MDPI

Article

Topology-Aware Scalable Relaxed Concurrent Priority Queues

Andrey Tabakov and Alexey Paznikov*

Department of Computer Science and Engineering, Saint-Petersburg Electrotechnical University "LETI", St. Petersburg, Russia;

* Correspondence: komdosh@yandex.ru, apaznikov@gmail.com

† This paper is an extended version of our report: A. Tabakov; A. Paznikov; "Optimization of Data Locality in Relaxed Concurrent Priority Queues" in the Majorov International Conference on Software Engineering and Computer Systems (MICSECS 2019), Saint-Petersburg, Russia, 12–13 December 2019.

Version May 6, 2020 submitted to Computers

Abstract: Distributed parallel computing is the most promising direction of study in computer science. A distributed computed computing system (CS) is the main mean of parallel processing information. The shared memory processing (SMP) computing node, connected with each other, form distributed CS. One SMP node can use single memory access by many multicore processors. Modern distributed CSs implement thread-level parallelism (TLP) within only one computing node (multi-core processor with shared memory) and process-level parallelism (PLP) for the entire distributed CS. We focused on TLP implementation. In shared-memory systems, relaxation of operation execution order it is a promising approach, to design a good scalable concurrent data structures. The growing demand of high-performance computing arouses such problems as scalability of parallel programs (especially in complex hierarchical computing systems). Concurrent data structures with relaxed semantic 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 article, we use the approach based on the design of concurrent data structure as multiple simple data structures distributed among the threads. For concurrent operation execution (insert, delete), a thread chooses a subset of sequential data structures and perform operations on them. We propose two optimized relaxed concurrent priority queues based on this relaxed approach. We designed algorithms for optimization of priority queues selection in multiqueues. It based on attaching thread to a specific set of data structures. It had better throughput for insert and delete operations in 1.2 and 1.6 times respectively. Balancing algorithm for providing better performance on continuous usage of multiqueues. We developed a circular relaxed concurrent data structure based on a linked circular list of sequential data structures.

Keywords: Multithreading; Concurrency; SMP; Relaxed Semantic; Concurrent Data Structure; Distributed Computer Systems; MPI

1. Introduction

Multicore shared-memory computer systems (CS) with shared memory are the primary means of high-performance computing to solve complex problems and processing large amounts of data. One computing node can be used as an autonomous machine with one or more processors with a single memory area, and as part of distributed CS (cluster, multi-cluster systems, systems with mass parallelism). Every computing node of distributed CS contains many multicore processors and has a hierarchical structure (Figure 1). Thus, the Summit (OLCF-4) supercomputer (17 million processor cores), which is first in the TOP500 rating, includes 4608 computing nodes [1]. Among the existing CS with shared memory, the SMP-systems, providing the same memory access rate for all processors on

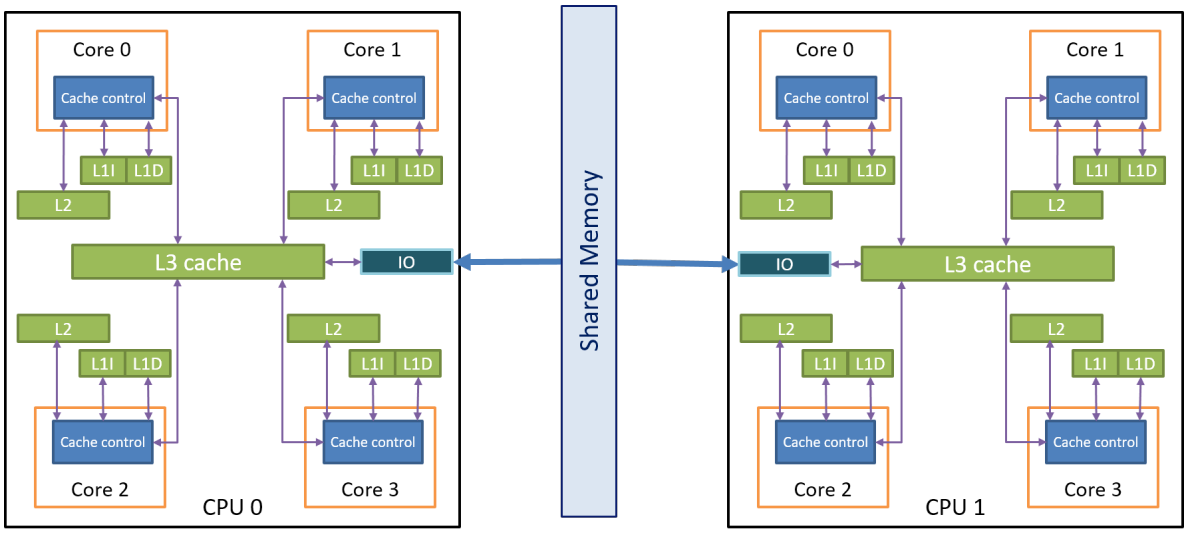

Figure 1. Hierarchical structure of shared-memory multicore system.

the node, and NUMA-systems, represented as a set of nodes (the composition of processor and local memory) and characterized by different latency for access of processors to local and remote memory segments [2].

