УДК 004.021 **DOI: 10.14529/cmse260205**

СИНХРОНИЗАЦИЯ В КОНКУРЕНТНЫХ СТРУКТУРАХ ДАННЫХ НА ОСНОВЕ АНАЛИЗА ЗАВИСИМОСТЕЙ МЕЖДУ ОПЕРАЦИЯМИ*

© 2026 Д.С. Коротченко, В.Е. Аксенов

Университет ИТМО

(197101 Санкт-Петербург, Кронверкский пр., д. 49, лит. А)

E-mail: korotchenkods@gmail.com, aksenov.vitaly@gmail.com

Поступила в редакцию: 28.07.2026

В работе рассматривается оптимизация синхронизаций операций в потокобезопасных структурах данных. В различных структурах данных операции могут по-разному конфликтовать друг с другом с точки зрения одновременного исполнения. Например, стандартная блокировка чтения—записи учитывает, что одновременное чтение из разных потоков часто допустимо, в то время как одновременная запись из разных потоков или параллельная запись и чтение требуют синхронизации. Такая зависимость может быть и более сложной: например, несколько одновременных операций увеличения всех элементов в контейнере на заданную величину допустимы с синхронизацией на уровне доступа к элементу, в то время как параллельное присвоение всем элементам контейнера одного числа из одного потока и другого числа из другого потока требует более сложной синхронизации. В представленной работе рассматривается новый подход к обеспечению синхронизации при выполнении операций с потокобезопасной структурой данных, основанный на анализе графа зависимостей между операциями, с целью обеспечения более эффективной работы такой структуры. Также приводятся результаты экспериментов по сравнению различных подходов, которые показывают эффективность предлагаемого решения.

Ключевые слова: потокобезопасность, конкурентные структуры данных, граф зависимостей, семантическая блокировка, линеаризуемость.

ОБРАЗЕЦ ЦИТИРОВАНИЯ

Коротченко Д.С., Аксенов В.Е. Синхронизация в конкурентных структурах данных на основе анализа зависимостей между операциями // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2026. Т. 15, № 2. С. 82–103. DOI: 10.14529/cmse260205.

Введение

При разработке многопоточных приложений часто возникает потребность в использовании потокобезопасных структур данных. Такие структуры данных инкапсулируют логику синхронизации потоков внутри себя, предоставляя пользователю удобный интерфейс, позволяющий не задумываться о корректности. Нередко потокобезопасные структуры не просто гарантируют корректное поведение при работе из нескольких потоков, но также пытаются сделать структуру *конкурентной*, то есть обрабатывать вызовы операций над этой структурой данных, совершенные разными потоками, параллельно, когда это возможно. Поддерживающая конкурентность структура данных позволяет улучшить метрику *пропускной способности* — число запросов к данной структуре, обрабатываемых в секунду на заданном профиле нагрузки (числе потоков, с ней работающих, и распределении типов операций между этими потоками).

Стандартные библиотеки большинства языков программирования содержат различные потокобезопасные конкурентные реализации наиболее часто используемых структур, на-

*Статья рекомендована к публикации Программным комитетом Всероссийской научной конференции с международным участием «Параллельные вычислительные технологии (ПаВТ) 2026».

пример словарей, списков, множеств и др. Однако нередко возникает необходимость реализации дополнительных операций на таких структурах (например, преобразование всех элементов структуры или суммирование ее элементов), которые также требуют синхронизации с остальными операциями для сохранения корректности. Также у пользователей может возникнуть потребность в требуемых для работы приложения собственных реализациях структур данных, отличных от представленных в языке по умолчанию.

Конкурентные структуры данных можно разделить на два типа по способу синхронизации — так называемые *lock-free* структуры, использующие специальные алгоритмы для поддержания целостности и корректности без использования блокировок, и так называемые *lock-based* или основанные на блокировках структуры. В этой работе мы сосредоточились на структурах, основанных на блокировках. Несмотря на появление в стандартных и сторонних библиотеках реализаций, основанных на идее *lock-free*, структуры, работающие на блокировках, по-прежнему востребованы и популярны. При этом расширить существующую *lock-free* структуру дополнительной операцией с сохранением свойства *lock-free* чаще всего крайне сложно: для этого требуется большое количество сложного кода, в том числе модификация уже существующих операций. Расширить *lock-based* структуру гораздо проще. Отметим, что при желании *lock-free* структуру можно расширить другими операциями с использованием нашего подхода, потеряв при этом свойство *lock-free*.

Ключевое и наиболее часто встречающееся требование при работе с потокобезопасными структурами данных, подразумеваемое под корректностью — *линеаризуемость*. Интуитивно линеаризуемая структура данных — это структура, на которой любая последовательность вызовов операций из нескольких потоков может быть промоделирована на одном потоке [1]. В разделе 2 мы детальнее опишем, что именно подразумевается под линеаризуемостью и конкурентностью структур данных.

Простейший способ сохранить линеаризуемость при добавлении новых операций — дополнительно синхронизировать все операции с помощью отдельной блокировки. Использование дополнительной синхронизации позволит гарантировать, что никакие две операции не выполняются одновременно в процессе работы со структурой, из чего автоматически следует линеаризуемость.

Однако такой подход является неэффективным — структура фактически перестает быть конкурентной: никакие операции не выполняются параллельно. Обзор различных подходов, позволяющих разрешить исполнение некоторых операций параллельно с целью увеличения производительности, приведен в разделе 1.

При решении проблемы синхронизации операций зачастую возможно более эффективное поведение, чем взятие блокировки на каждую операцию. Так, если в структуре есть операции чтения, которые не модифицируют ее внутреннее состояние в процессе выполнения, а лишь используют информацию о данных, хранящихся в структуре, то выполнение такой операции параллельно с другой операцией чтения не повлияет на результаты их исполнения. В таком случае можно было бы использовать блокировки чтения—записи, то есть такие блокировки, которые допускают параллельное исполнение нескольких операций чтения, но не допускают параллельного исполнения каких-либо операций параллельно с операцией записи. Также блокировку чтения—записи можно применить в ситуации расширения конкурентной структуры данных дополнительными операциями. В этом случае изначально представленные в структуре данных операции уже гарантируют линеаризуемость, используя, возможно, дополнительные оптимизации, чтобы даже те операции, которые мо-

дифицируют структуру данных (операции записи), могли бы исполняться параллельно в отдельных случаях. Такие оптимизации обычно зависят от специфики структуры данных и могут быть различными в различных реализациях. В таком случае с помощью блокировки чтения—записи можно расширить структуру данных новыми операциями, не ухудшая производительность исходных операций (с учетом заложенной в структуру конкурентности) путем взятия для выполнения таких операций блокировки чтения, но при этом не допускать ни параллелизации таких операций с новыми, добавляемыми в структуру данных, операциями, ни параллелизации новых операций между собой за счет взятия для всех новых операций блокировки записи.

Легко заметить, что и такое решение не всегда оптимально, например, если структура расширяется операциями чтения, то их можно было бы исполнить параллельно между собой (или с исходно имеющимися операциями чтения), но для этого необходимо использовать более сложную блокировку. Далее в работе мы приведем и другой пример структуры, в которой зависимость между операциями устроена более сложным образом.

В этой работе мы представляем новый подход, предлагающий использование нового алгоритма блокировки, которая реализует соответствующую логику на основе информации о зависимостях между различными операциями. Далее мы будем называть такую блокировку *семантической*.

Предлагаемый подход актуален в различных практических задачах: системах управления базами данных, выполняющими диапазонные запросы параллельно с точечными обновлениями; кэшах в оперативной памяти с операциями массового обновления или чтения; системах аналитики реального времени, где операции агрегации могут выполняться параллельно операциям обновления. В серверных приложениях типичное число потоков составляет от 4 до 32, что соответствует диапазону, исследованному в экспериментах в рамках этой работы.

Основными вкладами работы являются:

- Формализация зависимостей между операциями в структуре данных в виде графа зависимостей, обобщающая классическую модель блокировок чтения—записи на произвольные зависимости.
- Новый алгоритм семантической блокировки с двумя подходами к реализации логики проверки конфликтов: на основе дополнительных блокировок и на основе атомарных счетчиков.
- Экспериментальная оценка предложенного подхода с использованием бенчмарка Synchrobench, демонстрирующая преимущество семантической блокировки в различных сценариях, особенно с доминированием параллельно-совместимых операций.

