Тема работы:

Распределенная очередь без использования блокировок в модели удаленного доступа к памяти

Состав коллектива:

1. Пазников Алексей Александрович, к.т.н., с.н.с. СПбГЭТУ «ЛЭТИ», руководитель
2. Бураченко Александр, магистрант. СПбГЭТУ «ЛЭТИ», исполнитель

Информация о гранте:

РНФ, проект № 22-21-00686 «Алгоритмы и программные средства оптимизации выполнения параллельных программ в модели удаленного доступа к памяти», руководитель – Пазников А.А., 2022-2023

Научное содержание работы:

1. Постановка задачи:

Целью данной работы является разработка неблокируемых распределенных структур данных в модели удаленного доступа к памяти (MPI RMA) и исследование их эффективности. Конкретно, задачи, которые авторы ставят перед собой, включают:

1. Разработка неблокируемой очереди в модели MPI RMA.
2. Описание алгоритмов для выполнения основных операций над неблокируемой очередью, таких как вставка и извлечение элементов.
3. Исследование эффективности неблокируемой структуры данных и сравнение её с блокируемыми аналогами.
4. Проведение экспериментов для оценки производительности предложенной структуры данных и сравнения её с альтернативными решениями.

Ожидается, что разработка неблокируемых распределенных структур данных в модели MPI RMA и их сравнение с блокируемыми аналогами позволят выявить преимущества и недостатки такого подхода. Это может привести к повышению эффективности и упрощению разработки параллельных программ для распределенных вычислительных систем, использующих MPI RMA.

2. Современное состояние проблемы:

Основными операциями в модели RMA являются неблокируемые MPI_Put (запись в удаленную память) и MPI_Get (чтение), и набор атомарных операций: MPI_Accumulate, MPI_Get_accumulate, MPI_Compare_and_swap и др. RMA-вызовы должны находиться внутри областей (эпох, epochs), в рамках которых выполняется синхронизация. RMA предоставляет активный и пассивный методы синхронизации. В настоящей работе применяется пассивный метод синхронизации (passive target synchronization) [1, 2]: процесс открывает эпоху, затем выполняет RMA-операции для доступа к зарегистрированным сегментам памяти (окнам, window) других процессов. Таким образом, RMA-операции выполняются в одностороннем порядке, без явного участия других процессов, и сопровождаются меньшими накладными расходами, чем при активной синхронизации [5].

В рамках моделей RMA и PGAS имеет место задача обеспечения масштабируемого доступа к структурам данных, которые распределены между узлами ВС. Данная задача не является новой в параллельном программировании. Так для систем с общей памятью имеет место необходимость в синхронизации потоков, обращающихся к разделяемым (concurrent, thread-

safe) структурам данных. Такие структуры должны обеспечивать корректный (как правило линеаризуемый, linearizable) доступ параллельных потоков в произвольные моменты времени [4, 6, 7]. Эффективность синхронизации существенно влияет на время выполнения программ и обуславливает гарантии прогресса выполнения.

Существующие методы синхронизации можно разделить на два класса: с применением блокировок (locks) и без блокировок (non-blocking). Блокировки обладают простой семантикой и часто достаточно эффективны, однако неблокирующий подход обеспечивает гарантии выполнения и позволяет избежать тупиковых ситуаций (deadlocks), инверсий приоритетов (priority inversion) и некоторых других проблем блокировок. Неблокирующим называется такой алгоритм, в котором задержка выполнения одного потока не останавливает прогресс выполнения программы в целом (при соблюдении ряда условий) [4]. Значительная часть работ в области разделяемых структур данных направлена на создание средств синхронизации для ВС с общей памятью. К ним относятся алгоритмы блокировки потоков [4, 9] (TTS, Backoff, CLH, MCS, Oyama, Flat Combining, RCL и др.). Хотя некоторые методы (Hierarchical Locks, Cohorting и др.) учитывают иерархические уровни системы, они не применимы в ВС с распределенной памятью. Неблокируемые структуры [4, 6, 7, 9] также разработаны для многоядерных ВС и не применимы в распределенных ВС. Структуры, оптимизированные для NUMA [10], также неприменимы в распределенных средах.

