УДК 004.272.2

А. В. Табаков, А. А. Пазников
Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

Алгоритмы оптимизации потокобезопасных очередей с приоритетом на основе ослабленной семантики выполнения операций

При разработке масштабируемых потокобезопасных структур данных (concurrent data structures) для вычислительных систем с общей памятью перспективным является подход на основе ослабления порядка выполнения операций. Построенные на его основе ослабленные структуры данных (relaxed data structures) относятся к нелинеаризуемым (non-linearizable) и не ориентированы на обеспечение строгой семантики выполнения операций (порядок выполнения операции FIFO/LIFO для линейных списков, извлечение максимального (минимального) элемента для очередей с приоритетом и т. д.). В статье используется подход, основанный на представлении потокобезопасных структур данных в виде множества простых структур, распределенных между потоками. При выполнении операций (вставки, удаления элементов) случайным образом выбирается подмножество данных структур. Построены алгоритмы оптимизации потокобезопасных ослабленных очередей с приоритетом на основе данного подхода. Созданы алгоритмы оптимизации выбора очередей из множества при выполнении операций вставки и удаления элементов, а также предложен алгоритм балансировки элементов в очередях. Алгоритмы учитывают иерархическую структуру многоядерных вычислительных систем и обеспечивают локализацию доступа к данным за счет сокращения подмножества очередей для случайного выбора. Оптимизированные алгоритмы добавления и удаления элементов из очередей с приоритетом позволяют увеличить пропускную способность операций вставки/удаления в 1.2 и 1.6 раз соответственно.

Многопоточность, потокобезопасные структуры данных, ослабленная семантика выполнения операций

Многоядерные вычислительные системы (ВС) с общей памятью являются основным средством решения сложных задач и обработки больших объемов данных. Они применяются как автономно, так и в составе распределенных ВС (кластерные, мультикластерные системы, системы с массовым параллелизмом). Такие системы включают в себя множество многоядерных процессоров, разделяющих единое адресное пространство, и имеют иерархическую структуру (рис. 1). Так, занимающий первое место в рейтинге TOP500 суперкомпьютер Summit (17 млн процессорных ядер) включает в себя 4608 вычислительных узлов, каждый из которых укомплектован двумя 24-ядерными универсальными процессорами семейства IBM Power9 и шестью графическими ускорителями NVIDIA Tesla V100 (640 ядер) [1]. Суперкомпьютер Sunway Taihulight (более 10 млн процессорных ядер, второе место в рейтинге TOP500) укомплектован 40 960 процессорами Sunway SW26010, включающими 260 вычислительных ядер [2].

Среди существующих ВС с общей памятью выделяют SMP-системы, обеспечивающие одинаковую скорость доступа процессоров к памяти, и NUMA-системы, представленные в виде множества узлов (композиция процессора и локальной памяти) и характеризующиеся различными скоростями доступа процессоров к локальным и удаленным сегментам памяти. Параллельные программы для ВС с общей памятью создаются в модели многопоточного программирования. Основной проблемой, возникающей при разработке программ, является организация доступа к разделяемым структурам данных. Требуется реализовать корректное выполнение операций параллельными потоками (отсутствие состояний гонки, взаимных блокировок и т. д.) и обеспечить масштабируемость для большого числа потоков и высокой интенсивности выполнения операций. Для этого необходимо разработать средства синхронизации потоков и построить алгоритмы выполнения операций для потокобезопасных струк-

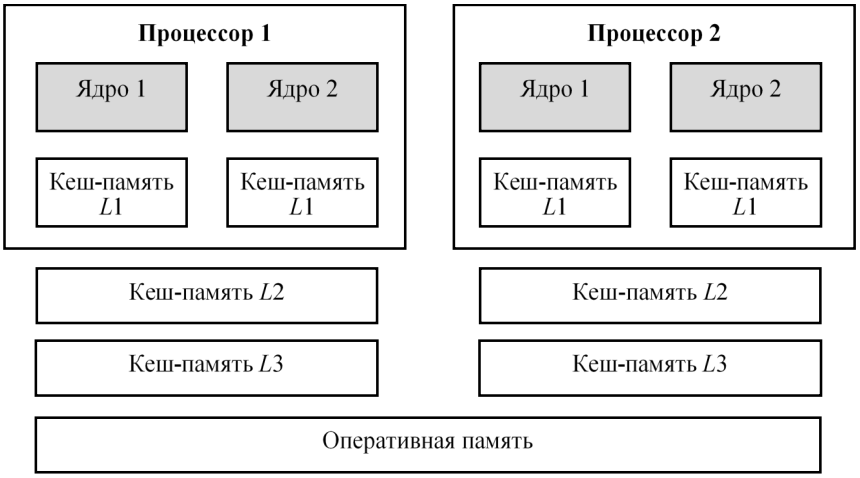


Рис. 1

тур данных. Основными методам синхронизации потоков являются методы блокировки, неблокируемые потокобезопасные структуры данных и транзакционная память.

