Масштабируемые алгоритмы реализации потокобезопасных пулов на основе рассеивающих деревьев

А. Д. Аненков
Институт физики полупроводников им. А.В. Ржанова
Сибирского отделения РАН
Новосибирск, Россия
anenkov.ru@gmail.com

А. А. Пазников[1], М. С. Куприянов[2]
Санкт-Петербургский государственный
электротехнический университет
«ЛЭТИ» им. В. И. Ульянова (Ленина)
Санкт-Петербург, Россия
[1]apaznikov@gmail.com, [2]mskupriyanov@etu.ru

***Аннотация*. Разработаны масштабируемые разделяемые пулы (pools) на основе рассеивающих деревьев (diffracting tree). Созданные структуры данных позволяют локализовать обращения потоков к разделяемым структурам данных для повышения пропускной способности. Выполнено натурное моделирование созданных пулов. Пулы характеризуются большей масштабируемостью, по сравнению с аналогичными имеющимися реализациями на основе рассеивающих деревьев. Построены рекомендации для практического применения пулов и экспериментальные данные моделирования на многопроцессорных вычислительных системах с общей памятью.**

Ключевые слова: многопоточное программирование; распределяющие деревья; неблокируемые структуры данных; масштабируемость; потокобезопасный пул

I. Введение

С ростом количества ядер в многоядерных вычислительных системах (ВС) всё более актуальной является задача построения масштабируемых средств синхронизации потоков для их доступа к разделяемым структурам данных. Среди широко используемых структур данных в многопоточном программировании в настоящее время можно выделить потокобезопасный пул. Пулом (pool) является неупорядоченный массив элементов, обеспечивающая операции добавления (push) и извлечения (pop) элементов [1]. Такие структуры данных широко применяются в многопоточных программах, реализующих модель производитель-потребитель (producer-consumer).

В работе предлагается новый подход к реализации масштабируемых потокобезопасных пулов на основе рассеивающих деревьев и алгоритмы его оптимизации. Данный подход основан на минимизации накладных расходов за счет локализации обращений к узлам дерева и применения локальной памяти потоков. Описанные подход позволяет существенно увеличить эффективность пула (пропускную способность) как для высокой, так и для низкой частоты операций, а также характеризуется высоким уровнем упорядоченности операций. Кроме того, пул обеспечивает низкую латентность прохода от корня рассеивающего дерева к его листьям.

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

II. Обзор существующих работ

Распространенные разделяемые пулы на базе блокируемых линейных списков [2, 3], в том числе с применением масштабируемых блокировок [4–6], слабо масштабируются при увеличении числа потоков и частоты выполнения операций. Подходы на основе work-pile, work-stealing [7, 8] и транзакционной памяти [9, 10] неэффективны, соответственно, при низкой и высокой частоте операций. В целях увеличения пропускной способности используется подход на основе устранения парных операций (elimination) [11–13]. Среди перспективных методов можно также выделить пулы на основе элиминирующих деревьев [14] и делегирования [15]. Основной недостаток применения описанных подходов в том, что голова и хвост списков становятся «узким местом», что приводит к увеличению состязательности и снижению эффективности кэш-памяти.

Среди наиболее эффективных подходов для сокращения конкурентного доступа потоков можно выделить метод на основе рассеивающих деревьев (diffracting tree) [16, 17]. Существующие реализации пулов на основе рассеивающих деревьев имеют высокие накладные расходы на активное ожидание в массиве устранения парных операций и выполнение атомарных операций при прохождении каждого уровня дерева. Пропускная способность рассеивающего дерева существенно сокращается с ростом его размера. Помимо этого, его эффективность падает в случае выполнения операций одним потоком. Кроме того, для практического использования таких пулов необходимо подобрать субоптимальные параметры (таймаут ожидания в массиве устранения, максимальное число коллизий и т.д.). Авторы статей [18, 19] предлагают оптимизированные пулы на основе адаптивного рассеивающего дерева, размер которого динамически зависит от частоты операций. Однако и такие методы не позволяют избежать накладных

расходов при синхронизации потоков в массивах устранения и узлах дерева.

