ИНФОРМАТИКА И ПРОГРАММИРОВАНИЕ

УДК 004.272
DOI: 10.17223/19988605/39/10

А.Д. Аненков, А.А. Пазников

АЛГОРИТМЫ ОПТИМИЗАЦИИ МАСШТАБИРУЕМОГО ПОТОКОБЕЗОПАСНОГО ПУЛА НА ОСНОВЕ РАСПРЕДЕЛЯЮЩИХ ДЕРЕВЬЕВ ДЛЯ МНОГОЯДЕРНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Работа выполнена при поддержке РФФИ (гранты № 15-07-02693, 15-37-20113, 15-07-00653, 16-07-00712, 15-07-00048).

Предложена реализация масштабируемого потокобезопасного пула на основе распределяющих деревьев (diffraction trees). Созданный пул обеспечивает локализацию обращений к разделяемым областям памяти с целью максимизации его пропускной способности. Выполнен анализ эффективности созданного потокобезопасного пула. Пул обеспечивает большую масштабируемость при выполнении многопоточных программ, по сравнению с аналогичными реализациями пула на основе распределяющих деревьев. В работе сформулированы рекомендации по использованию пула. Приведены результаты натурных экспериментов на многоядерной вычислительной системе.

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

Современные многоядерные вычислительные системы (ВС) [1] являются большемасштабными и мультиархитектурными. Эффективность использования таких систем существенно зависит от средств синхронизации потоков в параллельных программах. С увеличением количества процессорных ядер в современных ВС особенно остро ставится задача обеспечения масштабируемого доступа к разделяемым структурам данных.

Одной из наиболее используемых структур данных на сегодняшний день является потокобезопасный пул. Пул (pool) – это неупорядоченная коллекция 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 модели один или несколько потоков-производителей порождают объекты, которые используются потоками-потребителями.

Наиболее простым подходом к реализации пулов является использование потокобезопасных очередей. Существующие реализации пулов, основанные на блокируемых очередях [3, 4], обеспечивают высокую производительность при незначительной частоте выполнения операций, однако недостаточно масштабируются для большого количества потоков и высокой интенсивности обращений к пулу. Методы workpile и work-stealing [5, 6] характеризуются прогнозируемым временем выполнения операций, но неэффективны при низкой частоте обращений к пулу.

Для повышения масштабируемости широко применяются потокобезопасные пулы на основе линейных списков, свободных от блокировок (lockfree). Одним из наиболее распространенных способов снижения конкурентного доступа (access contention) параллельных потоков к разделяемым областям памяти является метод обработки комплементарных операций (elimination backoff) добавления и удаления элементов в отдельном массиве [7, 8]. В работе [9] предложена усовершенствованная версия данного метода, основанная на использовании циклического буфера. К альтернативным подходам для создания масштабируемых пулов можно отнести реализации на базе устраняющих деревьев (elimination trees) [10]. Хотя использование неблокируемых потокобезопасных списков позволяет обеспечить прогнозируемое выполнение операций, однако при реализации неблокируемых пулов с помощью линей-

ных списков вершины этих списков становятся «узкими местами» (bottleneck), что приводит к увеличению конкурентности доступа и снижению эффективности использования кэш-памяти. Этим же недостатком обладает метод делегирования выполнения операций с пулом потокам-серверам, выполняющихся на выделенных процессорных ядрах [11, 12].

Одним из перспективных подходов для сокращения конкурентного доступа параллельных потоков является применение распределяющих деревьев (diffraction tree) [13]. В статье [14] предложена возможная реализация потокобезопасных пулов на основе распределяющих деревьев с использованием метода устранения комплементарных операций. К недостаткам реализации можно отнести дополнительные накладные расходы, связанные с активным ожиданием в массиве устранения комплементарных операций, а также синхронизацией атомарных переменных на каждом уровне распределяющего дерева в худшем случае, что существенно увеличивает трудоёмкость обхода дерева от корня к листьям. Эффективность распределяющего дерева значительно снижается с увеличением его размера. Кроме того, в работе [13] существенно нарушается FIFO/LIFO порядок выполнения операций, а также не учитывается возможность использования пула для добавления и удаления элементов одним потоком. Также к недостаткам относится необходимость подбора таких параметров, как время ожидания поступления комплементарных операций, допустимое количество коллизий и т.д.

В работах [15, 16] с целью снижения вышеописанных накладных расходов предлагается оптимизация в виде адаптивного распределяющего дерева (self-tuning reactive diffraction tree), размер которого определяется текущим уровнем конкурентного доступа потоков к его листьям. Тем не менее такие пулы характеризуются существенными накладными расходами вследствие синхронизации во вспомогательных массивах и в узлах дерева.

В данной статье предложены оригинальный подход к реализации потокобезопасного пула без использования блокировок на основе распределяющих деревьев и методы его оптимизации для постоянного числа активных потоков. Подход основан на локализации обращений к узлам дерева и использовании локальной памяти потоков. Предлагаемый подход 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.