Статья организована следующим образом. В разделе 1 приводится обзор связанных работ. В разделе 2 формализуются требования к структурам данных, а также целевая метрика. В разделе 3 детально описаны различные подходы к синхронизации операций. В разделе 4 подробно описаны результаты сравнения производительности тестовой структуры данных с различными способами синхронизации операций, показывающие эффективность предлагаемого решения. В заключении приводится краткая сводка результатов, полученных в работе, и указаны направления дальнейших исследований.

1. Обзор литературы

Наиболее близкой к рассматриваемой постановке классической абстракцией является задача группового взаимного исключения (Group Mutual Exclusion, GME) [2]. В отличие от простого взаимоисключения, которое подразумевает эксклюзивный доступ к ресурсу, в алгоритмах группового взаимного исключения разрешается доступ к ресурсу нескольких параллельных процессов, относящихся к одной группе, но запрещен одновременный доступ процессов из разных групп. Произвольный граф зависимостей операций в структуре данных может выглядеть сложнее — операция A может не конфликтовать с B, но при этом конфликтовать сама с собой, что уже не позволит отнести эти две операции к одному классу. Эффективные алгоритмы решения задачи группового взаимного исключения широко исследуются в различных вариациях [3, 4], однако чаще всего направлены на решение проблемы в распределенных системах, что также отличает их от рассматриваемой в работе задачи.

Большое число работ посвящено исследованию различных подходов для ускорения выполнения долгих операций. Например, исполнение нескольких операций одним батчем под одной блокировкой [5]; подходы на основе идеи коллаборативности — переиспользование заблокированных потоков для быстрейшего выполнения операции, ставшей причиной блокировки [6]; добавление частичной персистентности, то есть хранения нескольких версий структуры для ускорения выполнения или параллелизации отдельных структур данных [7].

В то же время проблема синхронизации операций в структурах данных перекликается с проблемой выполнения транзакций и откатов в базах данных (и транзакционной памяти в общем случае). Для корректной поддержки транзакционности в базах данных широко исследуются различные типы блокировок, например блокировки отрезков [8], предикатные блокировки, позволяющие избавиться от части фантомных конфликтов [9], анализируются операции для выявления более слабых свойств по сравнению с коммутативностью и их применения для корректной обработки ситуаций отката транзакций [10]. Переход от режимов доступа к семантике абстрактного типа данных был формализован в управлении параллелизмом на основе коммутативности: две операции могут выполняться параллельно, если перестановка их порядка не меняет допустимые состояния и наблюдаемые результаты [11].

Обнаружение конфликтов по адресам в транзакционной памяти [12] приводит к ложным конфликтам. Transactional Boosting позволяет обнаружить конфликты на уровне методов: некоммутирующие вызовы захватывают конфликтующие блокировки, а откат выполняется с помощью журнала и обратных операций [13]. Однако эффективное использование транзакций и транзакционной памяти требует возможности выполнения дешевых обратных операций, что затруднено для операций, затрагивающих большую часть элементов структуры данных. Кроме того, расширение структур, реализованных для обычной памяти, потребует в этом случае их полного изменения для работы с транзакционной памятью.

2. Синхронизация операций в потокобезопасных структурах данных

Потокобезопасными структурами данных называются такие структуры данных, которые гарантируют корректное поведение при вызове их операций из нескольких потоков.

Под корректным поведением в этой работе мы будем подразумевать *линеаризуемость* результатов выполнения операций над структурой данных [1].

Определение 1. Результат выполнения вызванных из нескольких потоков операций над структурой данных, на которые получен ответ, *линеаризуем*, если можно последовательно выполнить тот же набор операций в одном потоке так, что результаты всех операций совпадут с исходными. При этом операции, которые изначально исполнялись на одном потоке, должны быть исполнены в том же взаимном порядке.

Линеаризуемость является ключевым и наиболее часто требуемым на практике свойством. Линеаризуемость может быть проверена различными инструментами: так, для языков Java и Kotlin общепринятым инструментом проверки свойства *линеаризуемости* является Lincheck [14]. С помощью Lincheck были проверены все реализации на языке Java описанных далее подходов, а также структуры данных, использованные в экспериментах в рамках этой работы.

Помимо *линеаризуемости* введем также понятие *справедливости* (fair) структуры. Справедливость определяет, как структура обрабатывает ситуацию, в которой поток, ожидающий выполнения операции из-за конфликта с уже исполняемыми операциями, может быть вытеснен новыми операциями, конфликтующими с его операцией. Структура называется справедливой, если она гарантирует, что ожидающий поток получит возможность выполнить свою операцию раньше, чем любая операция, запрошенная позже и конфликтующая с ней.

Определение 2. Поток A выполняет операцию a над структурой данных. При этом операция a не может быть исполнена из-за того, что другая операция, исполняемая в данный момент, не позволяет потоку A получить необходимую для исполнения операции a блокировку. После неудачной попытки получить блокировку потоком A другой поток B пытается выполнить операцию b, причем операция a не может быть выполнена параллельно с операцией b. Тогда структура называется *справедливой*, если она гарантирует, что операция a на потоке A будет исполнена раньше, чем операция b на потоке B, и *несправедливой*, если такой гарантии нет.

Приведем пример: рассмотрим структуру данных, которая хранит в себе одно число, позволяя параллельно выполнять чтение из нескольких потоков, но при этом гарантируя, что запись не выполняется параллельно с чтением или другой записью. Запустим один поток, который будет непрерывно выполнять запись, и на порядок большее число потоков, непрерывно выполняющих чтение. Если такой эксперимент провести на *справедливой* структуре данных, то соотношение выполненных операций чтения к выполненным операциям записи будет примерно соответствовать соотношению числа потоков, в которых выполняется операция записи, к числу потоков, в которых выполняется операция чтения. Если же используемая структура данных будет *несправедливой*, запись может быть выполнена очень малое (близкое к нулю) число раз. Это происходит из-за того, что после захвата блокировки на чтение одним из потоков другие потоки также начнут его выполнять параллельно. Это приведет к тому, что в любой момент времени почти наверняка будет хотя бы один поток, который выполняет чтение, а значит, хотя бы один поток будет владеть блокировкой на чтение, и, следовательно, операция записи не сможет быть исполнена.

На практике блокировки из библиотек языка зачастую поддерживают оба режима. Например, стандартная реализация блокировки чтения—записи (*ReentrantReadWriteLock*) в Java по умолчанию работает в *несправедливом* режиме, а *справедливый* режим включается явно. Предлагаемый в этой работе подход также поддерживает оба режима.

Помимо потокобезопасности от структур чаще всего ожидается *конкурентность*, то есть параллелизация исполнения операций, запрошенных разными потоками, там, где это возможно [5]. В разделе 3.1 мы опишем простейший подход обеспечения потокобезопасности, использующий дополнительную блокировку, запрашиваемую при исполнении каждой операцией. Такой подход лишает структуру данных конкурентности, так как все операции фактически исполняются последовательно. Целью этой работы является, наоборот, как можно большее увеличение конкурентности при использовании предлагаемого решения.

Ключевая метрика, которую мы будем рассматривать как целевую в работе, — *пропускная способность* структуры при заданных параметрах. Интуитивно она означает следующее: предположим, что структура данных рассматривается как черная коробка, с которой работают некоторые потоки, выполняя некоторый набор операций непрерывно. Пропускная способность отражает число операций, которые будут выполнены структурой в такой ситуации за некоторый промежуток времени. Более подробно метрика будет описана в начале раздела 4.

