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

1.1 Approaches to memory allocation, management and access

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

1.1.1 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()`:

aoeu!

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

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

aoeu!

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

To make sure, we have the graph containing both:

aoeu!

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

1.1.2 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 three 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:

aoeu

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

`vector.insert()`:

aoeu

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

Finally, we compare the two best:

aoeu

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

1.2 Quicksort

Quicksort is an algorithm, which is known to be good in practice. The performance of the algorithm does 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.

1.2.1 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.

1.2.2 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.

1.2.3 Limited parallel implementation

Following the naive implementation, we tried to make the same approach, but only spaw 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.

1.2.4 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. The overhead compared to the limited parallel is not very big.

1.2.5 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.

1.3 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.

1.4 Sequential

We started by making a normal sequential implementation. This implementation worked very well, both against Quicksort and `std::sort()`.

1.5 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.

1.6 Quickmerge

We also thought of having Mergesort at the top of the input, and on each split spawn a limited number of threads. This would remove the chance of bad splits in quicksort, as all of the threads receive equal amount of work.

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.

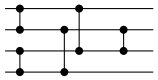


Figure 1: 4-input

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 [5][7] in length (number of comparators). The rest, inputs 10 to 16 was the best we could find with respect to length [1][7]. 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 [2]. 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 [6]. 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.

16-input sorting network over 10^5 size 16 arrays.

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

The time difference makes it quite clear that the branch optimization is non-negligible. And looking at brach misprediction with `valgrind` and `cachegrind` we see why:

16-input sorting network over 10^5 size 16 arrays.

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

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`.

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

Figure 2: Comparison of small sets specialized sortes and std::sort

size	std::sort	Sorting networks	ford johnson	fj branch optimized
2	326	979	825	331
4	321	1366	1282	307
6	331	2816	2532	373
8	359	2737	2662	391
10	363	3707	3856	498
12	424	3870	3861	457
14	438	5199	5199	452
16	470	5675	5647	508

3. Ford Johnson

When considering element comparison in sorting smaller sets of numbers the algorithm by Ford & Johnson is a classical example[3]. It is close to optimal in the number of element comparisons it needs to sort smaller sets of numbers [4]. We have implemented an in-place version of the Ford Johnson algorithm following the outline by Ford & Johnson[3].

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

References

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

[2] 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`.

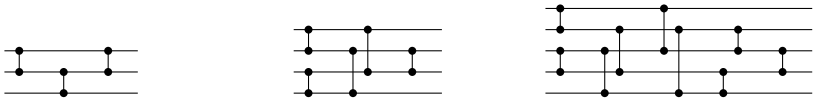
[3] J. Lester R. Ford and S. M. Johnson, A tournament problem, *The American Mathematical Monthly* **66**, 5 (1959), 387–389.

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

[5] 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).
[6] M. N. Michal Codish, Luis Cruz-Filipe and P. Scheider-Kamp, Applying sorting networks to synthesize optimized sorting libraries, *arXiv:1505.01962* (2015).
[7] 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`.

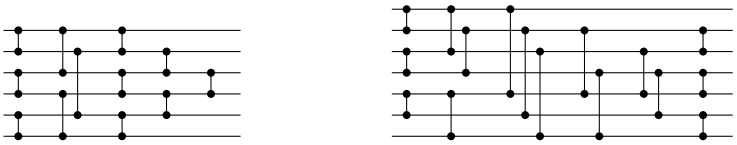
Appendix A. Sorting Networks

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



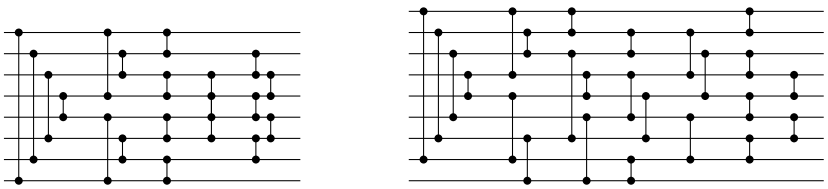
(a) 3-input (b) 4-input (c) 5-input

Figure 3: 3-5 input length optimal sorting networks [7]



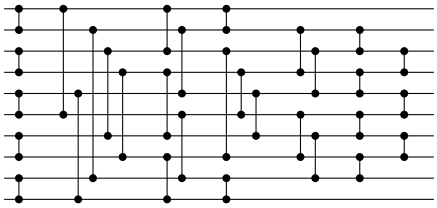
(a) 6-input (b) 7-input

Figure 4: 6 and 7 input length optimal sorting networks [7]

