Decentralized lock-free distributed queue in MPI remote memory access model

Alexey A. Paznikov*, *Alexander* V. Burachenko, and *Mohamed* M. Abuelsoud

Department of Computer Science and Engineering, Saint Petersburg Electrotechnical University "LETI", ul. Professora Popova 5, Saint Petersburg 197022, Russia

Abstract. In parallel programming for distributed-memory systems in MPI standard, remote memory access model (also known as one-sided communications, MPI RMA or RMA) is used along with the message-passing. This model in many cases leverages the performance and simplifies parallel programming. Here arises the problem of synchronization of multiple parallel processes accessing shared (concurrent, distributed) data structures. In shared-memory machines (such as SMP/NUMA systems), non-blocking (lock-free, wait-free, obstruction-free) synchronization is widely used to solve the similar problem. The main advantage of non-blocking synchronization is that delays in execution of one process (thread) do not suspend execution of other threads. This avoids deadlocks, priority inversions, etc. We suppose this approach could also be effective in designing distributed data structures (in the MPI RMA model particularly). In this article, we discuss the idea of building non-blocking distributed data structures in MPI RMA model on the example of a queue, describe the designed algorithms of basic operations, investigates the efficiency of data structures, and provides an experimental comparison with lock-based counterparts.

1 Introduction

A distributed-memory high-performance computer system (HPC) comprises elementary computers (EC) that interact through a communication network (interconnect). EC can be represented as CPU cores, CPUs, or individual computing nodes. In addition to the message passing model, alternative programming models are widely used in developing programs for high-performance systems, such as the remote memory access (RMA, MPI RMA) model, and the partitioned global address space (PGAS) model.

Let us delve into the MPI RMA model, which is one of the promising parallel programming models for HPC. It's implemented in the MPI standard as a subsystem of one-sided communications [1, 2]. In the MPI RMA model, processes directly access the memory of other processes instead of explicitly sending and receiving messages. Unlike the PGAS model (which includes UPC, Coarray Fortran, Cray Chapel, and IBM X10 languages), RMA is closely integrated with MPI libraries and can be used alongside the message-passing model. It also implements access to low-level communication and synchronization

* Corresponding author: apaznikov@gmail.com

primitives. Using RMA often reduces program execution time in comparison with the message-passing model [3]. The performance of RMA is on par, and sometimes even surpasses the performance provided by PGAS languages [4]. This optimization is primarily achieved due to the hardware support in most modern communication networks (such as Infiniband, PERCS, Gemini, Aries, RoCE over Ethernet, etc.) of the RDMA technology [5]. RDMA allows access to remote memory segments without the intervention of the central processor.

Within the realm of the MPI RMA model, key operations encompass non-blocking functions such as MPI Put (writing to remote memory) and MPI _Get (reading from remote memory), alongside a range of atomic operations like MPI Accumulate, MPI _Get accumulate, MPI Compare and _swap, and others. These functions must be executed within specific segments known as epochs, during which synchronization occurs. MPI RMA offers active and passive target synchronization methods. Active synchronization involves a designated process (referred to as the target) initiating an exposure epoch, allowing other processes access to its memory while engaging in synchronization. Conversely, other processes (known as the source) begin an access epoch to perform remote operations on the memories exposed by the target processes. In this work, passive target synchronization is employed [2]. With passive synchronization, an origin process opens an access epoch using MPI_Win_lock/MPI_Win_lockall functions, enabling RMA operations to access registered memory segments (windows) of target processes. Consequently, RMA operations in this scenario are one-sided, without requiring explicit synchronization function calls by the targets. This approach reduces overhead compared to active synchronization [6].

In both the RMA and PGAS models, ensuring scalable access to data structures distributed across the nodes of whole system presents a challenge. This challenge isn't new in parallel programming; for instance, shared memory systems require synchronization between threads accessing shared, concurrent, thread-safe data structures. Such structures need to provide correct (often linearizable) access to parallel threads at arbitrary times [5, 7, 8]. The efficiency of synchronization significantly impacts program execution time and determines execution progress guarantees.

Existing synchronization methods can be categorized into two classes: lock-based and nonblocking. Locks have simple semantics and are often comparably efficient to non-blocking methods. However, the non-blocking approach provides execution guarantees and avoids issues like deadlocks and priority inversions inherent in locks. A non-blocking algorithm is one where a delay in one thread's execution doesn't impede the overall program's progress (under certain conditions) [5]. Non-blocking algorithms can be further classified as wait-free (each thread completes its operation in a finite number of steps), lock-free (at least one thread completes its operation in a finite number of steps), and obstruction-free (one thread completes its operation in a finite number of steps even in the absence of other threads' operations).

Much of the work on concurrent data structures aims to develop synchronization tools shared memory multicore systems. These encompass thread-locking algorithms [5, 9] (TTS, Backoff, CLH, MCS, Oyama, Flat Combining, RCL, etc.). While some methods consider hierarchical system levels, they're inapplicable in distributed memory DCSs. Classical non-blocking concurrent structures [5, 7, 8, 9] are designed for shared-memory machines and aren't suitable for distributed ones. An implementation of a stack for NUMA systems was proposed in [10]. One reason for challenges in designing concurrent data structures for shared memory is the memory reclamation problem after an element is removed from the structure, known as the ABA problem [11, 12]. Modern relaxed concurrent data structures [13, 14] scale well, but totally rely on shared-memory machines.

