«Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И.Ульянова (Ленина)»
(СПбГЭТУ «ЛЭТИ»)

Направление	09.04.01 “Информатика и вычислительная техника”
Программа	Распределенные интеллектуальные системы и технологии
Факультет	КТИ
Кафедра	ВТ

К защите допустить
Зав. кафедрой
д. т. н., профессор
М. С. Куприянов

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

ТЕМА: «СРЕДСТВА ПОВЫШЕНИЯ МАСШТАБИРУЕМОСТИ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ НА ОСНОВЕ ПОТОКОБЕЗОПАСНЫХ СТРУКТУР ДАННЫХ С ОСЛАБЛЕННОЙ СЕМАНТИКОЙ ВЫПОЛНЕНИЯ ОПЕРАЦИЙ»

Студент А. В. Табаков
подпись

Руководитель к. т. н., доцент А. А. Пазников
подпись

Консультанты М. Н. Гречухин
подпись

к. э. н., доцент И. А Садырин.
подпись

Санкт-Петербург
2020 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Утверждаю
Зав. кафедрой ВТ
_____ М. С. Куприянов
«___»_____2020 г.

Студент А. В. Табаков Группа 4307

Тема работы: Средства повышения масштабируемости параллельных программ на основе потокобезопасных структур данных с ослабленной семантикой выполнения операций

Место выполнения ВКР: СПбГЭТУ “ЛЭТИ”

Исходные данные:

Научные работы и пособия на тему разработки программного обеспечения в многопоточной среде, методов синхронизации, методов к ослаблению семантики выполнения операций.

Платформа разработки: операционная система – Ubuntu Linux, MacOS X, Windows 10. Языки программирования – Kotlin, C++, Bash. Среда разработки – Intellij IDEA, CLion.

Технические требования:

– разработанные алгоритмы должны быть построены с учётом иерархической структуры многоядерных систем с общей памятью;

– алгоритмы должны иметь способность к динамическому масштабированию в зависимости от числа ядер многоядерной системы;

– предложенные методы должны иметь абстрактное представления для использования подходов с различными последовательными структурами данных.

Содержание ВКР:

Техническое задание, анализ методов ослабления семантики, описание разработки алгоритмов для Multiqueue, описание разработки подхода построения ослабленных структуры данных на основе циклических списков, реализация потокобезопасной ослабленной очереди Circular Relaxed Priority Queues, отладка и экспериментальное исследование, бизнес-план по коммерциализации результатов НИР магистранта

Перечень отчетных материалов: пояснительная записка, исходный код программных продуктов на языках программирования Kotlin и C++, презентация.

Дополнительные разделы: Составление бизнес-плана по коммерциализации результатов НИР

Дата выдачи задания «___»_____2020 г.

Дата представления ВКР к защите «___»_____2020 г.

Студент А. В. Табаков

Руководитель, к. т. н., доцент А. А. Пазников

КАЛЕНДАРНЫЙ ПЛАН ВЫПОЛНЕНИЯ ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Утверждаю

Зав. кафедрой ВТ

_____ М. С. Куприянов

«___»_____2020 г.

Студент Табаков А. В. Группа 4307

Тема работы: Средства повышения масштабируемости параллельных программ на основе потокобезопасных структур данных с ослабленной семантикой выполнения операций

№ п/п	Наименование работ	Срок выполнения
1	Техническое задание	01.09.2018 –03.11.2018
2	Анализ методов ослабления семантики	04.11.2018 – 12.12.2019
3	Разработка алгоритмов для Multiqueue	13.12.2019 –27.12.2019
4	Разработка подхода построения ослабленных структуры данных на основе циклических списков	02.01.2020 –05.03.2020
5	Отладка и экспериментальное исследование	06.03.2020 – 27.03.2020
6	Составление бизнес-плана по коммерциализации результатов НИР магистранта	30.03.2020 –30.04. 2020
7	Оформление пояснительной записки	01.05.2020 –20.05.2020
8	Предварительное рассмотрение работы	26.05.2020
9	Представление работы к защите	26.05.2020

Студент А. В. Табаков

Руководитель, к. т. н., доцент, с. н. с. А. А. Пазников

РЕФЕРАТ

Ключевые слова: Ослабленная семантика выполнения операций, потокобезопасные структуры данных, многоядерные системы, системы с общей памятью.

Цель работы: оптимизировать алгоритмы для структуры данных с ослабленной семантикой выполнения операций Multiqueue. Реализовать потокобезопасную ослабленную очередь с приоритетом Circular Relaxed Priority Queues. Распространить, разработать и формализовать подход к построению ослабленных структур данных на основе циклических списков для любых последовательных структур данных.

В результате выполнения выпускной квалификационной работы были спроектированы и реализованы оптимизированные алгоритмы для потокобезопасной ослабленной структуры Multiqueue. Созданный алгоритм увеличивает пропускную способность вставки и удаления в 1.2 и 1.6 раз соответственно.

Разработана циклическая ослабленная очередь с приоритетом Circular Relaxed Priority Queues. Предложенный подход к построению структур данных с ослабленными требованиями к семантике выполнения операций позволяет использовать в качестве базовой структуры, любую последовательную структуру данных. Реализация программных продуктов представлена на двух языках программирования Kotlin и C++. Подтверждение эффективности таких структур данных было получено в результате выполнения тестов на кластере суперкомпьютера.

Разработанные подходы и алгоритмы могут быть использованы в составе прикладного ПО, исполняемом в многопоточных средах, таких как многоядерные системы с общей памятью и мультикластерные вычислительные системы.

ABSTRACT

Keywords: Relaxed concurrent data structure, thread-safe data structure, multicore system, systems with shared memory.

Distributed parallel computing is the most promising direction of study in computer science. A distributed computed computing system (CS) is the main mean of parallel processing information. The symmetric multiprocessor (SMP) computing node, connected with each other, form distributed CS. One SMP node can use single memory access by many multicore processors. Modern distributed CSs implement thread-level parallelism (TLP) within only one computing node (multi-core processor with shared memory) and process-level parallelism (PLP) for the entire distributed CS. We focused on TLP implementation. In shared-memory systems, relaxation of operation execution order it is a promising approach, to design a good scalable concurrent data structure. The growing demand of high-performance computing arouses such problems as scalability of parallel programs (especially in complex hierarchical computing systems).

Concurrent data structures with relaxed semantic are non-linearizable and do not provide strong operation semantics (such as FIFO/LIFO for linear lists, delete max (min) element for priority queues, etc.).

In the paper, we use the approach based on the design of concurrent data structure as multiple simple data structures distributed among the threads. For concurrent operation execution (insert, delete), a thread chooses a subset of sequential data structures and perform operations on them. Was proposed two optimized relaxed concurrent priority queues based on this relaxed approach. Was designed algorithms for optimization of priority queues selection in multiqueues. It had better throughput for insert and delete operations in 1.2 and 1.6 times respectively. Was developed a circular relaxed concurrent data structure based on a linked circular list of sequential data structures.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

API (англ. Application Programming Interface) – набор готовых реализаций методов, классов, структур и констант, которые предоставляются сервисом или библиотекой для использования во внешних программных продуктах.

CPQ (англ. Circular Relaxed Priority Queue) – ослабленная очередь с приоритетом на основе циклического списка.

IDE (англ. Integrated Development Environment) – среда разработки, система программных компонентов, используемая программистами для разработки программного обеспечения.

JVM (англ. Java Virtual Machine) – виртуальная машина, исполняющая байт-код Java.

Kotlin – статически типизированный язык программирования, работающий поверх JVM и разрабатываемый Российской, Санкт-Петербургской компанией JetBrains.

SMP – Многоядерная система с общей памятью.

STL – стандартная библиотека шаблонов.

ВВЕДЕНИЕ

Многоядерные вычислительные системы (ВС) с разделяемой памятью являются основными средствами высокопроизводительных вычислений для решения сложных задач и обработки сверхбольших объемов данных [1]. Один вычислительный узел может использоваться как автономный компьютер с одним или несколькими процессорами с общей областью памяти, а также как часть распределенных ВС (кластерные, мультикластерные системы, системы с массовым параллелизмом). Каждый вычислительный узел распределенной ВС содержит большое число многоядерных процессоров и имеет иерархическую структуру, представленную на рисунке 1.

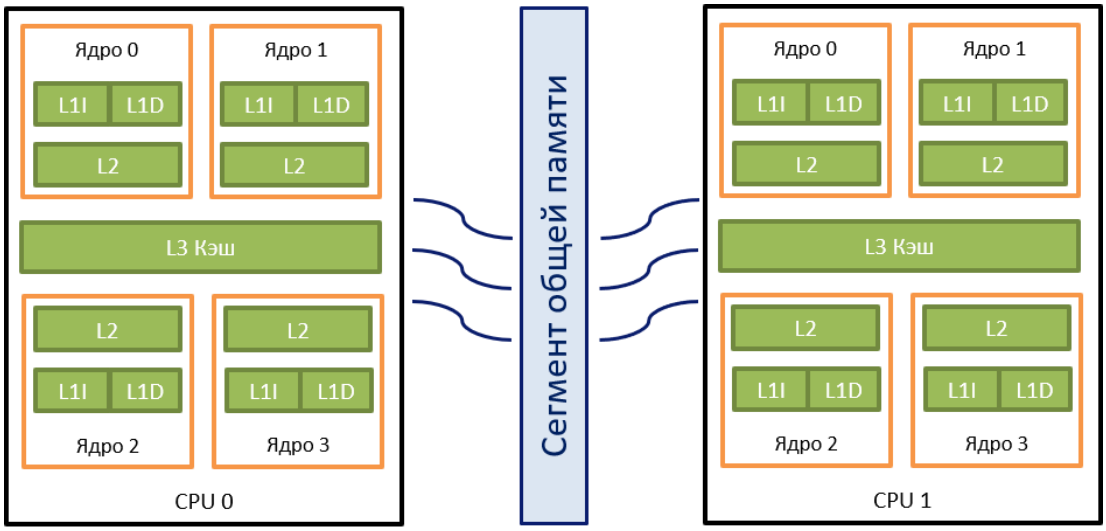


Рисунок 1 – Иерархическая структура многоядерной системы с общей памятью

Параллельные вычислительные технологии сегодня применяются повсеместно – от встроенных систем и мобильных устройств до систем с массовым параллелизмом, GRID-систем и облачных ВС. Кроме того, алгоритмический и программный инструментарий параллельного программирования является базой при построении современных систем обработки больших данных, машинного обучения и искусственного интеллекта.

Суперкомпьютер Summit (OLCF-4) (17 миллионов процессорных ядер), который является первым в рейтинге TOP500, включает 4608 вычислительных узлов [2].

Распределенные вычислительные системы (ВС) в настоящий момент остаются основным средством параллельной высокопроизводительной обработки информации. Такие системы представляют множества элементарных машин, взаимодействующих посредством коммуникационной среды. Для современных ВС характерным является большемасштабность, мультиархитектура и иерархическая структура. Среди существующих ВС с общей памятью выделяют SMP-системы. Они обеспечивают одинаковую скорость доступа к памяти для всех процессоров на узле. Также выделяют NUMA-системы, представленные в виде коммуникационной среды множества вычислительных узлов (в составе которых находится процессор и локальная память) и характеризующиеся различным временем доступа процессоров к сегментам локальной и удалённой памяти [3].

Многопоточность – основной подход к распараллеливанию в среде с общей памятью. Данный подход может обеспечить более высокую производительность, когда одну большую задачу можно разделить на более мелкие независимые подзадачи для выполнения. Данную задачу можно решить, используя параллельные подходы многопоточности или метода work-stealing [4]. Ключевой проблемой в многопоточности является организация параллельного доступа нескольких потоков к разделенным областям памяти. Все операции должны выполняться корректно (без состояния гонки, взаимных блокировок и т. д.). Более того, необходимо обеспечить высокую масштабируемость для большого количества потоков в системах с высокой интенсивностью параллельных операций. Для корректности выполнения операций в многопоточных средах используются методы синхронизации. В основе любого механизма синхронизации используются атомарные операции [5]. В ходе разработки параллельных программ следует учитывать архитектурные и иерархические особенности ВС для обеспечения максимальной производительности и загруженности системы. Неправильно спроектированная параллельная программа скорее всего увеличит время выполнения по сравнению с однопоточной версией, в худшем случае замедлит её.

Производительность существующих параллельных структур данных из-за строгой семантики выполнения операций может быть недостаточной для современных многопоточных программ. Данные структуры данных имеют узкое место в выполнении операций в виде критических секций. В то же время как параллельные структуры данных с ослабленными требованиями к семантике выполнения операций могут значительно уменьшить вероятность состязания потоков за разделяемый ресурс. В данной работе используется метод увеличения масштабируемости за счет ослабления семантики структур данных.

Цель работы: оптимизировать алгоритмы для структуры данных с ослабленной семантикой выполнения операций Multiqueue. Реализовать потокобезопасную ослабленную очередь с приоритетом Circular Relaxed Priority Queues. Распространить, разработать и формализовать подход к построению ослабленных структур данных на основе циклических списков для любых последовательных структур данных.

Предмет разработки: методы и алгоритмы над потокобезопасными структурами данных с ослабленной семантикой выполнения операций.

В первой главе представлен краткий обзор механизмов синхронизации, существующих решений ослабленных структур данных, а также описание проблем, связанных с их производительностью. Вторая глава содержит алгоритмы и описания подхода работы с ослабленными структурами данных. В третьей главе даны описания архитектур и особенности реализации данного вида потокобезопасных структур. Четвёртая глава представляет собой описание проекта с точки зрения коммерциализации, представлен план продвижения готового программного продукта в коммерческие организации. В заключении представлены выводы по настоящей работе.

1 Обеспечение синхронизации в параллельных программах

Параллельные программы для многоядерных ВС с общей памятью разрабатываются в парадигме многопоточного программирования и должны обеспечивать близкое к линейному ускорение при большом количестве параллельных потоков. Одной из важных проблем параллельных программ, является одновременный доступ к общим ресурсам, в частности к разделяемым структурам данных. Такие структуры должны обладать свойством потокобезопасности, чтобы одновременный доступ разных потоков не нарушал целостность и корректность работы данной структуры.

Для обеспечения корректной работы в многопоточной среде используются методы синхронизации, накладывающие ограничения на одновременный доступ к разделяемым ресурсам.

В данном разделе рассматриваются различные методы синхронизации выполнения параллельных программ в многопоточной среде. Приводится описание и анализ существующих представителей потокобезопасных структур данных с ослабленной семантикой выполнения операций.

