Methods for Optimization of Concurrent Pools Based on Diffracting Trees for Multicore Shared Memory Systems

Alexandr D. Anenkov[1], Alexey A. Paznikov[2]
Computer engineering department
Saint Petersburg Electrotechnical University "LETI"
St. Petersburg, Russia
[1]anenkov.ru@gmail.com, [2]apaznikov@gmail.com

***Abstract*— In this paper we represent the implementations of scalable concurrent pools based on diffracting trees. Developed pools localize access of parallel threads to shared variables in order to maximize data structure's throughput. We analyze the efficiency of developed pools. The pools ensure high scalability in multithreaded programs, in comparison with existing related pool's implementations based on diffracting trees. We give the recommendations for using of pools and the results of experiments on multicore systems.**

Keywords— multithreading; diffracting trees; lock-free concurrent data structures; scalability; concurrent pool

I. INTRODUCTION

With the increasing of processor core number in modern computer systems (CS) [1] the problem of scalable access to shared data structures is urgent today. One of the most widely used data structures today is concurrent pool. Pool is unordered object collection, realized push and pop operations [2]. Pools are widely used in producer-consumer model implementation in multithreaded programs. In this model one or multiple producer threads produce the object which is used by consumer threads.

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 acceptable level of FIFO/LIFO-order of operation execution and is characterized by low latency of tree traversal.

II. RELATED WORKS

One of the most straightforward approach for pool implementation is utilization of concurrent queues. Existing implementations of pools based on locks [3, 4] provide high performance on low operation rate, but doesn't scales for large number of threads and high intensity of addresses to pool. Work-pile and work-stealing methods [5, 6] are not efficient at low rate of pool address.

Concurrent pools based on lock-free linear lists are widely used for increasing of scalability. One of the most common ways to decrease access contention of parallel threads to shared memory is elimination back-off method [7, 8, 9]. One of the alternative approaches for concurrent pool is the implementations based on elimination trees [10]. Lock-free concurrent lists provide predictable execution time of operations, but the heads of lists become the bottlenecks, which increase the access contention and reduce cache-memory efficiency. The method of delegation of operation execution to dedicated threads [11] has the same drawback. One of the most promising approaches for reduction of access contention of parallel threads is diffraction trees [12]. The work [13] proposes possible implementation of concurrent pools based on diffraction trees with using of elimination back-off method.

One of the drawbacks of the existing pools are additional overheads, connected with active waiting in the elimination array and atomic variables synchronization in each level of diffraction tree, which significantly increases the complexity of tree traversal from the root to the leaves. The efficiency of diffraction tree is decreasing with increasing of its size. Moreover, the pool [12] violates the FIFO/LIFO operation order and does not consider the insertion and deletion of elements by single thread. The other drawbacks are the need of selection of parameters (time of waiting for the complementary operation, acceptable number of conflicts and so on).

The papers [14, 15] propose the optimized self-tuning reactive diffraction tree to reduce the overheads described above. The size of that tree is determined by the current level of access contention of the threads to the leaves. Nevertheless, these pools have heavy overheads due to synchronization in the auxiliary arrays and in the tree nodes.

The reported study was funded by RFBR according to the research project № 18-57-34001, by Russian Federation President Council on Grants for governmental support for young Russian scientists (project SP-4971.2018.5). The paper has been prepared within the scope of the state project "Initiative scientific project" of the main part of the state plan of the Ministry of Education and Science of Russian Federation (task № 2.6553.2017/BCH Basic Part). The paper is supported by the Contract № 02.G25.31.0149 dated 01.12.2015 (Board of Education of Russia).

978-1-5386-5824-6/18/$31.00 ©2018 IEEE

III. CONCURRENT POOL

A. Concurrent pool based on diffraction tree

Let there is multicore computer system (CS), comprising n processor cores. One complex problem is executed on the computer system and that program includes p parallel threads. Affinity of threads to processor cores is determined by function $a(i)$, where i is the thread number. The function $a(i)$ returns processor core $w \in \{1, 2, ..., n\}$, on which the thread is executing. Let the producer be the thread performing push an operation of insertion of an element into pool. The consumer thread is the thread performing an operation pop of deletion of an element out of pool.

Diffraction tree [12-15] is binary tree with height h, each node of is contains bits, determining further addresses of the threads. The tree nodes (balancers) redirect arriving from the threads queries for pushing and popping elements alternately to one of the child nodes. The leaves of diffraction tree correspond to concurrent queues $q = \{1, 2, ..., 2^h\}$. Performing the operations, the threads traverse the tree from the root to the leaves and push (pop) element into corresponding queue.