3. Подходы к обеспечению синхронизации операций в потокобезопасных структурах данных

Безусловно, в большинстве языков программирования уже существуют стандартные библиотеки, предлагающие пользователям большое число различных потокобезопасных структур данных. Однако потребность в механизме, помогающем с синхронизацией операций в структурах данных, все равно остается актуальной. Во-первых, такая потребность возникает при разработке собственных специфичных для конкретных задач структур данных. Во-вторых, потребность также возникает при расширении существующей структуры дополнительными операциями. Например, в произвольной структуре данных может возникнуть потребность в выполнении свертки (например, подсчет суммы всех значений) над элементами структуры, а в упорядоченных структурах данных может возникнуть потребность выполнять добавление или установление нового значения на отрезке. В-третьих, оказывается, что в реализациях из стандартных библиотек могут существовать ошибки, приводящие к проблемам с линеаризуемостью, что делает использование таких структур невозможным, если линеаризуемость является критичной для решаемой задачи.

3.1. Использование простой блокировки для синхронизации операций

Простейший способ гарантировать линеаризуемость — использовать одну дополнительную блокировку. В этом случае все операции для исполнения требуют взятия блокировки, а после выполнения ее отпускают. Псевдокод такого подхода приведен на рис. 1.

Многие языки предоставляют стандартный механизм превращения структуры данных в потокобезопасную с помощью такого подхода. Например, в языке программирования Java существует метод *Collections.synchronizedCollection(someStructure)*, который оборачивает все операции в структуре данных в дополнительную блокировку, превращая ее в потокобезопасную.

```
operation():
    lock.lock()
    original_operation()
    lock.unlock()
```

Рис. 1. Синхронизация с помощью дополнительной блокировки

3.2. Использование блокировки чтения—записи для синхронизации операций

Понятно, что такой подход не является оптимальным. Так, рассмотрим операции, которые не модифицируют внутренние данные структуры, а лишь читают их. Такие операции могут быть выполнены параллельно, поскольку выполнение одной операции не может повлиять на данные, а следовательно, и на результат выполнения параллельной операции.

Определение 3. Операции, которые не изменяют внутреннее содержимое структуры данных, будем называть *операциями чтения*.

Определение 4. Операции, которые могут изменить внутреннее содержимое структуры данных, будем называть *операциями записи*.

Операции чаще всего легко и достаточно удобно разделить на операции чтения и записи (при сомнениях можно просто отнести операцию к операциям записи); при этом реализация подхода, при котором операции чтения могут выполняться параллельно, может привести к значительному увеличению производительности, как мы увидим в разделе 4. Это приводит к появлению специального вида блокировок, которые реализованы в большинстве языков программирования в стандартных библиотеках — блокировок чтения—записи [15].

Определение 5. *Блокировка чтения—записи* — специальный вид блокировки, который предоставляет методы для взятия блокировок раздельно на чтение и запись. При этом блокировкой на чтение могут одновременно владеть несколько потоков при условии, что никакой поток не владеет блокировкой на запись, в то время как блокировка на запись может быть взята только одним потоком при условии, что никакие другие потоки не владеют ни блокировкой на чтение, ни блокировкой на запись.

Пример применения блокировки чтения—записи в методах структуры, осуществляющих доступ к одной переменной, приведен на рис. 2.

Блокировка чтения—записи также может быть использована для более эффективной синхронизации при расширении потокобезопасной структуры данных новыми операциями. Поскольку операции, представленные изначально в структуре данных, уже обеспечены потокобезопасностью, их не нужно дополнительно синхронизировать между собой дополнительной блокировкой. Если же операция расширяет структуру данных, то она потенциально может конфликтовать с любой из имеющихся или новых операций. Логика таких зависимостей совпадает с логикой зависимостей между операциями чтения и записи. Это позволяет использовать для дополнительной синхронизации блокировку чтения—записи. Так, все изначально представленные операции будут требовать взятия блокировки чтения перед своим исполнением, что позволит допускать исполнение таких операций из нескольких потоков без дополнительной синхронизации. А все новые операции будут требовать

```
get():
    readWriteLock.lockRead()
    result = value
    readWriteLock.unlockRead()
    return result

set(newValue):
    readWriteLock.lockWrite()
    prev = value
    value = newValue
    readWriteLock.unlockWrite()
    return prev
```

Рис. 2. Использование блокировки чтения—записи

взятия блокировки записи, гарантируя тем самым, что такие операции не будут исполнены параллельно с другими.

Пример применения блокировки чтения—записи при расширении структуры новыми операциями представлен на рис. 3.

```
operation1():
    readWriteLock.lockRead()
    originalOperation1() <- вызов исходного метода
    readWriteLock.unlockRead()

operation2():
    readWriteLock.lockRead()
    originalOperation2() <- вызов исходного метода
    readWriteLock.unlockRead()

newOperation3():
    readWriteLock.lockWrite()
    .. <- логика операции 3, расширяющей исходную структуру
    readWriteLock.unlockWrite()

newOperation4():
    readWriteLock.lockWrite()
    .. <- логика операции 4, расширяющей исходную структуру
    readWriteLock.unlockWrite()
```

Рис. 3. Использование блокировки чтения—записи при расширении структуры

3.3. Анализ зависимостей между операциями для их синхронизации

Использование блокировки чтения—записи позволяет улучшить производительность структуры данных по сравнению с использованием обычной блокировки, однако легко привести пример, в котором зависимости между операциями устроены сложнее: например, некоторые операции записи могут быть выполнены параллельно друг другу.

Рассмотрим структуру данных, представляющую упорядоченный набор *атомарных* (то есть таких, с которыми можно работать из нескольких потоков без дополнительных синхронизаций) чисел, на котором определены три операции: *addOnRange*, которая прибавляет ко всем числам на отрезке некоторое переданное значение, *setOnRange*, которая устанавливает всем числам на отрезке некоторое переданное значение, и *getRangeSum*, которая возвращает сумму чисел на отрезке. Заметим, что выполнение двух операций *addOnRange* параллельно друг с другом не приводит к проблемам с линеаризуемостью, поскольку после завершения обеих операций к каждому элементу структуры будут прибавлены все нужные значения. В то же время для операции *setOnRange* это неверно. У элементов, которые принадлежат обоим отрезкам, могут оказаться в итоге различные значения из-за состояния гонки, что невозможно при последовательном выполнении операций. Очевидно, что операции *getRangeSum* также можно выполнять параллельно друг другу, ведь это просто операции чтения. Получается, что в данном случае более оптимально было бы использовать блокировку, которая допускала бы одновременное исполнение нескольких операций *addOnRange* или *getRangeSum*, но не допускала бы одновременного выполнения операций различного типа или нескольких операций *setOnRange*.

Для того чтобы более эффективно работать с такими структурами данных, в этой работе мы предлагаем собственную *семантическую* блокировку, поведение которой основывается на информации о конфликтах между операциями структуры данных.

Определение 6. Две операции называются *зависимыми* (конфликтующими), если их параллельное исполнение может привести к потери линеаризуемости.

При создании семантическая блокировка принимает на вход информацию о зависимостях между операциями, представляющую из себя матрицу, где на i-й строке j-го столбца стоит логическое значение $True$, если операции j и i зависимы (то есть конфликтуют), и $False$ иначе. В матрице также отражается возможность или невозможность параллельного выполнения операций одного типа с помощью соответствующих значений в i-й строке i-м столбце. Пример матрицы зависимостей для структуры, описанной выше, приведен в табл. 1.

Таблица 1. Пример матрицы зависимостей

	addOnRange	**setOnRange**	**getRangeSum**
addOnRange	False	True	True
setOnRange	True	True	True
getRangeSum	True	True	False

Дальнейшая работа с блокировкой происходит с помощью двух основных методов: *tryLock(operation)* и *unlock(operation)*.

При выполнении каждой операции поток запрашивает разрешение, вызывая метод *tryLock* с идентификатором соответствующей операции в качестве параметра. В случае получения разрешения поток выполняет операцию и в конце вызывает метод *unlock*, сообщая тем самым о завершении операции.

Разберем детали работы перечисленных выше методов. Для этого введем понятие графа зависимостей операций.

Определение 7. Определим *граф зависимостей* операций как граф, построенный с помощью описанной выше матрицы смежности следующим образом: вершины в графе отвечают за операции, а ребра (в том числе петли) проведены между вершинами, соответствующими зависимым операциям.

