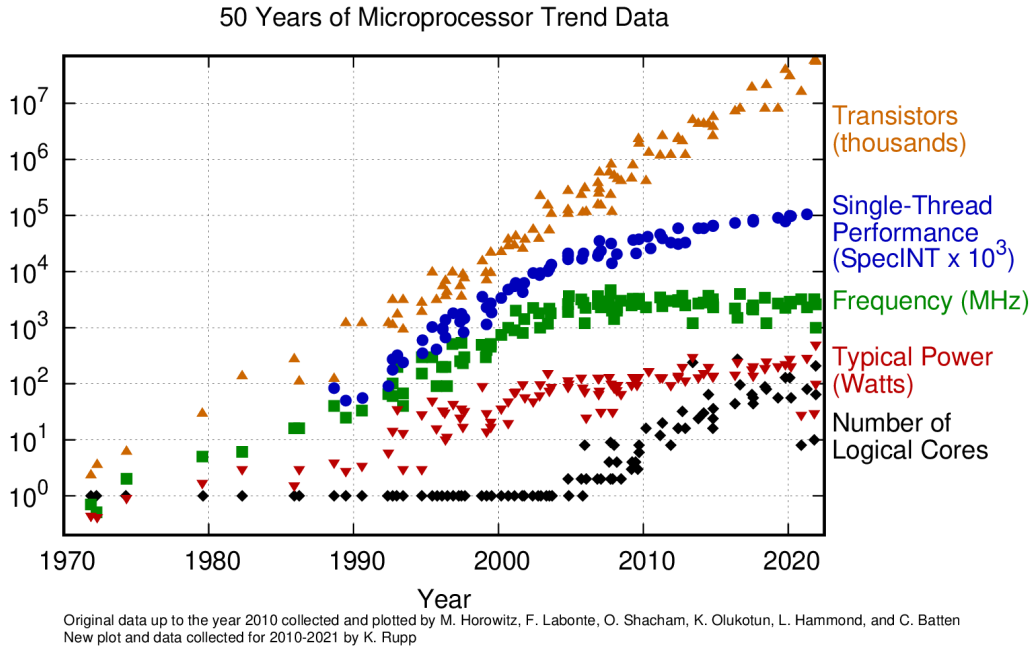
1 Введение.

Собственно, зачем это всё? У нас во всех наших системах (включая мобильные) есть множество ядер, и чтобы ваш код эффективно работал, необходимо уметь распределять его по всем этим ядрам.
К сожалению, в наше время закон Мура выполнен, но выполнен не так, как нам хотелось бы:



Количество транзисторов всё ещё экспоненциально растёт, но если раньше это осуществлялось расположением всё большего количества транзисторов в одном ядре (чтобы увеличить производительность этого самого одного ядра), то сейчас у нас просто становится больше ядер, а производительность одного ядра растёт очень медленно.
Раньше вообще было просто жить: ты написал программу, и через несколько лет она ускоряется вдвое, потому что железо работает быстрее. А сейчас бесплатные ленчи кончились, приходится самим писать код, который будет масштабироваться на множество ядер.
Казалось бы, ну напиши код как-нибудь, и с ростом количества ядер он будет ускоряться. Да вот тут проблема: не любое место кода может быть параллельным. И простая арифметика даёт нам очень грустные результаты: пусть мы можем распараллелить только долю p нашего кода (а $1-p$ останется последовательным). Тогда если нам дали n ядер, то количество раз, в которое наш код ускорится, равно

$$\frac{1}{1-p+\frac{p}{n}}$$

Это называется законом Амдала.
И отсюда следует, что если нам дали 16 ядер и мы имеем 60% кода параллельно, то ускорится он всего в 2.3 раза. Более того, если устремить n к бесконечности, мы получим максимальное ускорение кода $\frac{1}{1-p}$. Например, если доля параллельного кода 95%, то он не сможет ускориться больше, чем в 20 раз. А если 99%, то в 100 раз, что тоже очень грустно, потому что в серверных системах ядер дохера и больше, и всего в 100 раз — очень грустно. Поэтому надо реально очень сильно параллелить код.
Виды параллелизма.

- Instruction-level parallelism. Это когда вы имеете два действия, которые не связаны друг с другом,

и они автоматически процессором параллелятся. Сюда и конвейеры со спекулятивным исполнением, и Superscalar, и VLIW, и векторизация даже. Про всё, кроме последнего, можно почитать в конспекте ассемблера. Именно за счёт всего этого и выполнялся экспоненциальный рост производительности одного процессора. А перестал он расти потому, что у этого всего есть предел.

- Собственно, многопроцессорные системы. Работает хорошо, но надо явно параллелить самому.

