Algorithm Engineering

Carl-Johannes Johnsen (grc431) Martin Philip Poulsen (jfw772)
Daniel Lundberg Pedersen (vzb687)

Department of Computer Science, University of Copenhagen
Universistetsparken 5, DK-2100 Copenhagen East, Denmark
{grc431,jfw772,vzb687}@alumni.ku.dk

1. Introduction

Sorting is an essential problem in computer science, due to the historical background and the many places it is used as a subroutine. Sorting comes in many variations, methods can be in-place, stable, external, adaptive and so on. Similarly one can analyse many aspects of sorting algorithms, such as number of element comparisons, runtime and number of element swaps.

Typically a hybrid method is used for sorting, in which one switches between different methods dependent on for example the number of elements, in order to alleviate some of the known drawbacks of the methods. An examples of this would be introsort[8]. Usually these methods fall back to a quadratic method when they are nearly sorted. Another approach could be to fallback to a less general method when reaching smaller subsizes. With this approach the fallback method is called more than once and therefore any practical improvement in the runtime could yield a significant speedup. With this as motivation we look into alternatives for fallback methods, focusing on comparison (near) optimal algorithms.

The next section investigates the known sorting networks for small sets. It covers nearly comparison optimal sorting networks for 2 to 16 elements. They are considered for branch optimization, in order to speed up the networks.

The third section covers a classical method for nearly comparison optimal sorting of small sets. It is then optimized with respect to branches and branch misspredictions, in order to optimize the runtime.

The fourth section makes some comparisons between the considered fallback algorithms, and `std::sort()`

Finally the fifth section looks into the well-known Quicksort and Mergesort. It is then investigated if some of these methods can be speed up by proper use of allocation, memory and by manipulating access patterns. Following this the methods are parellelized. And it looks at what difference the use of a good fallback method does.

2016-10-30

2. Sorting networks

Sorting networks, are networks with some number of inputs, that then using connections for comparison and swaps, sort the inputs. They are usually depicted as seen in Figure 1, where the horizontal lines are the inputs, and the vertical lines are the comparators, the comparators compare the two inputs that it is connected to, and then swap the values if the value for the lower input is lower than the value for the upper input.

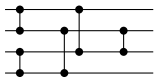


Figure 1: 4-input

2.1 Implementation

We have implemented sorting networks for inputs from 2 up to 16 in `nsort.h`, with the ones for inputs 2 to 9 being optimal [6][9] in length (number of comparators). The rest, inputs 10 to 16 was the best we could find with respect to length [2][9]. Also to have a nice drawing of them all they are presented in Appendix A.

2.2 Bitonic mergesort

Other than static sorting networks there also exists algorithms to generate sorting networks, one such algorithm is Ken Batchers Bitonic mergesort. We implemented the traditional version of the Bitonic sorter, which only works on input sizes of $n = 2^k$ for some k, though it is possible to extend it to accept arbitrary n inputs [3]. The implementation is present in `bitonic.h`, is non-recursive, uses the alternative representation thus only sorts in one direction, and has been tested with random input of different $n = 2^k$ lengths inputs.

2.3 Branch optimization

Since practically all a sorting network does is comparing values and swapping if needed, and ignoring the network is oblivious to previous branches, it lends itself to use the `cmov` instruction that is available in todays x86 CPUs [7]. To test the viability of this optimization we created `nsort.c` which is simply the 16 input sorting network and a main function to run the sorter on a specified number of size 16 arrays of random values, and finally print the total time in microseconds it took to sort them all, with the possibility to change the comparator macro (in code), between the one that should generate `cmov`s and the one that should generate jumps. The time difference, seen in Figure 2 makes it quite clear that the branch optimization is non-negligible. And

Type	Time (μs)
Branch optimized	5 854
Regular	18 433

Figure 2: 16-input sorting network over 10^5 size 16 arrays. Machine see Figure 5

Type	Branches	Mispredicts	Rate(%)
Branch optimized	19,334,579	155,776	0.8%
Regular	25 334 579	2,481,889	9.8%

Figure 3: 16-input sorting network over 10^5 size 16 arrays. Machine see Figure 5

looking at brach misprediction with `valgrind` and `cachegrind` in Figure 3 we see why.

However during testing a peculiarity of the `g++` optimizer came forth, if we compile `nsort.c` with `g++` instead of `gcc` but still the same flags then it will not optimize the comparators into `cmov` instructions. Quick to see by running `make nsort-c; make nsort-cpp` and then run `objdump -M intel -d build/nsortbo | grep cmov | wc -l` we get an output of `120` whichs seem reasonable enough but `objdump -M intel -d build/nsort | grep cmov | wc -l` gives an output of `0`. All this would make it seem that `g++` simply does not have the optimization that `gcc` has, but after implementing Bitonic mergesort we of course wanted to see what improvements the same branch optimization would have on the implementation, and quite unexpectedly we found that in this case `g++` would employ the `cmov` instruction as is evident when we run (Make sure the right macro is uncommented) `make bitonic` and afterwards `objdump -M intel -d build/bitonic | grep cmov | wc -l` and get the output `29`.

Finally Bitonic also lends itself to some parallelisation where we used *OpenMP* to parallelise the second loop of the implementation, there are more loops that can be executed in parallel, but from some testing they were not worth the cost.

