consists of three major tasks: the initial quicksort call, the partitioning and swapping of elements. With proper communication, these tasks can be run in parallel. For example, a main thread can determine the pivot and start the execution of partitioning. Other threads can simultaneously start comparing elements within the array to the pivot. Either marking the indexes that need to be swapped, or writing values to a new array. In the first case, a different thread needs to manage the swapping of elements within the array, after the elements have been marked. Requiring a lock to limit the elements that have not yet been checked. In the second case, a new array is created, which will demand more memory, but is less management-intensive.

The first strategy requires an executor to manage quicksort runnables and some processor management.

The second strategy either needs an extra array that stores states of the checked elements, or new smaller arrays that store checked elements. This will require more changes in the quicksort algorithm, and will result in less code reuse.

1.3.3. Concurrency Gains

With the first strategy the problem gets decomposed by delegating partitioning steps to their own threads, we expect that the amount of separate threads will grow exponentially, in the best case, and being executed concurrently. We also expect that the partitioning steps must be run serially still. Whenever an array is partitioned, two new method calls will run in four new threads. In a worst case scenario, only two threads will be utilized. This could be called “coarse grained parallelism”, because of the relative big functionalities that are handled by different threads.

In the second strategy of parallelising the algorithm, the different tasks get executed by different threads. We expect as many threads to run one task in parallel over the entire array by managing and reflecting on the states of a different array, increasing the communication overhead to some extend. If this strategy is properly implemented, we expect many threads to read and write to the same array, but to different indexes in the array in parallel. If this works as planned, we can have a one size fits all solution for all arrays, no matter the size. Therefore we can say that all cores can be fully utilized, making sure that tasks are managed properly. Because the work is split up in small specific tasks, we could call this execution “fine grained parallelism”.

1.4. Testing Correctness

There are a couple of ways to determine whether an array has been sorted in a correct fashion, using a method:

1. By comparing the output results of the sequential solution to the parallelised solution. Assuming that the sequential solution has been tested.
2. By generating a sorted array which is copied, shuffled and then sorted again. The copy of the array should equal the sorted array.

An example of such a method would be the following, as shown in script 1.4.1:

```
public static void checkArray(int[] array) {
	for (int i = 0; i < array.length; i++) {
		if (i != array[i]) {
			printArray(array, "FAIL: Array not sorted...");
			return;
		}
	}
	printArray(array, "SUCCESS: Array sorted");
}
```

Script 1.4.1 - Testing Correctness Method

Of course, this method would fail whenever duplicate elements, or missing elements are present in the array. But for our implementation, we will depend on a filled original array of n size, where each index contains a value that equals the index. This ensures that no duplicate or missing values are present in the array, and makes debugging a lot easier.

2. Basic Threads and Locks

A comparison between serial merge sort and parallel merge sort should show distinct improvements depending on the size of the array. We have timed the amount of time a sorting task would take in milliseconds using the EventProfiler class created by Konin and provided by the HvA. While measuring the time we excluded all setup variables and tasks from the measurements.

2.1. Serial Mergesort Benchmark

Our implementation of the MergeSort class, as can be seen in the repository on Gitlab (“GitLab,” n.d.), contains a method called “serial”, which contains our implementation of the serial version of the Mergesort algorithm based on the Vogella Mergesort Algorithm, which has also been used for the benchmark (Lars Vogel (c) 2009, n.d.).

Array size	Time (ms)
500.000	94.147519 ms
1.000.000	181.398842 ms
2.000.000	351.743466 ms
4.000.000	786.153961 ms
8.000.000	1520.87051 ms
16.000.000	3171.842303 ms
24.000.000	4519.452239 ms
48.000.000	9391.319549 ms
96.000.000	19319.92086 ms