For the practical purposes the sequential order of distribution of threads between among the leaves usually is not necessary [16] and it can be neglected to increase the scalability of pool. So, we propose the algorithms of concurrent pool implementation based on diffraction tree. The algorithms base on localization of addresses to the tree nodes and utilization of thread-local storage (TLS).

B. Optimized pool based on diffraction tree

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 $m \le p$ instead of two separate atomic bits (Fig. 2). Nodes of each next level of the tree contain twice less arrays compared with previous level.

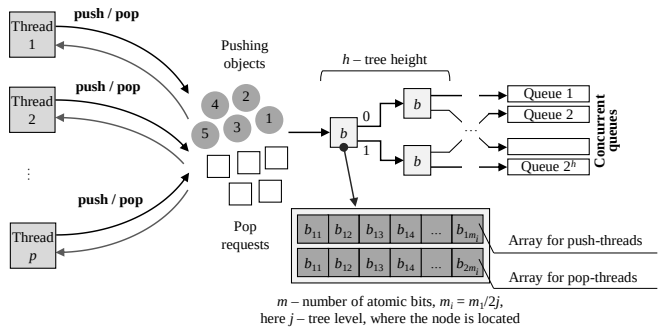

Fig. 1. Optimized pool LocOptDTPool based on diffraction tree ($h = 2$)

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.

Every time when a thread visits tree node it chooses the element in the atomic bit array by the value of hash function:

$$h(i) = i \bmod m \quad (1)$$

here $i \in \{1,2,...,p\}$ is the number of current thread, assigned at first pool visiting, m is the atomic array length.

Consider the scheme of distribution of concurrent queues in tree leaves among the processor cores. Each processor core $j \in \{1,2,...,p\}$ corresponds the queues $q_j = \{j2^h/n, j2^h/n + 1, ..., (j + 1)2^h/n-1\}$. 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.

When threads access to the structure fields, which includes the pool, we have to consider the false sharing, arising due to the placement of variables of the structures in the single cache line. For the false sharing solution sizes of the pool structured are aligned by the cache line size.

One the main drawbacks of existing pool implementation based on diffraction tree [13] is the increase in the time of operation performance at small number of threads (1-2 threads). For the solution of that problem pool LocOptDTPool takes into account current number of active threads in pool. If the current workload is low, then distribution of objects among the queues does not substantially increase throughput of the pool. In this case one queue is used for the storage of objects. Counting of active threads in the pool is realized by means of two atomic counters for producer threads and consumer threads correspondingly. This feature increases pool throughput at low workload.

Thus, described concurrent pool LocOptDTPool (Fig. 3) includes diffraction tree *tree*, arrays of atomic bites *prod_bits* and *cons_bits*, queue arrays *queues*, thread affinity manager *af_mgr*, counters *prod_num* and *cons_num* and methods push and pop for insertion and removing of the objects from the pool.

```
1  class LocOptDTPool {
2    Node tree
3    BitArray prod_bits[m], cons_bits[m]
4    ThreadSafeQueue queues[n]
5    AffinityManager af_mgr
6    AtomicInt prod_num, cons_num
7    push(data)
8    pop()
9  }
```

Fig. 2. The structure of concurrent pool LocOptDTPool

As concurrent *queues* in this work we used lock-free concurrent queue from the boost library [17], which has acceptable throughput for the practical reason.

The call of function push by thread j includes next steps:

1. Increase counter *prod_num* by one.

2. Affinity of current thread to processor core by means of affinity manager *af_mgr* if it has not been done yet.

3. Choose the queue, on which the object *data* will be inserted according the expression

$$q_j = (p \cdot l \bmod (2h / p) + a(j)) \quad (2)$$

here p is total number of processor core, l is tree leaf, visited by the thread, $a(j)$ is processor core number, to which thread j is affined, $2h$ is the total number of queues in the pool.

Method pop executed by the thread j includes the next steps:

1. The affinity of thread to processor core by means of affinity manager *af_mgr* and increasing of counter *cons_num* if it has not been done yet.

2. Choose the queue for removing the element by expression:

$$q_j = p \cdot \alpha + a(j) \quad (3)$$

here α is the shift factor, determined empirically.

3. If the queue q_j is empty, then the element is removed from the next first following non-empty queue. This approach is used in other pool implementations [16, 18]. In the case of success of the operation method pop returns removing object and in the case of failure the call is called again.

C. Optimized pool based on thread-local storage

We 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 [12-15].

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.

Threads cannot access to bits of neighbor threads in tree nodes. Thus, diffraction tree may not ensure uniform distribution if workload among queues. For the solving of that problem we propose next heuristic algorithm of initialization of elements of binary arrays in tree nodes. The algorithm is based on representation of thread identifier in binary form (Fig. 3). Each bit of identifier is a corresponding level of tree and the value of the bit points to initial state of all bits in tree nodes on this level.

