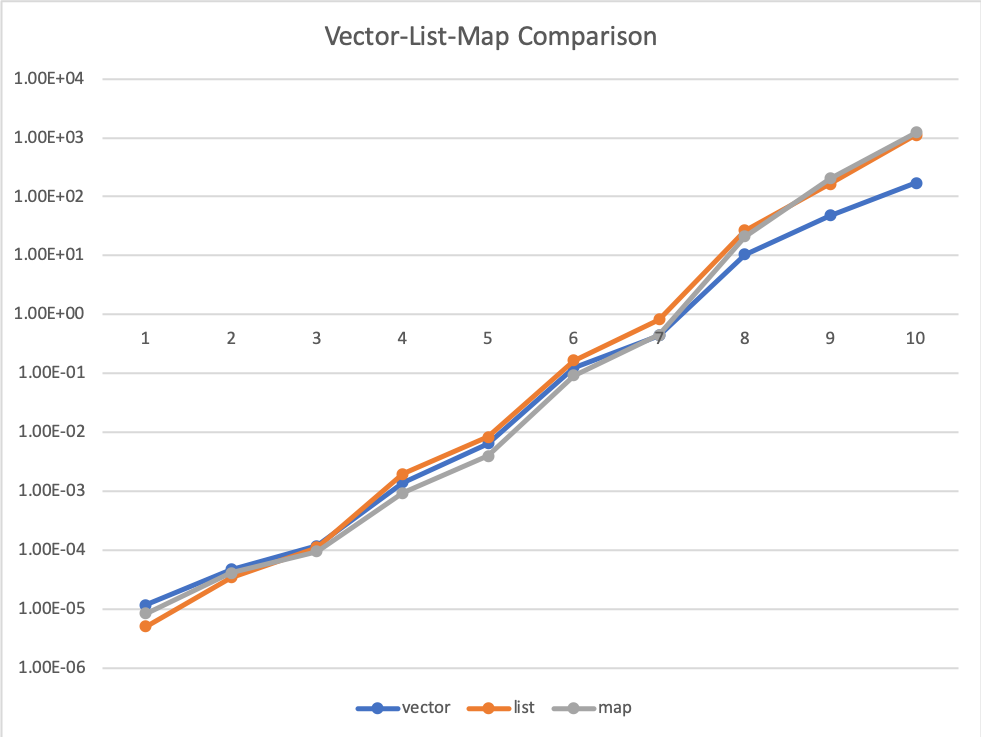
James Yang

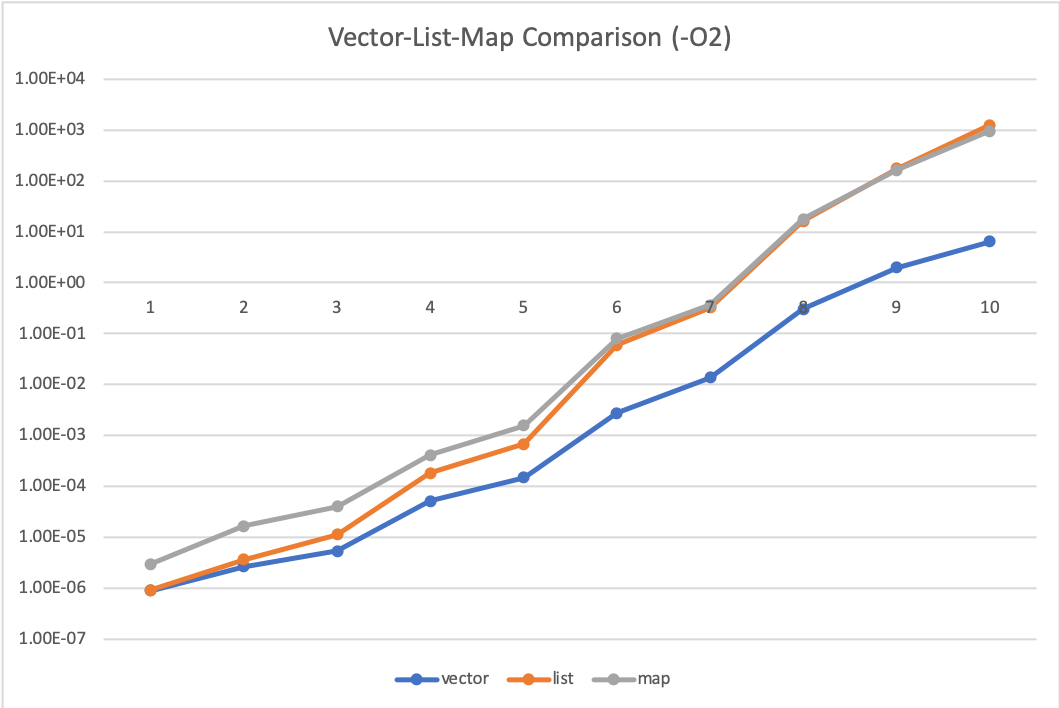
UNI: jy2816

March 7, 2020

Vector List Map Benchmark Writeup (with -O2)

