ВЕСТНИК ТОМСКОГО ГОСУДАРСТВЕННОГО УНИВЕРСИТЕТА

2023 Управление, вычислительная техника и информатика № 62

Tomsk State University Journal of Control and Computer Science

МАТЕМАТИЧЕСКОЕ МОДЕЛИРОВАНИЕ

MATHEMATICAL MODELING

Научная статья
УДК 004.272
doi: 10.17223/19988605/62/2

Распределенная очередь без использования блокировок в модели удаленного доступа к памяти

Александр Викторович Бураченко[1], Алексей Александрович Пазников[2], Денис Павлович Державин[3]

[1, 2, 3] *Государственный электротехнический университет «ЛЭТИ», Санкт-Петербург, Россия*
[1] *ss47305@gmail.com*
[2] *apaznikov@gmail.com*
[3] *derzhavinden002@gmail.com*

Аннотация. При разработке программного обеспечения для распределенных вычислительных систем в стандарте MPI наравне с моделью передачи сообщений (message-passing) используется модель удаленного доступа к памяти (remote memory access, MPI RMA, RMA). Модель во многих случаях позволяет повысить эффективность и упростить разработку параллельных программ. В рамках RMA имеют место задачи синхронизации параллельных процессов и потоков при обеспечении доступа к разделяемым (распределенным) структурам данных. В системах с общей памятью для аналогичной задачи активно используется неблокирующая синхронизация (non-blocking), гарантирующая прогресс выполнения операций (lock-free, wait-free, obstruction-free). При таком подходе задержка выполнения операций одним процессом не останавливает выполнения остальных процессов. Мы предполагаем, что такой подход может быть эффективным и при построении распределенных структур данных в модели RMA. Нами рассматривается идея построения неблокируемых распределенных структур данных в RMA на примере очереди, описаны построенные алгоритмы для выполнения основных операций, исследуется эффективность структуры данных, приведено экспериментальное сравнение с блокируемыми аналогами.

Ключевые слова: распределенная очередь; неблокирующие структуры данных; удаленный доступ к памяти; MPI; MPI RMA; lock-free; one-sided communications.

Благодарности: Исследование выполнено за счет гранта Российского научного фонда № 22-21-00686, https://rscf.ru/project/22-21-00686/

***Для цитирования*:** Бураченко А.В., Пазников А.А., Державин Д.П. Распределенная очередь без использования блокировок в модели удаленного доступа к памяти // Вестник Томского государственного университета. Управление, вычислительная техника и информатика. 2023. № 62. С. 13–24. doi: 10.17223/19988605/62/2

Original article
doi: 10.17223/19988605/62/2

Lock-free distributed queue in remote memory access model

Alexander V. Burachenko[1], Alexei A. Paznikov[2], Denis P. Derzhavin[3]

[1, 2, 3] *Electrotechnical University "LETI", St. Petersburg, Russian Federation*
[1] *ss47305@gmail.com*

© А.В. Бураченко, А.А. Пазников, Д.П. Державин, 2023

[2] *apaznikov@gmail.com*
[3] *derzhavinden002@gmail.com*

Abstract. In parallel programming for distributed-memory systems in MPI standard, remote memory access model (one-sided communications, MPI RMA, 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 and threads accessing shared (concurrent, distributed) data structures. In shared-memory machines, non-blocking synchronization (lock-free, wait-free, obstruction-free) is widely used to solve the similar problem. In non-blocking synchronization, delays in execution of one process (thread) do not suspend execution of other threads. We suppose that this approach could also be effective in designing distributed data structures (in the RMA model particularly). In this article, we discuss the idea of building non-blocking distributed data structures in RMA model on the example of a queue, describe the designed algorithms of operations, investigate the efficiency, and provide an experimental comparison with lock-based counterparts.

Keywords: distributed queue; non-blocking concurrent data structures; remote memory access; MPI; MPI RMA; one-sided communications.

Acknowledgments: This research was supported by Russian Science Foundation (RSF) project 22-21-00686, https://rscf.ru/en/project/22-21-00686/

***For citation*:** Burachenko, A.V., Paznikov, A.A., Derzhavin, D.P. (2023) Lock-free distributed queue in remote memory access model. *Vestnik Tomskogo gosudarstvennogo universiteta. Upravlenie, vychislitelnaja tehnika i informatika – Tomsk State University Journal of Control and Computer Science*. 62. pp. 13–24. doi: 10.17223/19988605/62/2

Введение

Под распределенной вычислительной системой (ВС) будем понимать композицию элементарных машин (ЭМ), взаимодействующих через коммуникационную сеть. ЭМ могут быть процессорными ядрами, процессорами и узлами. При разработке программ для распределенных ВС наравне с моделью передачи сообщений (message-passing) широко применяются альтернативные модели программирования, например модель удаленного доступа к памяти (Remote Memory Access, MPI RMA, RMA) и модель разделенного глобального адресного пространства (Partitioned Global Address Space, PGAS).

Опишем более детально модель RMA, которая является одной из перспективных и реализована в стандарте MPI в виде подсистемы односторонних коммуникаций (one-sided communications) [1, 2]. В RMA процессы напрямую обращаются к памяти других процессов вместо отправки и получения сообщений. В отличие от PGAS (UPC, CAF, Chapel, X10), RMA тесно интегрирована с библиотеками MPI и может быть использована наравне с моделью передачи сообщений, а также реализует доступ к низкоуровневым примитивам коммуникации и синхронизации. Применение RMA часто позволяет сократить время выполнения программ по сравнению с моделями передачи сообщений или PGAS [3]. Оптимизация достигается, в частности, благодаря поддержке современными коммуникационными сетями (Infiniband, PERCS, Gemini, Aries и др.) технологии RDMA [4], реализующей обращение к удаленным сегментам памяти без участия центрального процессора.

Основными операциями в модели RMA являются неблокируемые MPI_Put (запись в удаленную память) и MPI_Get (чтение) и набор атомарных операций: MPI_Accumulate, MPI_Get_accumulate, MPI_Compare_and_swap и др. RMA-вызовы должны находиться внутри областей (эпох, epochs), в рамках которых выполняется синхронизация. RMA предоставляет активный и пассивный методы синхронизации. В настоящей работе применяется пассивный метод синхронизации (passive target synchronization) [2]: процесс открывает эпоху, затем выполняет RMA-операции для доступа к зарегистрированным сегментам памяти (окнам, window) других процессов. Таким образом, RMA-операции выполняются в одностороннем порядке, без явного участия других процессов, и сопровождаются меньшими накладными расходами, чем при активной синхронизации [5].