Distinguished among implementations of distributed non-blocking queues are FastQueue, CircularQueue [15], and Active Message Queue [16]. These structures have a common

drawback: data centralization in one process's memory. This leads to other processes constantly accessing the dedicated process, becoming a bottleneck that could hamper performance, particularly with numerous processes. Authors of [17] designed a distributed non-blocking stack based on the Elimination-Backoff Stack algorithm is examined. Its performance study suggests scalability up to eight computing nodes, with scalability diminishing thereafter. The centralization of this structure is a drawback. In [18], a scalable distributed concurrent queue based one-sided MPI interface is proposed. Despite high scalability of this data structure, it's does not rely on non-blocking synchronization concept.

This study explores the feasibility of constructing non-blocking (lock-free) data structures for distributed HPC systems using the queue as an example. We propose an approach based on decentralizing data among parallel program processes, believing it could prove effective in building distributed data structures.

2 Decentralized non-blocking distributed queue

2.1 Queue model and structure

A queue embodies a collection of objects that adheres to the FIFO (first in, first out) principle. This entails two key operations:

- Insertion (insert, enqueue) of an element at the end position (tail, T).
- Removal (remove, dequeue) of an element from the front position (head, H).

A distributed queue is a queue whose elements reside in the memory of processes within a distributed HPC system and interact with it.

We consider a system equipped with N processing elementary computers (*EC*). Each *EC* m_i (i = 1, 2, ..., N) equipped with a local main memory m_i, where $m_i \cap m_j = \varnothing$, $\forall i, j \in 1, 2, \ldots, N$. Further, each element mi has a process p_i running on it.

Let w represent the RMA window required for inter-process operations. All processes p_i, $i = 1, 2, ..., N$, possess a descriptor of window w and reserve a segment of their memory m_i (referred to as m_i^*), allowing other processes access to this memory.

To store queue elements, memory is allocated for a pool of elements on each process (currently realized as an array). Let m_i denote the queue element pool in memory m_i^* of a process p_i. The size of array e_i is constant and set to K. Upon adding new elements, process p_i utilizes cells from array $e_i : e_{ij}$, $\forall i, j : i \in 1, 2, ..., N, j \in 1, 2, ..., N$. The head of the queue H is the element e_{ij} at the forefront, and the tail of the queue T is the element e_{ij} at the end.

For locating queue elements, the concept of an element reference e_{ij} is introduced. An element reference comprises a pair of numbers (i, j), serving to uniquely identify element j within the distributed queue situated in the memory of process i. An empty reference is represented by $(\varnothing, \varnothing)$.

Each element e_{ij} encompasses:

- A timestamp τ_{ij} indicating its addition to the queue.
- User data v_{ij}.
- A state c_{ij} of an element, which can be:

F – free, allowing the element for future queue additions.

A – used, signifying the element is in the queue.

D – deleted, indicating the element has been removed from the queue but is not yet available for reuse. The element's state cyclically changes from F to A upon addition to the queue, from A to D when deleted, and from D to F when freed.

- A link to the next element n_{ij}. If e_{ij} is succeeded by e_{gf}, then $n_{ij} = (g, f)$. Upon adding e_{ij} to the queue, $n_{ij} = (\varnothing, \varnothing)$.

- A self-reference link l_{ij}. For e_{ij}, $l_{ij} = (i, j)$. This link facilitates element information updates.

Each process p_i also maintains links to the head h_i and tail t_i of the queue in memory m_i^*, $\forall\, i \in 1, 2, ..., N$, to determine the current positions of the head H and tail T.

Additionally, process p_1 holds a reference s, an element for consensus when initially adding to the queue. In cases where multiple processes concurrently add a new element, a decision is reached regarding which element takes precedence. This is achieved by employing the Compare-And-Swap (CAS) operation using the current tail T and new tail n_{ij}. However, the queue is initially empty, lacking a head or tail. Thus, memory space is needed to determine the initial element to be added to the queue.

In the centralized approach, memory of the main process p_1 stores information about head and tail positions (Fig. 1). When adding an element to the queue's tail or removing from the queue's head, processes p_i update references t_1 or h_1 respectively. However, this leads to a bottleneck as p_i frequently accesses p_1, diminishing the data structure's throughput.

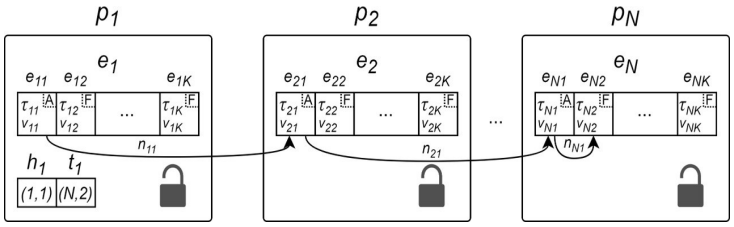


Fig. 1. The structure of a distributed lock-based queue with centralized storage of information about the position of the tail and head.

2.2 Decentralized queuing method

We introduce a decentralized method for storing information about the head and tail positions (Fig. 2). Each process p_i maintains local references h_i and t_i . When determining the current head or tail, process p_i accesses reference h_i or t_i, respectively. Post insertion or removal, process p_i updates references h_j and t_j (j = 1, 2, …, N) for all other processes through the developed notification algorithm.

In this algorithm, p_i initially updates its own references h_i and t_i, then proceeds to update references on other processes: h_j and t_j for all $j \in 1, 2, \ldots, N, j \neq i$. The algorithm operates across three modes: "head only", "tail only", and "tail then head". Depending on the mode, the algorithm updates solely the references h_j, t_j, or first t_j followed by h_j for each process, respectively. Detailed insight into the notification algorithm is provided in Section 2.5.

