Algorithm Engineering

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

Department of Computer Science, University of Copenhagen
Universistetsparken 5, DK-2100 Copenhagen East, Denmark
{grc431,mds111,vzb687}@alumni.ku.dk

1. Parallel

In this section, we look at implemnting parallel versions of known sorting algorithms, and compare them with different sequential versions.

1.1 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.

1.1.1 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.

1.1.2 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.

1.1.3 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.

1.1.4 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.

1.1.5 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.

1.2 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.

1.3 Sequential

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

1.4 Limited parallel

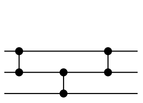
We took the same initial approach to parallizing as in the limited parallel Quicksort, i.e. spawing only a limited number of threads. Due to the balance of Mergesort, the processor should be at work all the time.

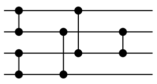
1.5 Quickmerge

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. Appendix

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


(a) 3-input


(b) 4-input

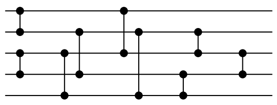
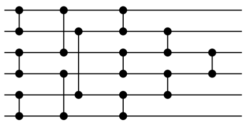

(c) 5-input

Figure 1: 3-5 input length optimal sorting networks [2]


(a) 6-input

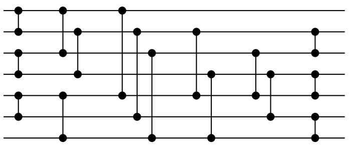
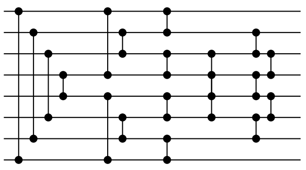

(b) 7-input

Figure 2: 6 and 7 input length optimal sorting networks [2]


(a) 8-input

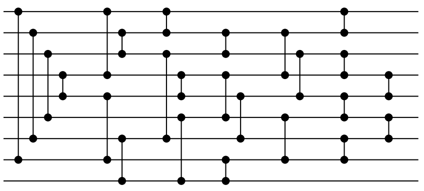
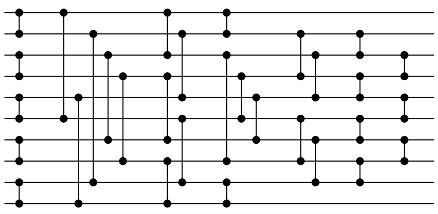
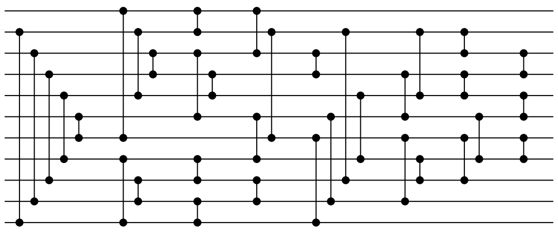

(b) 9-input

Figure 3: 8 and 9 input length optimal sorting networks [2]



(a) 10-input



(b) 11-input

Figure 4: 10 input length optimal, and 11 input best known sorting networks [2]

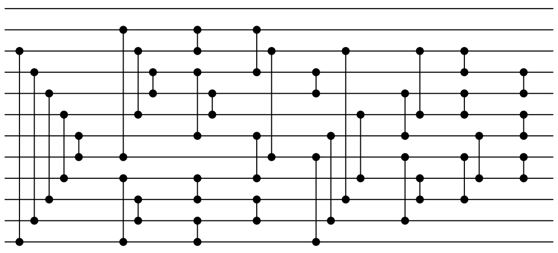


Figure 5: 12 input best known sorting networks [2]

References

[1] Michal Codish, Luis Cruz-Filipe, Markus Nebel, Peter Schneider-Kamp. *Applying Sorting Networks to Synthesize Optimized Sorting Libraries arXiv:1505.01962*

[2] A reference of the best-known sorting networks for up to 16 inputs - Ron Zeno (link)

[3] Sorting Networks and the END search algorithm - Hugues Juillé (link)

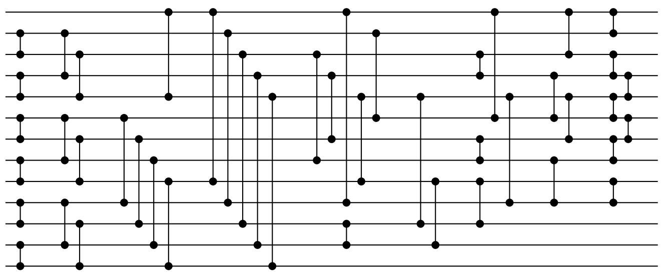


Figure 6: 13 input sorting network, derived from Greens 16 input sorting network [3]

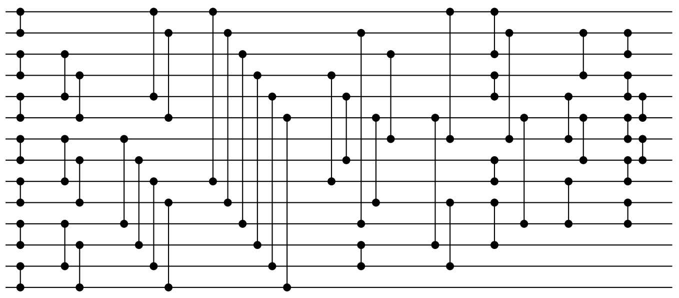


Figure 7: 14 input sorting network, derived from Greens 16 input sorting network [3]

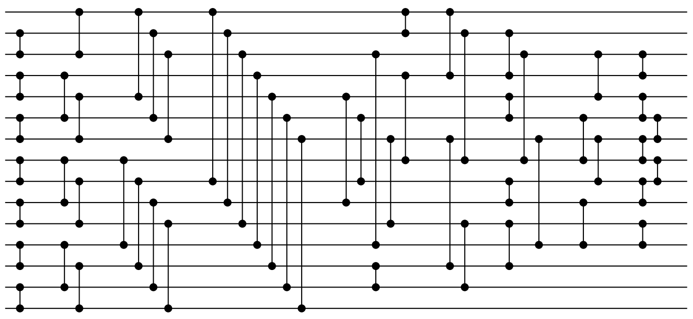


Figure 8: 15 input sorting network, derived from Greens 16 input sorting network [3]

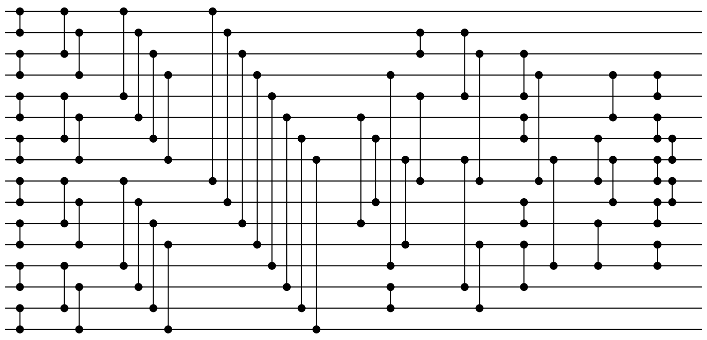


Figure 9: Greens 16 input sorting networks [3]