Методы оптимизации потокобезопасных пулов на основе распределяющих деревьев для многоядерных вычислительных систем с общей памятью

А. Д. Аненков[1], А. А. Пазников[2]
Санкт-Петербургский государственный электротехнический университет
«ЛЭТИ» им. В.И. Ульянова (Ленина)
[1]anenkov.ru@gmail.com, [2]apaznikov@gmail.com

***Аннотация*. В работе представлены реализации масштабируемых потокобезопасных пулов на основе распределяющих деревьев. Разработанные пулы обеспечивают локализацию обращений параллельных потоков к разделяемым переменным в целях максимизации пропускной способности. Проведен анализ эффективности разработанных пулов. Пулы обеспечивают большую масштабируемость в многопоточных программах, по сравнению с аналогичными имеющимися реализациями на основе распределяющих деревьев. Приведены рекомендации по использованию пулов и экспериментальные результаты моделирования на многоядерных вычислительных системах с общей памятью.**

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

I. Введение

С увеличением числа процессорных ядер в современных вычислительных системах (ВС) [1] остро становится задача обеспечения масштабируемого доступа к разделяемым структурам данных. Одной из наиболее широко используемых структур данных на сегодняшний день является потокобезопасный пул. Пул (pool) – это неупорядоченный набор объектов, реализующая операции добавления (push) и извлечения (pop) объектов [2]. Пулы широко применяются при реализации модели производитель-потребитель (producer-consumer) в многопоточных программах.

В данной работе предлагается оригинальный подход к реализации потокобезопасного пула без использования блокировок на основе распределяющих деревьев и методы его оптимизации. Подход основан на локализации обращений к узлам дерева и использовании локальной памяти потоков. Он позволяет повысить пропускную способность при высоких и низких загрузках пула, обеспечивает приемлемый уровень FIFO/LIFO-порядка выполнения операций и характеризуется незначительной временной задержкой прохода от корня распределяющего дерева к его листьям.

Работа выполнена при поддержке Совета по грантам Президента РФ для государственной поддержки молодых российских ученых (проект СП-4971.2018.5). Публикация выполнена в рамках государственной работы «Инициативные научные проекты» базовой части государственного задания Министерства образования и науки Российской Федерации (ЗАДАНИЕ № 2.6553.2017/БЧ). Работа выполнена при финансовой поддержке Министерства образования и науки Российской Федерации в рамках договора № 02.G25.31.0149 от 01.12.2015 г.

II. Обзор существующих работ

Существующие реализации пулов, основанные на блокируемых очередях [3, 4] недостаточно масштабируются для большого количества потоков и высокой интенсивности обращений. Методы work-pile и work-stealing [6, 5] неэффективны при низкой частоте обращений к пулу. Для повышения масштабируемости применяется метод обработки комплементарных операций (elimination) [7, 8, 9]. К альтернативным подходам можно отнести реализации на базе устраняющих деревьев [10]. Основной недостаток использования списков в том, что вершины списков становятся «узкими местами», что приводит к увеличению конкурентности доступа и снижению эффективности использования кэш-памяти. Этим же недостатком обладает метод делегирования выполнения операций выделенным потокам [11].

Одним из перспективных подходов для сокращения конкурентного доступа параллельных потоков является применение распределяющих деревьев (diffraction tree) [12, 13]. К недостаткам существующих реализаций можно отнести накладные расходы, связанные с активным ожиданием в массиве устранения комплементарных операций и синхронизацией атомарных переменных на каждом уровне дерева. Эффективность распределяющего дерева значительно снижается с увеличением его размера. Кроме того, не учитывается возможность использования пула для добавления и удаления элементов одним потоком. Также к недостаткам относится необходимость подбора параметров (время ожидания поступления парных операций, допустимое количество коллизий и т.д.). В работах [14, 15] предлагается оптимизация в виде адаптивного распределяющего дерева, размер которого определяется уровнем конкурентного доступа. Однако такие пулы характеризуются существенными накладными расходами вследствие синхронизации во вспомогательных массивах и в узлах дерева.

III. Потокобезопасный пул

A. *Потокобезопасный пул на основе распределяющего дерева*

Пусть имеется многоядерная ВС, состоящая из n процессорных ядер. В системе выполняется параллельная программа, включающая p потоков. Привязка потоков к процессорным ядрам определяется функцией $a(i)$, ставящей в соответствие потоку i процессорное ядро $w \in \{1, 2, ..., n\}$, к которому привязан поток. Будем называть производителем (producer) поток, выполняющий операцию добавления (push) элементов в пул, и потребителем (consumer) – поток, выполняющий операцию удаления (pop) элементов из пула.