1.1 Механизмы синхронизации

При проектировании параллельных программ для ВС (как с общей памятью, так и с распределенной памятью) требуется обеспечить синхронизацию параллельных потоков (процессов) для предоставления доступа к разделяемым структурам данных. Эффективность синхронизации в значительной степени определяет время выполнения программ. Средства синхронизации должны реализовывать безопасное обращение большого числа параллельных потоков (процессов) к разделяемым структурам данных в произвольные моменты времени и обеспечить максимальное количество операций в единицу времени. Традиционно методы синхронизации построены на атомарных операциях, эффективную реализацию которых предоставляют все современные

процессоры [6]. Данные примитивы позволяют построить более сложные механизмы синхронизации параллельных потоков.

1.1.1 Блокировки

Традиционный подход к синхронизации в многопоточных программах путем организации критических секций с помощью блокировок сегодня остаётся наиболее распространённым в практике программирования. Масштабируемость блокировок существенно зависит от решения проблем, связанных с конкурентным доступом (access contention) параллельных потоков к разделяемым областям памяти и пространственной локализацией обращений к памяти (locality of reference). Конкурентный доступ имеет место при одновременном обращении нескольких потоков к критической секции, защищённой одним объектом синхронизации, что в аппаратном плане приводит к увеличению загруженности шины данных и снижению эффективности кэш-памяти. Локальность кэш-памяти имеет существенное значение в том случае, когда внутри критической секции выполняется обращение к разделяемым данным, которые перед этим использовались на другом процессорном ядре. Данное обстоятельство приводит к промахам по кэшу (cache misses) и значительному увеличению времени выполнения критических секций.

Блокировки (Mutex, Semaphore) организуют ограниченный доступ к общему ресурсу. Доступ на одновременную работу с данным разделяемым ресурсом предоставляется только определённому количеству потоков. Можно выделить такие алгоритмы блокировок как TTAS, CLH, MCS, Lock Cohorting, Flat Combining, Remote Core Locking и др. [7].

Блокировки могут быть использованы разными способами, в том числе крупнозернистым (coarse-grained) способом, когда блокируется вся структура, и мелкозернистым (fine-grained) способом, в данном подходе одна структура данных обладает множеством блокировок для предоставления одновременного доступа к различным частям потокобезопасной структуры [8].

Недостатками использования блокировок является возможность возникновения тупиковых ситуаций (deadlocks, livelocks), голодания потоков (starvation), инверсии приоритетов (priority inversion) и высокая конкуренция потоков на захват блокировки (lock convoy).

1.1.2 Потокобезопасные алгоритмы и структуры данных свободные от блокировок

Другим подходом, направленным на обеспечение высокой масштабируемости разделяемых структур данных, является применение неблокирующих методов синхронизации (non-blocking synchronization), к которым относятся алгоритмы и структуры данных, свободные от блокировок.

Как и любая последовательная структура данных, неблокируемая потокобезопасные структуры данных обладают важным свойством линеаризуемости (lineariziability). Данное свойство гарантирует что любое параллельное исполнение операций имеет эквивалентное последовательное исполнение, причём каждая отдельная операция не требует завершения всех остальных операций, исполняющихся в данный момент.

Имеется три класса гарантии выполнения операций в неблокируемых структурах данных: wait-free - каждый поток завершает выполнение любой операции за конечное число шагов, lock-free - часть потоков завершают выполнение операций за конечное число шагов, obstruction-free - любой поток завершает выполнение операций за конечное число шагов, если выполнение других потоков приостановлено. Таким образом, неблокируемые структуры гарантируют выполнение операций за конечное число шагов.

Среди наиболее распространенных неблокируемых алгоритмов и структур данных можно выделить Treiber Stack, Michael & Scott Queue, Harris & Michael List, Elimination Backoff Stack, Combining Trees, Diffracting Trees, Counting Network, Cliff Click Hash Table, Split-ordered lists и др. [9]. Традиционно неблокируемые структуры данных используют механизмы атомарных примитивов.

Недостатками неблокируемых алгоритмов и структур данных, несмотря на их высокую масштабируемость, являются узкая область их применения и существенные трудозатраты, связанные с разработкой параллельных программ. Кроме того, при разработке неблокируемых алгоритмов и структур данных возникают проблемы, связанные с освобождением памяти (проблема ABA), низкой производительностью атомарных операций и невозможностью их выполнения для переменных большого размера. Кроме того, пропускная способность неблокируемых структур часто сопоставима с аналогичными структурами данных, основанными на блокировках.

1.1.3 Транзакционная память

На сегодняшний день транзакционная память является одним из наиболее перспективных механизмов синхронизации, её использование позволяет выполнять не конфликтующие между собой операции параллельно. Транзакционная память упрощает параллельное программирование, выделяя группы инструкций в атомарные транзакции – конечные последовательности операций транзакционного чтения/записи памяти [10]. Изменения, вносимые потоком внутри транзакционных секций, незаметны другим потокам до тех пор, пока транзакция не будет зафиксирована (commit). Если во время выполнения транзакции потоки обращаются к одной области памяти, и один из потоков совершает операцию записи, то транзакция одного из потоков отменяется (cancel, rollback). При выполнении транзакционной секции одним потоком, другие потоки наблюдают состояние либо непосредственно до, либо непосредственно после выполнения транзакции.

Важным свойством транзакционной памяти является линеаризуемость выполнения транзакций: ряд успешно завершённых транзакция эквивалентен некоторому последовательному их выполнению.

Таким образом, транзакции обладают качествами атомарности, согласованности, изолированности, устойчивости (atomicity, consistency, isolation, durability – ACID).

Атомарность – транзакция представляет собой единое целое, не может быть частичной транзакции, иначе говоря, если выполнена часть транзакции, она отменяется.

Согласованность – транзакция не нарушает логику и отношения между элементами данных.

Изолированность – результаты транзакции не зависят от предыдущих или последующих транзакций.

Устойчивость – после своего завершения транзакция сохраняется в системе, происходит фиксация транзакции.

Выделяют программную (LazySTM, TinySTM, GCC TM, etc.) и аппаратную (Intel TSX, AMD ASF, Oracle Rock etc.) реализации транзакционной памяти [11]. Ключевыми проблемами транзакционной памяти являются ограничения на выполнение некоторых видов операций с памятью внутри транзакционных секций, высокая сложность отслеживания изменений в памяти и отладка параллельных программ [12].

1.2 Критерии корректности выполнения параллельных операций с разделяемыми структурами данных

Существуют модели согласованности выполнения операций для многопоточных программ. Они позволяют определить, насколько корректно и прогнозируемо будет выполнение параллельного участка кода.

Одним из наиболее строгих критериев является линеаризуемость (linearizability). Данный критерий устанавливает, что параллельное выполнение операций должно быть эквивалентно последовательному выполнению данных операций [13]. Фактически гарантируется, что параллельная программа будет корректно исполняться и результат выполнения операции можно спрогнозировать в любой момент времени.

Менее требовательным критерием является согласованность покоя (quiescent consistency). Структура данных обладает согласованностью покоя, если она согласована между состояниями покоя [14]. Выполнение является

согласованным, если вызовы всех исполняемых методов могут быть расположены в последовательности, которая сохраняет порядок вызовов, разделенных покоем (период времени, когда ни один метод не выполняется ни в одном потоке).

Важным для многопоточности критерием корректности является последовательная согласованность (sequential consistency). Выполнение операций является последовательно согласованным, если вызовы методов могут быть правильно расположены в последовательности, которая сохраняет порядок вызовов методов в каждом отдельном потоке.

Следующие рассматриваемые критерии корректности предъявляют ослабленные требования к семантике выполнения операций.

Количественное ослабление (quantitative relaxation) – критерий устанавливает, что после выполнения некоторой последовательности действий над структурой, возможно спрогнозировать диапазон результатов операции [15]. Это означает, что результат будет одним из некоторого конечного множества предполагаемых значений, которое можно составить исходя из истории операций.

Принцип квази-линеаризуемости (quasi-linearizability) [16], предполагает, что во время выполнения некоторых операций могут произойти несколько событий, одновременно изменяющие структуру данных таким образом, что после выполнения одной из операций состояние структуры данных не определено. Таким образом, результат операции может, но не должен совпадать с подразумеваемым.

1.3 Потокобезопасные структуры данных с ослабленной семантикой выполнения операций

Перспективным подходом для повышения масштабируемости разделяемых структур данных является ослабление порядка выполнения операций (relaxation). В основе данного подхода лежит компромисс между масштабируемостью (производительностью) и корректностью семантики выполнения

операций. Предлагается ослабить семантику выполнения операций для повышения возможности масштабирования. В большинстве существующих неблокируемых потокобезопасных структур и алгоритмах блокировки существует единая точка выполнения операций над структурой. В случае многопоточной системы, данный факт является узким местом, так как каждый поток вынужден блокировать один элемент, заставляя другие потоки ожидать.

В ослабленных структурах данных единственная структура заменяется на набор простых структур, композиция которых рассматривается как логически единая структура. Вследствие этого увеличивается количество возможных точек обращений к данной структуре, что позволяет избежать возникновения узких мест. В подходе к ослаблению требований к семантике выполнения операций, каждый поток может выбрать из множества структур – свободную структуру. В случае если поток смог заблокировать данную структуру, он может выполнить операции над ней, в ином случае, он ищет другую свободную структуру из множества.

В рамках данного подхода каждая простая структура, как правило, защищается блокировкой. При выполнении операции, поток обращается к случайной структуре из набора и пытается её заблокировать. В случае успешной блокировки структуры, поток завершает выполнение операции; в противном случае, поток случайным образом выбирает новую структуру. Таким образом, синхронизация потоков сводится к минимуму, но допустимы потери точности выполнения операций.

Основными представителями ослабленных структур данных являются k-relaxed Stack, SprayList, k-LSM, Multiqueue.

1.3.1 k-Relaxed Queue

k-Relaxed Queue – очередь с ослабленной семантикой выполнения операций [17]. Вставка элемента происходит в одну из незаблокированных очередей из множества. Каждый элемент содержит информацию о времени добавления в очередь. Во время удаления элемента, поток блокирует k очередей и

просматривает временные метки их элементов, выбирая самую маленькую и удаляет самый старый элемент. Временные метки гарантируют удаление элемента первого элемента в диапазоне k.

Недостатком данного вида потокобезопасных ослабленных структур является блокировка нескольких очередей одним потоком, т. к. в этом случае другие потоки не могут выполнять операции над ними, что ограничивает масштабируемость и ухудшает точность выполнения операций.

1.3.2 SprayList

На рисунке 2 представлена структура SprayList, которая является ослабленной версией списка с пропусками (SkipList). Данная структура представляет собой связный граф, где на нижнем уровне структуры располагается связный сортированный список всех элементов, а на верхних уровнях с установленной вероятностью располагаются клонированные элементы данного списка [18].

Поиск по данной структуре осуществляется линейно сверху вниз и слева направо, каждую итерацию выполняется переход по одному указателю. В отличие от списка с пропусками, SprayList предполагает не линейный поиск сверху вниз и из начала в конец, а случайное перемещение сверху вниз и слева направо. Если поиск не дает результатов или элемент оказался заблокированным другим потоком, алгоритм возвращается на предыдущий элемент. После нахождения нужного элемента операции со списком выполняются также, как и в списке с пропусками.

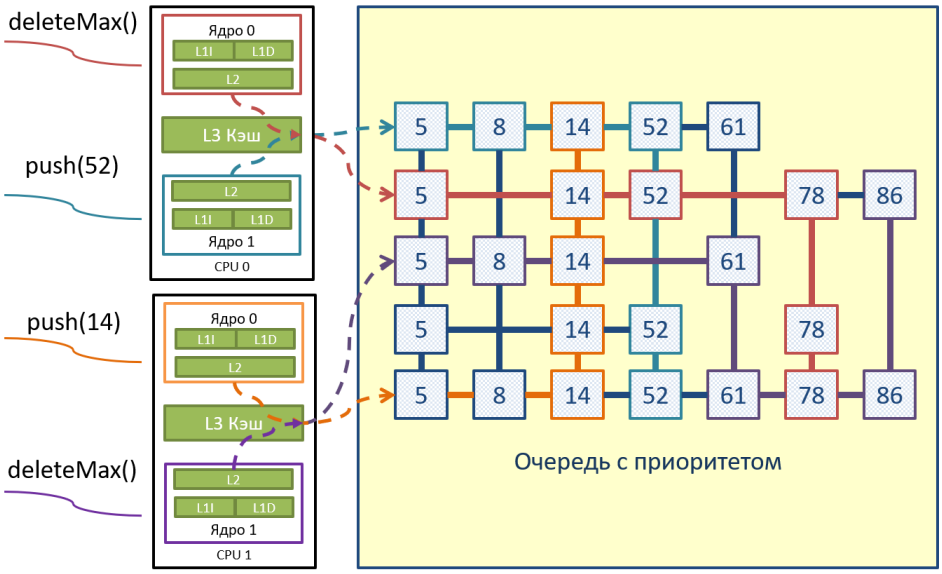


Рисунок 2 – SprayList – Разреженный список

В худшем случае, разреженный список, как и список с пропусками вызывает значительные накладные расходы на производительность, т. к. на нижнем уровне требуется поддерживать сортированный список [19].

1.3.3 k-LSM

В качестве базовой структуры k-LSM [20] используется журнально-структурированное дерево со слиянием (log-structured merge tree, LSM). Каждое изменение структуры записывается в отдельный лог файл, узлы дерева являются отсортированными массивами (блоками), каждый из которых находится на уровне L дерева и может содержать N элементов ($2L - 1 < N \leq 2L$). Каждый поток имеет локальную распределённую LSM, данные процесс представлен на рисунке 3. Общая LSM является результатом слияния нескольких распределённых LSM структур. Все потоки могут обращаться к общей LSM по единому указателю. Локальная LSM сливается с общей LSM в случае превышения фиксированного значения размера допустимого локальной LSM.