1. Потокобезопасный пул на основе распределяющего дерева

Пусть имеется многоядерная ВС, состоящая из n процессорных ядер. Считаем, что система функционирует в монопрограммном режиме решения одной параллельной задачи [1], которая включает в себя p параллельных 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) [13–16] представляет собой бинарное дерево высотой h, в каждом узле которого находятся биты, определяющие направления обращений потоков (рис. 1). Узлы дерева (balancers) перенаправляют поступающие от потоков запросы на добавление (push) или удаление (pop) элементов поочередно на один из узлов-потомков: если значение бита равно 0, то поток обращается к узлу правого поддерева, если 1 – левого поддерева, и так далее до тех пор, пока потоки не дойдут до листьев дерева. После прохождения каждого узла потоки инвертируют в нём соответствующий бит.

Листьям распределяющего дерева соответствуют потокобезопасные очереди $q = \{1, 2, ..., 2^h\}$ (рис. 2). При выполнении операций потоки проходят дерево от корня к листьям и помещают (извлекают) элемент в соответствующую очередь. В состоянии покоя (quiescent state), при котором дерево сбалансировано и не содержит поступающих от потоков запросов, выходящие из дерева элементы распределяются между очередями таким образом, что количество элементов в верхней очереди превышает количество элементов в нижних очередях не более, чем на один. Таким образом, распределяющее дерево позволяет сократить конкурентность доступа к структуре данных.

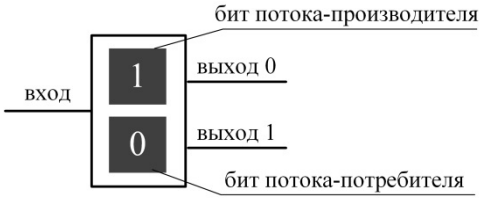

Рис. 1. Узел распределяющего дерева

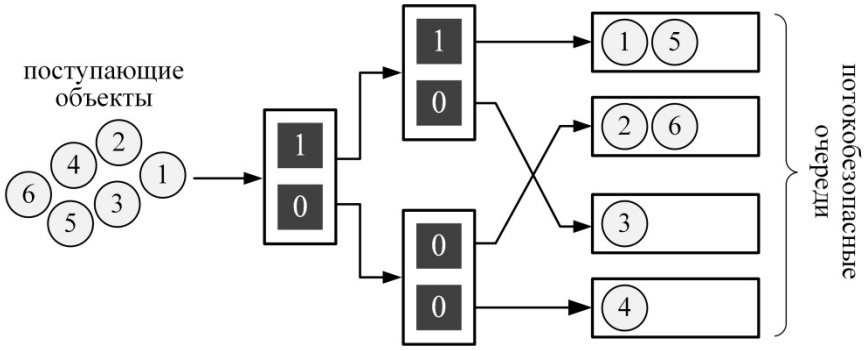

Рис. 2. Потокобезопасный пул на основе распределяющего дерева ($h = 2$)

Для практических целей последовательный порядок распределения потоков по листьям, как правило, не требуется [17, 18], и им можно пренебречь с целью повышения пропускной способности пула. На основе данного допущения авторами предложены алгоритмы реализации потокобезопасного пула на базе распределяющего дерева. В основе алгоритмов лежат идея локализации обращений к узлам дерева и использование локальной памяти потока (thread-local storage, TLS). TLS применяется с целью сокращения накладных расходов, связанных с доступом из разных потоков к разделяемым атомарным переменным и использованием массива устранения комплементарных операций в каждом узле дерева.

2. 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. 3). Узлы каждого следующего уровня дерева содержат в два раза меньшие по размеру массивы по сравнению с предыдущим уровнем.

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.

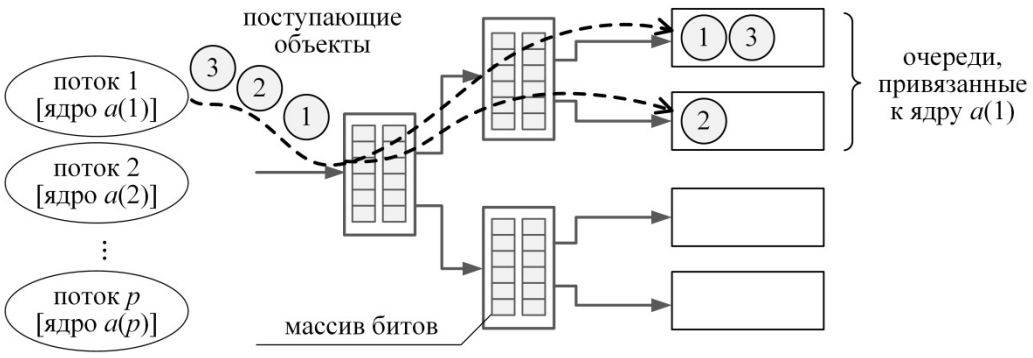

Рис. 3. Оптимизированный пул LocOptDTPool на основе распределяющего дерева

При каждом посещении потоком узла дерева в массиве атомарных битов выбирается ячейка по значению хеш-функции:

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

где $i \in \{1, 2, \ldots, p\}$ – это порядковый номер текущего потока, выданный ему при первом посещении пула, m – размер массива атомарных битов.

Рассмотрим схему распределения потокобезопасных очередей в листьях дерева между процессорными ядрами. Каждому процессорному 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 имеется поток i, привязанный 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.

