VU Parallel Computing
Project Report

Name: <u>Stefan Marschner, Moritz Macke</u>   Matr. number: <u>1426962, 0327088</u>

Due: January 30th, 2017

1 Abstract

We present three different implementations of a parallelized Quicksort algorithm in MPI, Cilk and OpenMP. By testing the implementations on multi-core systems we aim to compare the different algorithms in terms of runtime, speedup compared to the sequential algorithm as well as efficiency (speedup per thread).

2 Implementation - General

For all implementations a similar approach was used, utilizing prefix sums to parallelize the partition step of the Quicksort algorithm. Further the data is partitioned in the same way, between separation elements less than and equal from those greater than the pivot. The data size is therefore reduced only by one in the worst case, which is encountered if all elements are equal. These algorithms therefore are not suitable for data with more than a very small amount of equal elements. This issue could have easily been remedied in the simpler implementation variant but was left as is, since it was difficult to fix in the cilk2 variant, and test were conducted only on uniform random data in the range [0-n]. Furthermore all variants fall back onto the libc qsort function for sequentially sorting data as this was found to be significantly faster than self coded attempts. Choosing a fixed pivot, from the start, middle or end, or the median of those three proved to behave very badly in the case of sorted sequences, therefore the median of a number of randomly chosen elements is chosen for partitioning in all implementations. All OpenMP and Cilk variants are generic and can be used to sort any type if provided with a suitable compare function and type size, though only (unsigned) integer and double types were tested. For simplicity reasons the MPI variant was only implemented for integers.

3 Test System and Methology

Tests were conducted on sequences of randomly generated positive numbers in the range [0-n], where n is the length of the sequence. Most tests were run only on different sizes of integer type arrays. Each test was repeated 20 times, which turned out

to be on the low side for some tests, with noise still being quite evident. However it allowed for a wider range of coverage without taking an excessive amount of time. The cilk variants need an additional argument in the chunk size, which was fixed to n/200 -¿ 200 chunks for the first partition step for most tests. This is certainly not optimal for all sizes and processor counts and it is expected that performance could be improved by more carefully chosen numbers.

OpenMP & Cilk

The 48 core "saturn" server was made available to the students by the university. However by the time all programs were finished and bug free the server was hopelessly overburdened and no consistent results could be gathered. Therefore we decided to rent a server instance from Amazon EC2 for a few hours to run our tests on. This was a c4 compute optimized instance with 36 (virtualized) cores, the physical underlying machine uses *Intel Xeon E5-2666 v3* processors with 10 cores per CPU. Sporadic testing during development and incomplete tests on the saturn server showed comparable scaling performance to our final results, therefore we believe this to have been a suitable substitute.

MPI

The "jupiter" server with the SLURM workload manager was used to test our MPI implementation.

4 OpenMP

4.1 Algorithm

The OpenMP version uses a simple approach using prefix sums in a recursive implementation of Quicksort. At the start of the quicksort function the number of threads participating on this step is checked and if there is more than one the array is partitioned in parallel. In the partition step each processor gets a chunk of the array and partitions it into a buffer of size $\frac{n}{p}$, counts the number of less than or equal elements and stores it in a shared array. Thereafter it needs to synchronize with the other processors and the exclusive prefix sum is calculated for each of the chunks, which allows each processor to write both the smaller-equal and greater parts from the buffer back into the correct section of the array. The prefix step is currently implemented sequentially, since parallelisation would not pay off with the small number of processors involved. After the partition step is concluded all elements are on the right side of the array and the pivot is switched in between. Since the partition work is divided into (essentially) equal sized chunks in both partition phases each processor has to do an equal amount of work in regards to comparisons, memory reads and writes. The partition step should therefore be balanced, regardless of the choise of pivot, though this does not mean the algorithm as a whole is balanced, which depends on the quality of the split. The available processors are subsequently split into lower and upper halves, attempting to do some crude balancing depending on the size of the two subsequences, and the quicksort function is called recursively with the desired

number of processors per half. Once the number of threads taking part drops to one the algorithm switches to a Quicksort variant with sequential partition but tries to keep some parallelism by using tasks for the recursive calls. Once the array size drops below a threshold it is sorted purely sequentially.