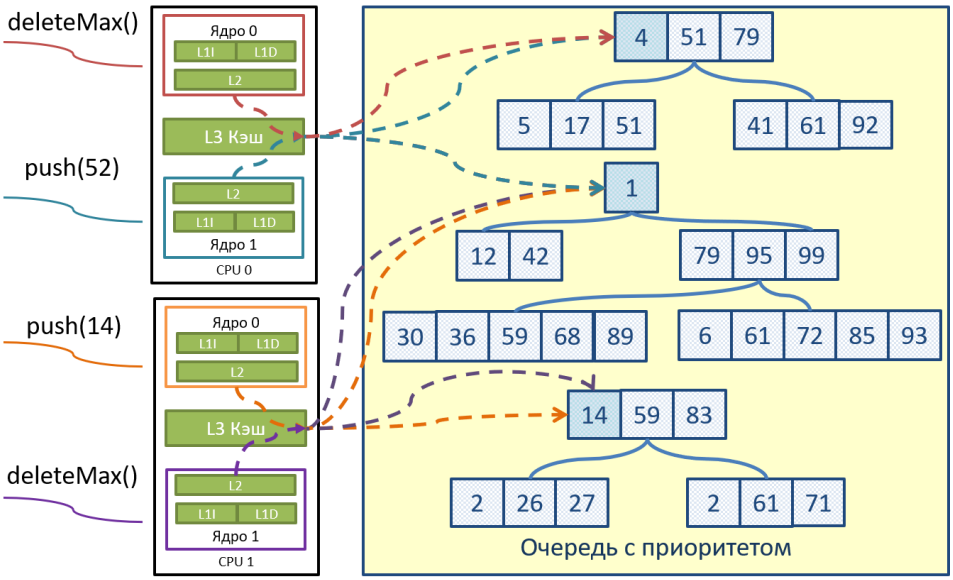


Рисунок 3 – k-LSM - журнально-структурированное дерево со слиянием

В результате объединения общей и распределённых LSM структур, была получена k-LSM структура. Выполняя операцию вставки элемента, поток сохраняет элемент в локальной LSM структуре. При операции удаления, используется поиск наименьшего ключа в локальной LSM, если локальная структура пуста, а требуется операция, отличительная от операции вставки, начинается попытка доступа к чужим LSM структурам, поиск осуществляется среди всех распределённых и общей LSM структур, и, если найденная структура не заблокирована, выполняется операция над ней.

Недостатками данной структуры является синхронизация обращений к общей LSM и попытка обращения к чужой LSM, так как нет гарантии, что структура не является пустой, что может привести к голоданию потоков.

1.3.4 Multiqueue

На рисунке 4 представлена структура Multiqueue [21], которая является композицией последовательных очередей с приоритетом, защищённые от одновременного доступа блокировками.

На каждый поток выделяется две и более очередей с приоритетом. Операция вставки элемента производится над случайно выбранной из множества – очередью, которая не заблокирована другими потоками. Операция удаления

максимального\минимального элемента выполняется следующим образом: поток из множества выбирает две случайные очереди, сравнивает их максимальные\минимальные элементы, и пытается захватить очередь с подходящим элементом, чтобы удалить элемент. В случае успеха, элемент удаляется из выбранной очереди, в ином случае, когда поток не смог заблокировать найденную очередь, он начинает алгоритм сначала. Удалённый элемент не всегда максимальный\минимальный во всей логической очереди, однако для настоящих задач, ошибка может быть несущественна.

Недостатком текущей реализации операций вставки и удаления в Multiqueue является алгоритм поиска случайной очереди. Поток, выполняющий операцию, с большой вероятностью обращается к очередям, заблокированными другими потоками. Структура Multiqueue включает в себя kp очередей, где k – число очередей на один поток, p – количество потоков.

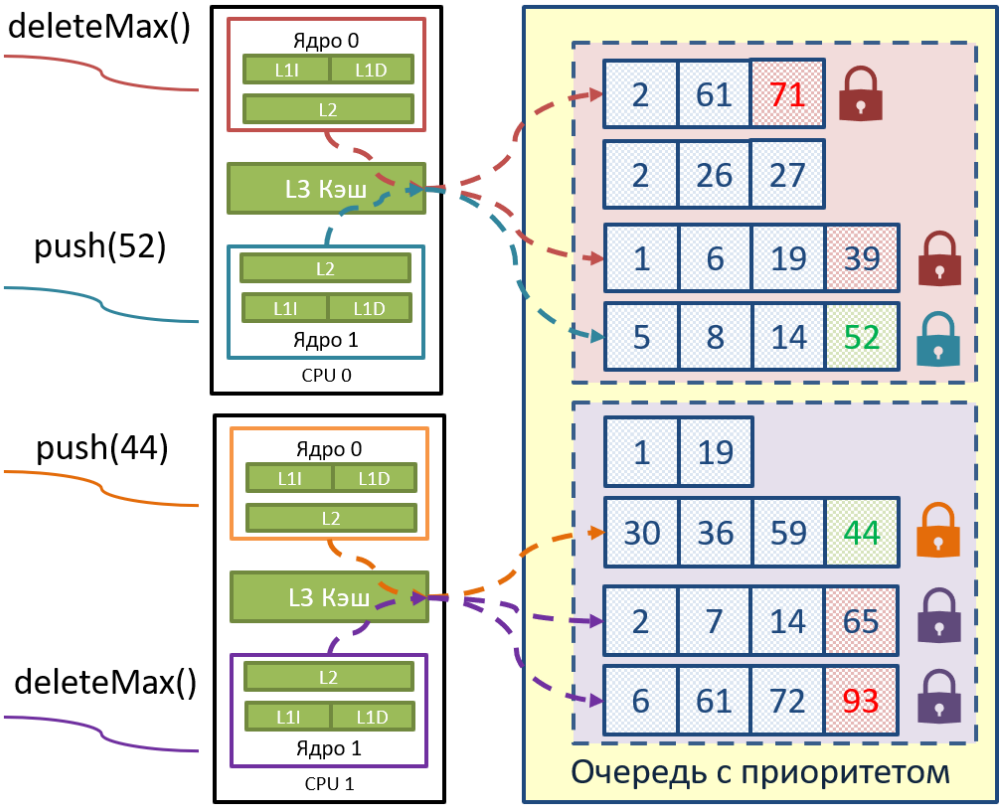


Рисунок 4 – Multiqueue – Ослабленная очередь с приоритетом

1.4 Постановка задачи

Учитывая вышерассмотренные недостатки существующих потокобезопасных структур данных, их производительность может быть недостаточна для современных многопоточных программ.

Построение эффективных потокобезопасных структур данных является одной из приоритетных задач параллельного программирования для вычислительных систем с общей памятью. В настоящей работе предложены масштабируемые структуры данных, не использующие блокировки. При реализации структур планируется использовать подход к ослаблению семантики операций, который обеспечит прогнозируемое время выполнения операций в многопоточных программах даже при большом количестве параллельных потоков и высокой интенсивности загрузки потоков. Структуры будут ориентированы на использование в иерархических мультиархитектурных вычислительных системах с общей памятью. В отличие от большинства существующих неблокируемых потокобезопасных структур данных и алгоритмов блокировки, где существует единая точка выполнения операций над структурой, в ослабленных структурах данных используется набор простых последовательных структур, композиция которых рассматривается как логически единая структура. Вследствие этого увеличивается количество возможных точек обращений к данной структуре, что позволяет избежать возникновения узких мест (bottlenecks). Данный подход позволит достичь значительно большей пропускной способности, по сравнению с существующими структурами данных.

В качестве результата ожидается программный продукт на языках программирования C++ и Kotlin, с проработанными интерфейсами, позволяющие другим разработчикам просто интегрировать данное решение в существующие проекты.

2 Алгоритмы реализации потокобезопасных структур данных с ослабленной семантикой выполнения операций

Данный раздел включает оптимизированные алгоритмы и методы построения структур данных с ослабленными требованиями к семантике операций.

2.1 Оптимизация времени выполнения алгоритмов вставки и удаления в Multiqueue

Структура Multiqueue включает kp очередей, где k – количество очередей на одну очередь, а p – количество запущенных потоков. Текущая реализация операций вставки и удаления имеют недостатки. Главная проблема алгоритмов лежит в поиске подходящей, незаблокированной другим потоком, очереди.

2.1.1 Вычисление идентификатора очереди по потокам

Когда несколько потоков работают с разделяемыми объектами, нет возможности избежать коллизий. Коллизии возникают, в тот момент, когда поток пытается обратиться к общей области памяти, которая уже заблокирована другим потоком. В контексте решения данной задачи, поток пытается заблокировать мьютекс найденной очереди, однако у потока нет никакой информации о том, как оптимально искать очередь в данный момент.

В данной работе предлагается метод, для снижения числа коллизий, основанный на ограничении множества выбора случайной очереди. Для уменьшения количества коллизий, предлагается разделить множество доступных очередей и потоки на две одинаковых группы. Обозначим i в (1) идентификатор потока, тогда для первой половины потоков (2), операция выбора случайной очереди будет выполняться среди очередей (3), где q – выбранная для выполнения операции, очередь в структуре Multiqueue. Для второй половины потоков (4), очередь выбирается среди второй половины множества очередей (5).

$$i \in 0,1,\dots,p \quad (1)$$

$$q \in 0,1,\dots,k \cdot p/2 \quad (2)$$

$$q \in k \cdot p/2 + 1,\dots,k \cdot p \quad (3)$$

$$i \in 0,1,\dots,p/2 \quad (4)$$

$$i \in p/2 + 1 \dots,p \quad (5)$$

$$q \in p \cdot i,\dots,p \cdot i + k \quad (6)$$

Также описан подход оптимизации выбора очередей, основанный на «закреплении» очередей к потокам. Данная схема позволяет задать порядок обращения потока к очередям. В реализации закрепления используется следующая модель: всего в множестве *kp* очередей, тогда каждый поток имеет q (6) закреплённых за потоком очередей. Выполняя операцию с очередью, поток сначала пытается заблокировать закреплённые очереди, для уменьшения количества вызовов заблокированных, другими потоками, очередей. Если очереди из множества (6) заблокированы, то применяется алгоритм поиск очереди алгоритмами (2,3) в зависимости от идентификатора потока (1).

2.1.2 Определения и обозначения

Представленные в данном разделе алгоритмы имеют следующую семантику:

– Lock/Unlock – блокировка и разблокировка мьютекса закреплённого за определённой очередью, поток выполняет операцию синхронно, дожидаясь результата выполнения;

– TryLock – проверяет, что мьютекс выбранной очереди не заблокирован другим потоком и в то же время идёт попытка заблокировать данный мьютекс. Результатом данной операции является булевское значение, где false – мьютекс не заблокирован, а true – свидетельствует о блокировке заданного мьютекса;

– RandomQueue – функция возвращает одну очередь, выбранную случайным образом из множества очередей;

– TwoRandomQueues – выполняет такую же операцию, как и RandomQueue, с тем отличием, что возвращается пара случайно выбранных очередей. threadId – идентификатор потока от 0 до количества потоков p.

2.1.3 Оптимизация алгоритма вставки

Алгоритм на рисунке 5, демонстрирует оптимизированный алгоритм вставки элементов в структуру Multiqueue. Последовательная очередь выбирается на основании того, к какой половине множества очередей имеет доступ текущий идентификатор потока.

```
1  do
2    if p_i ∈ {0, 1 … p/2} then
3      q = RandQueue(0, kp/2);
4    else
5      q = RandQueue(kp/2+1, kp);
6    end
7  while TryLock(q) == false;
8  Insert(q, el);
9  Unlock(q);
```

Рисунок 5 – Оптимизация времени выполнения операции вставка

В строке 2 рассчитывается в какой половине множества очередей будет осуществляться поиск, на основе идентификатора текущего потока. В строке 7 происходит попытка заблокировать мьютекс найденной очереди, если блокировка мьютекса успешна, то происходит вставка элемента в найденную очередь, иначе алгоритм ищет очередь повторно, до тех пор, пока не будет осуществлена блокировка мьютекса.

2.1.4 Оптимизация алгоритма удаления максимального элемента

Представленный ниже алгоритм на рисунке 6, описывает удаление максимального элемента в очереди Multiqueue. Различие между алгоритмом на рисунке 5 и на рисунке 6 заключается в том, что во время поиска оптимизированный алгоритм удаления возвращает пару очередей из половины множества. Данная особенность позволяет получать более точные результаты.

```
1   do
2     if p_i ∈ {0, 1 … p/2} then
3       (q1, q2) = TwoRandomQueues(0, kp/2);
4     else
5       (q1, q2) = TwoRandomQueues(kp/2+1, kp);
6     end
7     q = GetMaxElementQueue(q1, q2);
8   while TryLock(q) == false;
9   DeleteMax(q);
10  Unlock(q);
```

Рисунок 6 – Оптимизация времени выполнения операции удаления максимального элемента

Функция TwoRandomQueues на строках 3 и 5 возвращает пару выбранных случайным образом очередей из заданного диапазона, который вычисляется на основе идентификатора текущего потока. На строке 7 функция GetMaxElementQueue возвращает очередь с наибольшим элементом среди пары выбранных ранее очередей. После этого происходит попытка заблокировать мьютекс данной очереди, если мьютекс удалось заблокировать, максимальный элемент удаляется из очереди, иначе выбирается другая пара очередей.

Альтернативная оптимизация времени операции удаления максимального элемента представлена в алгоритме на рисунке 7. В основе данного метода лежит алгоритм представленный на рисунке 6, с тем отличием, что первая попытка поиска подходящих очередей осуществляется среди закреплённых за текущим идентификатором потока, множестве очередей. Первая попытка поиска, позволяет существенно снизить количество коллизий, в то время как следующие итерации должны выполнять поиск среди половины множества всех очередей. Данные действия повышают корректность и уменьшают вероятность выбора пустых очередей.

```
1   firstIteration = true;
2   do
3     if firstIteration then
4       (q1, q2) = TwoRandomQueues(p_i, p_i+k);
5     else
6       if p_i ∈ {0, 1 … p/2} then
          (q1, q2) = TwoRandomQueues(0, kp/2);
        else
          (q1, q2) = TwoRandomQueues(kp/2+1, kp);
        end
7     end
8     firstIteration = false;
9     q = GetMaxElementQueue(q1, q2);
10  while TryLock(q) == false;
11  DeleteMax(q);
12  Unlock(q);
```

Рисунок 7 – Оптимизация времени выполнения операции удаления максимального элемента на основе метода закрепления идентификатора потока к заданным очередям

На строке 3 показана проверка первой итерации цикла, это необходимо для справедливой стратегии удаления максимального элемента. Если все потоки будут работать только с, закреплёнными к их идентификатору, очередями, то точность операции может быть существенно снижена. Функция TwoRandomQueues на строке 4 возвращает пару очередей из диапазона $pi \dots pi + k$ в соответствии с идентификатором потока.

2.2 Алгоритм балансировки элементов в Multiqueue

Некоторые очереди могут заполняться существенно быстрее, чем другие. Это вызывает проблемы производительности и точности выполнения операций, потому что подмножество очередей закреплены заданным

идентификаторам потока. Это означает, что некоторые потоки будут выдавать лучшую точность и снижение производительности, в то время как оставшиеся потоки будут выдавать худшие результаты быстрее. Более того пустые очереди непригодны для выполнения операции удаления, что повышает вероятность возникновения состязания потоков.