2.4 Testing

Since it can be quite hard to see from the code if a specific sorting networks should work, testing them is quite important. So for all the implemented sorting networks for inputs > 4 there is a test for a reverse sorted list, and for a number of runs on random inputs. Futhermore there is the possibility to run a full permutation test, that is testing the sorting networks for all possible permutations of inputs of that size, though this should be possible to make in $O(2^n)$ instead of the current $O(n!)$ due to the Zero-one principle.

N	`std::sort`	Bitonic BO	Bitonic parallel	Bitonic
2^7	1	7	18	10
2^8	2	16	28	24
2^9	5	36	73	55
2^{10}	11	82	120	125
2^{11}	25	181	185	281
2^{12}	55	400	382	624
2^{13}	120	878	769	1386
2^{14}	256	1916	1790	3043
2^{15}	553	4160	3476	6625

Figure 4: Comparison of `std::sort` to the different versions of Bitonic mergesort. Time is in μs, and is the averege time to sort N elements over 10^4 runs. Machine see Figure 5

This permutation test has been run on all the implemented sorting networks up to and including `nsort_13()`

2.5 Performance

In Figure 4 we have compared the three versions of Bitonic mergesort to `std::sort()`. We clearly see that `std::sort()` wins by quite a bit, also unsurprisingly the branch optimized Bitonic is better than the regular one, and finally that the parallel version is worse upto around 2^{11} elements after which it begins to outperform the non-parallel version.

3. Ford Johnson

When considering element comparison in sorting smaller sets of numbers the algorithm by Ford & Johnson is a classical example[4]. It is close to optimal in the number of element comparisons it needs to sort smaller sets of numbers [5]. We have implemented an in-place version of the Ford & Johnson algorithm following the outline by Ford & Johnson[4].

3.1 Branch optimization

The Ford & Johnson may be close to optimal in number of element comparisons made, but it relies on binary search among other sub procedures. These along with the main method uses branching to choose between two ways through the code. This will make it difficult for the compiler to pipeline the instructions in a way that optimizes the runtime, due to the potential flushes of misspredicting the branches.

Motivated by this we have further implemented a version of Ford & Johnson where the branches has been optimized out. This has generally been done by utilizing the boolean values in the mathematical statements, used

OS:	Linux 64-bit
Version:	4.8.4-1-ARCH
Processor:	Intel i5 2.67 GHz
RAM:	12 GB
Compiler:	GNU, G++, 6.2.1

Figure 5: Benchmark machine 1

throughout the code, in order to capture the branching in statements that can be better anticipated by the compiler.

4. Empirical investigation

The sorting networks have been collected in a method that selects a prober sized network for the input. These methods has been tested on an environment with the specification depicted in figur 5

The methods have been averaged over a set of runs on a variation of smaller set sizes. The result is depicted in figur 6. It is clear from the results that the branch optimization of Ford & Johnson does not make a significant speedup, in fact it seems to slow it down. This can be due to a lot of reasons, but most likely stems from the fact that the few branches in the implementation does not contribute that much to the runtime and changing it to far bigger mathematical statements does contribute quite a lot to the runtime. This gives an indication that one of the parts taking up time is index calculations. It is further evident that the Ford & Johnson algorithm may be close to comparison optimal, but in practice the needed support for that is too slow to compete with other methods.

Compared to the standard library sorting method, it is clear that the specialized sorting networks achieve a better runtime. It seems to cut almost a third of most of the times. The standard library of the gnu gcc compiler uses a hybrid between introsort, itself being a hybrid, and insertion sort[1]. This method is slightly slower than then the networks, which could be due to overhead. Lastly we can see from the table that the optimal comparison sort achieves the same runtime as the sorting networks. This indicates that the networks achieves there potential in this particular setting.

5. Parallel

In this section, we will be looking at parallelising sorting algorithms. We will be looking at sorting elements residing within the `vector` data structure. Since we are trying to shave off time, we start by looking into different approaches to memory management, allocation and access. From there we will be implementing sequential versions of Quicksort and Mergesort. Then

N	`std::sort`	`nsort()`	`fsort()`	Ford Johnson	FJ branch optimized
2	12	9	N/A	110	122
4	32	21	21	247	319
6	57	38	37	501	673
8	89	60	60	603	683
10	119	85	N/A	895	1291
12	154	115	N/A	1018	1511
14	192	147	N/A	1375	2122
16	233	148	N/A	1609	2398

Figure 6: Comparison of small sets specialized sortes and `std::sort`. `nsort()` is the sorting networks without branch optimization (due to `g++` not outputing `cmov`s). `fsort()` is sorters with optimal number of comparisons for N elements. Time is in nanoseconds to sort all N elements averaged over 10^5 runs. Machine see Figure 5

we will be parallelising these algorithms, and finally combine a combination of both.

5.1 Approaches to pivot choosing, memory allocation, management and access

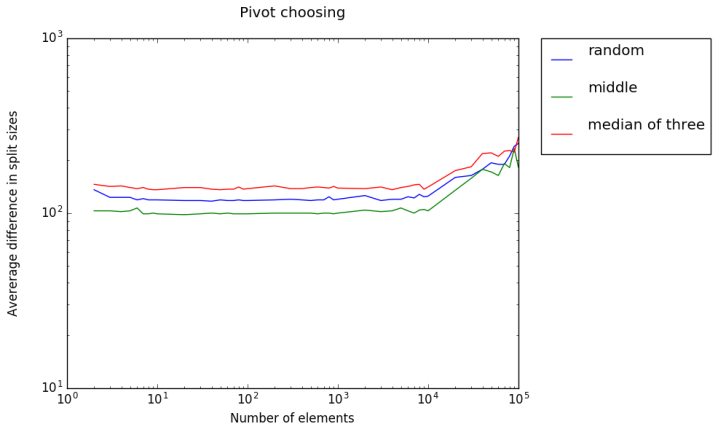
Here we will be looking at choosing a good pivot, inserting elements into a new empty `vector`, moving data from one `vector` to another, moving data from one `vector` into an empty one and extracting a subset `vector` from a `vector`.

