Тема работы:

Масштабируемые потокобезопасные структуры данных для вычислительных систем с общей памятью на основе метода ослабления (relaxation) требований к семантике выполнения операций

Состав коллектива:

1. Пазников Алексей Александрович, к.т.н., с.н.с. СПбГЭТУ «ЛЭТИ», руководитель
2. Табаков Андрей Викторович, магистрант. СПбГЭТУ «ЛЭТИ», исполнитель

Информация о гранте:

РФФИ, проект № 19-07-00784 «Разработка методов, алгоритмов и программного обеспечения масштабируемой синхронизации для многопроцессорных вычислительных систем», руководитель – Пазников А.А., 2019-2021

Научное содержание работы:

1. **Постановка задачи:**

Построить эвристические алгоритмы выбора очереди для выполнения операции. Предложить 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 счёт чего уменьшается вероятность выбора заблокированной очереди. Разработать подход оптимизации выбора очередей, основанный на «привязке» очередей к потокам. Данная схема позволяет задать порядок обращения потока к очередям. Построить алгоритм балансировки структуры Multiqueues, для большего равномерного распределения элементов среди очередей.

2. **Современное состояние проблемы:**

В основе подхода к ослаблению семантики выполнения операций 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) [1], который предполагает, что во время выполнения некоторых операций могут произойти несколько событий, одновременно изменяющие 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 существующих неблокируемых потокобезопасных структур и алгоритмах блокировки существует единая точка выполнения операций над структурой. Например, при вставке элемента в очередь, 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 её заблокировать. В случае успешной блокировки структуры, поток завершает выполнение операции; в противном случае, поток 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 [2] лежит структура список с пропусками (SkipList) [3]. 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. 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.

В качестве базовой структуры k-LSM [4] 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 которых находится на уровне L дерева и может содержать N элементов ($2L - 1 < N \leq 2L$). Каждый поток имеет локальную распределённую LSM. Общая LSM является результатом слияния нескольких распределённых LSM структур. Все потоки могут обращаться к общей LSM по единому указателю.

Multiqueues [5] 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, незаблокированную другим потоком, очередь. Операция удаления элемента с минимальным ключом выполняется следующим образом: выбираются две случайные незаблокированные очереди, сравниваются их значения минимальных элементов и выполняется удаление элемента с наименьшим значением из соответствующей очереди. Данный элемент не всегда является минимальным из вставленных в глобальную структуру, однако он близок к минимальному и для реальных задач данной погрешностью можно пренебречь.

1. Y. Afek, G. Korland, E. Yanovsky. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency // OPODIS. 2010.Vol. 6490.
2. Alistarh D. et al. The SprayList: A scalable relaxed priority queue // ACM SIGPLAN Notices. 2015. Vol. 50. no. 8. pp. 11-20.
3. Pugh W. Skip lists: a probabilistic alternative to balanced trees // Communications of the ACM. 1990. Vol. 33. no. 6. pp. 668-676.
4. Wimmer M. et al. The lock-free k-LSM relaxed priority queue // ACM SIGPLAN Notices. 2015. Vol. 50. no. 8. pp. 277-278.
5. Rihani H., Sanders P., Dementiev R. Multiqueues: Simpler, faster, and better relaxed concurrent priority queues // arXiv preprint arXiv:1411.1209. 2014.

3. Подробное описание работы, включая используемые алгоритмы:

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 счёт чего уменьшается вероятность выбора заблокированной очереди. Обозначим $i \in \{0, 1, ..., p\}$ 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 $i \in \{\lfloor p/2 \rfloor, + 1 \ldots, p\}$ – во второй половине очередей $q \in \{\lfloor kp / 2 \rfloor + 1, \ldots, kp]$.

Разработан подход 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 $\{pi, \ldots, pi + k\}$ закреплённых 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, тогда используется оригинальная схема выбора очереди среди всех очередей, 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.

```
do
    if i ∈ {0, 1, …, p / 2} then
        q = RandQueue(0, kp/2) // q ∈ {0, 1,…, ⌊kp / 2⌋}
    else
        q = RandQueue(kp / 2 + 1, kp) // q ∈ {⌊kp / 2⌋ + 1, …, kp]
    end
while isLocked(q);
Lock(q)
q.insert(Element)
Unlock(q)
```