III. Потокобезопасный пул

A. *Потокобезопасный пул на основе рассеивающего дерева*

Рассмотрим вычислительную систему с общей памятью, укомплектованную n процессорными ядрами. Обозначим p число параллельных потоков. Потоки привязаны (affinity, bind) к ядрам, привязка задана функцией $a(i)$, где i – номер потока, $w \in \{1, 2, ..., n\}$ – процессорное ядро, к которому привязан поток. Производитель (producer) – поток, реализующий операцию добавления (push, insert) данных в пул. Потребитель (consumer) – поток, реализующий удаление (pop, remove) данных из пула.

Рассеивающее дерево (diffraction tree) [16-19] – это бинарное дерево высотой h. Узел такого дерева (balancer) содержит биты, определяющие направления обращений потоков при их обходе дерева. Узлы дерева перенаправляет запросы на добавление или удаление элементов на один из узлов-потомков поочередно в зависимости от значения бита. В листьях дерева находятся потокобезопасные очереди $q = \{1, 2, ..., 2^h\}$. Потоки, выполняя операции добавления и удаления, проходят дерево от корня к листьям и добавляют (удаляют) элемент из соответствующей очереди.

Отметим, что в реальных многопоточных программах можно пренебречь строгим порядком распределения потоков по листьям [20]. Данный принцип используется в предложенных в данной работе структурах данных. Предложены две реализации потокобезопасного пула на базе рассеивающего дерева. Алгоритмы основаны на идее локализации обращений к узлам дерева и использование локальной памяти потока (thread-local storage, TLS).

B. *Оптимизированный пул на основе рассеивающего дерева*

Создан пул LocOptDTPool, в котором, в котором, в отличие от оригинального метода, узел рассеивающего дерева включает два массивы атомарных булевых переменных (для производителей и потребителей соответственно) длины $m \le p$ (рис. 1). На каждом следующем уровне узлы дерева содержат массивы длины в два раза меньшей, по сравнению с предыдущим уровнем.

Поток выполняет операции с соответствующим ему битом в массиве. Это гарантирует локализацию обращений к атомарным битам в узлах дерева. Кроме того, по сравнению с применением массива устранения парных операций, данный подход минимизирует время на обращение к ячейкам вспомогательного массива и активным ожиданием появления комплементарного потока. Поток, обращаясь к узлу дерева выбирает в массиве атомарных битов ячейку в соответствии со значением хеш-функции

$$h(i) = i \bmod m, \qquad (1)$$

где $i \in \{1, 2, \ldots, p\}$ – номер потока, который присваивается потоку при первом обращении к пула.

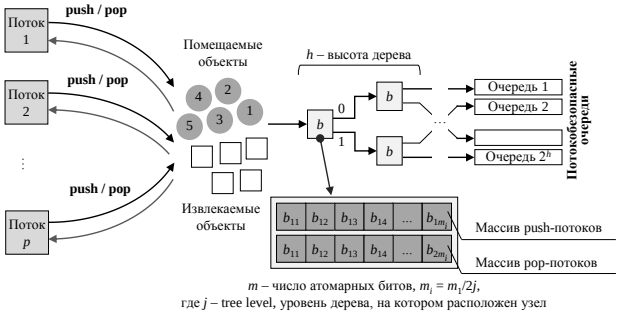

Рис. 1. Оптимизированный пул LocOptDTPool на основе рассеивающего дерева ($h = 2$)

Опишем метод распределения очередей в листьях дерева по ядрам. Каждое ядро $j \in \{1, 2, \ldots, p\}$ соответствует очереди $q_j = \{j2^h / n, \; j2^h / n + 1, \; ..., \; (j + 1)2^h / n - 1\}$. Пусть поток i привязан к ядру j ($a(i) = \{j\}$). Тогда все объекты, добавляемые (удаляемые) в пул данным потоком, должны распределяться между очередями множества q_j (очереди для хранения объектов, поступающих от потоков, привязанных к ядру j). Описанный подход обеспечивает минимизацию числа промахов по кэшу благодаря пространственной локализации обращений к разделяемым переменным.

Существенным недостатком оригинальной реализации пула на базе рассеивающего дерева [18] является увеличение накладных расходов при малом количестве активных потоков (1–2 потока). Для этого в созданном пуле LocOptDTPool используется адаптивная схема. Учитывается текущее количество активных потоков с помощью двух атомарных счетчиков. Если загрузка пула недостаточна, то распределение объектов между очередями не приводит к существенному повышению пропускной способности пула. В этом случае для хранения объектов используется одна потокобезопасная очередь.