5.1.1 Choosing the right pivot

We have three approaches: choosing a random element, choosing the middle element and choosing the median of three random elements. The goal is to choose a pivot element, which will as much as possible split the input into equal subsets.

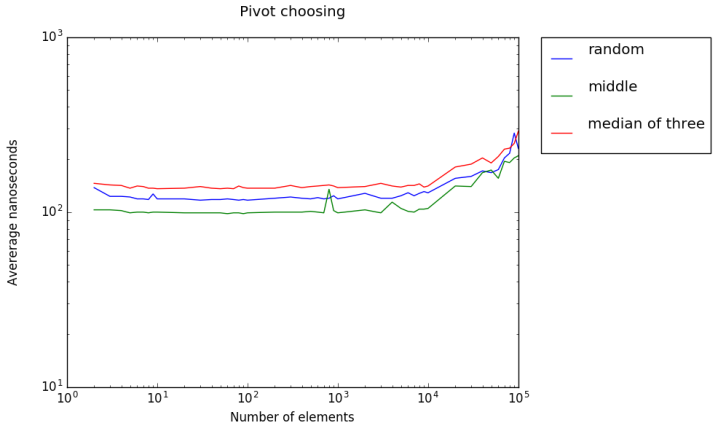
It has been showed that choosing a random element is usually good in practice. However, getting a random number can introduce some overhead, which is why we also tried the middle element approach. We could also have chosen to just selecting the first or the last element, however, this would be bad in the case of sorted or reversed sorted input. Finally, we tried the median of three approach, as the chances of choosing three elements, which are all bad, is less than the probability for making a single bad choice.

First, we look at how good the splits are:



Interestingly enough, choosing the middle element as pivot works suprisingly well on random input.

Then, we look at the perfomance of the three:

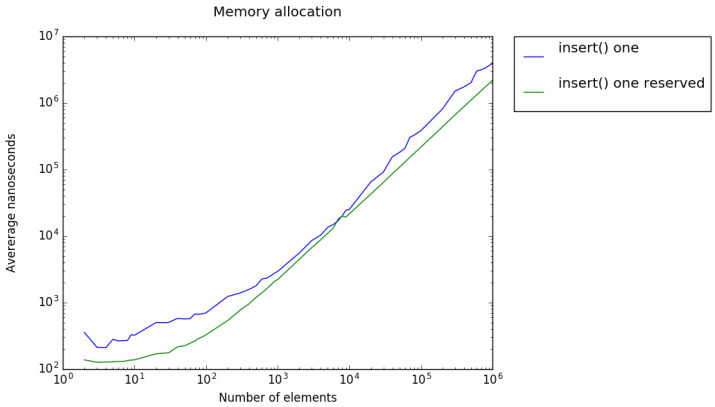


Here, we see that the median of three performs the worst. This was expected, as it requires more work. Coming in second is the random element, as the beforementioned overhead from getting random numbers is present. Finally, we have the best performance from choosing the middle element, which again is expected as it has the least work to do.

5.1.2 Memory allocation

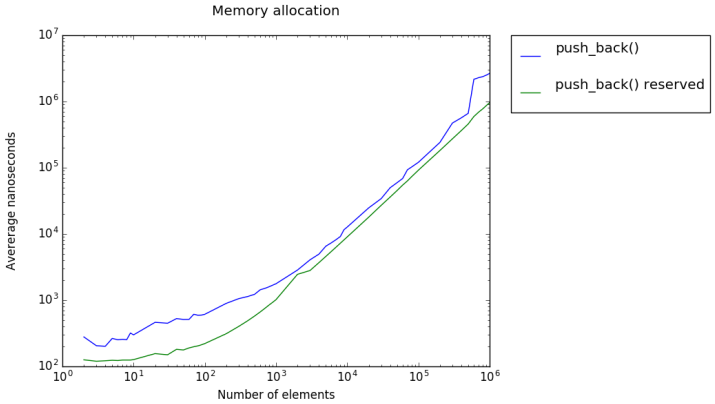
Within the different algorithms, we sometimes want to insert one element at a time into a `vector`. We have two approaches: calling `vector.insert()` one element at a time, and calling `vector.push_back()`. For each we look at the performance with and without calling `vector.reserve()`.

