Project 2

Performance and scalability analysis of a C application

Jonathan Åleskog, Adrian Nordin, Daniel Cheh

October 2019

Sytem Under Test

The experiments were done on a *Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz* which has two cores with hyper threading enabled on both cores. Main memory consists of two devices with 2048MB memory each and a speed of 1600 transfers/s. The operating system were Ubuntu 18.04.3.

The commands used to compile the un-optimized version:

```
gcc ReSoW.c -o test -pthread -O0 -lm
```

The commands used to compile the optimized version:

```
gcc ReSoW_Optimized.c -o test -pthread -O3 -lm
```

The performance test results before the optimization

We made 10 executions and got the average time spent in each section in the code.

Not Optimized	Avg. Time (ms)
Read	3.03
Sort	24774.50
Write	3.63
Calc. Average	0.0008
Calc. Min/Max	0.0008

Table 1: Data size of 131072 records with a buffer size of 1 record. Running 10 tests and calculating average.

The Hot Spot analysis

With the performance test, see table 1, we can see that the sorting algorithm is taking the most time. Which surely can be improved with a sorting algorithm with better time complexity. Furthermore the read and write operations takes longer time than the average and min/max calculations, which indicate that there can be optimizations done on read/write.

What can be improved?

We can try to improve the sorting algorithm with an algorithm which has a smaller time complexity. Another solution can be to enable optimization when compiling the source code or changing the buffer size.

Optimization Implementation

First we switched the sorting algorithm from Selection sort to Quicksort. To lower the time complexity and in turn lower the service time. Selection sort always has a time complexity of $O(n^2)$ compared to Quicksort which has an average time complexity of $O(n\ log\ n)$.

We also changed the buffer size to decrease the number of reads and writes which in turn decreased the execution time. Furthermore we enabled full optimization when compiling and observed a decrease in the execution time.

The Performance Test Results After Optimization

Here are all the results from the optimization that were done.

Quicksort

Algorithm	Avg. Time (ms)
Selection sort	24774.50
Quicksort	26.04

Table 2: Data size of 131072 records with a buffer size of 1 record. Running 10 tests and calculating average.

In table 2, we can see that changing the sorting algorithm to quicksort made an improvement of 951 times faster.

Change of buffer size

Optimized	Avg. Time (ms)
Read	0.219
Write	0.560

Not Optimized	Avg. Time (ms)
Read	3.03
Write	3.63

Table 3: Data size of 131072 records with a buffer size of 131072 record. Running 10 tests and calculating average.

We detected that increasing the buffer size made a significant improvement in execution time. Read became 13 times faster, while write became 6 times faster. We chose the buffer to be as big as the file size, which means only one big read is required for the whole file.

GCC Optimization

Optimized	Avg. Time (ms)	Not Optimized	Avg. Time (ms)
Read	2.76	Read	3.03
Sort	8595.36	Sort	24774.50
Write	3.3	Write	3.63
Calc. Average	0.0007	Calc. Average	0.0008
Calc. Min/Max	0.0006	Calc. Min/Max	0.0008

Table 4: Data size of 131072 records with a buffer size of 1 record. Running 10 tests and calculating average.

When we change from no optimization (-O0) to full optimization (-O3) we observed a gain in performance in all parts.

The scalability test/analysis results

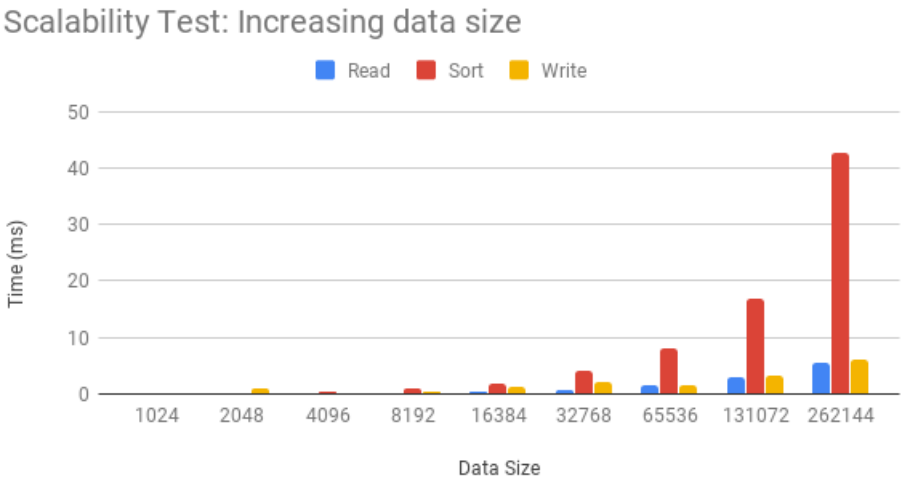

Figure 1: Increasing data size. Running 10 tests and calculating average.