Существует, однако, ряд работ по реализации распределенных неблокирующих структур: FastQueue, CircularQueue [11] и Active Message Queue [12]. Их общая проблема – централизация данных в памяти одного процесса. В результате подхода, остальные процессы, работающие со структурой данных, постоянно обращаются к центральному процессу. Это является узким местом и может быть причиной снижения производительности очереди, особенно на большом числе процессов. В [13] рассматривается реализация распределенного неблокирующего стека на основе алгоритма Elimination-Backoff Stack. Экспериментальные оценки указывают на неплохую производительность на небольших подсистемах. Основным недостатком структуры также является её централизованность.

В данной работе исследуется возможность построения неблокируемых (lock-free) структур данных для распределенных ВС на примере очереди. Предлагается подход на основе децентрализации данных между процессами параллельной программы. Мы предполагаем, что такой подход может быть эффективен при построении и других распределенных структур данных.

1. 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.
2. 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.
3. Gerstenberger, R., Besta, M., Hoefler, T. Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided // Scientific Programming. 2014. V. 2 Issue 2. P. 75–91.
4. Herlihy M., Shavit N. The art of multiprocessor programming. Morgan Kaufmann. 2012. 537 p.
5. Schuchart, J., Niethammer, C., Gracia, J., Bosilca, G., Quo Vadis MPI RMA? Towards a More Efficient Use of MPI One-Sided Communication // arXiv: 2111.08142. 2021.
6. Mark M., Shavit N. Concurrent Data Structures. Chapman and Hall/CRC Press. 2004. 32 p.
7. Shavit N. Data structures in the multicore age // Communications of the ACM. 2011. V. 54. P. 76–84.

8. 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.
9. 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.
10. I. Calciu, J. Gottschlich, and M. Herlihy Using elimination and delegation to implement a scalable NUMA-friendly stack // 5th {USENIX} Workshop on Hot Topics in Parallelism (HotPar 13), 2013.
11. Brock B., Buluc A. BCL: A cross-platform distributed data structures library // ICPP, 2019, pp. 1–10.
12. Schuchart J., Bouteiller A., and Bosilca G. Using MPI-3 RMA for active messages // ExaMPI. 2019, pp. 47–56.
13. Diep T. D., Fürlinger K. Nonblocking data structures for distributed-memory machines: stacks as an example // 2021 29th Euromicro Int. Conf on Parallel, Distributed and Network-Based Processing (PDP). – IEEE, 2021. – С. 9-17.

3. Подробное описание работы, включая используемые алгоритмы:

Очередь – коллекция объектов, реализующая дисциплину FIFO («первым вошел – первым вышел»). Основные операции: добавление (insert, enqueue) элемента в последнюю позицию (хвост, tail, T) и извлечение (remove, dequeue) элемента из первой позиции (голова, head, H). Элементы распределенной очереди находятся в памяти процессов, выполняющихся на распределенной ВС.

Пусть имеется ВС из N ЭМ. На каждой ЭМ $i = 1, 2, \ldots, N$ имеется локальная оперативная память m_i, причем m_i Ç m_j = Æ, $\forall i, j \in \{1, 2, \ldots, N\}$. Считаем, что на каждой ЭМ i запущен процесс p_i.

Обозначим RMA-окно w, необходимое для выполнения операций. Процессы p_i, $i = 1, 2, \ldots, N$, обладают дескриптором окна w и публикуют в нем часть своей памяти m_i (назовем её m^*_i), таким образом предоставляя доступ к m^*_i остальным процессам.

Для хранения элементов очереди на каждом процессе выделена память под пул элементов (в текущей реализации – массив). Пусть e_i – пул элементов очереди в памяти m^*_i процесса p_i. Размер массива e_i фиксирован и равен K. При добавлении новых элементов процесс p_i использует ячейки из массива e_i: e_{ij}, $\forall i, j: i \in \{1, 2, \ldots, N\}$, $j \in \{1, 2, \ldots, N\}$. Тогда голова очереди H – это элемент e_{ij}, являющийся первым в очереди, а хвост очереди T – это элемент e_{ij}, являющийся последним в очереди.

Для определения местоположения элементов очереди введем понятие *ссылка на элемент* e_{ij} – пара чисел (i, j), которая однозначно идентифицирует элемент j распределенной очереди, который находится в памяти процесса i. Пустую ссылку обозначим (Æ, Æ). Каждый элемент e_{ij} характеризуется:

– Временной меткой добавления в очередь – τ_{ij}.

