ВЕСТНИК ТОМСКОГО ГОСУДАРСТВЕННОГО УНИВЕРСИТЕТА

2020 Управление, вычислительная техника и информатика № 50

УДК 004.272
DOI: 10.17223/19988605/50/12

А.А. Пазников

РАСПРЕДЕЛЕННАЯ ОЧЕРЕДЬ С ОСЛАБЛЕННОЙ СЕМАНТИКОЙ ВЫПОЛНЕНИЯ ОПЕРАЦИЙ В МОДЕЛИ УДАЛЕННОГО ДОСТУПА К ПАМЯТИ

Публикация выполнена при финансовой поддержке РФФИ и СИТМА в рамках научных проектов № 19-07-00784, № 18-57-34001 и Совета по грантам Президента РФ (проект СП-4971.2018.5).

Модель удаленного доступа к памяти (RMA) является перспективным средством повышения эффективности и упрощения разработки параллельных программ для распределенных вычислительных систем. Модель реализована в стандарте MPI (Message Passing Interface) и применяется в языках семейства PGAS (Partitioned Global Address Space). Предлагается оригинальный подход для решения актуальной задачи разработки в модели RMA масштабируемых распределенных структур данных. Основная идея заключается в ослаблении (relaxation) семантики выполнения операций. Исследуется эффективность созданной ослабленной распределенной очереди; экспериментально показано, что подход обеспечивает большую эффективность по сравнению со структурами строгой семантики.

Ключевые слова: распределенная очередь; ослабленные структуры данных; масштабируемость; удаленный доступ к памяти; MPI; RMA.

При разработке параллельных программ для вычислительных систем (ВС) одной из ключевых является задача синхронизации процессов (потоков), обращающихся к разделяемым (concurrent, thread-safe) структурам данных. Разделяемые структуры данных являются базовым элементом в параллельном программировании, поэтому эффективность синхронизации существенно влияет на время выполнения программ. Такие структуры должны обеспечивать доступ параллельных процессов (потоков) в произвольные моменты времени [1–3].

Синхронизация в ВС реализуется средствами блокировок (locks) и неблокируемых (non-blocking) структур данных. Блокировки обладают интуитивной семантикой и часто не менее эффективны по сравнению с неблокируемыми методами. В то же время программирование без блокировок позволяет избежать тупиковых ситуаций (deadlocks), инверсий приоритетов (priority inversion) и обеспечивает гарантии выполнения. Независимо от реализации большая часть структур данных характеризуется наличием узких мест (bottlenecks) для операций, таких как вставка и удаление элементов (очереди, стеки), удаление максимального элемента (очереди с приоритетом).

Большая часть работ в области разделяемых структур данных ориентированы на ВС с общей памятью, производительность которых может быть недостаточной для решения современных задач. Например, размеры графов социальных сетей достигают нескольких петабайт, а число вершин в графах из теста Graph500 – нескольких триллионов. Проекты в области физики высоких энергий, такие как CMS и Atlas, производят десятки петабайт ежегодно. Планируется, что Большой обзорный телескоп будет каждую ночь генерировать около 20 терабайт. Поскольку ВС с общей памятью имеют технологический предел числа процессорных ядер, для решения таких задач требуется использовать ВС с распределенной памятью (кластерные ВС, ВС с массовым параллелизмом). В процессе программирования таких систем обрабатываемые данные представляются в виде

распределенных структур данных, для которых необходимо обеспечить масштабируемую синхронизацию.

Одной из перспективных моделей параллельного программирования для ВС с распределенной памятью является модель удаленного доступа к памяти (Remote Memory Access, RMA), реализованная в стандарте MPI [4, 5]. В рамках RMA процессы непосредственно обращаются к памяти других процессов вместо отправки и получения сообщений. В отличие от модели разделенного глобального адресного пространства (Partitioned Global Address Space, PGAS), представленной языками UPC, CAF, Chapel, X10, модель RMA тесно интегрирована с библиотеками MPI и может быть использована наравне с моделью передачи сообщений. Программы в модели RMA характеризуются меньшим временем выполнения по сравнению с моделью передачи сообщений и PGAS. Большая часть современных коммуникационных сетей (Infiniband, PERCS, Gemini, Aries, RoCE over Ethernet) обеспечивает поддержку RMA с помощью технологии RDMA [4], реализующей обращение к удаленным сегментам памяти без участия центрального процессора.