Алгоритм 1: OptHalfInsert

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

```
do
    if i ∈ {0, 1, …, p / 2} then
        [q1, q2] = Rand2Queue(0, kp/2) // [q1, q2] ∈ {0, 1,…, ⌊kp / 2⌋}
    else
        [q1, q2] = Rand2Queue(kp / 2 + 1, kp) // [q1, q2] ∈ {⌊kp / 2⌋ + 1, …, kp]
    end
    q = GetMaxElementQueue(q1, q2)
while isLocked(q);
Lock(q)
q.removeMax()
Unlock(q)
```

Алгоритм 2: OptHalfDelete

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

```
do
    if iteration == 0 then
        [q1, q2] = Rand2Queue(pi, pi+k) // [q1, q2] ∈ {0, 1,…, ⌊kp / 2⌋}
    else
        [q1, q2] = Rand2Queue(0, kp) // [q1, q2] ∈ {⌊kp / 2⌋ + 1, …, kp]
    end
    ++iteration;
    q = GetMaxElementQueue(q1, q2)
while isLocked(q);
Lock(q)
q.removeMax()
Unlock(q)
```

Алгоритм 3: OptExactDelete

Кроме того, разработан алгоритм балансировки ослабленной очереди с приоритетом. В результате продолжительной работы алгоритмов может иметь место дисбаланс ослабленной очереди с приоритетом: некоторые очереди могут содержать значительно больше элементов, чем в другие. Данное обстоятельство приводит к снижению производительности алгоритмов, так как пустые очереди становятся непригодными для операции удаления, что увеличивает время поиска подходящих очередей для выполнения операции. Создан алгоритм балансировки (Алгоритм 4) всей структуры Multiqueues.

```
q1 = FindLargestQueue()
q2 = FindShortestQueue()
if q1.size() > AvgSizeOfAllQueues()*0.2 then
        Lock(q1)
        Lock(q2)
        sizeToTransfer = q1.size()*0.3
        TransferElements(q1, q2, sizeToTransfer)
        Unlock(q1)
        Unlock(q2)
end
```

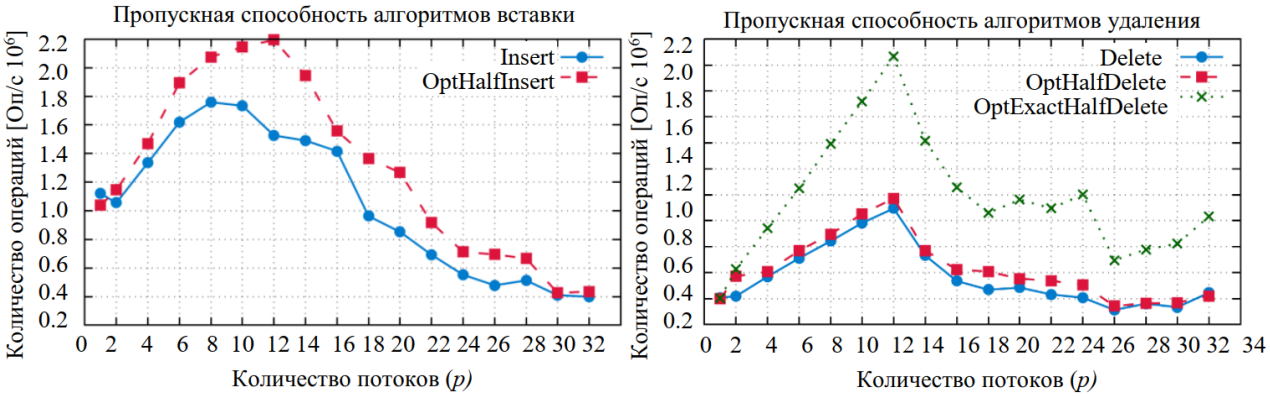
Алгоритм 4: Balancing

4. Полученные результаты:

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 $n = 0{,}5 \times 10^6$ операций удаления. 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)
3. Оптимизированный алгоритм вставки (OptHalfInsert) и альтернативный оптимизированный алгоритм удаления элементов (OptExactDelete)

5. Иллюстрация, визуализация результатов:



а *б*

Рис. 1 Пропускная способность оптимизированных алгоритмов вставки и удаления элементов.