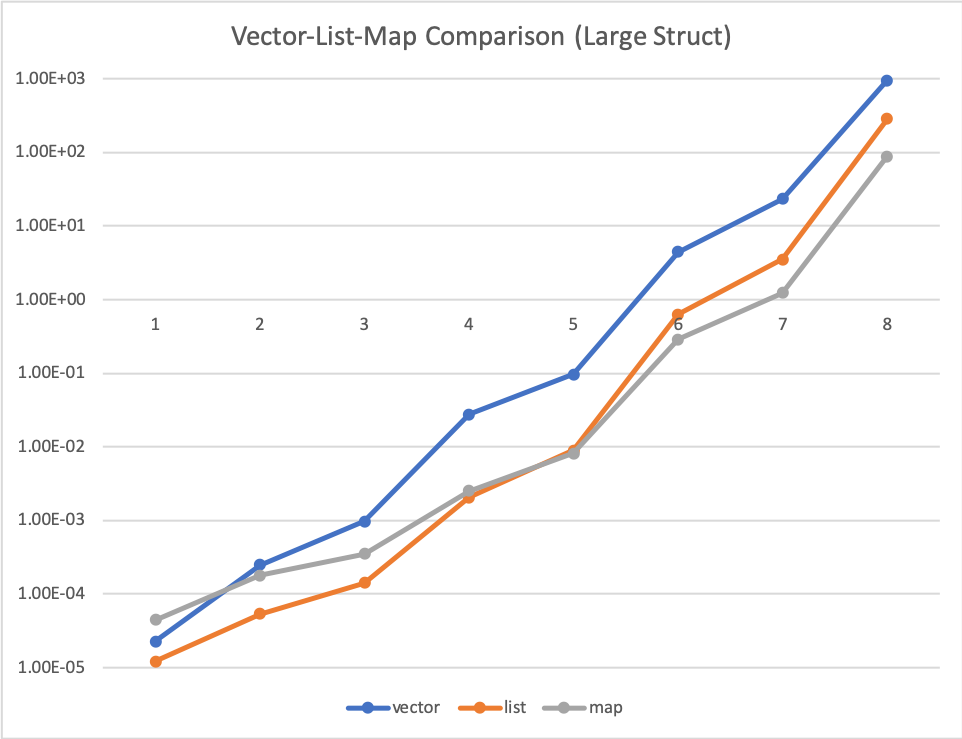
This is the updated write-up using -O2 flag. Turns out there truly is about 25 times difference for vector. Nonetheless, the general trend is still the same as before, just exaggerated more. I actually found the comparison between -O0 and -O2 interesting in-and-of-itself. I am attaching figures of the previous unoptimized benchmark results along with the optimized ones.