Опишем программную модель RMA в MPI. Основными являются неблокируемые функции MPI_Put (запись в память удаленного процесса) и MPI_Get (чтение из удаленной памяти), атомарные MPI_Accumulate, MPI_Get_accumulate, MPI_Compare_and_swap. RMA-вызовы должны находиться внутри областей (эпохи, epochs), в рамках которых выполняется синхронизация. В работе применяется пассивный метод синхронизации (passive target synchronization), реализованный в стандарте MPI [5]. При пассивной синхронизации процесс открывает эпоху реализации удаленного доступа (access epoch) посредством вызова функций MPI_Win_lock/MPI_Win_lockall, после чего он может выполнять RMA-операции для доступа к зарегистрированным сегментам памяти (окна, windows) других процессов. Таким образом, RMA-операции выполняются в одностороннем порядке, без явного вызова функций синхронизации другими процессами.

Основная часть работ в области разделяемых структур данных направлена на создание средств синхронизации для ВС с общей памятью. К ним относятся алгоритмы блокировки потоков [1, 6] (TTS, Backoff, CLH, MCS, Oyama, Flat Combining, RCL и др.). Хотя некоторые методы (Hierarchical Backoff (CLH, MCS), Cohorting и др.) учитывают отдельные иерархические уровни, они неприменимы в ВС с распределенной памятью. Неблокируемые структуры [1–3, 7] также разработаны для многоядерных ВС и неприменимы в распределенных ВС. Перспективным методом повышения масштабируемости разделяемых структур данных является ослабление их семантики (relaxation) [8–14]. Например, в ослабленной очереди с приоритетом извлекается не максимальный элемент, а элемент, близкий к максимальному. В ослабленной очереди (стеке) удаляется не первый (последний) добавленный элемент, а элемент в его окрестности. Ослабленные структуры обеспечивают высокую пропускную способность и приемлемый уровень упорядоченности операций в реальных программах. В работах [8, 9] для построения потокобезопасной очереди с приоритетом предлагается использовать набор последовательных очередей. Аналогичная реализация стека, основанного на временных метках, предлагается в [10]. Также построены аналитические модели ослабления [13, 14], включая квазилинеаризуемость (quasi linearizability), количественное ослабление (quantitative relaxation).

Насколько известно, для распределенных ВС не разработаны эффективные масштабируемые разделяемые структуры данных. Методы, предложенные для распределенных ВС, включают простые спинлоки, блокировки чтения-записи и MCS-блокировки [15]. Работы, посвященные распределенным структурам данных [16–18], неприменимы в модели RMA. В языках PGAS реализованы отдельные примитивы синхронизации и распределенные структуры, но они характеризуются наличием узких мест и высокими накладными расходам. Исходя из вышесказанного, задача разработки эффективных разделяемых структур данных для распределенных ВС является востребованной и нерешенной в настоящее время. В данной статье предлагается метод построения масштабируемых распределенных структур на основе ослабления их семантики, рассмотренный на примере очереди.

1. Распределенная ослабленная очередь

1.1. Ослабление семантики выполнения операций для распределенных очередей

Очередь – коллекция объектов, реализующая дисциплину FIFO («первым вошел – первым вышел»). Основные операции: добавление (insert, enqueue) элемента в последнюю позицию (хвост, tail) и извлечение (remove, dequeue) элемента из первой позиции (голова, head). В классических реализациях распределенных очередей необходимо обеспечивать актуальность данных о расположении (ранг процесса) головного и хвостового элементов. Процесс перед выполнением операций при необходимости обновляет данные о расположении головного (хвостового) элемента. Это приводит к дополнительным накладным расходам и увеличивает время выполнения операций. Другим значимым недостатком является наличие узких мест при одновременном обращении нескольких процессов к процессу, в памяти которого находится головной (хвостовой) элемент.

С целью увеличения масштабируемости распределенной очереди предлагается ослабить ее семантику и допустить извлечение элемента из окрестности первого добавленного элемента. Для этого распределенная структура представляется в виде множества последовательных структур, распределенных между процессами. Каждый процесс асинхронно обращается к удаленным сегментам посредством RMA-вызовов (рис. 1). Данный подход не предполагает выполнения операций для актуализации данных о расположении головы и хвоста очереди и позволяет избежать возникновения узких мест. Кроме того, за счет низкой латентности односторонних коммуникаций и аппаратной поддержки RDMA метод гарантирует снижение времени выполнения операций.