Разделяемый пул LocOptDTPool включает в себя рассеивающее дерево *tree*, массивы атомарных битов *prod_bits* и *cons_bits*, массивы очередей *queues*, менеджер *af_mgr* управления привязкой потоков к процессорным ядрам, счетчики *prod_num* и *cons_num* числа потоков в пуле и методы push и pop помещения и извлечения объектов из пула соответственно. В качестве очередей *queues* в данной реализации применяется очереди без блокировок из библиотеки boost. Рекомендуется также использовать очереди с ослабленной семантикой [21–24].

При вызове метода push потоком j выполняются следующие шаги:

1. Инкрементирование счетчика *prod_num* на 1.

2. Привязка потока к процессорному ядру с помощью менеджера привязки *af_mgr* (в случае, если это не было выполнено ранее).

3. Выбор очереди, в которую будет помещен объект *data* с помощью формулы:

$$q_j = (n \cdot l \bmod (2h / n) + a(j)), \quad (2)$$

где n – число процессорных ядер, l – лист дерева, посещенный потоком, $a(j)$ – номер процессорного ядра, к которому привязан поток j, 2^h – общее число разделяемых очередей.

Алгоритм удаления pop включает следующие шаги:

1. Привязка потока с помощью менеджера привязки *af_mgr* и инкрементирование счетчика *cons_num.*

2. Очередь для извлечения элемента выбирается в соответствии с формулой:

$$q_j = (p\alpha + a(j)), \quad (3)$$

где α – коэффициент сдвига, определяемый эмпирически.

3. Если очередь q_j пуста, то элемент извлекается из первой следующей за ней непустой очереди. Отметим, что такой подход используется и в других реализациях пулов [19, 25]. При успешном выполнении операции, метод pop возвращает извлеченный объект, в случае неудачи выполняется повторный вызов метода.

C. Оптимизированный пул с использованием локальных данных потока

Создан масштабируемый потокобезопасный пул TLSDTPool на базе размещения булевых переменных в узлах дерева в области локальной памяти потока (Thread-local storage, TLS). Данный подход минимизирует конкурентность доступа к разделяемым битам в узлах дерева [12–15]. Структура BitArray размещается в TLS потока. Это позволяет отказаться от использования атомарных операций с высокими накладными расходами при выполнении обращений к массиву *bits* в структуре BitArray; в качестве *bits* используется обычный массив булевых переменных.

Отметим, что так как потоки не могут прочитать значения битов других потоков, рассеивающее дерево может не обеспечивать равномерное распределение загрузки между очередями. Для решения данной проблемы предлагается использовать эвристический алгоритм инициализации элементов бинарных массивов в узлах. Алгоритм основан на представлении идентификатора потока в двоичном виде (рис. 2). Разряд в двоичном преставлении идентификатора является соответствующим уровнем дерева, а значение разряда – начальное состояние битов в узлах дерева на данном уровне.

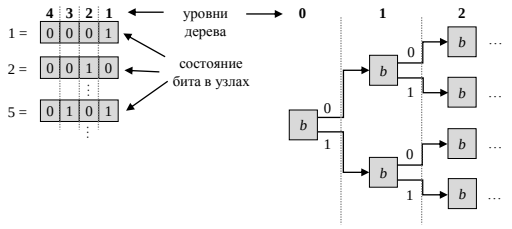

Рис. 2. Распределение потоков по узлам дерева в пуле TLSDTPool

Мы ожидаем, что описанная схема в случае постоянно активных потоков позволяет равномерно распределить обращения к узлам дерева для сокращения дисбаланса загрузки очередей в листьях дерева.

IV. Результаты моделирования

Экспериментальное исследование разработанных пулов проводилось на узле кластера Jet Центра параллельных вычислительных технологий Сибирского государственного университета телекоммуникаций и информатики. Показатели эффективности пула – пропускная способность $b = N / t$, где N – число выполненных операций, t – время моделирования. Эффективности пула также проанализирована при использовании различных реализаций очередей в листьях дерева. От данного выбора зависит пропускная способность пула; отметим, что такой подход к моделированию пулов применялся и в других работах [17]. Приводятся также результаты моделирования пула на основе одной неблокируемой очереди из библиотеки boost. Эксперименты выполнялись в два этапа: для числа потоков p = 1, 2, …, 8 (не более числа процессорных ядер), и для большого количества потоков p = 10, 20, …, 200.