4.2   Analysis

5   Cilk

5.1   Algorithm

First Variant (cilk0)
This Algorithm functions very similar to the OpenMP variant. During partitioning the data is written into a buffer array and then back into the original array. It therefore uses $O(n)$ additional space, plus $O(c)$ in each partition step, where c is the number of chunks = number of Cilk spawns. The main difference from the OpenMP version lies in how the Cilk system operates, as additional workers can only be spawned by function calls. Two passes are needed for partitioning between which synchronization is necessary. For each pass the work is divided with recursive calls and Cilk_spawn until it is small enough for sequential processing. The downside to this approach is that the overhead to spawn the Cilk workers is incurred thwice and there is no guarantee a worker in the second pass is assigned a chunk it worked on in the first pass. It is however by far the easiest to implement and requires no additional synchonization except the one provided by Cilk. The process of spawning the Cilk tasks is also used to calculate the prefix sum required in the second pass. While the assigned chunks have equal size, generally more tasks are spawned than availabe processors and the balancing in detail is left to the Cilk system.

First Variant (cilk1)
The motivation for this variant was trying to minimize the number of writes to memory and seeing if this improves performance over the first variant. It operates in the same way using two arrays, but instead of partitioning the assigned chunk in the first pass each worker only counts the number of LE elements. The prefix sum is calculated and in the second pass the data is written out to the second array, each worker knowing where to write its elements thanks to the prefix sum. Then the roles of the arrays are switched for the following recursive calls to Quicksort. Part of the data may end up in the second array after the parallel phase ends therefore first has to be moved back to the main array, this can be combined with a partition step however so the overhead should not be too large. Overall the performance of this variant turned out to be disappointing though, as evident from the benchmarks in Section 7.

First Variant (cilk2)
Focus here was to minimize the memory used and attempt to sort the data in place. The only additional memory required is the one for holding the chunk information. Like in the cilk0 variant in a first pass each chunk is partitioned, counted and used to calculate a prefix. The side to which smaller and bigger elements are partitioned

is alternated so as to generate less and longer runs, though it is not known whether this actually has a performance impact. The essential difference lies in the second pass. First the amount of swaps necessary to finish the partitioning is calculated and then divided evenly among workers (as long as there is more than a minimum). Each worker then computes the two locations above and below the split point (final pivot location), to start swapping elements. Using the partial prefix sums computed in the first pass this can be accomplished in $O(\log c)$ time. This variant turned out to perform the best.

5.2 Analysis

6 Test Results - Cilk, OpenMP
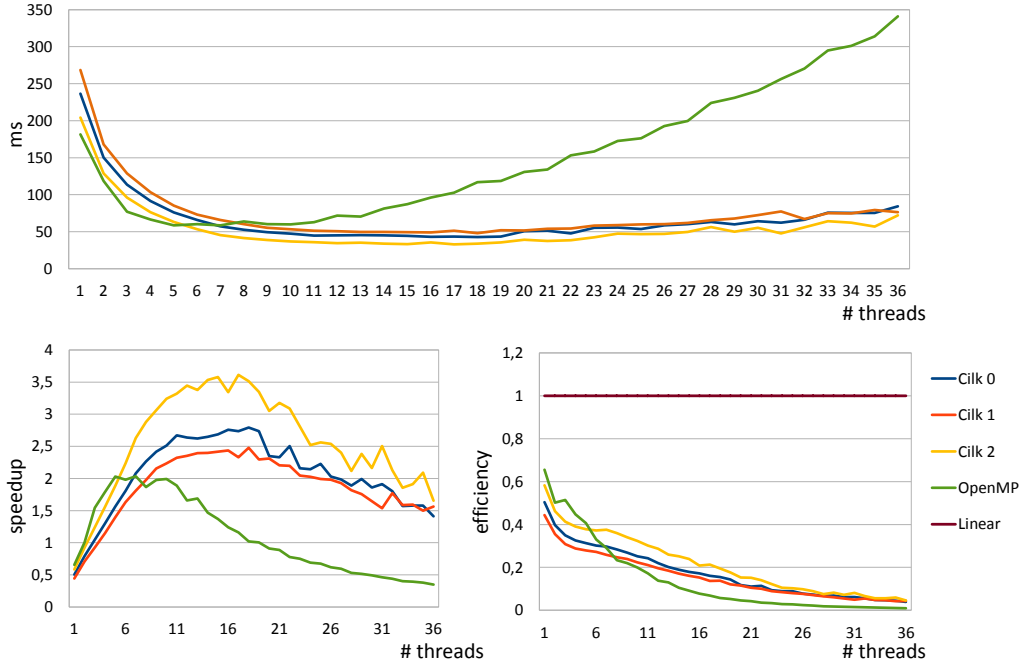
Figure 1: Runtime, efficiency and speedup results of the OpenMP/Cilk implementations for $n = 10^6$.

