МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Параллельные алгоритмы»

Тема: Реализация параллельной структуры данных с тонкой блокировкой.

Студентка гр. 9303 _____ Москаленко Е.М.

Преподаватель _____ Сергеева Е.И.

Санкт-Петербург
2022

Цель работы.

Реализовать взаимодействие потоков по шаблону “производитель-потребитель”.

Задание.

Обеспечить структуру данных из лаб.2 как минимум тонкой блокировкой (* сделать lock-free).

Протестировать доступ в случае нескольких потоков-производителей и потребителей. Сравнить производительность со структурой с грубой синхронизацией (т.е. с лаб.2).

В отчёте сформулировать инвариант структуры данных.

Выполнение работы.

На основе данных лабораторной работы №2 была реализована lock-free структура данных - неблокирующая очередь Майкла-Скотта.

Очередь построена на односвязном списке - класс LockFreeQueue. Каждый элемент списка Node содержит ссылку на хранимые в нём данные (умный указатель std::shared_ptr, в данном случае на пару матриц или матрицу-результат) и атомарный указатель на следующий элемент списка. (std::atomic<Node*>). Класс очереди хранит два поля - указатель на начало очереди (head) и конец (tail). Head всегда является фиктивным элементом, который просто на что-то ссылается, а его данные не важны. Удаление происходит со стороны головы, добавление - с конца. Изначально, когда очередь пустая, head и tail просто хранят нулевой указатель и никуда не ссылаются. Head должна указывать на узел, который находится не правее узла, на который указывает tail.

Так как мы не используем блокировки, доступ к очереди могут иметь несколько потоков, а добавление элемента в очередь и запись нового указателя в tail - два действия, которые выполняются не атомарно. Для добавления или удаления элементы будем использовать CAS- операции.

Операция добавления элемента produce()

Каждый поток копирует себе значение конца очереди tail, рассматриваемое в данный момент. Если поток вдруг видит, что рассматриваемый им конец очереди вдруг не указывает на nullptr, провалив CAS(рассматриваемый хвост. next, nullptr, новый элемент), (то есть на самом

деле это не хвост) , то он должен помочь обновить tail, выполнив CAS(tail, рассматриваемый хвост, рассматриваемый хвост.next). Если CAS выполнен успешно, то поток обновил tail и может добавить новый элемент в конец очереди.

Операция удаления элемента consume()

Необходимо не допустить, чтобы head указывал на элемент, находящийся правее элемента, на который указывает tail. Поэтому при перемещении указателя head, нужно следить, куда указывает tail. В методе удаления рабочий поток также будет помогать перенести указатель tail на последний добавленный элемент. Каждый поток при попытке удаления сохраняет в локальные переменные текущие голову, хвост и второй элемент очереди.

Если head и tail совпадают - то возникает два варианта: либо очередь пуста и оба они указывают на nullptr, либо просто не успели подвинуть указатель на tail. Поэтому проверяем - если наш второй элемент со значением nullptr, то очередь действительно пуста и мы просто ждем в цикле, когда produce() добавит в очередь новый элемент. Если же второй элемент не nullptr, то просто не был подвинут хвост и поток должен помочь записать в tail второй элемент: **CAS(T, tail, nextHead)**.

Если head и tail не совпадают, то просто проверяем второй элемент. Если он есть (head на что-то указывает), то он должен сам стать head (фиктивный элемент, а предыдущее значение head будет удалено. Сохраняем данные второго элемента в локальную переменную (так как head элемент-пустышка, то нас всегда будет интересовать только значение второго элемента) и с помощью CAS пробудем записать в реальный head очереди наш второй элемент. Если получилось - возвращаем сохраненное значение, иначе кто-то до этого потока уже успел обновить голову очереди, нужно вернуться в начало цикла и повторить операцию еще раз.

Сравнение производительности со структурой с грубой синхронизацией данных

Проведем сравнение со структурой с грубой синхронизацией данных из лабораторной работы №2 при различных количествах производителей, потребителей и итераций (работа с матрицами 10*10, 5 суммирующих потоков).

Количество производителей и потребителе	**Количество итераций**	**Lock-free, микросекунды**	**Грубая синхронизация (размер буфера 100),**	**Грубая синхронизация (размер буфера 1000),**

й			**микросекунды**	**микросекунды**
3	10	32 385	12 521	21 082
15	20	138 904	252 097	110 012
30	100	1 254 420	1 314 817	1 120 875
50	1000	21 025 686	17 598 299	20 765 709

Во всех случаях структура данных с блокировками оказалась лучше по производительности. В алгоритме без блокировок все потоки всегда в активном состоянии и постоянно крутятся в цикле, пытаясь удалить или добавить элемент, в алгоритме с блокировками потоки просто засыпают, пока работающий поток захватывает замок. К тому же примитив CAS – довольно тяжелая инструкция, сильно нагружающая кеш и внутренний протокол синхронизации, а атомарные операции зачастую выполняются медленнее атомарных.

Выводы.

На языке программирования C++ была реализована неблокирующая очередь Майкла-Скотта, проведено сравнение производительности этой структуры данных с очередью, использующую грубую синхронизацию, при различном количестве итераций и потоков, действующих по шаблону “производитель-потребитель”.