При обращении к полям структур (см. раздел 3), которые включает в себя пул, необходимо учитывать проблему ложного разделения данных (false sharing), возникающую вследствие расположения переменных структур, к которым обращаются несколько потоков, в одной кэш-линии. Для решения проблемы ложного разделения данных выполняется выравнивание размера структур пула на величину кэш-линии.

Одним из основных недостатков существующей реализации пула на основе распределяющего дерева [14] является увеличение времени выполнения операций при незначительном количестве активных потоков (1–2 потока). Для решения данной проблемы в созданном пуле LocOptDTPool учитывается текущее количество активных потоков в пуле. 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 для потоков-производителей и потоков-потребителей соответственно. Данная возможность позволяет повысить пропускную способность пула при низкой нагрузке.

Таким образом, описанный потокобезопасный пул LocOptDTPool (листинг 1) 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, а также методы push и pop для помещения и извлечения объектов из пула соответственно.

Листинг 1

Структура потокобезопасного пула LocOptDTPool

```
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   }
```

Потокобезопасные очереди *queues* могут быть реализованы различным образом; в данной работе применяется реализация очередей без использования блокировок из библиотеки boost [19]. Данный выбор объясняется приемлемой для практики пропускной способностью очереди. Атомарные переменные *prod_num* и *cons_num* используются для подсчета максимального количества уникальных потоков, обратившихся к пулу за время его использования. Каждый раз, когда потоки впервые обращаются к пулу для выполнения операций добавления или извлечения, соответствующий счетчик увеличивается на единицу. Счетчик *prod_num* отражает количество потоков-производителей, а *cons_num* – количество потоков-потребителей. Учет количества потоков в пуле необходим для присвоения потокам уникаль-

ных идентификаторов, которые в дальнейшем используются для расчета хеш-функции (1) при доступе к массивам битов в узлах распределяющего дерева.

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

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

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

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

1. Привязка потока с помощью менеджера привязки *af_mgr* и увеличение счетчика *cons_num*, если этого не было сделано ранее.

2. Выбор очереди для извлечения по формуле

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

где α – коэффициент сдвига, возвращаемый методом get_queue_offset (листинг 2).

3. Если очередь, выбранная по формуле (3), пуста, то элемент извлекается из первой следующей за ней непустой очереди. Такой метод используется в других реализациях пулов [17, 20]. В случае успешного выполнения операции метод pop возвращает извлеченный объект, а в случае неудачи выполняется повторный вызов метода.

Менеджер *af_mgr* реализует последовательную привязку потоков к ядрам (листинг 2).

Листинг 2

Класс, реализующий привязку потоков к процессорным ядрам

```
1   class AffinityManager {
2   thread_local int core
3   thread_local int queue_offset
4   AtomicInt next_core
5   AtomicInt next_offset
6   set_core()
7   get_core()
8   get_queue_offset()
9   }
```

При вызове метода set_core выполняется привязка потока, вызвавшего данный метод, к процессорному ядру. Выбор номера ядра происходит при помощи счетчика *next_core*, который инкрементируется при каждом успешном вызове метода set_core. При достижении данным счетчиком значения, равного максимально возможному числу ядер в системе, он сбрасывается в 0, обеспечивая тем самым последовательную привязку потоков для равномерной загрузки процессорных ядер. Кроме того, при вызове метода set_core происходит определение коэффициента α для текущего потока. Его значение сохраняется в переменной *queue_offset* и затем используется в методе push пула (см. листинг 1) при выборе очереди, в которую помещается объект. Выбор сдвига основывается на значении переменной *next_offset*, которая инкрементируется на единицу каждый раз при достижении переменной next_core значения, равного максимально возможному числу ядер в системе. Таким образом, метод get_core возвращает номер процессорного ядра, к которому привязан текущий поток, а метод get_queue_offset – сдвиг при выборе очереди.

Каждый узел распределяющего дерева *tree* (листинг 3) состоит из константных переменных *index* и *level*, которые хранят, соответственно, порядковый номер текущего узла и номер уровня дерева, на котором данный узел находится. Метод traverse вызывается текущим потоком в каждом посещенном узле и используется для того, чтобы посетить следующий узел-потомок. При вызове данного метода происходит переключение бита *bits*[*level*][*index*] в массиве *bits*, который передается в эту функцию в качестве аргумента. При этом первоначальное значение бита указывает на выход узла, через который

впоследствии обращается текущий поток, для того чтобы попасть в следующий дочерний узел. Метод traverse, вызванный из корневого узла, таким образом, возвращает в качестве результата индекс того листа дерева (очереди), который был посещен текущим потоком.

Листинг 3

Узел распределяющего дерева

```
1    class Node {
2    int index, level
3    Node children[2]
4    int traverse(BitArray bits)
9    }
```

Структура BitArray (см. листинг 4) описывает массив атомарных битов *bits_array*, биты в котором переключаются (flip) потоками при посещении узлов распределяющего дерева.

