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`).

The following tables includes cache and cache line sizes for different CPU models available on Euler cluster:

Model	L1D	L1I	L2	L3	Cache line
XeonGold_6150 (Euler IV)	32 kB	32 kB	1024 kB	25344 kB	64 B
XeonE3_1585Lv5 (Euler III)	32 kB	32 kB	256 kB	8192 kB	64 B
XeonE5_2680v3 (Euler II)	32 kB	32 kB	256 kB	30720 kB	64 B

L1D and L1I denote two parts of the L1 cache, one for data, one for instructions (the program itself). L2 and L3 cache are unified, they can store both data and instructions.

[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.

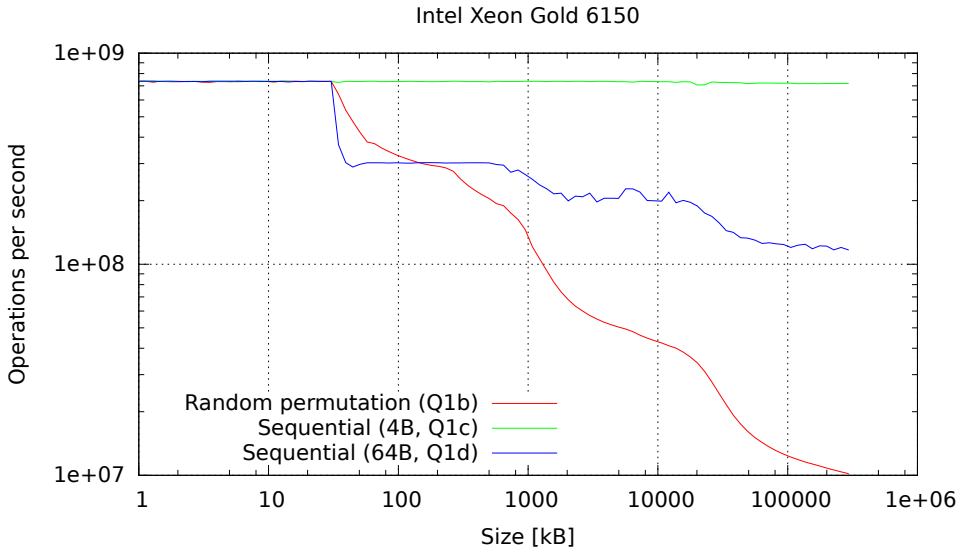

Figure 1: Performance of the permutation traversal for Question 1b, 1c and 1d on an Euler IV (Xeon Gold 6150) node. Note: The plot shows the performance for array size of up to 300 MB, but in the exercise you were asked to analyze only up to 20 MB. See text for more details.

For the interested reader: As can be seen from the table, the fastest cache has only 32 kB (typical values are 32–64 kB). Thus, if our code is memory-intensive, if possible, we should split the data into chunks of size about 30 kB and do as much computation as possible before going to the next chunk (alternatively, one can aim to fit into L2 or L3). This is a basic idea behind, for example, cache-efficient matrix multiplication algorithms.

To select a specific CPU model on Euler, add a flag `-R "select[model==MODELNAME]"`. See `get_euler_cache_info.sh` for more info.

Points:

- 1 **for getting the results (on any Euler node or other CPU).**

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?

The results for Intel Xeon Gold 6150 can be see in the Figure 1 (the red line). As we increase the array size, there is a clear drop in performance at 32 kB and at about 1024 kB, which amount to L1 and L2 cache sizes, respectively. The transitions are smooth because in a random permutation there is always a non-zero probability of jumping to a cache line that is already present in the cache, even though the total array is much larger (especially due to the fact that one cache line fits 16 32-bit integers). If we knew the exact cache policy

determining how old entries are removed from the cache, we would in principle be able to construct a permutation which would force the CPU to load a new line from DRAM after every jump.