Распределяющее дерево (diffraction tree) [12–15] представляет собой бинарное дерево высотой h, в каждом узле которого находятся биты, определяющие направления обращений потоков. Узлы дерева (balancers) перенаправляют поступающие запросы на добавление (push) или удаление (pop) элементов поочередно на один из узлов-потомков. Листьям дерева соответствуют очереди $q = \{1, 2, ..., 2^h\}$. При выполнении операций потоки проходят дерево от корня к листьям и помещают (извлекают) элемент в соответствующую очередь.

Для практических целей последовательный порядок распределения потоков по листьям не требуется [16], и им можно пренебречь с целью повышения пропускной способности. Авторами предложены алгоритмы реализации потокобезопасного пула на базе распределяющего дерева. В основе алгоритмов лежит идея локализации обращений к узлам дерева и использование локальной памяти потока (thread-local storage, TLS).

B. *Оптимизированный пул на основе распределяющего дерева*

Авторами был разработан пул LocOptDTPool, в котором каждый узел распределяющего дерева содержит два массива атомарных битов (для потоков-производителей и потоков-потребителей) размера $m \le p$ (вместо двух отдельных атомарных битов) (рис. 1). Узлы каждого следующего уровня дерева содержат в два раза меньшие по размеру массивы, по сравнению с предыдущим уровнем.

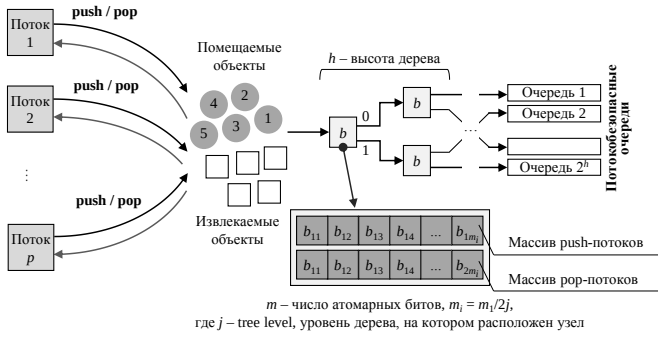

Рис. 1. Оптимизированный пул LocOptDTPool на основе распределяющего дерева (h = 2)

Каждый поток обращается к соответствующему ему биту в массиве, что обеспечивает локализацию обращений к атомарным битам в узлах дерева. Кроме того, по сравнению с использованием массива устранения комплементарных операций, данный подход позволяет сократить накладные расходы, связанные с обращением к ячейкам вспомогательного массива и активным ожиданием парного потока. При каждом посещении потоком узла дерева, в массиве атомарных битов выбирается ячейка по значению хеш-функции

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

где $i \in \{1, 2, \ldots, p\}$ – порядковый номер потока, выданный при первом посещении пула.

Рассмотрим схему распределения очередей в листьях дерева между процессорными ядрами. Каждому ядру $j \in \{1, 2, \ldots, p\}$ ставятся в соответствие 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, помещаемые (извлекаемые) в пул этим потоком, распределяются между очередями q_j, предназначенными для хранения объектов, поступающих от потоков, привязанных к ядру j. Данный подход позволяет сократить количество промахов по кэшу благодаря локализации обращений к разделяемым переменным.

Одним из основных недостатков существующей реализации пула на основе распределяющего дерева [13] является увеличение времени выполнения операций при малом количестве активных потоков (1-2 потока). Для решения проблемы в пуле LocOptDTPool учитывается текущее число активных потоков. Если текущая загрузка пула мала, то распределение объектов между очередями не приводит к существенному повышению пропускной способности пула. В этом случае для хранения объектов используется одна очередь. Подсчет количества активных потоков в пуле реализован в виде двух атомарных счетчиков.

Описанный потокобезопасный пул LocOptDTPool (рис. 2) включает в себя распределяющее 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.

```
1  class LocOptDTPool {
2    Node tree
3    BitArray prod_bits[m], cons_bits[m]
4    ThreadSafeQueue queues[n]
5    AffinityManager af_mgr
6    AtomicInt prod_num, cons_num
7    push(data)
8    pop()
9  }
```

Рис. 2. Структура потокобезопасного пула LocOptDTPool

В качестве потокобезопасных очередей *queues* в данной работе применяется реализация очередей без использования блокировок из библиотеки boost [17].

При вызове метода push потоком j выполняются следующие шаги:

1. Увеличение счетчика *prod_num* на единицу.