From figure 1 we can see that the the sorting algorithms time complexity increases exponentially. This is also true for read and write.

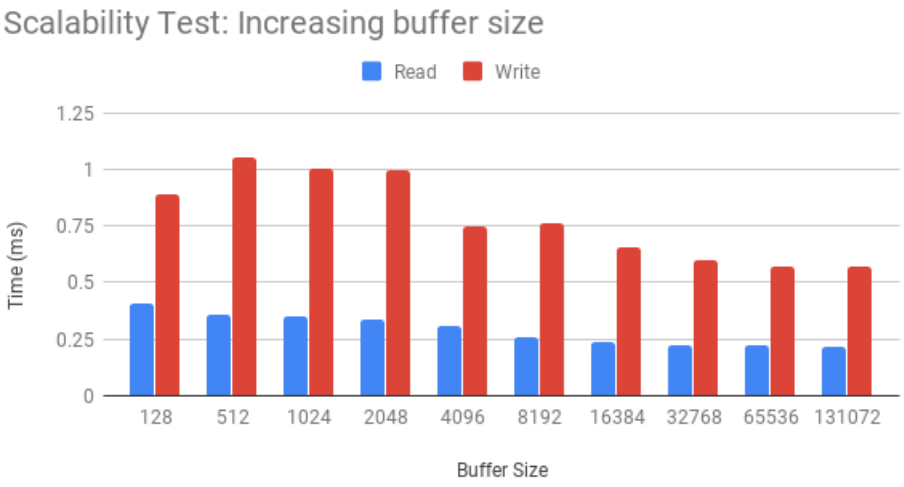

Figure 2: Increasing buffer size. Running 10 tests and calculating average.

Figure 2 shows us small decrements of read when increasing the buffer size. On the other hand, we see an inconsistent pattern on write because of an unknown factor, we suspect that it involves the operating systems way of writing memory in pages. But overall we see a linear decreasement of write.

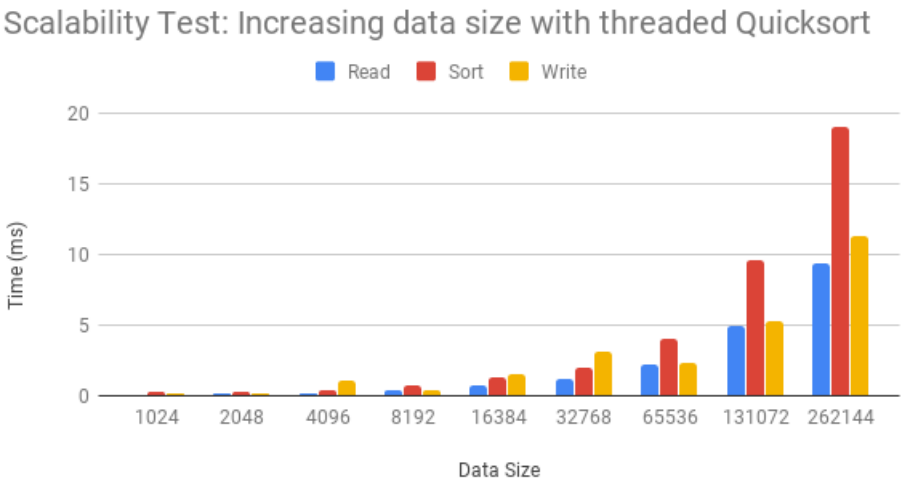

Figure 3: Increasing data size, with threaded Quicksort. Running 10 tests and calculating average.

When enabling threaded Quicksort we can still see the $O(n\ log\ n)$ nature of Quicksort, but the execution time has been decreased. See figure 3.

Quicksort	Avg. Time (ms)
Threaded	71.54
Not threaded	160.51

Table 5: Data size of 1048576 records with a buffer size of 1 record. Average time from 100 executions.

Implemented thread support for sorting. This was done with four threads running Quicksort on four separate regions of the unsorted list. Merge sort was then implemented to connect these separate regions into one. The performance gain we can see in table 5 is an improvement of 225% in processing time. This result was expected due to the SUT having two cores which are hyper-threaded. The result should be a little more than twice as fast, because of hyper-threading being able to yield a performance improvement of maximum 30% and utilization of two cores allowing the processing time to be halved.