Upon dequeuing elements, they transition to state D, indicating their removal from the queue while being unavailable for reuse. To transition elements e_{ij} to state F, i.e., freeing them, a cleaning algorithm for pool e_i is devised, elaborated further in Section 2.6. This algorithm utilizes timestamps τ_{ij} of queue elements, references to the head h_i and tail t_i, and an additional element o_i . o_i is leveraged by processes for reading queue elements, copying the contents of the currently processed element's fields into it. o_i is positioned in memory m_i^*, accessible to other processes, and boasts the same attributes as e_{ij} (corresponding field names for o_i are: τ_i ,v_i, c_i, n_i, l_i). Importantly, o_i doesn't belong to array e_i.

Given the significance of accurate and temporally consistent timestamps τ_{ij}.

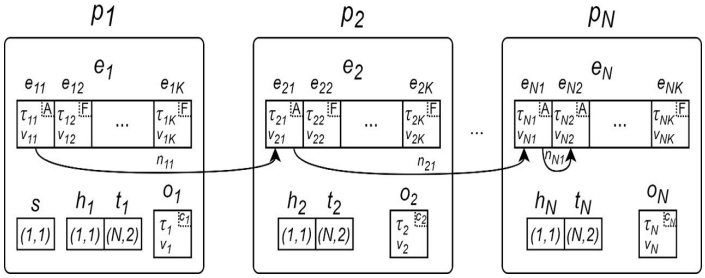


Fig. 2. Structure of a distributed non-blocking queue with decentralized storage of tail and head position information.

2.3 Enqueue operation

We outline the process of inserting elements to the queue through the enqueue operation (Fig. 3a). The input parameters for this operation include the element value *val* and the RMA operation window *win*. The operation *AtomicGet* refers to an atomic remote read operation, built on the foundation of MPI _Get accumulate. Additionally, the "CAS" operation represents a mechanism based on MPI Compare and _swap, resolving consensus when modifying shared memory across multiple processes without using locks. The *BcastT* and *BcastTH* operations are notifications triggered through the notification algorithm, adhering to the "tail only" and "tail then head" patterns, respectively.

The procedure for inserting an element to the queue by process pi is as follows (Fig. 3a): Sure, here's the improved version:

1. Initialize the new element e_{ij} by selecting a free cell e_{ij} within the array e_i and populating it with the data (v_{ij}= *val*, c_{ij}= A, n_{ij}= (∅, ∅)) (line 1).
2. If the array e_i is full, the cleaning algorithm is triggered. If no free cell is available after the cleaning algorithm, the execution of the algorithm terminates with an error (lines 1–2).
3. If the tail reference t_i = (∅, ∅), retrieve the reference s (lines 4–5). Otherwise, proceed to step 7.
4. If s = (∅, ∅), set the timestamp τ_{ij} within e_{ij} and set the reference (i, j) in s using the CAS operation (lines 6–8). Otherwise, proceed to step 6.
5. If the CAS operation is successful (a linearization point), inform all processes about the updated tail and new head e_{ij} and terminate the algorithm (lines 9–10). If not, update s (line 12) and return to step 6.
6. Write the content of s into t_i (line 14).
7. Retrieve the element referenced by t_i and store it in o_i (line 16).
8. If the status c_i = A (line 17), proceed to step 9. Otherwise, go to step 12.
9. If n_i = (∅, ∅), set the timestamp τ_{ij} within e_{ij} and set the reference (i, j) in n_i using the CAS operation (lines 18–20). Otherwise, move to step 11.
10. If the CAS operation is successful (a linearization point), notify all processes about the updated tail e_{ij} (line 21) and conclude the algorithm (line 22). If not, update oi and proceed to step 11 (line 24).
11. Retrieve the element referenced by n_i, store it in o_i, and return to step 8 (line 26).
12. If the status c_i = D, go to step 14 (line 29). Otherwise, return to step 3 (line 41).

```
Input data: val − data to be added
            win − window for RMA-operations
1  if INITELEM(val, &e_ij ) = false then
2    return BUFFER_FULL
3  end if
4  if t_i = (−1, −1) then
5    s = ATOMICGET(s, win)
6    if s = (−1, −1) then
7      e_ij.τ_ij = GETTS()
8      if CAS(s, (−1, −1), (i, j), win) then
9        BCASTTH(e_ij, win)
10       return SUCCESS
11     end if
12     s = ATOMICGET(s, win)
13   end if
14 t_i = s
15 end if
16 o_i = ATOMICGET(t_i, win)
17 if o_i.c_i = A then
18   if o_i.n_i = (−1, −1) then
19     e_ij.τ_ij = GetTS()
20     if CAS(o_i.n_i, (−1, −1), (i, j), win) then
21       BCASTT(e_ij, win)
22       return SUCCESS
23     end if
24     o_i = ATOMICGET(o_i.l_i, win)
25   end if
26   o_i = ATOMICGET(o_i.n_i, win)
27   goto 17
28 end if
29 if o_i.c_i = D then
30   if o_i.n_i = (−1, −1) then
31     e_ij.τ_ij = GETTS()
32     if CAS(o_i.n_i, (−1, −1), (i, j), win) then
33       BCASTTH(e_ij, win)
34       return SUCCESS
35     end if
36   else
37     o_i = ATOMICGET(o_i.n_i, win)
38     goto 17
39   end if
40 end if
41 goto 4
```