С точки зрения *графа зависимостей* при вызове *tryLock(a)* необходимо проверить, выполняется ли хотя бы одна операция в данный момент, вершина которой соединена ребром с вершиной операции a.

```
// сохраненные при инициализации номера компонент связанности для каждой операции
operationId2ComponentId = ..
// массив простых блокировок для каждой компоненты связанности
componentLocks = Lock[componentsCount]
// объект, в котором каждой операции по ее operationId сопоставлен список id операций,
//  с которыми она конфликтует
graph = ..
// массив счетчиков (обычных (неатомарных) чисел), отражающих число операций,
//  которые сейчас исполняются для каждого типа операций
inProgress = Int[operationsCount]

tryLock(operationId):
    // взятие блокировки, связанной с компонентой связанности
    componentId = operationId2ComponentId[operationId]
    componentLocks[componentId].lock()

    // проверка отсутствия конфликтов
    for conflictOperationId in graph[operationId]:
        if inProgress[conflictOperationId] > 0:
            return False

    // увеличение соответствующего счетчика
    inProgress[operationId]++
    componentLocks[componentId].unlock()
    return True

unlock(operationId):
    // взятие блокировки, связанной с компонентой связанности,
    //  необходимо из-за неатомарности счетчиков и возможности одновременного
    //  уменьшения счетчика из разных потоков
    componentId = operationId2ComponentId[operationId]
    componentLocks[componentId].lock()

    inProgress[operationId]--
    componentLocks[componentId].unlock()
```

Рис. 4. Проверка конфликтов с помощью отдельной блокировки

Для осуществления такой проверки будем поддерживать внутри семантической блокировки, помимо графа зависимостей, также счетчики для каждой операции, отражающие число операций заданного типа, выполняемых в данный момент. Заметим, что такое число может быть больше 1, если операция не конфликтует сама с собой.

Таким образом, при вызове операции *tryLock(a)* необходимо проверить счетчики всех операций, связанных с операцией a ребром. Если все такие счетчики равны 0, то операция a может быть исполнена.

Проблема, которая возникает с таким подходом, связана с тем, что метод *tryLock* может быть одновременно вызван из нескольких потоков, что может привести к тому, что две конфликтующие операции запустятся одновременно, так как на момент проверки счетчики были равны 0 для обеих из них.

Для того чтобы решить эту проблему, введем также внутренние блокировки для каждой компоненты связанности в графе зависимостей. Это позволит независимо проверять возможность исполнения операций, которые могут конфликтовать, а также безопасно менять счетчики без использования атомарных переменных.

Псевдокод реализации такой проверки приведен на рис. 4.

Другой возможный подход для решения этой проблемы — использование атомарных значений—счетчиков. При этом в начале проверки соответствующее значение увеличивается, а лишь затем проверяются счетчики конфликтующих операций. Псевдокод такой проверки приведен на рис. 5.

```
graph = ..
// массив счетчиков (атомарных чисел), отражающих число операций,
//  которые сейчас исполняются для каждого типа операций
inProgress = AtomicInteger[operationsCount]

tryLock(operationId):
    // увеличение счетчика
    if operationId in graph[operationId] and inProgress[operationId] > 0:
        return False
    inProgress[operationId]++

    // проверка отсутствия конфликтов
    for conflictOperationId in graph[operationId]:
        if conflictOperationId == operationId:
            if inProgress[operationId] > 1:
                inProgress[operationId]--
                return False
        elif inProgress[conflictOperationId] > 0:
            inProgress[operationId]--
            return False

    return True

unlock(operationId):
    inProgress[operationId]--
```

Рис. 5. Проверка конфликтов с помощью атомарных счетчиков

Сравнение эффективности таких подходов будет приведено в разделе 4.3.

3.4. Обеспечение справедливости структуры данных

Выше мы описали разделение структур на справедливые и несправедливые. Для обеспечения поддержки справедливости в структурах, использующих семантическую блокировку, перед первой попыткой взятия блокировки поток добавляет свой идентификатор в специальную очередь, связанную с компонентой связанности текущей операции, после чего метод *tryLock* перед началом исполнения всего остального кода проверяет соответствие первого элемента очереди идентификатору текущего потока. Если идентификаторы не совпали, то метод сразу возвращает отказ во взятии блокировки. Если же идентификаторы совпали, то поток выполняет все необходимые проверки и при отсутствии конфликтов удаляет первый элемент из очереди.

Способы гарантии справедливости структуры данных при использовании семантической блокировки могут быть возможным направлением дальнейших исследований.

Во всех экспериментах в разделе 4 используется справедливый режим работы блокировок.

4. Эксперименты

4.1. Методика проведения экспериментов и целевая метрика

Для проведения сравнений *семантическая* блокировка была реализована на языке Java. Для сравнения с другими подходами был выбран вариант реализации с дополнительными блокировками на компоненты связанности при проверке наличия конфликтов. Сравнение такой реализации с реализацией на основе атомарных счетчиков приведено в разделе 4.3.

Обычная блокировка и блокировки чтения—записи были взяты из стандартной библиотеки Java.

Для проведения экспериментов различные способы синхронизации были применены к тестовой структуре данных. Все реализации были успешно проверены на линеаризуемость с помощью инструмента Lincheck [16].

Исходные коды реализаций семантической блокировки и тестовой структуры данных свободно доступны в репозитории GitHub [17].

Тестовая структура данных представляет собой массив числовых значений со следующими операциями: *addOnRange(left, right, value)* — добавляет значение $value$ ко всем элементам на отрезке с индексами от $left$ (включительно) до $right$ (невключительно); *setOnRange(left, right, value)* — устанавливает значение $value$ всем элементам на отрезке с индексами от $left$ (включительно) до $right$ (невключительно); *getRangeSum(left, right)* — возвращает сумму значений элементов на отрезке с индексами от $left$ (включительно) до $right$ (невключительно).

Позднее мы усложним *граф зависимостей* структуры за счет дополнительных оптимизаций, рассмотренных в разделе 4.4.

В экспериментах использовалась модифицированная версия инструмента для тестирования конкурентных структур данных под нагрузкой — Synchrobench [18]. Модификация инструмента потребовалась из-за того, что исходно Synchrobench поддерживает лишь работу со словарями (интерфейс Map) и множествами (интерфейс Set).

Synchrobench поддерживает достаточно большой набор параметров; основные неизменные в различных запусках параметры перечислены в табл. 2.

Таблица 2. Постоянные параметры запуска Synchrobench

Параметр	Значение
длительность прогрева	10 секунд
длительность эксперимента	10 секунд
размер структуры данных	$2^{26} = 67108864$ (если не указано иное)
количество итераций	10

Помимо перечисленных параметров, при запуске указывались два переменных параметра — число потоков *threads*, используемых в эксперименте, а также дискретное вероятностное распределение P, используемое для выбора очередной операции.

После запуска Synchrobench создает *threads* потоков, каждый из которых в цикле пытается выполнить одну из возможных операций со структурой данных. При этом каждая операция выбирается заново случайным образом с вероятностями, соответствующими дискретному распределению P. Например, если структура поддерживает две операции *read* и *write*, то параметр P должен задавать вероятность выбора *read* и *write* соответственно, причем сумма этих вероятностей должна быть равна 1.

В качестве целевой метрики в экспериментах рассматривается пропускная способность структуры при заданных переменных параметрах (число потоков и распределение операций). Эта метрика отражает число операций, которые были завершены за некоторый фиксированный промежуток времени (обычно, за секунду).

Эксперименты проводились на машине с процессором Xeon Gold 5128 с 16 ядрами и поддержкой гипертрединга и с 64 Гб оперативной памяти. Результаты части экспериментов также были подтверждены на машине с процессором Apple M4 Pro с 14 ядрами (из которых 10 производительных) и с 48 Гб оперативной памяти, что позволяет сделать вывод об отсутствии значимого влияния используемой платформы на эффективность решения.

