ETH zürich

High Performance Computing for Science and Engineering I

P. Koumoutsakos
ETH Zentrum, CLT E 13
CH-8092 Zürich

Fall semester 2018

Set 2 - Cache Design, Mutual Exclusion

Issued: October 5, 2018
Hand in (optional): October 12, 2018 23:59

Question 1: Cache size and cache speed

This exercise shows how the performance of our program can be affected by the size of the data it operates with. Depending on whether the data fits into L1, L2, L3 cache, or not at all, we expect different memory access time. Furthermore, in which order we access the elements will also have an effect.

We will demonstrate this by traversing a linked list in the form of a permutation of size N, for different values of N. In other words, we will have an array of integers $a_0, \ldots, a_{N-1}$, where each a_i is a unique value from 0 to $N-1$. We start with the index $k = 0$, and then repeat M times the operation $k \leftarrow a_k$, for some $M \gg N$. This way we minimize memory-unrelated operations and measure virtually only the memory access time[1].

a) Before writing and running the code, check the sizes of L1, L2 and L3 cache by running the following command on an Euler compute node:

   `grep . /sys/devices/system/cpu/cpu0/cache/index*/*`

   The output will contain information from four different `index*` folders, each of which represents one cache level. Two are L1 caches (one for data, one for program code), one L2 and one L3 (both unified data and code). Extract the following information[2]:

   - total size (property `size`),
   - cache line size (property `coherency_line_size`).

b) You are provided with a skeleton code for sampling the execution time for different values of N. The code already selects the values of N and outputs the results.

   Fill out the `TODO` sections marked with *Question 1b* with the code for linked list traversal and time measurement. Use the provided `sattolo` function to generate a random one-cycle permutation. This function guarantees that the permutation is such that all of the N elements are visited. Compile the code with `make`, run with `bsub ... make run` and plot the results with `make plot`.

   What do you observe, do the drops in performance match the cache sizes? Are the transitions smooth or sharp, why?

[1] To be precise, we measure latency of reading a_k from memory (or cache), plus latencies of CPU instructions themselves. Thus, this will only give as an approximation of the memory and cache latencies.

[2] Depending on the node type your program assigned to, you will get different values. Optionally, you might also want to run `lscpu` (in the same run), to get the exact CPU model name.

c) Instead of jumping randomly through memory, initialize the array a such that k goes repeatedly as $0, 1, 2, \ldots, N-1, 0, 1, 2 \ldots$ Compare the performance of this (mostly) continuous access to the random access from the previous subquestions.

How would you explain the result?

d) Previous subquestion was somewhat unfair. We would load a single cache line (of 64 bytes), and then do several jumps for free, because the data is already there. Implement the third variant of the permutation, where k jumps by 64 bytes (how many elements is that?). If that would cause k to go above $N-1$, take the modulo N. It does not matter if not all elements are visited this way, we still do force the CPU to load whole array from memory, as we read from every cache line.

Compare the results with previous two cases. What limits the performance for very large N in this case, and what in the case of a random permutation?

Question 2: Cache associativity

Designing the CPU cache involves a trade-off between the cost (power and area) and performance. Thus, CPU vendors make some limitations that simplify the implementation, while still giving good performance. The purpose of this exercise is to show how performance can be affected if we are not aware of these limitations.

The concept we will look into is *cache associativity*. Consider L1 cache of size 32 kB and a cache line size of 64 B. This means that the L1 cache has a storage of 512 cache lines. However, not all memory addresses can be stored into all of the 512 cache line slots, as that would make the (hardware!) implementation of cache too complex and inefficient. Modern processors implement typically the so-called *n-way set associative cache*, where the cache is subdivided into blocks of n cache lines ($n = 4, 8, 12 \ldots$). Each memory location X can be stored only into a single block. That means it is possible to find $n+1$ memory locations $X_1, X_2 \ldots X_{n+1}$ such that not all of them can be stored simultaneously in the cache.

In this exercise we will measure the performance of memory access when this *cache thrashing* occurs, and show how to modify our code to avoid it.

a) Run the command from Question 1a again and look for `ways_of_associativity` property, which denotes the value n from the text above.

What are the values on Euler nodes for L1, L2 and L3 cache?

b) Implement a memory access pattern that causes cache thrashing.

Assume the following: Two memory locations X_1 and X_2 will be stored into the same cache block if their difference is a multiple of a large power of two (if n is a power of two, $\frac{\text{cache size}}{n}$ should suffice). Concretely, allocate a buffer of $N \cdot K$ doubles, where $N = 2^{20}$ and K is some small integer. This buffer represents K arrays of size N, stored in memory one after the other.