В рамках моделей RMA и PGAS имеет место задача обеспечения масштабируемого доступа к структурам данных, которые распределены между узлами ВС. Данная задача не является новой в параллельном программировании. Так, для систем с общей памятью имеет место необходимость в син-

хронизации потоков, обращающихся к разделяемым (concurrent, thread-safe) структурам данных. Такие структуры должны обеспечивать корректный (как правило линеаризуемый, linearizable) доступ параллельных потоков в произвольные моменты времени [4, 6, 7]. Эффективность синхронизации существенно влияет на время выполнения программ и обусловливает гарантии прогресса выполнения.

Существующие методы синхронизации можно разделить на два класса: с применением блокировок (locks) и без блокировок (non-blocking). Блокировки обладают простой семантикой и часто достаточно эффективны, однако неблокирующий подход обеспечивает гарантии выполнения и позволяет избежать тупиковых ситуаций (deadlocks), инверсий приоритетов (priority inversion) и некоторых других проблем блокировок. Неблокирующим называется такой алгоритм, в котором задержка выполнения одного потока не останавливает прогресс выполнения программы в целом (при соблюдении ряда условий) [4]. Значительная часть работ в области разделяемых структур данных направлена на создание средств синхронизации для ВС с общей памятью. К ним относятся алгоритмы блокировки потоков [4, 9] (TTS, Backoff, CLH, MCS, Oyama, Flat Combining, RCL и др.). Хотя некоторые методы (Hierarchical Locks, Cohorting и др.) учитывают иерархические уровни системы, они неприменимы в ВС с распределенной памятью. Неблокируемые структуры [4, 6–9] также разработаны для многоядерных ВС и неприменимы в распределенных ВС. Структуры, оптимизированные для NUMA [10], также неприменимы в распределенных средах.

Существует, однако, ряд работ по реализации распределенных неблокирующих структур: FastQueue, CircularQueue [11] и Active Message Queue [12]. Их общая проблема – централизация данных в памяти одного процесса. В результате подхода остальные процессы, работающие со структурой данных, постоянно обращаются к центральному процессу. Это является узким местом и может быть причиной снижения производительности очереди, особенно на большом числе процессов. В [13] рассматривается реализация распределенного неблокирующего стека на основе алгоритма Elimination-Backoff Stack. Экспериментальные оценки указывают на неплохую производительность на небольших подсистемах. Основным недостатком структуры также является ее централизованность.

В данной работе исследуется возможность построения неблокируемых (lock-free) структур данных для распределенных ВС на примере очереди. Предлагается подход на основе децентрализации данных между процессами параллельной программы. Мы предполагаем, что такой подход может быть эффективен при построении и других распределенных структур данных.

1. Неблокирующая распределенная очередь с децентрализацией хранения данных

1.1. Модель и структура очереди

Очередь – коллекция объектов, реализующая дисциплину FIFO («первым вошел – первым вышел»). Основные операции: добавление (insert, enqueue) элемента в последнюю позицию (хвост, tail, T) и извлечение (remove, dequeue) элемента из первой позиции (голова, head, H). Элементы распределенной очереди находятся в памяти процессов, выполняющихся на распределенной ВС.

Пусть имеется ВС из N ЭМ. На каждой ЭМ $i = 1, 2, \ldots, N$ имеется локальная оперативная память m_i, причем $m_i \cap m_j = \varnothing$, $\forall i, j \in \{1, 2, \ldots, N\}$. Считаем, что на каждой ЭМ i запущен процесс p_i.

Обозначим RMA-окно w, необходимое для выполнения операций. Процессы p_i, $i = 1, 2, \ldots, N$, обладают дескриптором окна w и публикуют в нем часть своей памяти m_i (назовем её m^*_i), таким образом предоставляя доступ к m^*_i остальным процессам.

Для хранения элементов очереди на каждом процессе выделена память под пул элементов (в текущей реализации – массив). Пусть e_i – пул элементов очереди в памяти m^*_i процесса p_i. Размер массива e_i фиксирован и равен K. При добавлении новых элементов процесс p_i использует ячейки из массива e_i: e_{ij}, $\forall i, j : i \in \{1, 2, \ldots, N\}$, $j \in \{1, 2, \ldots, N\}$. Тогда голова очереди H – это элемент e_{ij}, являющийся первым в очереди, а хвост очереди T – это элемент e_{ij}, являющийся последним в очереди.

Для определения местоположения элементов очереди введем понятие *ссылка на элемент* e_{ij} – пара чисел (i, j), которая однозначно идентифицирует элемент j распределенной очереди, который находится в памяти процесса i. Пустую ссылку обозначим $(\varnothing, \varnothing)$. Каждый элемент e_{ij} характеризуется:

– Временной меткой добавления в очередь – τ_{ij}.

– Пользовательскими данными – v_{ij}.

– Состоянием c_{ij}. Принимает одно из трех значений: F, A, D. F – элемент e_{ij} свободен для использования при следующем добавлении в очередь, A – элемент занят, находится в очереди, D – удален из очереди, но еще не доступен для повторного использования. Состояние элемента циклически меняется: из F в A при добавлении, из A в D при удалении, из D в F при освобождении.

– Ссылкой на следующий элемент – n_{ij}. Если за e_{ij} следует e_{gf}, то $n_{ij} = (g, f)$. При добавлении e_{ij} в очередь $n_{ij} = (\varnothing, \varnothing)$.

– Ссылкой на себя l_{ij}. Для e_{ij}, $l_{ij} = (i, j)$. Данная ссылка необходима для возможности обновления информации об элементе.

Также каждый процесс p_i имеет в своей памяти m^*_i ссылки на голову h_i и хвост t_i очереди, $\forall i \in \{1, 2, \ldots, N\}$, которые служат для определения текущего положения актуальных головы H и хвоста T.

Процесс p_1 дополнительно обладает ссылкой s – элемент для достижения консенсуса при добавлении первого элемента. Когда несколько процессов одновременно добавляют новый элемент в очередь, они принимают решение, чей элемент будет добавлен. Для этого применяется операция «сравнение с обменом» (Compare-And-Swap, CAS) с текущего хвоста T на новый хвост n_{ij}. Но сразу после инициализации очередь пуста: она не обладает ни головой, ни хвостом. Поэтому необходима область памяти, в которой можно было бы решить, чей элемент будет первым добавлен в очередь.