Результаты экспериментов демонстрируют, что пулы LocOptDTPool и TLSDTPool характеризуются высокой масштабируемостью в случае большого количества потоков. Это подтверждается ростом пропускной способности по мере достижения числа потоков, равного количеству процессорных ядер (рис. 3, 4). При количестве потоков около числа процессорных ядер получена максимальная пропускная способность. Эффективность пула LocOptDTPool на всём диапазоне числа потоков соответствует сопоставима с эффективностью TLSDTPool. С увеличением числа потоков применение неблокируемых очередей Lockfree queue в пулах LocOptDTPool и TLSDTPool позволяет увеличить пропускную способность, по сравнению потокобезопасными очередями на основе блокировок (рис. 3б, 4б). Во всех случаях эффективность отдельной потокобезопасной очереди Lockfree queue значительно ниже эффективности построенных пулов (рис. 3, 4). Важно отметить, что дисбаланс загрузки очередей зафиксирован не был, что подтверждает эффективность предложенного эвристического алгоритма снижения дисбаланса.

V. Заключение

Созданы масштабируемые разделяемые пулы на базе рассеивающих деревьев. Основная идея предлагаемых алгоритмов заключается в локализации обращений потоков к разделяемым структурам данных. Данная локализация обеспечивает повышение пропускной способности. Созданные пулы могут использоваться в программах в модели производитель-потребитель в случае постоянного числа активных потоков с целью обеспечения высокой пропускной способности и низкой латентности выполнения операций. Пул обеспечивает большую масштабируемость при выполнении многопоточных программ, по сравнению с аналогичными реализациями пула на основе рассеивающих деревьев. Максимальная

пропускная способность получена для количества потоков сопоставимого с числом процессорных ядер системы. Увеличение высоты рассеивающего дерева не снижает эффективность пулов. В качестве структур данных в листьях дерева можно рекомендовать потокобезопасные очереди без использования блокировок.

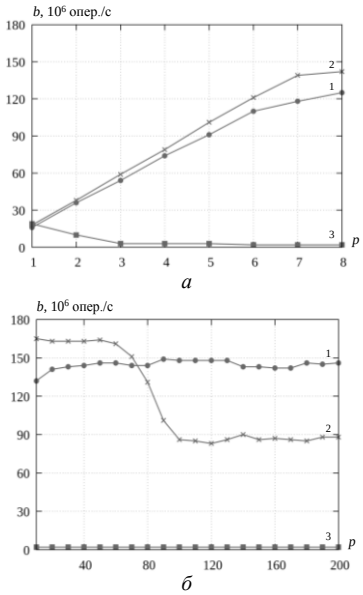

Рис. 3. Результаты анализа пропускной способности. *а* – p = 1, 2, …, 8 (8 – число ядер), *б* – p = 10, 20, …, 200. 1 – LocOptDTPool, неблокируемые очереди (boost), 2 – LocOptDTPool, блокируемые очереди (PThreads mutex), 3 – неблокируемая очередь (boost)

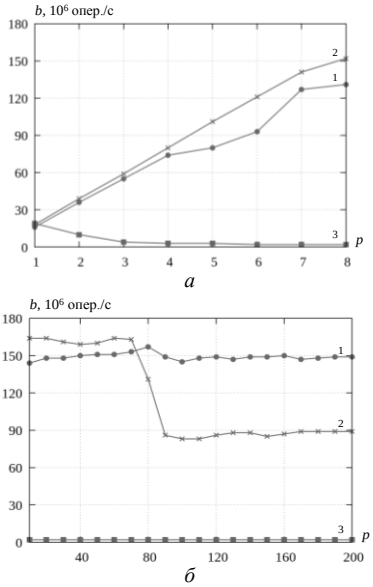

Рис. 4. Результаты анализа пропускной способности. *а* – p = 1, 2, …, 8 (8 – число ядер), *б* – p = 10, 20, …, 200. 1 – LocOptDTPool, неблокируемые очереди (boost), 2 – LocOptDTPool, блокируемые очереди (PThreads mutex), 3 – неблокируемая очередь (boost)

Список литературы

[1] Herlihy M., Shavit N. The Art of Multiprocessor Programming, Revised Reprint. Elsevier, 2012, 528 p.

