Optimization of Data Locality in Relaxed Concurrent Priority Queues*

Andrey Tabakov[1][0000-0001-7220-107X] and Alexey Paznikov[1][0000-0002-3735-6882]

Saint Petersburg Electrotechnical University "LETI", 5 Professora Popova Str., Saint Petersburg 197022, Russia `komdosh@yandex.ru`, `apaznikov@gmail.com`

Abstract. Parallel computing is one of the top priorities in computer science. The main means of parallel processing information is a distributed computing system (CS) - a composition of elementary machines that interact through a communication medium. Modern distributed CSs implement thread-level parallelism (TLP) within a single computing node (multi-core CS with shared memory), as well as process-level parallelism (PLP) process-level parallelism for the entire distributed CS. Design of scalable concurrent data structures for shared memory systems is one of promising approach to relaxation of operation execution order. The growing demand for scalable parallel programs (especially in complex hierarchical control systems) states the evolution of MPI, which nowadays supports not only common PLP, but TLP within each the MPI-process. Relaxed concurrent data structures are non-linearizable and do not provide strong operation semantics (such as FIFO/LIFO for linear lists, delete max (min) element for priority queues, etc.). In the paper, we use the approach based on design of concurrent data structure as multiple simple data structures distributed among the threads. For operation execution (insert, delete), a thread randomly chooses a subset of these simple structures and make actions on them. We propose optimized relaxed concurrent priority queues based on this approach. We designed algorithms for optimization of priority queues selection for insert/delete operations and algorithm for balancing of elements in queues.

Keywords: Distributed Computing Systems · Concurrent Data Structures · Relaxed Data Structures · Relaxed Semantics · Multithreading.

1 Introduction

The Multicore computing systems (CS) with shared memory are the primary means of solving complex problems and processing large amounts of data. It is

* The reported study was funded by RFBR according to the research projects 19-07-00784 and was supported by Russian Federation President Council on Grants for governmental support for young Russian scientists (project SP-4971.2018.5).

used both autonomously and as part of distributed CS (cluster, multi-cluster systems, systems with mass parallelism). Such systems include number of multicore processors that share a single address space and have a hierarchical structure (Fig. 1). Thus, the Summit (OLCF-4) supercomputer (17 million processor cores), which is first in the TOP500 rating, includes 4608 computing nodes. [3]. Among the existing CS with shared memory, there are SMP-systems, providing the same speed of access of processors to memory, and NUMA-systems, represented as a set of nodes (composition of processor and local memory) and characterized by different latency for access of processors to local and remote memory segments [1]. Parallel programs for shared-memory CS are created in a multithreaded programming model. The main problem arising in the development of programs is the organization of access to shared data structures. It is necessary to implement the correct execution of operations by parallel threads (no race conditions, dead-locks, etc.) and to provide scalability for a large number of threads and a high intensity of operations. For this, it is necessary to develop means for synchronizing access to shared memory.

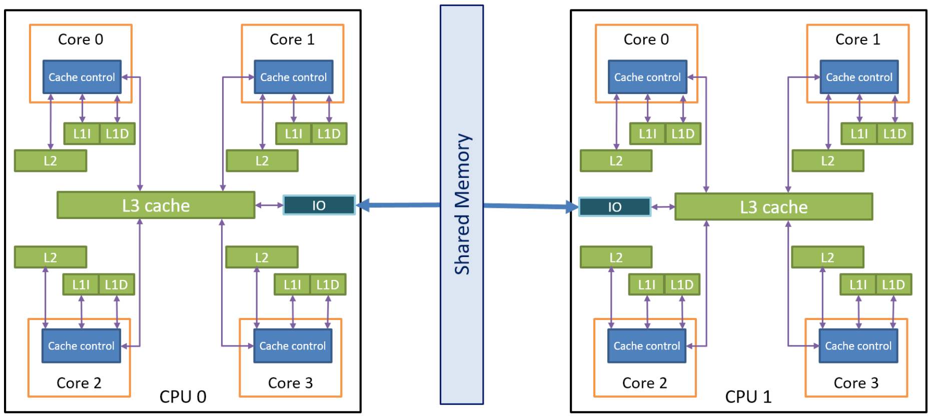

Fig. 1. Multicore system hierarchical structure.