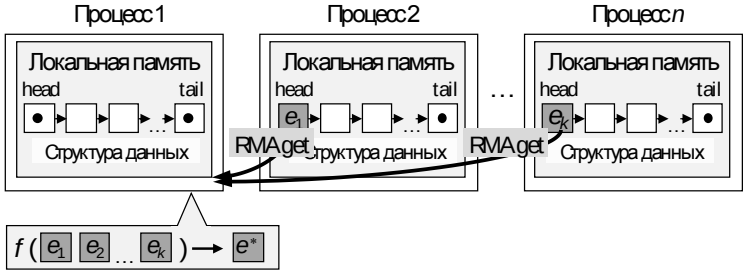

Рис. 1. Операция извлечения элемента в ослабленной распределенной очереди
Fig. 1. Execution of item remove operation for relaxed distributed queue

Опишем операции добавления и удаления элементов. Обозначим p – число процессов. Считаем, что часы процессов синхронизированы и каждому элементу e очереди соответствует временная метка $t(e)$ – момент добавления его в очередь. При выполнении операции insert процесс случайным образом выбирает очередь $s \in \{1, 2, \ldots, p\}$ и помещает в нее элемент. Отметим, что вместо случайного выбора можно использовать другие схемы для локализации обращений к памяти и других оптимизаций.

При выполнении операции удаления remove процесс выбирает k очередей-кандидатов $R \subseteq \{1, 2, \ldots, p\}$ и посредством RMA-операций получает значения элементов $\{e_1, e_2, \ldots, e_k\}$, находящихся в голове соответствующих очередей. Далее среди элементов-кандидатов определяется «лучший» элемент с минимальной временной меткой: $e^* = \mathrm{argmin}_{i \in \{1, 2, \ldots, k\}} t(e_i)$ (рис. 1).

Рассмотрим детально реализацию распределенной ослабленной очереди. При инициализации структуры данных на каждом процессе организуется циклический буфер фиксированного достаточно большого размера (100 000 в данной реализации) и синхронизируются часы процессов [19].

1.2. Операция добавления элементов

Входными данным операции insert добавления элемента являются значение элемента *val*, число процессов *p*, окно для выполнения RMA-операций *win* и массив блокировок *locks* для защиты данных в очередях на каждом процессе. Блокировки могут быть реализованы с помощью любого спинлока, в данной работе используется простой алгоритм TASLock [1]. Под MPI_Put_Atomic и MPI_Get_atomic

здесь и далее понимаются атомарные операции MPI_Put и MPI_Get, реализованные на основе MPI_Accumulate и MPI_Get_accumulate.

Таблица 1

Алгоритмы выполнения операций для распределенной ослабленной очереди: *а* – операция insert добавления элемента в очередь, *б* – операция remove извлечения элемента из очереди

a

Входные данные:	*val* – добавляемый элемент *locks* – массив блокировок для защиты очередей *p* – число процессов коммуникатора *win* – окно для выполнения RMA-операций
1	*nqueues* = *p*
2	**do**
3	*rank* = GETRAND(*p*)
4	*elem.val* = *val*
5	*elem.ts* = GETTIMESTAMP()
6	MPI_WIN_LOCK(*rank*, *win*)
7	LOCk(*rank*, *locks*[*rank*], *win*)
8	MPI_GET_ATOMIC(*rank*, *state*, *win*)
9	MPI_WIN_FLUSH(*win*)
10	**if** ISFULL(*state*) **then**
11	*nqueues* = *nqueues* – 1
12	**if** *nqueues* = 0 **then**
13	UNLOCK(*rank*, *locks*[*rank*], *win*)
14	MPI_WIN_UNLOCK(*rank*, *win*)
15	**return** ErrQueueFull
16	**end if**
17	**else**
18	MPI_PUT(*rank*, *elem*, *win*)
19	*state.tail* = (*state.tail* + 1) mod *size*
20	MPI_PUT_ATOMIC(*rank*, *state.tail*, *win*)
21	**end if**
22	UNLOCK(*rank*, *locks*, *win*)
23	MPI_WIN_UNLOCK(*rank*, *win*)
24	**while** ISFULL(*state*)