4.2. Сравнение различных подходов для синхронизации

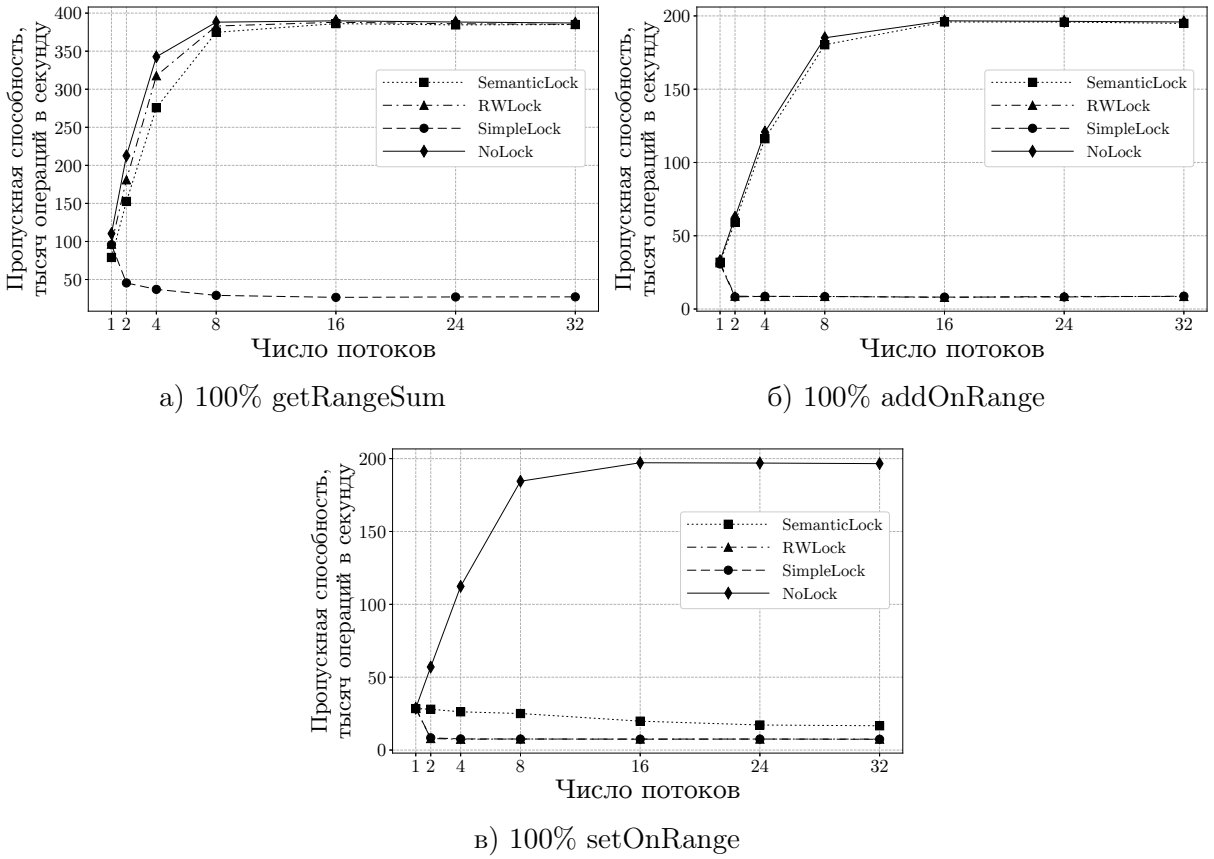
В этом эксперименте рассматриваются три подхода — использование дополнительной обычной блокировки (**SimpleLock** на рисунках ниже); использование дополнительной блокировки чтения—записи (**RWLock** ниже); использование семантической блокировки (в варианте неатомарных счетчиков и проверки конфликтов под дополнительной блокировкой, **SemanticLock** ниже).

Для сравнения также приводятся результаты для структуры, не использующей дополнительные блокировки, результаты работы над которой не гарантируют линеаризуемости (**NoLock** ниже).

Границы отрезков в рамках этого и следующих экспериментов выбирались следующим образом: сначала случайно выбирается левая граница $left$ равномерно по всей длине структуры, после этого выбирается длина отрезка $length$ случайно равномерно от 0 до $min(10^4, size - left)$, затем при помощи сложения левой границы и длины формируется параметр правой границы $right = left + length$.

На рис. 6 показаны зависимости пропускной способности от числа потоков при выполнении всеми потоками операций одного типа.

Видно, что для операции *getRangeSum* реализации и с блокировкой чтения—записи, и с семантической блокировкой показывают рост пропускной способности с ростом числа



а) 100% getRangeSum

б) 100% addOnRange

в) 100% setOnRange

Рис. 6. Результаты в случае выполнения операций одного типа

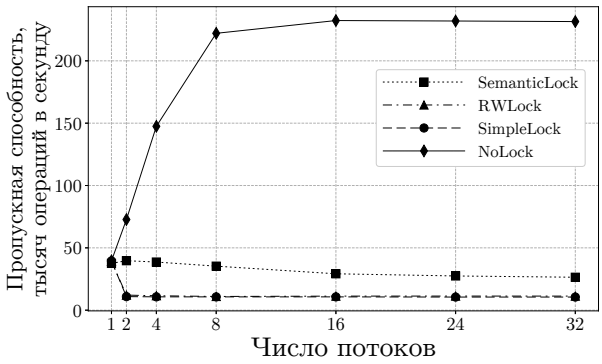
потоков, при этом находясь близко к значению пропускной способности нелинеаризуемой структуры без блокировок. Это свидетельствует, во-первых, о низких накладных расходах на обе блокировки, а во-вторых, об успешном параллельном исполнении операции чтения в этих случаях. Решение, основанное на простой блокировке, не только не показывает роста, но и, наоборот, с переходом к нескольким потокам пропускная способность падает. По остальным графикам видно, что в процентном соотношении падение для операции записи более значительно, чем для операции чтения. Это падение связано с работой кэшей и происходящими синхронизациями потоков.

Результаты для операции *addOnRange* также показывают преимущества семантической блокировки. Там, где блокировка чтения—записи уже не может поддержать корректное с логической точки зрения одновременное исполнение операций, семантическая блокировка позволяет это сделать, тем самым повышая пропускную способность с ростом числа потоков, по-прежнему позволяя поддерживать пропускную способность на уровне реализации без блокировок.

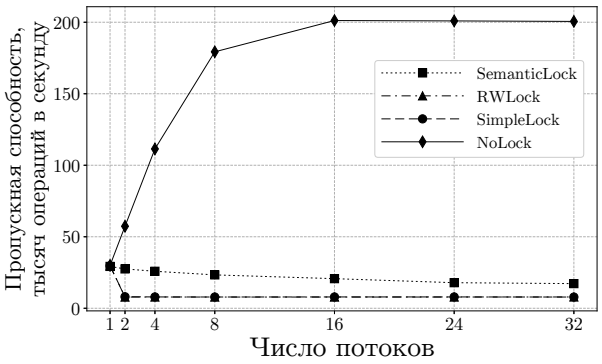
Для операции *setOnRange* пропускная способность всех реализаций падает при переходе к работе с несколькими потоками, аналогично реализации с простой блокировкой в случае чтения. При этом отметим, что использование семантической блокировки в этом случае позволило минимизировать падение производительности.

Рассмотрим теперь смешанные ситуации, в которых распределение операций устроено сложнее.

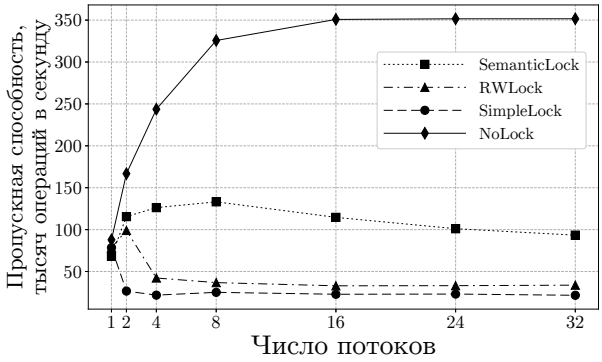
Возьмем несколько вариантов таких распределений: равномерное между всеми типами, доминирование операции чтения (90% *getRangeSum*, по 5% операций записи каждого типа), доминирование операции *addOnRange*, доминирование операции *setOnRange*.