Table 2.1.1 - Serial Mergesort Solution Benchmark Results

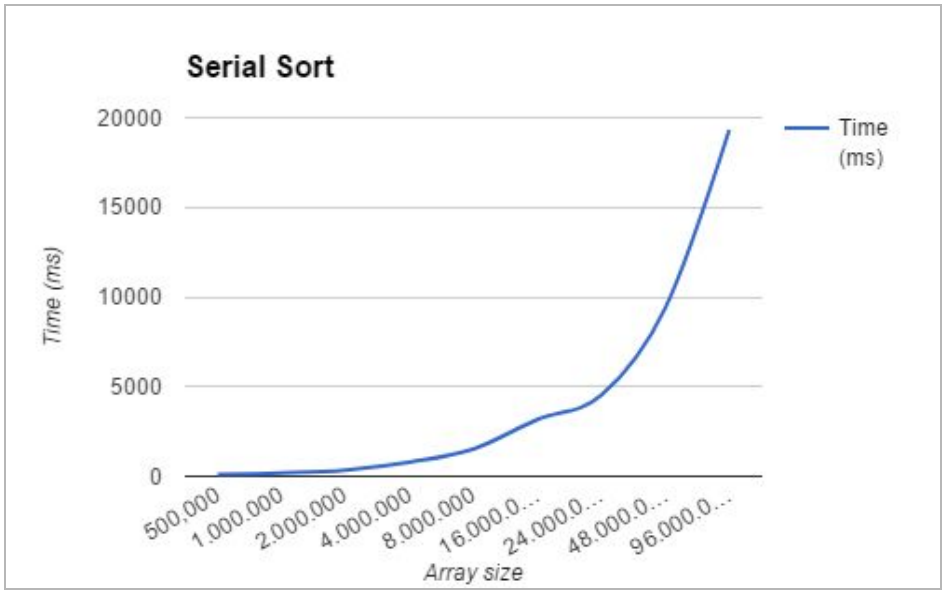

Image 2.1.1 - Serial Mergesort Solution

2.2. Parallel Merge Sort Benchmark

The Parallelized version of the algorithm splits the array into two subarrays, which are sorted in separate threads, and finally merged in the main thread again. The method within the MergeSort class that is responsible for this functionality is called “parallel”.

Array size	Time (ms)
500.000	91.325594
1.000.000	157.032234
2.000.000	290.836823
4.000.000	643.846066
8.000.000	919.276279
16.000.000	1934.266495
24.000.000	2829.137845
48.000.000	5576.584858
96.000.000	15104.24727

Table 2.2.1- Parallel Solution Benchmark Results

Compared to the serial solution, the parallel version is at 500.000 elements just a fraction faster, but when the array size grows, the difference become more significant. The bigger the array gets, the lesser the time it takes to distribute the array over the threads is influencing the results. We can conclude that it is better to sort smaller arrays with the

serial solution thanks to overhead the parallel solution gives, but when the array size grows it is recommended to, at least, parallelize the sorting algorithm.

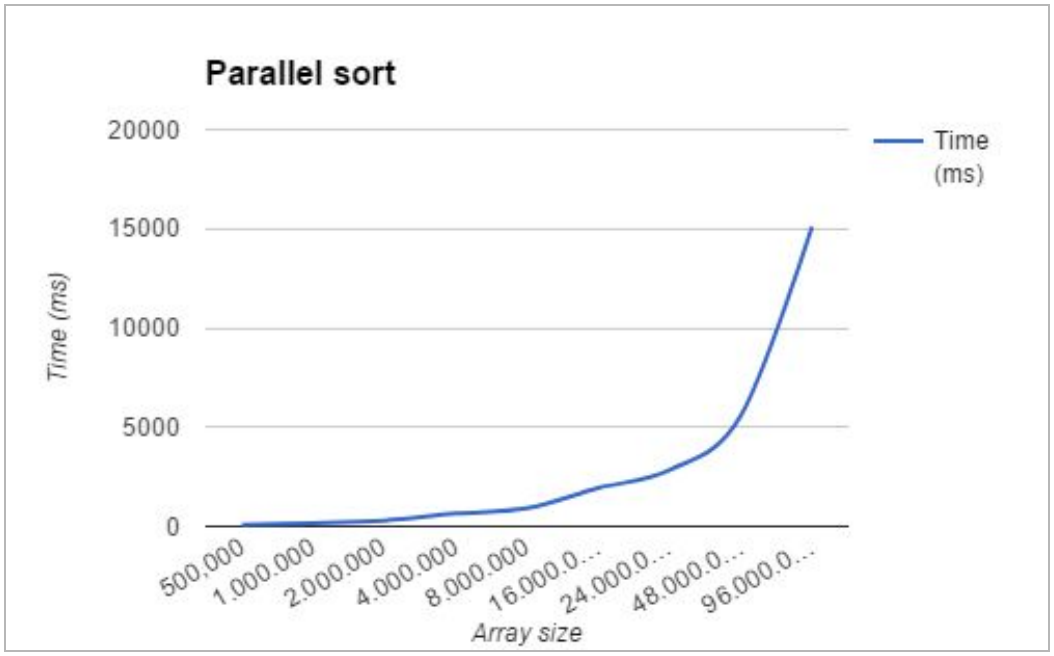

Image 2.2.1- Parallel Solution

2.3. Comparison

We notice a increasing performance gain when the array size increases. For example, the serial solution takes 19 seconds to sort with a array size of 19 million elements, but the parallel one takes about 15 seconds to sort the complete array.

Array size	Time (ms)
500.000	94.147519
1.000.000	181.398842
2.000.000	351.743466
4.000.000	786.153961
8.000.000	1520.87051
16.000.000	3171.842303
24.000.000	4519.452239
48.000.000	9391.319549
96.000.000	19319.92086

Table 2.3.1- Serial Solution

Array size	Time (ms)
500.000	65.152339
1.000.000	100.209949
2.000.000	158.547139
4.000.000	315.602167
8.000.000	626.461757
16.000.000	1253.873786
24.000.000	1780.738239
48.000.000	3666.77248
96.000.000	7654.366008