(a)

```
Input data: win − window for RMA-operations
1  if h_i = (−1, −1) then
2    s = ATOMICGET(s, win)
3    if s = (−1, −1) then
4      return QUEUE_EMTPY
5    end if
6    h_i = s
7  end if
8  o_i = ATOMICGET(h_i, win)
9  if o_i.c_i = A then
10   if CAS(o_i.c_i, A, D, win) then
11     val = o_i.v_i
12     o_i = ATOMICGET(o_i . l_i, win)
13     if o_i.n_i ≠ (−1, −1) then
14       BCASTH(o_i.n_i, win)
15     end if
16     return SUCCESS
17   end if
18   o_i = ATOMICGET(o_i . l_i, win)
19 end if
20 if o_i.c_i = D then
21   if o_i.n_i ≠ (−1, −1) then
22     o_i = ATOMICGET(o_i.n_i, win)
23     goto 9
24   else
25     return QUEUE_EMPTY
26   end if
27 end if
28 goto 1
```

(b)

Fig. 3. Comparison of parallel program mapping algorithms (*a* – insert operation, *b* – remove operation).

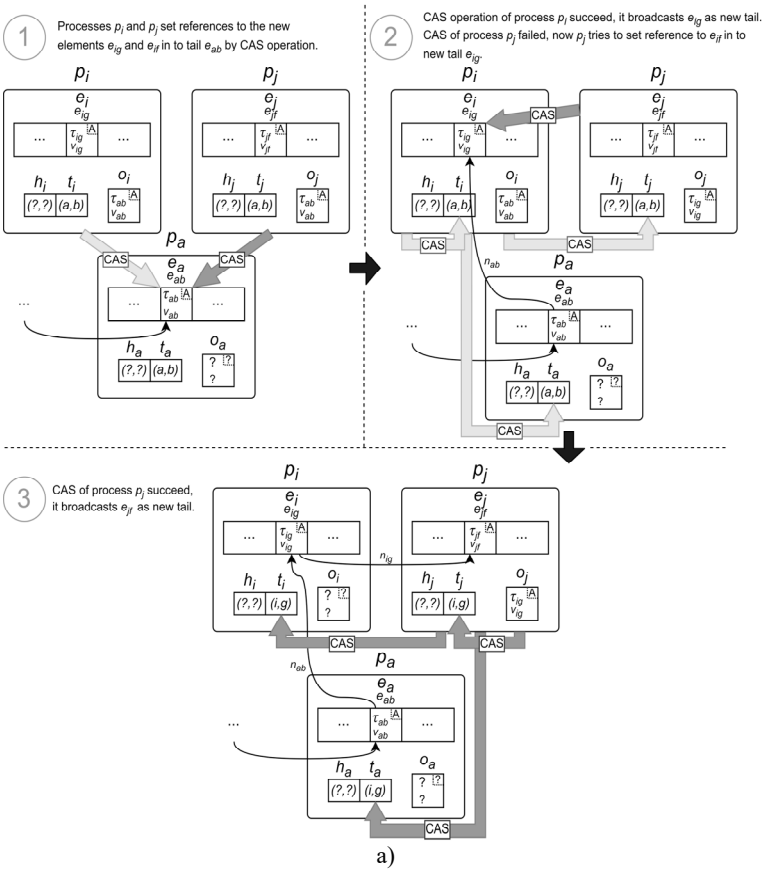
13. If ni = (∅, ∅), set the timestamp τij within eij and set the reference (i, j) in ni using the CAS operation (lines 30–32). Otherwise, retrieve the element referenced by ni, store it in oi, and return to step 8 (lines 37–38).
14. If the CAS operation is successful (a linearization point), inform all processes about the updated tail and new head eij and terminate the algorithm (lines 33–34).

2.4 Dequeue operation

The process of dequeuing an element (Fig. 3b) involves utilizing a window, represented as *win*, for executing remote memory access (RMA) operations. The operation *BcastH* corresponds to a notification algorithm employing the “head only” approach. The algorithm outlining the removal of an element from the queue by process p_i is presented in Fig. 3b:

1. If the current head reference of the queue, denoted as h_i, is empty, acquire a new reference s (lines 1-2). Alternatively, proceed to step 4.
2. If s is empty, it signifies an empty queue, leading to the termination of the algorithm (lines 3-4).
3. Update h_i with the value of s (line 6).
4. Fetch the element at reference h_i and write it to o_i (line 8).
5. In case the element's status is A, utilize the CAS operation to mark the element as deleted. Otherwise, if the element is already deleted, move to step 8 (lines 9-10).
6. If the CAS operation succeeds (indicating a linearization point), proceed to step 7. If not, update o_i and move to step 8 (lines 18-20).
7. If n_i holds a value, broadcast a notification to all processes regarding the new head (the element at reference n_i), and then terminate the algorithm (lines 12-16).
8. If the element's status is c_i = D, proceed to step 9 (line 20). Otherwise, return to step 1 (line 28).
9. If n_i contains a value, obtain the element at reference n_i, write it to o_i, and return to step 5 (lines 21-23). Alternatively, if n_i is empty, the queue is considered empty, leading to the termination of the algorithm (line 25).

Figure 4 represent the enqueue/dequeue operations for a distributed lock-free queue.



