Содержание

Список сокращений и условных обозначений

- CAS - Compare And Swap
- MWCAS - Multi Word Compare And Swap
- ОС - Операционная Система
- MPMC - Multiple Producer Mulptiple Consumer
- ЭВМ - Электронная вычислительная машина

Введение

На сегодняшний день в основе большинства разрабатываемых приложений и систем: веб-серверов, кластеров обработки данных, блокчейн сетей и тому подобных лежит инфрастуктурная основа в виде надёжной и производительной среды предоставления асинхронного исполнения - runtime исполнения асинхронных задач.

От него требуется обеспечение эффективной обработки задач, предоставления инструментов композиции и коммуникации между ними. Такая основа может быть встроена в сам язык, как например в Elixir и Go, или использоваться как отдельная библиотека. Примером такого подхода являются библиотеки Kotlin Coroutines и Rust Tokio.

Любой такой инструмент строит свои абстракции исполнения поверх процессов или потоков, предоставляемых операционной системой, поэтому во время его использования возникает потребность в синхронизации между ними. Особенно эта проблема актуальна для примитивов синхронизации предоставляемым данными инструментами.

Синхронизация по своей природе может быть нескольких типов. Каждый из них предоставлет как преимущества, так и недостатки. Современные архитектуры имеют тенденцию распараллеливать вычислетельные процессы. Однако в большом числе случаев для синхронизации до сих пор используются инструменты крайне неэффектино масштабирующиеся вслед за архитектурой процессоров ЭВМ. В большинстве случаев - это блокирующая синхронизация, несмотря на то, что существуют способы производить операции в неблокирующем режиме, который открывает возможности к существенному масштабированию вычислений. Связано это с тем, что неблокипющая синхронизация довольно сложна в реализации, в отличие

от блокирующей, а для комплексных структур данных эффективная и простая реализация становится практически невозможной.

В рамках данной работы будет рассматриваться асинхронная среда Tokio. В рам рассмотрена возможная оптимизация для примитивов синхронизации фреймворка Tokio в среде языка Rust с использованием такого типа синхронизации.

1 Цели и задачи работы

Целью работы является оптимизация модуля широковещательной рассылки фреймворка Tokio.

Для выполнения цели были выделены следующие задачи:

- Определить оптимальный дизайн очереди
- Реализовать полученную модель
- Провести сравнительное тестирование

2 Введение в предметную область

2.1 Актуальность и практическая значимость

Фреймворк Tokio на сегодняшний день является основным инструментом в экосистеме языка Rust для построения асинхронных приложений. Он предоставляет разработикам приложений и систем, не только среду исполнения асинхронных задач, но также и примитивы синхронизации, которые могут быть использованы для реализации задач коммуникации между исполняемыми потоками.

Так как Tokio в общем виде представляет собой среду исполнения, спроектированную для исполнения на многоядерных процессорах. Его планировщик а также инстументы, поставляемые вместе с ним, должны поддерживать соответсвующее масштабирование. Сам планировщик и часть примитивов синхронизации уже поддерживают эффективное масштабирование, однако остаётся часть, которая использует для этого неэффективные техники.

Одни из таких инструментов, канал, предоставляющий семантику широковещательной рассылки, при применении которого, отправленное сообещение должны увидеть все потоки, подписавшиеся на рассылку сообщений. В рамках работы производится исследование возможных оптимизаций данного примитива синхронизации.

Актуальность добавляемых оптимизаций обуславиливается возможностью повышения производительности предоставляемых фреймворком Tokio инструментов, что повысит производительность клиентских приложений, использующих его в качестве своей основы.

2.2 Многопоточная синхронизация

Существует несколько общих подходов к построению многопоточной синхронизации. Каждый из них предоставляет пользователю слудующие параметры:

- Гарантии прогресса многопоточного исполнения (liveness)
- Степень масштабирования вычислений
- Простота реализации
- Платформенная / аппаратная поддержка

Эти подходы можно разделить на три больших класса:

- Блокирующая синхронизация
- Неблокирующая синхронизация
- Транзакционная память

На данный момент не существует универсального: простого и одновременно эффективного способа синхронизации. Вместе с каждым классом возникают как свои определённые достоинтсва, так и недостатки, выражающиеся в отсутвие одного или нескольких из перечисленных параметров.