2. Привязка текущего потока к процессорному ядру с помощью менеджера привязки *af_mgr*, если это не было сделано ранее.

3. Выбор очереди, в которую будет помещен объект *data* в соответствии с формулой:

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

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

Метод pop, выполняемый потоком j, включает в себя следующие шаги:

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

2. Выбор очереди для извлечения по формуле:

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

где α – коэффициент сдвига, выбираемый эмпирически.

3. Если очередь q_j пуста, то элемент извлекается из первой следующей за ней непустой очереди. Такой подход используется в других реализациях пулов [16, 18]. В случае успешного выполнения операции метод pop возвращает извлеченный объект, а в случае неудачи выполняется повторный вызов метода.

C. Оптимизированный пул с использованием локальных данных потока

Разработан масштабируемый пул TLSDTPool, в основе которого лежит идея размещения битов в узлах дерева в области локальной памяти потока (Thread-local storage, TLS). Данный подход позволяет сократить конкурентность доступа к разделяемым битам в узлах дерева [12–15]. Суть предлагаемого подхода заключается в том, что структура BitArray размещается в TLS потока. Это позволяет отказаться от использования дорогостоящих атомарных операций при выполнении обращений к массиву *bits* в структуре BitArray; в качестве *bits* используется обычный массив булевых переменных.

Поскольку потоки не имеют доступа к состояниям битов других потоков, распределяющее дерево может не обеспечивать равномерное распределение загрузки между очередями. Для решения данной проблемы предлагается эвристический алгоритм инициализации элементов бинарных массивов в узлах дерева. Алгоритм основан на представлении идентификатора потока в двоичном виде (рис. 3). Каждый разряд в двоичном преставлении идентификатора является соответствующим уровнем дерева, а значение разряда указывает на начальное состояние битов в узлах дерева на данном уровне.

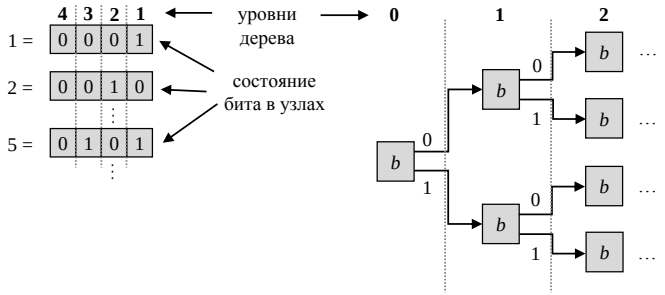

Рис. 3. Распределение потоков по узлам дерева в пуле TLSDTPool

Описанный алгоритм в случае постоянно активных потоков позволяет равномерно распределить обращения потоков к узлам дерева для предотвращения дисбаланса загрузки очередей в листьях дерева.

IV. Результаты экспериментов

Моделирование разработанных пулов проводилось на узле вычислительного кластера Jet Центра параллельных вычислительных технологий Сибирского государственного университета телекоммуникаций и информатики. В качестве показателя эффективности использовалась пропускная способность $b = N / t$, где N – число выполненных операций, t – время моделирования. Выполнен анализ эффективности пула при использовании различных типов очередей в листьях дерева. От выбора очередей зависит пропускная способность пула; такой подход при анализе эффективности пула применялся в других работах [16]. Для сравнения представлены результаты моделирования пула на основе одной неблокируемой очереди Lockfree queue из библиотеки boost [18]. Для каждого пула проводилось две серии экспериментов: для числа потоков p = 1, 2, …, 8, не превышающего число процессорных ядер, и для большого количества потоков p = 10, 20, …, 200.

Разработанные пулы LocOptDTPool и TLSDTPool хорошо масштабируется для большого количества потоков и демонстрирует рост пропускной способности по мере достижения числа потоков, равного количеству процессорных ядер (рис. 4, 5). Максимальная пропускная способность была получена при количестве потоков, равном количеству ядер процессора или незначительно его превышающем. Пропускная способность пула LocOptDTPool на всём диапазоне числа потоков соответствует пропускной способности пула TLSDTPool с применением локально-поточных битов. При большом количестве потоков применение неблокируемых очередей Lockfree queue в пулах LocOptDTPool и TLSDTPool обеспечивает большую пропускную способность, по сравнению блокируемыми потокобезопасными очередями (рис. 4*б*, 5*б*). Во всех случаях эффективность отдельной потокобезопасной очереди Lockfree queue значительно уступает эффективности разработанных пулов (рис. 4, 5). Отметим, что рассмотренные ранее случай дисбаланса загрузки очередей зафиксирован не был.