We have `vector.insert()`:



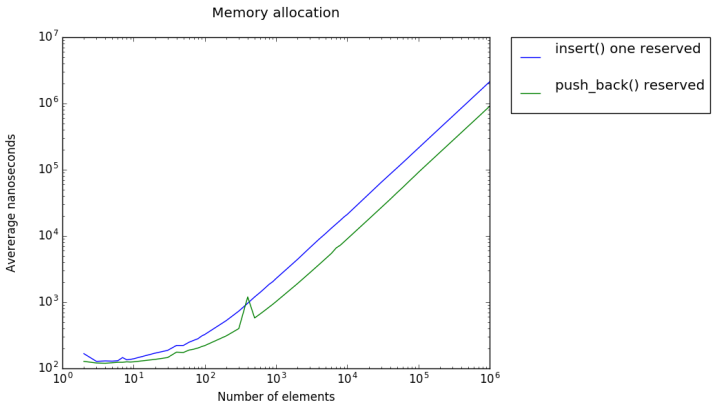
Here we see that calling `vector.reserve()` prior to the operations benefit the performance.

Then, we have `vector.push_back()`:



Again, we see that calling `vector.reserve()` prior to the operations benefit the performance.

To make sure, we have the graph containing both:

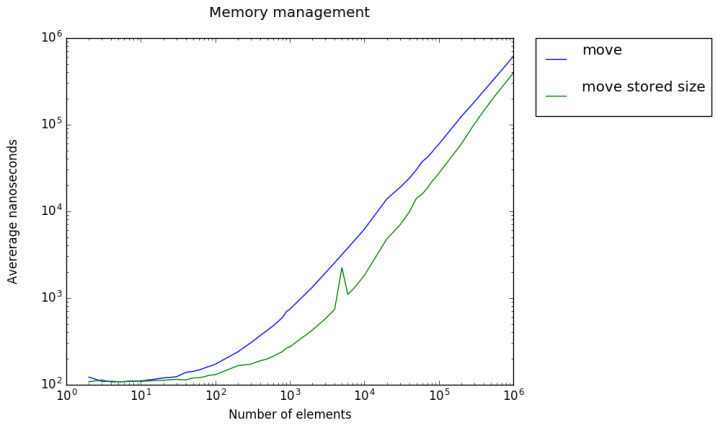


From this graph, we see that calling `vector.push_back()` gives the best performance.

5.1.3 Memory management and access

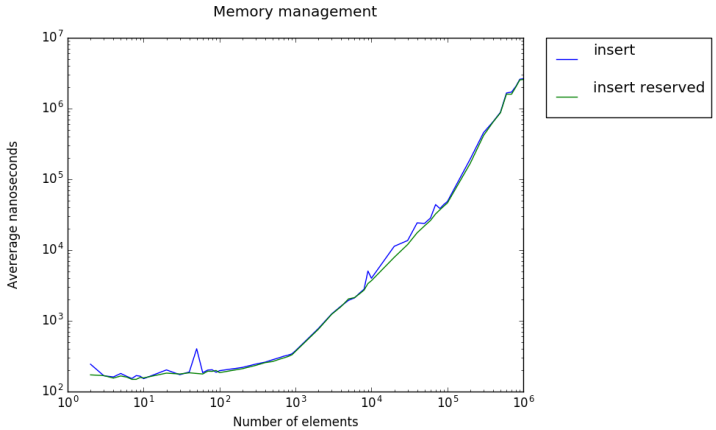
When we want to move data from one `vector` to another, not empty, `vector`. In these tests, we move a copy of a `vector` into the original `vector`. We have following two approaches: manual move using a for loop and using `vector.insert()`. For the manual move, we have where we call `vector.size()` in the loop condition, and where we store it in a variable before the loop. For insert, we have where we do and do not call `vector.reserve()`.

Manual move:



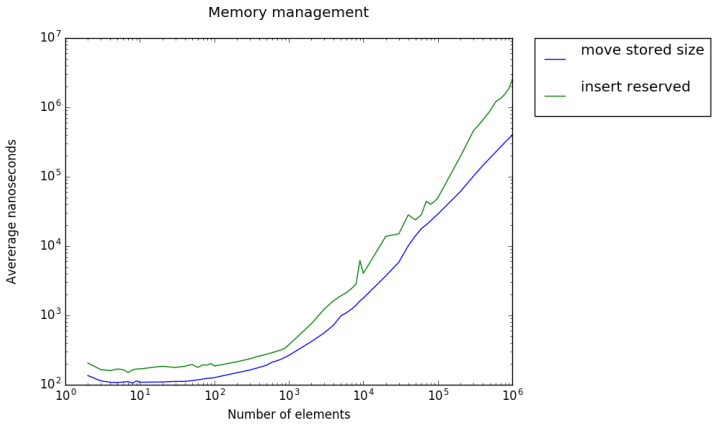
Here, we see that moving the call to `vector.size()` clearly improves the performance.

`vector.insert()`:



strangely enough, even though we are moving a copy, calling `vector.reserve()` beforehand improves performance.

Finally, we compare the two best:

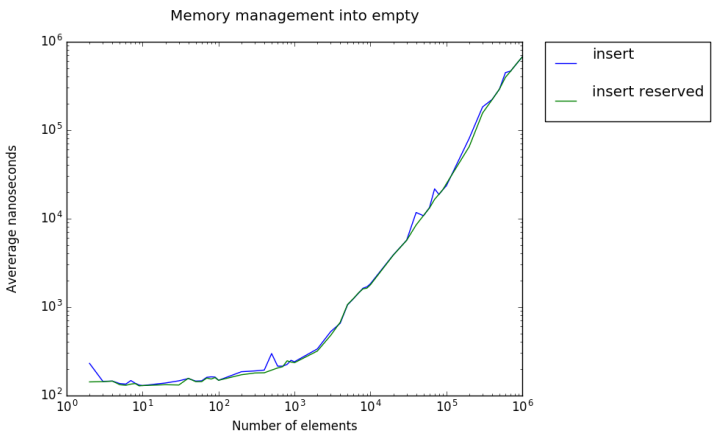


Here, we see that the manual move clearly beats the call to `vector.insert()`

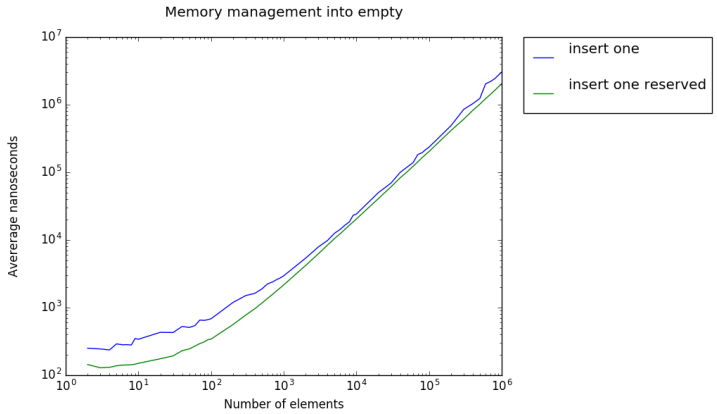
5.1.4 Memory management into empty `vector`

We have this test, to see the performance when copying a `vector` into another empty `vector`. We have the following four approaches: using `vecter.insert()`, using `vector.insert()` one element at a time, using `vector.push_back()` and using the constructor of `vector`. Each of the first three are tested with and without using `vector.reserve()` prior to the operations.

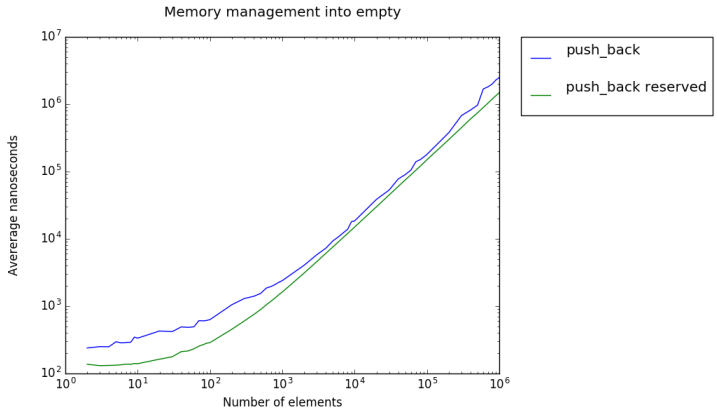
We start with `vector.insert()`:



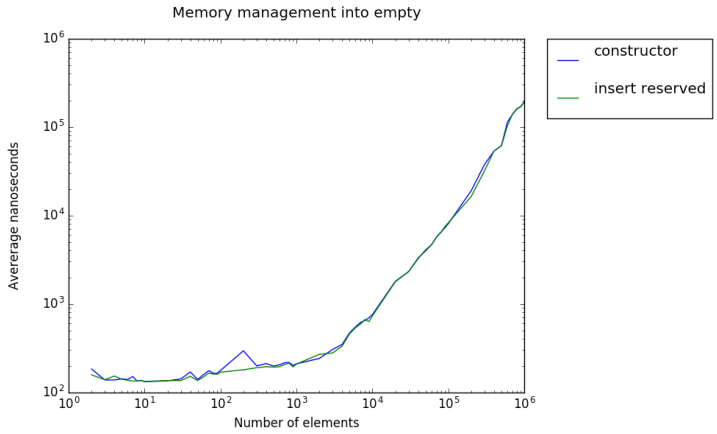
`vector.insert()` one element at the time:



`vector.push_back()`:



Now, we see that the reserved `vector.insert()` has the highest performance, so we will be using that for comparison to the constructor:



We see that in a few cases, calling `vector.insert()` beats using the constructor. However, using the cunstructor makes the code simpler, in the sense that it only uses 1 line of code, compared to the 3 lines of code when

using `vector.insert()`.

5.2 Quicksort

Quicksort is an algorithm, which is known to be good in practice. The performance of the algorithm doues depend on the quality of the split, i.e. if the splits are balanced. We have made different versions of Quicksort, and modified them based on observations when running them.

5.2.1 Sequential

To start off, we implemented an in-place Quicksort, which we would then parallize. To pick the pivot element, we just pick one at random.

5.2.2 Naive parallel implementation

The naive parallization, is to for each split, spawn a thread sorting each split. This however, has a far too big overhead, which results in error on larger input, as too many threads are spawned. On the smaller input, the overhead from spawning threads is far greater than the gains from parallizing, resulting in very bad performance, and as such we have not shown this approach in our graphs.

5.2.3 Limited parallel implementation

Following the naive implementation, we tried to make the same approach, but only spawn a limited number of threads. This did give a speedup, on larger input. However, the performance of this approach depends a lot on the splits. E.g. if we get a bad split in the beginning, i.e. one small portion and one big portion, the thread getting the smaller portion finishes faster, and does not do any work afterwards.

We also looked into if the bad performance could be due to the fact that the threads share the memory. As such, we tried to instead of sending along a pointer to the shared memory, we sent its own allocated `vector`, keeping the memory seperated. However, this made the performance worse. The overhead introduced when moving memory into new memory is greater than the gains. It does seem like since each thread works on its seperate part of the shared memory, there are no problems. Furthermore, having seperate memory doubles the memory used for every split, meaning we will have in the worst case $O(n \ln n)$ memory usage instead of $O(n)$.