Далее будут подробно рассмотрены все классы синхронизации.

2.2.1 Блокирующая синхронизация

Первый рассматриваемый класс представляет собой способ построения синхронизации вокруг критических секций - участков программы, одновременное исполнение которых возможно только одним, в случае модификации состояния, или несколькими определёнными выделенными потоками, в случае чтения. При этом происходит блокирование прогресса всех остальных исполняемых задач до момента завершения выполнения выделенных задач.

У такого типа синхронизации есть большое преимущество - он простой и не требует специального подхода к построению структуры данных над, которой производятся операции. Также при относительно небольшом числе параллельных задач, выполняющих критические секции, достигается оптимальное использование ресурсов, необходимых для координации задач.

Однако данный подход крайне неэффективно масштабируется, так как в единицу времени может выполняться только одна критическая секция. Остальные потокам необходимо ждать освобожнение блокировки. Это время может стать очень продолжительным из-за размера очереди ожидания. Ожидание может быть сопряжено с дополнительными системными вызовами, например с futex syscall. Или же могут возникать затраты на переключение контекста и координацию в очередях ожидиания, в случае использования корутин поверх потоков.

Если присутствует большое число потоков оперирующих над критической секцией, появляется большое число подобных накладных расходов. В худшем случае при таком использовании блокирующей синхронизации число исполняемых критических мекций в единицу времени может быть меньше, чем если бы они испольнялись последовантельно одним ядром процессора. Отдельно стоит также

сказать о проблеме, при которой поток или процесс захвативший блокировку и исполняющий критическую секцию, будет временно снят с исполнения планировщиком задач операционной системы до момента снятия блокировки, например по истечению выделенного ему временного кванта на выполнение. В данном случае возникает риск полной остановки прогресса исполнения системы до конечной разблокировки.

Самый простой пример такой синхронизации - использование мьютекса:

```
//  Располагается  в  общей  для  потоков  памяти
let   mutex   =   Mutex::<State>

//  Вызывается  разными  потоками
fn  doCriticalSection(){
     {
     mutex.lock()
          //  Критическая  секция
          mutex.unlock()
     }
}


fn  main()  {
     let  thread_1  :=  spawn(doCriticalSection)
     let  thread_2  :=  spawn(doCriticalSection)

     thread_1.join();
     thread_2.join();
}
```

2.2.2 Неблокирующая синхронизация

Для избавления от проблем присущих блокирующей синхронизации, существует альтернативный подход, при котором, операции над данными осуществляются в неблокирующем режиме. При этом исчезает понятие критической секции.

Неблокирующая синхронизация предоставляет следующие преимущества по сравнению с блокирующей:

- Гарантия прогресса системы в целом - означает, что при любом многопоточном исполнении, всегда есть поток или потоки успешно завершающие свои операции. Решения планировщика ОС теперь не могут привести к полной остановке системы.

- Сравнительно высокая масштабируемость по сравнению с блокирующей синхронизацией - при использовании неблокирующей синхронизации потоки не обязаны ждать друг друга, поэтому операции могут работать в параллель. Вся координация между потоками образуется в специальных точках синхронизации. В большинстве языков программирования - это атомарные переменные.

- Уменьшение накладных расходов на синхронизацию. Использование атомарных переменных не требует обращения к ядру операционной системы. Операции над атомарными переменными напрямую упорядочивают исполнение через L2 кэш ядер процессора, за счёт чего достигается синхронизация памяти ядер.

2.2.3 Построение неблокирущей синхронизации

При использовании неблокирующей синхронизации изчезает понятие критической секции. Вместо этого появляется понятие транзакции, совершаемой над состоянием структуры данных. Транзакция представляет собой серию операций чтения и записи в определённые ячейки памяти. Эти ячейки памяти - атомарны

Выделяются специальные операции над атомарными ячеейками памяти.

- read - чтение
- store - запись
- cas - условная запись
- Иные операции в зависимости от архитектуры процессора

Главная особенность эти операций в том, что они определяют порядок обращений потоков к определённой ячейке, за счёт чего достигается синхронизация.