Листинг 4

Массив атомарных битов

```
1    class BitArray {
2    Bit bits_array[n][m]
3    int flip(tree_level, node_index) {
4    return bits_array[tree_level][node_index].flip()
5    }
6    }
```

Реализация переключаемого атомарного бита представлена в листинге 5. Переключение бита (функция flip) основано на атомарной операции atomic_xor, которая заменяет текущее значение бита результатом логической операции XOR между значением этого бита и единицей.

Листинг 5

Переключаемый атомарный бит

```
1    class Bit {
2    AtomicInt bit
3    int flip() {
4    return bit.atomic_xor(1)
5    }
6    }
```

3. 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 [13–16].

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 (листинг 6).

Листинг 6

Массив битов

```
1    class BitArray {
2    bool bits[n][m]
3    int flip(tree_level, node_index) {
4    bits[tree_level][node_index] = bits[tree_level][node_index] XOR 1
     return bits[tree_level][node_index]
5    }
6    }
```

Поскольку потоки не имеют доступа к состояниям битов других потоков в узлах деревьев, распределяющее дерево может не обеспечивать равномерное распределение загрузки между очередями. В этом случае возможна ситуация, при которой в определённый момент времени большинство потоков в пуле одновременно обращаются к одной очереди. Таким образом, количество потоков, обращающихся к одному листу дерева, может превысить число потоков, добавляющих (удаляющих) элементы из очередей других узлов. Для решения данной проблемы предлагается следующий эвристический алгоритм инициализации элементов бинарных массивов в узлах дерева (см. листинг 7).

При первом посещении корневого узла потокам выдается целочисленный идентификатор (*id*). В соответствии с идентификатором потоки равномерно распределяются по узлам дерева. Принцип начального распределения основан на представлении идентификатора потока в двоичном виде (рис. 4). Каждый разряд в двоичном преставлении идентификатора является соответствующим уровнем дерева, а значение данного разряда указывает на начальное состояние всех битов в узлах дерева на данном уровне.

Листинг 7

Алгоритм инициализации битов распределяющего дерева

```
1    for (level = 0; level < max_levels; ++level) {
2    int max_range = pow(2, level)
3    for (node = 0; node < max_range; ++node) {
4    if (level > 0) {
5    array[level][node] = (id >> (level – 1)) & 1
6    } else {
7    array[level][node] = id % 2
8    }
9    }
10   }
```

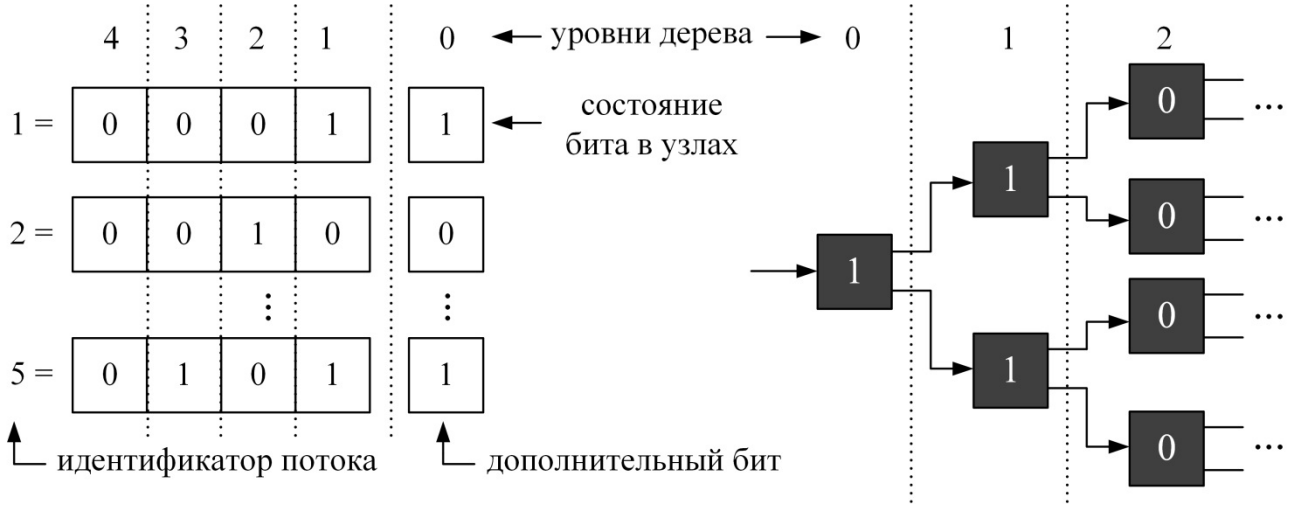

Рис. 4. Распределение потоков по узлам дерева в пуле TLSDTPool

Бит 0 или 1 в узлах дерева означает соответственно правый или левый узел-потомок, к которому обращается поток после прохождения данного узла. Проходя через каждый узел дерева, поток инвертирует соответствующий локальный бит в этом узле.

Для более равномерного распределения потоков по дереву вводится также дополнительный бит, (листинг 7, строка 7), который размещается в корне дерева. Аналогично другим битам с каждым увеличением идентификатора он переключается на противоположный (0, 1, 0, 1 и т.д.).