b

Входные данные:	*ncand* – число очередей-кандидатов *locks* – массив блокировок для защиты очередей *p* – число процессов коммуникатора *win* – окно для выполнения RMA-операций
1	MPI_WIN_LOCK_ALL(*win*)
2	*ncurr* = 0
3	*navail* = p
4	*nattempts* = 0
5	**while** *ncurr* < *ncand* **do**
6	*rank* = GETRAND(*p*)
7	*rc* = TRYLOCK(*rank*, *locks*[*rank*], *win*)
8	**if** LOCKISACQUIRED(*rc*) **then**
9	MPI_GET_ATOMIC(*rank*, *states*[*ncurr*], *win*)
10	MPI_WIN_FLUSH(*win*)
11	**if** ISEMPTY(*state*) **then**
12	UNLOCK(*rank*, *locks*[*rank*], *win*)
13	*navail* = *navail* – 1
14	**if** *navail* < *ncand* **then**
15	**if** *ncand* = 0 **then**
16	MPI_WIN_UNLOCK_ALL(*win*)
17	**return** ErrQueueEmpty
18	**end if**
19	*ncand* = *navail*
20	**end if**
21	**else**
22	MPI_GET(*rank*, *elems*[*ncurr*], *win*)
23	MPI_WIN_FLUSH(*win*)
24	ADDCAND(*rank*, *cands*)
25	*ncurr* = *ncurr* + 1
26	*nattempts* = 0
27	**end if**
28	**else if** LOCKISBUSY(rc) **then**
29	**if** *ncurr* > 0 **then**
30	*nattempts* = *nattempts* + 1
31	**if** *nattempts* = *max_nattempts* **then**
32	**for** *i* = 0 **to** *ncurr* **do**
33	UNLOCK(*cands*[*i*], *locks*[*cands*[*i*]], *win*)
34	**end for**
35	*ncurr* = 0
36	**end if**
37	**end if**
38	**end if**
39	**end while**
40	*bestrank* = GETBESTRANK(*cands*, *elems*)
41	*states*[*bestrank*].*tail* = (*states*[*bestrank*].*tail* + 1) mod
42	*size*
43	MPI_PUT_ATOMIC(*bestrank*, *states*[*bestrank*].*tail*, *win*)
44	**for** *i* = 0 **to** *ncand* **do**
45	UNLOCK(cands[i], *locks*[*cands*[*i*]], *win*)
46	**end for**
47	MPI_WIN_UNLOCK_ALL(win)
	return *elems*[*bestrank*].*val*

Основные шаги алгоритма (табл. 1, *a*):

1. Проинициализировать число доступных очередей *nqueues* (строка 1).

2. Случайным образом выбрать процесс *rank* (строка 3). Установить поля элемента (строки 4, 5). Начать эпоху синхронизации для выбранного процесса (строка 6). Заблокировать очередь процесса *rank* (строка 7) и получить ее состояние (строка 8).

3. Если очередь заполнена (строка 10), уменьшить *nqueues*. Если нет доступных очередей (строка 12), разблокировать очередь, завершить эпоху синхронизации, вернуть код ошибки (строки 13–15).

4. Если очередь не полна, добавить в нее элемент (строка 18), увеличить указатель *state.tail* на хвост очереди (строка 19) и установить новое значение состояния очереди (строка 20).

5. Разблокировать очередь (строка 22) и завершить эпоху пассивной синхронизации (строка 23).

6. Выполнять шаги 2–5, пока как минимум одна очередь не будет найдена (строка 24).

1.3. Операция удаления элементов

Входными данными для функции remove удаления элемента являются число кандидатов *ncand* для выбора элемента, количество процессов *p*, окно для выполнения RMA-операций *win* и массив блокировок *locks* для защиты данных очередей. Операция включает следующие шаги (табл. 1, *b*):

1. Начать эпоху пассивной синхронизации для всех процессов (строка 1), проинициализировать текущее число найденных очередей *ncurr* (строка 2), число доступных очередей *navail* (строка 3) и число попыток блокировки очереди *nattempts* (строка 4).

2. Если текущее число найденных кандидатов *ncurr* равно *ncand*, перейти на шаг 7. Если нет, случайно выбрать очередь *rank* (строка 6). Попытаться заблокировать очередь (строка 7).

3. Если очередь заблокирована, получить ее состояние *state* (строка 9). Если нет, переход на шаг 6.

4. Если очередь пуста (строка 11), разблокировать ее (строка 12), уменьшить *navail* (строка 13). Если *navail* < *ncand*, сбросить *ncand* до значения *navail*. Если не осталось доступных кандидатов (строка 15), завершить эпоху синхронизации (строка 16) и вернуть код ошибки (строка 17).

5. Если очередь не пуста, получить элемент в голове (строка 22), добавить в список кандидатов (строка 24), увеличить *ncurr* (строка 25), сбросить *nattempts* (строка 26). Перейти на шаг 2.

6. Если очередь не захвачена, увеличить *nattempts* (если это не первая очередь-кандидат) (строка 30). Если *nattempts* достигло максимального (строка 31), разблокировать захваченные очереди (строки 32–34), сбросить *ncurr* (строка 35), перейти на шаг 2. Данный шаг необходим для избежания взаимной блокировки, когда два процесса пытаются заблокировать уже захваченные очереди.