В большинстве случаев образуется общий подход при построении структуры даннных и операций над ней - необходима модификация структуры на основе её текущего состояния. Для синхронизации выделяется общее состоянии, которое становится атомарной ячейкой памяти, в том плане, что все операции над ней линеаризуемы и образуют некоторый порядок обращений.

Все операции абстрактно в рамках атомарной транзакции разбиваются на три этапа:

1. Копирование текущего состояния (snapshot).
2. Локальная модификация полученного состояния.

3. Попытка замена общего состояния на модифицированную копию, в случае, если общее состояние за время модификации не изменилось. Если состояние успело измениться - начать заного с шага №1.

В псеводокоде это можно представить так:

```
//  Располагается  в  общей  для  потоков  памяти
let  state  =  Atomic<State>

//  Вызывается  из  разных  потоков
fn  doLockFreeOperation(){
while(true){
    let  old_state  =  state.atomic_read()

    let  modified_state  =  modify(old_state)

    if(state.atomic_cas(old_state,modified_state)){
        //  За  время  транзакции  состояние  не
изменилось
        //  Поток  успешно  завершает  транзакцию
        break;
    }  else{
            //  Операция  по  замене  неуспешна
            //  Поток  повторяет  цикл
        continue;
    }
}
}

fn  main()  {
    let  thread_1  =  spawn(doLockFreeOperation)
    let  thread_2  =  spawn(doLockFreeOperation)

    thread_1.join();
```

```
        thread_2.join();
}
```

Очевидно, что при таком подходе обязательно будет существовать поток или потоки, успешно завершающиие свои транзакции, при этом остальным потокам нужно будет лишь повторить попытку. Синхронизация в описанном примере происходит в точках `state.atomic_read()` и `state.atomic_cas()`. При этом модификация состояния может происходить в параллель.

Несмотря на свои плюсы, такой подход обладает одним серъездным недостатком. Необходимая линеаризуемость и следующая из неё синхронизация, образуется лишь вокруг одной ячейки памяти. Однако в большинстве структур данных чаще всего требуется атомарная замена сразу нескольких ячеек памяти. Любые прямые изменения хотя бы двух ячеек памяти влекут за собой потерю порядка исполнения, так как между двумя атомарными операциями может произойти произвольное число сторонних событий, в зависимости от решения приоритета исполнения планировщика операционной системы.

Для решения описанной проблемы была представлена транзакционная память. Её можно реализовать как на уровне процессора, так и программно. Так как сейчас процессоры не поддерживают подобную опцию, будет рассмотрено использование программной реализации в виде примитива MWCAS.

2.2.4 Обзор MWCAS

MWCAS обобщает подход транзакции над одной ячейкой памяти до произвольного их числа.

2.2.5 Принцип использования MWCAS

Пример исполнения транзакции может выглядеть следующим образом:

```
// Ячейки располагаются в общей для потоков памяти
let state_1 = Atomic<State>
let state_2 = Atomic<State>

// Вызывается из разных потоков
fn doMultiTransaction(){
while(true){
      let old_state1 = state_1.atomic_read()
      let old_state2 = state_2.atomic_read()

      let modified_state_1 = modify(old_state_1)
      let modified_state_2 = modify(old_state_2)

      let mwcas = new Mwcas

      // Транзакция атомарно заменяет ожидаемые значения
            // на модифицированные копии.
            // В случае, если наблюдаемое старое
значение изменилось
            // Транзакция помогает завершиться другой
возможной транзакции
            // и сообщает о неуспешном завершении
      if(mwcas.transaction(
      memory_cell = [state_1, state_2]
      expected_states = [old_state1, old_state2],
      new_states = [modified_state_1, modified_state_2],
      )){
            break;
      } else{
               // Транзакция прошла неуспешно
               // Поток повторяет цикл
```

```
            continue;
        }
}
}

fn   main()   {
        let   thread_1   :=   spawn(doMultiTransaction)
        let   thread_2   :=   spawn(doMultiTransaction)

        thread_1.join();
        thread_2.join();
}
```