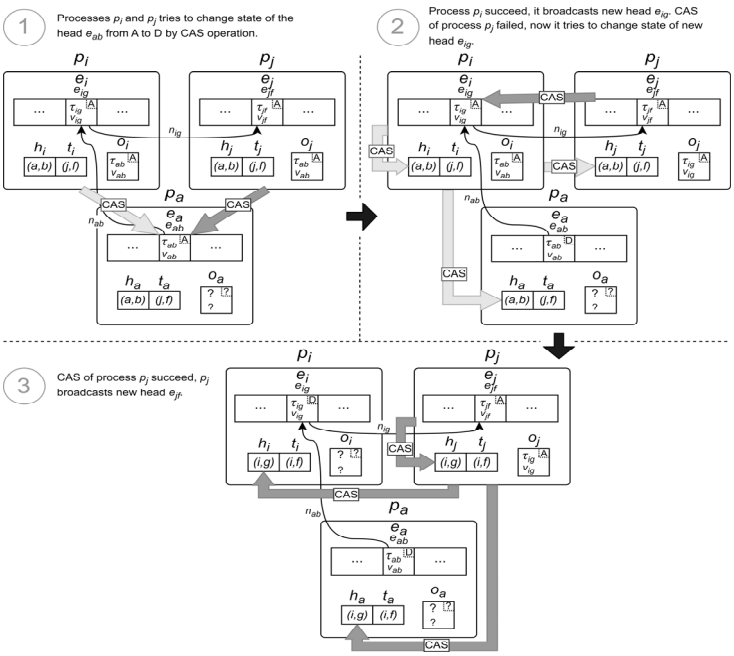


Fig. 4. Scheme of inserting/removing an item to the queue (*a* – insert/enqueue, *b* – remove/dequeue).

2.5 Notification algorithm (bcast)

For each process p_i, references to the head h_i and tail t_i of the queue are maintained. Whenever either the head or tail undergoes a change, it becomes necessary to update the references for all other processes, denoted as h_j and t_j, where $\forall\, j \neq i$. This synchronization is accomplished through the bcast notification algorithm. The execution of this algorithm involves three distinct modes, contingent upon which references necessitate an update:

1. Head Only: In this mode, solely the tail references t_j are updated. This occurs when a process adds an element to the queue.
2. Tail Only: This mode entails updating solely the head references h_j . It comes into play when a process removes an element from the queue.
3. Tail First, Then Head: In this mode, both tail and head references t_j and h_j are updated. This situation arises when a process adds an element to the queue following an element in the D state.

The bcast algorithm produces an execution result, which can be either SUCCESS (indicating successful link update) or FAIL (indicating that the link was not updated, likely due to another process having already performed the update). In cases where the algorithm yields the FAIL result, it serves as an indicator to cease the notification algorithm, as the distributed element no longer serves as the current tail or head. Another process has taken up the task of distributing the current information.

Outlined below is a template for the notification algorithm. The inputs comprise the replicated element e_{ij}, the chosen notification method bcast method, and the designated window for RMA operations, represented as *win*.

Let's delve into the notification algorithm for process p_i (Fig. 5a):

1. The algorithm initiates by creating an array *b* containing the ranks of all processes currently engaged with the queue (line 1).

2. Process p_i attempts to update its own references h_i and t_i (line 2). If the update is successful, process p_i removes its own rank from the array b (line 5).
3. Process p_i concludes the notification algorithm (lines 7-8). If unsuccessful, proceed to step 4.
4. Randomly choose a process p_a from the remaining set of processes in array b (line 10). Attempt to update the references ha and ta of process p_a using the specified BcastMethod. If the update fails (BcastMethod returns FAIL), terminate the algorithm (lines 11-12). Otherwise, remove the rank of process p_a from array b (line 14) and return to step 3.

Now, let's closely examine the algorithms for updating references to the head and/or tail in the memory of the j-th process.

For the "only head" method, with input data including the rank of process a, the replicated element e_{ij}, and the RMA operations window *win*, we have Fig. 5b:

1. Retrieve the head reference h_a from process a'*s* memory (line 2).
2. If h_a is not empty, extract the element at reference h_a and store it in o_i (lines 3-4). Otherwise, proceed to step 4.
3. If timestamp τ_{ij} is less than timestamp τ_i, return the FAIL code and terminate the algorithm (lines 5-6). Otherwise, proceed to step 4.
4. Attempt to update the head reference to (i, j) (line 9) using the CAS operation. If successful, return the *SUCCESS* code and conclude the algorithm (line 10). If not, return to step 1.

For the "only tail" method, with the same input data, we have Fig. 5c:

1. Retrieve the tail reference t_a from process a's memory (line 2).
2. If t_a is not empty, extract the element at reference t_a and store it in o_i (lines 3-4). Otherwise, proceed to step 4.
3. If timestamp τ_{ij} is less than timestamp τ_j, return the FAIL code and terminate the algorithm (lines 5-6). Otherwise, proceed to step 4.
4. Attempt to update the tail reference to (i, j) using the CAS operation (line 9). If successful, return the SUCCESS code and conclude the algorithm (line 10). If not, return to step 1.

The "tail then head" method updates the head and tail references in the order of tail followed by head. The input data include the process number a, replicated element e_{ij}, flags *CheckHead* and *CheckTail* (indicating whether to update head and tail references), and RMA operations window *win*. The consistent flags are vital as e_{ij} might cease to be the current tail or head, necessitating the algorithm to continue without checking the tail or head references respectively. Figure 5d illustrates this:

1. If *CheckTail* is true, proceed to step 2 (line 1). Otherwise, go to step 6.
2. Retrieve the tail reference t_a from process a's memory (line 3).
3. If the tail reference isn't empty, extract the element at the tail reference and store it in o_i (lines 4-5. Otherwise, proceed to step 5.
4. If timestamp τ_{ij} is less than timestamp τ_j, set *CheckTail* to false and proceed to step 6 (lines 6-8). Otherwise, proceed to step 5.
5. Using the CAS operation, update reference (i, j) to ta (line 11). If successful (linearization point), proceed to step 6. If not, return to step 2.
6. If *CheckHead* is true, proceed to step 7 (line 13). Otherwise, go to step 11.
7. Retrieve the head reference h_a from process a's memory (line 15).
8. If h_a isn't empty, extract the element at reference h_a and store it in o_i (lines 16-17). Otherwise, proceed to step 10.
9. If timestamp τ_{ij} is less than timestamp τ_j, set check head to false and proceed to step 11 (lines 18-20). Otherwise, proceed to step 10.