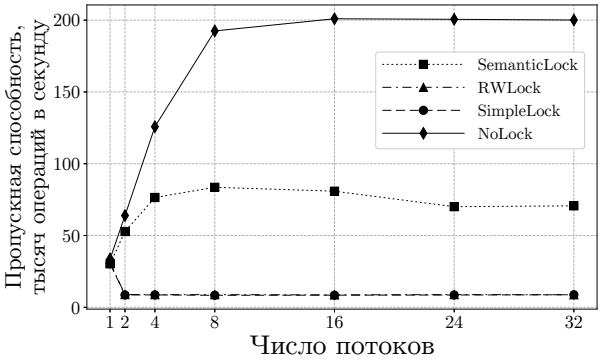
а) равномерное распределение операций



б) 90% setOnRange, 5% addOnRange, 5% getRangeSum



в) 5% setOnRange, 5% addOnRange, 90% getRangeSum



г) 5% setOnRange, 90% addOnRange, 5% getRangeSum

Рис. 7. Результаты в случае выполнения операций различного типа

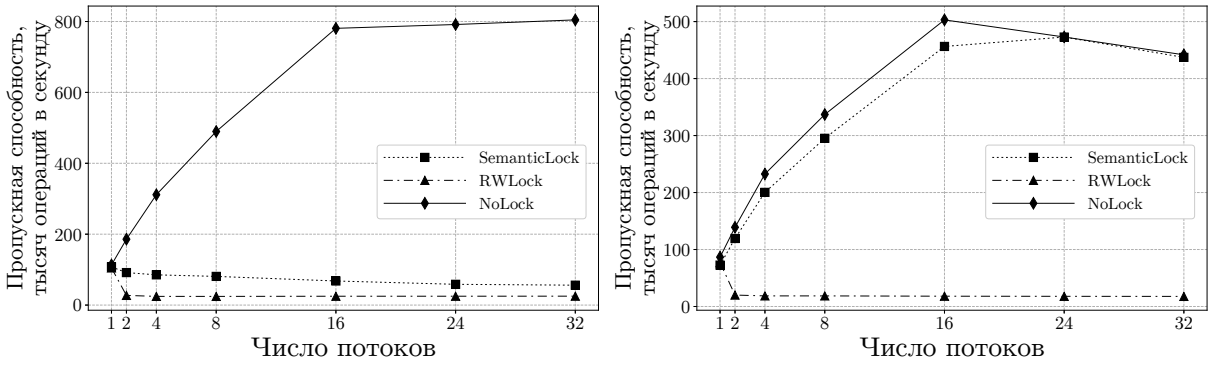
Результаты экспериментов, приведенные на рис. 7, также показывают эффективность предложенного решения.

В случае доминирования операций, которые могут быть выполнены параллельно, семантическая блокировка позволяет увеличить пропускную способность при переходе к нескольким потокам, в то время как даже для операции чтения блокировка чтения—записи показывает лишь небольшой рост при переходе к двум потокам, после чего пропускная способность падает, хоть и остается на уровне, более высоком, чем для обычной блокировки.

В случае же равномерного распределения операций использование семантической блокировки позволило лишь незначительно улучшить пропускную способность при работе из двух потоков; с дальнейшим увеличением числа потоков семантическая блокировка позволила сохранить пропускную способность примерно на том же уровне без значительного падения.

Выход на плато для реализации без блокировок связан с тем, что размер массива не позволяет полностью поместить его в кэш. Эту гипотезу подтверждают результаты подобных экспериментов с меньшим размером массива — 2^{13} элементов, которые приведены на рис. 8. Видно, что рост прекращается лишь после превышения числом потоков числа физических ядер. При этом поддержка гипертрединга в процессоре не помогает увеличить пропуск-

ную способность после превышения числом потоков числа ядер, так как в первую очередь направлена на решение проблемы блокирующих операций, которая в таких примерах не возникает.

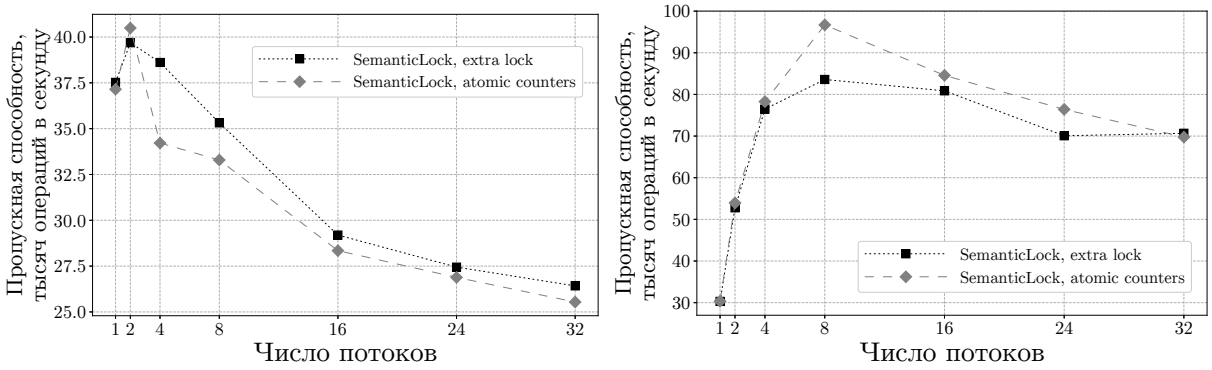


а) равномерное распределение операций б) 100% addOnRange

Рис. 8. Результаты с меньшим числом элементов (2^{13}) в массиве

4.3. Результаты сравнения различных способов проверки конфликта

В этом разделе приводятся результаты сравнения двух способов проверки наличия конфликта, изложенных в разделе 3.3. Использованный в остальных экспериментах подход, основанный на взятии дополнительной блокировки, связанной с компонентой связанности операции, для проверки конфликтов, сравнивается с подходом, основанным на использовании атомарных счетчиков.



а) равномерное распределение операций б) 5% setOnRange, 90% addOnRange, 5% getRangeSum

Рис. 9. Результаты сравнения различных реализаций проверки конфликта

Из результатов, приведенных на рис. 9, сложно сделать выбор в пользу того или иного решения.

Более детальное сравнение подходов может быть направлением дальнейших исследований и улучшений семантической блокировки.

4.4. Разделение операций на длинные и короткие

При работе со структурой значительного размера захват блокировок на всю структуру может быть неоптимальным, если в запросе используются небольшие отрезки. Действи-

тельно, в этом случае было бы эффективнее разделить весь массив на небольшие отрезки и захватывать блокировку лишь на те отрезки, которые пересекаются с отрезком запроса.

Однако накладные расходы на захват большого количества блокировок высоки, поэтому для длинных отрезков такой подход в свою очередь был бы неоптимальным.

Возможная оптимизация с учетом этой идеи состоит в разделении всех операций на *длинные* и *короткие*. Короткие операции не конфликтуют друг с другом (то есть соответствующие вершины не соединены ребром в графе зависимостей) и перед выполнением дополнительно с основной блокировкой (семантической или чтения—записи в зависимости от реализации) захватывают блокировки на все необходимые отрезки массива. Длинные же операции конфликтуют со всеми короткими и не требуют захвата дополнительных блокировок. При этом короткая операция чтения не конфликтует с длинной операцией чтения.

Что считать длинной операцией, а что короткой — гиперпараметр, который может быть предметом дополнительной оптимизации; в нашем примере длинной операцией считаются операции, работающие с отрезком длины больше, чем $2^{26}/10^4 = 6710$ элементов. При этом весь массив был разбит на $2 \cdot 10^4$ отрезков, каждый из которых при выполнении короткой операции защищен отдельной блокировкой. Такой подход позволяет захватывать не более двух отрезков при выполнении одной операции.

Подобная оптимизация приводит к гораздо более сложному графу зависимостей, который изображен на рис. 10.