5.2.4 Modified limited parallel implementation

Following the idle threads from the previous implementation, we thought of another way. We have a global integer variable, which denotes the maximum number of threads. Each time a thread is spawned, the integer is decremented. Every time a thread is done, the integer is incremented. As

such, if a thread is done with its part earlier than others, one of the threads with more work, can spawn more threads, thus keeping the processor fed. However, there is a big overhead, as the threads has to lock the variable when they are checking and updating it. This results in a not optimal CPU utilisation.

5.2.5 Thread pool approach

Another approach to keeping the processor at work at all times, are a thread pool approach. In this approach we have a queue and an array of threads. Each thread tries to get a "job", which is two indices, indicating the start and the end of the split to sort.

One problem we faced was when to stop the threads, as we could not use an empty queue, as the queue starts empty, and there are no real way to know that the entire input is sorted. The approach we took was to have a boolean value, indicating whether or not the bottom has been reached. Once it has, and once the queue is empty, threads return.

This approach keeps the processor at a high load. However, the overhead introduced from creating, and synchronizing, i.e. mutex locking and unlocking, is very large, so this approach only works on larger input, and even then it is beaten by the limited parallel approach.

5.3 Mergesort

Mergesort is an algorithm, which is known to have very good performance when having parallization. This is due to the equal splitting all the way through the algorithm, which is always perfectly balanced. The hard part of this algorithm was to make it in-place. We took a different approach: when we merge, we copy one half of the input into a temporary vector, merge the temporary vector and the other half into the original input.

5.3.1 Sequential

We started by making a normal sequential implementation. As this is straightforward.

5.3.2 Limited parallel

We took the same initial approach to parallizing as in the limited parallel Quicksort, i.e. spawing only a limited number of threads. Due to the balance of Mergesort, the processor should be at work all the time. In practice this worked very well.

We tried the same approach to seperate memory as with the parallel Quicksort. Just as before, the overhead from managing memory outshines the gains from having seperate memory.

OS:	Linux 64-bit
Version:	3.10.0-327 Red Hat
Processor:	2 × Intel Xeon E5-2650 2.60 GHz
RAM:	126 GB
Compiler:	GNU, G++, 4.8.5

Figure 7: Benchmark machine 2

5.4 Combination of Quicksort and Mergesort

As we can see in the sequential performance, Quicksort beats Mergesort. However, when we look at the parallel performance, Mergesort beats Quicksort. As such, we thought that a combination of the two would give the best performance.

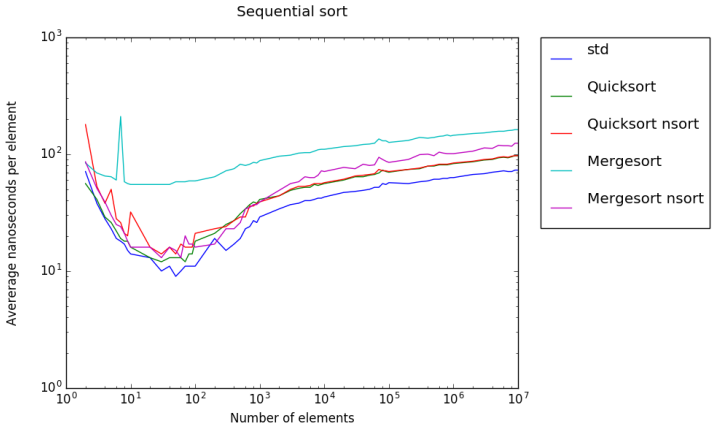
We start by using Mergesort on the top level, splitting the data in half, and spawing a thread for each split. We keep doing this until we run out of processor threads. When we do, we call the sequential Quicksort on the input, as this should be faster when running sequential.

Compared to the other parallel implementations, our intuition worked, as soon as the input gets large enough.

5.5 Benchmarking the different sorting methods

We have run the parallel benchmarks in the environment in figure 7

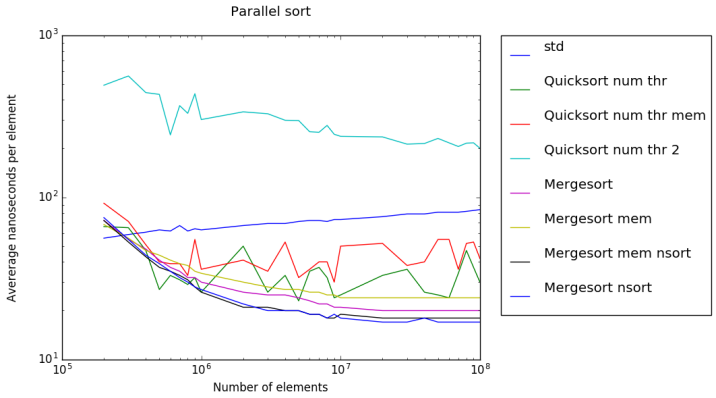
We start by looking at the sequential performance:



Here, we see that `std` wins among the sequential implementations. Among the ones we have made, Quicksort is the fastest. However, we also see that the normal Quicksort beats the Quicksort with nsort. This is probably due to memory management, as the nsort need a `vector` and sorts the initial elements, so we need to extract a sub `vector` to make use of nsort.

