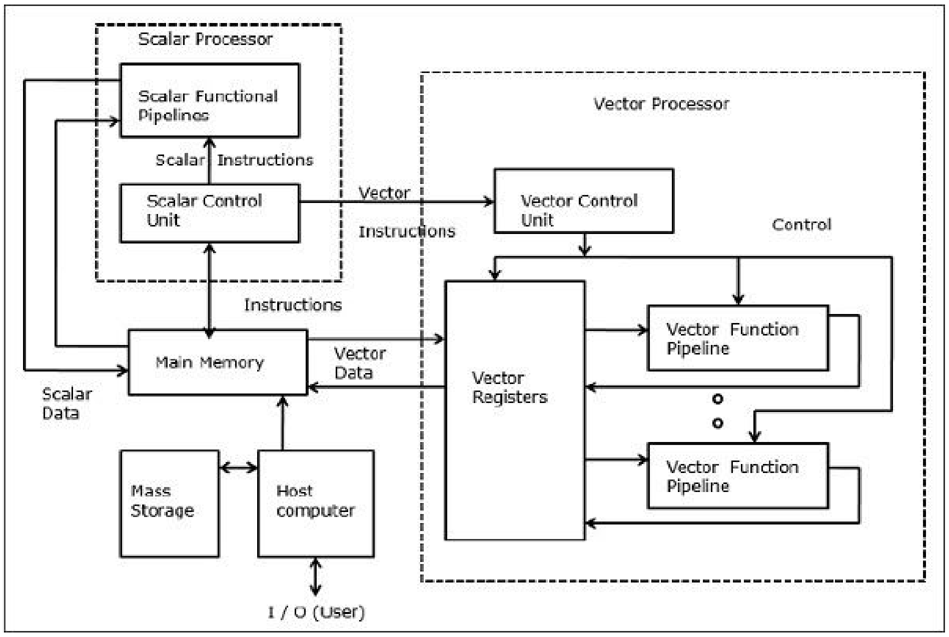
Parallel Computer Architecture



Winter 2020

Theodoros Michailidis - A53318129

Documentation

For this project, I implemented a simple pointer-chasing C++ program, with a random access pattern, on a void** array. I used pointer-chasing, in order to serialize the loads (since a load instruction uses the result for the previous load instruction), which resulted in having very low ILP. I used a random access pattern in order to negate the optimization from the cache prefetcher. These two implementation choices reduce the associated optimizations, allowing us to measure a more accurate latency.

The rest of my implementation's details are the following:

Randomness

In order to optimize the access pattern, it is important to have: 1) a sparse access pattern, 2) a small number of cycles (it is inevitable to have cycles), and 3) as big as possible cycles' length. All of these contribute in having a big memory footprint that surpasses the size of the different caches, when needed.

I tried, mainly, two different approaches, and some variations of them:

- The first was to populate each element in the array with the address of the next element and then use the std::shuffle function to shuffle the contents of this array. This resulted in having pointers back and forth in the whole array. (function in my program: **shuffle_pointers()**)
- The second was to give in each element an address of another random element, where the index of the former was lower than the index of the latter. The intuition behind this was to reduce the number of cycles, since all the elements were pointing forward (i.e. to a bigger-indexed element), and only the last element was pointing to the first one. (function in my program: **rearrange_pointers()**)

After careful experimentation, I used the first one, since it produced a sparser, bigger-sized cycle.

Timing measurements

For the latency measurement, I used the std::chrono::high_resolution_clock that offers very accurate timing methods. To get the average latency in multithreaded executions, I add the individual measurements (stored in an array at the end of the threads' execution) and divide with the number of threads.

Limitations

The results from the experiments show that my program scales to up to 16 threads. This is consistent with the fact that I run all the experiments in an 8-core (16 threads) machine. Also, my program is dependent of various C++11 features (such as std::shuffle, std::chrono, std::threads, etc)

Execution details

For the r/w part, I implemented the following routine (Read-Modify-Write) to avoid the compiler optimizing out the write on the array:

1. I initialized a temp variable to the current pointer to the array.
2. I used "bitwise and" between the value pointed by the temp variable and 0xffffffffffffff (since the values are 8 bytes).
3. I assigned this new value to the main pointer that I used to iterate the array.
4. I proceeded to the next element through the pointer.