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

4. Результаты экспериментов

Организация экспериментов

Моделирование пулов LocOptDTPool и TLSDTPool проводилось на узле вычислительного кластера Jet Центра параллельных вычислительных технологий Федерального государственного бюджетного образовательного учреждения высшего образования «Сибирский государственный университет телекоммуникаций и информатики». Узел кластера укомплектован двумя 4-ядерными процессорами Intel Xeon E5420 (2,5 GHz; Intel-64). Тестовая программа была разработана на языке программирования C++ и скомпилирована с использованием компилятора GCC 4.8.2. В качестве элементов, помещаемых (извлекаемых) в пул, использовались переменные целочисленного типа.

Под количеством p потоков подразумевается число потоков, помещающих элементы и извлекающих объекты из пула. В качестве показателя эффективности пула использовалась пропускная способность $b = N / t$ пула, где N – суммарное число выполненных операций добавления (извлечения), а t – время моделирования. Пропускная способность показывает, сколько операций было выполнено за 1 с. Реализовано сравнение эффективности пула при использовании различных типов очередей (с блокировками и без блокировок) в листьях дерева. Сравнение используемых в пуле очередей объясняется тем, что от выбора очередей во многом зависит пропускная способность пула; такой подход к моделированию применялся в других работах [16]. Также для сравнения представлены результаты моделирования пула на основе одной неблокируемой очереди *Lockfree queue* из библиотеки boost [19]. Для каждого пула проводились две серии экспериментов: для числа потоков $p = 1, 2, \ldots, 8$, не превышающего число процессорных ядер вычислительного узла, и для большого количества потоков $p = 10, 20, \ldots, 200$.

Пул LocOptDTPool на основе массивов атомарных битов

Результаты тестирования пропускной способности реализованного пула с использованием массивов атомарных битов в узлах дерева показаны на рис. 5.

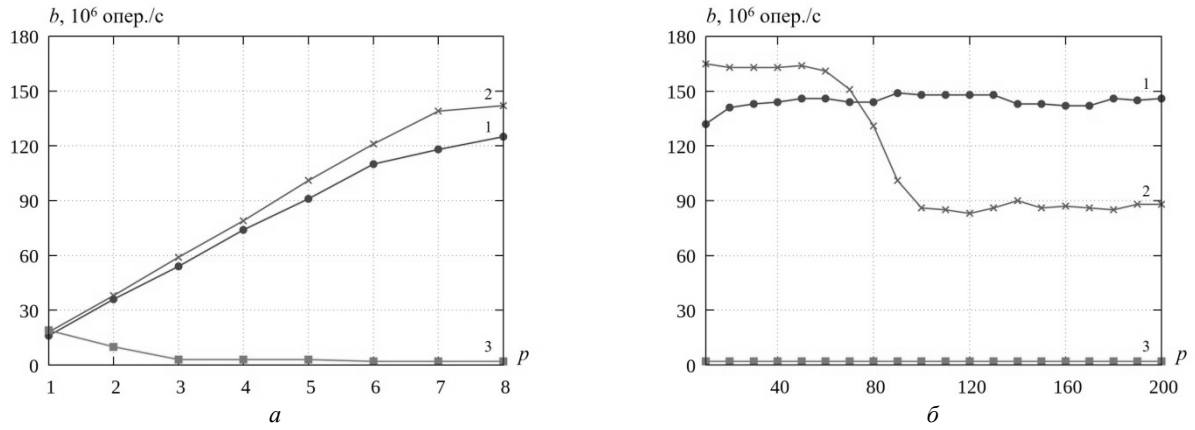

Рис. 5. Пропускная способность пула LocOptDTPool: *а* – число потоков не превышает количество процессорных ядер; *б* – число потоков превышает количество процессорных ядер.
1 – LocOptDTPool, неблокируемые очереди Lockfree queue из библиотеки boost; *2* – LocOptDTPool, блокируемые очереди на основе PThreads mutex; *3* – очередь Lockfree queue без использования блокировок из библиотеки boost

Реализованная структура данных хорошо масштабируется для большого количества потоков и демонстрирует рост пропускной способности по мере достижения числа потоков, равного количеству процессорных ядер. Максимальная пропускная способность, равная 170 млн опер./с, была получена при количестве потоков, равном количеству ядер процессора или незначительно его превышающем.

Пул TLSDTPool на основе Thread-local storage

На рис. 6 представлены результаты моделирования пропускной способности пула TLSDTPool с использованием локально-поточных битов.