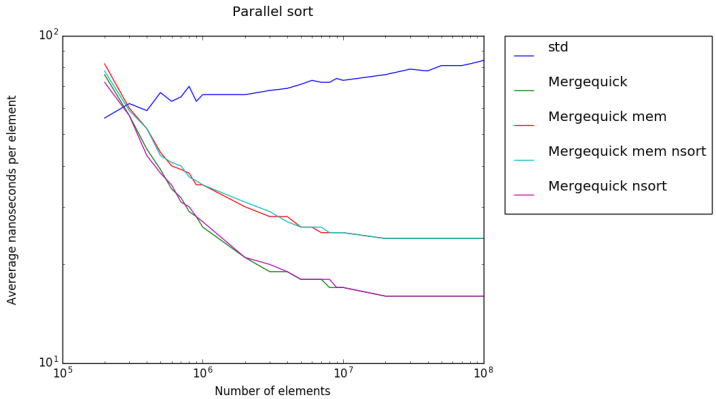
Then we look at the parallel implementations. Note that the naive and the pooled implementations are not present, due to too bad performance.

Also note that we start at a much higher number of elements, as the initial numbers are way too high.



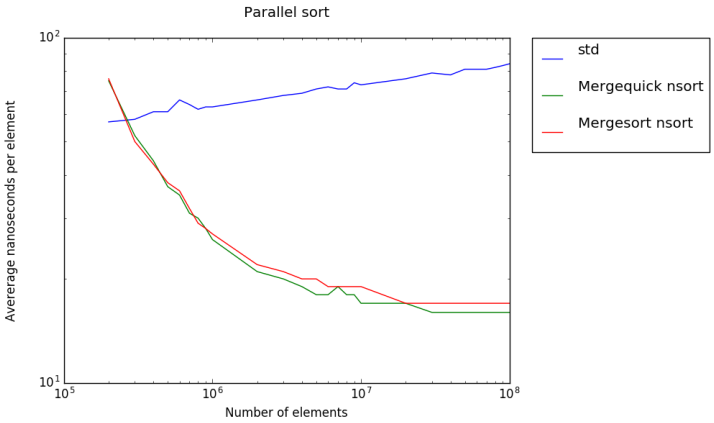
Here, we can see the beforementioned problem when having a global variable that needs to be locked in Quicksort num thr 2. In all the other cases, we can see the speedup of using parallel implementations. We can also see that we do not benefit from using seperate memory. Among these paralell, we can see that the fastest is the parallel Mergesort using nsort.

Then, we also have the combination of Quicksort and Mergesort:



Again, we do not see any benefit from using seperate memory, and again the winner is the one using nsort.

As a final comparison, we look at Parallel Mergesort, the combination of Mergesort and Quicksort, and `std` sort:



We see that the winner is the combination. In the best case, the combination of Mergesort and Quicksort, which uses 16 ns per element, is 5.25 times better than `std` sort, which uses 84 ns per element. It could be the case that the performance gains could be even better on larger input, however this would require more time for testing.

References

[1] , Sorting algorithms, Worldwide Web Document. Available at `https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01027.html` Accessed: 30-10-2016

[2] H. Juille, Sorting networks and the end search algorithm, Worldwide Web Document (2002). Available at `http://www.cs.brandeis.edu/~hugues/sorting_networks.html`.

[3] H. W. Lang, Bitonic sorting network for n not a power of 2, Worldwide Web Document (1998-2016). Available at `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm`.

[4] J. Lester R. Ford and S. M. Johnson, A tournament problem, *The American Mathematical Monthly* **66**, 5 (1959), 387–389.

[5] G. K. Manacher, The ford-johnson sorting algorithm is not optimal, *Journal of the Association for Computing Machinery* **26**, 3 (1977).

[6] M. F. Michal Codish, Luis Cruz-Filipe and P. Scheider-Kamp, Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten), *arXiv:1405.5754* (2014).

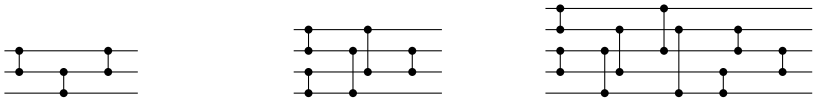
[7] M. N. Michal Codish, Luis Cruz-Filipe and P. Scheider-Kamp, Applying sorting networks to synthesize optimized sorting libraries, *arXiv:1505.01962* (2015).

[8] D. Musser, Introspective sorting and selection algorithms, *Software Practice and Experience* **27** (1997), 983–993.

[9] R. Zeno, A reference of the best-known sorting networks for up to 16 inputs, Worldwide Web Document (2002). Available at `http://www.angelfire.com/blog/ronz/Articles/999SortingNetworksReferen.html`.

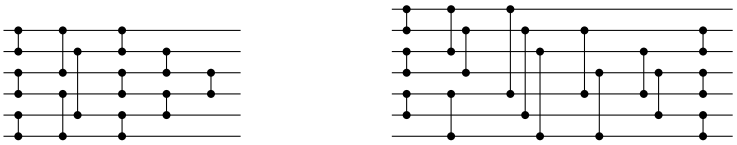
Appendix A. Sorting Networks

This is a compilation of figures of the sorting networks implemented.