Runtime, Speed and Efficiency
As expected, the runtime of the algorithm stagnates at about 10 threads due to the implementation overhead and hardware limitations on the test server. After this point, the results for the OpenMP implementation get noticeably worse than the Cilk counterparts (most evident from *Figure 1*. This behavior of the OpenMP implementation does not seem to be accelerated by higher input size, however. In *Figure 3*, the OpenMP implementation is almost as fast as the Cilk implementation for a higher number of threads. The OpenMP implementation is actually much faster than the Cilk implementation for a lower number of threads. It can be concluded that OpenMP is the preferred implementation method for a reasonable amount of threads, at least for this particular algorithm.
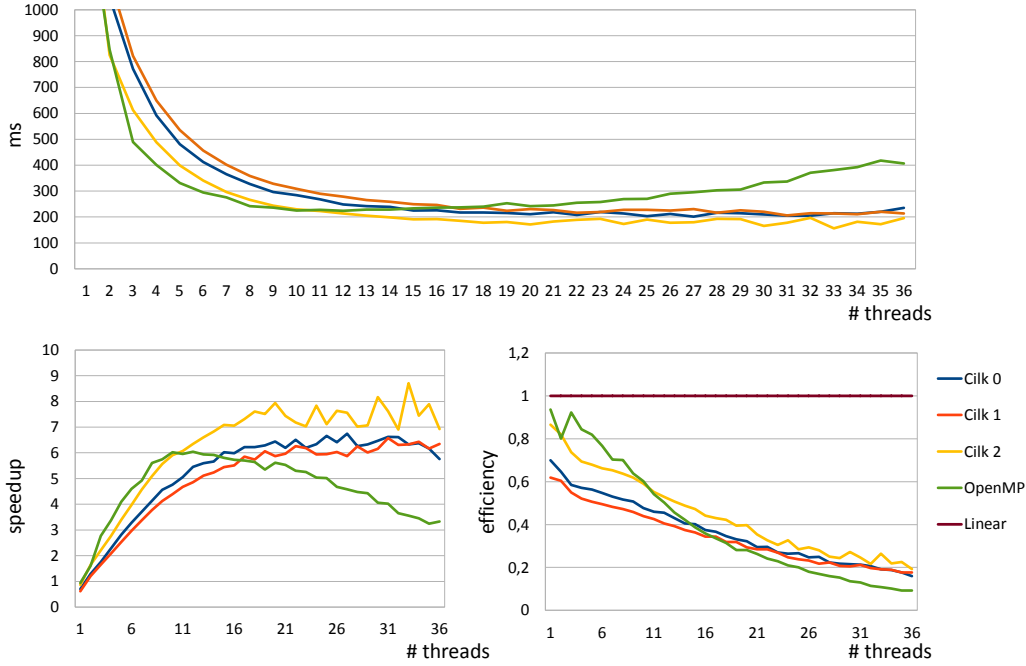
4

Figure 2: Runtime, efficiency and speedup results of the OpenMP/Cilk implementations for $n = 10^7$.

There is some odd behavior for the OpenMP implementation in *Figure 3* where the runtime results are much higher than expected and the efficiency results are impossible. We do not know what has caused this.

Comparison - Integer and Double Input

We have also compared the speedup of the Cilk2 and OpenMP implementations for $n = 10^8$ for integer and double arrays. The results are shown in *Figure 4*. As expected, the performance is very similar, with the OpenMP implementation being about 5% slower when doubles are used.

7 MPI

7.1 Algorithm

The partitioning algorithm for our MPI implementation roughly works as follows:

1. Divide input into (roughly) $\frac{n}{p}$ elements per process.

2. Node 0 is arbitrarily chosen as "master", all others as "slave".

3. At the start, all process are part of the same "set" (sets are emphasized by different colors in the graphic).

4. Every process picks 5 elements at random and computes median, then sends it to the master.

5. Master chooses pivots for every set of processes and sends it to them.
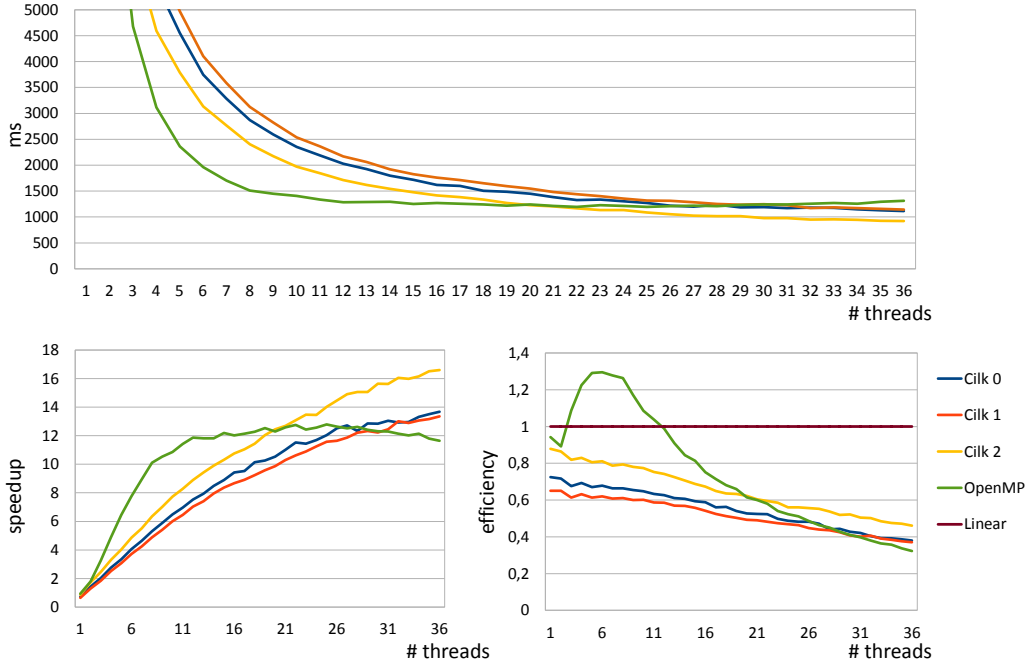
Figure 3: Runtime, efficiency and speedup results of the OpenMP/Cilk implementations for $n = 10^8$.

6. Every process reorders their dataset using the received pivot.

7. Depending on their rank, all processes send and receive part of a dataset to/from another process so that the lower $\frac{p}{2}$ processes only have elements smaller than the pivot and the higher $\frac{p}{2}$ processes only have elements bigger than the pivot.

8. Every set of processes is split in half and steps 4-7 are repeated until every process is in its own set.

9. A sequential Quicksort implementation is called in every process.

10. Every slave sends its sorted dataset to the master, which reassembles the datasets into the full sorted array.

However, it should be noted that the program can only be run with a number of processes that is a power of 2. However, the input size must not necessarily be a factor of the number of processes.

7.2   Analysis

Time complexity of the sequential Quicksort algorithm: $T_{seq}(n) = n \cdot \log(n)$

When the MPI implementation is run on p processors and we assume the best case scenario where the dataset belonging to a process is always split in half, p is a factor of n and the communication between the processes is instant, we can assume that the complexity of the parallel algorithm is $T_{par}(n, p) = \frac{n}{p} \cdot (\log(\frac{n}{p}) + \log(p))$

Here, $\frac{n}{p} \cdot \log(p)$ is the complexity of the partitioning after which every process ends up with a dataset the size $\frac{n}{p}$ which is then sorted using the sequential Quicksort
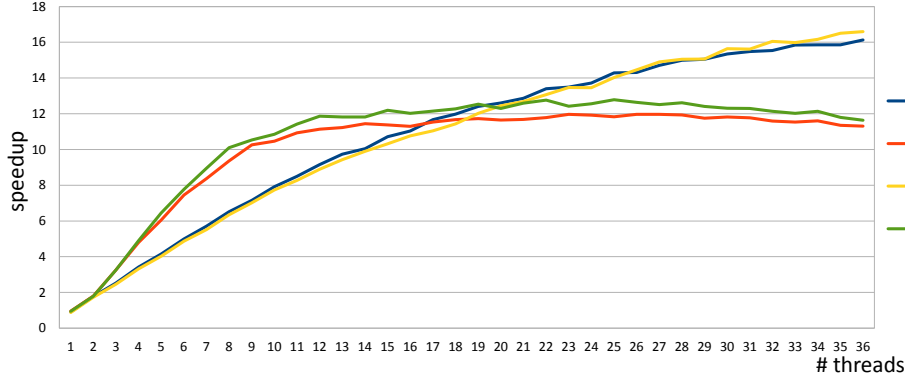
6