Multithreading is the main approach for parallelization in a shared-memory environment. This approach can provide better performance when one large task can be divided into smaller independent tasks for execution, this task can be solved, in a parallel way use threading or work-stealing [3] approaches. The key problem in multithreading is organizing parallel access of multiple threads to shared memory areas. All operations must be performed correctly (no race conditions, deadlocks, etc.). Moreover, we need to provide high scalability for a large number of threads and a high intensity of operations. At the heart of any synchronization implementation, atomic operations [4] are used. It is necessary to develop a high performance means for synchronizing access, with low thread contention, to shared memory.

The performance of existing concurrent data structures due to strict synchronization may be insufficient for modern multi-threaded programs. They have a bottleneck in operation execution, while relaxed concurrent data structures can avoid it. In this paper, the method of increasing scalability is used due to the relaxation of the semantics of data structures.

The article is organized as follows. Section 2 describes related works on thread synchronization methods and relaxed concurrent data structures. Section 3 provides operation performance optimized algorithms for multiqueues. Section 4 describes algorithms for concurrent priority queue based on relaxed semantic approach. Section 5 identifies measurement techniques and benchmarks for optimized multiqueue and relaxed circular priority queue algorithms. Section 6 provides an outlook on future work and research. Section 7 provides conclusions and links to algorithms implementation.

2. Related Works

2.1. Methods of Thread Synchronization

2.1.1. Locks

Locks (mutexes, semaphores) implement access to shared memory areas by a single thread at any time. We can distinguish the locking techniques like TTAS, CLH, MCS, Lock Cohorting, Flat Combining, Remote Core Locking, etc. [5]. Locks can be used in various ways, including coarse-grained (the whole data structure is locked by on lock) and fine-grained (one data structure uses several mutexes or

semaphores to provide access to different parts of data structure) [6]. The disadvantages of locks are the possibility of threads starvation, deadlocks (livelocks), lock convoy and priority inversion.

2.1.2. Non-blocking concurrent data structures

As any correct sequential data structure, non-blocking concurrent data structures provide 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 suspended. Atomic operations help to provide such non-blocking finite number of steps states. They are supported by CPU and very efficient in concurrent environments [7].

Among the most common non-blocking algorithms and data structures, we could mention 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. [8]. The disadvantages of non-blocking data structures include are 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 memory reclamation (ABA problem).

2.1.3. Transactional memory

Within this approach, we can organizes transactional sections within which thread-safety is guaranteed. Transaction is the finite sequence of transactional read read / write operations [9]. 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. The key issues of transactional memory are the restrictions on certain types of operations within the transactional sections, overhead in tracking changes in memory, the complexity of debugging the program [10].

2.2. Relaxed Concurrent Data Structures

The basis of this approach is a trade-off between scalability (performance) and the correctness of the semantics of operations [11]. It is proposed to relax the semantics of the operation execution to increase the scalability. In most existing non-blocking concurrent 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. This fact becomes a bottleneck, raising high contention, since each thread is forced to block one element, causing other threads to wait. In relaxed concurrent data structures approach thread select one of the available structures. In case of successful locking of the structure, the thread completes the operation; otherwise, it selects another structure. Thus, synchronization of threads is minimized, losses in the accuracy of operations are acceptable. The principle of quasi-linearizability [12] applies 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. The main representatives of relaxed data structures are SprayList, k-LSM, Multiqueue.

2.2.1. SprayList

This structure is based on the SkipList structure [13]. 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, it uses a random movement direction 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 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 [14].

2.2.2. k-LSM

The log-structured merge tree (LSM) is used as the basic structure of k-LSM [15]. 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.

2.2.3. Multiqueue

This structure [16] (Figure 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 selected queue (queue can't be locked by another thread). 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. Presented algorithms allow you to optimize the selection of suitable data structures for perform operation on them.

3. Operation Performance Optimization in Multiqueues

The structure of Multiqueues includes $k \cdot p$ queues, where k is the number of queues per thread, p is the number of threads. The current implementation of insert and delete operations in Multiqueues has issues. The main problem is the algorithm for searching selects a random queue. The thread tries to perform the operation on queues, but it has no information which queue can be not locked by other threads in that moment.