На рисунке 8, представлен алгоритм балансировки количества элементов в распределённых последовательных очередях Multiqueue. Данный алгоритм может быть запушен после выполнения n количества операций вставки и удаления. Количество операций n должно быть вычислено эмпирически в зависимости от поставленных задач. Данный алгоритм перераспределяет количество элементов среди очередей.

```
1   q1 = FindLargestQueue();
2   q2 = FindShortestQueue();
3   if size(q1) > AvgSizeOfAllQueues()*α then
4     Lock(q1);
5     q2IsLocked = LockWithTimeout(q2);
6     if q2IsLocked then
7       sizeToTransfer = q1.size()*ω
8       TransferElements(q1, q2, sizeToTransfer)
9       Unlock(q2);
10    end
11    Unlock(q1);
12  end
```

Рисунок 8 – Балансировка количества элементов в распределённых последовательных очередях с приоритетом

В строке 1 и 2 идёт поиск самой большой и самой маленькой очереди для операции трансфера элементов. В строке 5 установлена блокировка с задержкой во избежание возникновения тупиковой ситуации. Условие выполнения балансировки α и процент перемещения элементов ω от наибольшей по

численности элементов очереди к меньшей, также должно быть вычислено эмпирически в зависимости от постановки задач.

2.3 Методы построения потокобезопасных структур данных с ослабленной семантикой выполнения операций на основе циклических списков

Циклическая ослабленная структура данных, содержит циклический список узлов. Каждый узел содержит указатель на последовательную структуру данных. Данная структура имеет мелкозернистые (fine-grained) блокировки на каждую отдельную последовательную структура. Каждый поток может выполнять операции чтения без блокировки. Это позволяет потокам выбирать подходящую последовательную структуру и подготавливать выполнение операций записи над ними для ускорения времени выполнения. Операции записи требуют блокировки последовательной структуры, чтобы не нарушить корректность параллельного исполнения программы и целостность данной структуры.

На рисунке 9 изображена циклическая потокобезопасная очередь с приоритетом (Circular relaxed concurrent priority queue, CPQ).

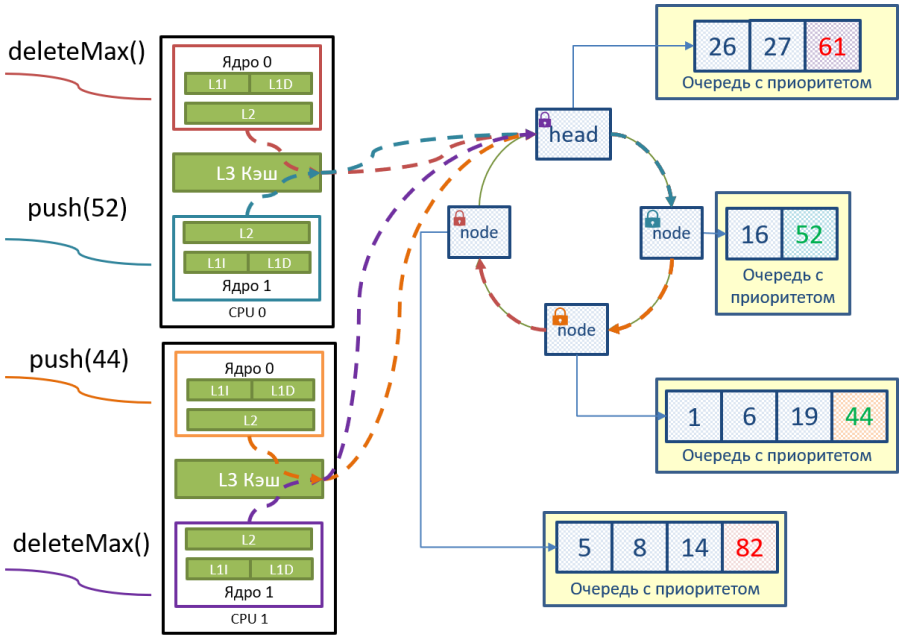


Рисунок 9 – Потокобезопасная очередь с приоритетом с ослабленной семантикой выполнения операций на основе циклических списков

Данный подход к построению потокобезопасных структур данных с ослабленным требованием к семантике выполнения операций легко масштабируется на любые последовательные структуры, такие как: стек, массив, дерево, графы и другие. Достаточно реализовать интерфейс необходимой структуры и заменить реализацию узла списка.

2.4 Алгоритмы ослабленной очереди с приоритетом на основе циклического списка

Представленные ниже алгоритмы относятся к потокобезопасной очереди с приоритетом с ослабленной семантикой выполнения операций на основе циклических списков. Несмотря на это, реализация для других последовательных структур данных не будет сильно отличаться. Каждый узел циклического списка содержит в себе указатель на последовательную структуру и предоставляет интерфейс для доступа к ней. Теоретически в списке могут находится узлы с разными структурами, в зависимости от потребностей решаемой задачи.

2.4.1 Алгоритм вставки

Изображённый на рисунке 5 алгоритм, демонстрирует операцию вставки в циклическую очередь с приоритетом. Алгоритм циклически проходит по каждому узлу списка и ищет первый незаблокированный другим потоком узел. Если такой узел найден, то поток осуществляет вставку элемента в текущий узел и завершает исполнение цикла. В ином случае, когда поиск по циклу узлов не дал результатов, алгоритм создаёт новый узел и прикрепляет его к потокобезопасной циклической очереди.

```
1   head = CPQHead();
2   node = head;
3   do
4     if TryLock(node) == true then
5       PushElement(node, el);
6       Unlock(node);
7       return;
8     end
9     node = NextNode(node);
10  while node != head;
11  node = CreateNewNode(head);
12  Lock(node);
13  PushElement(node, el);
14  Unlock(node);
```

Рисунок 10 – Вставка элементов в очередь с приоритетом на основе циклических списков

В алгоритме нет указания на определённую структуру данных, так как реализация сокрыта за интерфейсами узлов.

2.4.2 Алгоритм удаления максимального элемента

В структурах данных с ослабленным требованием к семантике выполнения операций, результат операции может отличаться от прогнозируемого, однако конечный результат должен быть близок к ожидаемому [22]. Принцип квази-линеаризуемости устанавливает, что вычисленный максимальный элемент должен быть одним из k максимальных элементов в данной очереди. Это означает, что алгоритм, представленный на рисунке 11, может предоставить не самый максимальный элемент всей циклической очереди, но он должен быть максимальным элементом отдельно взятой последовательной очереди.

```
1   head = CPQHead();
2   node = NextNode(node);
3   priorValue = GetTopValue(node);
4   priorNode = node;
5   do
6     value = GetTopValue(node);
7     if value > priorValue then
8       priorValue = value
9       priorNode = node;
10    end
11    node = NextNode(node);
12  while node != head;
13  Lock(priorNode);
14  DeleteMax(priorNode);
15  Unlock(priorNode);
```

Рисунок 11 – Удаление максимального элемента из очереди с приоритетом на основе циклических списков

Чтобы обеспечить лучшую точность выполнения операции максимального элемента, алгоритм проверяет максимальные элементы каждого узла циклической очереди циклом на строках 5 и 12. Затем на строке 13 происходит блокировка мьютекса очереди и удаление максимального элемента.

3 Программный инструментарий и отладка программного обеспечения

Данная глава описывает особенности реализаций данных алгоритмов на языках программирования. Ниже представлены реализации классов, интерфейсов и методов. Разработка данного проекта велась на двух языках программирования Kotlin и C++. Язык программирования Kotlin, который в свою очередь является строготипизированым, способен совмещать объектно-ориентированный и функциональный подход к программированию, вследствие программы имеют меньшее количество связей классов и методов. Популярность данного языка обусловлена работой на виртуальной машине Java (JVM), что позволяет использовать классы и функции при написании программ на Java, Scala, Closure и многих других, работающих на данной виртуальной машине языков. C++ - один из самых производительных языков программирования в мире. Он не заменим при разработке параллельных программ, специализирующихся на достижении максимальной производительности доступного аппаратного обеспечения. Реализация проекта представлена на двух языках, так как для проведения экспериментального исследования требуется показать максимальные возможности данных ослабленных очередей. В то же время, реализация на языке Kotlin позволяет внедрять данное программное обеспечение в корпоративные решения, так как большинство корпоративных программных продуктов разрабатывается на языке Java.

При реализации параллельных программ, рекомендуется использовать привязку потоков (thread affinity) к заданным ядрам процессора, это позволяет минимизировать сложность смены контекста между потоками [23].

3.1 Реализация оптимизированных алгоритмов Multiqueue

Multiqueue – является абстракцией предоставления доступа к множеству очередей с приоритетом. Данная структура должна предоставлять шаблонную типизацию очереди для взаимодействия с различными типами объектов в

зависимости от поставленных задач. Элементами очереди могут быть как примитивы (числа, строки), так и классы.

Конечный программный продукт должен предоставлять следующие интерфейсы со следующим описанием работы:

– void insert(const T insertElement) – операция вставки оригинальным алгоритмом. В цикле происходит поиск случайной очереди из множества, до тех пор, пока не найдётся мьютекс незаблокированный другим потоком. После захвата мьютекса очереди, элемент вставляется в неё;

– void insertByThreadId(const T insertElement) – операция вставки оптимизированным алгоритмом. В первую очередь вычисляется идентификатор текущего потока, затем вычисляется к какой половине множества принадлежит данный идентификатор. Далее в цикле происходит поиск случайной очереди среди половины множества, чей мьютекс не заблокирован и, если потоку удалось захватить мьютекс, элемент вставляется в данную очередь;

– T deleteMax() – операция удаления максимального элемента оригинальным алгоритмом. В цикле выбирается два случайных идентификатора очередей, затем их максимальные элементы сравниваются и вычисляется идентификатор с максимальным элементом. Далее идёт попытка захвата мьютекса найденной очереди и если захват успешен, то элемент удаляется, иначе цикл поиска начинается сначала;

– T deleteMaxByThreadId() – Операция удаления максимального элемента оптимизированным алгоритмом. Как и в оптимизированном алгоритме вставки в первую очередь вычисляет идентификатор потока и к какой половине множества очередей у потока имеется доступ. Затем случайным образом выбираются два идентификатора очередей и сравниваются их максимальные элементы, далее процесс происходит как в оригинальном алгоритме;

– T deleteMaxByThreadOwn() – Операция удаления максимального элемента альтернативно оптимизированным алгоритмом. Изначально вычисляется идентификатор потока, на его основе вычисляется диапазон идентификаторов очередей, среди которых будет осуществляться первый поиск

идентификаторов очередей. Среди них идёт выбор очереди с наибольшим элементом и затем идёт попытка захвата мьютекса, если мьютекс захватить не удалось, алгоритм начинает работать как deleteMaxByThreadId;

– void balance() – запуск алгоритма сортировки. При старте вычисляются по размеру минимальная и максимальная очередь, затем если размер максимальной очереди превосходит некоторый процент среднего размера всех очередей, алгоритм начинает перестановку элементов. Так как данный алгоритм запускается спустя некоторое количество операций над структурами, следует учитывать, что состояние объекта Multiqueue может быть любым, вследствие требуется обеспечить надёжность и потокобезопасность выполнения данного метода.

3.2 Реализация потокобезопасной очереди с приоритетом на основе циклического списка CPQ

Потокобезопасная структура данных с ослабленной семантикой выполнения операций на основе циклического списка содержит два основных компонента: циклический односвязный список – хранящий в себе указатель на стартовый узел и предоставляющий основной интерфейс; узел – элемент, скрывающий основную реализацию структуры данных, которую требуется разработать.

Каждый узел хранит указатель на внутреннюю Узел должен предоставлять все основные интерфейсы требуемой структуры, в данном разделе в качестве примера была взята очередь с приоритетами, в таком случае интерфейс узла будет содержать следующие методы:

– Node<T> createNewNext() – создание нового узла списка, который присоединяется к стартовому элементу и перезаписывает указатель на следующий элемент для сохранения целостности;

– void push(T el) – вставка элемента во внутреннюю структуру;

– T pop() – удаление максимального элемента из внутренней очереди;

– bool isEmpty() – проверка является ли внутренняя структура пустой;

– int size() – функция возвращает количество элементов структуры;

Сама структура представляет абстракцию над циклическим списком, где узлом выступает класс, описанный выше. Далее описан интерфейс и реализация алгоритмов циклического списка:

– void push(T el) – вставка элемента в очередь. Алгоритм проходит по всем узлам циклического списка до того момента, пока не встретит стартовый элемент. Если во время поиска был найден, незаблокированный другим потоком, узел, то вставка осуществляется в данный узел, в ином случае создаётся новый узел функцией createNewNext;

– T pop() – алгоритм удаления максимального элемента из структуры. Функция выполняет поиск по всем узлам циклического списка и записывает указатель на узел с наибольшим элементом в данной структуре. После нахождения необходимого узла, происходит удаление элемента из очереди и если структура после выполнения данной операции оказывается пуста, она удаляется из циклического списка;

– T top() – операция поиска максимального элемента. Алгоритм проходит по всем элементам списка и запрашивает максимальный элемент у каждого узла.

3.3 Детали реализации на языках программирования

Для решение определённых бизнес задач, выбирают определённый язык программирования. В разных областях IT используются разные языки, т.к. в одном случае необходимо разработать прототип для обработки больших массивов данных и проверить бизнес теорию, обычно для такой цели используется Python, в других же случаях требуется сделать динамическую веб-страницу и в этом случае на помощь приходят JavaScript и TypeScript.

Для разработки программного обеспечения, которое было бы способно на максимум использовать представленное аппаратное обеспечение, чаще всего используют C++. В тоже время для разработки огромных корпоративных проектов, в настоящий момент в большинстве случаев используется Java.

В настоящей работе требуется разработать программное обеспечение как для широкого использования в корпоративных приложениях, так и для достижения максимальной производительности, именно поэтому в качестве языков программирования для реализации были выбраны C++ и Kotlin.

3.3.1 C++

Для разработки сложного программного обеспечения на языке высокого уровня C++ не обойти без использования подключаемых библиотек. Для сборки проекта используется CMake, который позволяет настраивать разные версии сборок, подключать зависимости и настраивать ход сборки. Стандартная библиотека шаблонов (STL) не предоставляет необходимый набор функциональности. Поэтому в качестве вспомогательной библиотеки, было решено добавить Boost. Boost – это набор готовых реализаций различных структур данных, механизмов синхронизации, алгоритмов над графами и др.