10. Using the CAS operation, update reference (i, j) to h_a (line 23). If successful (linearization point), proceed to step 11. If not, return to step 7.
11. If both *CheckTail* and *CheckHead* are false, return the FAIL code (lines 25-26). Otherwise, return the SUCCESS code and conclude the algorithm (line 28).

2.6 Cleaning algorithm

In the queue operation process, elements e_{ij} transition from state F (free) to A (active) when added to the queue, and from A to D (deleted) when removed. The array e_i might eventually run out of available free elements (F state), causing subsequent enqueue operations to fail. To prevent this situation, a cleaning algorithm is employed to transfer elements from the D state back to the F state.

The cleaning algorithm is divided into two parts:

1. Determining the minimum timestamp τ_{min} of currently active elements.
2. Marking all elements in the array e_i with timestamps less than τ_{min} as available (*F* state).

The algorithm's detailed steps are as follows (for process p_i):

1. Create an array b containing the ranks of processes working with the queue.
2. Retrieve the element at address t_i and store it in o_i, also store τ_i as τ_{min}.
3. Retrieve the element at address h_i and store it in o_i. If $\tau_i < \tau_{min}$, update τ_{min}.
4. Remove rank i from array b.
5. If array b is empty, proceed to step 11. Otherwise, continue to step 6.
6. Randomly select process p_a from the remaining ranks in array b.
7. Retrieve the element at address h_a and store it in o_i. If $\tau_i < \tau_{min}$, update τ_{min}.
8. Retrieve the element at address t_a and store it in o_i. If $\tau_i < \tau_{min}$, update τ_{min}.
9. Retrieve element o_a and store it in o_i. If $\tau_i < \tau_{min}$, update τ_{min}.
10. Remove rank a from array b. Return to step 5.
11. Iterate over array e_i . For each e_{ij} : if c_{ij} = D and $\tau_{ij} < \tau_{min}$, set c_{ij} to F.
12. Finish executing the algorithm.

Input data:	e_{ij} – replicated element *bcast_method* – notification method *win* – window for RMA-operations
1	b = GETALLRANKS()
2	**if** BCASTMETHOD(i, e_{ij}, *win*) = FAIL **then**
3	**return**
4	**end if**
5	EXCLUDERANK(i, &b)
6	**while true**
7	**if** ISEMPTY(b) **then**
8	**return**
9	**end if**
10	a = GETNEXTRANKRAND(b)
11	**if** BCASTMETHOD(a, e_{ij}, *win*) = FAIL **then**
12	**return**
13	**end if**
14	EXCLUDERANK(a, &b)
15	**end**

(a)

Input data:	a – rank of the notified process e_{ij} – replicated element *win* – window for RMA-operations
1	**do**
2	h_a = ATOMICGET(a, *win*)
3	**if** $h_a \neq$ (-1, -1) **then**
4	o_i = ATOMICGET(h_a, *win*)
5	**if** $e_{ij}.\tau_{ij} < o_i.\tau_i \neq$ (-1, -1) **then**
6	**return** FAIL
7	**end if**
8	**end if**
9	**while** CAS(h_a, h_a, (i, j), *win*) = false
10	**return** SUCCESS

(b)

Input data:	a – rank of the notified process e_{ij} – replicated element *win* – window for RMA-operations

Input data:	a – rank of the notified process e_{ij} – replicated element *CheckHead* – request flag to update head link *CheckTail* – request flag to update tail link

```
1   do
2     t_a = AtomicGet(a, win)
3     if t_a ≠ (-1, -1) then
4       o_i = AtomicGet(t_a, win)
5       if e_ij.τ_ij < o_i.τ_i ≠ (-1, -1) then
6         return FAIL
7       end if
8     end if
9   while CAS(t_a, t_a, (i, j), win) = false
10  return SUCCESS
```

(c)

win – window for RMA-operations

```
1   if CheckTail then
2     do
3       t_a = AtomicGet(a, win)
4       if t_a ≠ (-1, -1) then
5         o_i = AtomicGet(t_a, win)
6         if e_ij.τ_ij < o_i.τ_i != (-1, -1) then
7           CheckTail = false
8           break
9         end if
10      end if
11    while CAS(t_a, t_a, (i, j), win) = false
12  end if
13  if CheckHead then
14    do
15      h_a = AtomicGet(a, win)
16      if h_a != (-1, -1) then
17        o_i = AtomicGet(h_a, win)
18        if e_ij.τ_ij < o_i.τ_i != (-1, -1) then
19          CheckTail = false
20          break
21        end if
22      end if
23    while CAS(h_a, h_a, (i, j), win) = false
24  end if
25  if CheckTail = false and CheckHead = false
26  then
27    return FAIL
28  end if
    return SUCCESS
```

(d)

Fig. 5. Notification algorithms for a distributed non-blocking queue (*a* – notification algorithm template, *b* – "head only" method, *c* – "tail only" method, *d* – "tail first, then head" method).