3 Обзор модуля широковещательной рассылки Tokio

Модуль широковещательной рассылки фреймворка tokio представляет собой канал для пересылки сообщений между между потоками программы в режиме доступа Multiple Producer Multiple Consumer (MPMC): в канал конкуррентно могут одновременно отправлять сообщения сразу несколько потоков. При этом каждое значение доступно для чтения всем подписавшимся на момент отправки сообщения потокам читателям, достигается это за счёт дополнительного счётчика, устанавливаемого вместе с сообщением. Счётчик обновляется с добавлением или удалением потоков-читателей. При определённых обстоятельствах медленный поток-читатель может пропустить некоторые сообщения. В таком случае он переходит на последние актуальные сообщения. В случае если сообщений нет, поток-читатель становится в очередь ожидания значения.

4 Внутреннее состояние

С точки зрения программной реализации канал представляет собой общее состояние в памяти, обращения к которому совершаются с помощью интерфейсов структур двух типов: Sender и Reciever. Каждая структура предоставляет лёгковесный способ управления каналом через определённый интерфейс.

```
pub struct Sender<T> {
    // Ссылка на общее состояние
        shared: Arc<Shared<T>>,
}

pub struct Receiver<T> {
    // Ссылка на общее состояние
```

```
            shared:   Arc<Shared<T>>,

            // Локальная позиция для следующего чтения
            next:   u64,
}
```

Общее состояние содержит в себе:

1. Память самой очереди: она представляется в виде обычного массива слотов (структура типа Slot), логически представленного в виде кольцевого буффера (buffer). Каждому слоту в соотетсвие ставится RwLock - блокирующий примитив синхронизации позволяющий производит параллельное чтение, при отсутвии записи.

Каждый слот содержит в себе:

- Текущее соообщение (val)
- Ассоциированное сило потоков читателей, для которых доступно сообещение (rem)
- Логическую позицию слота (pos)

2. Битовая маска для быстрого определения позиции (mask). При создании канала, его длина округляется до ближайшей степени двойки.

3. Основная информация для координации операций над очередью, представлена структурой Tail, это:
    1. Логическая позиция нового следующего сообщения (pos)
    2. Текущее число потоков-читателей (rx_cnt)
    3. Состояние канала (открыт / закрыт) (closed)
    4. Очередь (связный список) ожидания следующего значения (waiters)

Синхронизация доступа к этим полям осуществляется с помощью мьютекса. Вокруг этой точки происходит осно

4. Текущеее число потоков-отправителей (num-tx)

5. Примитив оповещения о закрытии последнего потока-читателя (notify_last_rx_drop)

```
struct Shared<T> {
    buffer: Box<[RwLock<Slot<T>>]>,

    mask: usize,

    tail: Mutex<Tail>,

    num_tx: AtomicUsize,

    notify_last_rx_drop: Notify,
}

struct Tail {
    pos: u64,

    rx_cnt: usize,

    closed: bool,

    waiters: LinkedList<Waiter, <Waiter as
linked_list::Link>::Target>,
}

struct Slot<T> {
    rem: AtomicUsize,

    pos: u64,
```

```
            val:   UnsafeCell<Option<T>>,
}
```

5 Алгоритм канала

Массив слотов ограничен в длине значением, задаваемым пользователем. В этом возникает необходимость из-за проблемы следующей из архитектуры канала: так как значение должно быть доставлено всем получателям, оно должно всё это время оставаться в памяти, и, если для каждого нового сообщения будет выделяться новая память, наличие лишь одного медленного потока-читателя может привести к переполнению памяти.

Две основные операции канала - отправление и получение сообщений.

Операция отправления - синхронная. Потоки-писатели не обязаны ждать потоков-получателей. В связи упомянутой раньше проблемой, при которой выделение памяти может привести к её неконтролируемогу росту, буффер очереди ограничен, и при этом новые сообщения, превышающие длину очереди перезаписывают старые. Потоки-читатели, не успевшие получить отправленные ранее сообщения, с помощью своего собственного локального счётчика позиции сообщений, обнаруживают неконсистентность и переводят счётчик на текущую актуальную позицию в очереди, добавляя к нему один круг длины очереди.