В качестве базовой структуры данных для Multiqueue в библиотеке Boost было выбрано B-tree (boost::heap::d_ary_heap). Это n-арное дерево, предоставляющее интерфейс очереди с приоритетом. Данная очередь не обладает потокобезопасностью, поэтому все необходимые синхронизации требуется реализовывать в классе Multiqueue. Каждая внутренняя очередь хранится в массиве очередей, тем самым индекс очереди в массиве является уникальным идентификатором очереди. Также имеется массив мьютексов для обеспечения блокировок каждой отдельной очереди.

В конструкторе класса Multiqueue задаются следующие параметры:

– numOfThreads – количество потоков исполняемой параллельной программы;

– numOfQueuesPerThread – количество очередей на один поток, для создания внутреннего пула очередей с приоритетом.

Оптимизация алгоритмов в большей степени зависит от идентификатора потока. Для идентификации номера потока используется метод из STL std::this_thread::get_id(), он возвращает уникальный идентификатор, который

записывается в словарь потоков threadMap, на основе которого вычисляется номер идентификатора потока в реализации.

3.3.2 Kotlin

При разработке на языке Kotlin следует учитывать особенности и возможности данного языка. Он является строготипизированным, объектно-ориентированным и функциональным языком программирования одновременно.

Для сборки и подключения дополнительных зависимостей используется Gradle. Языком скрипта сборки также является Kotlin. В качестве дополнительных библиотек используется Junit для написания юнит тестов.

Kotlin позволяет разрабатывать параллельные программы как с использованием потоков системы (thread), так и с использованием легковесных потоков (coroutine).

В качестве базовой очереди с приоритетом была взята реализация PriorityQueue из стандартной библиотеки java.util. Она предоставляет все необходимые интерфейсы для реализации абстракции пула очередей с приоритетом. Механизмы потокобезопасности отсутствуют у данной очереди, поэтому требуется реализовать потокобезопасность на уровне класса Multiqueue.

3.4 Экспериментальные исследования

Данный раздел включает описание и сравнительный анализ реализаций ослабленных потокобезопасных структур данных.

3.4.1 Метрики

В качестве метрики был выбран показатель пропускной способности b выполнения операций. Общая пропускная способность высчитывается как сумма индивидуальных пропускных способностей потоков (7), где n количество операций вставки/удаления потока i, а t – время выполнения операций [24].

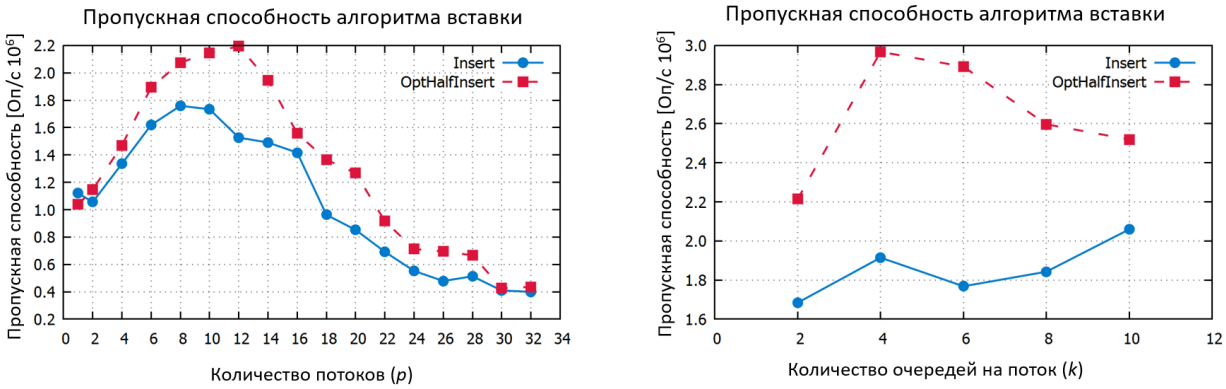
$$b_i = n/t \quad (7)$$

Каждый тест был выполнен несколько раз. Результатом является усреднённое значение сумм пропускных способностей операций.

3.4.2 Экспериментальное исследование оптимизированных алгоритмов Multiqueue

Экспериментальное исследование, представленное в этом разделе, запускалось на одном узле, вычислительного кластера, с общей памятью. Данный узел оборудован двумя шестиядерными процессорами Intel Xeon X5670 с тактовой частотой 2.93 ГГц каждый (Hyper-Threading, HT, был отключен). Модуль оперативной памяти имел следующие характеристики: DDR3 1666 МГц 16 Гб.

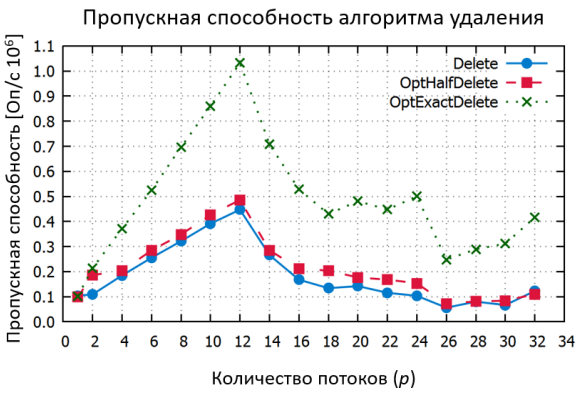
На рисунке 6а представлены графики сравнения пропускных способностей оригинальных (Insert) и оптимизированных (OptHalfInsert) алгоритмов вставки в зависимости от количества потоков, что показывает, насколько хорошо данное решение масштабируется, количество очередей на один поток фиксировано и равняется $k = 2$. На рисунке 6б изображена пропускная способность в зависимости от количества очередей на поток, количество потоков зафиксировано и равняется $p = 12$. В каждом исследовании было выполнено $n = 10^6$ операций вставки.



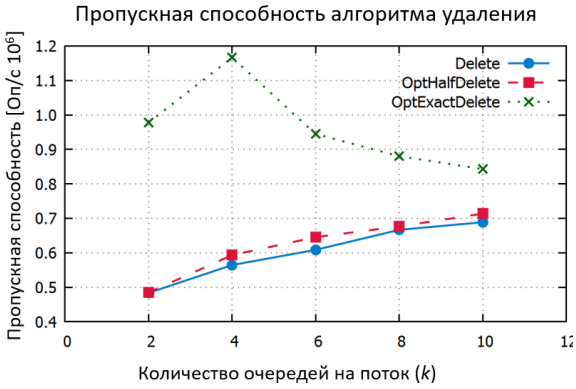
а *б*

Рисунок 12 – Сравнение пропускной способности алгоритмов вставки

На рисунке 7.а изображены графики сравнения пропускных способностей оригинальных (Delete), оптимизированных (OptHalfDelete) и альтернативно оптимизированных (OptExactDelete) алгоритмов удаления максимальных элементов в зависимости от количества потоков. Количество очередей на один поток фиксировано и равняется $k = 2$. На рисунке 7.б представлен график пропускной способности операции удаления в зависимости от количества очередей на поток, количество потоков фиксировано и равняется $p = 12$. В каждом исследовании удаления максимальных элементов было выполнено $n = 0{,}5 * 10^6$ операций.



а



б

Рисунок 13 – Сравнение пропускной способности алгоритмов удаления максимального элемента

На рисунке 8 показан график случайных операций вставки и удаления. Каждый график представляет собой комбинацию алгоритмов, что позволяет наглядно оценить какие алгоритмы имеют наибольшую пропускную способность. В каждом исследовании было выполнено $n = 10^6$ операций.

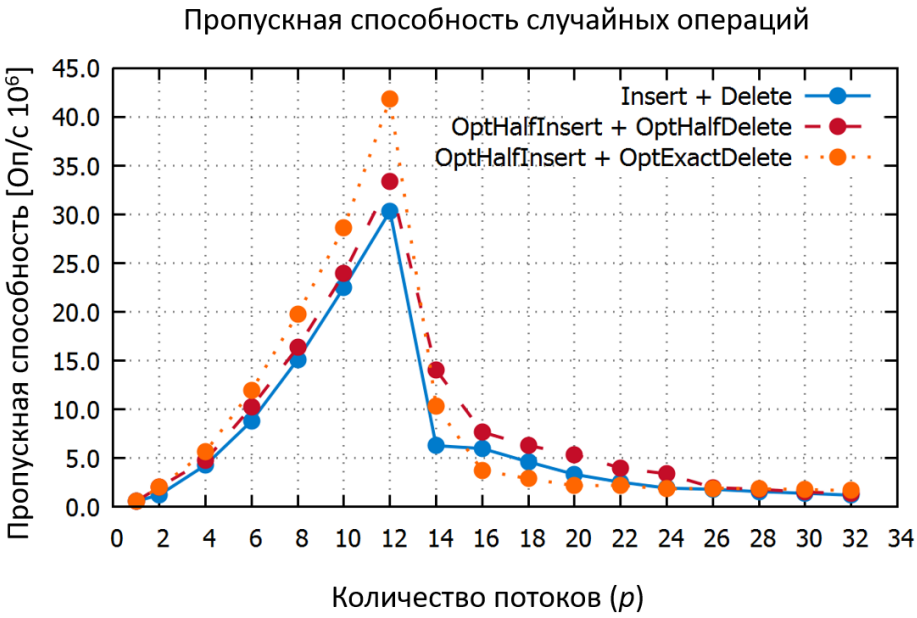


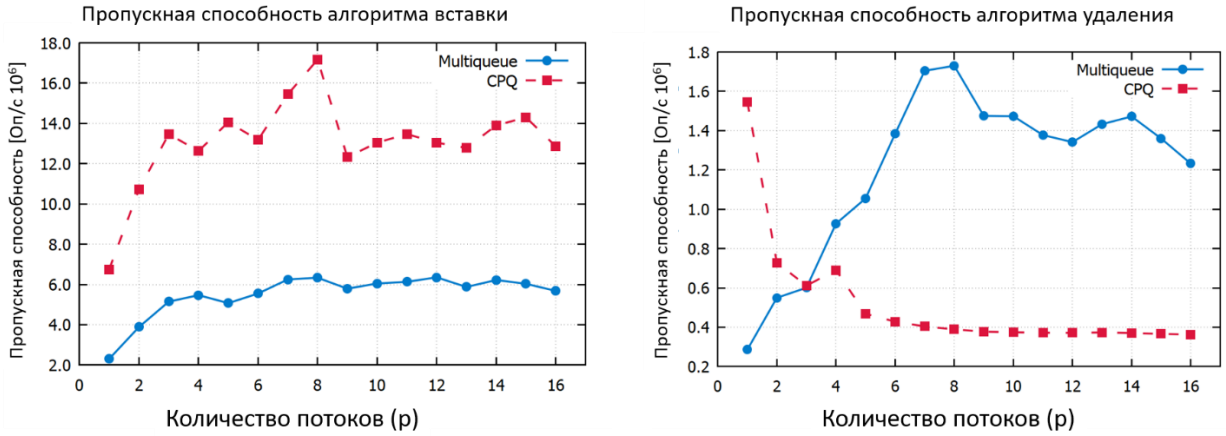
Рисунок 14 – Сравнение пропускной способности случайных операций вставки и удаления максимального элемента

Как видно из представленных графиков оптимизированные алгоритмы OptHalfInsert и OptExactDelete достигают максимальную пропускную способность при количестве потоков равному количеству ядер $p = 12$ (для данной системы) и количеству очередей на один поток $k = 4$, большее количество очередей на один поток не требуется.

3.4.3 Экспериментальное исследование алгоритмов потокобезопасной очереди с приоритетом на основе циклического списка

Экспериментальное исследование, представленное в этом разделе, было запущено на десктоп компьютере, оснащённым восьмиядерным процессором AMD Ryzen 7 3800X с 4.3 ГГц на каждое ядро (Simultaneous multi-threading, SMT, был отключён). Также использовалось два модуля оперативной памяти DDR4 3200 МГц с 16 Гб каждый.

Для сравнительного анализа использовалась ослабленная очередь с приоритетами Multiqueues с оптимизированными алгоритмами OptHalfInsert и OptExactDelete. В каждом тестировании было выполнено $n = 10^7$ операций вставки и $n = 5 * 10^6$ операций удаления максимального элемента.



а *б*

Рисунок 15 – Сравнение пропускной способности алгоритмов циклической ослабленной очереди и Multiqueues

На рисунке 9.а показано, что потокобезопасная ослабленная очередь с приоритетами на основе циклических списков, на операции вставки, показывает лучшую пропускную способность и возможность к масштабированию чем Multiqueues. Возможность к масштабируемости очень важна для систем общей памяти с большим количество центральных процессоров. Вся эффективность многопоточной программы в целом зависит от данного параметра. Однако, на рисунке 9.б, демонстрируется, что пропускная способность циклической ослабленной очереди с приоритетом сильно страдает от увеличения числа потоков, из-за возникновения состязаний потоков за единый ресурс. Это вызывает проблемы производительности, поскольку для операции удаления максимального элемента, необходимо учесть максимальный элемент каждой последовательной очереди с приоритетом, чтобы предоставить наиболее близкий к прогнозируемому - результат. Это означает что поток обязан обойти каждый узел циклического списка и проверить значение каждой последовательной структуры, после этого также необходимо время на синхронизацию и блокировку мьютекса подходящей очереди, чтобы не нарушить корректность и целостность данной структуры.

4 Составление бизнес-плана по коммерциализации результатов НИР магистранта

Параллельные вычислительные технологии сегодня применяются повсеместно – от встроенных систем и мобильных устройств до систем с массовым параллелизмом, GRID-систем и облачных ВС. Кроме того, алгоритмический и программный инструментарий параллельного программирования является базой при построении современных систем обработки больших данных, искусственного интеллекта и машинного обучения.

В ходе выполнения ВКР были разработаны оптимизированные алгоритмы ослабленной очереди Multiqueue и предложен подход построения структур данных с ослабленной семантикой выполнения операций на основе циклических списков.

Результаты выполненной работы позволяют создать коммерческую реализацию продукта. Для достижения поставленной цели были определены следующий набор задач:

– описание программного продукта;

– анализ рынка сбыта продукта и конкуренции;

– планирование маркетинговой стратегии и продаж;

– планирование работ;

– расчёт затрат на реализацию программного продукта;

– определение финансовых показателей проекта.

4.1 Описание программного продукта

Разрабатываемое программное обеспечение предназначено для внедрения в существующие и новые исходные коды программных продуктов, используемые для обработки больших массивов данных в составе программных комплексов, запускаемых на кластере суперкомпьютера. Главным преимуществом структур данных с ослабленной семантикой над последовательными

структурами является высокая способность к масштабируемости, за счёт отсутствия единой точки выполнения операций.