7. Выбрать кандидата с минимальным значением временной метки (строка 40). Для данной очереди инкрементировать указатель на голову и обновить состояние очереди (строка 41, 42). Разблокировать все очереди-кандидаты (строки 43, 45) и завершить эпоху синхронизации (строка 46).

2. Проведение экспериментов

Экспериментальное исследование ослабленной очереди проводилось на вычислительном кластере Jet Центра параллельных вычислительных технологий Сибирского государственного университета телекоммуникаций и информатики. Кластер укомплектован 18 вычислительными узлами, оборудованными двумя 4-ядерными процессорами Intel Xeon E5420 (суммарное число ядер 144). В качестве MPI-библиотеки применялась MPICH 3.2.1.

Разработан синтетический тест, выполняющий $n = 100\,000$ операций вставки / удаления (тип операции выбирается случайно). Число процессов p варьировалось от 16 до 144. Созданная распределенная ослабленная очередь Relaxed Queue сравнивалась со связным списком, реализованным в библиотеке MPICH (MPICH linked list) в модели RMA. Использовалось два типа списка: на основе эксклюзивной и разделяемой пассивной синхронизации (MPI_LOCK_EXCLUSIVE и MPI_LOCK_SHARED). Тип синхронизации определяет, допускается ли одновременное обращение нескольких процессов

к памяти удаленного процесса. Также исследовалось влияние числа очередей-кандидатов *ncand* на эффективность очереди. Для этого *ncand* варьировалось от 1 до 4. Кроме того, анализировалась зависимость эффективности очереди от типа пассивной синхронизации в функции вставки. Измерялась пропускная способность $b = t / n$, где t – время проведения эксперимента.

Пропускная способность разработанной очереди значительно превосходит пропускную способность линейного списка строгой семантики (рис. 2, *a*). Оптимизация достигается за счет сокращения накладных расходов, возникающих при выполнении доступа к элементам. Недостатки классических распределенных списков – необходимость актуализации данных о расположении головного (хвостового) элементов и возможность образования узких мест при одновременном обращении нескольких процессов к ним. Разработанная ослабленная очередь не требует поддержания согласованного состояния головы (хвоста) очереди, поскольку каждый раз процесс-кандидат и соответствующая очередь выбираются случайно. Такой подход также позволяет распределить нагрузку между процессами и избежать возникновения узких мест. Линейный список строгой семантики на основе разделяемой синхронизации более эффективен, по сравнению с эксклюзивным режимом (рис. 2, *b*), поскольку во время вставки / удаления несколько процессов одновременно обращаются к одному процессу, в памяти которого находится головной (хвостовой) элемент.

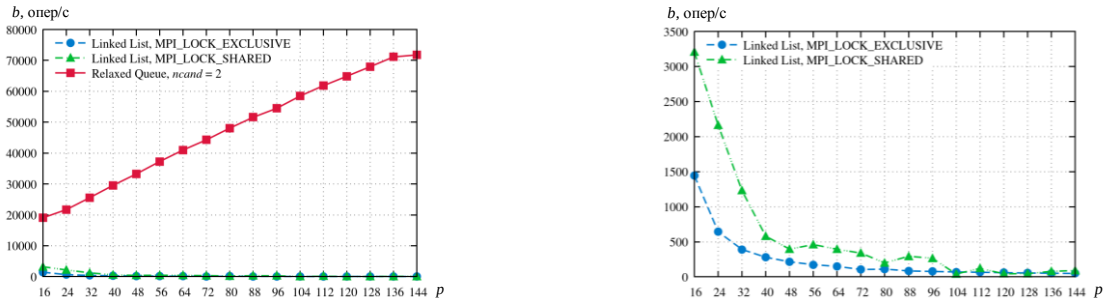

Рис. 2. Сравнение эффективности структур данных: *a* – пропускная способность;
b – пропускная способность линейного списка строгой семантики для разных режимов синхронизации
Fig. 2. Comparison of efficiency of data structures: *a* – throughput;
b – throughput of list with strong semantics for different synchronization modes

