АЛГОРИТМ ДИНАМИЧЕСКОГО МАСШТАБИРУЕМОГО УВЕЛИЧЕНИЯ РАЗМЕРОВ ХЭШ-ТАБЛИЦЫ В ОБЩЕЙ ПАМЯТИ С ПОЭЛЕМЕНТНЫМ ПЕРЕМЕЩЕНИЕМ ДАННЫХ.

А.А. Малахов

Пока ведутся споры, закончился уже "бесплатный обед" [1] или нет [2], мы хотим сделать обед как минимум дешевле.

Для того чтобы использовать вычислительные ресурсы многоядерных процессоров наиболее эффективно [3], программа должна уметь синхронизовать процесс и результаты вычисления между потоками инструкций с общей памятью. Однако производить синхронизацию следует как можно более аккуратно и редко, чтобы потоки не мешали друг другу. Например, структуры данных в общей памяти (контейнеры в терминах библиотеки C++) должны быть оптимизированы для минимизации накладных расходов на синхронизацию и для поддержания хорошей масштабируемости при возрастающем числе процессоров.

Одна из базовых структур данных - это ассоциативный массив [4], который позволяет хранить пары ключ-значение и, как правило, поддерживает операции вставки, поиска и удаления. А наиболее эффективной разновидностью ассоциативного массива является неупорядоченный ассоциативный массив или "хэш-таблица", поскольку среднее время доступа к элементу можно оценить как O(1) (в отличие от O(log n) для упорядоченных массивов).

Суть организации данных в хэш-таблицах с прямой адресацией заключается в следующем. Существует некий массив, каждый элемент которого может содержать некое множество пар ключ-значение. Назовем такой элемент бакетом (англ. "корзинка"), а массив хэш-таблицы - соответственно, массивом бакетов. Для каждого ключа вычисляется значение хэш-функции, которое используется для определения индекса в массиве бакетов. Индексом, как правило, является остаток от деления хэш-значения на размер этого массива:

$$i = h \bmod s; \qquad [Ф1]$$

Где, i - индекс; h - значение хэш-функции для данного ключа; s - размер массива.

При размере массива бакетов существенно большем количества хранимых пар, можно говорить о неэффективном использовании памяти, а в обратном случае (массив меньше количества пар) увеличивается время доступа к элементу из-за необходимости поиска нужного ключа среди множества других пар с тем же индексом в массиве. Поэтому, для хэш-таблиц важно поддерживать динамическое изменение размера с целью оптимизировать доступ к её элементам.

Процесс увеличения количества бакетов представляет особенную сложность при использовании таблицы в многопоточной программе. Суть проблемы заключается в том, что даже если, например, заполнение массива и может быть организовано на нескольких процессорах, то выделение под него памяти, организация заполнения, да и запись данных в конкретную ячейку памяти - это задачи для отдельного потока инструкций. Проблема осложняется тем, что изменение размера массива бакетов ведёт к перерасчёту индексов всех пар ключ-значение (согласно Ф1), и появляется необходимость переместить часть элементов по новым адресам. Это порождает сериализацию программы, и даже если организовать перемещение несколькими потоками, ни один другой поток не может производить иные операции над таблицей (как минимум в полном объёме и по обычной схеме). Таким образом, теряется ценность хэш-таблицы как контейнера с константным и предсказуемым временем доступа.

Предлагаемый алгоритм позволяет решить поставленные задачи, не прибегая к продолжительной блокировке доступа к данным и распределяя стоимость перемещения по всем последующим обращениям к таблице, тем самым, делая время доступа равномерным и предсказуемым.

Динамическое увеличение размера

Для ясности положим, что начальный размер таблицы равен степени двойки. Это также позволяет использовать для вычисления индекса в массиве более дешёвую побитовую операцию "логическое И" (относительно деления в Ф1):

$$i = h \,\&\, (s - 1) \qquad [Ф2]$$

Также это позволяет применить в качестве массива хэш-таблицы динамический массив, который для увеличения размера не освобождает, а продолжает использовать ранее выделенные элементы, но зато дополнительно выделяет количество элементов, равное сумме уже существующих. Это известный приём [5,6,7], который, в частности, применяется в контейнере concurrent_vector библиотеки Intel(R) Threading Building Blocks

[8]. Таким образом, массив бакетов состоит из сегментов, размер которых составляет последовательность: 2, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 и т.д.

Из этого следует, что каждый отдельно взятый сегмент содержит количество бакетов равное количеству бакетов во всех предыдущих сегментах (исключая первый сегмент). И если их мысленно расположить в две строки, то каждому бакету из этого сегмента будет соответствовать бакет входящий в один из предыдущих (меньших) сегментов, смотрите рисунок 1:

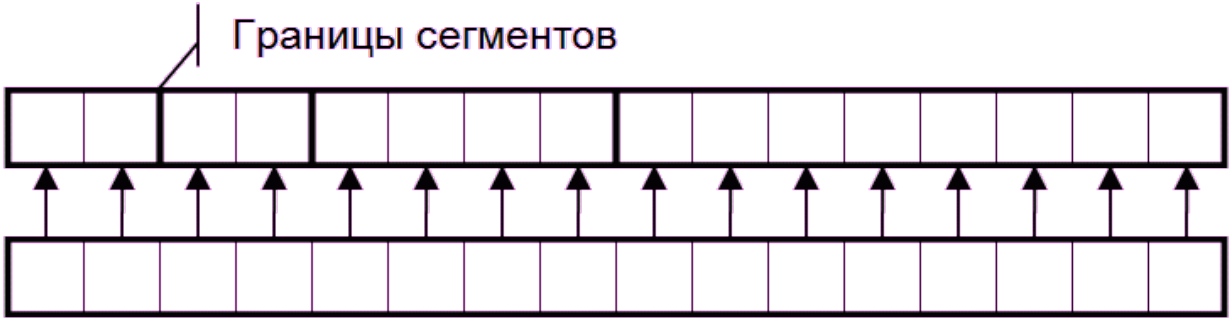

Рис. 1 Сегменты массива бакетов и их связи

В этом случае, каждый бакет имеет связанный бакет в предыдущих сегментах, индекс которого можно