Многопроцессорные системы:

- Simultaneous multiprocessing — есть несколько процессов и шина. Туда же simultaneous threading, в котором процессор один, и в нём потоки, который для вас выглядит ровно как SMP.
- На деле используется non-uniform memory access (NUMA). Для вас это всё ещё выглядит как SMP, но когда вы знаете вашу архитектуру, вы можете писать специальные алгоритмы.

Операционные системы:

- Однозадачные.
- Пакетные задания (batch processing).
- Многозадачные / с разделением времени. Такие создают для программиста иллюзию, что его программа — единственная. Такая иллюзия достигается вытесняющей многозадачностью. Исторически до неё была кооперативная многозадачность, где программист вставлял в свой код специальные команды, чтобы отдать время другим процессам. Сейчас ОС с кооперативной многозадачностью нет, но как программисты мы всё ещё можем жить с кооперативной многозадачностью: у нас есть одно приложение, и там физика, анимация, куча другой фигни, — и чтобы это написать, мы пишем кооперативную многозадачность.

Процесс — это сущность ОС, которые независимы, и именно для процессов ОС делает иллюзию, что они одни.
Поток — контекст исполнения внутри процесса.
Но тут проблема: теория многопоточки сильно древнее, чем многопоточка. Поэтому в теории термином «процесс» называется то, что на самом деле поток.

2 Формализм.

Нужна формальная модель параллельных вычислений. Зачем? Чтобы доказывать корректность алгоритмов и и невозможность построения других алгоритмов. Но ещё он нужен, чтобы вы как программист понимали, что вам обещает разработчик языка, и что вы хотите от него.
Какие у нас модели бывают? Ну, у нас раньше были многочисленные однопоточные модели (RAM-модель, pointer-machine, машина тьюринга, куча другого мусора). Но у нас всё это одно и то же, у нас есть совершенно не важно, какие вычислители, нам важно лишь, как эти вычислители взаимодействуют.

- Модель с общей памятью. Именно об этом мы и будем думать на этом курсе, потому что многоядерники в жизни так и работают. Да, на самом деле внутри каждого процесса передача сообщений, но для нас (программиста) нет ничего, кроме абстракции общей памяти.
- Модель с передачей сообщений. И это уже распределённые системы, потому что у них память у каждого своя.

Из интересного эти системы эквивалентны. Но они будут сильно отличаться по производительностью.
Возьмём чуть более конкретную модель вокруг общей памяти: у нас будут общие объекты, а не просто куски памяти. Простейший вариант общего объекта — общая переменная, в которую можно читать и писать. Это база, строительный блок для всего более сложного.
Тут опять же проблема с терминологией старых исследований. Общие переменные называются регистрами. Но нам, впрочем, регистры внутри процессора не нужны, так что путаницы не будет.

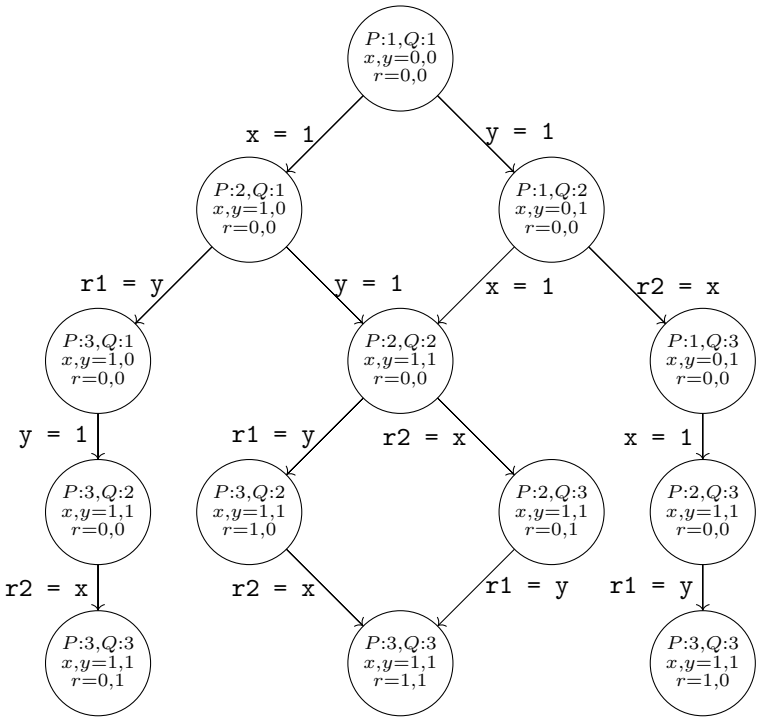
Тут мы имеем интересность. Если нам дана программа и входные данные, то результат будет всегда один и тот же. С многопоточкой не так, поэтому если мы хотим утверждать что-то про программу, то нам нужно проверить искомое свойство при любом исполнении.
Пример: пусть у нас есть два процесса (`P` и `Q`), у которых вот что:

```
shared int x = 0, y = 0;
```

```
    P:
1: x = 1
2: r1 = y
3: stop
```

```
    Q:
1: y = 1
2: r2 = x
3: stop
```

Как можно посмотреть на исполнение этой программы? Ну, у нас есть начальное состояние (где `x` и `y` равны нулю), а потом у нас может выполниться либо шаг `P`, либо шаг `Q`.