2 Methods of Thread Synchronization

Locks Semaphores, mutexes implement access to shared memory areas with a single thread at any time. Among the locking algorithms, we can distinguish TTAS, CLH, MCS, Lock Cohorting, Flat Combining, Remote Core Locking, etc. [4]. The disadvantages of locks are the possibility of deadlocks (livelocks), threads starvation, priority inversion and lock convoy.

Non-blocking concurrent data structures they have the property of linearizability, assuming that any parallel execution of operations is equivalent to some sequential execution of them, and the completion of each operation does not require the completion of any other operation. Among non-blocking data structures, there are three classes: wait-free – each thread completes the execution of any operation in a finite number of steps; lock-free – some threads complete the execution of operations in a finite number of steps; obstruction-free – any thread completes the operations for a finite number of steps, if the execution of other threads is suspend-ed. Among the most common non-blocking algorithms and data structures, we could highlight Treiber Stack, Michael Scott Queue, Harris Michael List, Elimination Backoff Stack, Combining Trees, Diffracting Trees, Counting Network, Cliff Click Hash Table, Split-ordered lists, etc. [5]. The disadvantages of non-blocking data structures include the high complexity of their development (compared to other approaches), the lack of hardware support for atomic operations for large or non-adjacent memory (double compare-and-swap, double-width compare-and-swap), problems of freeing memory (ABA problem).

Transactional memory Within this approach, the program organizes transactional sections in which the protection of shared memory areas is implemented. Transaction is the finite operations sequence of the transactional read / write actions [16]. The transactional read operation copies the contents of the specified section of shared memory into the corresponding section of the local thread memory. The transactional write copies the contents of the specified section of local memory into the corresponding section of shared memory accessible to all threads. After completion of the execution, the transaction can either be committed or canceled. Committing a transaction implies that all changes made to memory become irreversible. There are software (LazySTM, TinySTM, GCC TM, etc.) and hardware (Intel TSX, AMD ASF, Oracle Rock, etc.) transactional memory. Among the shortcomings of the transactional memory, it is possible to distinguish restrictions on certain types of operations within the transactional sections, overhead in tracking changes in memory, the complexity of debugging the program. Given the above discussed shortcomings of the existing synchronization tools, their performance may be insufficient for modern multithreaded programs. In this paper, the method of increasing scalability is used due to relaxation of the semantics of data structures.

3 Relaxed Concurrent Data Structures

The basis of this approach is a compromise between scalability (performance) and the correctness of the semantics of operations [2]. It is proposed to relax the semantics of the implementation of operations to increase the possibility of scaling. In most existing non-blocking thread-safe structures and blocking algorithms, there is a single point of execution of operations on the structure. For example, when inserting an element into a queue, it is necessary to use a single pointer to the first element of the structure, in the case of a multi-threaded system. This fact is a bottleneck, since each thread is forced to block one element, causing other threads to wait. The principle of quasi-linearizability [13] is applicable to this approach, which assumes that several events may occur during the execution of some operations, simultaneously changing the data structure in such a way that after performing one of the operations, the state of the data structure is undefined. Relaxed concurrent data structures randomly select on of available structure. In case of successful locking of the structure, the thread completes the operation; otherwise, randomly selects an another structure. Thus, synchronization of threads is minimized, losses in the accuracy of operations are acceptable [2]. The main representatives of relaxed data structures are SprayList, k-LSM, Multiqueue.