– Пользовательскими данными – v_{ij}.

– Состоянием c_{ij}. Принимает одно из трех значений: F, A, D. F – элемент e_{ij} свободен для использования при следующем добавлении в очередь, A – элемент занят, находится в очереди, D –удален из очереди, но еще не доступен для повторного использования. Состояние элемента циклически меняется: из F в A при добавлении, из A в D при удалении, из D в F при освобождении.

– Ссылкой на следующий элемент – n_{ij}. Если за e_{ij} следует e_{gf}, то $n_{ij} = (g, f)$. При добавлении e_{ij} в очередь $n_{ij} = (Æ, Æ)$.

– Ссылкой на себя l_{ij}. Для e_{ij}, $l_{ij} = (i, j)$. Данная ссылка необходима для возможности обновления информации об элементе.

Также каждый процесс p_i имеет в своей памяти m^*_i ссылки на голову h_i и хвост t_i, $\forall i \in \{1, 2, \ldots, N\}$ очереди, которые служат для определения текущего положения актуальных головы H и хвоста T.

Процесс p_1 дополнительно обладает ссылкой s – элемент для достижения консенсуса при добавлении первого элемента. Когда несколько процессов одновременно добавляют новый элемент в очередь, они принимают решение, чей элемент будет добавлен. Для этого применяется операция «сравнение с обменом» (Compare-And-Swap, CAS) с текущего хвоста T на новый хвост n_{ij}. Но сразу после инициализации очередь пуста: она не обладает ни головой, ни хвостом. Поэтому необходима область памяти, в которой можно было бы решить, чей элемент будет первым добавлен в очередь.

4. Полученные результаты

В рамках централизованного подхода информация о положении головы и хвоста находится в памяти главного процесса p_1 (рис. 1а). Процессы p_i при добавлении или удалении элемента обновляют ссылки t_1 или h_1 соответственно. Такой подход порождает узкое место (bottleneck), так как p_i постоянно обращаются к p_1, что ведет к снижению пропускной способности (throughput) структуры данных.

Экспериментальное исследование проводилось на кластере Информационно-вычислительного центра Новосибирского государственного университета. В экспериментах использовались 6 узлов по 2 6-ядерных процессора Intel Xeon X5670 на каждом (суммарно 72 ядра). MPI: Open MPI 4.1.0.

Был разработан синтетический тест, выполняющий $n = 10000$ операций вставки/удаления (тип операции выбирается равновероятно). Число процессов N варьировалось от 2 до 72. Измерялась пропускная способность $b = n \times p / t$, где t – время проведения эксперимента, p – количество процессов.

Реализованная распределенная неблокирующая очередь сравнивалась с двумя очередями на основе блокировок. Первая – блокирующая централизованная очередь (рис. 1а): информация о местоположении головы/хвоста находится в памяти процесса 0. Блокировка устанавливается на процесс. Перед чтением/изменением информации о голове/хвосте или элемента очереди из памяти некоторого процесса i читающий/изменяющий данные процесс захватывает блокировку процесса i; после окончания работы – освобождает. Вторая – блокирующая очередь с децентрализованным хранением положения головы/хвоста (рис. 1б). Она аналогична неблокирующей, за исключением использования блокировок. В отличие от централизованной очереди, здесь блокируется не процесс, а отдельные элементы. Перед работой с элементом или ссылкой на голову/хвост, процесс захватывает блокировку на этом элементе, после работы освобождает.

Пропускная способность разработанной очереди значительно превосходит пропускную способность блокирующих аналогов (рис. 4). Lock-free подход избавляет от накладных расходов на захват и освобождение блокировки и позволяет потоку в случае неудачи операции сразу повторить попытку, не затрачивая время на ожидание. Блокирующая децентрализованная очередь производительнее централизованной: децентрализация позволяет распределить нагрузку на процессы и повышает пропускную способность. Все рассматриваемые очереди масштабируются в умеренной степени. С ростом числа процессов

пропускная способность структур данных падает, тем не менее, в случае неблокируемой очереди, остается на достаточно высоком уровне.