Блокировки (locks, mutexes) реализуют обращение к общим участкам памяти единственным потоком в любой момент времени. Среди алгоритмов блокировки можно выделить TTAS, CLH, MCS, Lock Cohorting, Flat Combining, Remote Core Locking и др. [3]. Недостатками использования блокировок является возможность возникновения тупиковых ситуаций (deadlocks, livelocks), голодания потоков (starvation), инверсии приоритетов (priority inversion) и высокая конкуренция потоков на захват блокировки (lock convoy).

Неблокирующие структуры данных (non-blocking concurrent data structures) обладают свойством линеаризуемости (linearizability), предполагающим, что любое параллельное выполнение операций эквивалентно некоторому последовательному их выполнению, при этом завершение выполнения каждой операции не требует завершения выполнения какой-либо другой операции [4]. Среди неблокируемых структур данных выделяют классы wait-free (каждый поток завершает выполнение любой операции за конечное число шагов), lock-free (часть потоков завершают выполнение операций за конечное число шагов), obstruction-free (любой поток завершает выполнение операций за конечное число шагов, если выполнение других потоков приостановлено). Среди наиболее распространенных неблокируемых алгоритмов и структур данных можно выделить Treiber Stack, Michael & Scott Queue, Harris & Michael List, Elimination Backoff Stack, Combining Trees, Diffracting Trees, Counting Network, Cliff Click Hash Table, Split-ordered lists и др. [5]. К недостаткам неблокируемых структур данных можно отнести высокую трудоемкость их разработки (по сравнению с другими подходами), отсутствие аппаратной поддержки атомарных операций для больших или несмежных участков памяти (double compare-and-swap, double-width compare-and-swap), проблемы освобождения памяти (ABA problem).

В рамках транзакционной памяти (transactional memory) в программе организуются транзакционные секции, в которых реализуется защита разделяемых областей памяти. Транзакция (transaction) – это конечная последовательность операций транзакционного чтения/записи памяти [6], [7]. Операция транзакционного чтения выполняет копирование содержимого указанного участка общей памяти в соответствующий участок локальной памяти потока. Транзакционная запись копирует содержимое указанного участка локальной памяти в соответствующий участок общей памяти, доступной всем потокам. После завершения выполнения транзакция может быть либо зафиксирована (commit), либо отменена (cancel). Фиксация транзакции подразумевает, что все сделанные в ее рамках изменения памяти становятся необратимыми. Выделяют программную (LazySTM, TinySTM, GCC TM и др.) и аппаратную (Intel TSX, AMD ASF, Oracle Rock и др.) транзакционную память. Из недостатков транзакционной памяти можно выделить ограничения на отдельные виды операций внутри транзакционных секций, накладные расходы при отслеживании изменений в памяти, сложность отладки программы.

Учитывая вышерассмотренные недостатки существующих средств синхронизации, их производительность может быть не достаточна для современных многопоточных программ. В данной статье описывается метод повышения мас-

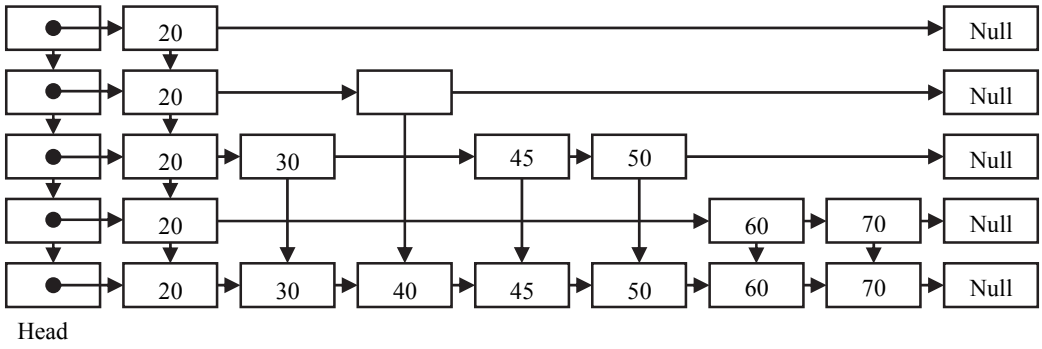


Рис. 2

штабируемости за счет ослабления (relaxation) [8] семантики структур данных.

