A4: Programopførsel

Bjørn Christian Vedel Bennetsen (qfr834) &
Kalle Kromann (zmr905)

29. Oktober 2017

1 Measuring programs and determining constants

Because we have to measure all of the x64-programs for different n and on 3 different machine types, we have chosen to write a shell-script that assembles, runs and retrieves the total number of cycles of the x64-programs we want to measure, and stores the data in appertaining data-file. Each line of the data-files has the format (n *num_cycles*), and is stored in `src/data_files/` and has the naming convention `*program name*_n_data*i*`, where `*program name*` is the name of the program without the extension and `*i*` correspond to a machine type($i = 1 =$ `machine 1`, etc). The handed out x64 programs is placed in the directory `src/examples/` and our programs is placed in `src/asm_programs/`. To determine the constants for the different machines, we have written a python-script that does this for matrix-multiplication- and sorting-programs respectively. It prints out the constants for each simulation with input n on a machinetype and a mean value for the constant. Lastly it produces a single plot containing the data of a simulation by using the `matplotlib` library. The plot contains the running-time data-points of the sorting-algorithms with $y = k \cdot x \cdot log(x)$ and matrix-multiplication-algorithms with $y = k \cdot x^3$ where k is the mean value of the constants for each of the machine simulation-runs. This help us see if the algorithm does in fact have the expected running time.

In order to run the scripts, the current working directory has to be `/src`.

To run the shell-script and to make the data-files run the command:

```
./performance_test.sh
```

To plot one or more data files and finding constants, run the command:

```
./analyze_data.py arg1 [data-file](s)
```

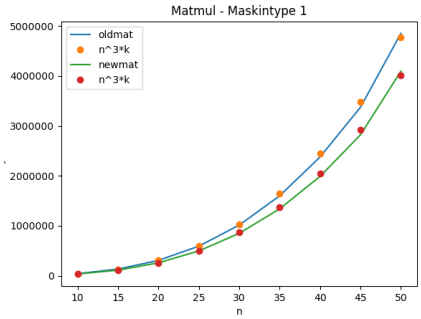
Where arg1 is 1 if the programs being plotted are matmul and 2 if they are sorting programs. You can also run make_plots.sh to make all the plots.
Of course the plots will appear in this report along with the constants we have found, so there is no need to run any of them.

2 Matrix-multiplication

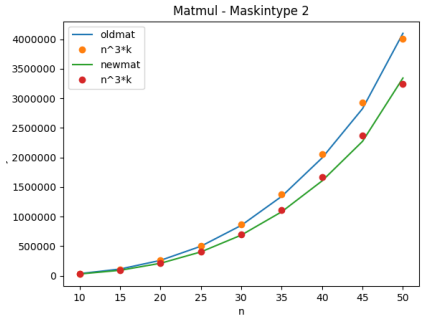
The given matrix-multiplication x64-program has a expected running time complexity of $O(n^3) = k \cdot n^3$. To verify that the running time of the program is what we expect it to be, we should find constants that is roughly the same for any of the simulators machine-types. If that's the case, we can use the mean value of the constant to represent k.
Furthermore, by also plotting the function $f(x) = k \cdot x^3$ and seeing that the points fits the actual measured points, it means that the program must indeed have the expected time complexity.
For the matrix-multiplication-program we found that the constants were consistent:
Machine 1: $\bar{k} = 38.21$
Machine 2: $\bar{k} = 32.11$
Machine 3: $\bar{k} = 6.67$

2.1 Development of a faster matrix-multiplication program

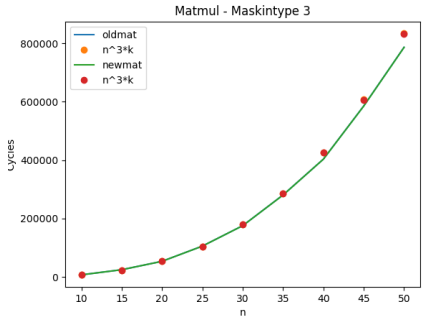
We are to make a x64-matrix-multiplication-program with the same time-complexity, but performs better than the given matrix-multiplication-program i.e. it has a lower a constant k.
In the given program, the dimension and the indexes `i,j` and `k` is being incremented by 1, and then for each indexing it is multiplied by 8 obtain the correct index in the array. We know that multiplying is a relatively expensive instruction - taking 4 cycles to execute, so we want to avoid that as much as possible.
By not incrementing i, j and k by 1, we can instead increment them by 8(bytes). The dimension is also multiplied by 8.
By doing this we avoid having to multiply with dimension each time we have to index into the array, and thus saving us 4 cycles per indexing!
We found that the constants were consistent and we determined the final constants to be:



(a) Matrix multiplication - Machine 1



(b) Matrix multiplication - Machine 2



(c) Matrix multiplication - Machine 3

Machine 1: $\bar{k} = 32.08$

Machine 2: $\bar{k} = 25.98$

Machine 3: $\bar{k} = 6.66$

As its seen by looking at the plots and the constants, we can see that our version of the matrix-multiplication program does in fact perform better than the given version. The plotted points from the function $f(x) = k \cdot x^3$ does fit nicely to the points of the actual measurements, and thus indicating that our version also has the time-complexity $O(n^3 = k \cdot n^3$.

Anything else would not make sense, since the only difference is that we represent our variables i, j, k and dim as bytes and because of that, we can avoid multiplying each time we index into the array.

It's worth noting that the last plot (for machine 3) only appears to show one program, but it is because the results are so close that the other can't be seen.

3 Sorting programs

We look at the constants and find them to be consistent for each of the sorting programs, including the our new sorting program, quicksort. The following is the mean of the constants for each program and for each machine type.