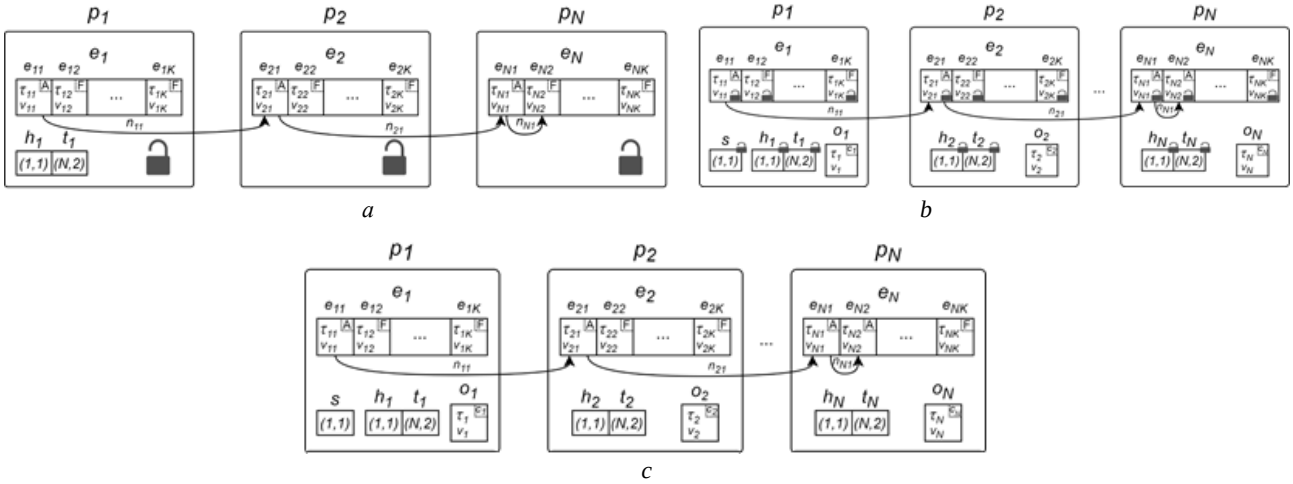

Рис. 1. Структура распределенной очереди. *a* – блокирующая, с централизованным хранением информации о голове / хвосте, *b* – блокирующая, децентрализованная, *c* – неблокирующая, децентрализованная
Fig. 1. The structure of a distributed queue. *a* – lock-based, with centralized head/tail information storage, *b* – lock-based, decentralized, *c* – non-blocking, decentralized

В рамках централизованного подхода информация о положении головы и хвоста находится в памяти главного процесса p_1 (рис. 1, *a*). Процессы p_i при добавлении или удалении элемента обновляют ссылки t_1 или h_1 соответственно. Такой подход порождает узкое место (bottleneck), так как p_i постоянно обращаются к p_1, что ведет к снижению пропускной способности (throughput) структуры данных.

1.2. Децентрализованный метод организации очереди

Предлагается подход с децентрализованным хранением информации о положении H и T (рис. 1, *в*). Каждый процесс p_i обладает локальной ссылкой h_i и t_i. При поиске актуальных головы или хвоста

процесс p_i обращается к ссылке h_i или t_i соответственно. После вставки или удаления процесс p_i обновляет ссылки h_j и t_j (j = 1, 2, …, N) для всех остальных процессов согласно разработанному *алгоритму оповещения*, в рамках которого p_i сначала обновляет свои ссылки h_i и t_i, затем ссылки на остальных процессах: h_j и t_j $\forall\, j \in \{1, 2, \ldots, N\}$, $j \neq i$. Алгоритм работает по трем методам: «только голова», «только хвост» и «сначала хвост, затем голова». В зависимости от метода алгоритм обновляет либо ссылки h_j, либо t_j, либо t_j затем h_j для каждого процесса. Подробнее алгоритм рассмотрен в разд. 1.5.

При удалении элементов они переходят в состояние D, т.е. считаются удаленными, но еще не доступны для повторного использования. Для освобождения элементов e_{ij}, т.е. перевода их в состояние F, разработан *алгоритм очистки* пула e_i (см. разд. 1.6). Алгоритм использует дополнительный элемент o_i. Этим элементом процессы оперируют при чтении элементов очереди: в него копируется содержимое полей элемента, с которым работает процесс p_i в данный момент. o_i находится в памяти m^*_i и обладает теми же атрибутами, что и e_{ij} (для o_i обозначения полей: τ_i, v_i, c_i, n_i, l_i), но не входит в состав пула e_i. Так как важна точность временных меток τ_{ij} и их согласованность, часы процессов синхронизируются.

1.3. Операция добавления элемента в очередь

Опишем операцию добавления (enqueue) (рис. 2). Под *atomic_get* здесь и далее понимается атомарная операция удаленного чтения на основе MPI_Get_accumulate. Операция CAS (MPI_Compare_and_swap) позволяет разрешать консенсус при изменении данных в общем участке памяти для любого числа процессов, модифицирующих очередь. Операции *bcast_t* и *bcast_th* – это вызовы алгоритма оповещения по схемам «только хвост» и «сначала хвост, затем голова».

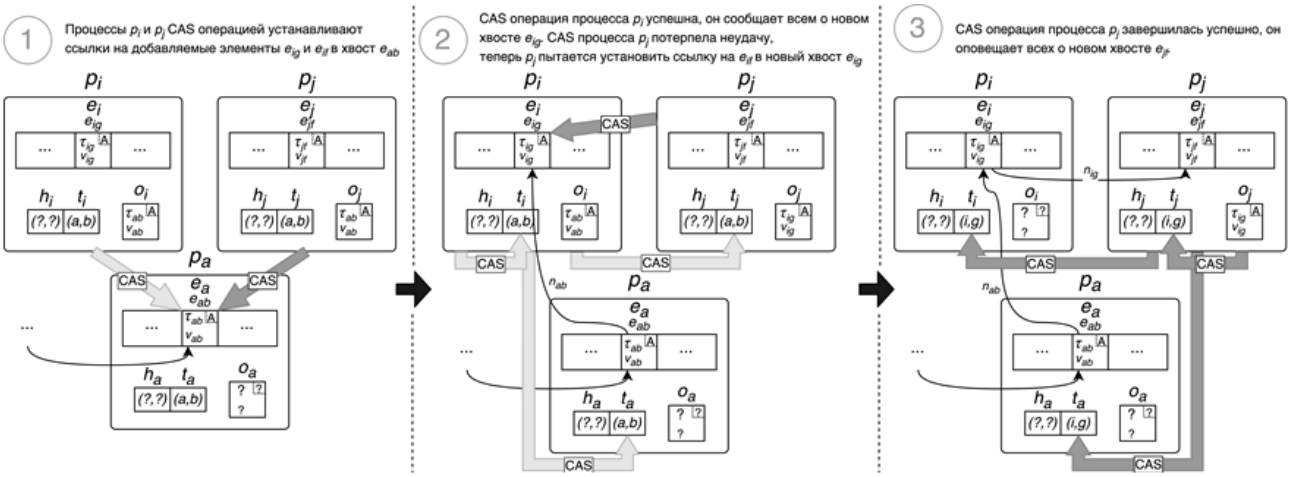

Рис. 2. Схема добавления элемента в очередь
Fig. 2. Scheme of adding an item to the queue

Алгоритм добавления в очередь процессом p_i (табл. 1, *a*):

1. Инициализация добавляемого элемента e_{ij}. Выбирается свободная ячейка e_{ij} в массиве e_i, в ячейку записываются данные (v_{ij} = *val*, c_{ij} = A, n_{ij} = (∅, ∅)) (строка 1).