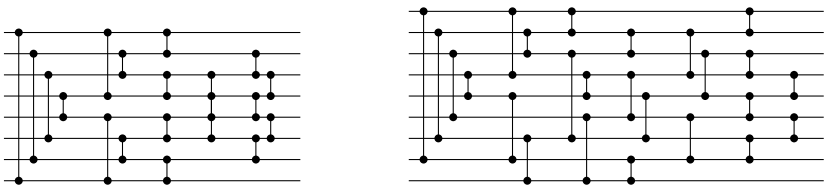
(a) 3-input (b) 4-input (c) 5-input

Figure 8: 3-5 input length optimal sorting networks [9]



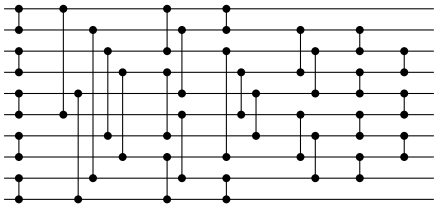
(a) 6-input (b) 7-input

Figure 9: 6 and 7 input length optimal sorting networks [9]

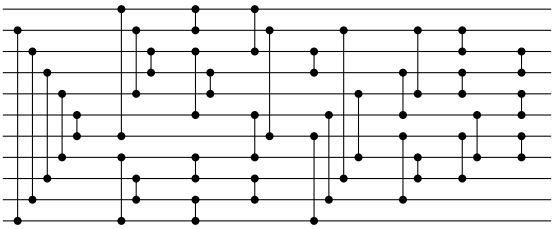


(a) 8-input (b) 9-input

Figure 10: 8 and 9 input length optimal sorting networks [9]



(a) 10-input



(b) 11-input

Figure 11: 10 input, and 11 input best found sorting networks [9]

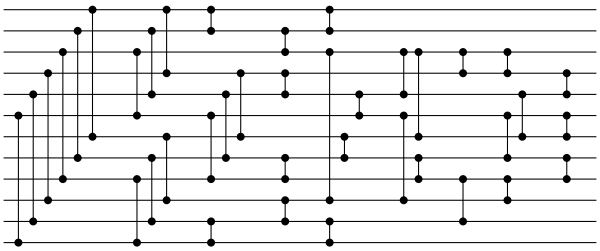


Figure 12: 12 input best found sorting networks [9]

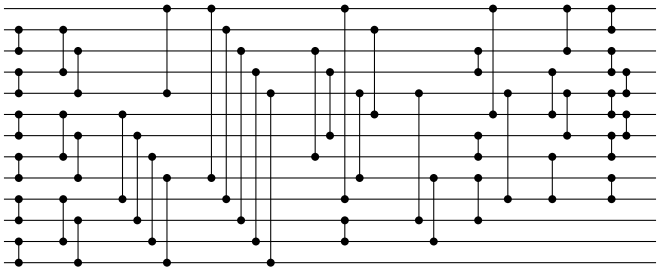


Figure 13: 13 input sorting network, derived from Greens 16 input sorting network [2]

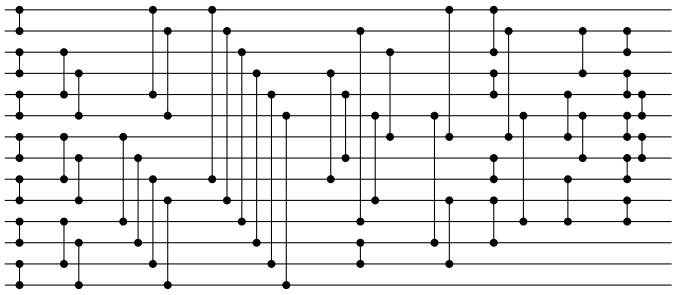


Figure 14: 14 input sorting network, derived from Greens 16 input sorting network [2]

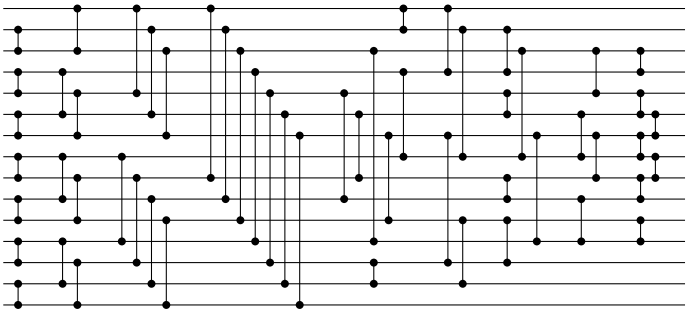


Figure 15: 15 input sorting network, derived from Greens 16 input sorting network [2]

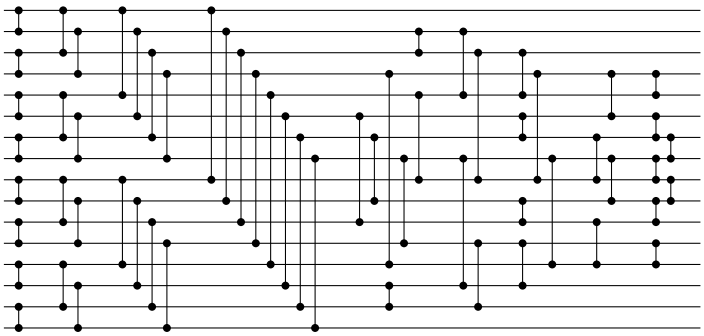


Figure 16: Greens 16 input sorting networks [2]