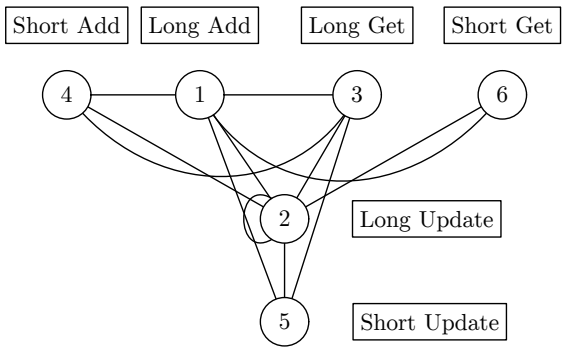


Рис. 10. Граф зависимостей при разделении на долгие и короткие операции

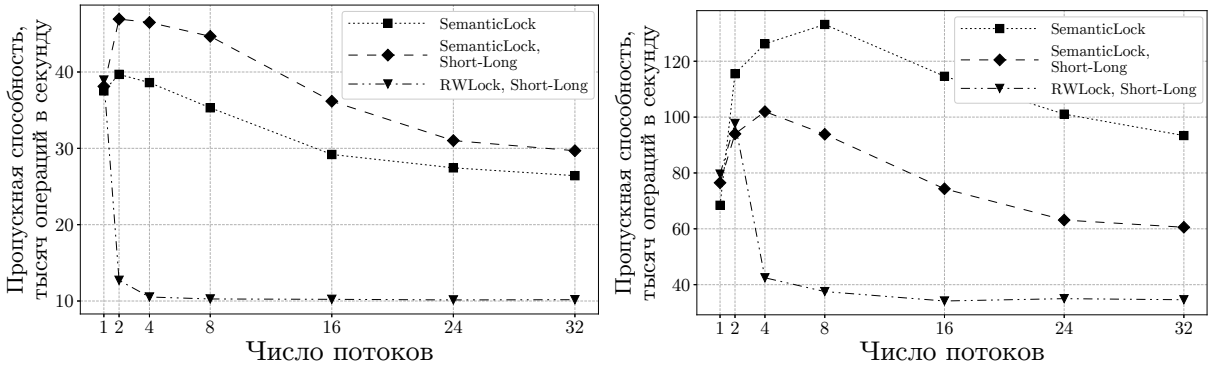
На рис. 11 приведены результаты сравнения трех реализаций: с разбиением на долгие и короткие операции и семантической блокировкой, с разбиением на долгие и короткие операции и блокировкой чтения—записи, без разбиения и с семантической блокировкой.

При равномерном распределении операций использование оптимизации позволило на 10–20% увеличить пропускную способность. Однако при доминировании операции чтения результаты оказались противоположными — использование оптимизации, наоборот, уменьшило пропускную способность на 20–40%.

Отметим при этом следующий результат: использование в таком более сложном графе зависимостей семантической блокировки не привело к заметному усложнению кода, но при этом дало гораздо больший эффект, чем такой же набор операций, синхронизирующийся с помощью блокировки чтения—записи.

4.5. Сравнение различных реализаций массива

В качестве дополнения мы также рассмотрели две возможные реализации атомарности на уровне доступа к элементу в языке Java.



а) равномерное распределение операций

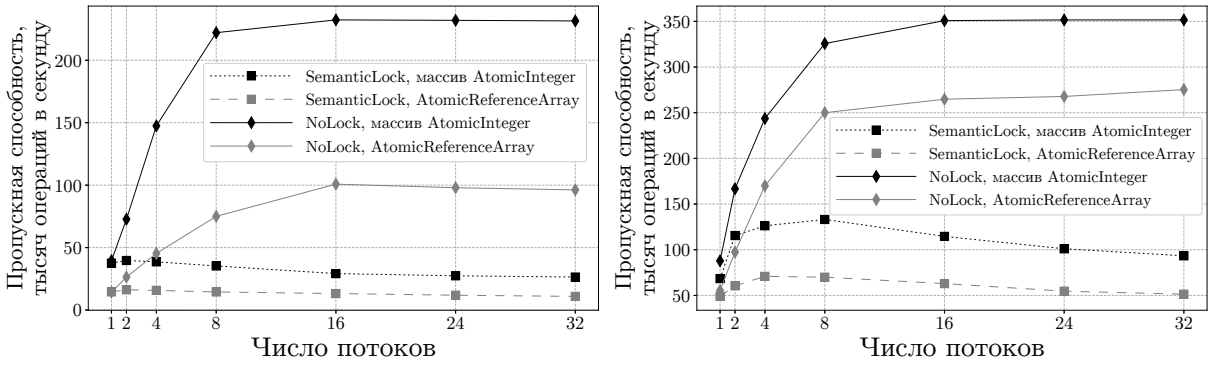
б) 5% setOnRange, 5% addOnRange, 90% getRangeSum

Рис. 11. Использование разбиения на долгие и короткие операции

В экспериментах выше массив представлял собой стандартный массив элементов типа AtomicInteger. Это позволяет обращаться к отдельным элементам, не задумываясь о дополнительных блокировках, что позволяет, например, исполнять параллельные записи во время одновременного исполнения нескольких операций *addOnRange*.

Другой подход основан на использовании класса AtomicReferenceArray, который позволяет атомарно оперировать отдельными элементами памяти; в данном случае, работая с массивом обычных числовых значений, мы позволяем атомарно взаимодействовать с отдельными элементами за счет операций контейнера.

Результаты экспериментов, приведенные на рис. 12, показывают, что подход, основанный на AtomicReferenceArray, оказывается менее эффективным для реализаций с различными видами блокировок.



а) равномерное распределение операций

б) 5% setOnRange, 5% addOnRange, 90% getRangeSum

Рис. 12. Результаты сравнения разных реализаций массива атомарных значений

Заключение

В работе был предложен новый алгоритм семантической блокировки, основанный на анализе графа зависимостей между операциями. Использование такой блокировки показало хорошие результаты в различных экспериментах, моделирующих ситуации, возникающие при реальной работе с потокобезопасными структурами данных. В то же время такая блоки-

ровка предоставляет простой интерфейс для использования, позволяющий без увеличения сложности кода гарантировать корректность синхронизации операций при создании новых структур данных или расширении существующих структур данных новыми операциями.

Дальнейшим направлением исследований по данной теме может стать разработка более эффективных алгоритмов проверки наличия конфликтующей операции в данный момент времени при имеющемся графе зависимостей, автоматизация построения графа зависимостей на основе анализа кода, а также построение математической модели, позволяющей показать преимущество семантической блокировки над стандартными типами блокировок.

Литература

1. Herlihy M.P., Wing J.M. Linearizability: a correctness condition for concurrent objects // ACM Trans. Program. Lang. Syst. 1990. Vol. 12, no. 3. P. 463–492. DOI: 10.1145/78969.78972.
2. Joung Y.-J. Asynchronous group mutual exclusion (extended abstract) // Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing. 1998. P. 51–60. PODC '98. DOI: 10.1145/277697.277706.
3. Yuan He K.G., Gafni E. Group mutual exclusion in linear time and space // Theoretical Computer Science. 2018. Vol. 709. P. 31–47. DOI: 10.1016/j.tcs.2017.05.030.
4. Maor L., Taubenfeld G. Constant RMR Group Mutual Exclusion for Arbitrarily Many Processes and Sessions // 35th International Symposium on Distributed Computing (DISC 2021). Vol. 209. 2021. P. 30:1–30:16. Leibniz International Proceedings in Informatics (LIPIcs). DOI: 10.4230/LIPIcs.DISC.2021.30.
5. Aksenov V., Kuznetsov P., Shalyto A. Parallel Combining: Benefits of Explicit Synchronization // 22nd International Conference on Principles of Distributed Systems (OPODIS 2018). Vol. 125. 2019. P. 11:1–11:16. Leibniz International Proceedings in Informatics (LIPIcs). DOI: 10.4230/LIPIcs.OPODIS.2018.11.
6. Brown T., Prokopec A., Alistarh D. Non-blocking interpolation search trees with doubly-logarithmic running time // Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA, 2020. P. 276–291. PPoPP '20. DOI: 10.1145/3332466.3374542.
7. Blelloch G.E., Wei Y. VERLIB: Concurrent Versioned Pointers // Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. 2024. P. 200–214. PPoPP '24. DOI: 10.1145/3627535.3638501.
8. Lomet D.B. Key Range Locking Strategies for Improved Concurrency // Proceedings of the 19th International Conference on Very Large Data Bases. 1993. P. 655–664. VLDB '93.
9. Eswaran K., Gray J., Lorie R., Traiger I. The Notions of Consistency and Predicate Locks in a Database System // Readings in Artificial Intelligence and Databases. Morgan Kaufmann, 1989. P. 523–532. DOI: 10.1016/B978-0-934613-53-8.50039-X.
10. Badrinath B.R., Ramamritham K. Semantics-Based Concurrency Control: Beyond Commutativity // ACM Trans. Database Syst. 1992. Vol. 17, no. 1. P. 163–199. DOI: 10.1145/128765.128771.
11. Weihl W. Commutativity-based concurrency control for abstract data types // IEEE Transactions on Computers. 1988. Vol. 37, no. 12. P. 1488–1505. DOI: 10.1109/12.9728.