SprayList This structure is based on the SkipList structure [11]. SprayList is a connected graph, where at the lower level of the structure there is a connected sorted list of all elements, and each next level with a given fixed probability contains the elements of the list of the lower level. The search for this structure is carried out linearly from top to bottom and from left to right; one pointer follows each iteration. Unlike the SkipList, SprayList assumes not a linear search from top to bottom and from beginning to end, is uses a random movement from top to bottom and from left to right. If the search fails or another thread locks the item, the algorithm returns to the previous item. After finding the necessary element, the operations with the list are performed in the same way as in the SkipList. However, in the worst case, inserting items into a SprayList causes significant overhead due to the need to maintain an ordered list.

k-LSM The log-structured tree with merge (LSM) is used as the basic structure of k-LSM [12]. Each structure change is recorded in a separate log file, tree nodes are sorted arrays (blocks), each of which is at the L level of the tree and can contain N elements ($2L–1 < N \leq 2L$). Each thread has a local distributed LSM. Common LSM is the result of the merging of several distributed LSM structures. All threads can access common LSM using a single pointer. As a result of combining common and distributed LSM structures, a k-LSM structure was obtained. When performing the insert operation, the thread stores the element in the local LSM structure. During a delete operation, the search for the smallest key in the local LSM is used. If the local structure is empty, and not insert operation is required, an attempt is made to access foreign LSM structures, the search is performed among all distributed and common LSM structures, and if it was found the structure that is not locked, an operation is performed on it. The disadvantages of this structure are the synchronization of calls to a common LSM and an attempt to call someone else's LSM, since there is no guarantee that it is not empty.

Multiqueue This structure [9] (Fig. 2) is a composition of simple priority queues protected by locks. Each thread has two or more priority queues. The operation of inserting an element is performed in a random, unlocked by another thread, queue. The operation of deleting an element with the minimum key is performed as follows: two random unlocked queues are selected, their values of the minimum elements are compared and the element with the smallest value from the corresponding queue is deleted. This element is not always the minimum inserted in the global structure, but it is close to the minimum and for real problems, this error can be neglected. This paper proposes algorithms to optimize the execution of operations for priority queues based on Multiqueues. Algorithms allow you to optimize the choice of structure for performing the operation [8].

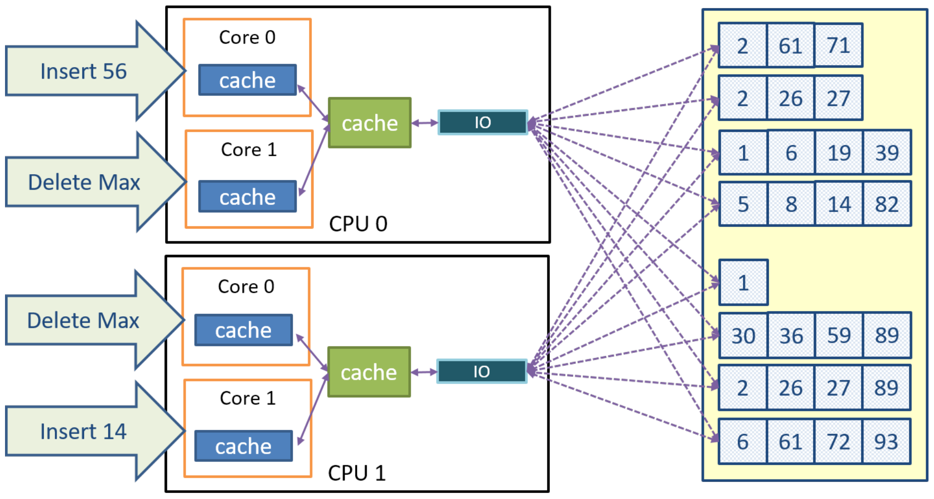

Fig. 2. Multiqueues - Relaxed Concurrent Priority Queue

4 Operation Execution Optimization

The disadvantage of the current implementation of insert and delete operations in Multiqueues is the algorithm for searching for a random queue. The thread performing the operation most likely turns to queues locked by other threads. The structure of Multiqueues includes kp queues, where k is the number of queues per thread, p is the number of threads.