Table 2.3.2- Parallel Solution

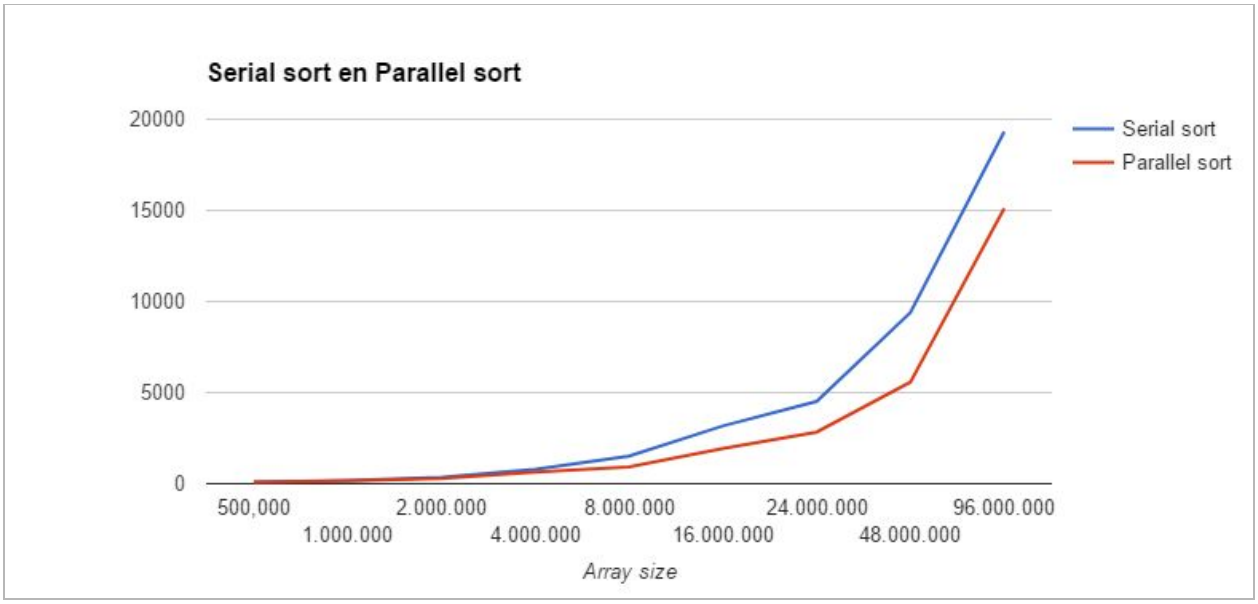

Image 2.3.1 - Serial & Parallel Solution

When we first look at the array sizes, and compare the serialized and parallelized version. We can see that the parallel version performs the sorting task in a less amount of time, but not near a half amount of time. Because we split the problem in two, one could argue that the parallel implementation could in theory be two times as fast.

During execution of the serial and parallel methods, the serial method is starting and finishing without overhead. While the parallel version has to do additional work in setting up threads and then finalizing the two results by merging them together by another merge call.

Just because we're only using a maximum of three cores (because threads can't be split over multiple cores, and the main thread was used for merging the results of the preceding two threads), distributing arrays over even more threads would yield a substantial increase in concurrency. Additionally, the main thread could've acted as the third thread, while it was sitting idle for two spawned threads to join, but this will most likely not have an impact on the results.

3. Optimal Threads and Locks

Because mergesort is a recursive decomposition algorithm, it's recursively decomposing the problem until the arrays only contain a single element. Before this point is reached, it is beneficial to sequentially solve the small problem, so that no resources are wasted on just reading and returning a single element.

```
R solve(Problem<R> problem) {
    if (problem.isSmall())
        return problem.solveSequentially();
    R leftResult, rightResult;
    CONCURRENT {
        leftResult = solve(problem.left());
        rightResult = solve(problem.right());
    }
    return problem.combine(leftResult, rightResult);
}
```

Image 3.1 Concurrent Recursive Decomposition (*From Concurrent to Parallel*, 2016)

3.1. Concurrent Data Structures

For the optimization we've implemented an executorservice along with a blockingqueue, this structure wasn't feasible because not all tasks could be executed out of order. With mergesort, each division and merge step is depending on preceding and future steps.

As our second solution we've used the fork-join pattern to manage threads, and this time we haven't used a particular concurrent data structure to manage parallelisation, because the fork-join pattern would define the amount of threads that could possibly run up until the threshold would be met.

3.2. Concurrency Pattern

As stated, we've used the fork-join pattern because it manages each depth of parallel execution to finishes before the new depth is executed of the merging tree. The mergesort algorithm is a divide and conquer algorithm, for which a fork-join pattern is an ideal use case. Secondly, we've considered mapreduce, which could have been used in a Java 8 stream. Mapreduce would have to run each depth level, which is essentially what forkjoin is already achieving. The method that is run in the benchmark is called "forkjoin" contained in the MergeSort class.

3.3. Benchmark

When comparing the results with the normal parallel version of the algorithm, it becomes evident that there has been an increase in concurrency, using the same array for testing, per array size. The implementation of the fork-join algorithm is more efficient in managing threads. Compared to the previous parallel version, which only used two threads to sort the array, this version may have as many threads as the system allows, and thus has a much higher level of parallelism.

Array size	Normal **Time (ms)**	Optimized **Time (ms)**
500.000	65.152339	64.692417
1.000.000	100.209949	116.253957
2.000.000	158.547139	153.241072
4.000.000	315.602167	233.047706
8.000.000	626.461757	390.180265
16.000.000	1253.873786	740.056536
24.000.000	1780.738239	1199.232937
48.000.000	3666.77248	2327.205824
96.000.000	7654.366008	5058.238887

Table 3.3.1 Comparison Benchmark Between Optimized And Normal Parallel Sort

The bigger the unsorted array, the bigger the difference in performance gain. Still, the implementation introduces more overhead because an exponential amount of Sorter objects must be created for each depth of the mergesort algorithm. And maybe there are solutions using different concurrent datastructures that minimize communication overhead.

