DM207 I/O-Efficient Algorithms and Data Structures

Fall 2015
Project 1

Mikkel Levisen and Jesper Lund

October 19, 2015

Contents

Introduction

The goal of this project is to a feeling of the possible impact of differences in memory access patterns on the running time of programs.

Through this report have the running times been plotted along with an estimate on of cache and ram sizes.

The machines used in this assignment are the ones from the IMADA terminal room and had the following specifications:

Memory type		size
L1	32	kB
L2	256	kB
L3	4096	kB
RAM	3910084	kB

Since these computers are available to other student at the same time as the tests were running it might have effected the results.

Where appropriate we have added dashed lines indicating the estimated cache and RAM limits. These guides are only estimates and don't consider other running processes.

Task 1

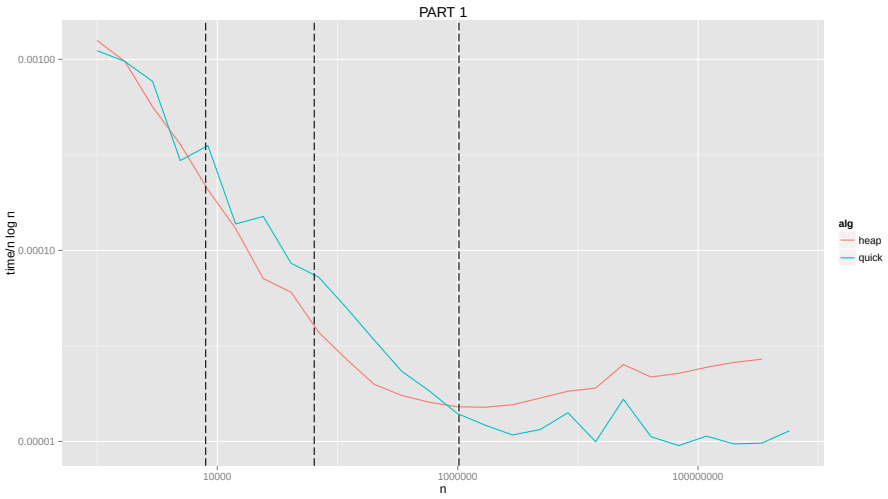

Figure 1: Quicksort vs. Heapsort on arrays of `int`.
(`int`= 4 bytes)

In figur 1 are arrays of sizes between 1000 and 577011370 containing integers sorted with Heapsort and Quicksort. The plot shows the running time of each algorithm relative to the expected running time $O(n \log n)$.

Based on the memory access pattern of the algorithms we are surprised that Heapsort seems to outperform Quicksort while handling arrays which fit in cache.

Once the array sizes exceeds the cache limit it clearly show that the spread out memory access of Heapsort is heavily penalized.

Task 2

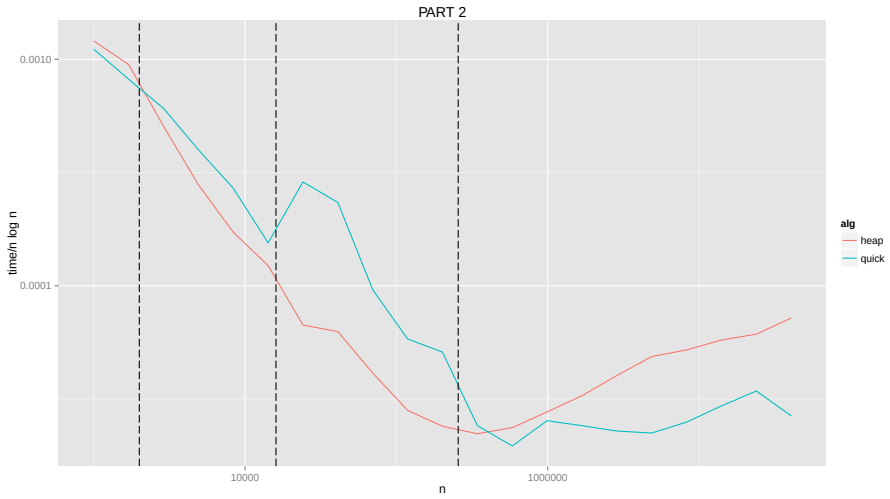

Figure 2: Quicksort vs. Heapsort on arrays of `Integer`.
(`Integer`= 16 bytes)

Because Java uses *pass-by-reference* when dealing with objects will an array of objects simply be pointers to memory locations in the heap witch removes Quicksort memory access pattern advantage with sequential data. In figure 2 this is especially visible when the data size reaches L3 cache. In RAM Quicksort still has an advantage because it focuses on distinct partitions of the date rather than the whole set. This leads to a more efficient use of the cache.