The aforementioned routine can be found in the functions **do_read_pointer_chasing()** and **do_rw_pointer_chasing()** in my program.

The program requires the following arguments: number of threads, number of loops, mode (0 read or 1 for r/w) and total bytes for the array.
The usage is the following:
[*exe*] [-*t threads_num*] [-*n number_of_accesses*] [-*b total_bytes*] [-*m mode (0 for read only, 1 for read/write)*]

These arguments are parsed by the function **parse_args()** in my program, and can be arranged in any order.

Additionally, I have implemented:

- A makefile
- A python script (**run_all.sh**) for bulk execution, with a variable number of loops, threads, array sizes (in bytes) and both modes. In the start of the script, it automatically cleans and compiles my program. In my evaluation I used 100,000,000 iterations and variable loops, threads, array sizes for both modes.

CPU Info

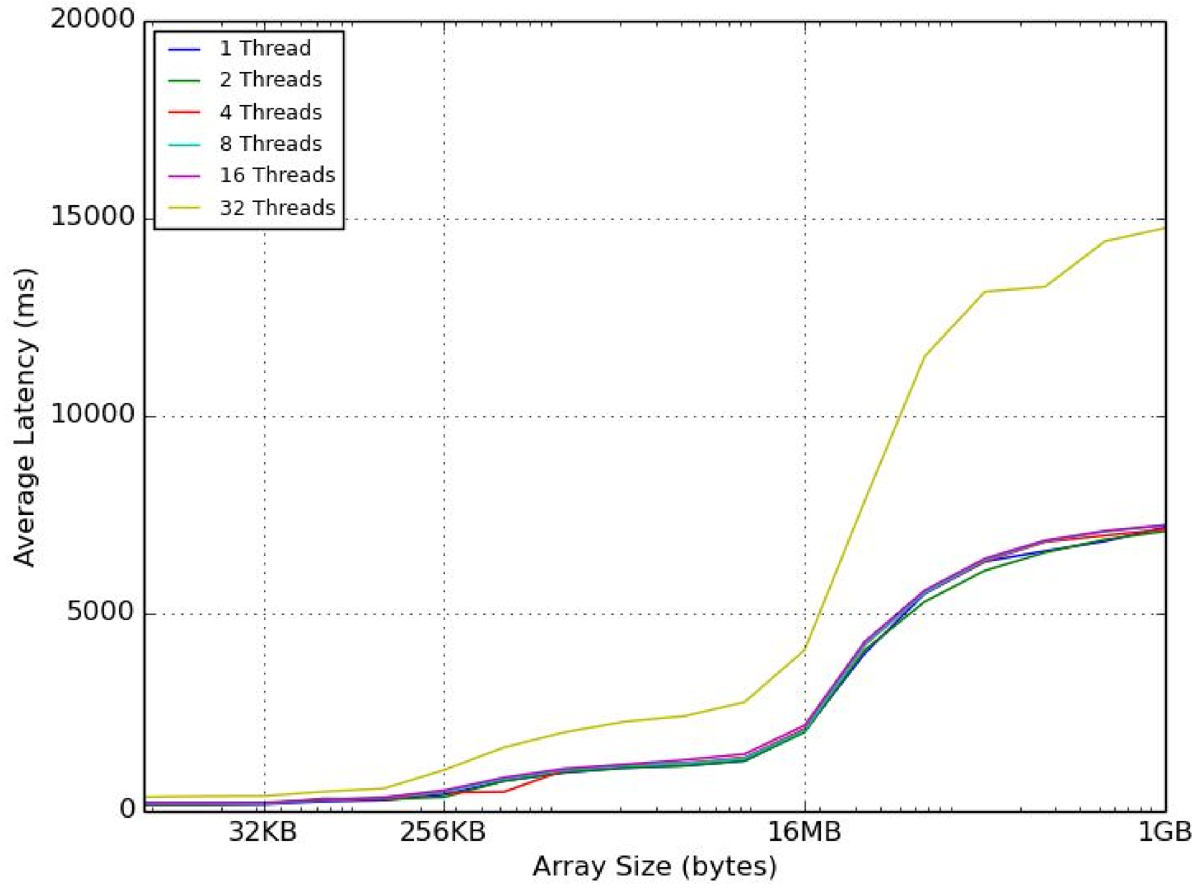
The CPU of the machine (i9900k) that I used had the following cache parameters:

Cache Type	Size	Associativity	Line Size	Max Threads	Inclusive	Invalidates Lower
L1 Data	32KB	8	64B	2	No	Yes
L2 Unified	256KB	4	64B	2	No	Yes
L3 Unified	16MB	16	64B	16	Yes	No

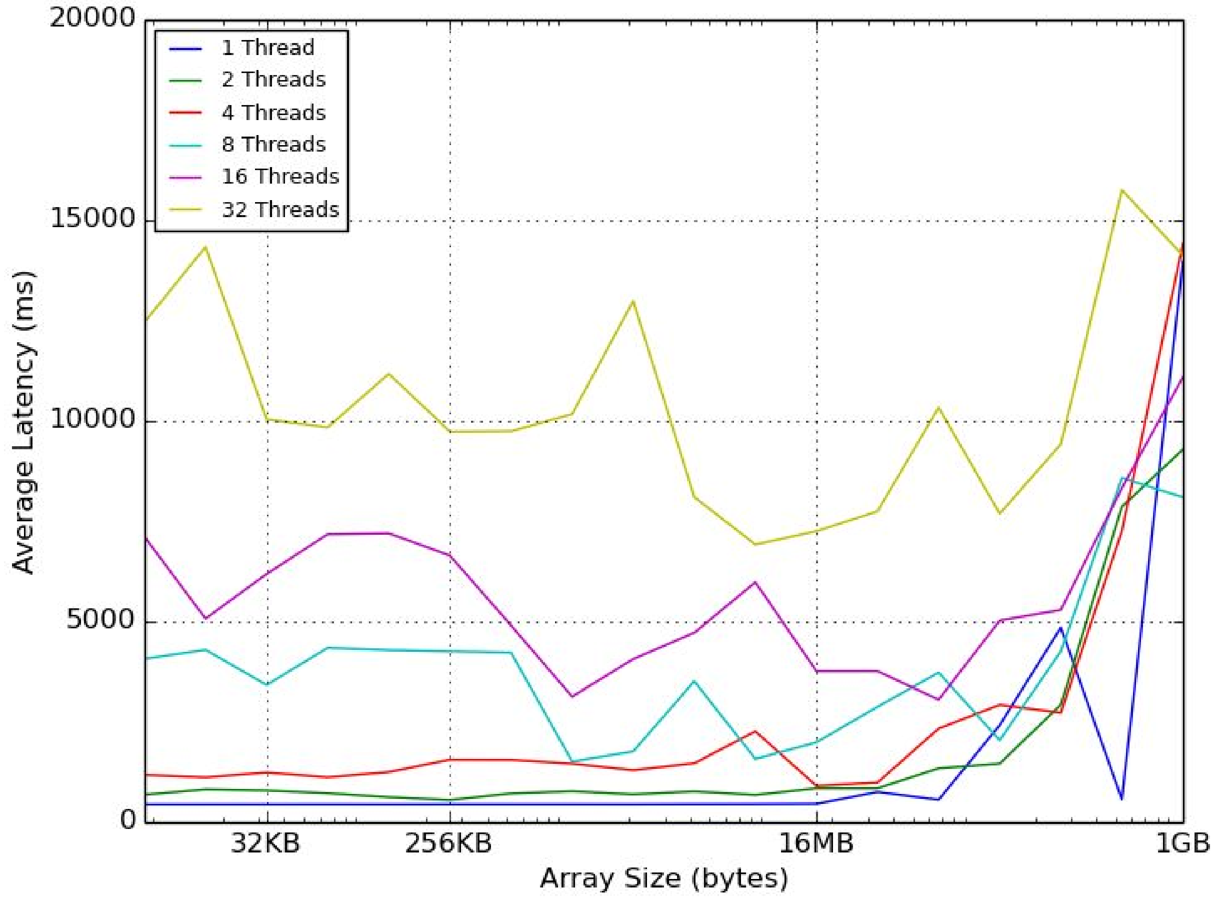
Intel uses the MESIF cache coherent protocol.