3.1. Evaluation Queue Identifier by Thread

When multiple threads work with shared object, we can't avoid collisions. Collisions cause when we trying to access a memory area that has been already locked by another thread. We propose methods to reduce the number of collisions based on the limitation of the range for randomly selecting

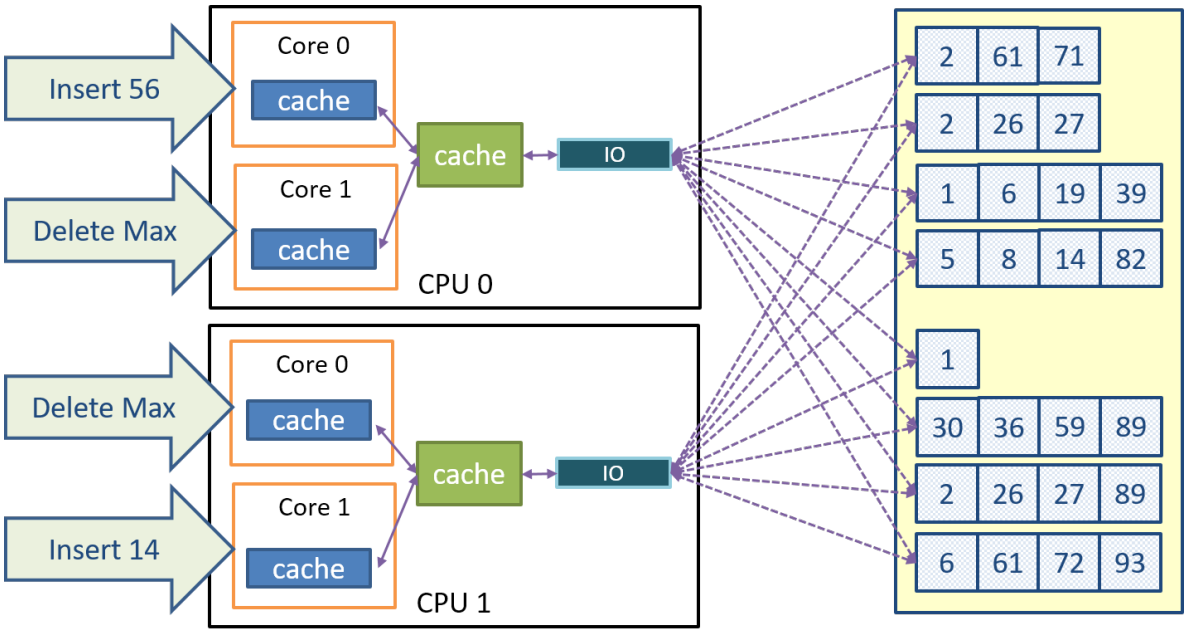

Figure 2. Multiqueues - Relaxed Concurrent Priority Queue

structures. To reduce the number of collisions, we propose to divide the set of queues and threads 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 subset of queues (3), where q is selected to perform the operation queue in the Multiqueue structure. For the second half of threads (4), a queue is selected within the second half of queues (5).

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

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

$$q \in k * p/2 + 1, \ldots, k \cdot p \quad (3) \qquad q \in p \cdot i, \ldots, p \cdot i + k \quad (6)$$

An approach for optimizing the selection of queues, based on the "binding" of queues to threads, has been developed. This scheme allows you to specify the order in which the thread accesses queues. In the concurrent priority queue implementation of fastening, the following model is used: in total, the set contains $k \cdot p$ queues, then each thread has q (6) fixed queues. Performing an operation, the thread first address to the one of q (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.

3.2. Optimized Insert and Delete Algorithms

In this section algorithms have the following semantics: Lock/Unlock – lock and unlock thread mutex for one of the specified queues, TryLock — check that queue is not locked by another thread at the same time and try to lock a mutex, RandomQueue – return one queue from a set of all queues by its identifier that in the specified range, TwoRandomQueues — same as a RandomQueue, but returns pair of randomly selected queues, *threadId* — identifier of the current thread from 0 to p. For a software implementation we recommend to use thread affinity with cores, because it minimize overheads to context switching between threads [17].

3.2.1. Optimized Half Insertion

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

belongs to.

```
1 do
2 |  if i ∈ 0, 1, . . . , p/2 then
3 |  |  q = RandomQueue(0, k · p/2);
4 |  else
5 |  |  q = RandomQueue(k · p/2 + 1, k · p);
6 |  end
7 while TryLock(q) = false;
8 Insert(q, element);
9 Unlock(q);
```

Algorithm 1: OptHalfInsert

On the line 2 we evaluate in which range we should search for queues. On the line 7 we try to lock found priority queue and if it is successful, we insert an element into it.