Псевдокод операции:

```
fn   send(newValue){
      //   Захват   общей   блокировки   очереди
      tail.lock()

      //   Захват   блокировки   слота,   в   который   будет
произведена   запись
      buffer[nextId].lockWrite()

      //   Запись   нового   значения
      buffer[nextId].value   =   newValue

      //   Обновление   мета-информации   и   необходимых
счётчиков
      ...

      //   Освобождение   блокировки   слота
      buffer[nextId].unlock()
      //   Запуск   на   исполнение   всех   ожидающих   задач
потоков-читателей
      queue.notify_all()

      //   Общая   разблокировка   очереди
      tail.unlock()
}
```

Операция получения - асинхронная. При попытке получить сообщение, поток-читатель может обнаружить ситуацию при которой готовых сообщений в очереди нет. Это может произойти как сразу при совпадении локального счётчика потока-отправителя и счётчика позиции слота, так и после добавления дополнительного круга длины, в случае, описанном ранее. В этом случае поток-читатель добавляет задачу на очередную проверку очереди в специальную очередь ожидания. Потоки-отправители после отправки сообщений проверяют эту очередь и запланирую на исполнение все оставшиейся в ней задачи.

Псевдокод выполнения операции:

```
// taskHandler - структура, отвечающая за планирование
// задачи, вызвавшей recv, на исполнение

async fn recv(taskHandler) -> T {

    // Следующий слот на чтение значения
    let slot = buffer[nextId]

    // Блокировка слота на чтение
    // Допускается параллельное чтение
    slot.lockRead()

    // Поток-читатель отстал от актуальной информации
    // как минимум на один круг очереди
    if slot.pos != self.next {

        // Блокировка состояния канала
        tail.lock()

        // Добавление одного круга очереди
        let next_pos = slot.pos + buffer.len()

        // Позиция потока-читателя совпадает
        // с суммой позициии слота и длины
одного круга очереди
        // В таком случае готового значения ещё
нет.
        // Поток оставляет задачу в очереди
ожидания
        if self.next == next_pos{
            qeueu.park(taskHandler)
        }
```

```
            //  Поток-читатель  утанавливает  указатель
            //  на  самое  старое  текущее  значение  в
канале
            //  Все  остальные  значения  считаются
пропущенными
            let  next  =  tail.pos  -  buffer.len()
            let  missed  =  next  -  self.next
            let  self.next  =  next

            //  Блокировка  состояния  канала  и  слота
            slot.unlockRead()
            tail.unlock()

      }  else{
            //  Если  первая  проверка  показала,
            //  что  текущее  значение  self.next  совпадает
с  позицией  слота,
            //  это  означает,  что  поток  читает
актуальную  информацию,
            //  В  таком  случае  он  инкрементирует
локальный  счётчик  и  возвращает  значение
            let  result  =  slot.value
            slot.unlock()
            self.next++
            return  result
      }


}
```

Новый способ синхронизации позволит полностью избавиться от блокирующих примитивов Mutex и RwLock

Анализ оптимального дизайна для очереди

6 Использование памяти

Существует два возможных варианта использования выделения памяти под сообщения:

1. Выделять под каждое новое сообщение новую ячейку памяти

2. Выделить ограниченное число ячеек с перезаписью (используется сейчас)

Использование первого варианта гарантирует то, что ни одно сообщение не будет потеряно. При этом, несмотря на то, что можно аллоцировать память группами (аллоцировать сразу несколько ячеек), это не избавит в принципе от необходимости данных аллокаций, что может существенно повлиять на производительность. Кроме того, при наличие медленных потоков-читателей можно прийти к ситуации переполнения памяти.

При использовании второго способа присутсвует только одна аллокация - при инициализации буффера памяти и при этом отчутсвует проблема переполнения памяти.

Так как риск потери сообщений и последующая обработка таких случаев, возложенная на потоки - читатели, во втором случае существенно меньше риска потери функционала программы в целом в первом случае, второй вариант предпочтителен.