Graphs

Read only graph



Read write graph



Analysis

Both graphs

When using 32 threads (and above), the latency increases significantly. This can also serve as an indication for the maximum number of threads that the processor can handle equally efficiently at any point.

Read only graph

We can see 3 fluctuations: the first one around 32KB-64KB (even though due to the growth of the latency size, this is not very visible), the second around 128KB-512KB, and the third one around 8MB-16MB. This tells us that most probably, the machine that we used has 3 caches, since the latency can be divided into 4 distinct regions.

If we had to guess, it would be a little bit difficult to accurately determine the size of each cache. That stands, mainly, because the memory latency depends from the access pattern that we use. In the worst case, if we have a very short cycle that accesses only e.g. 8KB of data, regardless of the size of the array, then all memory requests are served from the L1 cache. In the best case, we access the whole array with poor spatial locality. This graph confirms that the random access pattern that was used is nearly perfect.

Another, minor, reason for this small disparity between the actual cache sizes and the guessed ones can be the fact that the L2 and L3 caches are unified, which leaves less room for data in these caches. Additionally, the L3 cache is shared between all cores. This disparity is also affected by other programs' instructions and memory accesses.

Read/Write graph

First of all, the read/write pointer chasing function uses more instructions, which justifies a slightly increased latency, since we have 2 accesses per iteration, instead of 1 (as in the read only pointer chasing).

Additionally, this graph is far less stable; it is clear that as the number of threads increase, the corresponding lines become more unstable. The reason behind that is cache coherency and the associated overheads, including cache invalidations, requests, responses or even multiple redundant responses. More threads lead to more writes to the shared array, and this leads to more invalidations and changes in the coherency-related cache states.
Another reason that causes this instability, as the number of threads increase, is the fact that each core supports two threads. This means that changes that originate from the threads of the same core, are shared, since they share the same L1 and L2 caches. Thus, as the number of threads increases, the possibility of another core having a valid copy of the data increases too.
To this instability adds a little bit the fact that we have doubled the number of accesses per iteration, which again, leads to more changes in the coherency-related cache states.

