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.

Студент гр. 0381 _____ Соколов Д. В.

Преподаватель _____ Сергеева Е.И.

Санкт-Петербург
2022

Цель работы.

Реализация параллельной структуры данных с тонкой блокировкой для сложения матриц.

Задание.

Реализация параллельной структуры данных с тонкой блокировкой. 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.

В качестве структуры данных был выбран стек Трайбера, он реализован соответствующим классом *TRStack*. Основная идея данной структуры – перед добавлением (или удалением из стека) какого-либо элемента необходимо для начала убедиться, что этот элемент будет единственным изменением в стеке, с момента начала операции. Таким образом, обеспечивается предотвращение состояние гонки, в случае работы с множеством процессов в один момент времени. Конкретный механизм синхронизации будет описан подробно позднее.

Структурно *TRStack* был реализован на базе односвязного списка. Для этого была написана вспомогательная структура узла односвязного списка Node. Каждый узел Node имеет константное поле data, в котором и будут храниться данные (в нашем случае матрицы), а также указатель *shared_ptr* на следующий узел.

Основным инструментом синхронизации (обеспечения единственности изменений в определенный момент времени) является операция **CAS** (compare-and-set), реализованная в стандартной библиотеке *std* функцией *atomic_compare_exchange_weak()*. Операция атомарна, т.е. имеет лишь 2 исхода – успех или неудача. Функция принимает 3 аргумента: Адрес проверяемого объекта в памяти *obj*, адрес объекта с ожидаемым значением *expected* и желаемое

значение 1 объекта *desired*. Выполняется сравнение на равенство значение по адресам 1 и 2 аргументов.

- В случае успеха: по адресу *obj* заносится значение *desired* и возвращается *true*;
- В случае неравенства значений: в *expected* заносится значение из *obj*.

Описания структуры стека:

- Стек имеет единственно приватное поле – указатель *shared_ptr* на «голову стека» head. Данный указатель может быть *nullptr*, что свидетельствует о том, что стек пустой в данный момент времени. Такая ситуация обрабатывается при попытке изъять элемент из стека.
- Публичный метод *push()* позволяет добавлять элемент в стек. При помощи копирования «головы» и последующего сравнения **CAS** в бесконечном цикле *while* гарантируется, что в один момент времени не будет произведено множество изменений над стеком разными потоками.
- Публичный метод *pop()* позволяет получать элемент из стека. Он также копирует голову для обеспечения синхронизации между потоками и в бесконечном цикле *while* производит проверку на то, что указатель не *nullptr* (т.е. стек не пустой) и **CAS**.

Инвариант структуры данных:

Вышеописанные условия дают нам определенные гарантии при работе с получившимся стеком.

1. Описанный выше принцип синхронизации гарантирует, что в один конкретный момент времени общее состояние стека может быть изменено только единожды одним потоком, независимо от количества потоков, пытающихся внести изменения.
2. Если стек пуст, то поток, пытающийся получить из него элемент, будет ожидать пока элемент в стеке не появится.
3. В связанном списке не может возникнуть циклов.

Сравнение со структурой, использующей грубую синхронизацию:

Для проверки производительности была выбрана для сравнения блокирующая очередь, реализованная в предыдущей лабораторной работе. В нее были внесены незначительные изменения для улучшения производительности (метод push теперь также, как и аналогичный у стека принимает значение по ссылке, вместо копирования). Были проведены исследования времени выполнения работы программы, считающей сумму случайно сгенерированных матриц по схеме, производитель-потребитель (producer-consumer), с множественными производителями и потребителями.

Были проведены эксперименты с фиксированными размерами матриц 10x10, 100x100, 1000x1000. Для каждого постепенно увеличивалось количество задействованных операций. Количество потоков – показатель того, сколько пар потоков производитель-потребитель будет создано, а количество итераций – сколько раз каждая пара потоков будет производить операции, таким образом – количество операций = количество потоков * количество итераций. При это очевидно, что по времени итерации дешевле, чем создание новых потоков.

Исходя из этого были выбраны следующие шаги экспериментов:

- 1 операция – 1 поток с 1 итерацией

- 5 операций – 1 поток с 5 итерациями
- 10 операций – 2 потока с 5 итерациями
- 25 операций – 5 потоков с 5 итерациями
- 50 операций – 5 потоков с 10 итерациями
- 100 операций – 10 потоков с 10 итерациями
- 300 операций – 10 потоков с 30 итерациями

Количество операций	**Время выполнения с блокировкой (мс)**	**Время выполнения с lock-free (мс)**
1	2	1
5	8	17
10	26	39
25	**73**	**104**
50	160	190
100	266	263
300	823	683

Таблица 1. Результаты эксперимента с матрицами 10x10

Количество операций	**Время выполнения с блокировкой (мс)**	**Время выполнения с lock-free (мс)**
1	7	8
5	33	36
10	53	78
25	**182**	**139**
50	275	240
100	657	553
300	1827	1255

Таблица 2. Результаты эксперимента с матрицами 100x100

Количество операций	**Время выполнения с блокировкой (мс)**	**Время выполнения с lock-free (мс)**
1	350	409
5	1591	1682
10	1742	1991
25	2629	3288
50	4962	7527
100	8285	9622
300	27243	29806
500(10x50)	**41232**	**40805**
900 (30x30)	142087	108080

Таблица 3. Результаты эксперимента с матрицами 1000x1000

Опираясь на полученные результаты, можно сделать следующие выводы: lock-free структуры данных более оптимальны для большого количества операций между потоками. Однако, «переломный момент» напрямую зависит от объема данных, обработкой которых занимается программа (В нашем случае, чем больше матрицы, тем менее эффективно работает lock-free алгоритм относительно алгоритма с толстой блокировкой).

Выводы.

В ходе выполнения лабораторной работы была реализована lock-free структура данных. Было проведено сравнительное исследование между lock-free и толстой синхронизацией. По результатам исследования было выявлено, что при большом количестве взаимодействий между потоками большую производительность показывает lock-free структура, однако есть еще прямая зависимость от объема данных, с которыми работает программа.