2. Если массив e_i переполнен, запускается алгоритм очистки. Если после отработки алгоритма очистки свободной ячейки нет, завершить выполнение алгоритма ошибкой (строки 1, 2).

3. Если ссылка на хвост t_i = (∅, ∅), получить ссылку s (строки 4, 5). Иначе – перейти к шагу 7.

4. Если s = (∅, ∅), установить временную метку τ_{ij} в e_{ij}, установить в s ссылку (i, j) CAS-операцией (строки 6–8). Иначе – перейти к шагу 6.

5. Если CAS завершилась успешно (точка линеаризации), оповестить все процессы о новых хвосте и голове и завершить алгоритм (строки 9, 10). Иначе – обновить s (строка 12) и перейти к шагу 6.

6. Записать s в t_i (строка 14).

7. Получить элемент по ссылке t_i и записать его в o_i (строка 16).

8. Если статус c_i = A (строка 17), перейти к шагу 9. Иначе – перейти к шагу 12.

9. Если $n_i = (\varnothing, \varnothing)$, установить временную метку τ_{ij} в e_{ij}, установить в n_i ссылку (i, j) CAS-операцией (строки 18–20). Иначе – перейти к шагу 11.

10. Если CAS завершилась успешно (точка линеаризации), оповестить процессы о новом хвосте e_{ij} (строка 21) и завершить алгоритм (строка 22). Иначе – обновить o_i и перейти к шагу 11 (строка 24).

11. Получить элемент по ссылке n_i, записать его в o_i и перейти к шагу 8 (строка 26).

12. Если статус c_i = D, перейти к шагу 14 (строка 29). Иначе – перейти к шагу 3 (строка 41).

13. Если $n_i = (\varnothing, \varnothing)$, установить метку τ_{ij} в e_{ij}, установить в n_i ссылку (i, j) CAS-операцией (строки 30–32). Иначе – получить элемент по ссылке n_i, записать его в o_i и перейти к шагу 8 (строки 37, 38).

14. Если CAS-операция завершилась успешно (точка линеаризации), оповестить все процессы о новом хвосте и новой голове e_{ij} и завершить выполнение алгоритма (строки 33, 34).

Таблица 1

Алгоритмы выполнения операций для распределенной неблокирующей очереди: *a* – операция enqueue добавления элемента в очередь, *b* – операция dequeue извлечения элемента из очереди

a

Входные данные: *val* – добавляемые данные; *win* – окно для выполнения RMA-операций

```
1  if init_elem(val, &eij) == false then
2    return BUFFER_FULL
3  end if
4  if ti == (-1, -1) then
5    s = atomic_get(s, win)
6    if s == (-1, -1) then
7      eij.τij = get_ts()
8      if cas(s, (-1, -1), (i, j), win) then
9        bcast_th(eij, win)
10       return SUCCESS
11     end if
12     s = atomic_get(s, win)
13   end if
14   ti = s
15 end if
16 oi = atomic_get(ti, win)
17 if oi.ci == A then
18   if oi.ni == (-1, -1) then
19     eij.τij = get_ts()
20     if cas(oi.ni, (-1, -1), (i, j), win) then
21       bcast_t(eij, win)
22       return SUCCESS
23     end if
24     oi = atomic_get(oi.li, win)
25   end if
26   oi = atomic_get(oi.ni, win)
27   goto 17
28 end if
29 if oi.ci == D then
30   if oi.ni == (-1, -1) then
31     eij.τij = get_ts()
32     if cas(oi.ni, (-1, -1), (i, j), win) then
33       bcast_th(eij, win)
34       return SUCCESS
35     end if
36   else
37     oi = atomic_get(oi.ni, win)
38     goto 17
39   end if
40 end if
41 goto 4
```

b

Входные данные: *win* – окно для выполнения RMA-операций

```
1  if hi == (-1, -1) then
2    s = atomic_get(s, win)
3    if s == (-1, -1) then
4      return QUEUE_EMTPY
5    end if
6    hi = s
7  end if
8  oi = atomic_get(hi, win)
9  if oi.ci == A then
10   if cas(oi.ci, A, D, win) then
11     val = oi.vi
12     oi = atomic_get(oi.li, win)
13     if oi.ni != (-1, -1) then
14       bcast_h(oi.ni, win)
15     end if
16     return SUCCESS
17   end if
18   oi = atomic_get(oi.li, win)
19 end if
20 if oi.ci == D then
21   if oi.ni != (-1, -1) then
22     oi = atomic_get(oi.ni, win)
23     goto 9
24   else
25     return QUEUE_EMPTY
26   end if
27 end if
28 goto 1
```

1.4. Операция удаления элементов

Опишем алгоритм удаления (dequeue) элементов. Операция *bcast_h* – это вызов алгоритма оповещения с методом «только голова». Алгоритм удаления из очереди процессом p_i представлен на рис. 3, в табл. 1, *б*).