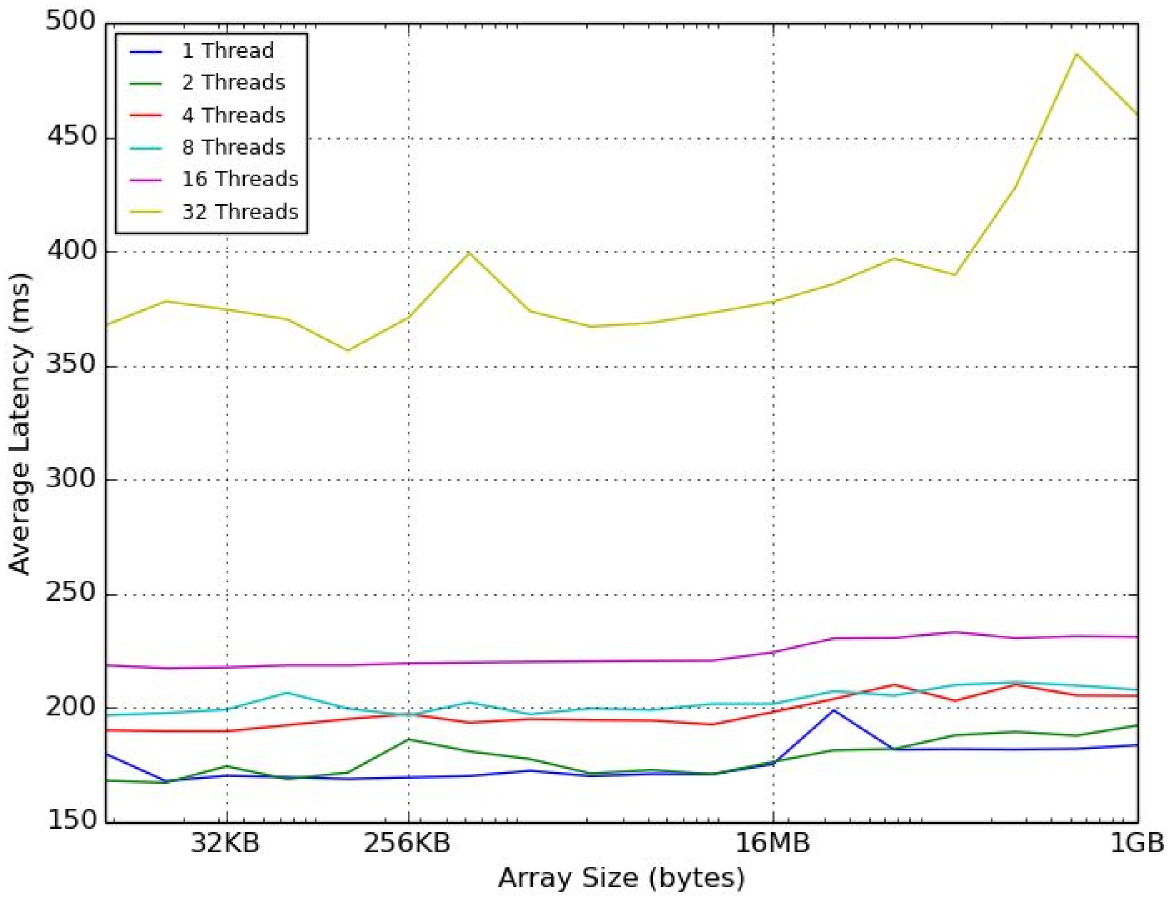
The read program's results could be validated by using a perfect random access pattern that overcomes the sizes of the caches and accesses each cache line once before it gets evicted.

I could validate the results of the read/write program by using the following approach: I could use three threads, two from the same core, and one from a different one. The first one would read-modify-write the shared contents, the second one would read only the contents, and the third one would also read only the contents. I expect that these three threads would exhibit different latency, with the second one having the best, since it would only read the contents from the shared caches (no invalidations), and the third one having the worst, since it would have the most invalidated cache lines.

Bonus

For this part, I disabled the shuffling and ran again the same python script for 100,000,000 iterations. That means that the accesses are serial, not random.