[2] Anderson T.E. The performance of Spin Lock Alternatives of Shared Memory Multiprocessors // TPDS. 1990. V.1(1). P. 6-16.

[3] Mellor-Crummey J. M., Scott M. L. Synchronization without contention // ACM SIGPLAN Notices. 1991. V. 26(4). P. 269-278.

[4] Paznikov A., Shichkina Y. Algorithms for Optimization of Processor and Memory Affinity for Remote Core Locking Synchronization in Multithreaded Applications // Information, 2018. Vol. 9. N. 1. P. 1-12.

[5] 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.

[6] Goncharenko E.A., Paznikov A.A., Tabakov A.V. Evaluating the performance of atomic operations on modern multicore systems // Journal of Physics: Conference Series, 2019, Vol. 1399, No. 3, P. 033107.

[7] Rudolph L., Slivkin M., Upfal E. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. // SPAA. 1991. P. 237-245.

[8] Blumofe R., Leiserson C. Scheduling multithreaded computations by work stealing // Journal of the ACM. 1999. V. 46(5). P. 720-748.

[9] D. Touitou and N. Shavit. Software transactional memory. “Distributed Computing,” 1997, vol. 10, no. 2, P. 99-116.

[10] Смирнов В.А., Омельниченко А.Р., Пазников А.А. Алгоритмы реализации потокобезопасных ассоциативных массивов на основе транзакционной памяти // Известия СПбГЭТУ «ЛЭТИ». 2018. № 1. С. 12-18.

[11] Hendler D., Shavit N., Yerushalmi L. A scalable lock-free stack algorithm // SPAA. 2004. P. 206-215.

[12] Moir M. et al. Using elimination to implement scalable and lock-free FIFO queues // SPAA. 2005. P. 253-262.

[13] Afek Y., Hakimi M., Morrison A. Fast and scalable rendezvousing // Distributed computing. 2013. Vol. 26(4). P. 243-269.

[14] Shavit N., Touitou D. Elimination trees and the construction of pools and stacks: preliminary version // SPAA. 1995. P. 54-63.

[15] Calciu I., Gottschlich J.E., Herlihy M. Using elimination and delegation to implement a scalable NUMA-friendly stack // HotPar. 2013. P. 1-7.

[16] Shavit N., Zemach A. Diffracting trees // ACM Transactions on Computer Systems (TOCS). 1996. V. 14(4). pp. 385-428.

[17] Afek Y., Korland G., Natanzon M., Shavit N. Scalable Producer-Consumer Pools based on Elimination-Diffraction Trees // European Conference on Parallel Processing. 2010. P. 151-162.

[18] Della-Libera G., Shavit N. Reactive diffracting trees. // Journal of Parallel and Distributed Computing. 2000. V. 60(7). P. 853-890.

[19] На P.H., Papatriantafilou M., Tsigas P. Self-tuning reactive distributed trees for counting and balancing. // OPODIS. 2004. P. 213-228.

[20] Shavit N. Data Structures in the Multicore Age. // Communications of the ACM. 2011. Vol. 54(3). P. 76-84.

[21] Tabakov A.V., Paznikov A.A. Using relaxed concurrent data structures for contention minimization in multithreaded MPI programs // Journal of Physics: Conference Series. 2019. Vol. 1399. No. 3. P. 033037.

[22] Paznikov A., Anenkov A. Implementation and Analysis of Distributed Relaxed Concurrent Queues in Remote Memory Access Model // Proc. of the 13th International Symposium “Intelligent Systems – 2018” (INTELS’18). Procedia Computer Science, 2019. Vol. 150. P. 654-662.

[23] 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

[24] Пазников А.А. Распределенная очередь с ослабленной семантикой выполнения операций в модели удаленного доступа к памяти // Вестник Томского государственного университета. Управление, вычислительная техника и информатика. 2020. № 50. С. 97-105.

[25] Chase D., Lev Y. Dynamic circular work-stealing deque // SPAA. 2005. P. 21-28.