4.1 Evaluation Queue Identifier by Thread

In the article methods are proposed for reducing the number of collisions based on the limitation of the range for randomly choosing a structure. By collisions is meant access a memory area that was already locked by another thread. The set of queues and threads is divided in half. Let i in (1) be the thread identifier, then for the first half of all threads (2), the random queue selection operation is performed among the queues (3), where *1* is selected to perform the operation queue in the Multiqueue structure. For the second half of threads (4), queue searching in the second half of queues (5).

$$i \in 0, 1, \ldots, p \quad (1) \qquad i \in 0, 1, \ldots, p/2 \quad (4)$$

$$q \in 0, 1, \ldots, kp/2 \quad (2) \qquad i \in p/2 + 1 \ldots, p \quad (5)$$

$$q \in kp/2 + 1, \ldots, kp \quad (3) \qquad q \in pi, \ldots, pi + k \quad (6)$$

An approach to optimizing the selection of queues, based on the "binding" of queues to threads, has also been developed. This scheme allows you to specify the order in which the thread accesses queues. In the implementation of fastening, the following model is used: in total, the set contains *kp* queues, then each thread has (6) fixed queues. When performing an operation, the thread first turns to the one of (6) queue, thereby minimizing calls to blocked queues. If all queues from the set (6) are blocked, then the original queue selection algorithm among all queues is used.

4.2 Optimized Algorithms

Algorithms presented in this section have the following semantics: Lock/Unlock – lock and unlock thread mutex for one of specified queues, tryLock – check that queue is not processing by another thread at the exact same time and try to lock mutex, RandQueue – return one queue from set of all queues by its identifier that in the specified range, Rand2Queue – same as a RandQueue, but returns pair of randomly selected queues, threadId - identifier of current thread for this concrete process from 0 to *p*. For implementation it is recommended to use thread affinity with cores, it helps to minimize context switching between threads [15].

OptHalfInsert Algorithm 1 represents an optimized insertion algorithm of an element in a Multiqueues structure. The queue is selected depending on which half of the threads the current thread identifier belongs to.

```
do
    if $i \in 0, 1, \ldots, p/2$ then
        q = RandQueue(0, kp/2);
    else
        q = RandQueue(kp/2+1, kp);
    end
while $tryLock(q) = false$;
insert(q, element);
Unlock(q);
```

Algorithm 1: OptHalfInsert

OptHalfDelete Algorithm 2 is exactly the same as OptHalfInsert, but it takes two queues for increase accuracy of deleted max (min) element.

```
do
    if $i \in 0, 1, \ldots, p/2$ then
        (q1, q2) = Rand2Queue(0, kp/2);
    else
        (q1, q2) = Rand2Queue(kp/2+1, kp);
    end
    q = GetMaxElementQueue(q1, q2);
while $tryLock(q) = false$;
removeMax(q);
Unlock(q);
```

Algorithm 2: OptHalfDelete

OptExactDelete Alternative optimized Algorithm 3. for deleting the maximum element, is similar to Algorithm 3. but with one change: at first it attempt to lock queues that "attached" to the thread (as it is threadId), and only after failing it search queues among all.

```
firstIteration = true;
do
    if firstIteration then
        (q1, q2) = Rand2Queue(threadId, threadId+p);
    else
        if $i \in 0, 1, \ldots, p/2$ then
            (q1, q2) = Rand2Queue(0, kp/2);
        else
            (q1, q2) = Rand2Queue(kp/2+1, kp);
        end
    end
    q = GetMaxElementQueue(q1, q2);
    firstIteration = false;
while $tryLock(q) = false$;
removeMax(q);
Unlock(q);
```