Для оценки времени выполнения операций добавления и удаления, для неблокирующей очереди выполнялось измерение времени реализации операций и их ключевых этапов. Каждый процесс самостоятельно выполняет измерения с последующей отправкой результатов процессу 0, который вычисляет средние значения. Перечень измерений: 1) Общее время операции добавления (enq_all); 2) Время поиска хвоста при добавлении (enq_hop); 3. Общее время операции удаления (deq_all); 4) Время поиска головы при удалении (deq_hop); 5. Общее время алгоритма оповещения (bcast_all).

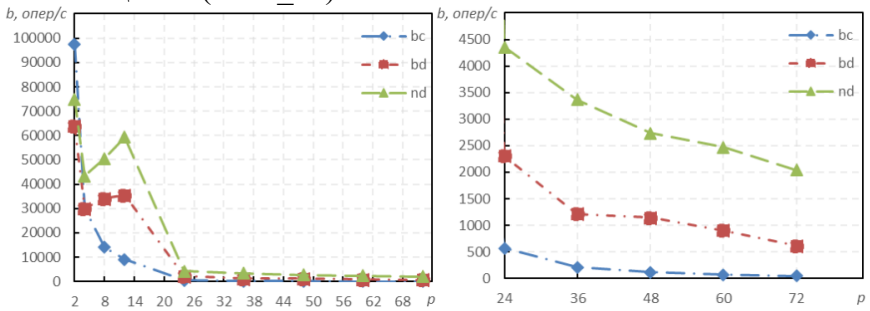

Рис. 4. Пропускная способность очередей (bc – блокирующая централизованная, bd – блокирующая децентрализованная, nd – неблокирующая децентрализованная)

Большая часть времени уходит на поиск головы/хвоста (рис. 5). Объясняется тем, что с ростом числа процессов увеличивается число операций с очередью в единицу времени. Отсюда растет вероятность для процесса p_i получить по ссылке t_i элемент, который уже не является хвостом. Тогда процессу приходится переходами по ссылкам на следующие элементы добираться до хвоста. Чем больше операций добавления в очередь выполняется в единицу времени, тем больше времени тратится на поиск хвоста. Та же проблема и с операцией удаления. Как и следовало ожидать, в пределах одного узла (до 12 процессов) операции выполняются достаточно быстро, а с увеличением числа узлов время растет. Существенно влияние оказывают накладные расходы на передачу данных между узлами.

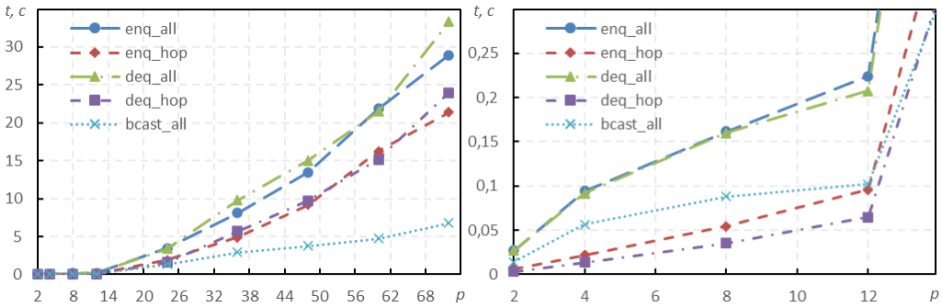

Рис. 5. Время операций добавления и удаления из неблокирующей очереди

5. Эффект от использования кластера в достижении целей работы:

Использование кластера в рамках распределенной неблокирующей очереди приводит к увеличению пропускной способности и повышению производительности по сравнению с блокирующими аналогами. Кластер позволяет распределить нагрузку на процессы, что приводит к более эффективному использованию вычислительных ресурсов. Неблокирующий

подход позволяет избежать накладных расходов на захват и освобождение блокировок, а также позволяет повторить операцию немедленно в случае неудачи, что сокращает время ожидания. В случае увеличения числа процессов пропускная способность структуры данных падает, но все равно остается на достаточно высоком уровне.

Перечень публикаций, содержащих результаты работы

1. Бураченко А.В., Пазников А.А., Державин Д.П. Распределенная очередь без использования блокировок в модели удаленного доступа к памяти // Вестник ТГУ. Управление, вычислительная техника и информатика. – 2023. – № 1 (62). – С. 13-24.