OSC project 3

Dieks Scholten s1098110

October 2024

Summary of the article by Kamp

In this article, Kamp highlights that the standardized methods of analysing the performance of algorithms are out-of-date. Classically the performance is done in big-O where $0.001 \cdot n^2 = O(n^2)$ is worse than $10^9 \cdot n = O(n)$. This is theoretically true but in practice, the first complexity is much better. The problem is not just this oversight, but also that virtual memory is an important factor for run-time in modern systems and is completely ignored in this form of analysis. To illustrate, Kamp explains that by changing the design of certain data structures, such as optimizing a binary heap into a B-heap, you can better optimize for certain underlying protocols like paging. The problem explained by Kamp is that we should update and re-evaluate the methods of analysing algorithms and look at what matters. It is not just the most important action, like a comparison in sorting, but also modern hardware characteristics like VM pressure, paging, and energy consumption.

Question on B-Heaps

The idea behind delaying the expansion of the first generation of elements in a B-Heap page is related to optimizing memory locality and minimizing page faults under high VM pressure, which leads to fewer page swaps. In binary heap structures, the relation between parent and child nodes is often very important and many algorithms rely on making fast comparisons between them. If the children of a parent happen to be on different pages (which is common in a classical binary heap), these comparisons might be slow, especially under high VM pressure, as page swaps become likely.

By deferring the expansion in the cases where children and parents would not fit on the same page, this problem can be mitigated. In a B-Heap, the parents and children always stay on the same page which results in fewer page swaps and thus faster comparisons.

As for page sizes, the above-explained theory works for all page sizes (as long as they are not too small to hold multiple nodes). Larger page sizes do allow for more tree nodes to be stored which results in the effectiveness of this technique being less noticeable. However, even with larger pages, the technique still applies and remains effective in scenarios where memory locality is critical for performance.

Baseline and Parameters

The following are my machine specifications:

- Processor: Intel i5 7200U
- Number of cores: 4x 2.50GHz
- Memory (RAM): 8.0 GiB
- Operating System: Ubuntu 22.04.4 LTS (on performance mode)
- Cache: L1 = 128KiB, L2 = 512KiB, L3 = 3 MiB

My machine is dual-booting with Windows 10. My Ubuntu partition has about 344GB available on SSD.

My baseline was set with the following parameters (unchanged):

- SIZE = 16384ULL
- REPEAT = 1ULL

My baseline results were:

- turn-around time: 24.677191065 s
- CPU cycles: 76,141,704,349 cycles
- getrusage()

  user time: 23.681019 s

  soft page faults: 524395

  hard page faults: 0

  max memory: 2099840 KiB

  voluntary context switches: 1

  involuntary context switches: 1720

  typical page size: 4096

Modifications

Swapping variables in double-loop

The first modification is on line 43 and line 44. I swapped the loops. Initially, line 43 looped over `i` and line 44 looped over `j`. This was switched to line 43 l0oping over `j` and line 44 over `i`. Results of this change:

- turnaround time: 24.67.. $s \rightarrow$ 15.32.. s

- CPU cycles: $76,141,704,349$ cycles $\rightarrow$ 47.295.676.308 cycles
- involuntary context switches: 1720 $\rightarrow$ 769

An involuntary context switch occurs when the CPU is forced to switch its attention to another process, which, for example, might have a higher priority. In this case, the switches are probably, overwriting memory in the cache, which we use as a proxy for page swaps. The decrease in involuntary context switches means there are considerably fewer "page swaps", which is expected. In the original version of the code an assignment of `res[j * SIZE + i] = 0;` is directly followed by `res[(j+1) * SIZE + i] = 0;`. Since `SIZE` is so large the location of these two values are far apart and thus the element at the second index will likely not be pre-cached since it does not fall into "spatial locality". This means this element needs to be loaded into the cache, which is slow. When swapping `i` and `j`, as described above, the assignments which follow each other are: `res[j * SIZE + i] = 0;` and `res[j * SIZE + i + 1] = 0;`. These values are very close to each other in memory and will likely both be in the cache due to pre-caching. They will both be pre-cached since they are in close proximity in memory. The number of "page swaps" or involuntary context switches is, therefore, a lot lower.