Algorithm 3: OptExactDelete

As a result of continuous running program, there may be an imbalance in a relaxed priority queues: some queues may contain significantly more elements than others. This circumstance leads to a decrease in the performance of the algorithms, since empty queues become unsuitable for a delete operation, which increases the search time for suitable queues to perform the operation. A balancing algorithm (Algorithm 4) for the complete Multiqueues structure has been created, for a uniform distribution of elements among the queues.

```
q1=FindLargestQueue();
q2=FindShortestQueue();
if $size(q1) > AvgSizeTotalSize() * \alpha$ then
    Lock(q1);
    q2IsLocked=LockWithTimeout(q2);
    if q2IsLocked then
        sizeToTransfer = size(q1)*$\beta$
        TransferElements(q1, q2, sizeToTransfer);
        Unlock(q2);
    end
    Unlock(q1);
end
```

Algorithm 4: Balancing algorithm

5 Evaluation

5.1 Testing Platform

All experiments reported in this paper were processed on one node of cluster, which equipped with dual-socket Intel Xeon X5670 where each socket contains six cores with 2.93 GHz each (Hyper-Threading is disabled). 16Gb of RAM was used.

5.2 Measurement Technique

As an indicator of efficiency, the throughput was used, which is calculated as the sum of the carrying capacities of the threads (7), where n is the number of insert / delete operations by the thread i, t is the execution time of operations [14].

$$b_i = n/t \tag{7}$$

5.3 Benchmarks

The effectiveness of the original and optimized multiqueues was compared. Individual insert / delete operations were investigated. Each thread was allocated $k = 2$ queues. Each thread performed $n = 10^6$ insert operations and $n = 0{,}5 * 10^6$ delete operations. The following experiment shows the dependence of the number of random operations (insertion, deletion) on the number of threads used. The following options for using the insertion and deletion algorithms were analyzed:

– The original insertion algorithm (Insert) and the original element deletion algorithm (Delete)
– Optimized insertion algorithm (OptHalfInsert) and optimized element deletion algorithm (OptHalfDelete)

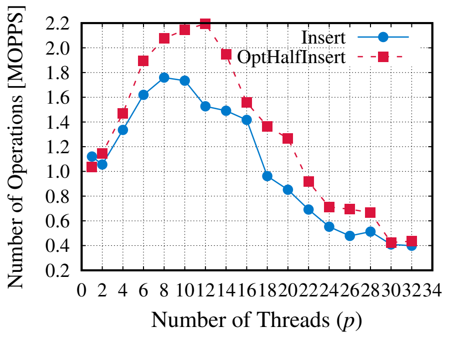

Fig. 3. Insert Throughput by Threads.

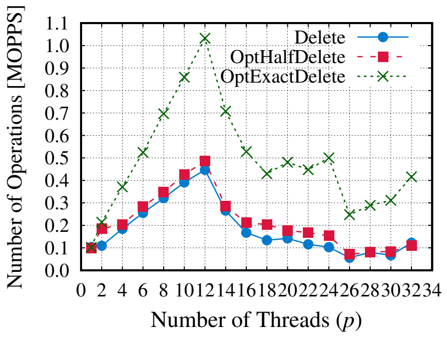

Fig. 4. Deletion Throughput by Threads.

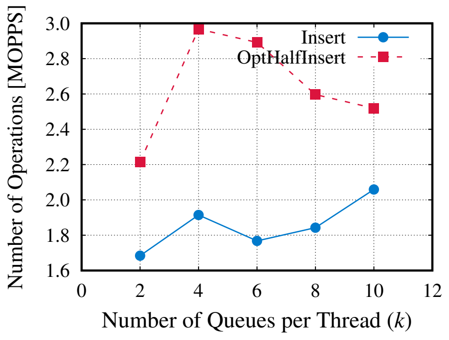

Fig. 5. Insert Throughput by Number of Queues.

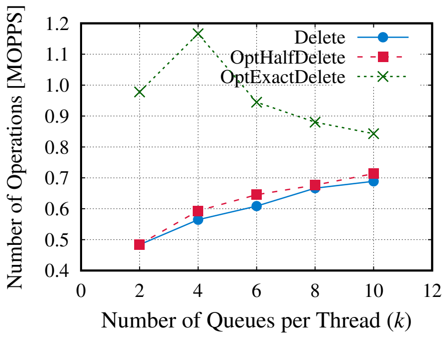

Fig. 6. Deletion Throughput by Number of Queues.