Основные характеристики предложенного подхода к созданию ослабленных структур данных на основе циклических списков:

– простота разработки и внедрения;

– надёжность;

– высокая производительность;

Результаты выполнения операции могут отличаться от прогнозируемых.

Реализации продукта представлены как на языке программирования C++, так и на Kotlin, что позволяет расширить число пользователей данного фреймворка.

В качестве коммерческого решения предлагается разработка и поддержка большего числа альтернатив последовательных структур данных.

4.2 Анализ рынка сбыта продукта и конкуренции

Исследование рынка сбыта состоит из различных коммерческих организаций, которым в своих многопоточных приложениях необходимо повышать пропускную способность потокобезопасных структур данных. Данное требование вытекает из ограничений, связанных с физическими возможностями пропускных способностей оперативной памяти и центрального процессора.

Среди основных сегментов сбыта программного продукта выделяются следующие:

– коммерческие IT организации, разрабатывающие высокопроизводительные многопоточные программы;

– военные предприятия, которые разрабатывают программные продукты для обработки сверхбольших массивов данных;

– прочие организации, заинтересованные в использовании многоядерных и кластерных вычислительных машин.

Потребителями продукта можно обозначить организации, где производительность вычислительной многоядерной машины играют важную роль. К продукту предъявляются следующие требования:

– надёжность и отказоустойчивость;

– простота внедрения программного решения в существующие продукты;

– высокая производительность;

Спрос на данное программное обеспечение будет расти в связи с ростом популярность многоядерных и многопроцессорных вычислительных машин от смартфонов до суперкомпьютеров.

4.3 Маркетинговая стратегия и план продаж

Основная цель первого года производства заключается в том, чтобы зарекомендовать себя надежным поставщиком ПО.

Основными задачами ценообразования являются:

– покрытие расходов на разработку;

– обеспечение прибыли, достаточной для бесперебойного функционирования организации;

– развитие и поддержка данного программного продукта.

Цена годовой лицензии на использование фреймворка для обработки больших массивов данных составляет 149 000 рублей. Выпуск обновлений, обращение в техническую поддержку за консультациями по работе системы, помощь с первоначальной установкой и настройкой включены в стоимость годовой лицензии.

Исходя из анализа рынка, затраченных ресурсов и заинтересованности пользователя в данном программном продукте цена является обоснованной и конкурентоспособной.

Интерес к продукту рассчитывается привлечь за счет следующих маркетинговых мероприятий:

– прямых связей с ключевыми покупателями;

– рекламы через компьютерные сети (например, в форме статей на сайтах habr.ru и vc.ru);

– персональные продажи;

– стимулирование продаж;

– реклама на конференциях.

В течении шести месяцев осуществляется разработка и отладка данного фреймворка. Таким образом осуществление продаж стартует с началом третьего квартала реализации проекта.

План продаж спрогнозирован по методу безубыточности с учетом следующих факторов:

– характеристики фактической емкости рынка и уровня спроса (в натуральных и стоимостных показателях);

– основные тенденции рынка (рост, насыщение, спад);

– привычки и предпочтения покупателей;

– уровень безубыточности;

– стратегию маркетинга.

План продаж представлен в таблице 1.

Таблица 1 — План продаж

Показатели	**Квартал**				**Всего**
	I	**II**	**III**	**IV**	
Ожидаемый объём продаж, ед.	0	0	15	55	**70**
Цена с НДС, руб.	0	0	159 000	159 000	**318 000**
Выручка с НДС, руб.	0	0	2 385 000	8 745 000	**11 130 000**
Нетто-выручка (без НДС), руб.	0	0	1 987 500	7 287 500	**9 275 000**
Сумма НДС, руб.	0	0	397 500	1 457 500	**1 855 000**

Согласно значениями показателей ожидается продажа 70 лицензий за первый год работы над проектом, которая в сумме валовая выручку в размере 9 275 000 рублей.

По плану продаж построен график ожидаемых денежных поступлений, который отражен в таблице 2.

Таблица 2 — График ожидаемых денежных поступлений

Показатели	**Квартал**				**Всего**
	I	**II**	**III**	**IV**	
Остаток денежных средств на начально периода, руб.	0	0	0	0	0
Продажи за квартал	0	0	2 385 000	8 745 000	11 130 000
I	0	-	-	-	0
II	-	0	-	-	0
III	-	-	2 385 000	-	2 385 000
IV	-	-	-	8 745 000	8 745 000
Всего поступлений денежных средств, руб.	**0**	**0**	**2 385 000**	**8 745 000**	**11 130 000**

Сумма ожидаемых поступлений от продаж программного продукта за первый год составляет 11 130 000 рублей.

4.4 Производственный план

Для удобной разработки программного продукта команде потребуется офисное помещение и оборудование. Характеристика помещения и перечень необходимого оборудования представлены в таблице 3 и таблице 4.

Таблица 3 — Характеристика необходимых зданий и сооружений

Наименование помещения	**Площадь, м^2**	**Условия получения**	**Стоимость, руб.**	**Период, мес.**	**Всего**
Административные	40	аренда	30000	12	**360000**

Затраты на аренду офисного помещения составляют 360 000 рублей в год.

Таблица 4 — Перечень необходимого оборудования

Показатель	**Кол-во**	**Способ получения**	**Стоимость, руб.**		**Итого вкл. НДС**
			Без НДС	**НДС**	
Персональный компьютер	4	Покупка	124800	24960	**599040**
Монитор	4	Покупка	21000	4200	**100800**
Клавиатура и Мышь	4	Покупка	2350	470	**11280**
Стол	4	Покупка	2520	504	**12096**
Стул	4	Покупка	2050	410	**9840**
МФУ	1	Покупка	7450	1490	**8940**
Всего	**22**	**Покупка**	**165570**	**33114**	**748476**

Суммарная стоимость всего оборудования составляет 748 476 рублей. Потребность в материально-производственных запасах отсутствует, так как для разработки программного обеспечения достаточно оборудования, указанного в таблице.

Проектная команда состоит из двух программистов: на языках программирования Kotlin и C++, тестировщика, проектного и аккаунт менеджеров. Последний вступает в должностные обязанности на последнем месяце третьего квартала. Фонд оплаты труда на 12 месяцев отображен в таблице 5.

Таблица 5 — Фонд оплаты труда

Показатели		**Заработная плата по кварталам**				**Всего**
Должность	**Кол-во**	**I**	**II**	**III**	**IV**	
Программист	2	390 000	390 000	390 000	390 000	1 560 000
Тестровщик	1	135 000	135 000	135 000	135 000	540 000
Проектный менеджер	1	165 000	165 000	165 000	165 000	660 000
Аккаунт менеджер	1	0	0	120 000	120 000	240 000
Всего выплат	**-**	**690 000**	**690 000**	**810 000**	**810 000**	**3 000 000**

Согласно результатам расчетов, представленным в таблице. Фонд оплаты труда на год составляет 3 000 000 рублей. Теперь необходимо учесть 40% накладных расходов и 30.2% отчислений с заработных плат сотрудников организации.

Накладные расходы покрывают:

– аренду и содержание помещений;

– амортизацию основных средств;

– заработную плату административно-управленческого персонала;

– создание нормальных условий труда;

– затраты на рекламу;

– затраты на почту, телефон, интернет;

– затраты материалы для офиса.

Результаты расчетов накладных расходов и отчислений на социальные нужды представлены в таблице 6.

Таблица 6 — Отчисления на социальные нужды и накладные расходы

Показатели	**Квартал**				**Всего**
	I	**II**	**III**	**IV**	
Фонд оплаты труда	690000	690000	810 000	810 000	3 000 000
Социальные отчисления	208380	208380	244 620	244 620	906 000
Накладные расходы	276000	276000	276 000	276 000	1 104 000
Всего затрат	**1174380**	**1174380**	**1 330 620**	**1 330 620**	**5 010 000**

Сумма затрат на фонд оплаты труда, социальные отчисления и накладные расходы составляет 5 010 000 рублей.

Таким образом, суммируя расходы на оборудование, фонд оплаты труда, отчисления на социальные нужды и накладные расходы за первые два квартала, для реализации проекта потребуются инвестиции в размере 3 100 000 рублей. Было принято решение привлечь данную сумму за счёт инвесторов. В качестве дивидендов инвестор будет получать 20% от чистой

(нераспределенной) прибыли. Выплаты дивидендов начнутся в первом квартале следующего года.

4.5 Показатели экономической эффективности и оценка рисков

План прибылей и убытков представляет собой финансовый документ, в котором отражаются доходы, расходы и финансовые результаты за период реализации проекта. Данный документ представлен в таблице 7.

Основная задача формирования этого документа состоит в том, чтобы рассчитать размер и структуру себестоимости продукции, соотношение затрат и результатов деятельности за определенный период, по которым можно будет судить о рентабельности проекта, его окупаемости.

Таблица 7 — План прибылей и убытков

Показатели	**Квартал**				**Всего**
	I	**II**	**III**	**IV**	
Выручка-нетто (без учета НДС) от реализации	0	0	2 385 000	8 745 000	11 130 000
Фонд оплаты труда	690 000	690 000	810 000	810 000	3 000 000
Социальные отчисления	208 380	208 380	244 620	244 620	906 000
Накладные расходы	276 000	276 000	276 000	276 000	1 104 000
Прибыль до налогообложения	0	0	1 054 380	7 414 380	8 468 760
Налог на прибыль	0	0	210 876	1 482 876	1 693 752
Чистая (нераспределенная) прибыль	-1 174 380	-1 174 380	843 504	5 931 504	4 426 248

Согласно таблице чистая (нераспределенная) прибыль составит 4 426 248 рублей. Данное значение потребуется для расчета показателей экономической эффективности.

Денежные потоки от операционной (текущей), инвестиционной и финансовой деятельности на планируемый период отражены в таблице 8.

Таблица 8 — Движение денежных средств

Показатели	**Квартал**				**Всего**
	I	**II**	**III**	**IV**	
1. Остаток денежных средств на начало периода	3 100 000	1 177 144	2 764	448 768	**4 728 676**
2. Поступление денежных средств					
2.1. Поступления от продажи продукции	0	0	2 385 000	8 745 000	**11 130 000**
2.2. Прочие поступления	0	0	0	0	**0**
3. Всего наличие денежные средств (1 + 2)	3 100 000	1 177 144	2 387 764	9 193 768	**15 858 676**
4. Выбытие денежных средств (платежи)					
4.1. Оплата труда	690 000	690 000	810 000	810 000	**3 000 000**
4.2. Накладные расходы	276 000	276 000	276 000	276 000	**1 104 000**
5. Инвестиции (покупка оборудования)	748 476	0	0	0	**748 476**

Продолжение Таблицы 8

6. Уплата налогов					
6.1. НДС	0	0	397 500	1 457 500	**1 855 000**
6.2. Отчисления на социальные нужды	208 380	208 380	244 620	244 620	**906000**
6.3. Налог на прибыль	0	0	210 876	1 482 876	**1 693 752**
6.4. Прочие налоги	0	0	0	0	**0**
7. Всего оттоки (4 + 5 + 6)	1 922 856	1 174 380	1 800 668	4 069 329	**8 967 233**
8. Чистый денежный поток (3–7)	1 177 144	2 764	437 096	4 562 767	**6 179 771**

Для оценки экономической эффективности проекта используются следующие показатели:

– рентабельность инвестиций;

– чистая текущая стоимость проекта;

– срок окупаемости проекта.

Показатель рентабельности инвестиций ROI (Return On Investments) определяется как отношение среднегодовой прибыли к суммарным инвестиционным затратам в проекте:

$$ROI = \frac{\sum_{t=1}^{T} P_t}{I},$$

где P_t – чистая прибыль от проекта в году t, T – количество лет в инвестиционном периоде; I – величина инвестиционных затрат, связанных с реализацией проекта.

$$ROI = \frac{4\,426\,248}{3\,100\,000} \cdot 100\% = 143\%$$

Показатель рентабельности инвестиций за первый год составляет 143%.

Чистая текущая стоимость проекта NPV (Net Present Value) рассчитывается по формуле:

$$NPV = \sum_{t=1}^{T} \frac{NCF_t}{(1+R)^t} - I_0,$$

где I_0 – единовременные затраты, совершаемые в инвестиционном (нулевом) интервале; $NCF_t = (CIF_t - COF_1)$ – чистый денежный поток.

$$NPV = -\frac{3\,100\,000}{(1+0{,}05)^0} + \frac{448\,768}{(1+0{,}05)^3} + \frac{4\,922\,772}{(1+0{,}05)^4} = 1\,337\,639,$$

Исходя из формулы чистая текущая стоимость продукта равна 3 038 160 рублей.

Далее рассчитаем срок окупаемости проекта по формуле:

$$T_{\text{ок}} = \frac{I}{\sum_{t=1}^{T} P_t},$$

где $T_{\text{ок}}$ – срок окупаемости проекта; P_t – чистая прибыль от проекта в году t; I – величина инвестиционных затрат, связанных с реализацией проекта.

$$T_{\text{ок}} = \frac{3\,100\,000}{4\,426\,248} = 0{,}7$$

Срок окупаемости проекта составит 0,7 года, примерно 8,5 месяцев. Однако продажи продукта стартовали с началом третьего квартала, следовательно, срок окупаемости равен 14,5 месяцев.

Для оценки риска проекта применим метод чувствительности. В качестве варьируемых факторов выбраны цена продукта и объем продаж. Рассмотрим, как изменится чистая текущая стоимость проекта при снижении цены на 10% и 15% соответственно. В таблице 9 представлены результаты расчетов.

Таблица 9 — Зависимость NVP от снижения цены.

Снижение цены, %	**NPV, руб**
0	1 337 639
10	627 234
15	272 031

Теперь рассчитаем значение NVP при снижении объемов продаж до 60 шт. и 55 шт. соответственно. Результат расчетов представлен в таблице 10.

Таблица 10 — Зависимость NVP от снижения объемов продаж

Объем продаж, шт	**NPV, руб**
70	1 337 639
60	509 178
55	94 947

Так как показатели чистой текущей стоимости проекта остаются положительными при максимальном рассчитанном снижении цены или объемов продаж, то проект считается финансово устойчивым.

Стоит учитывать, что значения приблизительные и могут колебаться в пределах 10-15%.

4.6 Выводы по главе

В данном разделе было произведено технико-экономическое обоснование разработки потокобезопасных структур данных с ослабленной семантикой выполнения операций.

Основными потребителями продукта являются:

– коммерческие IT организации, разрабатывающие высокопроизводительные многопоточные программы.