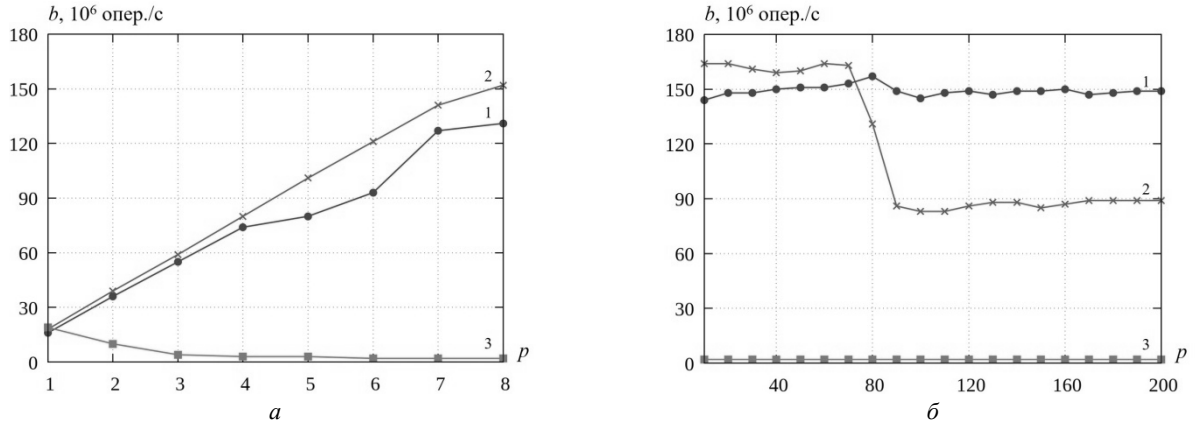

Рис. 6. Пропускная способность пула TLSDTPool: *а* – число потоков не превышает количество процессорных ядер; *б* – число потоков превышает количество процессорных ядер. *1* – TLSDTPool, неблокируемые очереди *Lockfree queue* из библиотеки boost; *2* – TLSDTPool, блокируемые очереди на основе PThreads mutex; *3* – очередь Lockfree queue без использования блокировок из библиотеки boost

Пропускная способность пула LocOptDTPool на всем диапазоне числа потоков соответствует пропускной способности пула TLSDTPool с применением локально-поточных битов. При этом также была достигнута максимальная пропускная способность, равная 170 млн опер./с, при количестве потоков, равном числу процессорных ядер или незначительно его превосходящем.

В разделе 3 была рассмотрена ситуация, когда при использовании поточно-локальных переменных число потоков, одновременно обращающихся к одному листу дерева, может значительно превосходить число потоков, выполняющих операции с другими листьями дерева. Однако в ходе выполнения экспериментов такой случай не был зафиксирован и снижения пропускной способности пула не наблюдалось. Тем не менее при построении пулов необходимо учитывать, что с увеличением количества уровней распределяющего дерева вероятность появления «худшего случая» уменьшается (при этом возрастают затраты по памяти).

При большом количестве потоков применение неблокируемых очередей Lockfree queue в пулах LocOptDTPool и TLSDTPool обеспечивает большую пропускную способность, по сравнению с блокируемыми потокобезопасными очередями (см. рис. 5, *б*; 6, *б*). Во всех случаях эффективность отдельной потокобезопасной очереди Lockfree queue значительно уступает эффективности разработанных пулов (рис. 5, 6).

Заключение

Разработаны алгоритмы реализации масштабируемых потокобезопасных пулов на основе распределяющих деревьев без использования блокировок. Суть оптимизаций заключается в локализации обращений потоков к разделяемым областям памяти с целью максимизации пропускной способности пула.

Разработанные пулы могут применяться при реализации модели производитель – потребитель в многопоточных программах с постоянным числом активных потоков, где требуются высокая пропускная способность и быстрый возврат потоков из структуры с целью минимизации времени выполнения операций. 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 в системе. Увеличение размеров дерева в пуле не снижает пропускную способ-

ность пула. В качестве структур данных в листьях дерева для хранения объектов пула рекомендуется использовать потокобезопасные очереди без использования блокировок.

ЛИТЕРАТУРА

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. Morgan Kaufmann, NY, USA, 2008. P. 529.
3. Anderson T.E. The performance of Spin Lock Alternatives of Shared-Memory Multiprocessors // IEEE Transactions on Parallel and Distributed Systems. 1990. P. 6–16.
4. Mellor-Crummey J.M., Scott M.L. Synchronization without Contention // Proceedings of the 4th International Conference on Architecture Support for Programming Languages and Operating Systems. 1991.
5. Rudolph L., Slivkin M., Upfal E. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines // Proceeding of the 3rd ACM Symposium on Parallel Algorithms and Architectures. 1991. P. 237–245.
6. Blumofe R.D., Leiserson C.E. Scheduling Multithreaded Computations by Work Stealing // Proceeding of the 35th Symposium on Foundations of Computer Science. 1994. P. 365–368.
7. Hendler D., Shavit N., Yerushalmi L.A scalable lock-free stack algorithm // Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures. ACM, 2004. P. 206–215.
8. Moir M. et al. Using elimination to implement scalable and lock-free FIFO queues // Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures. ACM, 2005. P. 253–262.
9. Afek Y., Hakimi M., Morrison A. Fast and scalable rendezvousing // Distributed computing. 2013. V. 26, No. 4. P. 243–269.
10. Shavit N., Touitou D. Elimination trees and the construction of pools and stacks: preliminary version // Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures. ACM, 1995. P. 54–63.
11. Calciu I., Gottschlich J.E., Herlihy M. Using elimination and delegation to implement a scalable NUMA-friendly stack // Proc. Usenix Workshop on Hot Topics in Parallelism (HotPar). 2013.
12. Lozi J.P. et al. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications // USENIX Annual Technical Conference. 2012. P. 65–76.
13. Shavit N., Zemach A. Diffracting trees // ACM Transactions on Computer Systems (TOCS). 1996. V. 14, No. 4. P. 385–428.
14. 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.
15. Della-Libera G., Shavit N. Reactive diffracting trees // Journal of Parallel and Distributed Computing. 2000. V. 60. P. 853–890.
16. Ha P.H., Papatriantafilou M., Tsigas P. Self-tuning reactive distributed trees for counting and balancing // Principles of Distributed Systems: 8th International Conference, OPODIS. 2004. P. 213–228.
17. Shavit N. Data Structures in the Multicore Age // Communications of the ACM. 2011. V. 54, No. 3. P. 76–84.
18. Shavit N., Moir M. Concurrent Data Structures // Handbook of Data Structures and Applications. 2007. P. 47–14.
19. Blechmann T. Chapter 19. Boost.Lockfree. URL: http://www.boost.org/doc/libs/1_61_0/doc/html/lockfree.html (дата обращения: 14.09.2016).
20. Chase D., Lev Y. Dynamic circular work-stealing deque // Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures. 2005. P. 21–28.