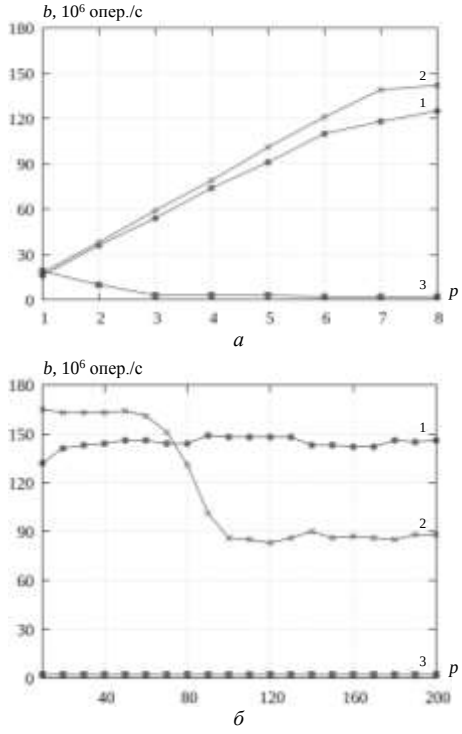

Рис. 4. Результаты анализа пропускной способности. *а* – $p = 1, 2, \ldots, 8$ (8 – число ядер), *б* – $p = 10, 20, \ldots, 200$. 1 – LocOptDTPool, неблокируемые очереди (boost), 2 – LocOptDTPool, блокируемые очереди (PThreads mutex), 3 – неблокируемая очередь (boost)

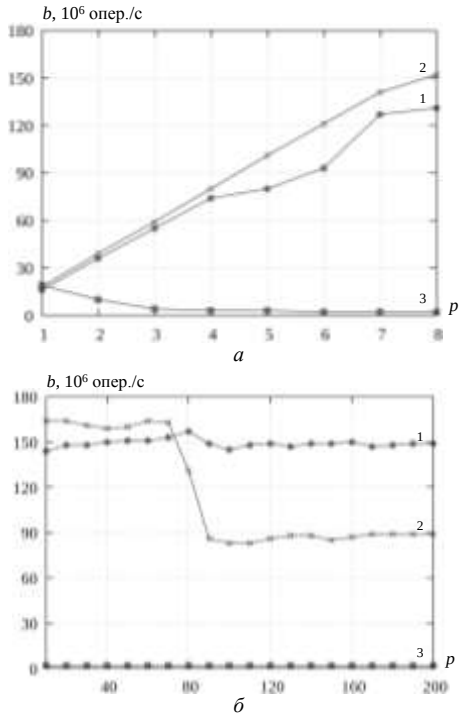

Рис. 5. Результаты анализа пропускной способности. *а* – $p = 1, 2, \ldots, 8$ (8 – число ядер), *б* – $p = 10, 20, \ldots, 200$. 1 – LocOptDTPool, неблокируемые очереди (boost), 2 – LocOptDTPool, блокируемые очереди (PThreads mutex), 3 – неблокируемая очередь (boost)

V. Заключение

Разработаны масштабируемые потокобезопасные пулы на основе распределяющих деревьев без использования блокировок. Суть оптимизаций заключается в локализации обращений потоков к разделяемым областям памяти с целью максимизации пропускной способности пула. Пулы могут применяться при реализации модели производитель-потребитель в многопоточных программах с постоянным числом активных потоков, где требуется высокая пропускная способность и быстрый возврат потоков из структуры. Пул обеспечивает большую масштабируемость при выполнении многопоточных программ, по сравнению с аналогичными реализациями пула на основе распределяющих деревьев. Наибольшая эффективность алгоритмов достигнута при числе активных потоков, равном количеству процессорных ядер. Увеличение размеров дерева в пуле не снижает пропускную способность. В качестве структур данных в листьях дерева рекомендуется использовать потокобезопасные очереди без использования блокировок.

Список литературы

[1] Хорошевский В.Г. Распределённые вычислительные системы с программируемой структурой // Вестник СибГУТИ. 2010. №2 (10). С. 3–41.

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

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

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

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

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

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

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

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

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

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

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

[13] 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.

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

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

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

[17] Blechmann T. Chapter 19. Boost.Lockfree. URL: http://www.boost.org/doc/ libs/1_61_0/doc/html/lockfree.html.

[18] Chase D., Lev Y. Dynamic circular work-stealing deque // SPAA. 2005. pp. 21–28.