Figure 4: Speedup comparison integer vs. double for $n = 10^8$.
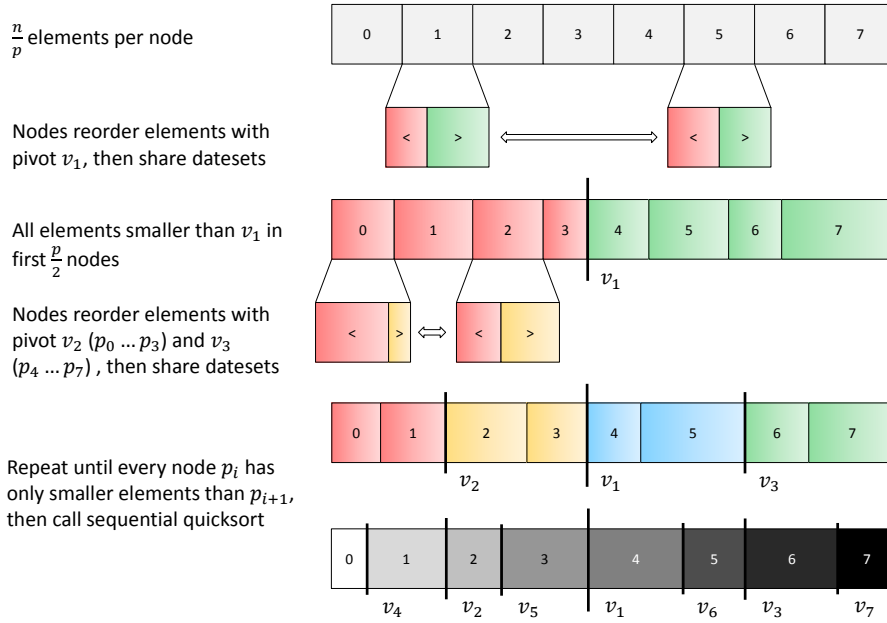

Figure 5: Example of the partitioning for the parallel Quicksort implementation for $p = 8$.

implementation. If we assume that we have infinite processors available we can set $p = n$ and have a time complexity of $T_\infty(n,p) = \log(n)$, and the *parallelism* of the algorithm is therefore $\frac{T_{seq}}{T_\infty(n,p)} = n$.

The speedup is calculated by $S_P(n) = \frac{T_{seq}}{T_{par}(n,p)} = \frac{n \cdot \log(n)}{\frac{n}{p} \cdot (\log(\frac{n}{p}) + \log(p))} = p\frac{\log(n)}{\log(n)} = p$

However, it should be obvious that such a speedup cannot be achieved in practice.

Drawbacks

The implementation works best for randomized input. If the array is pre-sorted, it would have a significant negative effect on the runtime of the algorithm because most of the work would be done by a single process. "Rebalancing" the datasets for a better utilization of all processes would be difficult and may not necessarily improve runtime.
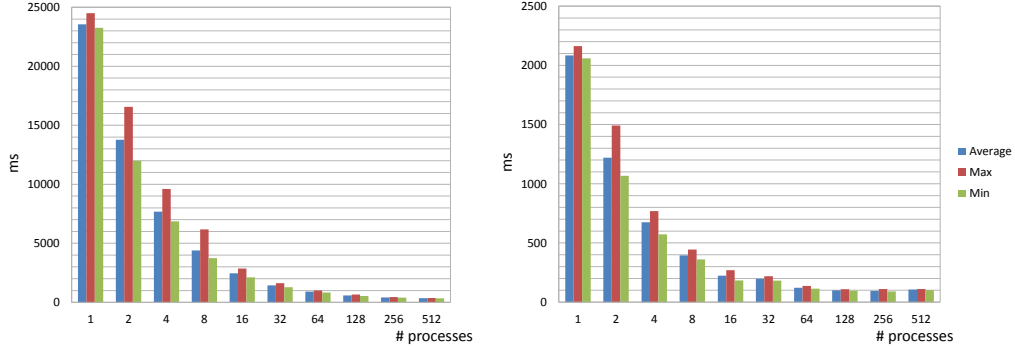
8 Test Results - MPI