12. Harris T., Larus J., Rajwar R. Software Transactional Memory // Transactional Memory. Springer International Publishing, 2010. P. 101–145. DOI: 10.1007/978-3-031-01728-5_4.

13. Herlihy M., Koskinen E. Transactional boosting: a methodology for highly-concurrent transactional objects // Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2008. P. 207–216. PPoPP '08. DOI: 10.1145/1345206.1345237.

14. Koval N., Fedorov A., Sokolova M., *et al.* Lincheck: A Practical Framework for Testing Concurrent Data Structures on JVM // Computer Aided Verification. 2023. P. 156–169. DOI: 10.1007/978-3-031-37706-8_8.

15. Courtois P.J., Heymans F., Parnas D.L. Concurrent control with "readers" and "writers" // Commun. ACM. 1971. Vol. 14, no. 10. P. 667–668. DOI: 10.1145/362759.362813.

16. Potapov A., Zuev M., Moiseenko E., Koval N. Testing Concurrent Algorithms on JVM with Lincheck and IntelliJ IDEA // Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2024. P. 1821–1825. ISSTA 2024. DOI: 10.1145/3650212.3685301.

17. Коротченко Д.С. SemanticLock. URL: https://github.com/ITMO-PTDC-Team/SemanticLock/tree/PCT-2026 (дата обращения: 28.07.2026).

18. Gramoli V. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms // Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2015. P. 1–10. PPoPP 2015. DOI: 10.1145/2688500.2688501.

Коротченко Денис Сергеевич, аспирант, институт прикладных компьютерных наук, Национальный исследовательский университет ИТМО (Санкт-Петербург, Российская Федерация)

Аксенов Виталий Евгеньевич, PhD, доцент, институт прикладных компьютерных наук, Национальный исследовательский университет ИТМО (Санкт-Петербург, Российская Федерация)

DOI: 10.14529/cmse260205

SYNCHRONIZATION IN CONCURRENT DATA STRUCTURES BASED ON ANALYSIS OF DEPENDENCIES BETWEEN OPERATIONS

© 2026 D.S. Korotchenko, V.E. Aksenov

ITMO University (pr. Kronverkskiy 49A, St. Petersburg, 197101 Russia)

E-mail: korotchenkods@gmail.com, aksenov.vitaly@gmail.com

Received: 28.07.2026

This paper examines the synchronization of operations in thread-safe data structures. In different data structures, operations may conflict with each other in different ways in terms of concurrency. For example, a standard Read-Write lock recognizes that concurrent read accesses from different threads are often permissible, while parallel writes from different threads or parallel reads and writes most often require synchronization. This dependency can also be more complex: for example, incrementing all elements in a container by a given value is permissible with synchronization at the element access level, while concurrently assigning one number from one thread and a different number from another thread to all elements of a container requires more complex synchronization. This paper examines various methods for organizing synchronization between operations. The main result is a proposed new approach to ensuring such synchronization, based on the analysis of the dependency graph between operations, which ensures more efficient synchronization of the structure. Experimental results comparing various approaches are also presented, demonstrating the effectiveness of the proposed solution.

Keywords: thread-safe, concurrent data structures, dependency graph, semantic lock, linearizability.

FOR CITATION

Korotchenko D.S., Aksenov V.E. Synchronization in Concurrent Data Structures Based on the Analysis of Dependencies between Operations. Bulletin of the South Ural State University. Series: Computational Mathematics and Software Engineering. 2026. Vol. 15, no. 2. P. 82–103. (in Russian) DOI: 10.14529/cmse260205.

References

1. Herlihy M.P., Wing J.M. Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 1990. Vol. 12, no. 3. P. 463–492. DOI: 10.1145/78969.78972.
2. Joung Y.-J. Asynchronous group mutual exclusion (extended abstract). Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing. 1998. P. 51–60. PODC '98. DOI: 10.1145/277697.277706.
3. Yuan He K.G., Gafni E. Group mutual exclusion in linear time and space. Theoretical Computer Science. 2018. Vol. 709. P. 31–47. DOI: 10.1016/j.tcs.2017.05.030.
4. Maor L., Taubenfeld G. Constant RMR Group Mutual Exclusion for Arbitrarily Many Processes and Sessions. 35th International Symposium on Distributed Computing (DISC 2021). Vol. 209. 2021. P. 30:1–30:16. Leibniz International Proceedings in Informatics (LIPIcs). DOI: 10.4230/LIPIcs.DISC.2021.30.

5. Aksenov V., Kuznetsov P., Shalyto A. Parallel Combining: Benefits of Explicit Synchronization. 22nd International Conference on Principles of Distributed Systems (OPODIS 2018). Vol. 125. 2019. P. 11:1–11:16. Leibniz International Proceedings in Informatics (LIPIcs). DOI: 10.4230/LIPIcs.OPODIS.2018.11.

6. Brown T., Prokopec A., Alistarh D. Non-blocking interpolation search trees with doubly-logarithmic running time. Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA, 2020. P. 276–291. PPoPP '20. DOI: 10.1145/3332466.3374542.

7. Blelloch G.E., Wei Y. VERLIB: Concurrent Versioned Pointers. Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. 2024. P. 200–214. PPoPP '24. DOI: 10.1145/3627535.3638501.

8. Lomet D.B. Key Range Locking Strategies for Improved Concurrency. Proceedings of the 19th International Conference on Very Large Data Bases. 1993. P. 655–664. VLDB '93.

9. Eswaran K., Gray J., Lorie R., Traiger I. The Notions of Consistency and Predicate Locks in a Database System. Readings in Artificial Intelligence and Databases. Morgan Kaufmann, 1989. P. 523–532. DOI: 10.1016/B978-0-934613-53-8.50039-X.

10. Badrinath B.R., Ramamritham K. Semantics-Based Concurrency Control: Beyond Commutativity. ACM Trans. Database Syst. 1992. Vol. 17, no. 1. P. 163–199. DOI: 10.1145/128765.128771.

11. Weihl W. Commutativity-based concurrency control for abstract data types. IEEE Transactions on Computers. 1988. Vol. 37, no. 12. P. 1488–1505. DOI: 10.1109/12.9728.

12. Harris T., Larus J., Rajwar R. Software Transactional Memory. Transactional Memory. Springer International Publishing, 2010. P. 101–145. DOI: 10.1007/978-3-031-01728-5_4.

13. Herlihy M., Koskinen E. Transactional boosting: a methodology for highly-concurrent transactional objects. Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2008. P. 207–216. PPoPP '08. DOI: 10.1145/1345206.1345237.

14. Koval N., Fedorov A., Sokolova M., *et al.* Lincheck: A Practical Framework for Testing Concurrent Data Structures on JVM. Computer Aided Verification. 2023. P. 156–169. DOI: 10.1007/978-3-031-37706-8_8.

15. Courtois P.J., Heymans F., Parnas D.L. Concurrent control with "readers" and "writers". Commun. ACM. 1971. Vol. 14, no. 10. P. 667–668. DOI: 10.1145/362759.362813.

16. Potapov A., Zuev M., Moiseenko E., Koval N. Testing Concurrent Algorithms on JVM with Lincheck and IntelliJ IDEA. Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2024. P. 1821–1825. ISSTA 2024. DOI: 10.1145/3650212.3685301.

17. Korotchenko D.S. SemanticLock. URL: https://github.com/ITMO-PTDC-Team/SemanticLock/tree/PCT-2026 (accessed: 28.07.2026) (in Russian).

18. Gramoli V. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2015. P. 1–10. PPoPP 2015. DOI: 10.1145/2688500.2688501.