7 Оптимизация с помощью неблокирующей синхронизации

Перед потоками-отправителелями становится задача атомарно произвести следующие операции:

- [Событие А] Обновить состояние следующего слота. Включает также обновление глобального счётчика позиции слотов.
- [Событие Б] Запланировать на исполнения все существующие в очереди ожидания задачи потоков-читателей, ожидающих нового значения

Потокам-читателям необходимо атомарно произвести следующие операции:

- [Событие В] Проанализировать значение последнего актуального слота
- [Событие Г] В случае если актуальных слотов нет, добавить задачу на повторную проверку в очередь ожидания

Применении классической неболокирующей синхронизации с использованием одной атомарной переменной подразумевает создание единой точки синхронизации вокруг которой будут упорядочиваться обращения к структуре данных. Этого можно достичь путём создания указателя (атомарные операции возможны над максимальной битовой размерностью указателя архитектуры) на общее состояние.

Проблема возникает в том, что для такого подхода необходима полная копия глобального состояния, включающее копию всей очереди ожидания и всего буффера очереди, что при большом числе потоков-читателей и/и/или большом размере буффера, становится нецелесообразно с точки зрения эффективного использования.

Если же попробовать разделить состояние на несколько атомарных ячеек: Сделать отдельный указатель на очередь, а также отдельные указатели на каждый из слотов буффера, что решит проблему полного копирования буффера и очереди ожидания, может произойти следующая ситуация -

Если события А, Б, В, Г, описанные выше произойдут для двух потоков, один из которых поток-читатель, а другой отправитель, в следующем порядке: В, А, Б, Г, что допускается при наличии нескольких атомарных переменных, возникнет ситуация при которой состояние канала, успеет обновиться потоком-отправителем до постановки задачи потока-читателя в очередь исполнения. В итоге поток-читатель оставит задачу на повторную проверку в очереди, которая никогда уже не будет запланирована на исполнение.