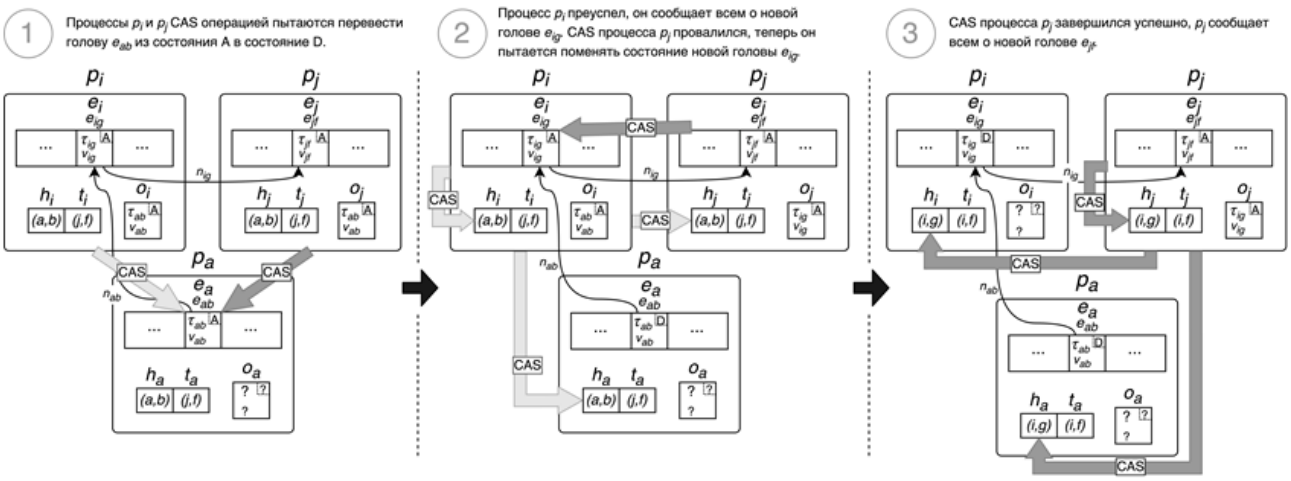

Рис. 3. Схема удаления элемента из очереди
Fig. 3. Scheme of removing an item out of the queue

1. Если ссылка на голову $h_i = (\varnothing, \varnothing)$, получить ссылку s (строки 1, 2). Иначе – перейти к шагу 4.
2. Если $s = (\varnothing, \varnothing)$, значит очередь пуста, завершить выполнение алгоритма (строки 3, 4).
3. Записать s в h_i (строка 6).
4. Получить элемент по ссылке h_i и записать его в o_i (строка 8).
5. Если статус c_i = A, изменить c_i на D операцией CAS (строки 9, 10). Иначе – перейти к шагу 8.
6. Если CAS-операция завершилась успешно (точка линеаризации), перейти к шагу 7. Иначе – обновить o_i и перейти к шагу 8 (строки 18–20).
7. Если $n_i \neq (\varnothing, \varnothing)$, оповестить все процессы о новой голове (элемент по ссылке n_i) (строки 12–16). Завершить выполнение алгоритма.
8. Если статус c_i = D, перейти к шагу 9 (строка 20). Иначе – перейти к шагу 1 (строка 28).
9. Если $n_i \neq (\varnothing, \varnothing)$, получить элемент по ссылке n_i, записать его в o_i и перейти к шагу 5 (строки 21–23). Иначе очередь пуста – завершить выполнение алгоритма (строка 25).

1.5. Алгоритм оповещения (bcast)

Каждый процесс p_i обладает ссылками на голову h_i и хвост t_i. При изменении головы или хвоста необходимо актуализировать ссылки h_j, t_j, $\forall j \neq i$, всех процессов – для этого используется алгоритм оповещения *bcast*. Алгоритм выполняется в соответствии с тремя схемами (*bcast_method*):

1. Только голова. Обновляются ссылки на хвост t_j. Вызывается после того, как процесс добавил элемент в очередь.
2. Только хвост. Обновляются ссылки на голову h_j. Вызывается после того, как процесс удалил элемент.
3. Сначала хвост, затем голова. Обновляются ссылки t_j и h_j. Вызывается после того, как процесс добавил элемент в очередь вслед за элементом, находящимся в состоянии D.

Алгоритм *bcast* возвращает результат выполнения: SUCCESS – ссылка успешно обновлена, FAIL – ссылка не обновлена (означает, что ее актуализировал другой процесс). Если метод возвращает код FAIL, это признак остановки алгоритма оповещения, так как распространяемый элемент уже не является актуальным хвостом или головой, а актуальную информацию раздает другой процесс.

Ниже описан шаблон алгоритма оповещения для процесса p_i (табл. 2, *a*). На вход передается тиражируемый элемент e_{ij}, метод оповещения *bcast_method* и окно для выполнения RMA-операций *win*.

1. Сформировать массив *b* рангов процессов, работающих с очередью (строка 1).

2. Обновить h_i и t_i согласно указанному методу *bcast_method* (строка 2). Если *bcast_method* вернул код FAIL, завершить выполнение алгоритма (строки 2, 3). Иначе – удалить ранг *i* из *b* (строка 5).

3. Если массив *b* пуст, завершить выполнение алгоритма (строки 7, 8). Иначе – перейти к шагу 4.

4. Случайно выбрать процесс p_a из оставшихся в *b* (строка 10). Обновить ссылки h_a и t_a согласно указанному методу *bcast_method* (строка 11). Если *bcast_method* вернул код FAIL – завершить выполнение алгоритма (строки 11, 12). Иначе – удалить ранг *a* из *b* (строка 14) и перейти к шагу 3.

Таблица 2

Алгоритмы оповещения для распределенной неблокирующей очереди: *a* – шаблон алгоритма оповещения, *b* – метод «только голова», *c* – метод «только хвост», *d* – метод «сначала хвост, затем голова»

a

Входные данные: e_{ij} – тиражируемый элемент; *bcast_method* – метод оповещения; *win* – окно для выполнения RMA-операций

```
1  b = get_all_ranks()
2  if bcast_method(i, e_ij, win) == FAIL then
3    return
4  end if
5  exclude_rank(i, &b)
6  while true
7    if is_empty(b) then
8      return
9    end if
10   a = get_next_rank_rand(b)
11   if bcast_method(a, e_ij, win) == FAIL then
12     return
13   end if
14   exclude_rank(a, &b)
15 end
```

b

Входные данные: *a* – ранг оповещаемого процесса; e_{ij} – тиражируемый элемент; *win* – окно для выполнения RMA-операций

```
1  do
2    h_a = atomic_get(a, win)
3    if h_a != (-1, -1) then
4      o_i = atomic_get(h_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(h_a, h_a, (i, j), win) == false
10 return SUCCESS
```

c

Входные данные: *a* – ранг оповещаемого процесса; e_{ij} – тиражируемый элемент; *win* – окно для выполнения RMA-операций

```
1  do
2    t_a = atomic_get(a, win)
3    if t_a != (-1, -1) then
4      o_i = atomic_get(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
```

d

Входные данные: *a* – ранг оповещаемого процесса; e_{ij} – тиражируемый элемент; *check_head* – флаг требования обновить ссылку на голову; *check_tail* – флаг требования обновить ссылку на хвост; *win* – RMA-окно

```
1  if check_tail then
2    do
3      t_a = atomic_get(a, win)
4      if t_a != (-1, -1) then
5        o_i = atomic_get(t_a, win)
6        if e_ij.τ_ij < o_i.τ_i != (-1, -1) then
7          check_tail = 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 check_head then
14   do
15     h_a = atomic_get(a, win)
16     if h_a != (-1, -1) then
17       o_i = atomic_get(h_a, win)
18       if e_ij.τ_ij < o_i.τ_i != (-1, -1) then
19         check_tail = 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 check_tail == false and check_head = false then
26   return FAIL
27 end if
28 return SUCCESS
```

Рассмотрим подробнее алгоритмы методов обновления в памяти *j*-го процесса ссылок на голову и / или хвост. Метод «только голова». Исходные данные: номер процесса *a*, тиражируемый элемент e_{ij} и RMA-окно *win*. Алгоритм тогда будет выглядеть (табл. 2, *b*):

1. Получить ссылку на голову h_a из памяти процесса *a* (строка 2).