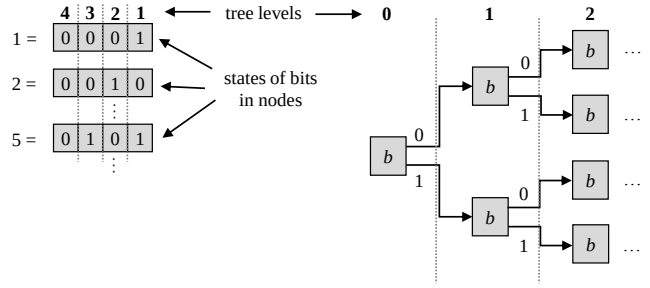

Fig. 3. Distribution of threads among tree nodes in TLSDTPool

Described algorithm in case of constantly active threads uniformly distributes the references of threads to tree nodes for prevention of imbalance of queue workload in tree leaves.

IV. EXPERIMENTAL RESULTS

Experiments for pool LocOptDTPool and TLSDTPool are conducted on the node of computer cluster Jet of Center of parallel computational technologies of Siberian state university of telecommunications and information sciences. As a performance indicator we used throughput $b = N / t$ of pool, where N is summary number of push (pop) operations and t is time of the experiment. We compared efficiency of pool for different queue types (lock-based and lock-free) in tree leaves. The queue comparison is important because the queue selection affects on pool throughput; this approach for experiments was applied in other works [16]. Experimental results for pool based on single non-blocking lock-free queue from boost library [19] are presented for the comparison. We made two experimental series for each pool: for thread number $p = 1, 2, ..., 8$ which do not exceed processor core number on computer node, and for greater number of threads $p = 10, 20, ..., 200$.

The experimental results for throughput of implemented pool based on arrays of atomic bits in tree nodes are represented on Fig. 4. Fig. 5 shows the experimental results for TLSDTPool throughput using thread-local bits.

Designed data structure scales well for large number of threads and shows the increase of throughput as the number of threads comes near the number of processor cores (Fig. 4, 5). Maximal throughput 170 million operations per second was obtained for number of threads which equals to number of processor cores or slightly exceed it.

Throughput of LocOptDTPool for whole range of number of threads corresponds to throughput of TLSDTPool. Maximum throughput 170 million operations per second was achieved for both pools for number of threads which equals to number of processor cores or slightly exceed it.

Previously we considered the case when number of threads addressing to single tree leaf of a pool using thread-local variables may significantly exceed number of threads making operations with other leaves of a tree. However, during the experiments this case has not been stated and there was not decrease of pool throughput. Nevertheless, in the pool design we should consider that increase of number of diffraction tree levels reduces the probability of "worst case" (at the same time it increases the memory consumption)

For a large number of threads lock-free queues in the pools LocOptDTPool TLSDTPool increases the throughput, comparing with lock-based concurrent queues (Fig. 4b, 5b). In all cases the efficiency of single concurrent queue is much less than the efficiency of developed pools (Fig. 4b, 5b).

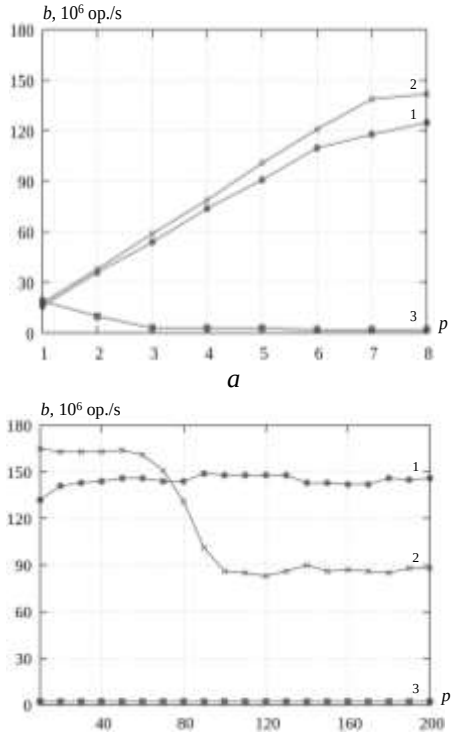

Fig. 4. Throughput analysis. a – $p = 1, 2, \ldots, 8$ (8 is the number of cores), b – $p = 10, 20, \ldots, 200$. 1 – LocOptDTPool, lock-free queues (boost), 2 – LocOptDTPool, lock-based queues PThread mutex, 3 – single lock-free queue (boost).