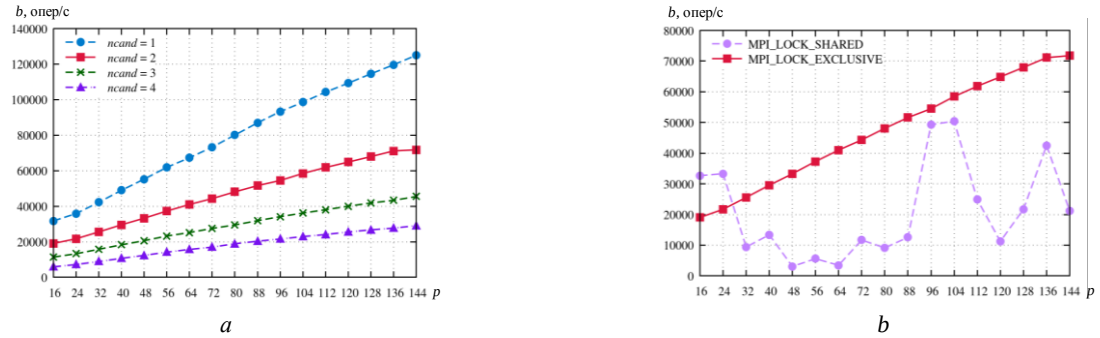

Рис. 3. Анализ эффективности Relaxed Queue: *a* – пропускная способность в зависимости от числа очередей-кандидатов;
б – пропускная способность ослабленной очереди для разных режимов синхронизации
Fig. 3. Analysis of Relaxed Queue efficiency: *a* – throughput depending on number of candidates;
b – throughput of Relaxed Queue for different synchronization modes

Как и ожидалось, пропускная способность уменьшается с увеличением числа кандидатов *ncand* (рис. 3, *a*). Это объясняется дополнительными накладными расходами на выполнение блокировки очередей, получение состояний и значений элементов очередей-кандидатов. На наш взгляд, *ncand* = 2 является достаточным для большинства случаев и обеспечивает приемлемый для практики уровень упорядоченности операций. Данные выводы согласуются с результатами аналогичной структуры для ВС с общей памятью [8]. Тем не менее для повышения близости порядка вставки / удаления элемен-

тов к порядку FIFO можно увеличить *ncand* до 3 и 4. В данной работе не выполняется оценка близости к FIFO, но это планируется сделать в будущем.

В отличие от распределенных списков со строгой семантикой, для ослабленной очереди эксклюзивный режим пассивной синхронизации обеспечивает большую пропускную способность по сравнению с разделяемым режимом (рис. 3, *b*). Это объясняется тем, что организация разделяемого режима является дорогостоящей операцией. Вместе с тем в ослабленной очереди одновременное обращение к одному процессу (последовательной очереди) является редким событием. Поэтому мы полагаем, что разделяемый режим является избыточным.

Заключение

В данной статье разработаны эвристические алгоритмы реализации ослабленных распределенных очередей в модели RMA. Созданная очередь основана на множестве последовательных очередей, распределенных между процессами. Очередь характеризуется значительно большей пропускной способностью по сравнению с линейными списками строгой семантики в модели RMA. Оптимизация достигается за счет устранения узких мест при выполнении операций. При реализации очереди рекомендуется использовать 2 или 3 очереди-кандидата и эксклюзивный тип пассивной синхронизации (MPI_LOCK_EXCLUSIVE).

ЛИТЕРАТУРА

1. Herlihy M., Shavit N. The art of multiprocessor programming. Morgan Kaufmann, 2012. 537 p.
2. Mark M., Shavit N. Concurrent Data Structures. Chapman and Hall / CRC Press, 2004. 32 p.
3. Shavit N. Data structures in the multicore age // Communications of the ACM. 2011. V. 54. P. 76–84.
4. 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.
5. 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, No. 2. P. 9.
6. 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.
7. 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.
8. Rihani H., Sanders P., Dementiev R. Brief announcement: Multiqueues: Simple relaxed concurrent priority queues // Proc. of the 27th ACM symposium on Parallelism in Algorithms and Architectures. 2015. P. 80–82.
9. 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.
10. Dodds M., Haas A., Kirsch C.M. A scalable, correct time-stamped stack // ACM SIGPLAN Notices. 2015. V. 50, No. 1. P. 233–246.
11. Alistarh D., Kopinsky J., Li J., Shavit N. The Spray List: a scalable relaxed priority queue // ACM SIGPLAN Notices. 2015. V. 50, No. 8. P. 11–20.
12. Wimmer M. et al. The lock-free k-LSM relaxed priority queue // ACM SIGPLAN Notices. 2015. V. 50, No. 8. P. 277–278.
13. Henzinger T.A., Kirsch C.M., Payer H., Sezgin A., Sokolova A. Quantitative relaxation of concurrent data structures // ACM SIGPLAN Notices. 2013. V. 48, No. 1. P. 317–328.
14. Afek Y., Korland G., Yanovsky E. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency // Int. Conf. on Principles of Distributed Systems. 2010. P. 395–410.
15. Schmid P., Besta M., Hoefler T. High-Performance Distributed RMA Locks // Proc. of the 25th ACM Int. Symposium on High-Performance Parallel and Distributed Computing, HPDC 2016, Kyoto, Japan, May 31 – June 04, 2016. ACM 2016. P. 19–30.
16. Mans B. Portable distributed priority queues with MPI // Concurrency – Practice and Experience. 1998. V. 10, No. 3. P. 175–198.
17. Brodal G.S., Traff J.L., Zaroliagis C.D. A parallel priority queue with constant time operations // J. of Parallel and Distributed Computing. 1998. Vol. 49, No. 1. P. 4–21.
18. Zanny R. Efficiency of distributed priority queues in parallel adaptive integration. MS thesis. Kalamazoo, MI : Western Michigan University, 1999. 148 p.
19. Курносов М.Г. MPIPerf: пакет оценки эффективности коммуникационных функций стандарта MPI // Вестник Нижегородского университета им. Н.И. Лобачевского. 2012. № 5 (2). С. 385–391.