Потокобезопасные структуры с ослабленной семантикой выполнения операций. В основе данного подхода лежит компромисс между масштабируемостью (производительностью) и корректностью семантики выполнения операций. Предлагается ослабить семантику выполнения операций для повышения возможности масштабирования. Например, при поиске максимального элемента в массиве поток может пропустить заблокированные другими потоками участки массива для повышения производительности операции поиска, при этом теряется точность выполнения данной операции.

К данному подходу применим принцип квазилинеаризации (quasi-linearizability) [4], который предполагает, что во время выполнения некоторых операций могут произойти несколько событий, одновременно изменяющих структуру данных таким образом, что после выполнения одной из операций состояние структуры данных не определено. Таким образом, результат операции может, но не должен совпадать с подразумеваемым.

В большинстве известных неблокируемых потокобезопасных структур и алгоритмов блокировки существует единая точка выполнения операций над структурой. Например, при вставке элемента в очередь необходимо использовать единственный указатель на первый элемент структуры. В случае многопоточной системы данный факт является узким местом, так как каждый поток вынужден блокировать один элемент, заставляя другие потоки ожидать. В ослабленных структурах данных единая структура заменяется на набор простых структур, композиция которых рассматривается как логически единая структура. Вследствие этого увеличивается количество возможных точек обращений к данной структуре, что позволяет избежать возникновения узких мест.

В рамках данного подхода каждая простая структура, как правило, защищается блокировкой. При выполнении операции поток обращается к случайной структуре из набора и пытается ее заблокировать. В случае успешной блокировки структуры поток завершает выполнение операции, в противном случае – случайным образом выбирает новую структуру. Таким образом, синхронизация потоков сводится к минимуму, но допустимы потери точности выполнения операций. Основными представителями ослабленных структур данных являются SprayList, *k*-LSM, Multiqueue.

В основе SprayList [9] (рис. 2) лежит структура «список с пропусками» (SkipList) [10]. SprayList является связным графом, где на нижнем уровне структуры находится связный отсортированный список всех элементов, а каждый следующий уровень с заданной фиксированной вероятностью содержит элементы списка нижнего уровня.

Поиск по данной структуре осуществляется линейно сверху вниз и слева направо, каждую итерацию выполняется переход по одному указателю. В отличие от списка с пропусками SprayList предполагает не линейный поиск сверху вниз и из начала в конец, а случайное перемещение сверху вниз и слева направо. Если поиск не дает результатов или элемент оказался заблокирован другим потоком, алгоритм возвращается на предыдущий элемент. После нахождения нужного элемента операции со списком выполняются так же, как и в списке с пропусками. Однако в худшем случае вставка элементов в SprayList вызывает значительные накладные расходы из-за необходимости поддерживать упорядоченный список.

В качестве базовой структуры *k*-LSM [11] используется журнально-структурированное дерево со слиянием (log-structured merge tree, LSM). Каждое изменение структуры записывается в отдельный лог-файл, узлы дерева являются отсортированными массивами (блоками), каждый из кото-

рых находится на уровне L дерева и может содержать N элементов ($2L - 1 < N \leq 2L$). Каждый поток имеет локальную распределенную LSM. Общая LSM является результатом слияния нескольких распределенных LSM-структур. Все потоки могут обращаться к общей LSM по единому указателю. В результате объединения общей и распределенных LSM-структур была получена *k*-LSM-структура (рис. 3). Выполняя операцию вставки элемента, поток сохраняет элемент в локальной LSM-структуре. При операции удаления используется поиск наименьшего ключа в локальной LSM. Если локальная структура пуста, а требуется операция, отличная от операции вставки, начинается попытка доступа к чужим LSM-структурам, поиск осуществляется среди всех распределенных и общей LSM-структур, и, если найденная структура не заблокирована, выполняется операция над ней. Данный подход называется work-stealing и описан в [12]. Недостатками данной структуры является синхронизация обращений к общей LSM и попытка обращения к чужой LSM, так как нет гарантии, что она не является пустой.