3 Conducting experiments

Experimental research on the lock-free distributed queue was conducted using the computing cluster at the Information and Computing Center of Novosibirsk State University. The cluster consisted of 6 computing nodes, each equipped with two 6-core Intel Xeon X5670 2.9 GHz processors, totaling 72 cores. The MPI library used was Open MPI 4.1.0.

A synthetic test involving $n = 10000$ insertion/deletion operations (randomly chosen) was conducted. The number of processes N varied from 2 to 72. Throughput $b = n \times k / t$ was measured, where t is the experiment's time and K is the number of processes.

The implemented lock-free distributed queue was compared with two lock-based queue implementations in the RMA model.

The first implementation involves a blocking queue with centralized head/tail position storage (see Fig. 6). In this variant, information about the queue's head/tail position is stored in the memory of the process with rank 0. Locking is applied at the process level. Before reading/modifying the information about the head/tail position or a queue element from the memory of a certain process i, the process performing the read/write operation acquires the lock of process i. After completing the operation with the data, the lock is released.

The second implementation was a blocking queue with decentralized head/tail storage (Fig. 8). This variant resembled the non-blocking implementation described earlier but used locks. The head/tail positions were stored individually on each process, and locks were used to handle access to elements. Unlike the first blocking queue, individual elements are locked,

not the process. Before working with an element or a reference to the head/tail, the initiator process acquires a lock on that element and releases it after working. This ensures that only the elements that are being worked with are blocked, and not the entire queue.

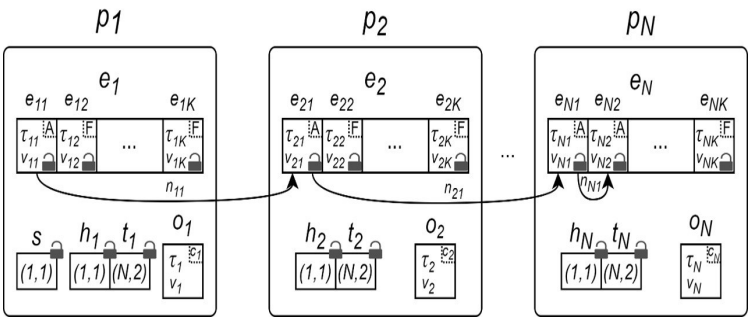


Fig. 6. Blocking Queue with Decentralized Head/Tail Position Storage.

The non-blocking queue significantly outperformed both blocking variants in terms of throughput (Fig. 7a, 7b). The lock-free approach avoided the overhead of lock acquisition and release, enabling threads to retry operations without waiting in case of failure. The blocking queue with decentralized head/tail storage was more efficient than the centralized one, distributing the load across processes and improving throughput.

All the analyzed queues exhibit a reasonable degree of scalability. While the throughput of these data structures does experience a decline as the number of processes increases, the non-blocking queue manages to maintain a notably elevated performance level.

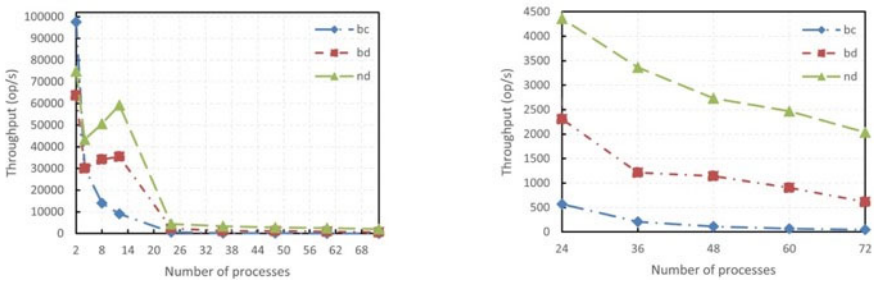


Fig. 7. Queue Throughput Dependency (*bc* – Blocking Centralized, *bd* – Blocking Decentralized, *nd* – Non-blocking Decentralized).

To assess the execution time of insertion and deletion operations, the non-blocking queue's performance was measured, focusing on the time taken to execute these operations and their crucial phases. Each process autonomously conducted the measurements. Subsequently, all operation results were transmitted to the primary process (rank 0), where average values were computed. The measured aspects encompass:

1. The overall execution duration of the insertion operation (EnqAll).
2. The time required to locate the queue's tail during insertion (EnqHop).
3. The comprehensive execution time of the deletion operation (DeqAll).
4. The time taken to identify the queue's head during deletion (DeqHop).
5. The total execution time of the broadcast algorithm (BcastAll).

The results clearly indicate (see Fig. 8) that a substantial portion of the processing time is devoted to locating the head or tail of the queue. This phenomenon arises due to the escalating number of queue operations per unit of time in correspondence with the growth of processes. Consequently, the likelihood of process p_i receiving an element that is no longer the tail via

reference t_i increases. Subsequently, the process must navigate through references to successive elements in order to reach the actual tail. With a surge in the volume of operations added to the queue per unit of time, the duration spent searching for the tail escalates. A similar predicament surfaces with deletion operations within the queue.

As anticipated, operations are executed expeditiously within a single node, particularly up to 12 processes. However, the time required increases with an augmentation in the number of nodes. This time escalation is attributed to the notable overhead incurred by data transmission between nodes across the network.