2. Если $h_a \neq (\varnothing, \varnothing)$, получить элемент по ссылке h_a и записать его в o_i (строки 3, 4). Иначе – к шагу 4.

3. Если метка $\tau_{ij} < \tau_j$, вернуть FAIL и завершить алгоритм (строки 5, 6). Иначе – переход к шагу 4.

4. С помощью CAS установить ссылку (*i*, *j*) в h_a (строка 9). Если операция завершилась успешно (точка линеаризации), вернуть SUCCESS и завершить алгоритм (строка 10). Иначе перейти к шагу 1.

Метод «только хвост». Исходные данные те же. Алгоритм (табл. 2, *c*):

1. Получить ссылку на хвост t_a из памяти процесса *a* (строка 2).

2. Если $t_a \neq (\varnothing, \varnothing)$, получить элемент по ссылке t_a и записать его в o_i (строки 3, 4). Иначе – к шагу 4.

3. Если метка $\tau_{ij} < \tau_j$, вернуть код FAIL и завершить алгоритм (строки 5, 6). Иначе – к шагу 4.

4. С помощью CAS установить ссылку (*i*, *j*) в t_a (строка 9). Если операция завершилась успешно (точка линеаризации), вернуть SUCCESS и завершить алгоритм (строка 10). Иначе – переход к шагу 1.

Метод «сначала хвост, затем голова». Данные: номер процесса *a*, элемент e_{ij}, флаги *check_head* (обновлять ли ссылку на голову) и *check_tail* (обновлять ли ссылку на хвост). Параметры *check_head* и *check_tail* нужны, так как в какой-то момент e_{ij} может перестать быть актуальным хвостом (головой), и нужно иметь возможность продолжить алгоритм, не проверяя ссылки. Алгоритм (табл. 2, *d*):

1. Если *check_tail* = *true* перейти к пункту 2 (строка 1). Иначе – перейти к пункту 6.

2. Получить ссылку на хвост t_a из памяти процесса *a* (строка 3).

3. Если $t_a \neq (\varnothing, \varnothing)$, получить элемент по ссылке t_a и записать в o_i (строки 4, 5). Иначе – к шагу 5.

4. Если метка $\tau_{ij} < \tau_j$, записать false в *check_tail* и перейти к шагу 6 (строки 6–8). Иначе – к шагу 5.

5. CAS-операцией установить ссылку (*i*, *j*) в t_a (строка 11). Если операция завершилась успешно (точка линеаризации), перейти к пункту 6. Иначе – перейти к шагу 2.

6. Если *check_head* = true, перейти к шагу 7 (строка 13). Иначе – перейти к шагу 11.

7. Получить ссылку на голову h_a из памяти процесса *a* (строка 15).

8. Если $h_a \neq (\varnothing, \varnothing)$, получить элемент по ссылке h_a и записать в o_i (строки 16–17). Иначе – к шагу 10.

9. Если метка $\tau_{ij} < \tau_j$, записать *false* в *check_head* и перейти к шагу 11 (строки 18–20). Иначе – к шагу 10.

10. CAS-операцией установить ссылку (*i*, *j*) в h_a (строка 23). Если операция завершилась успешно (точка линеаризации), перейти к шагу 11. Иначе – перейти к шагу 7.

11. Если *check_tail* = false и *check_head* = false, вернуть код FAIL (строки 25–26). Иначе вернуть код SUCCESS и завершить выполнение алгоритма (строка 28).

1.6. Алгоритм очистки (cleaning)

Элементы e_{ij} переходят из состояния F в A при добавлении, затем из A в D при удалении. В какой-то момент в массиве e_i не остается доступных элементов (в состоянии F), и последующие операции добавления завершаются ошибкой. Для избегания ситуации применяется *алгоритм очистки*: он переводит элементы из D в F и вызывается, когда при поиске свободной ячейки в пуле e_i не остается свободных элементов при добавлении в очередь. Алгоритм состоит из двух частей: 1) определение минимальной временной метки τ_{min}, элементы с которой сейчас используются процессами; 2) перевод всех элементов в массиве e_i в состояние D с меткой меньше, чем τ_{min}. Опишем алгоритм (для p_i):

1. Сформировать массив *b* рангов процессов, работающих с очередью. Получить элемент по ссылке t_i и записать его в o_i. Записать τ_i в τ_{min}. Получить элемент по ссылке h_i и записать его в o_i. Если $\tau_i < \tau_{min}$, то записать τ_i в τ_{min}. Удалить ранг *i* из массива *b*.

2. Если массив *b* пуст, перейти к пункту 4. Иначе – перейти к пункту 3.

3. Случайно выбрать процесс p_a из оставшихся в массиве *b*. Получить элемент по ссылке h_a и записать его в o_i. Если $\tau_i < \tau_{min}$, то записать τ_i в τ_{min}. Получить элемент по ссылке t_a и записать его в o_i.

Если $\tau_i < \tau_{min}$, то записать τ_i в τ_{min}. Получить элемент o_a и записать его в o_i. Если $\tau_i < \tau_{min}$, то записать τ_i в τ_{min}. Удалить ранг a из массива b. Перейти к пункту 2.

4. Пройти по массиву e_i . Для каждого e_{ij}: если c_{ij} = D и $\tau_{ij} < \tau_{min}$, записать F в c_{ij}. Завершить выполнение алгоритма.

2. Проведение экспериментов

Экспериментальное исследование проводилось на кластере Информационно-вычислительного центра Новосибирского государственного университета. В экспериментах использовались 6 узлов по два 6-ядерных процессора Intel Xeon X5670 на каждом (суммарно 72 ядра). MPI: Open MPI 4.1.0.

Был разработан синтетический тест, выполняющий n = 10 000 операций вставки / удаления (тип операции выбирается равновероятно). Число процессов N варьировало от 2 до 72. Измерялась пропускная способность $b = n \times p / t$, где t – время проведения эксперимента, p – количество процессов.