– военные предприятия, которые разрабатывают программные продукты для обработки сверхбольших массивов данных.

– прочие организации, заинтересованные в использовании многоядерных и кластерных вычислительных машин.

Для реализации проекта требуется 3 100 000 рублей первоначальных инвестиций. Покрытие данной суммы было решено произвести за счет инвесторов.

Чистая (нераспределенная) прибыль за первый год составила 3 949 576 рублей.

Для оценки экономической эффективности проекта были выбраны следующие показатели:

– рентабельность инвестиций;

– чистая текущая стоимость проекта;

– срок окупаемости проекта.

Результаты расчетов данных показателей представлены в таблице 11.

Таблица 11 — Экономические показатели эффективности проекта

Показатель	**Значение**
NPV	1 337 639 руб.
ROI	143%
T	14,5 мес.

Оценка рисков проекта была произведена с применением анализа чувствительности. По результатам оценки рисков проект является финансово устойчивым.

Таким образом проект по потокобезопасных структур данных с ослабленной семантикой выполнения операций считается целесообразным.

ЗАКЛЮЧЕНИЕ

Исследования потокобезопасных структур данных с ослабленными требованиями к семантике выполнения операций показывают, что существует достаточно много нерешённых задач в данной области параллельного программирования. Рассмотренные работы демонстрируют преимущества данного подхода к масштабируемости многопоточных программ, однако они не лишены своих недостатков в виде неточного результата выполнения, которым при достаточном объёме данных можно пренебречь.

Разработана оптимизированная версия ослабленной потокобезопасной очереди с приоритетом на основе Multiqueue. Разработанные алгоритмы вставки и удаления показывают прирост производительности в 1,2 и 1,6 раза соответственно, по сравнению с оригинальными алгоритмами вставки и удаления. Оптимизация достигается за счёт уменьшения количества коллизий на основе ограничения диапазона выбора случайной структуры.

Предложен подход к построению ослабленных структур данных на основе циклических списков. На основе данного подхода была разработана циклическая потокобезопасная ослабленная очередь с приоритетом. Данная структура демонстрирует высокую возможность масштабирования с достаточно хорошей точностью результата выполнения операций. Подход построения можно экстраполировать на такие структуры данных как: стеки, очереди, графы и другие.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Табаков А. В., Пазников А. А. Алгоритмы оптимизации потокобезопасных очередей с приоритетом на основе ослабленной семантики выполнения операций //Известия СПбГЭТУ «ЛЭТИ. – 2018. – С. 42-49.
2. TOP500 supercomputers list [Электронный ресурс] // Top500 URL: https://www.top500.org/news/summit-up-and-running-at-oak-ridge-claims-first-exascale-application (Дата последнего посещения 7 ноября 2019).
3. Anenkov A. D., Paznikov A. A., Kurnosov M. G. Algorithms for access localization to objects of scalable concurrent pools based on diffracting trees in multicore computer systems //2018 XIV International Scientific-Technical Conference on Actual Problems of Electronics Instrument En-gineering (APEIE). – IEEE, 2018. – С. 374-380.
4. Blumofe R. D., Leiserson C. E. Scheduling multithreaded computations by work stealing //Journal of the ACM (JACM). – 1999. – Т. 46. – №. 5. – С. 720-748.
5. Goncharenko E. A., Paznikov A. A., Tabakov A. V. Performance Modeling of Atomic Opera-tions in Control Systems Based on Multicore Computer Systems //2019 III International Confer-ence on Control in Technical Systems (CTS). – IEEE. – С. 140-143.
6. Goncharenko E. A., Paznikov A. A., Tabakov A. V. Evaluating the performance of atomic opera-tions on modern multicore systems //Journal of Physics: Conference Series. – IOP Publishing, 2019. – Т. 1399. – №. 3. – С. 033107.
7. Herlihy M., Shavit N. The art of multiprocessor programming. – Morgan Kaufmann, 2011. – С. 219-225
8. Schenkel R. Locking Granularity and Lock Types //Encyclopedia of Database Systems. – Spring-er, 2009. – С. 1641-1643.
9. Cederman D. et al. Lock-free concurrent data structures //Programming Multicore and Many-core Computing Systems. – 2017. – Т. 86. – С. 59

10. Кулагин И. И. Средства архитектурно-ориентированной оптимизации выполнения парал-лельных программ для вычислительных систем с многоуровневым параллелизмом: дис. – СибГУТИ. Новосибирск, 2018. – С. 77-82.
11. Shavit N., Touitou D. Software transactional memory //Distributed Computing. – 1997. – Т. 10. – №. 2. – С. 99-116.
12. Paznikov A. A., Smirnov V. A., Omelnichenko A. R. Towards Efficient Implementation of Concurrent Hash Tables and Search Trees Based on Software Transactional Memory //2019 In-ternational Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon). – IEEE, 2019. – С. 1-5.
13. Herlihy M. P., Wing J. M. Linearizability: A correctness condition for concurrent objects //ACM Transactions on Programming Languages and Systems (TOPLAS). – 1990. – Т. 12. – №. 3. – С. 463-492.
14. Derrick J. et al. Quiescent consistency: Defining and verifying relaxed linearizability //International Symposium on Formal Methods. – Springer, Cham, 2014. – С. 200-214.
15. Henzinger T. A. et al. Quantitative relaxation of concurrent data structures //Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – 2013. – С. 317-328.
16. Afek Y., Korland G., Yanovsky E. Quasi-linearizability: Relaxed consistency for improved concurrency //International Conference on Principles of Distributed Systems. – Springer, Berlin, Heidelberg, 2010. – С. 395-410.
17. Talmage E., Welch J. L. Improving average performance by relaxing distributed data structures //International Symposium on Distributed Computing. – Springer, Berlin, Heidelberg, 2014. – С. 421-438.
18. Alistarh D. et al. The spraylist: A scalable relaxed priority queue //Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. – 2015. – С. 11-20.
19. Paznikov A. A., Anenkov A. D. Implementation and Analysis of Distributed

Relaxed Concurrent Queues in Remote Memory Access Model //Procedia Computer Science. – 2019. – Т. 150. – С. 654-662.

20. Wimmer M. et al. The lock-free k-LSM relaxed priority queue //ACM SIGPLAN Notices. – 2015. – Т. 50. – №. 8. – С. 277-278.
21. Rihani H., Sanders P., Dementiev R. Multiqueues: Simpler, faster, and better relaxed concurrent priority queues //arXiv preprint arXiv:1411.1209. – 2014.
22. Tabakov A., Paznikov A. Optimization of Data Locality in Relaxed Concurrent Priority Queues⋆. – 2019.
23. Paznikov A., Shichkina Y. Algorithms for Optimization of Processor and Memory Affinity for Remote Core Locking Synchronization in Multithreaded Applications //Information. – 2018. – Т. 9. – №. 1. – С. 21.
24. Tabakov A. V., Paznikov A. A. Algorithms for Optimization of Relaxed Concurrent Priority Queues in Multicore Systems //2019 IEEE Conference of Russian Young Researchers in Electri-cal and Electronic Engineering (EIConRus). – IEEE, 2019. – С. 360-365.
25. Tabakov A. V., Paznikov A. A. Modelling of Parallel Threads Synchronization in Hybrid MPI+ Threads Programs //2019 XXII International Conference on Soft Computing and Measurements (SCM)). – IEEE, 2019. – С. 197-199.
26. Amer A. et al. MPI+ threads: Runtime contention and remedies //ACM SIGPLAN Notices. – 2015. – Т. 50. – №. 8. – С. 239-248.
27. Tabakov A. V., Paznikov A. A. Using relaxed concurrent data structures for contention minimi-zation in multithreaded MPI programs //Journal of Physics: Conference Series. – IOP Publishing, 2019. – Т. 1399. – №. 3. – С. 033037.
28. Садырин И. А., Магомедов М. Н. Учебно-методическое пособие «Коммерциализация результатов НИР» // СПбГЭТУ «ЛЭТИ». – 2019.

ПРИЛОЖЕНИЕ А. Исходный код оптимизированных алгоритмов вставки и удаления multiqueues на языке программирования C++

```
template<typename T>
class Multiqueues {
    int numOfThreads;
    int numOfQueuesPerThread;
    int numOfQueues = 2;
    unsigned int *seed = new unsigned int[1];
    typedef typename boost::heap::d_ary_heap<T, boost::heap::mutable_<true>,
boost::heap::arity<2>> PriorityQueue;
    PriorityQueue *internalQueues;
    std::timed_mutex *locks;
    std::thread::id *threadsMap;
    int threadsMapSize = 0;

    int getQueueIndexForDelete(int queueIndex, int secondQueueIndex) const {
        if (internalQueues[queueIndex].empty() && internalQueues[secondQueue-
Index].empty()) {
            return -1;
        } else if (internalQueues[queueIndex].empty() && !internalQueues[secondQueue-
Index].empty()) {
            queueIndex = secondQueueIndex;
        } else {
            while (!locks[queueIndex].try_lock());
            int value = -1;
            if (!internalQueues[queueIndex].empty())
                value = internalQueues[queueIndex].top();
            locks[queueIndex].unlock();
            while (!locks[secondQueueIndex].try_lock());
            int value2 = -1;
            if (!internalQueues[secondQueueIndex].empty())
                value2 = internalQueues[secondQueueIndex].top();
            locks[secondQueueIndex].unlock();
            if (value2 > value) {
                queueIndex = secondQueueIndex;
            }
        }

        if (queueIndex != -1 && internalQueues[queueIndex].empty()) {
            return -1;
        }
        return queueIndex;
    }

    inline int getRandomQueueIndexForHalf() const { return rand_r(this->seed) % (this-
>numOfQueues / 2); }

    inline int getRandomQueueIndex() const { return rand_r(this->seed) % this-
>numOfQueues; }

    T getTopValue(const int queueIndex) const {
        int topValue = -1;
        if (!internalQueues[queueIndex].empty()) {
            topValue = internalQueues[queueIndex].top();
            internalQueues[queueIndex].pop();
        }
```

```
        locks[queueIndex].unlock();
        return topValue;
    }

    inline int getThreadId() {
        std::thread::id threadId = std::this_thread::get_id();
        for (int i = 0; i < threadsMapSize; ++i) {
            if (threadsMap[i] == threadId) {
                return i;
            }
        }
        threadsMap[threadsMapSize] = threadId;
        ++threadsMapSize;
        return threadsMapSize - 1;
    }

public:

    Multiqueues(int numOfThreads, int numOfQueuesPerThread) {
        this->numOfThreads = numOfThreads;
        this->numOfQueuesPerThread = numOfQueuesPerThread;
        this->numOfQueues = numOfThreads * numOfQueuesPerThread;
        this->threadsMap = new std::thread::id[numOfThreads];
        internalQueues = new PriorityQueue[numOfQueues];
        locks = new std::timed_mutex[numOfQueues];
    }

    ~Multiqueues() {
        for (int i = 0; i < numOfQueues; ++i) {
            internalQueues[i].clear();
        }
        delete[] internalQueues;
        delete[] locks;
    }

    void insert(const T insertNum) {
        int queueIndex;
        do {
            queueIndex = getRandomQueueIndex();
        } while (!locks[queueIndex].try_lock());
        internalQueues[queueIndex].push(insertNum);
        locks[queueIndex].unlock();
    }

    void insertByThreadId(const T insertNum) {
        const int threadId = getThreadId();
        int queueIndex;
        do {
            const int halfOfThreads = this->numOfThreads / 2;

            if (threadId < halfOfThreads) {
                queueIndex = getRandomQueueIndexForHalf();
            } else {
                queueIndex = getRandomQueueIndexForHalf() + this->numOfQueues / 2;
            }
        } while (!locks[queueIndex].try_lock());
        internalQueues[queueIndex].push(insertNum);
        locks[queueIndex].unlock();
    }
```

```
    T deleteMax() {
        int queueIndex;
        int secondQueueIndex;
        do {
            queueIndex = getRandomQueueIndex();
            secondQueueIndex = getRandomQueueIndex();
            queueIndex = getQueueIndexForDelete(queueIndex, secondQueueIndex);
        } while (queueIndex == -1 || !locks[queueIndex].try_lock());
        return getTopValue(queueIndex);
    }

    T deleteMaxByThreadId() {
        const int threadId = getThreadId();
        int queueIndex;
        int secondQueueIndex;
        do {
            int halfOfThreads = this->numOfThreads / 2;
            if (threadId < halfOfThreads) {
                queueIndex = getRandomQueueIndexForHalf();
                secondQueueIndex = getRandomQueueIndexForHalf();
            } else {
                queueIndex = getRandomQueueIndexForHalf() + this->numOfQueues / 2;
                secondQueueIndex = getRandomQueueIndexForHalf() + this->numOfQueues /
2;
            }

            queueIndex = getQueueIndexForDelete(queueIndex, secondQueueIndex);
        } while (queueIndex == -1 || !locks[queueIndex].try_lock());
        return getTopValue(queueIndex);
    }

    T deleteMaxByThreadOwn() {
        const int threadId = getThreadId();
        int queueIndex;
        int secondQueueIndex;
        bool firstIteration = true;
        do {
            if (firstIteration) {
                queueIndex = threadId * numOfQueuesPerThread;
                secondQueueIndex = threadId * numOfQueuesPerThread + 1;
                firstIteration = false;
            } else {
                queueIndex = getRandomQueueIndex();
                secondQueueIndex = getRandomQueueIndex();
            }

            queueIndex = getQueueIndexForDelete(queueIndex, secondQueueIndex);
        } while (queueIndex == -1 || !locks[queueIndex].try_lock());
        return getTopValue(queueIndex);
    }

    void balance() {
        int sumOfSizes = 0;
        int sizes[this->numOfQueues];
        int indexWithMax = 0, indexWithMin = 0;
        for (int i = 0; i < this->numOfQueues; ++i) {
            sizes[i] = internalQueues[i].size();
```

```
            sumOfSizes += sizes[i];
            if (sizes[indexWithMax] < sizes[i]) {
                indexWithMax = i;
            }
            if (i == 0 || sizes[indexWithMin] > sizes[i]) {
                indexWithMin = i;
            }
        }

        int averageSize = sumOfSizes / this->numOfQueues; //double is needless
        if (sizes[indexWithMax] > averageSize * 1.2) { // if max sized queue is 20%
bigger and more than average
            while (!locks[indexWithMax].try_lock());

            if (locks[indexWithMin].try_lock_for(std::chrono::milliseconds(1000))) {
                int sizeOfTransfer = static_cast<int>(sizes[indexWithMax] *
                                                      0.3); // 30% elements transfer
to smallest queue
                for (int i = 0; i < sizeOfTransfer; ++i) {
                    internalQueues[indexWithMin].push(internalQueues[indexWith-
Max].top());
                    internalQueues[indexWithMax].pop();
                }
                locks[indexWithMin].unlock();
            }

            locks[indexWithMax].unlock();
        }
    }

    void printSize() {
        for (int i = 0; i < this->numOfQueues; ++i) {
            std::cout << "Queue " << i << " has size " << internalQueues[i].size() <<
std::endl;
        }
    };

    long getSize() {
        long numOfElements = 0;
        for (int i = 0; i < this->numOfQueues; ++i) {
            numOfElements += internalQueues[i].size();
        }
        return numOfElements;
    }

};

#endif //MULTIQUEUES_H
```