Multiqueues [13], [14] (рис. 4) представляет собой композицию простых очередей с приоритетом, защищенных блокировками. На каждый поток приходится две и более очередей с приоритетом. Операция вставки элемента осуществляется в случайную, не заблокированную другим потоком, очередь. Операция удаления элемента с минимальным ключом выполняется следующим образом: выбираются две случайные незаблокированные очереди, сравниваются значения их минимальных элементов и удаляется элемент с наименьшим значением из соответствующей очереди. Данный элемент не всегда является минимальным из вставленных в глобальную структуру, одна-

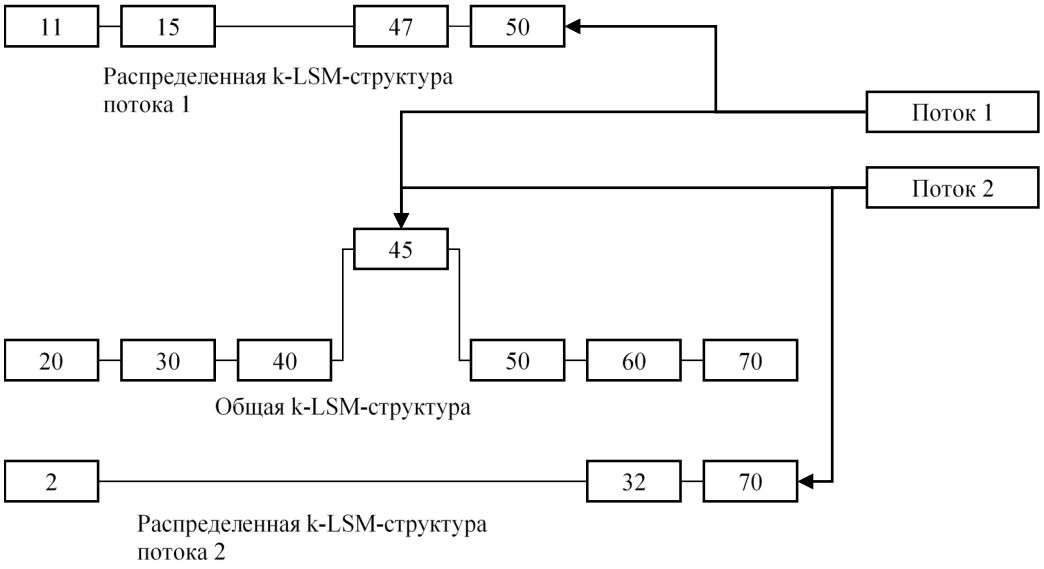


Рис. 3

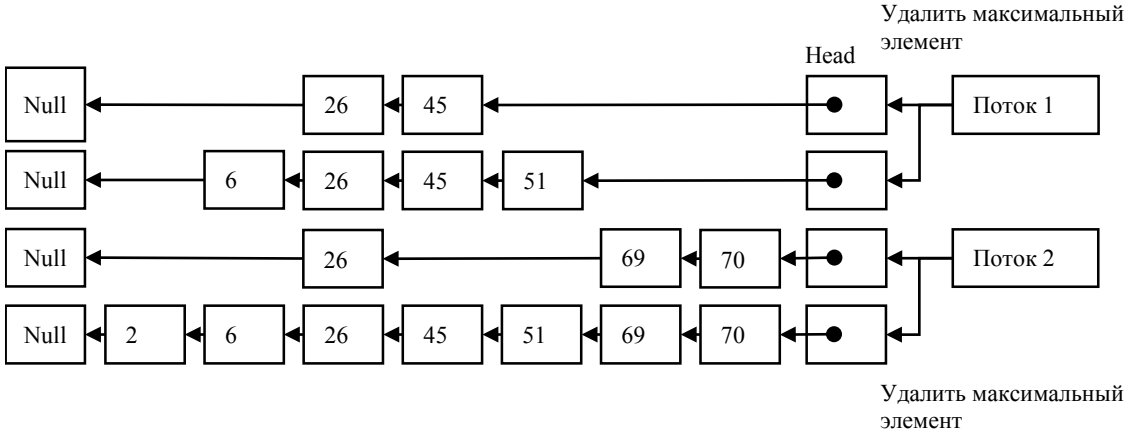


Рис. 4

ко он близок к минимальному и для реальных задач данной погрешностью можно пренебречь.

В данной статье предлагаются алгоритмы оптимизации выполнения операций для очередей с приоритетом на основе Multiqueues. Алгоритмы позволяют оптимизировать выбор структуры для выполнения операции.