Реализованная распределенная неблокирующая очередь сравнивалась с двумя очередями на основе блокировок. Первая – блокирующая централизованная очередь (см. рис. 1, *a*): информация о местоположении головы / хвоста находится в памяти процесса 0. Блокировка устанавливается на процесс. Перед чтением / изменением информации о голове / хвосте или элемента очереди из памяти некоторого процесса i читающий / изменяющий данные процесс захватывает блокировку процесса i; после окончания работы – освобождает. Вторая – блокирующая очередь с децентрализованным хранением положения головы / хвоста (см. рис. 1, *b*). Она аналогична неблокирующей, за исключением использования блокировок. В отличие от централизованной очереди, здесь блокируется не процесс, а отдельные элементы. Перед работой с элементом или ссылкой на голову / хвост процесс захватывает блокировку на этом элементе, после работы освобождает.

Пропускная способность разработанной очереди значительно превосходит пропускную способность блокирующих аналогов (рис. 4). Подход lock-free избавляет от накладных расходов на захват и освобождение блокировки и позволяет потоку в случае неудачи операции сразу повторить попытку, не затрачивая время на ожидание. Блокирующая децентрализованная очередь производительнее централизованной: децентрализация позволяет распределить нагрузку на процессы и повышает пропускную способность. Все рассматриваемые очереди масштабируются в умеренной степени. С ростом числа процессов пропускная способность структур данных падает, тем не менее в случае неблокируемой очереди остается на достаточно высоком уровне.

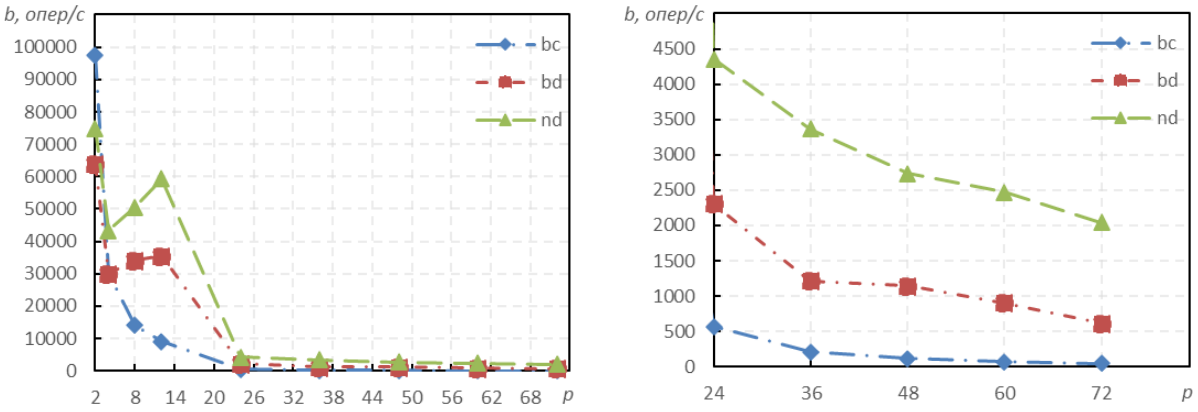

Рис. 4. Пропускная способность очередей (*bc* – блокирующая централизованная, *bd* – блокирующая децентрализованная, *nd* – неблокирующая децентрализованная)

Fig. 4. Throughput of queues (*bc* – blocking centralized, *bd* – blocking decentralized, *nd* – non-blocking decentralized)

Для оценки времени выполнения операций добавления и удаления для неблокирующей очереди выполнялось измерение времени реализации операций и их ключевых этапов. Каждый процесс самостоятельно выполняет измерения с последующей отправкой результатов процессу 0, который вычисляет средние значения. Перечень измерений:

1) общее время операции добавления (enq_all);
2) время поиска хвоста при добавлении (enq_hop);
3) общее время операции удаления (deq_all);
4) время поиска головы при удалении (deq_hop);
5) общее время алгоритма оповещения (bcast_all).

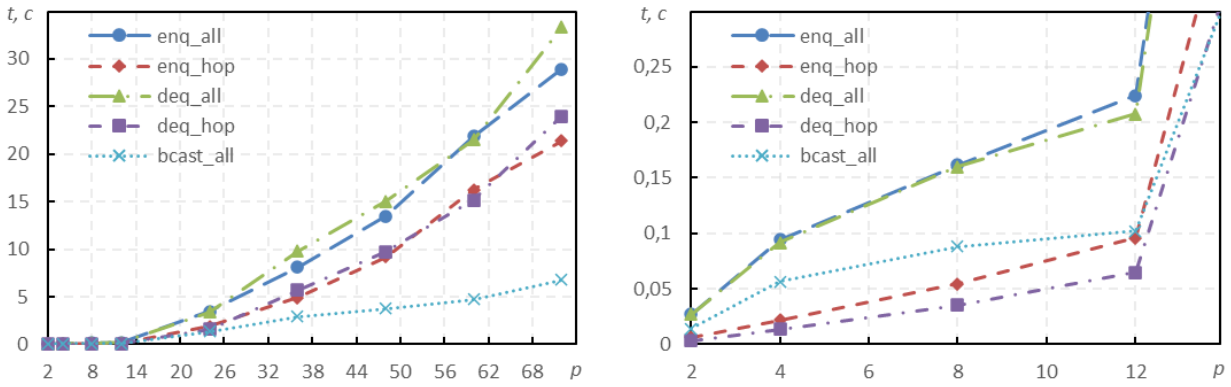

Рис. 5. Время операций добавления и удаления из неблокирующей очереди
Fig. 5. Time of operations of inserting and removing for the non-blocking queue

Большая часть времени уходит на поиск головы / хвоста (рис. 5). Объясняется это тем, что с ростом числа процессов увеличивается число операций с очередью в единицу времени. Отсюда растет вероятность для процесса p_i получить по ссылке t_i элемент, который уже не является хвостом. Тогда процессу приходится переходами по ссылкам на следующие элементы добираться до хвоста. Чем больше операций добавления в очередь выполняется в единицу времени, тем больше времени тратится на поиск хвоста. Та же проблема и с операцией удаления. Как и следовало ожидать, в пределах одного узла (до 12 процессов) операции выполняются достаточно быстро, а с увеличением числа узлов время растет. Существенное влияние оказывают накладные расходы на передачу данных между узлами.

Заключение

В данной статье разработан алгоритм распределенной неблокирующей (lock-free) очереди с децентрализацией хранения информации о положении головы и хвоста в модели MPI RMA. Очередь характеризуется значительно большей пропускной способностью по сравнению с блокирующими аналогами. Оптимизация достигается за счет неблокируемого подхода к синхронизации и уменьшения вероятности появления узких мест при выполнении операций благодаря децентрализации. Созданная структура данных умеренно масштабируется. Выполнен анализ эффективности предложенного алгоритма.