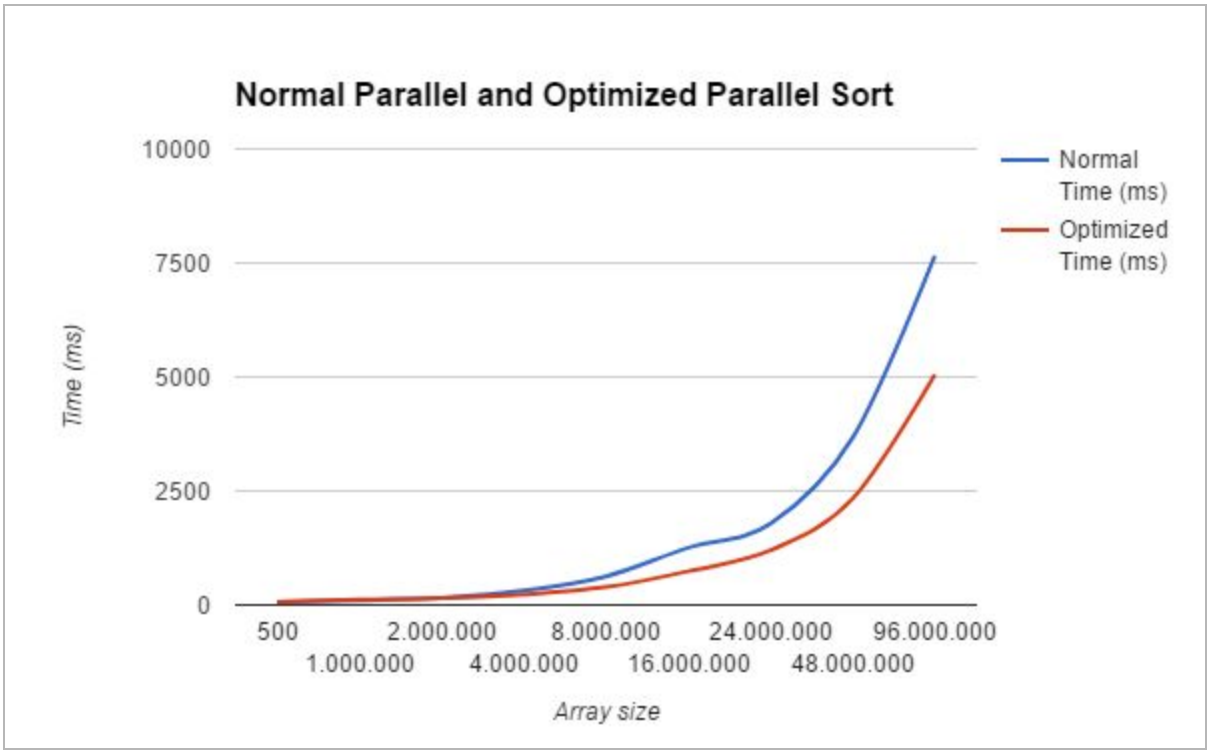

Image 3.3.1 Comparison Benchmark Between Optimized And Normal Parallel Sort

Within any system, this implementation would take full advantage of all processor cores because of the vast amount of threads that are spawned. This, however, doesn't imply that the actual work that is done is really usefull. For example, the overhead of spawning a thread, merely for returning an array or an element, may be unbeneficial. Therefore we could state that this implementation works when a threshold is set properly, depending on the task and system it is used for.

4. CSP

Up until this point, we've implemented three different solutions of mergesort. In order to communicate and delegate mergesort work, arrays have to be transmitted and sorted for each step down the tree. For this assignment we've tried two of our established approaches resulting in our final solution.

4.1. Solution

We've chosen the Java Message Service and ActiveMQ to delegate work to different nodes. Our first solution had a master node delegating an array to slave nodes running two processes. The first process would wait for the first queue to contain an unordered array to be split. When split, the arrays are put back on the same queue, unless the array size is smaller than an arbitrary threshold. In that case, the array will be sorted, and put on the second queue. The second process would merge two arrays and put them back on the same queue unless the length of the array equals the length of the input array provided by the master. This caused some problems regarding communicating which arrays needed to be merged, and when the actual result was obtained.

Our new solution is based on splitting the unsorted array into an amount of subarrays, preferably equalling the amount of available nodes in a Distributed Memory / Message Passing model. These subarrays would be put on the first queue where slave nodes will be able to consume the unsorted arrays, and perform our forkjoin mergesort algorithm on them. When the subarray within the slave node is sorted, it is put on a second queue, on which the master node is waiting for the amount of chopped arrays it's put on the first queue. When all expected arrays are received by the master node, it performs a last forkjoin mergesort to finalize the sort.

We started the slave programs to prepare for consumption and production of ordered arrays before we started the master program producing the unordered arrays in order to run the sort in the fastest possible way.