3.2.2. Optimized Half Deletion

Deletion Algorithm 2 is similar to OptHalfInsert, but it picks two queues and returns the maximal (minimal) element among them for increasing the accuracy.

```
1  do
2  |  if i ∈ 0, 1, . . . , p/2 then
3  |  |  (q1, q2) = TwoRandomQueues(0, k · p/2);
4  |  else
5  |  |  (q1, q2) = TwoRandomQueues(k · p/2 + 1, k · p);
6  |  end
7  |  q = GetMaxElementQueue(q1, q2);
8  while TryLock(q) = false;
9  DeleteMax(q);
10 Unlock(q);
```

Algorithm 2: OptHalfDelete

Function TwoRandomQueues on the lines 3 and 5 provide a pair of queues selected is specified half range. Then we compare the max (min) element of these queues and return the queue with this element. After that we try to lock it, and if it is succeed, return the element deleted from the queue.

3.2.3. Optimized Exact Deletion

Alternative optimized Algorithm 3 for deleting the maximum element, is similar to OptHalfDelete algorithm. It has the only difference: at first it attempts to lock queues that "bound" to the thread (as it is $threadId$), and only after failure it search queues among all. The first attempt allow us to avoid a large number of collisions, meanwhile next iterations we should take action on subset of half queue for

increase correctness and to avoid empty state.

```
1  firstIteration = true;
2  do
3      if firstIteration then
4          (q1, q2) = TwoRandomQueues(threadId, threadId + p);
5      else
6          if i ∈ 0, 1, . . . , p/2 then
7              (q1, q2) = TwoRandomQueues(0, k · p/2);
8          else
9              (q1, q2) = TwoRandomQueues(k · p/2 + 1, k · p);
10         end
11     end
12     q = GetMaxElementQueue(q1, q2);
13     firstIteration = false;
14 while TryLock(q) = false;
15 DeleteMax(q);
16 Unlock(q);
```

Algorithm 3: OptExactDelete

On line 3 we check that it is the first iteration in a cycle, it is necessary for a fair strategy of deletion max (min) element. If all threads will work only with their local queues, the accuracy can be significantly decreased. Function on line 4 returns a pair of queues which in $p \dots p \cdot i$ range according to the thread identifier.

3.3. Balancing Algorithm

Several queues can be filled faster than others. It will cause performance and accuracy issues because these queues will contain more elements, than others. This is unacceptable for our optimization methods, because we attach some range of queues to specified threads and it means that several threads will provide better accuracy with slow down performance and other threads will offer not accurate results faster. Moreover, empty queues become unsuitable for a delete operation, which increases the thread contention and as it search time for suitable queues to perform the operation. In Algorithm 4 we present the balancing algorithm for the complete Multiqueues structure. This algorithm can be started after n operation of insert and delete. The number of operation n can be empirically evaluated for needs. This balancing act will a uniform distribution of elements among the queues.

```
1  q1 = FindLargestQueue();
2  q2 = FindShortestQueue();
3  if size(q1) > AvgSizeTotalSize() · α then
4      Lock(q1);
5      q2IsLocked = LockWithTimeout(q2);
6      if q2IsLocked then
7          sizeToTransfer = size(q1) · β
8          TransferElements(q1, q2, sizeToTransfer);
9          Unlock(q2);
10     end
11     Unlock(q1);
12 end
```

Algorithm 4: Balancing Algorithm

On the lines 1 and 2 we find the largest and shortest queue for transfer elements. To avoid deadlocks we have an additional check on line 6. The balancing condition α and size of transferred elements β can be adjusted and should be evaluated empirically.

4. Circular Relaxed Concurrent Priority Queue

Circular relaxed concurrent priority queue (CPQ) represents by a set of traditional priority queues connected into one circular linked list. The main idea is close to multiqueues. The whole priority queue consists of several traditional priority queues with fine-grained locks. For write operation thread-locks only one priority queue. Each thread can perform read operation on any structure. It allows to threads select suitable structure and prepare their operation for faster execution. The scheme of CPQ is present on Figure 3.