– Optimized insertion algorithm (OptHalfInsert) and an alternative optimized element removal algorithm (OptExactDelete)

The analysis of the optimal number of k queues per threads (Fig. 5 and fig. 6). We used a fixed number of threads $p = 12$. The algorithms OptHalfInsert and OptExactDelete has the maximum throughput of the system is achieved when the number of queues $k = 4$ per thread.

6 Future Work

Coming up with high-performance data processing of large data volumes is challenging in modern control systems. Concerning this, modern systems for complex object control (especially large-scale distributed hierarchical control systems) are based on multi- and many-core distributed computer systems. Computer systems include a wide class of systems – from embedded systems and mobile devices to cluster computing systems, massively parallel systems, GRID systems, and cloud-based CS. Algorithmic and software tools for parallel programming is the basis for building modern systems for processing big data, machine learning and artificial intelligence. The main class of systems used for high-performance information processing are distributed CS - collectives of elementary machines interacting through a communication environment. In design of parallel programs for distributed CS, the de-facto standard is the messaging model, which is primarily represented by the MPI (Message Passing Interface) standard. The scalability of MPI programs depends significantly on the efficiency of the implementation of collective information exchange operations (collectives) [7]. Such operations are used in most of the MPI programs, they account for a significant proportion of the total program execution time. An adaptive approach for development of collectives is promising. Nowadays, using only the message passing model (MPI-everywhere) may not be sufficient to develop effective MPI programs. In this regard, a promising approach is to use MPI for interaction between computer nodes and multithreading support systems (PThreads, OpenMP, Intel TBB) inside the nodes. The main task of implementation of hybrid mode is the organization of scalable access of parallel threads to shared data structures (context identifiers, virtual channels, message queues, request pools, etc.). Types of MPI standard hybrid mode:

- MPI_THREAD_SINGLE - one thread of execution
- MPI_THREAD_FUNNELED - is a multi-threaded program, but only one thread can perform MPI operations
- MPI_THREAD_SERIALIZED - only one thread at the exact same time can make a call to MPI functions
- MPI_THREAD_MULTIPLE - each program flow can perform MPI functions at any time.

One of the implementations of the hybrid multi-threaded MPI program in the MPI_THREAD_MULTIPLE mode is the MPICH version CH4 library, which defines standards for using lock-free data structures. In this mode, two types of synchronization are available: trylock - in which the program cyclically tries to capture the mutex and access the queue; handoff - a thread-safe queue when accessed which causes an active wait for an item by a thread.

It is proposed to replace the thread-safe work queue from the izem library with a relaxed thread-safe queue - Multiqueues, in which the mechanism for accessing queue elements has been improved

The use of a Multiqueues with relaxed semantics for performing operations will avoid the occurrence of bottlenecks when synchronizing threads [6]. Unlike most existing lock-free thread-safe data structures and locking algorithms, where there is a single point of execution of operations on the structure, a set of simple sequential structures is used in relaxed data structures, the composition of which is considered as a logical single structure. As a result, the number of possible points of access to this structure increases. This approach will allow to achieve much greater throughput compared to existing data structures.

7 Conclusion

An optimized version of a relaxed concurrent priority queue based on Multiqueues has been developed. The developed insertion and deletion algorithms has a 1.2 and 1.6 times performance boost, respectively, compared to the original insertion and deletion algorithms. Optimization is achieved by reducing the number of collisions based on the limitation of the random structure selection. In implementation is the thread affinity is used for minimization contention between cores. It is proposed to use a scalable thread-safe queue with relaxed semantics in hybrid multi-threaded MPI programs (MPI + threads model), which will reduce the overhead of synchronizing threads when performing operations with a working task queue. There are some evidences, that this queue is highly scalable and reduce overheads for thread synchronization. The implementation of these algorithms is publicly available at https://github.com/Komdosh/Multiqueues.