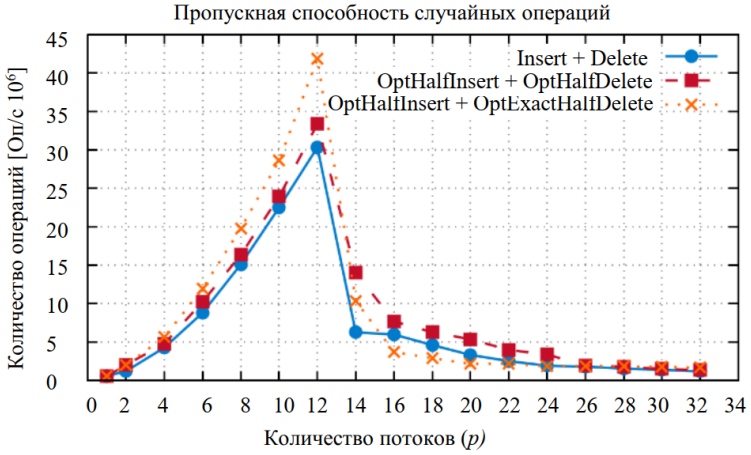
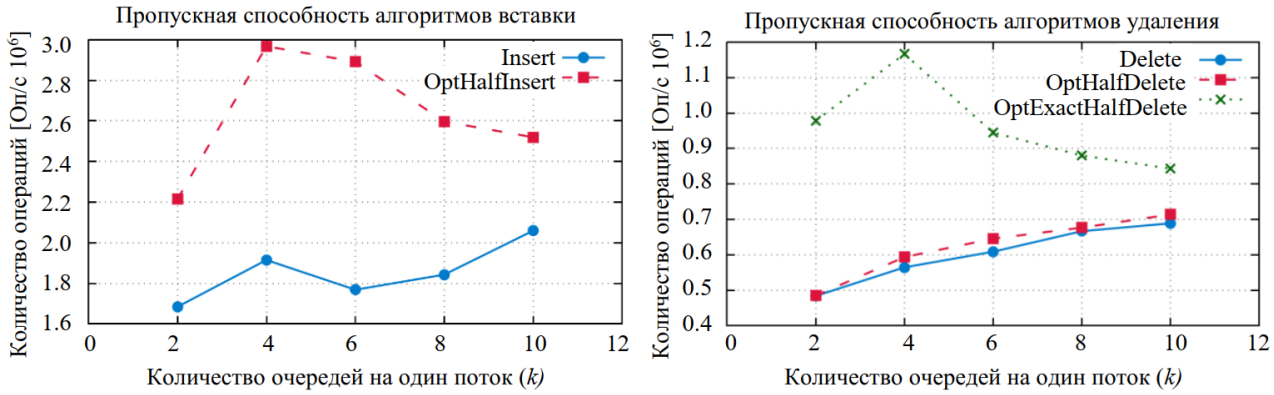


Рис. 2 Зависимость пропускной способности случайных операций, для исходных и оптимизированных алгоритмов.



а *б*

Рис. 3 Анализ оптимального количества k очередей на один поток. Использовалось фиксированное количество потоков p=12.

6. Эффект от использования кластера в достижении целей работы:

Экспериментальные исследования проводились на SMP узле кластерной вычислительной системы. Для эксперимента использовалось 16 гигабайт оперативной памяти и 2 процессора по 6 ядер без Hyper-threading. Кластер позволил провести исследование для 12 параллельных независимых потоков, выполняя операции вставки и удаления в единый момент времени.

Перечень публикаций, содержащих результаты работы:

1. A. V. Tabakov and A. A. Paznikov, "Using relaxed concurrent data structures for contention minimization in multithreaded MPI programs" 2019 J. Phys.: Conf. Ser. 1399 033037
2. A. V. Tabakov and A. A. Paznikov, "Modelling of Parallel Threads Synchronization in Hybrid MPI + Threads Programs" 2019 XXII International Conference on Soft Computing and Measurements (SCM)), St. Petersburg, Russia, 2019, pp. 197-199. doi: 10.1109/SCM.2019.8903806
3. Табаков А. В., Пазников А. А. Моделирование синхронизации параллельных потоков при выполнения гибридных MPI+threads программ // Материалы XXI IEEE международной конференции по мягким вычислениям (SCM), 2019. – С. 293-295
4. Tabakov A., Paznikov A. Algorithms for Optimization of Relaxed Concurrent Priority Queues in Multicore Systems // Proc. of the 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), 2019. – pp. 360-365. 10.1109/EIConRus.2019.8657105
5. Табаков А. В., Пазников А. А. Алгоритмы оптимизации потокобезопасных очередей с приоритетом на основе ослабленной семантики выполнения операций // Известия СПбГЭТУ «ЛЭТИ». – 2018. – № 10. – С. 42-49