Task 3

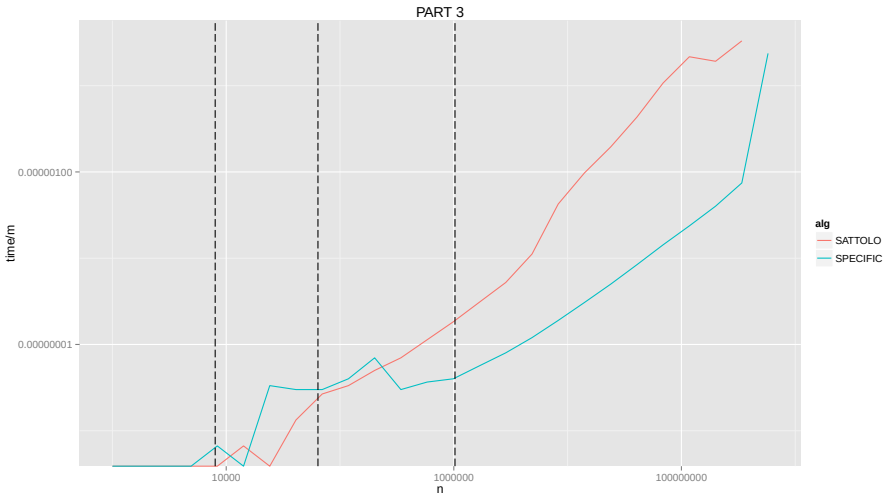

Figure 3: n-cyclic list lookup

Here we clearly see that SATTOLO performs much worse than a standard n-cycle because it need to load the same blocks in over and over again where SPECIFIC only needs to load a block once.

Task 4

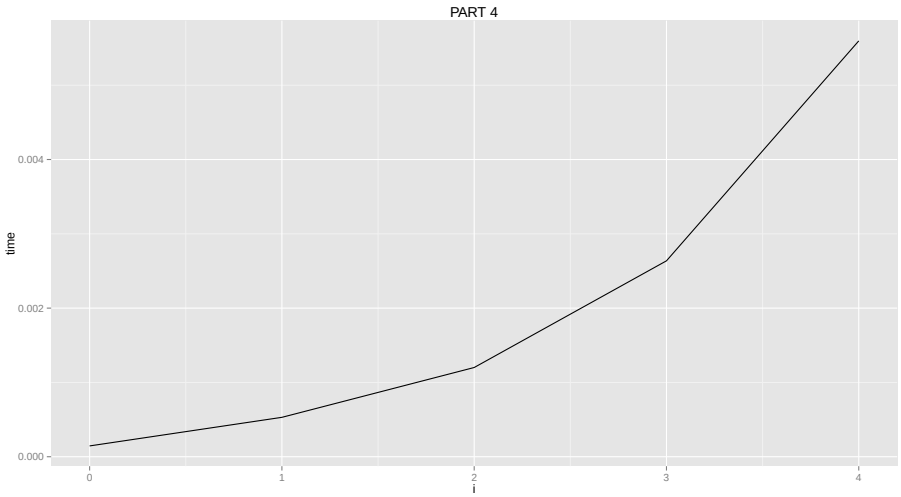

Figure 4: Random array access

At each i the number of elements used in each block is halved. This leads to doubling the number of block loads. We expected to see an exponential increase in running time until reaching a maximum of one block load pr operation.

Because our cacheline is 64 kB and one `int` is four bytes d^i would have to be equal or higher than 16000.

Task 5

5.1

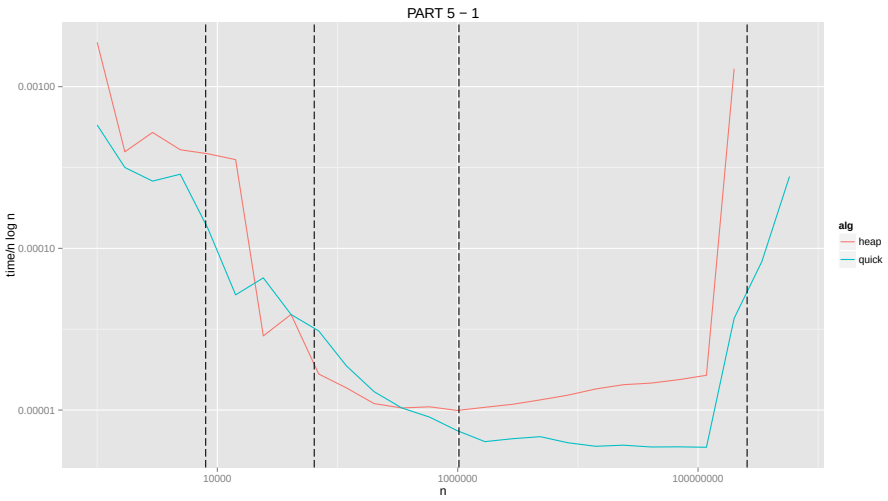

Figure 5: Quicksort vs. Heapsort on arrays of `int`. (`int`= 4 bytes)

Here we can clearly see the difference in memory access time when data size exceeds cache and RAM. Both Heapsort and Quicksort suffers in speed due to the extremely slow load time of the disk.

Quicksort still outperforms Heapsort since because it deals with data in sequential chunks.

