Optimization of Scalable Concurrent Pool Based on Diffraction Trees

Alexandr Anenkov, Alexey Paznikov
Saint Petersburg Electrotechnical University "LETI"

apaznikov@gmail.com

Abstract. The work proposes the implementation of scalable concurrent pool based on diffraction trees. Developed pool ensures localization of addresses to shared variables to maximize its throughput. The pool provides large scalability of multithreaded programs compared with similar implementation of pool based on diffraction trees.

Keywords: multithreaded programming, diffracting trees, lock-free data structures, scalability, thread-safe pool

Full paper: https://goo.gl/CtGb3X (in Russian)
https://www.dropbox.com/s/xlzzbd7mh9d0bcw/anenkov-paznikov-pool.pdf?dl=0 (in English)

This paper proposes the novel approach for concurrent lock-free pool implementation based on diffraction trees and the methods for its optimization for constant number of active threads. The approach is based on localization of access to tree nodes and thread local storage utilization. The proposed approach increases the throughput at high and low workload and provides good level of FIFO/LIFO-order of operation execution.

We developed the pool LocOptDTPool in which each node of diffraction tree contains two arrays of atomic bits (for producer and consumer threads) of length $m \leq p$ instead of two separate atomic bits (figure **??**). Nodes of each next level of the tree contains twice less arrays compared with previous level. Each thread addresses to corresponding bit in the array to ensure localization of references to atomic bits in the tree nodes. Furthermore comparing with elimination array method this approach reduces the overheads, connected with the references to the elements of elimination array and active waiting of paired thread.

Each time when a thread visits tree node it chooses the element in the atomic bit array by the value of hash function. Each core $j \in \{1, 2, \ldots, p\}$ corresponds the queues $q_j = \{j2^h/n, j2^h/n + 1, \ldots, (j + 1)2^h/n - 1\}$. Let there is thread i, affined with the core j ($a(i) = \{j\}$). Then all the objects pushed (popped) to the pool by this thread are distributed among the queues q_j destined for the objects arriving from the threads affined with the core j. This approach reduces the number of cache misses thanks to reference localization to shared variables.

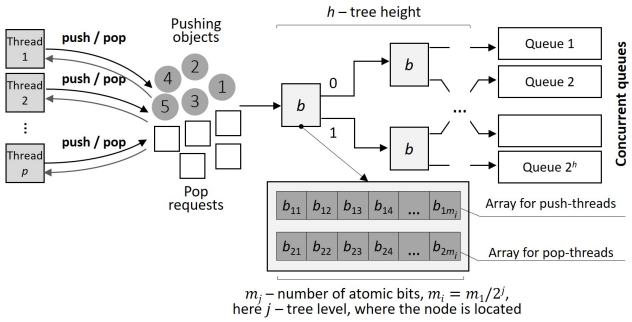

Fig. 1: Optimized pool LocOptDTPool based on diffraction tree

We also developed scalable concurrent pool TLSDTPool. The pool is based on allocation of tree node bits in thread-local storage (TLS). This approach decreases access contention to shared bits in tree nodes. The main idea of proposed approach is the allocation of structure BitArray in thread's TLS. This helps to avoid heavy atomic operations while addressing to array *bits* in the BitArray structure; *bits* now becomes regular boolean array.

The experimental results for throughput of implemented pool based on arrays of atomic bits in tree nodes are represented on figure **??**.

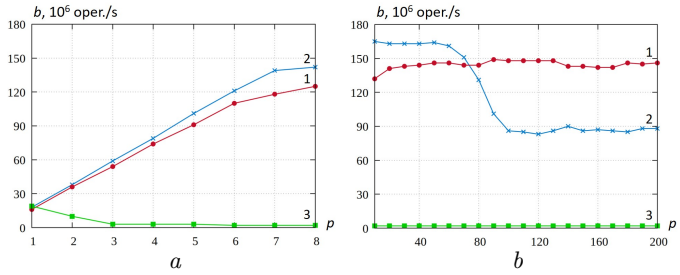

Fig. 2: Throughput of pool LocOptDTPool
1 – LocOptDTPool, non-blocking queues Lockfree queues from boost library,
2 – LocOptDTPool, concurrent queues based on PThread mutex,
3 – single non-blocking concurrent queue Lockfree queue from boost library.

Implemented pools scales well for large number of threads and increase the throughput as the number of threads comes near the number of processor cores. Maximal throughput was obtained for number of threads which equals to number of processor cores or slightly exceed it. For large number of threads lock-free queues in the pools LocOptDTPool TLSDTPool increases the throughput, com-

paring with lock-based concurrent queues (figures **??**b). In all cases the efficiency of single concurrent queue is much less than the efficiency of developed pools.

Thus the pools maximize throughput in multithreading programs compared with similar pool implementation based on diffraction trees. The highest efficiency of the algorithms is achieved with the number of active threads equals to the number of processor cores in the system. Increasing of tree size in the pool does not reduce the pool throughput.