Heapsort 1:
Machine 1: $\bar{k} = 39.09$
Machine 2: $\bar{k} = 30.72$
Machine 3: $\bar{k} = 21.60$

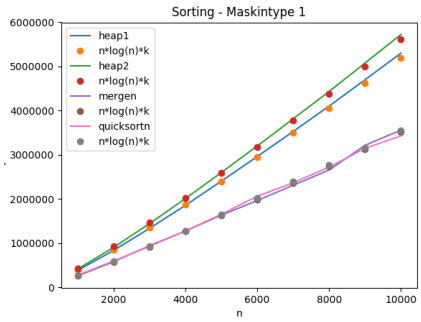
Heapsort 2:
Machine 1: $\bar{k} = 42.26$
Machine 2: $\bar{k} = 33.87$
Machine 3: $\bar{k} = 19.00$

Mergesort:
Machine 1: $\bar{k} = 26.45$
Machine 2: $\bar{k} = 19.65$
Machine 3: $\bar{k} = 15.32$

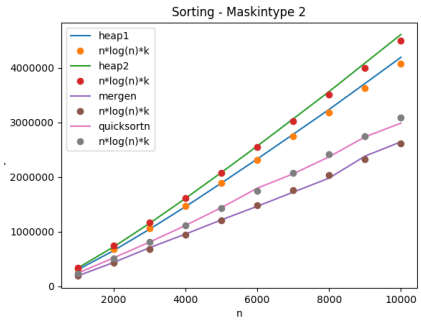
Quicksort:
Machine 1: $\bar{k} = 26.75$
Machine 2: $\bar{k} = 23.28$
Machine 3: $\bar{k} = 15.17$

By plotting the function $n\log(n) \cdot k$ (for each machine's k) we find that all the programs have very near the expected time complexity of $O(n\log(n))$.
From the plots and constants it is easy to see that quicksort is consistently faster than both heap sort programs.
Mergesort and quicksort have very nearly the same constants, and it is also evident from the plots that the two programs perform roughly equally good. On machine 2, mergesort seems to be consistently faster than quicksort, but both on machine 1 and 3 quicksort outperforms mergesort, especially on higher n's. Even though it was not part of the assignment we did go ahead and test quicksort and mergesort for higher n's, namely n=10000-100000. From this data it is evident that quicksort is consistently faster than

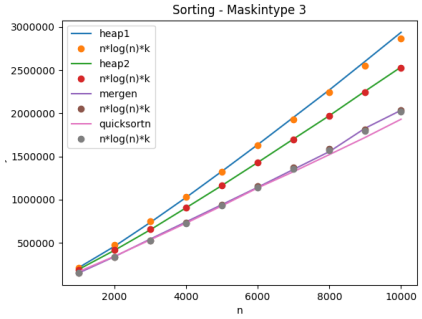
mergesort on each of the 3 machines on high n's. We also included these big n plots on the last page.



(a) Sorting - Machine 1



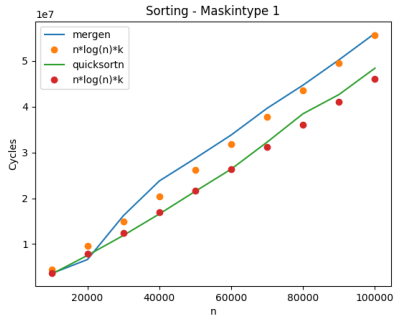
(b) Sorting - Machine 2



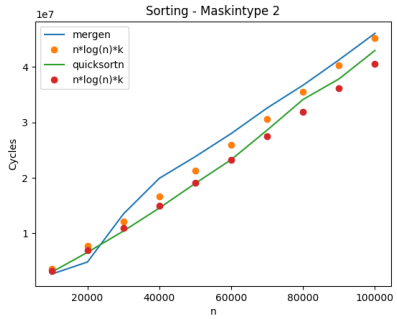
(c) Sorting - Machine 3

3.1 Development of a faster sorting program

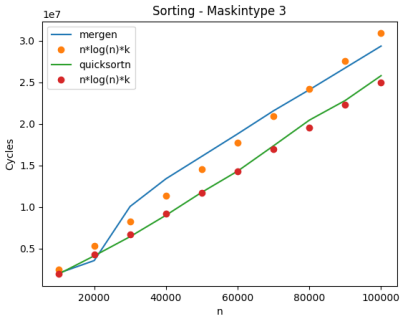
We decided as suggested to develop quicksort as the faster sorting program. We used the Hoare partition scheme, where two pointers are moved towards each other from each end of the array until they pass eachother, swapping "inverted pairs" as they come across them. An inverted pair is where one element is greater or equal, and one lesser than or equal to the pivot and these two elements are in the wrong order in the array. This partition scheme is good for caching because we start by moving one pointer, and then the other, meaning we never jump around in the array. Since we store the value of the array in a register when done with each loop there should be no jumping in memory (from i to j) at all, which will have a bigger impact on the running time, the greater the array size is.



(a) Mergesort vs Quicksort on large arrays - Machine 1



(b) Mergesort vs Quicksort on large arrays - Machine 2



(c) Mergesort vs Quicksort on large arrays - Machine 3

We also utilize the "trick" that the low and high indices are both incremented/decremented with 8 (meaning they are already aligned), and are added with the array position from the start which means we save some instructions (approx. 3 instructions per memory look up), and can use the values to directly look up memory.

Since quicksort works in place, meaning the elements of the array is swapped around, it means the cache has a higher hitrate. It also means that it's not obvious from the output of the -diff simulator flag that the array was sorted correctly. Therefore we have included another program, $newsort_debug.x64$, which before halting will send the array to a new place in memory in order, and it can be verified from the memory diff dump that the array was indeed sorted.