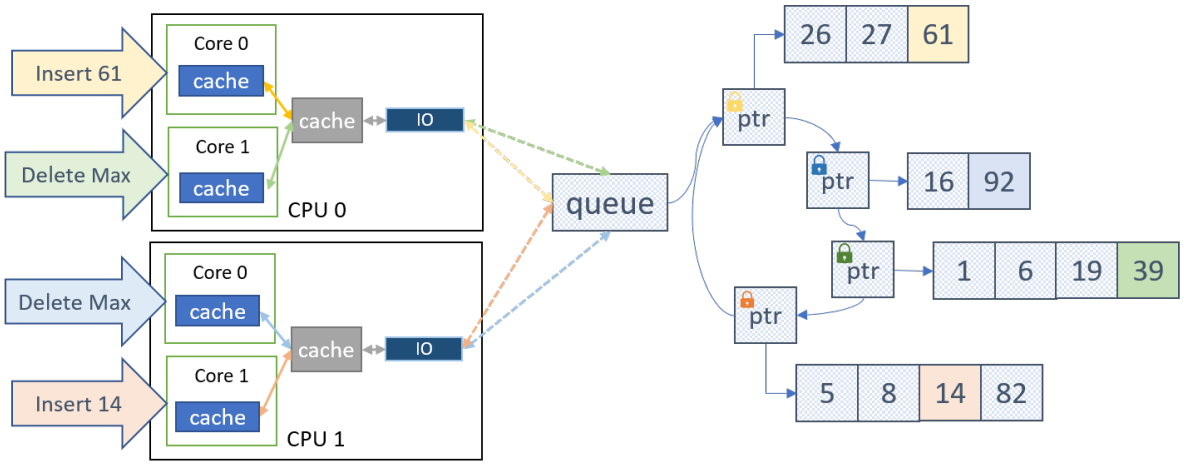

Figure 3. Circular Relaxed Concurrent Priority Queue

The next subsections provide algorithms for implementation such circular relaxed concurrent data structures. This idea is abstract and can be easily used with any other data structures such as stack, queue, tree, etc.

4.1. Push Algorithm

Push algorithm search not locked node to perform to push an element in the corresponding sequential data structure. It step by step try to lock every node. If it succeeded then it pushes element to the current node and break a while cycle, otherwise, when a cycle is complete it creates and attaches a new node to the CPQ. The pseudo-code of push operation is presented in Algorithm 5.

```
1  head = CPQHead();
2  node = head;
3  do
4      if TryLock(node) then
5          PushElement(node, el);
6          Unlock(node);
7          break;
8      end
9      node = NextNode(node);
10 while node! = head;
11 node = CreateNewNode(head);
12 Lock(node);
13 PushElement(node, el);
14 Unlock(node);
```

Algorithm 5: Push Algorithm

As you can mention, there is no presented concrete data structure, it is because node decorates push operation. This allows us to use any data structure into node.

4.2. Delete Max Element Algorithm

In relaxed concurrent data structures, execution of operation can provide unexpected results, but the accuracy of that result should be close to the expected [18]. The quasi-linearizability principle set that evaluated top element should be one of k top elements. This means that Algorithm 6 can provide not the maximum element in the whole CPQ, however it should be a top of one of sequential priority queue.

```
1  head = CPQHead();
2  node = NextNode(head);
3  priorValue = GetTopValue(head);
4  priorNode = head;
5  do
6  |  value = GetTopValue(node);
7  |  if value > priorValue then
8  |  |  priorValue = value;
9  |  |  priorNode = node;
10 |  end
11 |  node = NextNode(node);
12 while node != head;
13 Lock(priorNode);
14 DeleteMax(priorNode);
15 Unlock(priorNode);
```

Algorithm 6: Delete Max Element Algorithm

To offer better accuracy, algorithm checks all top elements of every node in CPQ. After that it will delete the max of the best suitable node. Again, there is no concrete data structure underneath, we can change this to any other.

5. Results and Discussion

5.1. Metrics

The throughput b was used as a metric. It is calculated as the sum of the individual throughput of the threads (7), where n is the number of insert / delete operations by the thread i, t is the execution time of operations [19].

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

Each test was evaluated multiple times. The result is the average of throughput sum by operation.

5.2. Optimization of Multiqueue Algorithms

5.2.1. Testing Platform

All experiments reported for multiqueues in this section were processed on one node of a cluster, which equipped with dual-socket Intel Xeon X5670 where each socket contains six cores with 2.93 GHz each (Hyper-Threading is disabled). Memory module DDR3 1666MHz with 16Gb of RAM was used.

5.2.2. Benchmarks

The throughput of optimized and the original multiqueues was compared. Individual delete / insert operations were investigated. Each thread was used $k = 2$ queues parameter. There was performed $n = 10^6$ insert operations and $n = 0.5 \cdot 10^6$ delete operations on every thread. The following experiment shows a correlation of the number of random operations (insertion, deletion) on the

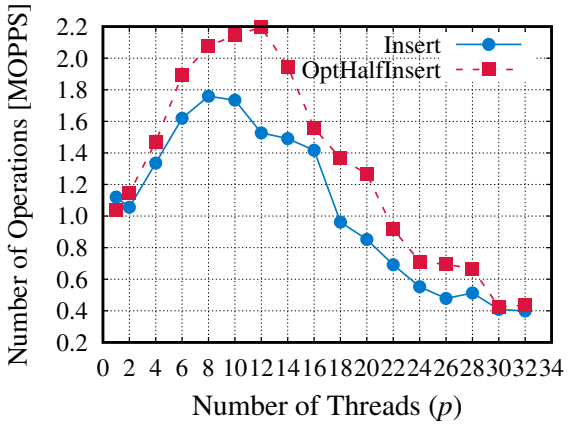

Figure 4. Insert Throughput by Threads.