Оптимизация выполнения операций. Недостатком текущей реализации операций вставки и удаления в Multiqueues является алгоритм поиска случайной очереди. Поток, выполняющий операцию, с большой вероятностью обращается к очередям, заблокированными другими потоками. Структура Multiqueues включает в себя *kp* очередей, где *k* – число очередей на один поток; *p* – количество потоков.

Построены эвристические алгоритмы выбора очереди для выполнения операции. Предложены методы уменьшения количества коллизий на основе ограничения диапазона для случайного выбора структуры. Под коллизиями подразумевается обращение потока к заблокированному другим потоком области памяти. Множество очередей и потоков делится пополам, каждый поток во время выбора очереди обращается только к половине всех очередей, за счет чего уменьшается вероятность выбора заблокированной очереди. Обозначим $i \in \{0, 1, \ldots, p\}$ идентификатор потока, тогда для первой половины потоков $i \in \{0, 1, \ldots, \lfloor p/2 \rfloor\}$ операция выбора случайной очереди выполняется среди очередей $q \in \{0, 1, \ldots, \lfloor kp/2 \rfloor\}$, где q – выбранная для выполнения операции очередь в структуре Multiqueues, а для второй половины потоков $i \in \{\lfloor p/2 \rfloor + 1, ..., p\}$ – во второй половине очередей $q \in \{\lfloor kp/2 \rfloor + 1, ..., kp\}$. Также разработан подход к оптимизации выбора очередей, основанный на «привязке» очередей к потокам. Данная схема позволяет задать порядок обращения потока к очередям. В реализации закрепления используется следующая модель: всего в множестве *kp* очередей, тогда каждый поток имеет $\{pi, \ldots, pi+k\}$ закрепленных очередей. При выполнении операции поток сначала обращается к очереди $q \in \{pi, \ldots, pi+k\}$, тем самым сведя обращения к заблокированным очередям к минимуму. Если все очереди из множества $\{pi, \ldots, pi+k\}$ заблокированы, используется оригинальная схема случайного выбора очереди среди всех очередей Multiqueues для выполнения данной операции.

Алгоритм 1 представляет оптимизированный алгоритм вставки (OptHalfInsert) элемента в структуру Multiqueues. Очередь выбирается в зависимости от того, какой половине потоков принадлежит текущий идентификатор потока.

Алгоритм 1: OptHalfInsert

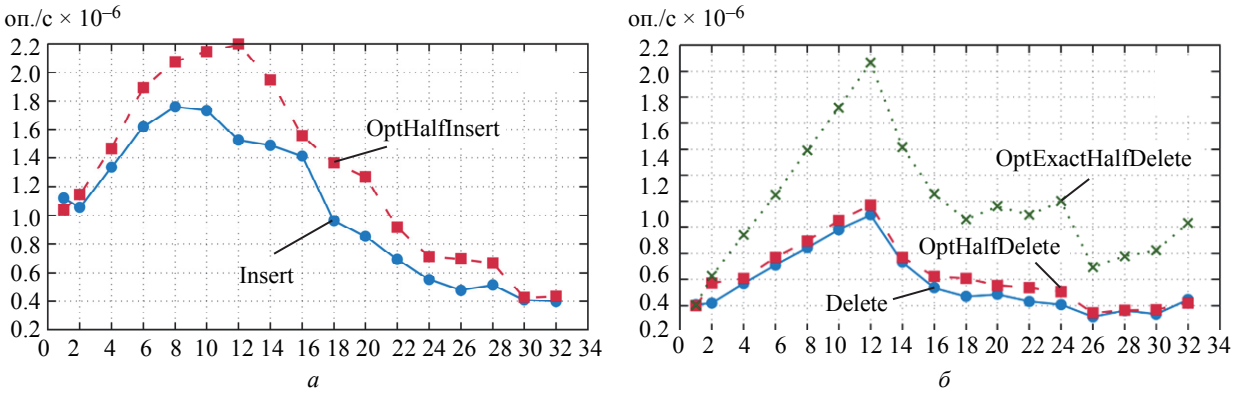
do
  if $i \in \{0, 1, \ldots, \lceil p/2 \rceil\}$ **then**
    $q = \text{RandQueue}(0, kp/2)$ //
    $q \in \{0, 1, \ldots, \lfloor kp/2 \rfloor\}$
  else
    $q = \text{RandQueue}(kp/2+1, kp)$ //
    $q \in \{\lfloor kp/2 \rfloor + 1, \ldots, kp\}$
  end
while isLocked(q);
Lock(q)
q.insert(Element)
Unlock(q)

Алгоритм 2 предоставляет псевдокод оптимизированного алгоритма удаления максимального элемента (OptHalfDelete), выбор очереди происходит так же, как и в алгоритме OptHalfInsert.