Below is a visualized explanation of our approach, Image 4.1.1:

The unordered integer array is chopped up in n pieces, where n equals the amount of available slave nodes.

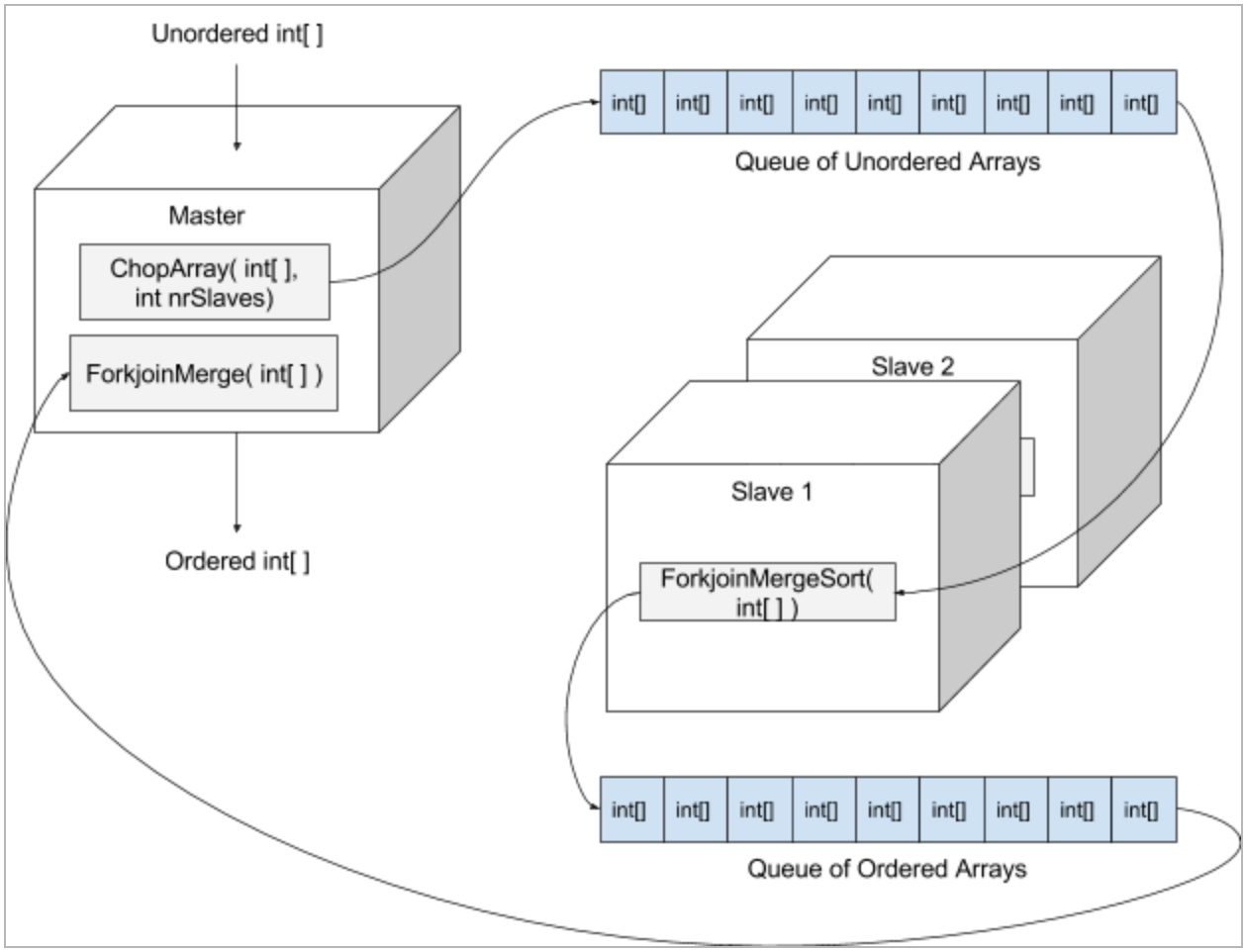

Image 4.1.1 - Solution Visualization

4.2. Benchmark

We've tested the algorithm by using array sizes of 500.000 to 48.000.000, running on one node, and multiple nodes.