For the interested reader: For completeness, the plot was extend from $20\,\text{MB}$ to $300\,\text{MB}$ to show what happens the when array does not fit the L3 cache, which allows us to estimate the latency between CPU and DRAM. Considering that the execution time per jump for small arrays is negligible to those for very large arrays, we can estimate that the memory read has a latency of about $100\,\text{ns}$. Assuming the CPU frequency of $2.7\,\text{GHz}$, this amounts to about 270 cycles. Note that the memory throughput here is only $0.64\,\text{GB/s}$, much smaller than the maximum (you fetch $64\,\text{B}$ [cache line] per element!).

Note: The exact plot shape (for all three lines) may change for different architectures! You should expect only qualitatively similar results. This applies to Question 2 as well.

Points:

- **5 for implementing the jump for-loop,**
- **5 for implementing time measurement,**
- **5 for finishing and running the 1b part of the code and generating plots,**
- **2 for matching the plot shape to the cache size,**
- **2 for explaining smoothness.**

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?

The performance is shown in the same Figure 1 as the green line.

There are two reasons why the performance here is much better, and in fact does not depend on the array size. First, our data element a_k is only 4 bytes large (`sizeof(int)`), thus after we read 1 element from memory, we get another 15 for free. Secondly, the CPU is *prefetching* data from the memory. Namely, CPU detects that we are reading memory sequentially and fetches the data in advance. By the time we want to read a specific array element, it is already present in the cache.

For the interested reader: What this benchmark measures is in fact the total latency of instructions required to perform the operation `k = a[k]`. It takes $1.4\,\text{ns}$ to do one jump, which amounts to 5 cycles[3]. If we relate this to instruction pipelines shown in the lecture, we see that the pipeline is greatly underutilized. The reason is that each jump operation must wait for the previous one to finish. This means that if, in the same for-loop, we add another "jumper" k_2 with `k2 = a[k2]`, we could expect the number of for-loop iterations per second not to drop at all (for small array sizes). Of course, to be sure, benchmarking would be required.

Points:

- **5 for implementing the sequential case 1c,**
- **2 for arguing about the prefetcher (1 pt if you only mentioned that a single cache line has multiple a_k elements).**

[3] If the base frequency of 2.7 GHz is assumed, the value of 3.66 cycles/jump is retrieved, but if boost frequency of 3.7 GHz is assumed the result is 5.02 cycles/jump. Thus, probably the CPU was in the boost mode, as that number is closer to an integer.

d) The 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 the whole array from memory, as we read from every cache line.

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

See the blue line in the Figure 1 for results.

As explained before, for very large N, the random permutation case is limited by the latency to access DRAM. In other words, it measures the cost of a *cache miss*. This case, on the other hand, is closer to benchmarking maximum memory bandwidth, as the CPU prefetcher already "knows" what to read (see below for details).

For the interested reader: For very large N, the number of jumps per second is about $0.11 \cdot 10^9$, amounting to 7 GB/s, which is still several times smaller than the expected maximum bandwidth, which is of the order of 50 GB/s. This should be expected, as we use a single core only. On the Euler IV nodes, if instead of doing `k = a[k]` we simple copy or read large arrays, we can achieve about 14 GB/s (because there are no data interdependencies). The single-core bandwidth is related to many factors, one of which could be the prefetcher and its limitations. For example, see Section 7.5.2 in the Intel Optimization Reference Manual.

The performance of a jump for different array sizes in this case can be partially related to cache latencies (see Agner's Optimization Guide, Section 11.12).

Points:

- **5 for implementing the sequential case with 64 B stride (1d),**
- **3 for saying that random permutation case is limited by the latency to DRAM (or cache miss cost),**
- **3 for saying that the 64 B stride case is related to memory bandwidth.**

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 too expensive. 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 *cache thrashing* occurs, and show how to modify our code to avoid it.

a) Run the command from Question 1a again and look for the `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?

Associativity of the caches on Euler nodes are shown in the following table:

Model	L1D	L1I	L2	L3
XeonGold_6150 (Euler IV)	8	8	16	11
XeonE3_1585Lv5 (Euler III)	8	8	4	16
XeonE5_2680v3 (Euler II)	8	8	8	20

Points:

- 1 **for getting the results (on any Euler node or other CPU).**

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[4]?