Алгоритм 3 содержит псевдокод альтернативного оптимизированного алгоритма удаления максимального элемента (OptExactDelete), сначала алгоритм выбирает очередь из «привязанных» к потоку, а только затем среди всех.

Алгоритм 2: OptHalfDelete
do
  if $i \in \{0, 1, \ldots, p/2\}$ **then**
    $[q1, q2] = \text{Rand2Queue}(0, kp/2)$ //
    $[q1, q2] \in \{0, 1, \ldots, \lfloor kp/2 \rfloor\}$
  else
    $[q1, q2] = \text{Rand2Queue}(kp/2+1, kp)$ //
    $[q1, q2] \in \{\lfloor kp/2 \rfloor + 1, \ldots, kp\}$
  end
  q = GetMaxElementQueue($q1$, $q2$)
  while isLocked(q);
  Lock(q)
  q.removeMax()
  Unlock(q)

Алгоритм 3: OptExactDelete
do



а *б*

Рис. 5

if *iteration* == 0 **then**
$[q1, q2] = \text{Rand2Queue}(pi, pi + k)$//
$[q1,\ q2] \in \{0, 1, \ldots, \lfloor kp/2 \rfloor\}$
else
$[q1, q2] = \text{Rand2Queue}(0, kp)$//
$[q1,\ q2] \in \{\lfloor kp/2 \rfloor + 1, \ldots, kp\}$
end
++*iteration*;
q = GetMaxElementQueue(*q*1, *q*2)
while isLocked(*q*);
Lock(*q*)
q.removeMax()
Unlock(*q*)

Алгоритм балансировки. В результате продолжительной работы алгоритмов может возникнуть дисбаланс ослабленной очереди с приоритетом: некоторые очереди могут содержать значительно больше элементов, чем другие. Данное обстоятельство приводит к снижению производительности алгоритмов, так как пустые очереди становятся не пригодными для операции удаления, что увеличивает время поиска подходящих очередей для выполнения операции. Создан алгоритм балансировки (*Алгоритм 4*) всей структуры Multiqueues для большего равномерного распределения элементов среди очередей.

Алгоритм 4: Balancing
*q*1 = FindLargestQueue()
*q*2 = FindShortestQueue()
if *q1*.size() > AvgSizeOfAllQueues()·0.2 **then**
Lock(*q*1)
Lock(*q*2)
sizeToTransfer = *q*1.size()*0.3
TransferElements(*q*1, *q*2, *sizeToTransfer*)
Unlock(*q*1)
Unlock (*q*2)
end

Результаты экспериментов. Экспериментальные исследования проводились на узле кластерной вычислительной системы Информационно-вычислительного центра Новосибирского государственного университета со следующими характеристиками: два Intel Xeon X5670 2.9 GHz, 16Gb RAM (6 физических ядер, 12 логических ядер).

В качестве показателя эффективности использовалась пропускная способность, которая рассчитывается как сумма пропускных способностей потоков $b_i = n/t$, где n – число операций вставки/удаления элементов потоком i; t – время выполнения операций.

В ходе первого эксперимента выполнялось сравнение эффективности оригинальной и оптимизированной ослабленной очереди с приоритетом. Исследовались отдельные операции вставки/удаления. На каждый поток было выделено $k = 2$ очередей. Каждый поток выполнял $n = 10^6$ операций вставки и $n = 0.5 \cdot 10^6$ операций удаления.

На рис. 5, *а* показана пропускная способность алгоритмов вставки, на рис. 5, *б* – пропускная способность алгоритмов удаления. На рис. 5 по оси y – количество операций в секунду (оп./с), по оси x – количество потоков p. Как видно из графиков, оптимизированные алгоритмы показывают существенный прирост производительности за счет использования оптимизированных алгоритмов, уменьшающих вероятность коллизий.

Следующий эксперимент показывает зависимости количества случайных операций (вставки, удаления) от количества используемых потоков. Анализировались следующие варианты использования алгоритмов вставки и удаления:

1. Исходный алгоритм вставки (Insert) и исходный алгоритм удаления элемента (Delete).