Аненков Александр Дмитриевич. E-mail: alex.anenkov@outlook.com
Пазников Алексей Александрович, канд. техн. наук. E-mail: apaznikov@gmail.com
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

Поступила в редакцию 31 октября 2016 г.

Anenkov Alexandr D., Paznikov Alexey A. (Siberian State University of Telecommunications and Information Sciences, Russian Federation).

Algorithms of optimization of scalable thread-safe pool based on diffracting trees for multicore computing systems.

Key words: multithreaded programming; diffracting trees; lock-free data structures; scalability; thread-safe pool.

DOI: 10.17223/19988605/39/10

Most of all modern multicore computer systems (CS) are large-scale, hierarchically organized and include multiple architectures. Program execution time on these systems strongly depends on the efficiency of parallel thread synchronization methods. With increasing number of CPU cores in computer systems the problem of scalable concurrent data structures development dramatically arises. Such data structures must ensure scalable access with increasing number of parallel threads and workload.

Thread-safe pool is one of the most widely used concurrent data structures. Pool is an unordered set of objects, which supports operations for insertion (push) and removal (pop) of the objects. Pools are widely used while producer-consumer model implementation in multithreaded programs. In this model some producer threads generate the objects followed by their utilization by consumer threads.

Simple concurrent pools implementations based on concurrent queues (both lockable and lock-free) poorly scale for large number of threads and high pool access rate. There are some methods of access contention reduction like elimination arrays or delegation of pool operation to the remote CPU cores. Nonetheless these methods lead to the bottlenecks on certain elements and severe throughput reduc-

tion in the case of large number of threads and high intensity of pool operations. Workpile and work-stealing methods ensure predictable operations time performance but they are not effective on low frequency of pool treatment.

Diffraction trees is one of the perspective approaches for access contention reduction in thread-safe data structures. There are some works which propose pool implementations based on diffraction trees with using of elimination arrays. The main drawback of these implementations are high overheads on active waiting and atomic variable states synchronization on each of the tree node. This fact severely increases the complexity of tree traversal from the root to the leaves. Thus the tree efficiency decreases with increasing of its size. Some pool implementations severely violate FIFO/LIFO order while operations performance. The other drawback is the optimization of variable parameters of these structures.

In this paper we propose the novel approach for scalable concurrent lock-free pool implementation on the basis of diffraction trees. The approach is based on localization of tree nodes access and Thread-local storage (TLS) utilization. This approach increases throughput at high and low pool load and minimizes the operation latency. The concurrent pools LocOptDTPool and TLSDTPool were developed by the authors on the basis of proposed approach. The pools contain the diffraction tree, atomic bool arrays, corresponding to the tree nodes, array of concurrent queues, corresponding to the tree leaves, the CPU core affinity manager, thread number counters and push (insert) and pop (remove) methods. The concurrent queues in the tree leaves may be implemented in different ways. In the current implementation of the pools we used lock-free concurrent queues from boost library.

In the LocOptDTPool each node of diffraction tree contains two bool atomic arrays which size is no more than thread number. Each thread accesses to the corresponding array's element in order to localize tree nodes' atomic bit access. Moreover, compared with the elimination array methods, new approach allows to reduce the overheads, arising from additional (elimination) array's elements and active waiting for the pair thread. For the minimization of operation latency in case of low workload LocOptDTPool implements the counting of the current number of active threads in the pool. If the current workload (thread number) is low, then objects distribution among the queues doesn't lead to the significant increase of pool throughput. In this case the only queue is used for element storage in the pool. This algorithm increases the pool throughput in case of low workload.

The main idea of TLSDTPool is the allocation of the arrays bits in the tree nodes in the Thread-local storage (TLS). The tree nodes contain ordinary bool arrays. This approach allows to avoid the expensive atomic operations and reduces access contention for the shared bits in the tree nodes. For the queues load uniformity in the TLSDTPool we proposed the algorithm of tree nodes initialization based on the binary representation of thread identifiers. This algorithm submits the initial state of the array's bits according to the thread identifier. That scheme minimizes the impact of the "worst case", rising from the imbalance of the queues workload.

