Survey of Optimized Functional Data Structures

As computers become highly parallelized and memory becomes faster and cheaper, we must begin to consider new programming paradigms, like functional programming, to tackle the challenges that arise from these systems. Functional programming has for a long time been a programming paradigm of interest to computer scientists. It offers many benefits like safe multithreading, less error prone code and even the possibility of automatic parallelism. One of the challenges is that it uses high amounts of memory. Using functional data structures that are optimized can reduce the memory usage significantly. In our research, we created four optimized functional data structures and compared its efficiency to similar non-optimized functional data structures and non-functional data structures.

Moore's law states that chip performance doubles roughly every two years. While this has held true for the past 50 years, semiconductor industry now believes this is coming to an end. In order to keep making faster and faster computers, we will have to instead increase the number of cores. Computer engineers could opt for manycore processors. Instead of relying on multiple layers of cache computers could be optimized for highly parallelized computing. As memory speed increases, the bottle neck of the program would be the throughput instead of the memory. This would put the burden on programmers to create efficient parallel programs. This would result in code becoming susceptible to bugs and race conditions.

Functional programming provides many benefits that allow programmers to tackle this possible complexity increase. Creating programs that are stateless result in code that is easier to maintain, easier to reason about, less bug prone and and eliminates most of the issues and complexity that result from multi-threaded code. In fact, the programs could be atomically parallelized. For example, Swift-T is a scripting language for high performance computing that can automatically distributes tasks across hundreds of thousands of nodes without any explicitly paralyzed code[?].

One of the challenges of functional programming is that is can use a lot of memory. If you cannot change a data structure's state, programmers then have to make a copy of the data structure every time they wish to make a change. Functional programmers have created many optimized versions of functional data structures that significantly reduce the memory usage by sharing parts of data structure that are unchanged. From the perspective of the client it looks like they created a completely new instance of and data structure, but in reality the copies share much of the same memory. Since the data structures are cannot be changed we are guaranteed to avoid any side effects that would normally arise from the data structures sharing memory. We will look at four optimized functional data structures and compare to un-optimized version as well as non functional counterparts as well. These four optimized functional data structure are the list, the banker's queue, the red black tree and the bit vector tri.

A list is one of the most basic optimized functional data structure. It is a FIFO container that looks very much like a linked list and has very memory efficient push and pop functions. Whenever an element is pushed, instead of returning a copy of the list with the element added, a new head node, containing the element and pointing to the original list, is return. Likewise, when the top element is removed, the next node on the list is returned as if it was a completely new list without the top element. Notice that neither operation alters the state of the original data structure, yet the "new" list returned looks exactly as the client would expect. The resulting data structure is one that does not use any additional memory compared to a non-functional stack and maintains constant time push and pop. Alternatively, a non optimized version would have to create a new copy of the list for both push and pop. This would result in not only a lot of memory usage, but also relatively slow push and pop.

The banker's queue