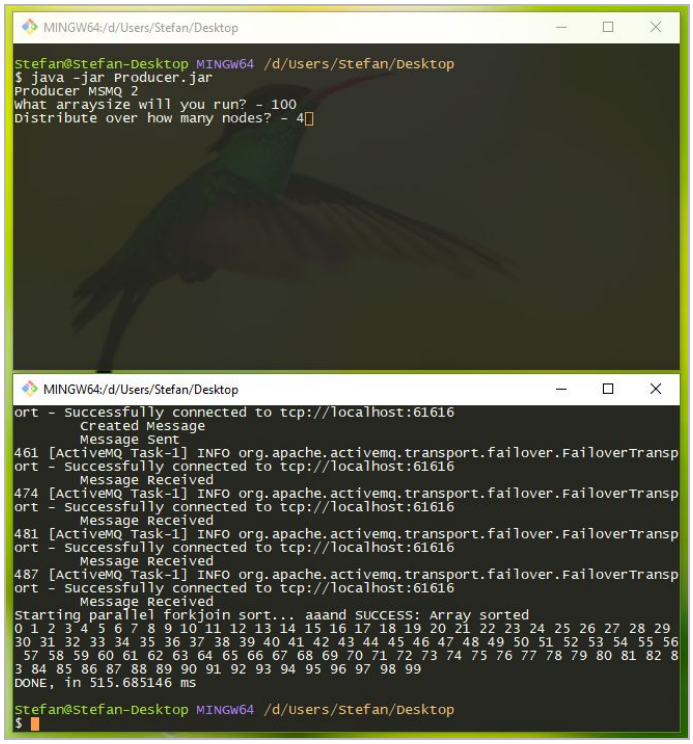

Image 4.2.1 - Running Benchmark

We've split the array into sixteen subarrays in each test for consistency, not for performance gains. Ideally, the array would be split into one subarray for each node. But when the array size increases, the messages fail to send.

Array size	Time (ms)
500.000	22954.865390 ms
1.000.000	18581.046872 ms
2.000.000	23083.524829 ms
4.000.000	31196.848633 ms
8.000.000	51581.728176 ms
16.000.000	98470.653748 ms
24.000.000	*OutOfMemory*
48.000.000	*OutOfMemory*

Table 4.2.1 - Single Slave Node

Array size	Time (ms)
500.000	20957.829691 ms
1.000.000	26084.095928 ms
2.000.000	25763.540967 ms
4.000.000	32987.796774 ms
8.000.000	48402.942260 ms
16.000.000	79507.338606 ms
24.000.000	*OutOfMemory*
48.000.000	*OutOfMemory*

Table 4.2.2 - Four Slave Nodes

Additionally, at some point the program throws an OutOfMemoryException, probably because it has to store large strings before converting them all to arrays. That's why we've only been able to run tests up until the size of 16 million integers.

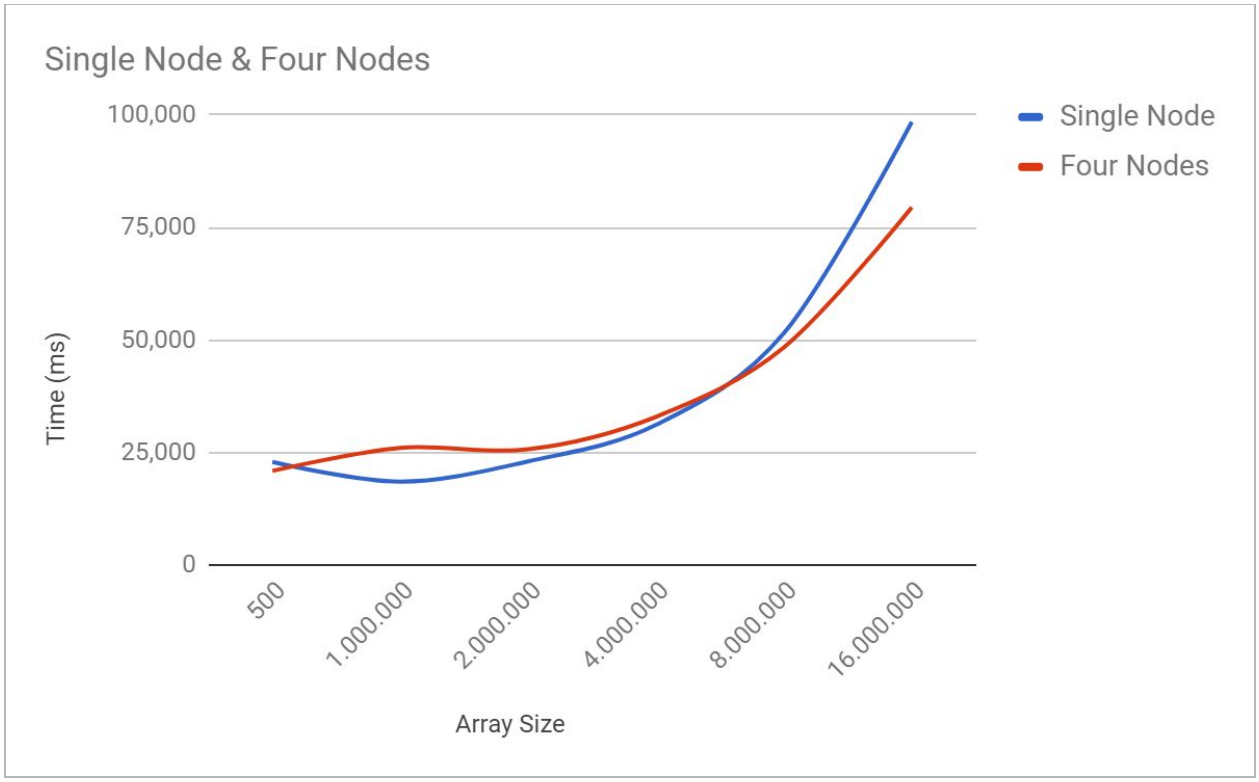

Image 4.2.1 - Benchmark Single & Multiple Nodes

Each raspberry pi has one quad-core processor. Each consumer runs our forkjoin implementation of mergesort, thus making use of multiple cores. In the best case scenario, 16 cores will be utilized in parallel, to their maximum potential.