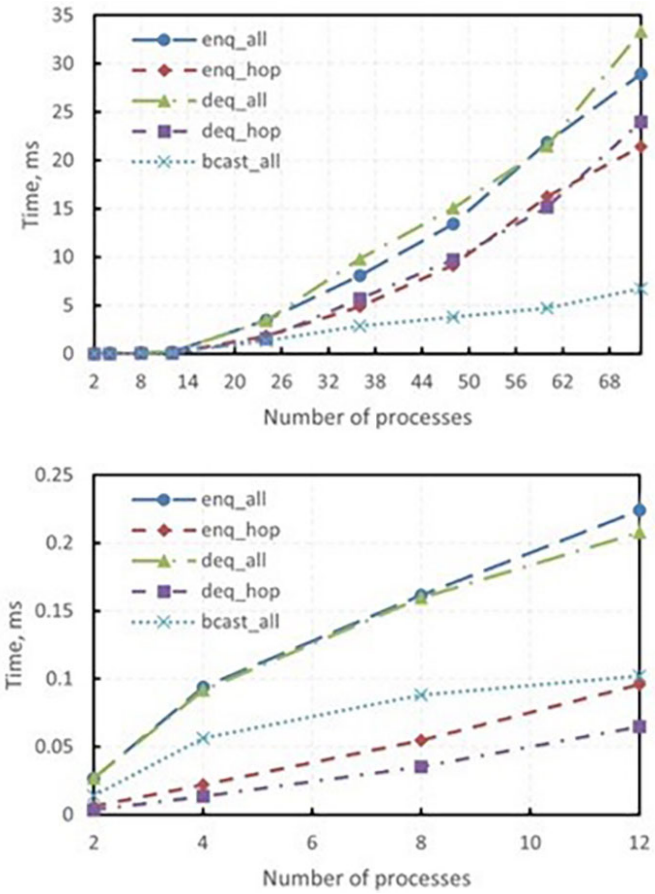


Fig. 8. Dependency of time t for adding and removing operations from the non-blocking queue on the number of processes p. The right graph presents an enlarged view of the scale from 2 to 12 processes.

4 Conclusion

In this paper, we introduce a lock-free distributed queue with decentralized storage of head and tail information within the MPI RMA model. The proposed queue demonstrates significantly superior throughput in comparison to its blocking counterparts. This achievement is primarily attributed to the non-blocking synchronization strategy employed and the reduction of potential bottlenecks through the decentralized storage of head and tail information. The scalability of the constructed data structure is moderate. Performance

analysis underscores that the majority of execution time is dedicated to the search for the current head or tail of the queue.

This research was supported by Russian Science Foundation (RSF) (project № 22-21-00686, https://rscf.ru/project/22-21-00686/).

References

1. J. Liu, J. Wu, D.K. Panda, Int. J. of Parallel Programming **32**, 167-198 (2004)
2. T. Hoefler et al., ACM Transactions on Parallel Computing **2**, 9 (2015)
3. V. Tipparaju, J. Nieplocha, D. Panda, *Fast collective operations using shared and remote memory access protocols on clusters*, Proc. Int. Parallel and Distributed Processing Symposium (IPDPS), 10 (2003)
4. R. Gerstenberger, M. Besta, T. Hoefler, Communications of the ACM **61(10)**, 106-113 (2018)
5. M. Herlihy, N. Shavit, *The art of multiprocessor programming* (Burlington, Morgan Kaufmann, 2011)
6. S. Haque, Z. Eberhart, A. Bansal, C. McMillan, *Semantic Similarity Metrics for Evaluating Source Code Summarization*, Int. Conf. on Program Comprehension, 36-47 (2022)
7. M. Mark, N. Shavit, *Concurrent Data Structures,* Handbook of Data Structures and Applications, 22 (2004)
8. N. Shavit, Communications of the ACM **54(7)**, 76-84 (2011)
9. A. Paznikov, Y. Shichkina, Information **9(1)**, 21 (2018)
10. I. Calciu, J. Gottschlich, M. Herlihy, *Using elimination and delegation to implement a scalable NUMA-friendly stack*, 5th USENIX Workshop on Hot Topics in Parallelism, 7 (2013)
11. M.M. Michael, *Safe memory reclamation for dynamic lock-free objects using atomic reads and writes*, Proc. of the twenty-first annual symposium on Principles of distributed computing, 21-30 (2002)
12. D. Dechev, P. Pirkelbauer, B. Stroustrup, *Lock-free Dynamically Resizable Arrays*, Principles of Distributed Systems 10th Int. Conf., 142-156 (2006)
13. A.V. Tabakov, A.A. Paznikov, *Algorithms for optimization of relaxed concurrent priority queues in multicore systems*, In 2019 IEEE Conf. of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), 360-365 (2019)
14. A.D. Anenkov, A.A. Paznikov, M.G. Kurnosov, *Algorithms for access localization to objects of scalable concurrent pools based on diffracting trees in multicore computer systems*, In 2018 XIV Int. Scientific-Technical Conf. on Actual Problems of Electronics Instrument Engineering (APEIE), 374-380 (2018)
15. B. Brock, A. Buluc, K. Yelick, *BCL A cross-platform distributed data structures library*, Proc. of the 48th Int. Conf. on Parallel Processing, 1-10 (2019)
16. J. Schuchart, A. Bouteiller, G. Bosilca, *Using MPI-3 RMA for active messages*, IEEE/ACM Workshop on Exascale MPI (ExaMPI), 47-56 (2019)
17. T.D. Diep, P.H. Ha, K. Fürlinger, J. of Parallel and Distributed Computing **173**, 48-60 (2023)
18. A.A. Paznikov, A.D. Anenkov, Procedia Computer Science **150**, 654-662 (2019)