Поступила в редакцию 5 апреля 2019 г.

Paznikov A.A. (2020) DISTRIBUTED RELAXED QUEUE IN REMOTE MEMORY ACCESS MODEL. *Vestnik Tomskogo gosudarstvennogo universiteta. Upravlenie vychislitelnaja tehnika i informatika* [Tomsk State University Journal of Control and Computer Science]. 50. pp. 97–105

DOI: 10.17223/19988605/50/12

Remote memory access (RMA) technique is a very attractive way for improving efficiency of high-performance computations (HPC) and simplifying parallel program development. Unlike message-passing, in RMA programs processes communicate by directly accessing each other's memories. RMA model is implemented in MPI standard and offers Partitioned Global Address Space (PGAS). Many applications have been shown to benefit from RMA communications, where a process directly accesses the memory of another process instead of sending and receiving messages. To the best of knowledge, there are no efficient scalable concurrent data structures designed in RMA model.

In modern computer systems, multiple processes (threads) execute concurrently and synchronize their activities through shared (concurrent) data structures. Such data structures are therefore key building blocks in parallel programming, and their efficiency is crucial for the overall performance. Concurrent data structures are, however, much more difficult to design than their sequential counterparts, as processes executing concurrently might interleave their shared-memory steps in very complicated ways, with often unexpected outcomes. Coming up with efficient concurrent data structures for distributed environments with deep hierarchy, such as computer clusters and data centers, is challenging. A lot of prior work focused on designing efficient synchronization techniques for shared-memory systems. However, in such systems the shared memory itself may become an impediment for scalability. Moreover, shared-memory systems are not sufficient for processing of large data volumes in current applications. Hence, there is growing demand for efficient concurrent data structures in hierarchical distributed systems (supercomputers, clusters, grids).

Regardless of the design, many data structures are subject to inherent sequential bottlenecks for some operations, such as the delete-min operation for priority queues or insert and remove operations for queues and stacks. A promising way to alleviate the bottleneck problem is relaxing the consistency requirements for such operations. There are evidences that, on most workloads, relaxed data structures outperform data structures with strict semantics and ensure acceptable degrees of operation reordering. However, to the best of our knowledge, nobody looked at relaxed concurrent structures in the distributed environment.

As data structures to study, we consider relaxed queues. Relaxed queues do not guarantee strict FIFO order: remove operation might not remove exactly the first inserted element, but an element close to it. We propose an approach based on multiple sequential data structures distributed among the processes. This approach is well approved for shared-memory systems and outperform data structures with strong ordering. Every process can asynchronously access to the remote segment via RMA calls. Thanks to low latency of one-sided communications and hardware support of RDMA this scheme will guarantee high performance. When a process executes insert operation for the relaxed queue, it sets timestamp value and just picks (randomly or by a specified algorithm) the remote process and inserts the element along with timestamp to its data structure. When it executes a remove operation, the calling process selects a subset of other processes and remotely gets "candidate" elements from the set of processes. Finally, it chooses among candidate elements the "best" one with minimal timestamp and returns it.

We evaluated developed relaxed queue on computer cluster. In experiments we compared developed distributed queue with the linked list, implemented in MPICH library (MPICH linked list) in RMA model. Throughput of developed relaxed distributed queue substantially outperforms MPICH linked lists. Optimization was achieved by reducing overheads for communications while accessing the elements of the structures. The main drawback of common distributed lists that the head pointers become sequential bottlenecks. Unlike them developed distributed queue has multiple access points distributed among the processes and no bottlenecks. In the work we also investigate the influence of candidate elements number and chosen type of synchronization on the efficiency of the queues. As expected, throughput is decreasing with increase of number of candidates. This is explained the additional overheads for locking, getting states and values from candidate queues. We also found for the relaxed queue exclusive mode of synchronization provides better throughput compared with shared mode.