Access the arrays in the following manner: Update the 0th element of each array (by adding a constant), then update all 1st elements, then all 2nd, and so on. Repeat multiple times and measure the total time.

Plot the performance for all K from 1 to 40. Again, you can use `make` to compile, `bsub ... make run` to run and `make plot` to plot the results. Describe the results, can you relate the line shape with the value(s) n of your L1, L2 and L3 caches[3]?

[3] It is possible you will not see all caches, especially if the corresponding n is not a power of two.

c) Modify the problem to avoid cache thrashing.

This is accomplished e.g. by adding a gap of 1 cache line between the arrays. Or, even simpler, by increasing N by $\frac{\text{cache line size}}{\text{sizeof(double)}}$.

Plot the performance for K from 1 to 40, together with the results from the previous subquestions. What do you observe, did the performance increase?

Comment: You might have heard that for performance reasons you should avoid using powers of two as matrix sizes. The reason behind it is the cache thrashing demonstrated here. Of course, depending on how you access the matrix elements, this argument may or may not apply.

Question 3: Anderson's Lock and Mutual Exclusion

In this exercise we will implement Anderson's lock[4] (*ALock*) and visualize mutual exclusion, waiting time and execution time.

ALock is a simple array-based queue lock, which works for arbitrary number of threads and ensures first-come-first-served policy. It is simple in the sense that the maximum number of threads must be given in advance and that it uses spinning (an empty loop) for waiting.

The state of an ALock is defined by:

- an atomic integer `tail`, shared by all threads,
- a volatile boolean array `flag[MAX_T]`, shared by all threads,
- an integer array `slot[MAX_T]`, where each item belongs to one thread.

Each thread `tid` currently waiting in the lock or already executing the critical region has one slot assigned, specified by `slot[tid]`. The slot represents the index of the `flag` array. If `flag[s]` is *true*, then the thread with `slot[tid] == s` is allowed to acquire the lock. Initially, all `flag[s]` are *false*, except `flag[0]`, which is set to *true*.

The `tail` variable specifies the next available slot. Initially it is set to 0. Thus, the first thread reaching the lock will get the slot 0 and thereby acquire the lock (because `flag[0]` is *true*).

In general, when a thread reaches the lock, it will read the current value of `tail` and increment it by 1 (atomically). It then waits until the corresponding flag is *true*. In an unlock, the thread resets its flag to *false*, and sets the flag of the next slot to *true*, giving access to the next thread in the queue.

The flag array is to be treated as a cyclic array, e.g. `flag[MAX_T + 3]` is the same as `flag[3]`.

a) Implement the ALock using the provided skeleton code.

Verify that your implementation is correct by running the provided test code.

Note: On Euler, use `make submit` to submit a job. See this and Makefile for details.

b) Emulate a scenario where multiple threads reach the lock, perform some work W_{inside}, release the lock and then again perform other work W_{outside}. This scenario is repeated 5 times by each thread. Instead of actually doing any work, suspend the thread for some randomly generated number of milliseconds (e.g. 50-200 ms).

[4] Reference: *The Art of Multiprocessor Programming*, M. Herlihy & N. Shavit (see *Array-Based Locks*). Original paper: *The performance of spin lock alternatives for shared-money multiprocessors*, T. E. Anderson (see Table V).

Print[5] the time when each thread reaches the lock (t_A), when it acquires it (t_B), and when it releases it (t_C). Run `make plot` to visualize the events. In the generated plot, x-axis denotes the time and y-axis the thread ID. For each thread and for each repetition, a line t_A–t_B and a line t_B–t_C are drawn.

In order to make the C++ code compatible with the provided plotting script (`make plot`), use the following format:

```
<thread_id> <BEFORE/INSIDE/AFTER> <time since start>
```

For example:

```
0 BEFORE 0.00000
0 INSIDE 0.00010
1 BEFORE 0.00015
0 AFTER  0.01015
1 INSIDE 0.01050
...
```

Is the Mutual Exclusion satisfied, i.e. does it hold that at each instance of time at most one thread is in the critical region (solid line)?

If W_{inside} and W_{outside} take about the same time, what percentage of time do threads waste waiting in the lock? What is the percentage if you decrease W_{inside} by a factor of 10, keeping W_{outside} the same?

[5] You may prefer using `printf` over `std::cout`, as multiple threads will simultaneously be printing to `stdout`. Or, why not use a lock you just implemented? Note: use different instances of the lock for printing and for W_{inside}.