The experiments for the developed pools on cluster computer systems has shown, that LocOptDTPool scales well for high number of threads and shows an increase of throughput until the number of threads is equal to the number of CPU cores. Throughput of LocOptDTPool for the entire range of thread number corresponds the results of TLSDTPool based on Thread-local storage. At the same time the maximum throughput was achieved for thread number equals to processors core number or slightly more. Thread-safe lock-free queues are recommended as the objects storage in tree leaves. Thanks to tree initialization algorithm the concurrent queues were balanced well and possible "worst case" didn't significantly effect on pool efficiency.

Designed pools can be used in the producer-consumer model in multithreading programs with constant number of active threads, where high throughput and low latency is highly desirable. The pools provides high scalability at multithreading programs execution, in compared with the similar pool implementations on the basis of diffraction trees. The maximum algorithm efficiency is achieved at the thread number, equals to total processor cores number. Tree size increasing doesn't lead to throughput reduction.

REFERENCES

1. Khoroshevsky, V.G. (2010) Raspredelennye vychislitel'nye sistemy s programmiruemoy strukturoy [Distributed programmable structure computer systems]. *Vestnik SibGUTI.* 2(10). pp. 3–41. (In Russian).
2. Herlihy, M. & Shavit, N. (2008) *The Art of Multiprocessor Programming.* New York: Morgan Kaufmann.
3. Anderson, T.E. (1990) The performance of Spin Lock Alternatives of Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems.* pp. 6–16. DOI: 10.1109/71.80120
4. Mellor-Crummey, J.M. & Scott, M.L. (1991) Synchronization without Contention. *Proc. of the 4th International Conference on Architecture Support for Programming Languages and Operating Systems*. DOI: 10.1145/106975.106999
5. Rudolph, L., Slivkin, M. & Upfal, E. (1991) A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. *Proc. of the 3rd ACM Symposium on Parallel Algorithms and Architectures.* pp. 237–245. DOI: 10.1145/113379.113401
6. Blumofe, R.D. & Leiserson, C.E. (1994) Scheduling Multithreaded Computations by Work Stealing. *Proc. of the 35th Symposium on Foundations of Computer Science.* pp. 365–368. DOI: 10.1145/324133.324234
7. Hendler, D., Shavit, N. & Yerushalmi, L. (2004) A scalable lock-free stack algorithm. *Proc. of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures. ACM.* pp. 206–215. DOI: 10.1145/1007912.1007944
8. Moir, M. et al. (2005) Using elimination to implement scalable and lock-free FIFO queues. *Proc. of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures. ACM.* pp. 253–262. DOI: 10.1145/1073970.1074013
9. Afek, Y., Hakimi, M. & Morrison, A. (2013) Fast and scalable rendezvousing. *Distributed computing.* 26(4). pp. 243–269. DOI: 10.1007/978-3-642-24100-0_2
10. Shavit, N. & Touitou, D. (1995) Elimination trees and the construction of pools and stacks: preliminary version. *Proc. of the seventh annual ACM symposium on Parallel algorithms and architectures. ACM.* pp. 54–63. DOI: 10.1145/215399.215419
11. Calciu, I., Gottschlich, J.E. & Herlihy, M. (2013) Using elimination and delegation to implement a scalable NUMA-friendly stack. *Proc. Usenix Workshop on Hot Topics in Parallelism* (*HotPar*).
12. Lozi, J.P. et al. (2012) Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. *USENIX Annual Technical Conference.* pp. 65–76. DOI: 10.1006/jpdc.1994.1056
13. Shavit, N. & Zemach, A. (1996) Diffracting trees. *ACM Transactions on Computer Systems (TOCS).* 14(4). pp. 385–428. DOI: 10.1145/181014.181326

14. Afek, Y., Korland, G., Natanzon, M. & Shavit, N. (2010) Scalable Producer-Consumer Pools based on Elimination-Diffraction Trees. *European Conference on Parallel Processing,* pp. 151-162. DOI: 10.1007/978-3-642-15291-7_16
15. Della-Libera, G. & Shavit, N. (2000) Reactive diffracting trees. *Journal of Parallel and Distributed Computing.* 60. pp. 853–890. DOI: 10.1145/258492.258495
16. Ha, P.H., Papatriantafilou, M. & Tsigas, P. (2004) Self-tuning reactive distributed trees for counting and balancing. *Principles of Distributed Systems.* 8th International Conference, OPODIS. pp. 213–228.
17. Shavit, N. (2011) Data Structures in the Multicore Age. *Communications of the ACM.* 54(3). pp. 76–84. DOI: 10.1145/1897852.1897873
18. Shavit, N. & Moir, M. (2007) Concurrent Data Structures. In: Mehta, D.P. & Sahni, S. (eds) *Handbook of Data Structures and Applications*. Boca Raton London New York Washington, D.C.: Chapman & Hall/CRC. pp. 47–14.
19. Blechmann, T. (2016) *Chapter 19. Boost.Lockfree.* [Online] Available from: http://www.boost.org/doc/libs/1_61_0/doc/html/lockfree.html. (Accessed: 14th September 2016).
20. Chase, D. & Lev, Y. (2005) Dynamic circular work-stealing deque. *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures.* pp. 21–28. DOI: 10.1145/1073970.1073974