Thus, proposed decentralized asynchronous approach for designing relaxed distributed data structures, as we expected, eliminates bottlenecks, minimizes latency of the operations and provides high scalability of parallel programs.

Keywords: distributed queue; relaxed data structures; remote memory access; MPI; RMA.

PAZNIKOV Alexey Aleksandrovich (Candidate of Technical Sciences, Senior Researcher, Department of Computer Science and Engineering, Saint Petersburg Electrotechnical University "LETI", Saint Petersburg, Russian Federation).
E-mail: apaznikov@gmail.com

REFERENCES

1. Herlihy, M. & Shavit, N. (2012) *The Art of Multiprocessor Programming*. Morgan Kaufmann.
2. Mark, M. & Shavit, N. (2004) *Concurrent Data Structures*. Chapman and Hall/CRC Press.
3. Shavit, N. (2011) Data structures in the multicore age. *Communications of the ACM*. 54. pp. 76–84. DOI: 10.1145/1897852.1897873

4. 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. DOI: 10.1023/B:IJPP.0000029272.69895.c1
5. 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). pp. 9. DOI: 10.1145/2780584. 30
6. 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
7. 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
8. Rihani, H., Sanders, P. & Dementiev, R. (2015) Brief announcement: Multiqueues: Simple relaxed concurrent priority queues. *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. pp. 80-82. DOI: 10.1109/EIConRus.2019.8657105
9. Tabakov, A.V. & Paznikov, A.A. (2018) Algorithms for optimization of relaxed concurrent priority queues in multicore systems. *Izvestia SPbGETU "LETI"*. 10. pp. 42–49.
10. Dodds, M., Haas, A. & Kirsch, C.M. (2015) A scalable, correct time-stamped stack. *ACM SIGPLAN Notices*. 50(1). pp. 233–246. DOI: 10.1145/2775051.2676963
11. Alistarh, D., Kopinsky, J., Li, J. & Shavit, N. (2015) The Spray List: A scalable relaxed priority queue. *ACM SIGPLAN Notices*. 50(8). pp. 11–20.
12. Wimmer, M., Gruber, J., Träff, J.L., & Tsigas, P. (2015) The lock-free k-LSM relaxed priority queue. *ACM SIGPLAN Notices*. 50(8). pp. 277–278. DOI: 10.1145/2858788.2688547
13. Henzinger, T.A., Kirsch, C.M., Payer, H., Sezgin, A. & Sokolova, A. (2013) Quantitative relaxation of concurrent data structures. *ACM SIGPLAN Notices*. 48(1). pp. 317–328. DOI: 10.1145/2480359.2429109
14. Afek, Y, Korland, G. & Yanovsky, E. (2010) Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. In: Lu, C., Masuzawa, T. & Mosbah, M. (eds) *Principles of Distributed Systems. OPODIS 2010. Lecture Notes in Computer Science*. Vol 6490. Berlin, Heidelberg: Springer.
15. Schmid, P., Besta, M. & Hoefler, T. (2016) High-Performance Distributed RMA Locks. *Proc. of the 25th ACM Int. Symposium on High-Performance Parallel and Distributed Computing, HPDC 2016*. Kyoto, Japan, May 31 – June 4, 2016. ACM 2016. pp. 19–30.
16. Mans, B. (1998) Portable distributed priority queues with MPI. *Concurrency – Practice and Experience*. 10(3). pp. 175–198.
17. Brodal, G.S., Traff, J.L. & Zaroliagis, C.D. (1998) A parallel priority queue with constant time operations. *Journal of Parallel and Distributed Computing*. 49(1). pp. 4–21. DOI: 10.1006/jpdc.1998.1425
18. Zanny, R. (1999) *Efficiency of distributed priority queues in parallel adaptive integration*. MS Thesis. Western Michigan University.
19. Kurnosov, M.G. (2012) MPIPerf: paket otsenki effektivnosti kommunikatsionnykh funktsiy standarta MPI [MPIPerf: a toolkit for benchmarking MPI libraries]. *Vestnik Nizhego-rodskogo universiteta im. N.I. Lobachevskogo – Vestnik of Lobachevsky University of Nizhni Novgorod*. 5(2). pp. 385–391.