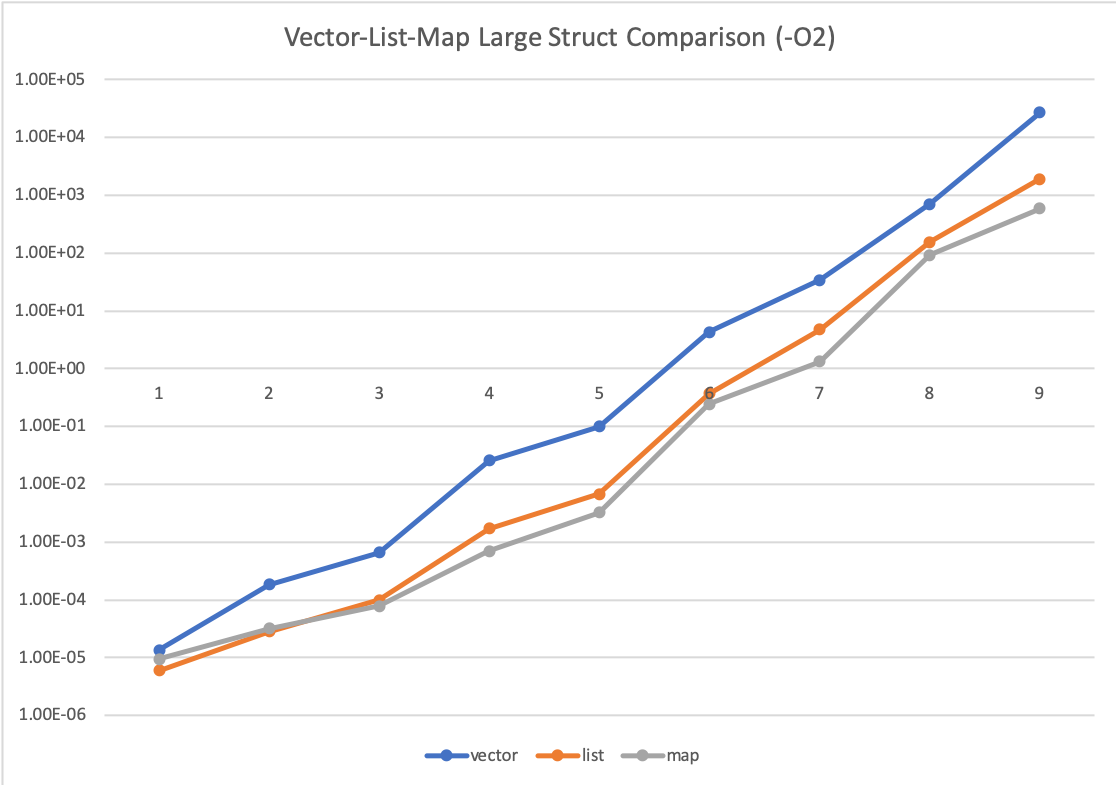




The above figures show the first benchmark results for vector, list, and map without and with optimizer. The graphs are in log-scale (base 10) of number of seconds vs. container size. The number of elements ranged from [10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 200000]. The results show that vector actually performs consistently the best with the optimizer. Without -O2 flag (first figure), we see that vector performs the worst for smaller sizes most likely because vector resizes more frequently and every resize allocates a new chunk of memory that will lead to cache misses. Furthermore, inserting into vector causes shifting of the elements that requires copying. However, with the optimizer (second figure), it performs the best and this probably occurs because vector may be now optimized for smaller sizes to store an array directly on the stack. List is relatively cheap in terms of allocating memory and inserting (O(1)) - the pointer indirection in the for-loop is too few to cause much overhead in this case, which is why it performs close to vector. Map takes a little more time because not only does it require the O(1) allocation, but also the log(N) reordering that happens with lots of cache misses because each node of the RB tree is scattered throughout the heap. This is also why vector still performs better than map, even though there are for more operations (copying) for vector. Vector only shifts elements by one in a sequential order. To visualize better, if we assume the cache holds 10 elements of the vector at a time, once we copy an element in position 0 to position 1, by cache locality, we load the 11th element. This way, we have incredibly high cache hits, which reduces the time.

The previous analysis of why vector is far superior holds more generally with N sized containers. One interesting observation is that map consistently performs the worst with optimizer. And for large N, it performs as worse as list. Since both list and map have the same poor memory layout (with nodes scattered throughout heap), they both lose the cache benefit. However, now we must compare their algorithms. Map uses a log(N) algorithm to keep its data sorted, and therefore should have better asymptotic performance. Note that we clearly see for N < 100k, map performs worse than list. Starting from 100k, map begins to perform much better. Although the log scale does not clearly show it, for N = 200k, list took about 1260s whereas map took 955s, which is about 25% improvement. At N = 100k, list took about 175s whereas map took 163s, which is only about 7% improvement. This suggests that with N > 200k, the time difference will explode even more. It may be the case that for lower sizes (N < 100k), restructuring the map takes far more indirections (constant term of log(N) is actually high) than the O(N) look-up for list (smaller constant term). The turning point seems to be around 100k where list for-loop has more pointer indirections than map during reordering.





The above shows the same benchmark but with a large struct without and with optimizer. I could not benchmark beyond 100k because it just took way too long (ran for a day with 100k). We created a struct to contain an array of 1000 integers and also contain an integer representing the randomly generated number. This exercise essentially mimics the structure of list in vector by separating out the actual randomly generated integers in each element by 4000 bytes. Now we lose the cache benefits for the vector because each element is probably too big to fit into the cache, so we cannot load the next elements into the cache. To get a feel for how terrible vector performs, with N = 100k (x = 9), vector finished in 26884.1s whereas map finished in 588.269s, which is about 98% improvement for map.

It is difficult to feel the gap between the list and map performance, but at about N=50k, there is already about 40% improvement with list finishing in 155s and map finishing in 93s, which is huge. One question I was wondering was why this turning point occurs at a lower N than in the original benchmark. I think the difference occurs because of where the data lives. Specifically, I have high suspicion that the RB tree nodes do not contain the data directly but rather keep a pointer to another heap-allocated data. Hence, when the node structures change due to reordering, we are accessing smaller data structures. Of course, these RB tree nodes are comparable in size to list nodes, which do contain the data directly, when the data is simply an int. But list nodes are much larger when data is a large struct. As a consequence, there may also be slightly better caching for map since we may be able to store a couple of RB tree nodes, but probably cannot store even a single list node with large struct data.