Figure 2 shows the performance benchmarks for the Euler IV nodes.

The plot shows a clear drop in performance at around $K = 8$ in the case when array size is a large power of two ($N = 2^{20}$, i.e. 8 MB). However, the performance is starting to decrease even for smaller K. This is possibly due to false dependence between memory addresses on the cache level (see Agner's Optimization Guide, Section 11.12), where it is not possible to simultaneously write and read at addresses differing by a multiple of 4096 (which may not even be stored in the same cache block!). Another example for a false dependence problem, in a somewhat different setup, can be found here.

For the interested reader: Notice that the $K = 1$ case is slower than $K = 2$. The reason here is simply the overhead of the for-loop itself. A for-loop with a single iteration is more costly (per iteration) than a loop with more than one iterations.

Points:

- 5 **for allocation and deallocation of memory,**
- 10 **for implementing the array traversal,**
- 5 **for time measurement code, running and plotting,**
- 2 **for relating the drop in performance to (L1) cache associativity.**

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

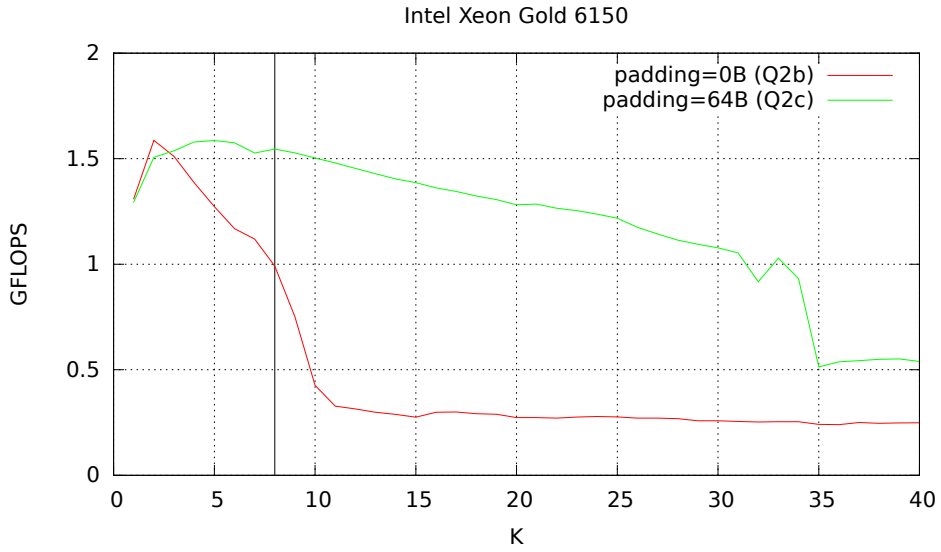

Figure 2: Operations per second vs. K for $N = 2^{20}$. The red line shows the performance when no padding is added (when stride is exactly 8 MB), while the green line shows the performance with 64 B padding. The black line denotes $K = 8$. Results are shown for the Intel Xeon Gold 6150 nodes (Euler IV).

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.

As shown in the Figure 2 (green line), the performance recovers when padding is used.

For the interested reader: The performance still drops greatly at $K \approx 35$, which does not match the associativity of L2 nor L3. Counting cache misses with `perf stat -ddd ./associativity` shows that the number of L3 cache misses starts to rise at $K = 33$ and reaches 2x at $K = 35$ when compared to the $K = 32$ case (L1 cache misses are the same, while information about L2 is unavailable). The reason is that the mapping from memory to L3 cache blocks is not that simple as the one described in the exercise.

Moreover, Euler II nodes show a different behavior than Euler IV or Euler III. Namely, there the performance does not recover fully for 64 B padding. `perf stat` shows that L1 cache misses start to increase for $K \geq 9$. For padding of 512 B we get the desired speed-up.