Список источников

1. Liu J., Wu J., Panda D.K. High performance RDMA-based MPI implementation over InfiniBand // International Journal of Parallel Programming. 2004. V. 32. P. 167–198.
2. Hoefler T., Dinan J., Thakur R., Barrett B., Balaji P., Gropp W., Underwood K. Remote memory access programming in MPI-3 // ACM Transactions on Parallel Computing. 2015. V. 2 (2). Art. 9. 26 p.
3. Gerstenberger R., Besta M., Hoefler T. Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided // Scientific Programming. 2014. V. 2, is. 2. P. 75–91.
4. Herlihy M., Shavit N. The art of multiprocessor programming. Amsterdam et al. : Morgan Kaufmann, 2012. 537 p.
5. Schuchart J., Niethammer C., Gracia J., Bosilca G. Quo Vadis MPI RMA? Towards a More Efficient Use of MPI One-Sided Communication // arXiv: 2111.08142. 2021.
6. Mark M., Shavit N. Concurrent Data Structures. Chapman and Hall/CRC Press, 2004. 32 p.
7. Shavit N. Data structures in the multicore age // Communications of the ACM. 2011. V. 54. P. 76–84.
8. Пазников А.А. Оптимизация делегирования выполнения критических секций на выделенных процессорных ядрах // Вестник Томского государственного университета. Управление, вычислительная техника и информатика. 2017. № 38. С. 52–58.

9. Аненков А.Д., Пазников А.А. Алгоритмы оптимизации масштабируемого потокобезопасного пула на основе распределяющих деревьев для многоядерных вычислительных систем // Вестник Томского государственного университета. Управление, вычислительная техника и информатика. 2017. № 39. С. 73–84.
10. Calciu I., Gottschlich J., Herlihy M. Using elimination and delegation to implement a scalable NUMA-friendly stack // 5th {USENIX} Workshop on Hot Topics in Parallelism (HotPar 13). 2013. P. 1–7.
11. Brock B., Buluc A. BCL: A cross-platform distributed data structures library // ICPP. 2019. P. 1–10.
12. Schuchart J., Bouteiller A., Bosilca G. Using MPI-3 RMA for active messages // ExaMPI. 2019. P. 47–56.
13. Diep T.D., Fürlinger K. Nonblocking data structures for distributed-memory machines: stacks as an example // 2021 29th Euromicro Int. Conf on Parallel, Distributed and Network-Based Processing (PDP). IEEE, 2021. С. 9–17.

References

1. Liu, J., Wu, J. & Panda, D.K. (2004) High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*. 32. pp. 167–198.
2. Hoefler, T., Dinan, J., Thakur, R., Barrett, B., Balaji, P., Gropp, W. & Underwood, K. (2015) Remote memory access programming in MPI-3. *ACM Transactions on Parallel Computing*. 2(2). p. 9.
3. Gerstenberger, R., Besta, M. & Hoefler, T. (2004) Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided. *Scientific Programming*. 2(2). pp. 75–91.
4. Herlihy, M. & Shavit, N. (2012) *The art of multiprocessor programmi*ng. Amsterdam et al.: Morgan Kaufmann.
5. Schuchart, J., Niethammer C., Gracia J. & Bosilca G. (2021) Quo Vadis MPI RMA? Towards a More Efficient Use of MPI One-Sided Communication. *arXiv*. 2111.08142.
6. Mark, M. & Shavit, N. (2004) *Concurrent Data Structures*. Chapman and Hall/CRC Press.
7. Shavit, N. (2001) Data structures in the multicore age. *Communications of the ACM*. 54. pp. 76–84.
8. Paznikov A.A. (2017) Optimization method of remote core locking. *Vestnik Tomskogo gosudarstvennogo universiteta. Upravlenie, vychislitel'naya tekhnika i informatika – Tomsk State University Journal of Control and Computer Science*. 38. pp. 52–58. DOI: 10.17223/19988605/38/8
9. Anenkov, A.D. & Paznikov, A.A. (2017) Algorithms of optimization of scalable thread-safe pool based on diffracting trees for multicore computing systems. *Vestnik Tomskogo gosudarstvennogo universiteta. Upravlenie, vychislitel'naya tekhnika i informatika – Tomsk State University Journal of Control and Computer Science*. 39. pp. 73–84. DOI: 10.17223/19988605/39/10
10. Calciu, I., Gottschlich, J. & Herlihy, M. (2013) Using elimination and delegation to implement a scalable NUMA-friendly stack. *5th {USENIX} Workshop on Hot Topics in Parallelism (HotPar 13)*. pp. 1–7.
11. Brock, B. &, Buluc, A. (2019) BCL A cross-platform distributed data structures library. *ICPP*. pp. 1–10.
12. Schuchart, J., Bouteiller, A. & Bosilca, G. (2019) Using MPI-3 RMA for active messages: *ExaMPI.* pp. 47–56.
13. Diep, T.D. & Fürlinger, K. (2021) Nonblocking data structures for distributed-memory machines: stacks as an example. *29th Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP).* IEEE. pp. 9–17.

***Информация об авторах*:**
Бураченко Александр Викторович – магистрант Санкт-Петербургского государственного электротехнического университета «ЛЭТИ» (Санкт-Петербург, Россия). E-mail: ss47305@gmail.com
Пазников Алексей Александрович – кандидат технических наук, доцент Санкт-Петербургского государственного электротехнического университета «ЛЭТИ» (Санкт-Петербург, Россия). E-mail: apaznikov@gmail.com
Державин Денис Павлович – магистрант Санкт-Петербургского государственного электротехнического университета «ЛЭТИ» (Санкт-Петербург, Россия). E-mail: derzhavinden002@gmail.com

Вклад авторов: все авторы сделали эквивалентный вклад в подготовку публикации. Авторы заявляют об отсутствии конфликта интересов.

***Information about the authors*:**
Burachenko Alexander V. (Saint Petersburg Electrotechnical University "LETI", St. Petersburg, Russian Federation). E-mail: ss47305@gmail.com
Paznikov Alexei A. (Candidate of Technical Sciences, Associate Professor, Saint Petersburg Electrotechnical University "LETI", St. Petersburg, Russian Federation). E-mail: apaznikov@gmail.com
Derzhavin Denis P. (Saint Petersburg Electrotechnical University "LETI", St. Petersburg, Russian Federation). E-mail: derzhavinden002@gmail.com

Contribution of the authors: the authors contributed equally to this article. The authors declare no conflicts of interests.

Поступила в редакцию 19.08.2022; принята к публикации 01.03.2023

Received 19.08.2022; accepted for publication 01.03.2023