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

# ОТЧЕТ

**по лабораторной работе №2-3**
**по дисциплине «Параллельные алгоритмы»**
**Тема: Реализация потокобезопасных структур данных с и без блокировок**

Студент гр. 0303 _____ Скиба В.В.

Преподаватель _____ Сергеева Е.И.

Санкт-Петербург
2023

**Цель работы.**

Рассмотреть модель исполнения MPMC (много производителей много потребителей), в основе которой находится общая очередь задач, исследовать различные сценарии используемой очереди (с грубой блокировкой, с тонкой блокировкой, lock-free очередь). Сравнить эффективность разных подходов к решению задачи.

В реализации представлены несколько сценариев, для которых абстрактный интерфейс исполнения требует от пользователя следующие значения:

- Количество производителей: Количество потоков, которые необходимо создать, а также общее количество задач умножения матриц, которые необходимо выполнить. Каждый поток генерирует пару квадратных матриц размера N и кладет в общую очередь задач.

- Количество потребителей (исполнителей): Количество потоков, которые выполняют задачу/задачи перемножения матриц. Для создания масштабируемой архитектуры все потребители были наделены возможностью исполнять несколько задач если они еще есть и завершать исполнение если свободные задачи закончились.

- Размер матрицы N.

Список абстрактных реализаций:

1) С использованием грубой блокировки (**my_blocking_queue**). Вся очередь задач находится под одним мьютексом. Операции положить в очередь и достать из очереди происходят при заблокированном мьютексе. Для избавления от поллинга под мьютексом используется condvar, который оповещает исполнителя в момент, когда в очередь задач кладется одна задача. Когда кладется последняя задача, вызывается оповещение всех ожидающих потоков, чтобы они моги убедиться, что очередь пуста и завершить свои циклы исполнения.

2) С использованием тонкой блокировки (**my_shared_queue**). В этом случае мьютекс находится на указателе на голову и указателе на хвост, давая возможность однотипным операциям потенциально исполняться не мешая друг другу. Для оповещения потребителей также используется condvar.

3) С использованием структуры SegQueue из библиотеки crossbeam (**crossbeam_queue**). Подробности реализации данной структуры неизвестны, однако библиотека предоставляет достаточно удобный вариант использования очереди предоставляя внутреннюю синхронизацию. К этой очереди можно обращаться из разных потоков, поэтому используется тот же исполнитель, что и в прошлом варианте.

4) С использованием очереди lock-free из библиотеки lockfree (**lock_free_queue**). Все то же самое, но в качестве очереди используется lockfree::queue::Queue

5) Дополнительный вариант с асинхронностью и передачей сообщений (**async_channels_threads**). Данный вариант раскрывает наиболее идиоматическое решение, доступное для языка Rust, которое делает код более компактным.

Кроме того, реализованы дополнительные фичи, при активации которых, каким-либо образом меняется реализация:

**1) advanced_sync** – При активации данной фичи, все примитивы синхронизации из стандартной библиотеки заменяются на альтернативные из библиотеки parking_lot. Создатель библиотеки утверждает, что использование их дает прирост к производительности как в свободном случае исполнения, так и в нагруженном, где много потоков конкурируют за доступ к переменной.

2) **parking-backoff-runner** – Данная фича заменяет реализацию для сценариев 2-4 (с shared очередью задач): вместо condvar используется поллинг с применением стратегии exponential backoff, которая включает в себя операции пробуждения потока и отправки потока в сон.

Для оценки работы структур, рассмотрим среднее арифметическое времени работы для всех сценариев на основе следующих наборов данных:

1) Оценка нагруженного сценария: 1000 производителей, 100 потребителей-исполнителей, 100x100 размер матриц
2) Оценка сбалансированного сценария (число исполнителей равно числу логических ядер в системе): 100 производителей, 12 потребителей, 300x300 размер матриц.
3) Оценка свободного сценария: 100 производителей, 1 потребитель, 100x100 размер матриц.
4) Оценка нагрузки на операции с очередью: 1000 производителей, 12 потребителей, 10x10 размер матриц

Результаты для каждого набора данных

1. 1000 100 100

| | no enhancements | Advanced sync | Advanced sync + backoff |
|---|---|---|---|
| Blocking queue | 215 | 229 | 213 |
| Shared queue | 221 | 215 | 228 |
| Crossbeam queue | 230 | 217 | 218 |
| Lock-free queue | 218 | 215 | 223 |
| async + message passing | 270 | 265 | 263 |

2. 100 12 300

| | no enhancements | Advanced sync | Advanced sync + backoff |
|---|---|---|---|
| Blocking queue | 591 | 579 | 585 |
| Shared queue | 585 | 580 | 583 |
| Crossbeam queue | 574 | 579 | 590 |
| Lock-free queue | 588 | 581 | 584 |
| async + message passing | 584 | 585 | 588 |

3. 100 1 100

| | no enhancements | Advanced sync | Advanced sync + backoff |
|---|---|---|---|
| Blocking queue | 157 | 160 | 148 |
| Shared queue | 144 | 147 | 149 |
| Crossbeam queue | 155 | 150 | 152 |
| Lock-free queue | 147 | 151 | 148 |
| async + message passing | 148 | 147 | 148 |

4. 1000 12 10

| | no enhancements | Advanced sync | Advanced sync + backoff |
|---|---|---|---|
| Blocking queue | 34 | 34 | 34 |
| Shared queue | 34 | 35 | 36 |
| Crossbeam queue | 36 | 35 | 36 |
| Lock-free queue | 34 | 35 | 36 |
| async + message passing | 7.8 | 7.0 | 7.1 |

**Выводы.**

По данным измерений сложно сделать конкретные выводы из-за большого числа факторов, однако некоторые моменты прослеживаются довольно явно:

Почти во всех случаях все 4 варианта реализации очереди показали себя одинаково (различия на уровне погрешности). Backoff стратегия, похоже, помогла больше всего при использовании грубой блокировки с одним исполнителем. Версия с использованием асинхронности дала большую прибавку в случае, когда нагрузка задачи на процессор оказалась небольшой, однако в случае числа исполнителей сильно больше числа логических ядер в системе результат получился хуже.