We tested not only E5-2680 (Euler II), but also E5-2650/70/90. E5-2650/70 behave similarly to Gold 6150 (Euler IV), while both E5-2680/90 deviate from it. This is particularly confusing as all these four models should have exactly the same microarchitecture. The take-home message is that if you are optimizing your code (trying to reach the ceiling of the roofline model), you should do the benchmark on the same machines where you plan to run your computations.

Lastly, you might notice that if you ran your code on Euler II and Euler III, even for $K = 1$ the case of no padding is slower. This is due to the clock rate, which is not at its maximum when the program starts. Switching the order of the no-padding and 64 B padding case causes the 64 B padding to appear slower.

Points:

- **5 for the running and plotting the part 2c,**
- **1 for showing that the performance recovers.**

Question 3: Anderson's Lock and Mutual Exclusion

In this exercise we will implement Anderson's lock[5] (*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.

The crucial part of the implementation is handling the variable `tail`. In the `lock` function we need to atomically read the variable `tail` and increase its value by 1. This can be accomplished by declaring `tail` as `std::atomic<int>` and then accessing it as `int s = tail.fetch_add(1);`. That way, `tail` is increased by 1, and `s` contains its value prior to the increment. Because the slots are values from `0` to `MAX_T-1`, you are

[5] 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).

supposed to take the modulo of `s` with respect to `MAX_T`, e.g. `s %= MAX_T`. Note that this modulo is done *after* reading/modifying `tail`. Performing the modulo operation on `tail` directly is probably possible, but very hard to implement, for an arbitrary value of `MAX_T`.

For the interested reader: In this exercise we did not require you to take care of the overflow of `tail`, which may happen if we invoke the `lock` function more than $2 \cdot 10^9$ times. To handle this case, simply define `tail` as `std::atomic<unsigned>` and change `MAX_T` to a power of two, e.g. to 128. Note that this way we speed up the lock by not having a modulo operation (it is replaced with an AND operation)!

Points:

- **3 for declaring the ALock member variables,**
- **3 for initializing the lock,**
- **10 for implementing the lock function (5 pt max. if you don't handle the variable tail properly),**
- **4 for implementing the unlock function.**

Make sure you are really tested the lock with multiple threads. Compare with the provided solution code. If your implementation differs much, is it really correct?

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).

Print[6] 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?

[6] 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}.

You may have encountered that some `INSIDE` events happen just before `AFTER` events. This happens when a thread X releases the lock, thread Y acquires it, prints that it is inside, and then thread X continues and prints that it is outside (after). Thus, this is an artifact of the logging, not the lock.

As expected, if we draw a vertical line on the plot, it would intersect at most one critical region (or touch two), confirming mutual exclusion.

If the duration time for W_{inside} is W_{outside} is chosen to be a randomly generated value between 50 and 200 ms, the waiting time is about 40–50%. If the W_{inside} is shortened by a factor of 10, the waiting time drops below 1%, to about 0.6–0.8%. This demonstrates that it is very important to keep the critical regions as short as possible.

For the interested reader: Depending on the size of the critical region, we may use different locks. For example, spinning locks such as this one are preferable if we know the duration of the critical region is very small. If the critical region is long, insteading of wasting resources by spinning the threads in a loop, the thread may go to sleep and yield the resources to other threads (not necessarily of the same program). If the critical region is extremely small, i.e. a single instruction, we use atomic operations wherever possible. As the exact CPU instructions to accomplish this may change from architecture to architecture, for portability we can use implementations such as `std::atomic`. If the target architecture cannot perform the operation atomically using a single instruction, `std::atomic` falls back to a lock. In LLVM (clang), a spinning lock is used.

Points:

- 5 **for the log function implementation (only** 2 **pt if it is not thread-safe),**
- 10 **for implementing the BEFORE/INSIDE/AFTER events,**
- 2 **for noting that critial regions do not intersect,**
- 10 **for measuring wait time (**5 **pt for the first number,** 5 **pt for the second).**