References

1. Anenkov, A. D., Paznikov, A. A., Kurnosov, M. G.: Algorithms for access localization to objects of scalable concurrent pools based on diffracting trees in multicore computer systems. In: 2018 XIV International Scientific-Technical Conference on Actual Problems of Electronics Instrument Engineering (APEIE) pp. 374-380. IEEE (2018) https://doi.org/10.1109/APEIE.2018.8545197
2. Tabakov A. V., Veretennikov L. M.: Relaxed Concurrent Data Structures. In III Science of Present and Future, pp. 105–107, ETU, Russia, (2018)
3. TOP500 supercomputers list, https://www.top500.org/news/summit-up-and-running-at-oak-ridge-claims-first-exascale-application. Last accessed 7 Nov 2019
4. Herlihy M., Shavit N.: The art of multiprocessor programming. Morgan Kaufmann, Boston (2011)
5. Shavit N., Touitou D.: Software transactional memory. Distributed Computing **10**, 99–116 (1997)
6. Tabakov, A. V., Paznikov, A. A.: Using relaxed concurrent data structures for contention minimization in multithreaded MPI programs. Journal of Physics: Conference Series **1399**(3), pp. 033037. IOP Publishing (2019) https://doi.org/10.1088/1742-6596/1399/3/033037
7. Tabakov A., Paznikov A.: Modelling of parallel threads synchronization in hybrid MPI+Threads programs. In XXI IEEE International Conference on Soft Computing and Measurements (SCM), 4–7 (2019)
8. Paznikov A., Anenkov A.: Implementation and Analysis of Distributed Relaxed Concurrent Queues in Remote Memory Access Model In XIII International Symposium "Intelligent Systems – 2018" (INTELS'18). Procedia Computer Science **50**, 654–662 (2019) https://doi.org/10.1016/j.procs.2019.02.101
9. Rihani H., Sanders P., Dementiev R.:Multiqueues: Simpler, faster, and better relaxed concurrent priority queues. arXiv pre-print arXiv:1411.1209 **50**, 277–278 (2015) https://arxiv.org/pdf/1411.1209.pdf. Last accessed 7 Nov 2019
10. Goncharenko E.A., Paznikov A.A., Tabakov A.V.: Evaluating the performance of atomic operations on modern multicore systems. Journal of Physics: Conference Series **1399**, 033107 (2019) https://doi.org/10.1088/1742-6596/1399/3/033107

11. Alistarh D. et al. The SprayList: A scalable relaxed priority queue. ACM SIGPLAN Notices **10**, 11–20 (2015).
12. Wimmer M. et al. The lock-free k-LSM relaxed priority queue. ACM SIGPLAN Notices **50**, 277–278 (2015).
13. Y. Afek, G. Korland, E. Yanovsky.: Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. OPODIS **6490**, 3–10 (2010)
14. Tabakov, A. V., Paznikov, A. A.: Algorithms for optimization of relaxed concurrent priority queues in multicore systems. In: 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus) 360–365. (2019) https://doi.org/10.1109/EIConRus.2019.8657105
15. Paznikov, A., Shichkina, Y.: Algorithms for optimization of processor and memory affinity for Remote Core Locking synchronization in multithreaded applications. Information **9**(1), 21–24 (2018) https://doi.org/10.3390/info9010021
16. Kulagin I. I.: Means of architectural-oriented optimization of parallel program execution for computing systems with multilevel parallelism. NUSC **10**, 77–82 (2017)
17. Paznikov, A. A., Smirnov, V. A., Omelnichenko, A. R. Towards Efficient Implementation of Concurrent Hash Tables and Search Trees Based on Software Transactional Memory. In: 2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon) 1–5 (2019). https://doi.org/10.1109/FarEastCon.2019.8934131