It is worth mentioning that regardless of the difference in speed between nodes, the master will wait for all results are received before continuing processing the final result. Therefore it's advised, using our implementation, to produce more subarrays so that whenever a node is finished, it can quickly pick up a new task from the queue.

The graph shows again a small increase in speed with larger arrays. This time, the overhead is even greater because data has to be converted, sent, received and converted back before any work can be done. Then after the work has been done, the result has to be sent back again.

Because our MSMQ implementation closely resembles our normal parallel solution, it's absolutely clear that, even with four nodes having four cores, the problem takes a lot

longer to be solved because of the overhead introduced through networking and conversion of the data that needs to be transmitted.

When we imagine an array of infinite length, the problem could be solved with an infinite amount of nodes if the last step in merging all results could be possible by storing such large amounts of data. In any other implementation, the limited amount of cores available in one system would make it impossible to solve. Therefore, this implementation, regardless of the bad performance, is the most scalable implementation in theory.

5. Conclusion

Now that we've benchmarked four implementations of mergesort, on multiple architectures in regards to Flynn's taxonomy, on different types of processors, under different circumstances, it's impossible to just throw all benchmarks together in one graph and draw a valuable conclusion. Although it may provide some insights in the sheer time differences of performing sorting tasks with different implementations in order to describe the best implementation from our collection.

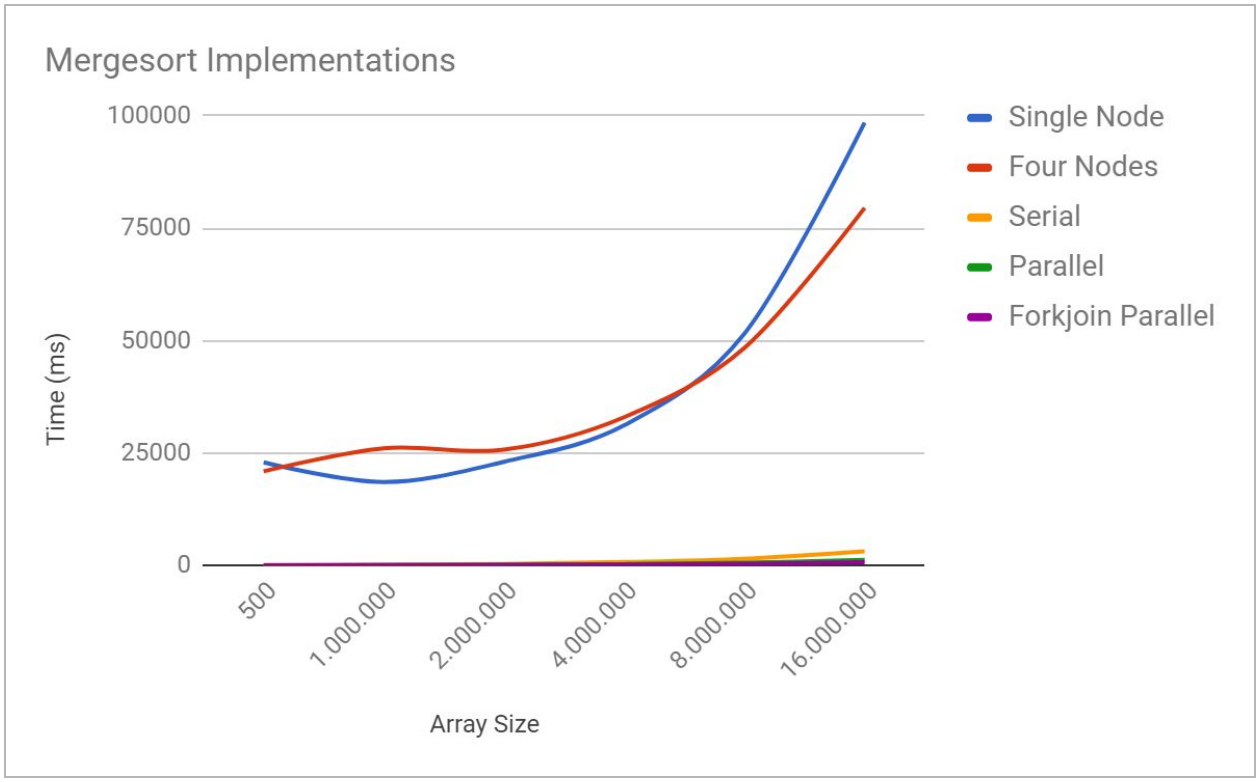

Image 5.1 - Benchmark All Implementations

On all array sizes, it's evident that the forkjoin implementation outperforms all other implementations.

Now the speedup can be computed by Amdahl's Law, by determining the percentage of parallelized code. The serial solution has 100% parallelizable code, and can be referred to as B, because every single step in the mergesort algorithm could in theory be executed in parallel recursively. But as the parallel solution is not twice as fast as the serial solution, there must be some overhead which can be referred to as B.