[Ф3]

рассчитать по формуле:

Здесь, p - индекс предыдущего связанного бакета. Все операции являются целочисленными с округлением в меньшую сторону, поэтому log2 представляет собой, по сути, индекс старшего значащего бита. И, следовательно, операция, записанная этой формулой, суть обнуление старшего значащего бита.

Перемещение данных

Увеличение размера массива бакетов производится независимо от перераспределения пар ключ-значение и завершается публикацией нового размера массива. Каждый новый бакет должен быть отмечен специальным признаком (флагом). Когда последующие операции поиска, использовавшие новый размер для расчёта индекса (см. Ф2), попадают на бакет, отмеченный как новый, они будут рекурсивно обращаться к предыдущим связанным бакетам до тех пор, пока не будет найден некий бакет, не имеющей такой отметки. Желательно (хотя и не обязательно), чтобы такие операции над таблицей перемещали пары ключ-значение в соответствующие новые бакеты, снимая этот признак (перемещение производится только для тех пар, индекс которых соответствует индексу новых бакетов).

Возможно несколько вариантов реализации алгоритма. Например, перемещение может происходить только между двумя бакетами, тогда может потребоваться рекурсивное перемещение данных. А можно перемещать данные за раз во все новые бакеты (см. рис. 2); в этом случае необходим такой дополнительный признак в каждом бакете, как количество инициализированных сегментов.

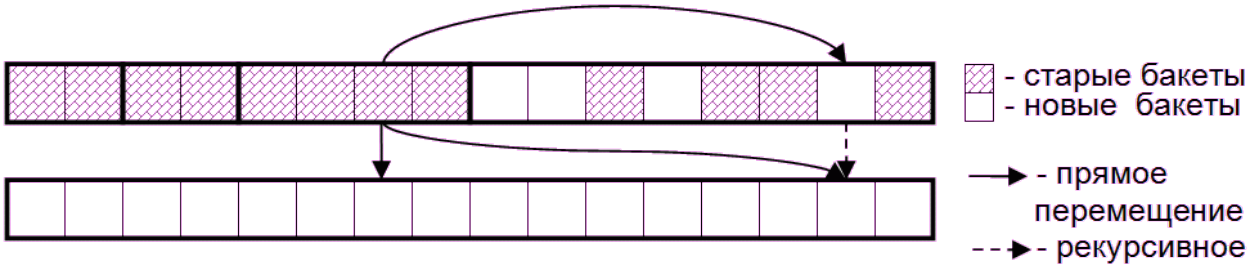

Рис. 2 Перемещение данных

В любом случае, подобно операциям вставки и удаления элемента, операция перемещения также требует синхронизации доступа к элементу между потоками. Однако это уже не является препятствием для работы других потоков.

Корректность операций поиска

Но в отсутствии глобальной синхронизации при такой схеме перемещения пар данных возникает новая проблема. Если ключ не найден в данном бакете, возможно он просто перемещён другим потоком. Поэтому,

чтобы однозначно определить наличие данного ключа, необходимо заново рассчитать индекс бакета для этого ключа, используя свежее значение размера массива, которое может измениться параллельно операции поиска. В случае если индексы различны и бакет с новым индексом не помечен как новый, то есть вероятность, что искомая пара была перемещена и следует повторить поиск по новому адресу. Таким образом, отпадает необходимость в дополнительной явной внешней синхронизации и достаточно описанной "пассивной" синхронизации с использованием только операций чтения, что не мешает масштабируемости алгоритма.

Заключение

Алгоритм был реализован в контейнере concurrent_hash_map входящего в библиотеку Intel(R) Threading Building Blocks [8] и показал значительный прирост производительности контрольных приложений по сравнению с предыдущей реализацией контейнера.

ЛИТЕРАТУРА:

1. Sutter, H. 2005. "The free lunch is over: A fundamental turn toward concurrency in software", Dr. Dobb's Journal, 30(3), http://www.gotw.ca/publications/concurrency-ddj.htm
2. Wrinn, M. 2008. "Is the Free Lunch Really Over? Scalability in Many-core Systems" http://www.ddj.com/go-parallel/article/showArticle.jhtml?articleID=212700023
3. B. Nichols, D. Buttlar, and J. Farrell, "Pthreads Programming", O'Reilly & Associates Inc., ISBN: 1-56592-115-1, стр. 25
4. http://ru.wikipedia.org/wiki/Ассоциативный_массив
5. P.-A. Larson, "Dynamic hash tables", Communications of the ACM, 31, (4), (1988).
6. Griswold, W.G. and Townsend, G.M., "The Design and Implementation of Dynamic Hashing for Sets and Tables in Icon", Software-Practice and Experience, vol. 23(4), 351-367 (April 1993)
7. Shalev, O. and Shavit, N., "Split-Ordered Lists: Lock-Free Extensible Hash Tables", Journal of the ACM, Vol. 53, No. 3, May 2006, pp. 379-405.
8. Intel(R) Threading Building Blocks, http://www.threadingbuildingblocks.org