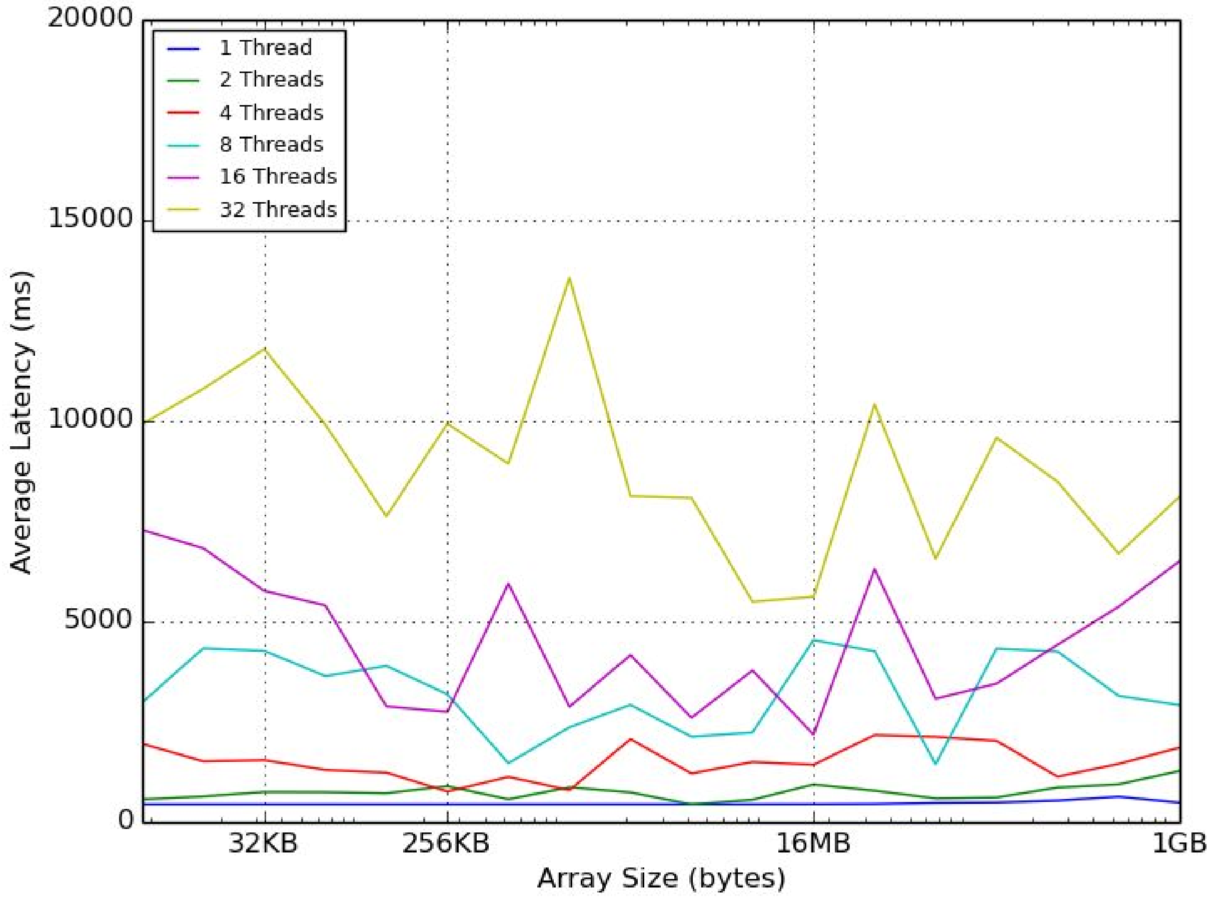
Read only graph



From this graph, we can still see that for 32 threads, the program exhibits increased latency. The interesting part is after approximately 32MB, the latency is increased significantly, which is probably caused by the fact that the L3 cache can be accessed by up to 16 threads at the same time.

For 1 up to 16 threads it is clear that the average latency is nearly the same; this is the result of the prefetching and cache line locality. Note that the Y-axis in the previous graphs varies from 150 to 20,000 milliseconds, while this graph's Y-axis varies from 150 - 500 milliseconds.

Read/Write graph



This graph exhibits almost the same kind of instability (as the random r/w execution), for the same reasons. The only difference between the random and sequential r/w is the spike after accessing mostly the L3 cache. I suspect that, again, the prefetching and reuse of cache lines that are brought to higher level caches, alleviate the cost of accessing these common data locations.

General Note

Initially, I enforced false sharing by accessing different data on the same cache lines by different threads (and varying the cache line for all threads in each pointer hop). The problem with this approach was that it was impossible to find a good way to greatly randomize the accesses and keep the false sharing condition between the threads, at the same time. All these approaches led to smaller cycles with narrow patterns.

Instead, I chose to make every thread have the same pattern. When used the same methods between these two approaches I had the same results. This is not false sharing per se (since every thread accesses the exact same memory locations), but it allowed me to use wider, bigger cycles, that exhibit more accurately the effects of cache coherency.