The forkjoin implementation has every single step parallelized, except for the first call to Mergesort, and the last merge step, which indicates that depending on the problem size, the amount of parallelized code can be proportionally expressed in milliseconds. Assuming the array size of 8 million, the times can be visualized as done in Image 5.2, inspired by the slides provided by the HvA, ("HBO-ICT - VLO 2016-2017 - Themasemester: Software Engineering," n.d.):

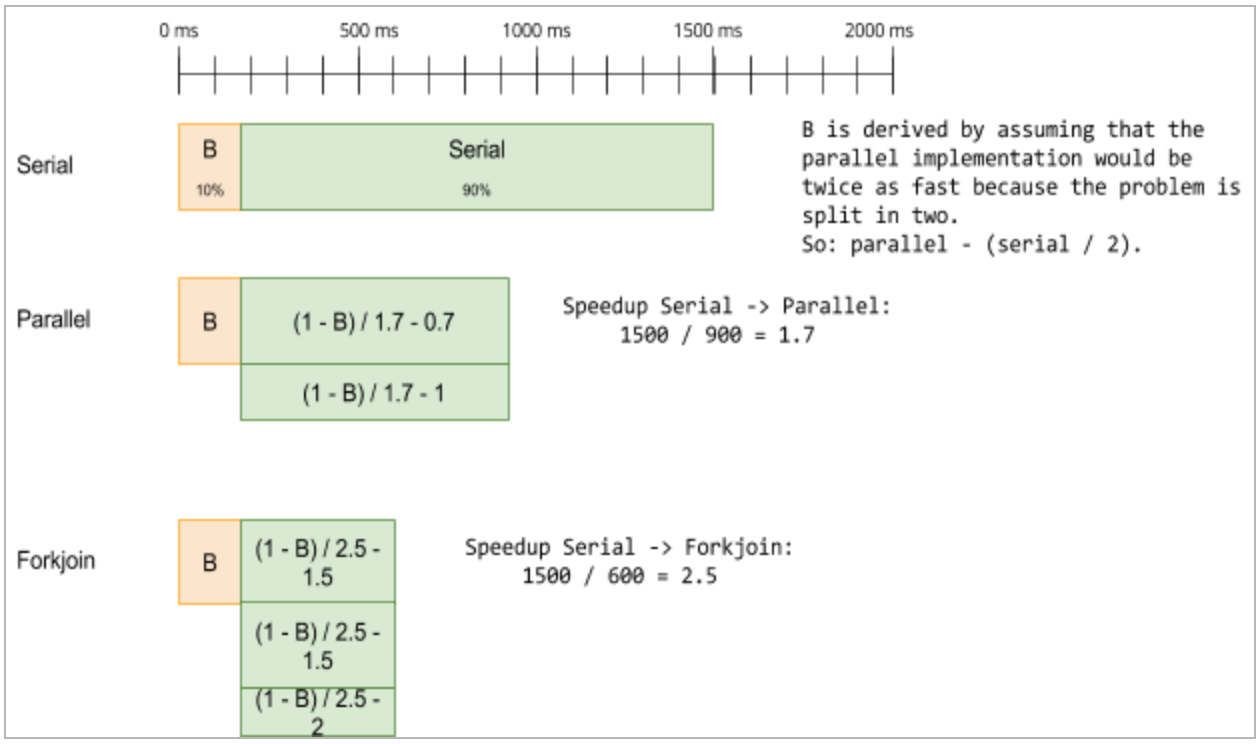

Image 5.1 - Amdahl's Law Visualized

According to Amdahl's Law, a 90% parallelization would yield a 10 times speedup, which in our case doesn't align.

As stated in the previous chapter, the n-Nodes implementation will be the most scalable implementation if the last merge step would not be executed on one system. However, our solution will currently have the same or more limitations than the parallel implementations. The best scalable solution in regards to speed, task granularity, node failure and network speeds will obviously be the forkjoin implementation, because it won't have to deal with network or node instabilities. It has the finest task granularity with adjustable thresholds to prevent wasting resources. As demonstrated in our MSMQ solution, it can be used as the main sorting implementation on a distributed system, making it versatile, performant, ultimately concurrent, dynamic and easily reusable.

Looking back at our case study, we've learned that implementing parallel implementations of a problem requires a totally different way of thinking, but pays off in the end. We've also learned that distributing a problem over multiple systems is even harder, could pay off when scalability is required, but totally depends on the nature of the problem.

Lets discuss the implementation in class

6. Bibliography

From Concurrent to Parallel. (2016). Retrieved from https://www.youtube.com/watch?v=NsDE7E8sIdQ

GitLab. (n.d.). Retrieved June 8, 2017, from https://gitlab.dmci.hva.nl/srink/2016_2017_block4_pc_ivse4/commits/schenk82

HBO-ICT - VLO 2016-2017 - Themasemester: Software Engineering. (n.d.). Retrieved June 9, 2017, from https://vlo.informatica.hva.nl/main/document/showinframes.php?cidReq=BUSINESSUNITSOFTWAREENGINEERING&id_session=0&gidReq=0&gradebook=0&origin=&id=21115

Lars Vogel (c) 2009, 2016 Vogella Gmbh. (n.d.). Mergesort in Java - Tutorial. Retrieved May 9, 2017, from http://www.vogella.com/tutorials/JavaAlgorithmsMergesort/article.html