Сравнительный анализ производительности

Описание тестов …

Блокирующая реализации broadcast

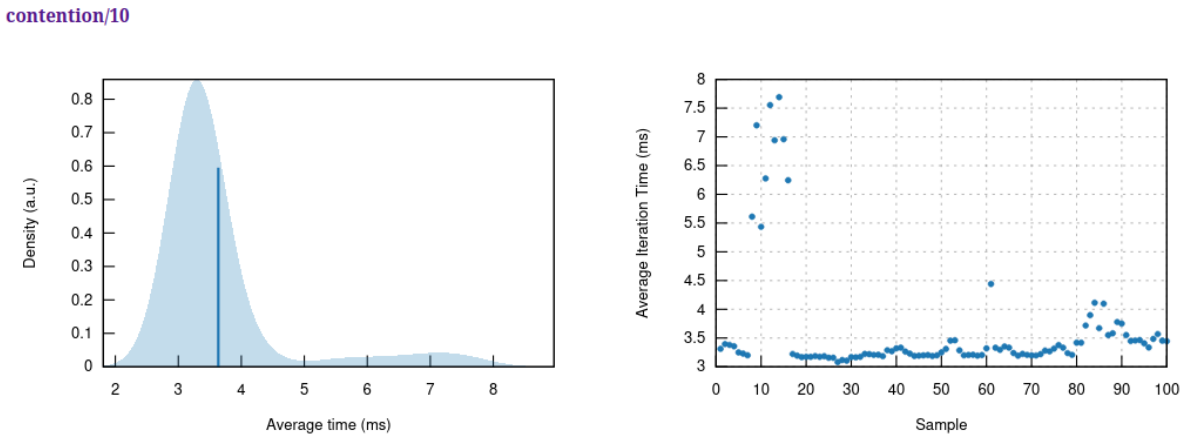

Рисунок 1 — пример изображения

contention/100

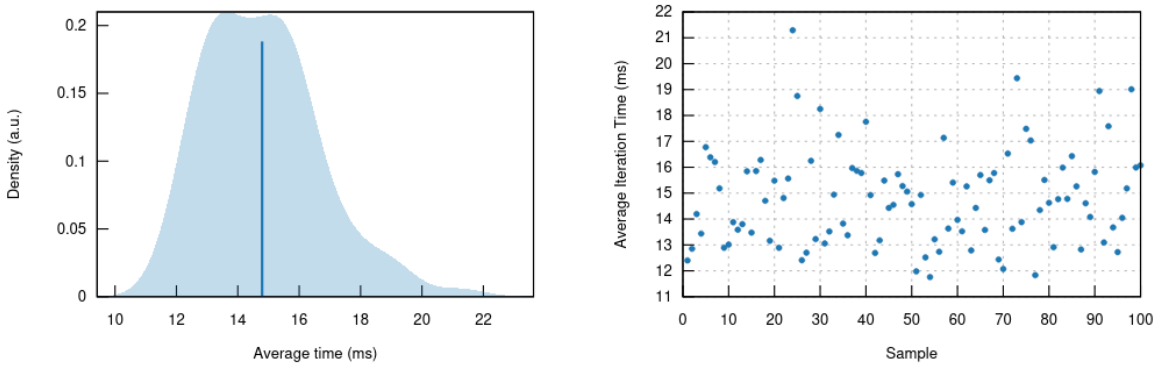

Рисунок 1 — пример изображения

contention/1000

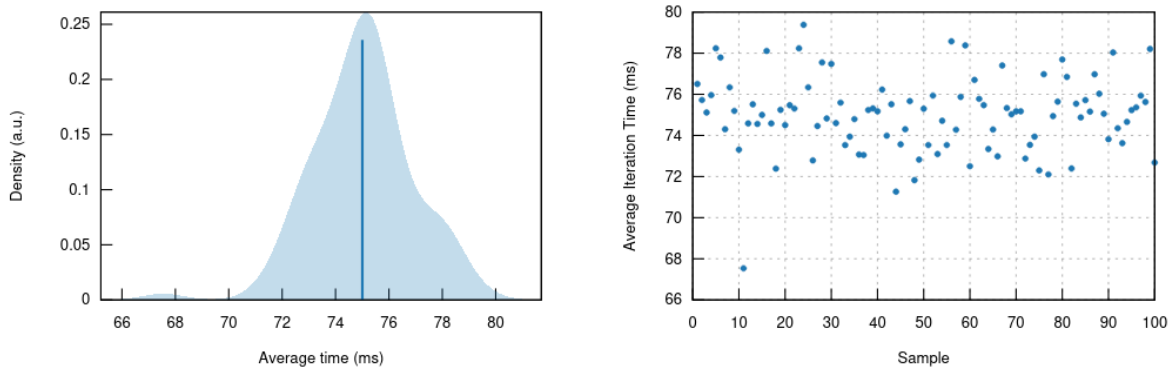

Рисунок 1 — пример изображения

contention/10000

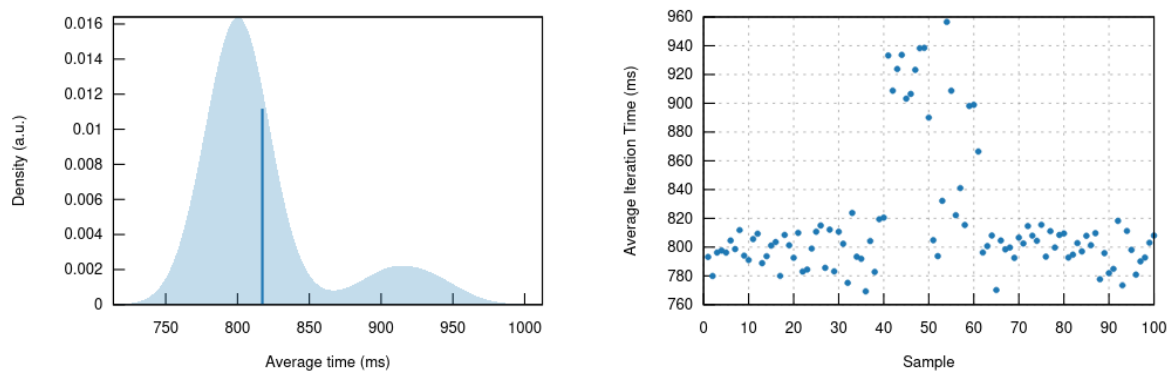

Рисунок 1 — пример изображения

Список использованных источников

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.

Приложение

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.