ПРИЛОЖЕНИЕ Б. Исходный код оптимизированных алгоритмов вставки и удаления multiqueues на языке программирования kotlin

```
class MultiqueueImp<T : Comparable<T>>(
    private val numOfThreads: Int = 2,
    private val numOfQueuesPerThread: Int = 2,
    private val numOfQueues: Int = numOfThreads * numOfQueuesPerThread,
    private val internalQueues: List<PriorityQueue<T>> = MutableList<Priori-
tyQueue<T>>(numOfQueues) { PriorityQueue() },
    private val locks: Array<Mutex> = Array(numOfQueues) { Mutex() }
) : Multiqueue<T> {

    override fun printSize() {
        internalQueues.forEachIndexed { i, q ->
            println("Queue $i has size ${q.size}")
        }
    }

    override fun getSize(): Long {
        var numOfElements: Long = 0;
        for (i in internalQueues) {
            numOfElements += i.size
        }
        return numOfElements;
    }

    override suspend fun balance() {
        val sizes = IntArray(numOfQueues) { i -> internalQueues[i].size }

        val indexWithMax: Int = sizes.withIndex().maxBy { it.value }!!.index
        val indexWithMin: Int = sizes.withIndex().minBy { it.value }!!.index

        val sumOfSizes: Int = sizes.sum()
        val averageSize = sumOfSizes / numOfQueues
        val sizeCoeff = 1.2
        val transferCoeff = 0.3

        if (sizes[indexWithMax] > averageSize * sizeCoeff) { // if max sized queue is
20% bigger and more than average
            while (!locks[indexWithMax].tryLock());

            runBlocking(Dispatchers.Default) {
                withTimeout(TIMEOUT_IN_MILLIS) {
                    while (!locks[indexWithMin].tryLock());

                    val sizeOfTransfer =
                        (sizes[indexWithMax] * transferCoeff).toInt() // 30% elements
transfer to smallest queue
                    for (i in 0..sizeOfTransfer) {
                        internalQueues[indexWithMin].add(internalQueues[indexWith-
Max].peek())
                        internalQueues[indexWithMax].poll()
                    }
                    locks[indexWithMin].unlock()
                }
            }
```

```
            locks[indexWithMax].unlock()
        }
    }

    fun getRandomQueueIndexHalf(): Int {
        return Random().nextInt(numOfQueues / 2)
    }

    private fun getRandomQueueIndex(): Int {
        return Random().nextInt(numOfQueues)
    }

    private fun getQueueIndexForDelete(queueIndex: Int, secondQueueIndex: Int): Int? {
        val firstQueue = internalQueues[queueIndex]
        val secondQueue = internalQueues[secondQueueIndex]
        val foundQueueIndex: Int? = if (firstQueue.isEmpty() && secondQueue.isEmpty())
{
            return null
        } else if (firstQueue.isEmpty() && !secondQueue.isEmpty()) {
            secondQueueIndex
        } else {
            findAppropriateIndex(queueIndex, secondQueueIndex)
        }

        return if (foundQueueIndex != null && internalQueues[foundQueueIndex].is-
Empty()) {
            null
        } else foundQueueIndex
    }

    private fun findAppropriateIndex(
        firstQueueIndex: Int,
        secondQueueIndex: Int
    ): Int? {
        var foundQueueIndex: Int? = firstQueueIndex
        val value: T? = getValue(firstQueueIndex)
        val value2: T? = getValue(secondQueueIndex)

        if (value2 == null && value == null) {
            foundQueueIndex = null
        } else if (value != null && value2 == null) {
            foundQueueIndex = firstQueueIndex
        } else if (value == null && value2 != null) {
            foundQueueIndex = secondQueueIndex
        }

        if (value2 != null && value != null && value2 > value) {
            foundQueueIndex = secondQueueIndex
        }
        return foundQueueIndex
    }

    private fun getValue(foundQueueIndex: Int): T? {
        while (!locks[foundQueueIndex].tryLock());
        var value: T? = null
        val priorityQueue = internalQueues[foundQueueIndex]
        if (!priorityQueue.isEmpty())
            value = priorityQueue.peek()
        locks[foundQueueIndex].unlock()
```

```
    return value
}

override suspend fun insert(el: T) {
    var queueIndex: Int
    do {
        queueIndex = getRandomQueueIndex()
    } while (!locks[queueIndex].tryLock())
    internalQueues[queueIndex].add(el)
    locks[queueIndex].unlock()
}

override suspend fun insertByThreadId(el: T, threadId: Int) {
    var queueIndex: Int
    do {
        val halfOfThreads = numOfThreads / 2;

        if (threadId < halfOfThreads) {
            queueIndex = getRandomQueueIndexHalf()
        } else {
            queueIndex = getRandomQueueIndexHalf() + numOfQueues / 2;
        }
    } while (!locks[queueIndex].tryLock());
    internalQueues[queueIndex].add(el);
    locks[queueIndex].unlock();
}

override suspend fun deleteMax(): T? {
    var queueIndex: Int
    do {
        queueIndex = randomIndexSelection()
    } while (queueIndex == -1 || !locks[queueIndex].tryLock())
    return getTopValue(queueIndex)
}

private fun randomIndexSelection(): Int {
    var queueIndex = getRandomQueueIndex()
    var secondQueueIndex: Int
    do {
        secondQueueIndex = getRandomQueueIndex()
    } while (secondQueueIndex == queueIndex)
    queueIndex = getQueueIndexForDelete(queueIndex, secondQueueIndex) ?: -1
    return queueIndex
}

override suspend fun deleteMaxByThreadId(threadId: Int): T? {
    var queueIndex: Int
    do {
        queueIndex = getTopRandomHalfIndex(threadId)
    } while (queueIndex == -1 || !locks[queueIndex].tryLock())
    return getTopValue(queueIndex)
}

private fun getTopRandomHalfIndex(threadId: Int): Int {
    var queueIndex: Int
    val secondQueueIndex: Int
    val halfOfThreads = numOfThreads / 2
    if (threadId < halfOfThreads) {
        queueIndex = getRandomQueueIndexHalf()
```

```
            secondQueueIndex = getRandomQueueIndexHalf()
        } else {
            queueIndex = getRandomQueueIndexHalf() + numOfQueues / 2
            secondQueueIndex = getRandomQueueIndexHalf() + numOfQueues / 2
        }

        queueIndex = getQueueIndexForDelete(queueIndex, secondQueueIndex) ?: -1
        return queueIndex
    }

    override suspend fun deleteMaxByThreadOwn(threadId: Int): T? {
        var queueIndex: Int = threadId * numOfQueuesPerThread
        val secondQueueIndex = threadId * numOfQueuesPerThread + 1
        queueIndex = getQueueIndexForDelete(queueIndex, secondQueueIndex) ?: -1
        if (queueIndex != -1 && locks[queueIndex].tryLock()) {
            return getTopValue(queueIndex)
        }

        do {
            queueIndex = randomIndexSelection()
        } while (queueIndex == -1 || !locks[queueIndex].tryLock())
        return getTopValue(queueIndex)
    }

    private fun getTopValue(queueIndex: Int): T? {
        var topValue: T? = null
        if (!internalQueues[queueIndex].isEmpty()) {
            topValue = internalQueues[queueIndex].peek()
            internalQueues[queueIndex].poll()
        }
        locks[queueIndex].unlock()
        return topValue
    }

    companion object {
        private const val TIMEOUT_IN_MILLIS: Long = 10000
    }
}
```

ПРИЛОЖЕНИЕ В. Исходный код потокобезопасной ослабленной очереди с приоритетом CPQ на основе циклического списка на языке программирования C++

```
template<typename T>
class CircularPriorityQueue {
    Node<T> *head = nullptr;

    Node<T> *getPrevPriorNode() {
        Node<T> *node = head->next;

        if (node->isHead) {
            return node;
        }

        Node<T> *prevNode = head->next;
        Node<T> *prevPriorNode = prevNode;

        T priorValue = prevNode->top();
        do {
            T value = node->top();

            if (priorValue == CPQ_NULL || (value != CPQ_NULL && priorValue < value)) {
                priorValue = value;
                prevPriorNode = prevNode;
            }
            node = node->next;
            prevNode = node;
        } while (!node->isHead);

        return prevPriorNode;
    }

public:

    CircularPriorityQueue() {
        this->head = new Node<T>(true);
    }

    void push(T el) {
        Node<T> *node = head;

        do {
            if (node->usedMutex.try_lock()) {
                node->pushAndUnlock(el);
                return;
            }
            node = node->next;
        } while (!node->isHead);

        node = node->createNewNext();
        node->pushAndUnlock(el);
    }

    void pop() {
        Node<T> *prev = getPrevPriorNode();
        Node<T> *nodeToPop = prev->next;
```

```
        nodeToPop->usedMutex.lock();
        bool needToDelete = nodeToPop->pop();
        if (needToDelete && prev->next != nodeToPop->next) {
            prev->next = nodeToPop->next;
        }
        nodeToPop->usedMutex.unlock();
    }

    T top() {
        Node<T> *prev = getPrevPriorNode();
        return prev->next->top();
    }

    bool isEmpty() {
        Node<T> *node = head;
        do {
            if (!node->isEmpty()) {
                return false;
            }
            node = node->next;
        } while (node != head);
        return true;
    }

    int size() {
        int size = 0;
        Node<T> *node = head;
        do {
            size += node->size();
            node = node->next;
        } while (node != head);
        return size;
    }

    int nodes() {
        int size = 0;
        Node<T> *node = head;
        do {
            ++size;
            node = node->next;
        } while (node != head);
        return size;
    }
};
```

ПРИЛОЖЕНИЕ Г. Исходный код потокобезопасной ослабленной очереди с приоритетом на основе циклического списка CPQ на языке программирования kotlin

```
class CircularPriorityQueueImp<S : BlockingQueue<T>, T : Comparable<T>>(
    priority: Priority = Priority.MAX
) : RelaxedCircularDS<S, T> {

    private val priorityComparator: Comparator<T>? = if (priority == Priority.MAX)
Collections.reverseOrder() else null

    enum class Priority {
        MAX, MIN
    }

    private val head: Node<S, T> = Node(
        structure = PriorityBlockingQueue(1024, priorityComparator) as S,
        isHead = AtomicBoolean(true),
        priorityComparator = priorityComparator
    )

    override fun offer(el: T): Boolean {
        var node = head
        if (!node.isUsed.compareAndSet(false, true)) {
            if (!node.next.isHead.get()) {
                do {
                    node = node.next
                } while (!node.isHead.get() && !node.isUsed.compareAndSet(false,
true))
            }
            if (node.isHead.get()) {
                val next = node.createNewNext()
                head.next = next
                node = next
            }
        }
        val inserted = node.insert(el)
        node.isUsed.set(false)
        return inserted
    }

    override fun poll(): T? {
        val prev = getPrevPriorNode()
        val nodeToPop = prev.next

        val value: T? = nodeToPop.pop()
        if (nodeToPop.readyToDelete()) {
            prev.next = nodeToPop.next
        }

        return value
    }

    override fun peek(): T? {
        val prev = getPrevPriorNode()
        return prev.next.peek()
    }
```

```
    private fun getPrevPriorNode(): Node<S, T> {
        var node = head.next
        var prevNode: Node<S, T> = head

        if (node.isHead.get()) {
            return node
        }

        var priorValue: T? = prevNode.peek()
        var prevPriorNode: Node<S, T> = prevNode

        do {
            val value = node.peek()
            if (priorValue == null || (value != null && priorValue < value)) {
                priorValue = value
                prevPriorNode = prevNode
            }
            node = node.next
            prevNode = node
        } while (!node.isHead.get())

        return prevPriorNode
    }


    override fun isEmpty(): Boolean {
        var node = head

        do {
            if (!node.isEmpty()) {
                return false
            }
            node = node.next
        } while (node != head)

        return true
    }

    override fun printInfo() {
        var node = head
        var size = 0
        var structures = 0
        do {
            structures++
            size += node.size()
            node = node.next
        } while (!node.isHead.get())
        println("Size: $size, Structures $structures")
    }

    override fun clear() {
        head.next = head
        head.clear()
    }
}
```

ПРИЛОЖЕНИЕ Д. Пример логгирования результатов экспериментального исследования

```
Thread;OriginalInsert;OriginalDelete;OriginalRandom;OptimizedHalfInsert;Opti-
mizedHalfDelete;OptimizedHalfRandom;OptimizedExactInsert;OptimizedExactDe-
lete;OptimizedExactRandom
1;2802736;317527;786713;2802711;339186;801196;2854715;356318;828102
2;3382191;431493;845877;4897199;701352;1617702;4959054;695940;1652734
3;5270871;610074;1350722;6070879;826520;1757457;5857297;1000667;2035380
4;5450717;668361;1480115;6931930;905992;1982942;6688509;1274071;2305392
5;5000631;736078;1725129;6069489;892451;2148735;5814895;1514067;2456087
6;5125059;822017;2026368;5779335;996941;2343799;5726121;1769810;2833472
7;5585799;917963;2250845;6436246;1074192;2584578;6258257;2013452;3100007
8;5509090;878433;2183937;6149199;994616;2715788;6496100;2245743;3586201
9;5301426;619061;1261339;6135702;827553;2190922;6211027;2340049;3104312
10;4984840;620207;1094961;5963502;752700;1873740;6613820;2349823;3331097
11;4914344;435251;951804;6418248;673349;1741925;6288749;2551405;2984611
12;5457578;448154;979728;6045616;682706;2053849;6499462;2666085;2899374
13;5488925;310789;711842;6269297;721220;1491238;5992092;2901653;2933483
14;5586559;330576;842741;5557420;539317;1439043;6331646;2935031;2917214
15;4841015;279965;704098;6407644;630320;1679578;5780912;3134490;3134877
16;4932455;295059;654981;5176200;591685;1461744;6346248;3138865;2936832
```