Получается, что у этой программы могут быть разные ответы. И у нас модель (модель чередования действий) очень проста, к тому же, а на практике вариантов сильно больше.
Насколько такая модель соотносится с реальностью? Ну, например, так?

```java
@JCStressTest
@State
@Outcome(id = "0, 1", expect = Expect.ACCEPTED)
@Outcome(id = "1, 1", expect = Expect.ACCEPTED)
@Outcome(id = "1, 0", expect = Expect.ACCEPTED)
public class SimpleTest1 {
    int x;
    int y;

    @Actor
    public void threadP(IntResult2 r) {
```

```
        x = 1;
        r.r1 = y;
    }

    @Actor
    public void threadQ(IntResult2 r) {
        y = 1;
        r.r2 = x;
    }
}
```

Эта байда падает, потому что появляется вариант `"0, 0"`. Как, почему?
А потому что модель плоха. Вместо неё рассмотрим TSO, где каждая запись последовательно попадает в буфер памяти (у каждого потока свой), из которой потом когда-нибудь попадает в память. Отсюда и такой вариант. И это на x86, а если на ARM посмотреть, то там всё далеко от TSO, там хуже.
Но то про исполнение, а у нас же между ним и нами есть компилятор, и он может делать любую хрень, например, переставлять инструкции.
А ещё в модели чередования есть фундаментальная проблема: она последовательная. А в жизни операции чтения и записи не мгновенные, и происходят истинно параллельно. Даже без многопоточки, потому что конвейеры и Superscalar. А ещё свет за такт процессора с частотой 3ГГц проходит 10см (и это в вакууме), то есть синхронизироваться соседние процессоры на плате не успевают.
Собственно, СТО даёт нам терминологию: a предшествует b, если свет из a успевает дойти до b. И это частичный порядок, а не полный.
Из этого есть как раз терминология в виде «произошло до» (**happens before**). Исполнение системы — пара H и $\to_H$, где $\to_H$ — частичный (строгий) порядок (транзитивное, антирефлексивное, асимметричное отношение) на H, а H — множество чтений и записей общих объектов. $e \to_H f$ читается «e произошло до f». Ещё букву под стрелкой будем опускать, когда очевидно. Каждая операция будет состоять из двух событий: $\operatorname{inv} e$ — вызов операции и $\operatorname{res} e$ — завершение e.
И тут важная деталь модели с общей памятью: события (не операции) строго упорядочены. Этот строгий порядок будет обозначаться знаком $<_H$. Эта иллюзия действительно поддерживается многоядерными системами. И тут, зная этот полный порядок, мы можем индуцировать отношение «произошло до» так: будем говорить, что e случилось до f, если $\operatorname{res} e <_H \operatorname{inv} f$.
При этом все операции будем рисовать на графике глобального времени: есть какое-то глобальное время, и на нём расположены события. И на этой модели будут наглядно видны $\to$.
Система — набор всех возможных исполнений программы.
Кроме системы у нас есть какие-то гарантии. В спецификации у нас написано, что есть. Есть операции синхронизации (например, действие над `std::atomic`) и, в более общем, модель памяти. Она описывает, что может произойти в многопоточной программе, а что — не может.
Пусть у нас произошло какое-то исполнение. Тогда мы можем описать его в терминах таких операций: `x.w(1)` — запись, и `x.r:1` — чтение и результат.
Исполнение называется **последовательным**, если все операции линейно упорядочены отношением «произошло до».
Две операции над одной переменной, хотя бы одна из которых запись, называются **конфликтующими**. Конфликтующие операции не коммутируют в модели чередования.
Если две конфликтующие операции в данном исполнении произошло параллельно, такая ситуация называется **гонкой данных** (data race). Программа называется корректно синхронизированной, если в ней при любом исполнении нет гонок.
Сужение исполнения на поток P ($H\big|_P$) — это исполнение, в котором остались только операции из P. Исполнение называется **правильным** (well-formed), если его сужение на каждый поток последовательно.
Программный порядок — это такой порядок: берём исполнение, сужаем на каждый поток, получаем порядок операций в нём, объединяем это для всех потоков.
Сужение исполнения на объект — множество всех операций, оперирующих с ним. Если сужение исполнения на объект является последовательным, то можно проверить его на соответствие **последовательной спецификации объекта** (т.е., по сути, контракту). Например, у регистра процессора