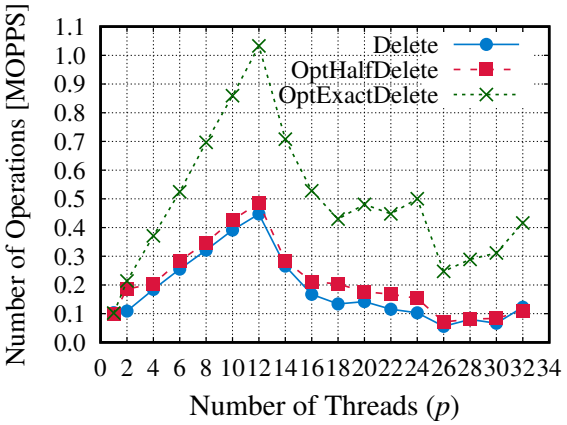

Figure 5. Deletion Throughput by Threads.

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

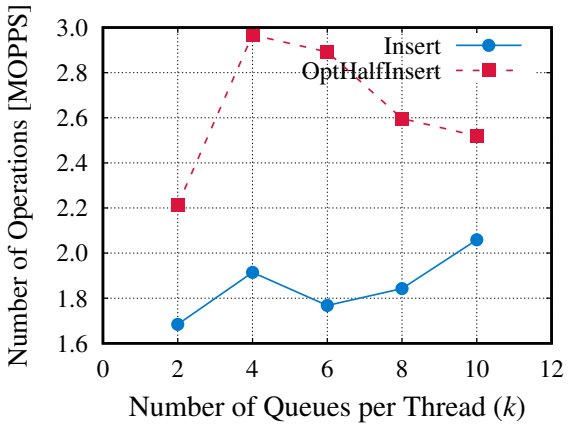

Figure 6. Insert Throughput by Number of Queues.

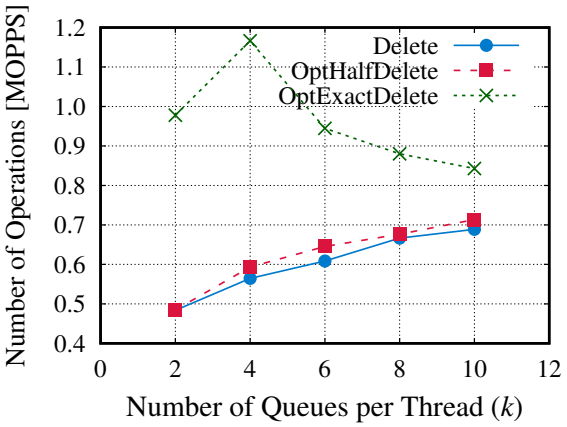

Figure 7. Deletion Throughput by Number of Queues.

On Figure 6 and Figure 7 presented the optimal number of k queues per threads. For this experiments we used a fixed number of threads $p = 12$. The optimized algorithms OptHalfInsert and OptExactDelete has the maximum throughput of a system is achieved when the number of queues $k = 4$ per thread, more queues per thread is unnecessary.

5.3. Circular Relaxed Priority Queue

5.3.1. Testing Platform

All experiments reported for circular relaxed priority queue in this section were processed on a desktop computer equipped AMD Ryzen 7 3800X eight-core with 4.3GHz each (SMT, simultaneous multi-threading is disabled). Two memory module DDR4 3200Mhz with 16GB each.

5.3.2. Benchmarks

For compare analysis Multiqueue with OptHalfInsert and OptExactDelete algorithms was used. Every test performed $n = 10^7$ insert operations and $n = 5 * 10^6$ delete operations.

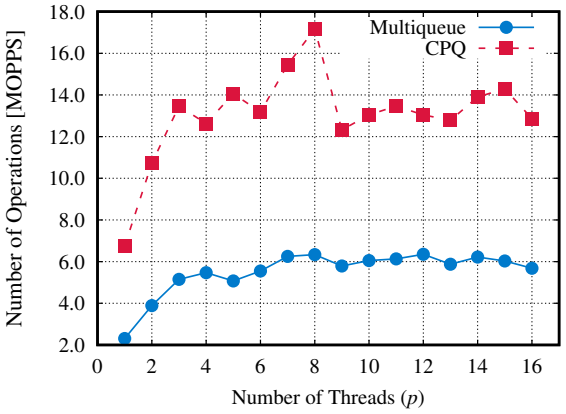

Figure 8. Insert Throughput by Number of Queues.