The second modification is very similar to the previous modification. On line 46 I swapped `k` with `l` and on line 47 it was vice versa. The results of this change are:

- turnaround time: 15.32.. s $\rightarrow$ 12.51.. s
- CPU cycles: $47,295,676,308$ cycles $\rightarrow$ $38,706,857,210$ cycles
- involuntary context switches: 769 $\rightarrow$ 100

The reason for the optimization is the same as in modification 1. The reason why there is less improvement is because the loops are smaller thus the impact is also smaller.

The third modification like the first one involves swapping `i` and `j`. This time on line 55 and line 56. The improved results are:

- turnaround time: 12.51.. s $\rightarrow$ 8.11.. s
- CPU cycles: $38,706,857,210$ cycles $\rightarrow$ $25,095,201,911$ cycles
- involuntary context switches: 100 $\rightarrow$ 93

Removed additional loop

The fourth modification I performed was removing the loop on lines 55 to 59. Instead, the dummy variable is updated on line 52 after `res[j * SIZE + i] /= 9;`. Although this mitigates the improvements made in modification three I added modification three to show improvement by removing the additional loop made. The improved results are:

- turnaround time: $8.11..\ s \rightarrow 7.51..\ s$
- CPU cycles: $25,095,201,911$ cycles $\rightarrow 23,173,302,545$ cycles
- involuntary context switches: $93 \rightarrow 129$

Blocking

The fifth optimiation is applying Blocking or Loop-Tiling. The idea is that looping through `img` and `res` row-by-row is inefficient since the amount of memory required to store one row is more than fits in the L1 cache. Looping over small blocks of `img` and `res` could improve performance since there will be fewer cache misses. To do this a block size must be determined. Since my L1 cache is 128KiB the block size should be such that it gets close to the value, but not exceed it since this will result in more cache misses. The number of bytes which need to be allocated expressed in block size is roughly: $B^2 \cdot 8 \cdot 2$ bytes[1] (where B is the block size). This table provides the rough memory size and the test results of different values of B:

Table 1: Test result for different block-sizes (B)

B	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
$\approx$ KiB	1.0	4.1	16.4	65.5	262.1	1048.6	4194.3	16777.2
Turnaround time (s)	6.89	6.88	6.80	6.75	6.78	7.47	7.03	7.16
CPU cycles (10^9 cycles)	$21,3$	$21,3$	$21,0$	$20,8$	$21,0$	$23,0$	$21,7$	$22,1$

From the table, it is easily seen that $2^6 = 64$ is the most optimal block size. This corresponds to the theory since 65.5KiB fits in the L1 Cache while more does not fit and less does fit but, as is evident from the table, smaller block sizes perform worse. It is unnecessary to search for something more optimal than a block size of 64 since the difference between 32, 64 and 128 is already very small.

This improvement has the following result:

- turnaround time: $7.51..\ s \rightarrow 6,75..\ s$
- CPU cycles: $23,173,302,545$ cycles $\rightarrow 20,826,543,869$ cycles
- involuntary context switches: $129 \rightarrow 36$

[1] B is squared since the block is square so there are B-squared elements in the block. It is multiplied by 8 since a 64int_t is 8 bytes. It is multiplied by 2 because elements in `res` and `img` are accessed. This is not precise there are a couple of extra elements accessed in `img` but it is roughly correct.

Final Measurement

I will now list the final measurements compared to the baseline:

- turn-around time: $24.677191065 \rightarrow 6.752919891$ s
- CPU cycles: $76,141,704,349 \rightarrow 20,826,534,869$ cycles
- getrusage()

  user time: $23.681019 \rightarrow 5.795102$ s

  soft page faults: $524395 \rightarrow 524395$

  hard page faults: $0 \rightarrow 0$

  max memory: $2099840 \rightarrow 2099968$ KiB

  voluntary context switches: $1 \rightarrow 1$

  involuntary context switches: $1720 \rightarrow 36$

  typical page size: $4096 \rightarrow 4096$

The improvement in turnaround time is about $72,6\%$. If `REPEAT` is set to `0` the turnaround time is about $4,003$ seconds. Subtracting this from the baseline turnaround time and the final turnaround time gives an improvement of about $86,7\%$ of the code which was allowed to be changed.

The soft page faults are unchanged but this is because we are not optimizing for page faults but for caching, which is used as a proxy for pages. Thus the change in involuntary context switches better reflect the improvement. When something is not in the cache the CPU has to switch context to write something to the cache. This has been reduced drastically. The left-over context switches are in part because more programs are running at ones which also want some cache but also because `res` and `img` do not fit in the cache thus there needs to be some overwriting of the cache or "swapping".