2. Оптимизированный алгоритм вставки (OptHalfInsert) и оптимизированный алгоритм удаления элементов (OptHalfDelete).

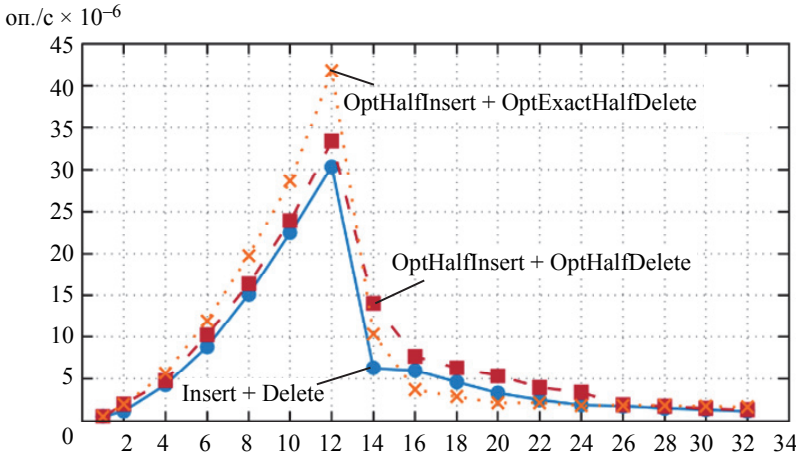
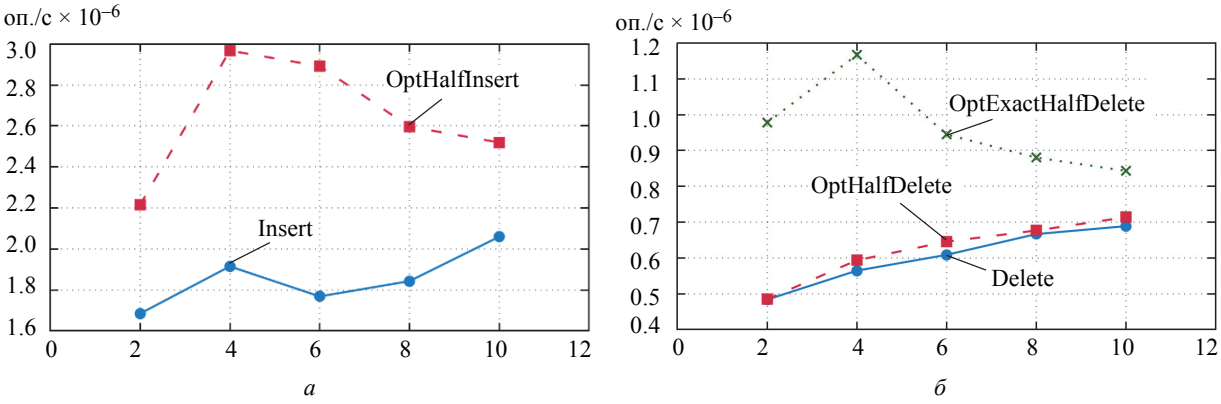


Рис. 6



а *б*

Рис. 7

3. Оптимизированный алгоритм вставки (OptHalfInsert) и альтернативный оптимизированный алгоритм удаления элементов (OptExactDelete).

График на рис. 6 (по оси y – количество операций в секунду, по оси x – количество потоков p) показывает зависимость пропускной способности случайных операций для исходных и оптимизированных алгоритмов. Из графика видно, что случайные операции OptHalfInsert и OptExactDelete на 12 потоках показывают пропускную способность на 30 % выше, чем случайные операции в исходных алгоритмах вставки и удаления.

Проведен анализ оптимального количества k очередей на один поток. На рис. 7, *а* показана пропускная способность алгоритмов вставки, на рис. 7, *б* – пропускная способность алгоритмов удаления. На рис. 7 по оси y – количество операций в секунду, по оси x – количество очередей на один поток k. Использовалось фиксированное количество потоков $p = 12$. Для алгоритмов OptHalfInsert и OptExactDelete максимальная пропускная способность системы достигается при количестве очередей $k = 4$ на один поток. Однако при использовании исходных алгоритмов вставки/удаления или OptHalfDelete выгоднее увеличивать параметр k, так как в этом случае уменьшается вероятность коллизии и, как следствие, уменьшается время поиска незаблокированного ресурса.