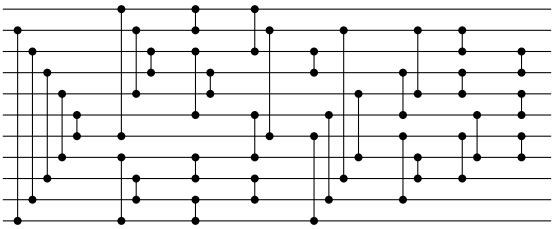


(a) 8-input (b) 9-input

Figure 5: 8 and 9 input length optimal sorting networks [7]



(a) 10-input



(b) 11-input

Figure 6: 10 input, and 11 input best found sorting networks [7]

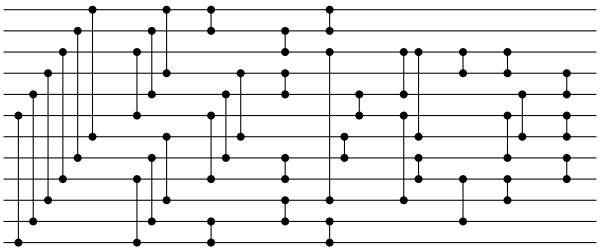


Figure 7: 12 input best found sorting networks [7]

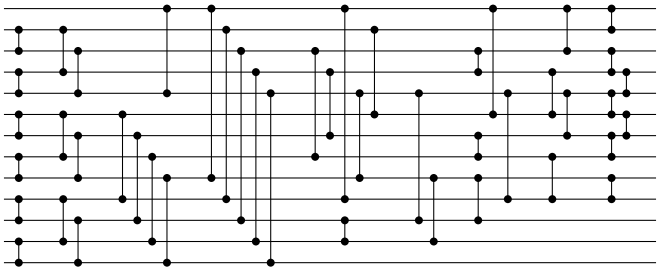


Figure 8: 13 input sorting network, derived from Greens 16 input sorting network [1]

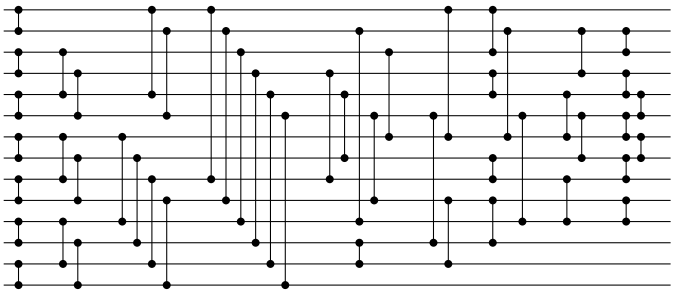


Figure 9: 14 input sorting network, derived from Greens 16 input sorting network [1]

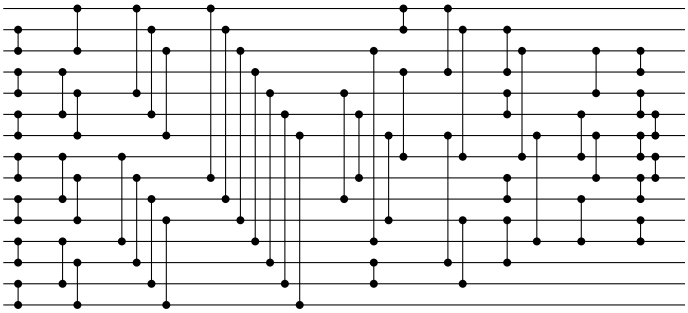


Figure 10: 15 input sorting network, derived from Greens 16 input sorting network [1]

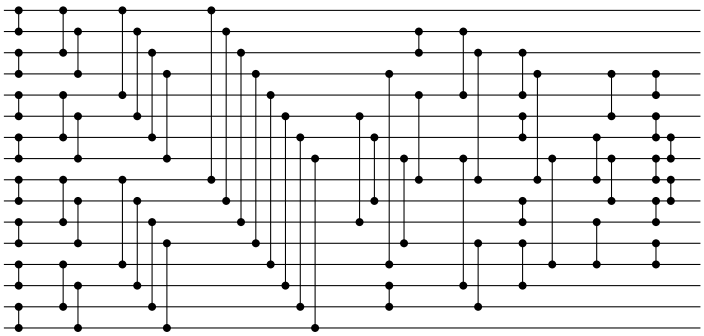


Figure 11: Greens 16 input sorting networks [1]