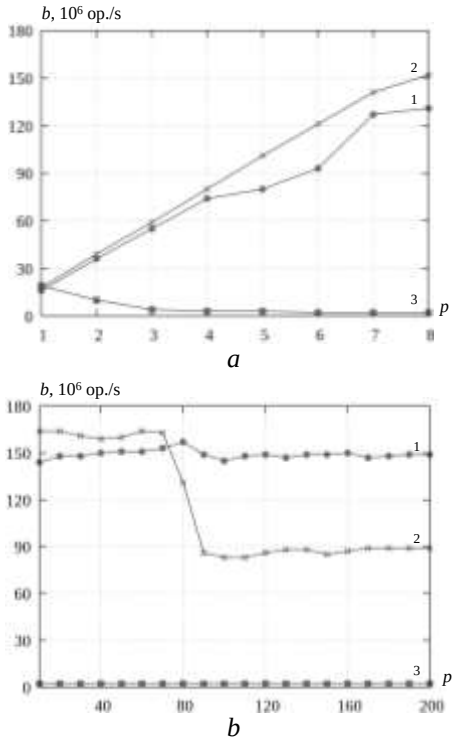

Fig. 5. Throughput analysis. a – $p = 1, 2, \ldots, 8$ (8 is the number of cores), b – $p = 10, 20, \ldots, 200$. 1 – TLSDTPool, lock-free queues (boost), 2 – TLSDTPool, lock-based queues PThread mutex, 3 – single lock-free queue (boost)

V. Conclusion

The algorithms of scalable lock-free concurrent pools based on diffraction trees are developed. The key idea of optimization is localization of references of threads to shared memory areas for maximization of pool throughput.

Developed pools may be applied for producer-consumer model implementation in multithreading programs with constant number of active threads and requirements of high throughput of pools and low latency of operations with pools. 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. Concurrent lock-free queues may be recommended as data structures in tree leaves for object storage.

References

[1] Khoroshevsky V.G. Distributed programmable structure computer systems. Vestnik SibGUTI [SibSUTIS Bulletin], 2010, V. 10(2), pp. 3-41 (in Russian).

[2] Herlihy M., Shavit N. The Art of Multiprocessor Programming, Revised Reprint. Elsevier, 2012, 528 p.

[3] Anderson T.E. The performance of Spin Lock Alternatives of Shared Memory Multiprocessors. TPDS, 1990, V. 1, I. 1, pp. 6-16.

[4] Mellor-Crummey J.M, Scott M.L. Synchronization without Contention. ACM SIGPLAN Notices, 1991, V. 26(4), pp. 269-278.

[5] Rudolph L., Slivkin M., Upfal E. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. SPAA, 1991, pp. 237-245.

[6] Blumofe R.D., Leiserson C.E. Scheduling Multithreaded Computations by Work Stealing. Journal of the ACM, 1994, V. 46, I. 5, pp. 365-368.

[7] Hendler D., Shavit N., Yerushalmi L. A scalable lock-free stack algorithm. SPAA, 2004, pp. 206-215.

[8] Moir M. et al. Using elimination to implement scalable and lock-free FIFO queues. SPAA, 2005, pp. 253-262.

[9] Afek Y., Hakimi M., Morrison A. Fast and scalable rendezvousing. Distributed computing, 2013, V. 26, I. 4, pp. 243-269.

[10] Shavit N., Touitou D. Elimination trees and the construction of pools and stacks: preliminary version. SPAA. 1995. pp. 54-63.

[11] Calciu I., Gottschlich J.E., Herlihy M. Using elimination and delegation to implement a scalable NUMA-friendly stack. HotPar, 2013, pp. 1-7.

[12] Shavit N., Zemach A. Diffracting trees. ACM Transactions on Computer Systems (TOCS), 1996, V. 14, I. 4. pp. 385-428.

[13] Afek Y., Korland G., Natanzon M., Shavit N. Scalable Producer-Consumer Pools based on Elimination-Diffraction Trees. European Conference on Parallel Processing, 2010, pp. 151-162.

[14] Della-Libera G., Shavit N. Reactive diffracting trees. Journal of Parallel and Distributed Computing, 2000, V. 60, I. 7, pp. 853-890.

[15] Ha P.H., Papatriantafilou M., Tsigas P. Self-tuning reactive distributed trees for counting and balancing. International Conference on Principles Of Distributed Systems, 2004, pp. 213-228.

[16] Shavit N. Data Structures in the Multicore Age. Communications of the ACM. 2011, V. 54, I. 3, pp. 76-84.

[17] Blechmann T. Chapter 19. Boost.Lockfree. Available at: http://www.boost.org/doc/libs/1_61_0/doc/html/lockfree.html (accessed 29 May 2018).

[18] Chase D., Lev Y. Dynamic circular work-stealing deque. SPAA, 2005, pp. 21-28.