Figure 6: Runtime results of the MPI implementation for $n = 10^8$ (left) and $n = 10^7$ (right).
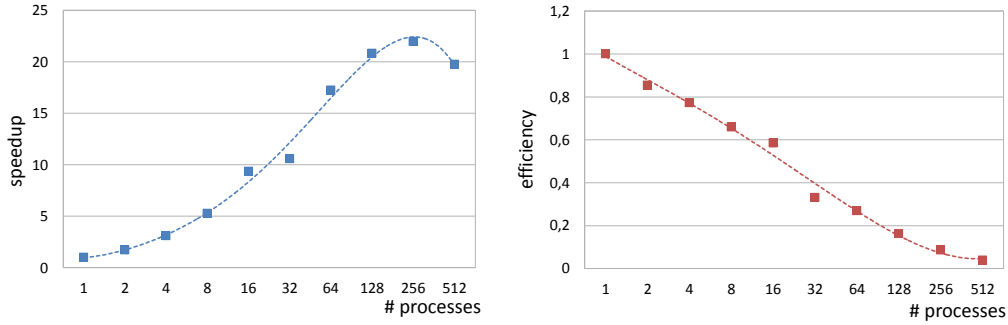

Figure 7: Speedup and efficiency results of the MPI implementation for $n = 10^7$.

Runtime, Speed and Efficiency

Figure 7 shows the runtime results of our implementation and *Figure 8* and *Figure 9* the efficiency and speedup results. Even for a high number of processes a respectable speedup can be achieved, at least until about $p = 256$ where the communication overhead decreases the performance significantly. As evident from the efficiency results, the algorithm scales more poorly the higher the number of processes is. At $p = 32$ and above the algorithm is obviously inefficient because the number of physical cores on the system is exceeded by the number of processes.

Pivot Selection

As noted above, in our implementation the median of 5 random elements is sent to the master, which calculates the median of all received pivots from a set of processors and sends this new pivot back to the set of processors. We tested our implementation with $n = 10^8$ and different numbers of elements randomly selected for the calculation of the pivot. We also tested the case that the pivot is always 0. This is the worst case scenario in which every element is received by processor 0 and the algorithm degenerates into sequential Quicksort with the additional overhead of the partitioning.
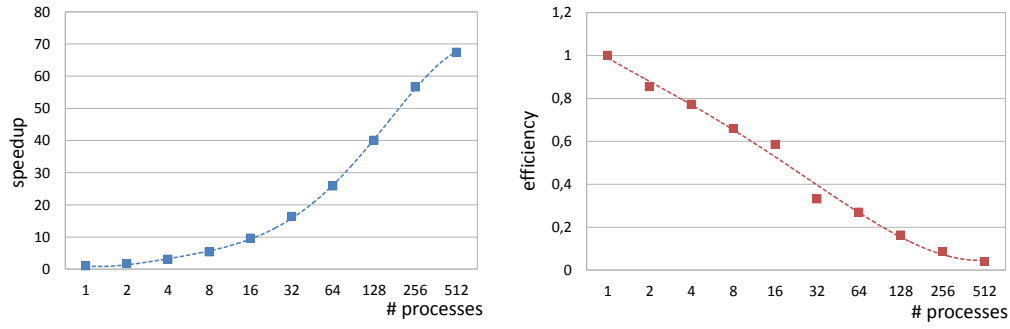
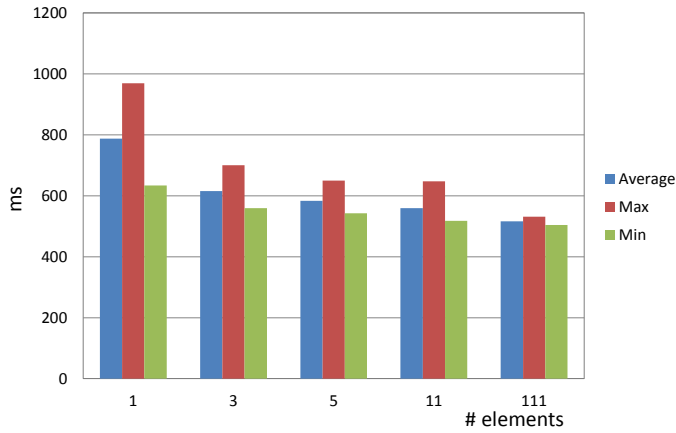Figure 8: *Speedup and efficiency results of the MPI implementation for $n = 10^8$.*

Figure 9: *Runtime for $n = 10^8$ and 128 processes, different number of elements to compute pivot.*

9   Conclusion

For all our parallelized Quicksort implementations, a good speedup could be achieved. We did, however, see that the limits of parallel computing are noticeable even though the Quicksort algorithm itself is very suitable for parallelization. At 16 threads and above (and even sooner for the OpenMP implementation), the implementation overhead already makes the speedup stagnate. The MPI implementation also suffers from the same limitations, and while the speedup results look better than for Cilk/OpenMP it should be noted that a (somewhat) linear speedup for exponentially higher number of processors is not desirable in practise.