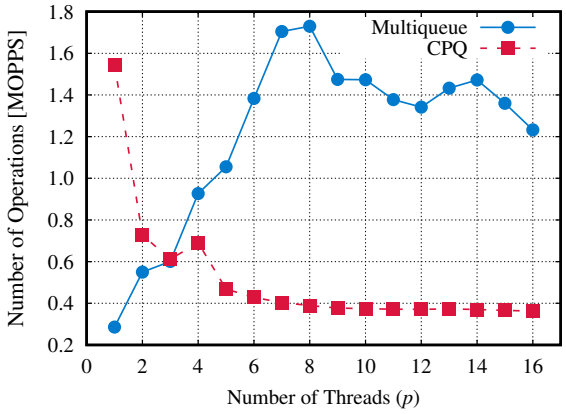

Figure 9. Deletion Throughput by Number of Queues.

Circular Priority Queue shows better performance and scalability on insert operation than Multiqueues, Figure 8. Better scalability is very important for shared memory systems with a large number of CPU, the overall efficiency of a multithreaded program depends on that parameter. On the other hand, deletion throughput of CPQ Figure 9 suffers from a contention due to the large number of cores. It causes lower throughput performance, because for this operation we need to take into account every top element on every sequential data structure, to provide better accuracy. It means that we should go through all elements in a circular linked list and check that value after the top element was found, we also should lock this traditional structure to prevent execution on it by other threads, until we delete our element.

6. Future Work

Coming up with high-performance data processing of large data volumes is challenging in modern computing. Concerning this, modern computer systems (CS), 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 GRID systems, cloud-based CS, cluster computing systems, and massively parallel systems. Algorithms and software for parallel programming is the basis for building modern systems for processing big data, artificial intelligence, and machine learning. The main class of systems used for high-performance information processing are distributed CS - collectives of elementary machines interacting through a communication environment. In the design of parallel programs for distributed CS, the messaging model is the de-facto standard. It 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) [20]. These 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 the development of collectives is promising. 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 same time can make a call to MPI functions
- MPI_THREAD_MULTIPLE - each program flow can perform MPI functions at any time.

MPI_THREAD_MULTIPLE mode is one of the implementations of the hybrid multi-threaded MPI program in 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. This MPI_THREAD_MULTIPLE mode also has disadvantages. In [21] author shows that in the existing implementations is used unfair thread synchronization algorithms. It leads to a decrease in the efficiency of information exchanges. The blocking algorithms are used in MPI libraries, however it is not guarantee fair capture of critical sections by threads. This can cause access to monopolization by one of the threads.

It is proposed to replace the thread-safe work queue from the izem library with a relaxed thread-safe queue - Multiqueues or CPQ, in which accessing the queue element mechanism has been improved. In some cases there is no need to provide strong semantic for this queues.

The use of a Multiqueues or CPQ with relaxed semantics for performing operations will avoid the occurrence of bottlenecks while synchronization of threads [22]. Most existing lock-free thread-safe data structures and locking algorithms have one coarse-grained locks, this means that there is a single point of execution of operations on the structure. A set of simple sequential structures is used in relaxed data structures, then we have multiple entry points with fine-grained locks approach. The composition of traditional data structures is considered as a logical single structure. This approach will allow us to achieve much greater throughput compared to existing data structures.

7. Conclusions

An optimized version of a relaxed concurrent priority queue based on Multiqueues has been developed. Compared with the original insertion and deletion algorithms the developed algorithms provide a 1.2 and 1.6 times performance boost, respectively. Achieved by reducing the number of collisions, optimization based on the limitation of the range to select random data structure. In implementation is the thread affinity is used for minimization contention between cores. Circular relaxed concurrent priority queue algorithms was developed. This data structure demonstrates high scalability efficiency for insert operation with better accuracy on delete max operation than Multiqueues. It can be easily used with any traditional data structure such as queue, stack, tree, list, etc. 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 reduces overheads for thread synchronization. All implementations of these algorithms are publicly available:

- https://github.com/Komdosh/Multiqueues - Multiqueues Implementation
- https://github.com/Komdosh/CircularPriorityQueue - CPQ C++ implementation
- https://github.com/Komdosh/RelaxedCycleDS - CPQ Kotlin implementation
- https://github.com/Komdosh/MPICH_DEV - MPICH build script with replaced working queue

Author Contributions: conceptualization, A. Tabakov. and A. Paznikov.; methodology, A. Tabakov. and A. Paznikov.; software, A. Tabakov.; validation, A. Tabakov. and A. Paznikov.; formal analysis, A. Tabakov. and A. Paznikov.; investigation, A. Tabakov.; resources, A. Paznikov.; data curation, A. Paznikov.; writing–original draft preparation, A. Tabakov.; writing–review and editing, A. Paznikov.; visualization, A. Tabakov.; supervision, A. Paznikov.; project administration, A. Paznikov.; funding acquisition, A. Paznikov.