5.2

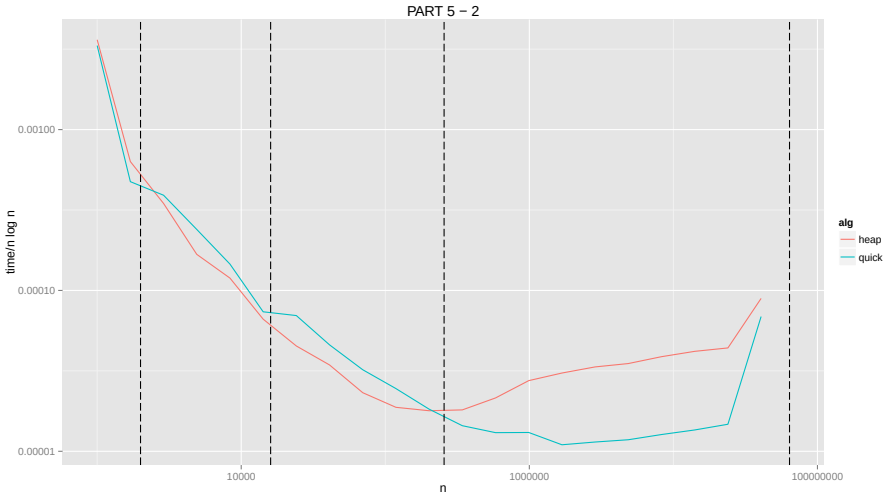

Figure 6: Quicksort vs. Heapsort on arrays of `Integer`.
(`Integer`= 16 bytes)

Quicksort still outperforms Heapsort but the effect is less significant because of how Java handles objects. Objects outside of ram can now be stored in different sectors and would dramatically reduce the efficiency of Quicksort.

5.3

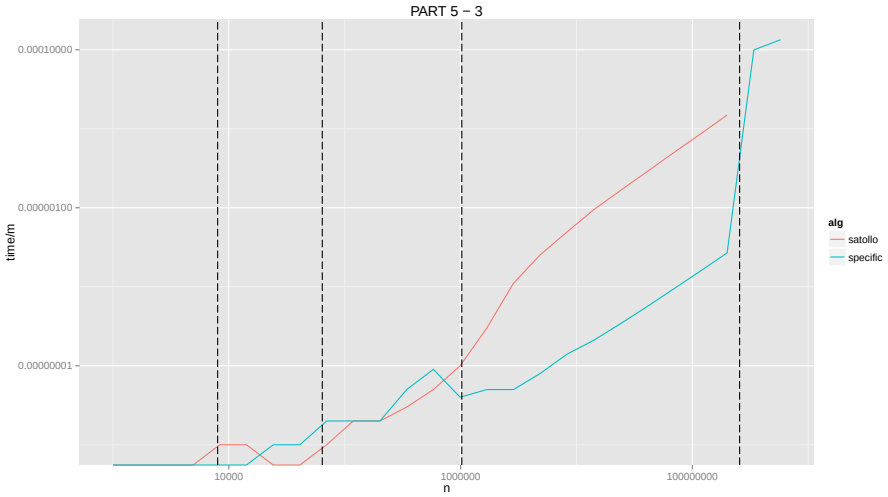

Figure 7: n-cyclic list lookup

Again we see a huge running time penalty from utilizing the disk. Note that SATTOLO timed out when it hit the disk.

5.4

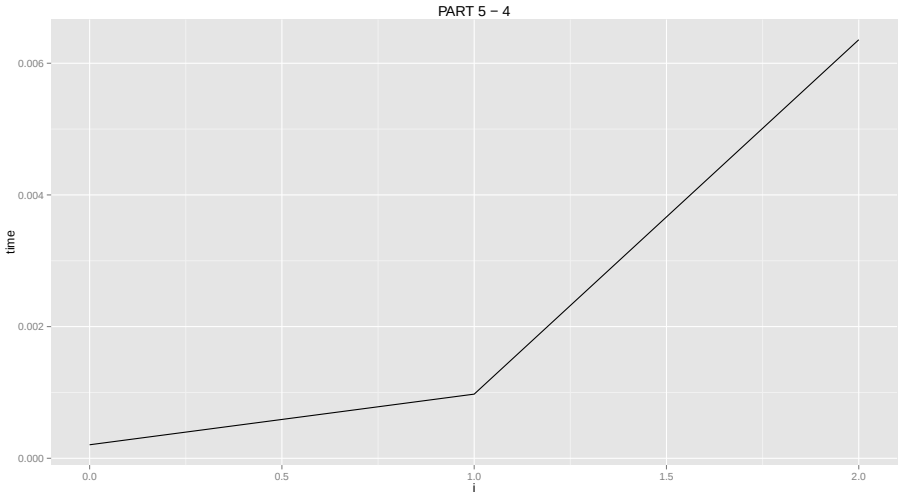

Figure 8: Random array access

Since the array is larger than the RAM + cache is the operation very expensive right from the start, timeing out at $i = 3$.