последовательная спецификация в том, чтобы чтение возвращало результат последней записи.
Если в исполнении для любого объекта (для которого можно проверить) выполнена последовательная спецификация, исполнение называется **допустимым** (legal).
Но как определить допустимость параллельного исполнения? Тут так: каждому исполнению сопоставим допустимое последовательное исполнение. А как — вопрос. Всё это — условия согласованности. Их вагон, но среди них есть основное правило: корректные последовательные программы должны считаться согласованными при любом их исполнении в одном потоке.
Нас больше всего интересуют два условия согласованности:

- Исполнение **последовательно согласовано**, если можно сопоставить эквивалентное ему допустимое последовательное исполнение, которое сохраняет программный порядок. Беда: последовательная согласованность на каждом объекте не влечёт последовательной согласованности всего исполнения. Поэтому можно давать гарантии про всю систему (вся JVM последовательно согласована), а про отдельные объекты давать такую гарантию невозможно и групо.
- Исполнение **линеаризуемо**, если можно сопоставить эквивалентное ему допустимое последовательное исполнение, которое сохраняет весь порядок «произошло до». Такое последовательное исполнение называется **линеаризация**.

Теорема 1 (Локальность линеаризуемости)**.** *Исполнение линеаризуемо тогда и только тогда, когда линеаризуемо исполнение на каждом объекте по отдельности.*

Доказательство.

Тогда. Очевидно.

Только тогда. Пусть H линеаризуемо на каждом объекте x. Возьмем объединение:

- Линерации отношения «произошло до» на каждом объекте $\to_x$.
- Исходного отношения «произошло до» $\to_H$.

Транзитивно замкнем (сохранился исходный порядок и на объектах).
Почему нет циклов? От противного.
Поскольку $\to_H$ и $\to_x$ по отдельности транзитивны, наш цикл можно порезать так, чтобы одинаковых подряд не было.
Ещё у нас не может быть $e \to_x f \to_y g$, потому что тогда f должно делать что-то и с x, и с y, а такого у нас в модели нет.
Значит в нашем цикле стрелки чередуются. Посмотрим $e \to_H f \to_x g \to_H h$. Это значит следующее:

- $\operatorname{res} e < \operatorname{inv} f$.
- $\operatorname{inv} f < \operatorname{res} g$, иначе было бы $g \to_H f$, что противоречит $f \to_x g$.
- $\operatorname{res} g < \operatorname{inv} h$.

Отсюда $e \to_H g$, а значит цикл можно срезать ещё, и резать его, пока он не кончится. А цикл длины 2 некорректен.

□

Замечание. В распределённых системах нет линеаризуемости :(

Замечание. А ещё в модели глобального времени можно рисовать точки линеаризации внутри операций. По сути это и есть линеаризуемость: внутри операции можно выделить момент, когда она возымела эффект.

Замечание. Исполнение системы, выполняющей операции над линеаризуемыми объектами, можно анализировать в модели чередования.

Замечание. Если нам дают простые линеаризуемые объекты, из них мы сможем построить более сложный линеаризуемый объект, доказать линеаризуемость и забыть о том, как мы этот объект сделали.

Определение 1. Говорят, что объект **потокобезопасен** (thread-safe), если он не линеаризуем (если не сказано иного).

Пример. В JVM все операции над `volatile`-переменными операциями синхронизации (17.4.2), которые всегда линейно-упорядочены в любом исполнении (17.4.4) и согласованы с точки зрения чтения/записи (17.4.7) (следовательно, они линеаризуемы). Над не-`volatile`-полями операции могут быть даже не последовательно согласовано.

Пример. В JVM любая корректно синхронизированная программа последовательно согласована.

Пример. В C++ `std::memory_order::seq_cst` в комбинации с `std::atomic` даёт гарантию линеаризуемости.

Ну так лол, поправим наш код:

```
@JCStressTest
@State
@Outcome(id = "0, 1", expect = Expect.ACCEPTED)
@Outcome(id = "1, 1", expect = Expect.ACCEPTED)
@Outcome(id = "1, 0", expect = Expect.ACCEPTED)
public class SimpleTest1 {
    volatile int x;
    volatile int y;

    @Actor
    public void threadP(IntResult2 r) {
        x = 1;
        r.r1 = y;
    }

    @Actor
    public void threadQ(IntResult2 r) {
        y = 1;
        r.r2 = x;
    }
}
```

А это байда уже работает. Почему? Потому что на самом деле теперь между операциями с `x` и `y` стоит инструкция `mfence`, которая сбрасывает буфер TSO.
Но проблема: `volatile` медленный, паскуда. Поэтому искуство многопоточки в том, чтобы расставить `volatile` только там, где надо, а где не надо — не расставить.