Разработана оптимизированная версия ослабленной потокобезопасной очереди с приоритетом на основе Multiqueues. Разработанные алгоритмы вставки и удаления показывают прирост производительности в 1.2 и 1.6 раза соответственно по сравнению с оригинальными алгоритмами вставки и удаления. Оптимизация достигается за счет уменьшения количества коллизий на основе ограничения диапазона выбора случайной структуры. Реализация данных алгоритмов находится в открытом доступе по адресу https://github.com/Komdosh/Multiqueues.

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 18-57-34001, при поддержке Совета по грантам Президента РФ для государственной поддержки молодых российских ученых (проект СП-4971.2018.5) и в рамках государственной работы «Инициативные научные проекты» базовой части государственного задания Министерства образования и науки Российской Федерации (ЗАДАНИЕ № 2.6553.2017/БЧ).

СПИСОК ЛИТЕРАТУРЫ

1. TOP500 supercomputers list. URL: https://www.top500.org/news/summit-up-and-running-at-oak-ridge-claims-first-exascale-application/ (дата обращения 25.06.2018).

2. Dongarra J. Report on the sunway taihulight system. URL: http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf (дата обращения 12.05.2018).

3. Herlihy M., Shavit N. The art of multiprocessor programming. Boston: Morgan Kaufmann, 2012. 536 p.

4. Afek Y., Korland G., Yanovsky E. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency // OPODIS. 2010. Vol. 6490. P. 3–10.

5. Cederman D. Lock-free concurrent data structures // Programming Multicore and Many-core Computing Systems. 2017. Vol. 86. P. 59.

6. Shavit N., Touitou D. Software transactional memory // Distributed Computing. 1997. Vol. 10, № 2. P. 99–116.

7. Кулагин И. И. Средства архитектурно-ориентированной оптимизации выполнения параллельных программ для вычислительных систем с многоуровневым параллелизмом: дис. … канд. техн. наук / СибГУТИ. Новосибирск, 2018. 155 с.

8. Henzinger T. A. Quantitative relaxation of concurrent data structures // ACM SIGPLAN Notices. 2013. Vol. 48, № 1. P. 317–328.

9. Alistarh D. The SprayList: A scalable relaxed priority queue // ACM SIGPLAN Notices. 2015. Vol. 50, № 8. P. 11–20.

10. Pugh W. Skip lists: a probabilistic alternative to balanced trees // Communications of the ACM. 1990. Vol. 33, № 6. P. 668–676.

11. Wimmer M. The lock-free k-LSM relaxed priority queue // ACM SIGPLAN Notices. 2015. Vol. 50, № 8. P. 277–278.

12. Blumofe R. D., Leiserson C. E. Scheduling multithreaded computations by work stealing // J. of the ACM (JACM). 1999. Vol. 46, № 5. P. 720–748.

13. Rihani H., Sanders P., Dementiev R. Multiqueues: Simpler, faster, and better relaxed concurrent priority queues. URL: https://arxiv.org/pdf/1411.1209.pdf (дата обращения 11.03.2018).

14. Sanders P. Randomized priority queues for fast parallel access // J. Parallel and Distributed Computing, Parallel and Distributed Data Structures. 1998. № 49 (1). P. 86–97.

A. V. Tabakov, A. A. Paznikov
Saint Petersburg Electrotechnical University «LETI»

ALGORITHMS FOR OPTIMIZATION OF RELAXED CONCURRENT PRIORITY QUEUES IN MULTICORE SYSTEMS

In designing the scalable concurrent data structures for shared-memory systems, one promising approach is the relaxation of operation execution order. Relaxed concurrent data structures are non-linearizable and don't provide strong operation semantics (such as FIFO/LIFO for linear lists, delete max (min) element for priority queues, etc.). In the paper, use is made of the approach based on the design of concurrent data structures in the form of multiple simple data structures distributed among the threads. For the operation execution (insert, delete), a thread randomly chooses a subset of these simple structures and performs actions on them. An optimized relaxed concurrent priority queues based on this approach is proposed in the paper. The algorithms for optimization of priority queues selection for insert/delete operations and the algorithm for balancing the elements in queues have been designed. The algorithms consider the hierarchical structure of multicore systems, provide the data access localization and reduce the range of possible options for random selection. The optimized insertion algorithms increase the throughput by 22 % as compared with the original algorithms. The developed deletion algorithms improve the performance from 16 % up to 2 times.

Multithreading, threadsafe concurrent data structures, relaxed concurrent execution