Funding: The research 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).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CPQ	Circular Priority Queue
CS	Computer System
LSM	Log-Structured Merge tree
MPI	Message Passing Interface
SMP	Shared Memory Processing

References

1. TOP500 supercomputers list. Available online: https://www.top500.org/news/summit-up-and-running-at-oak-ridge-claims-first-exascale-application (Last accessed 7 Nov 2019).
2. Anenkov, A., Paznikov, A., Kurnosov, M. Algorithms for access localization to objects of scalable concurrent pools based on diffracting trees in multicore computer systems. *XIV International Scientific-Technical Conference on Actual Problems of Electronics Instrument Engineering (APEIE) IEEE*, **2018**, pp. 374–380, doi:10.1109/APEIE.2018.8545197.
3. Blumofe R., Leiserson C. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, **1999**, *46*, pp. 720–748.
4. Goncharenko E., Paznikov, A., Tabakov, A., Performance Modeling of Atomic Operations in Control Systems Based on Multicore Computer Systems, *2019 III International Conference on Control in Technical Systems (CTS)*, **2019**, pp. 140–143.
5. Herlihy M., Shavit N. In *The art of multiprocessor programming* **2008**, Morgan Kaufmann: Boston, USA, pp. 219–225.
6. Schenkel, R., Liu, L., Zsu, M. Locking Granularity and Lock Types. In *Encyclopedia of Database Systems* **2009**, Springer US: Boston, USA pp. 1641–1643.
7. Goncharenko E., Paznikov A., Tabakov A. Evaluating the performance of atomic operations on modern multicore systems. *Journal of Physics: Conference Series*, **2019**, *1399* (3), pp. 1–6, doi: 10.1088/1742-6596/1399/3/033107.
8. Shavit N., Touitou D. Software transactional memory. *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, **1995**, pp. 99–116.
9. Kulagin I. Means of architectural-oriented optimization of parallel program execution for computing systems with multilevel parallelism. *NUSC*, **2017**, *10*, pp. 77–82.
10. Paznikov A., Smirnov, V., Omelnichenko, A. Towards Efficient Implementation of Concurrent Hash Tables and Search Trees Based on Software Transactional Memory. *2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*, **2019**, pp. 1–5, doi: 10.1109/FarEastCon.2019.8934131.
11. Henzinger T. et al Quantitative relaxation of concurrent data structures. *ACM SIGPLAN Notices*, **2013**, pp. 317–328.
12. Afek Y., Korland G., Yanovsky E. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. *OPODIS*, **2010**, *6490*, pp. 3–10.
13. Alistarh D. et al. The SprayList A scalable relaxed priority queue. *ACM SIGPLAN Notices*, **2015**, *10*, pp. 11–20.
14. 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 2019**, pp. 654–662 doi: 10.1016/j.procs.2019.02.101.
15. Wimmer M. et al. The lock-free k-LSM relaxed priority queue. ACM SIGPLAN Notices, **2015** *50*, pp. 277–278.
16. Rihani H., Sanders P. Dementiev R.:Multiqueues: Simpler, faster, and better relaxed concurrent priority queues. Available Online: https://arxiv.org/pdf/1411.1209.pdf (Last accessed 7 Nov 2019)

17. Paznikov, A., Shichkina, Y. Algorithms for optimization of processor and memory affinity for Remote Core Locking synchronization in multithreaded applications. Information, **2018**, *9*, pp. 21–24. doi: 10.3390/info9010021.
18. Tabakov, A., Paznikov, A. Optimization of data locality in relaxed concurrent priority queues. *CEUR Workshop Proceedings* **2020**, *2590*
19. Tabakov, A., Paznikov, A. Algorithms for optimization of relaxed concurrent priority queues in multicore systems. *2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, **2019**, pp. 360–365. doi: 10.1109/EIConRus.2019.8657105.
20. Tabakov A., Paznikov A. Modelling of parallel threads synchronization in hybrid MPI+Threads programs. *XXI IEEE International Conference on Soft Computing and Measurements*, **2019**, pp. 4–7.
21. Amer A. et al MPI+ threads: Runtime contention and remedies. *ACM SIGPLAN Notices*, **2015**, pp. 239–48.
22. Tabakov, A., Paznikov, A. Using relaxed concurrent data structures for contention minimization in multithreaded MPI programs. *Journal of Physics: Conference Series*, **2019**, *1399 (3)* pp. 1–5, doi: 10.1088/1742-6596/1399/3/033037.