**Слайд 1.**

**Тема**: «Сравнение производительности java streamAPI в параллельных вычислениях.

**Цель:** показать практически и графически есть и когда начинают играть существенную роль выполнение параллельных вычислений при использовании метода parallel().

**Слайд 1. Введение**

Стримы сделали Java-код компактнее и читаемее.

Идеей при создании Stream API было предоставить разработчику простой способ распараллеливания задач, чтобы можно было получить выигрыш в производительности на многоядерных машинах. При этом нужно было избежать сложности, присущей многопоточному программированию. Это удалось сделать, в Stream API.

В данной презентации не будет заумной теории, только практика, только хардкор, заваривайте кофеек и погнали.

**Слайд 2. С чем будем работать ?.**

Итак сразу к делу. Давайте с помощью доступных средств посмотрим что там разбивает \ собирает наш Stream на паралельные операции, и вообще сколько доступных ядер в тестовой машине:

private static void getInfo() {  
  
 System.*out*.println("CPU: " + Runtime.*getRuntime*().availableProcessors());  
 System.*out*.println("freeMemory: " + Runtime.*getRuntime*().freeMemory() / 1048576);  
 System.*out*.println("maxMemory: " + Runtime.*getRuntime*().maxMemory() / 1048576);  
 System.*out*.println("totalMemory: " + Runtime.*getRuntime*().totalMemory() / 1048576);  
 Set<String> collect = IntStream.*range*(0, Integer.*MAX\_VALUE* / 20)  
 .parallel()  
 .mapToObj(i -> Thread.*currentThread*().getName() + "\n")  
 .collect(Collectors.*toSet*());  
 System.*out*.println(collect);  
}

Внимание на **Thread.*currentThread*().getName()**

собственно что мы тут видим , 7 ForkJoinPool (хотя ядер 8).

Вывод:

CPU: 8

freeMemory: 13

maxMemory: 247

totalMemory: 15

[ForkJoinPool.commonPool-worker-6

, ForkJoinPool.commonPool-worker-7

, ForkJoinPool.commonPool-worker-4

, ForkJoinPool.commonPool-worker-5

, ForkJoinPool.commonPool-worker-2

, ForkJoinPool.commonPool-worker-3

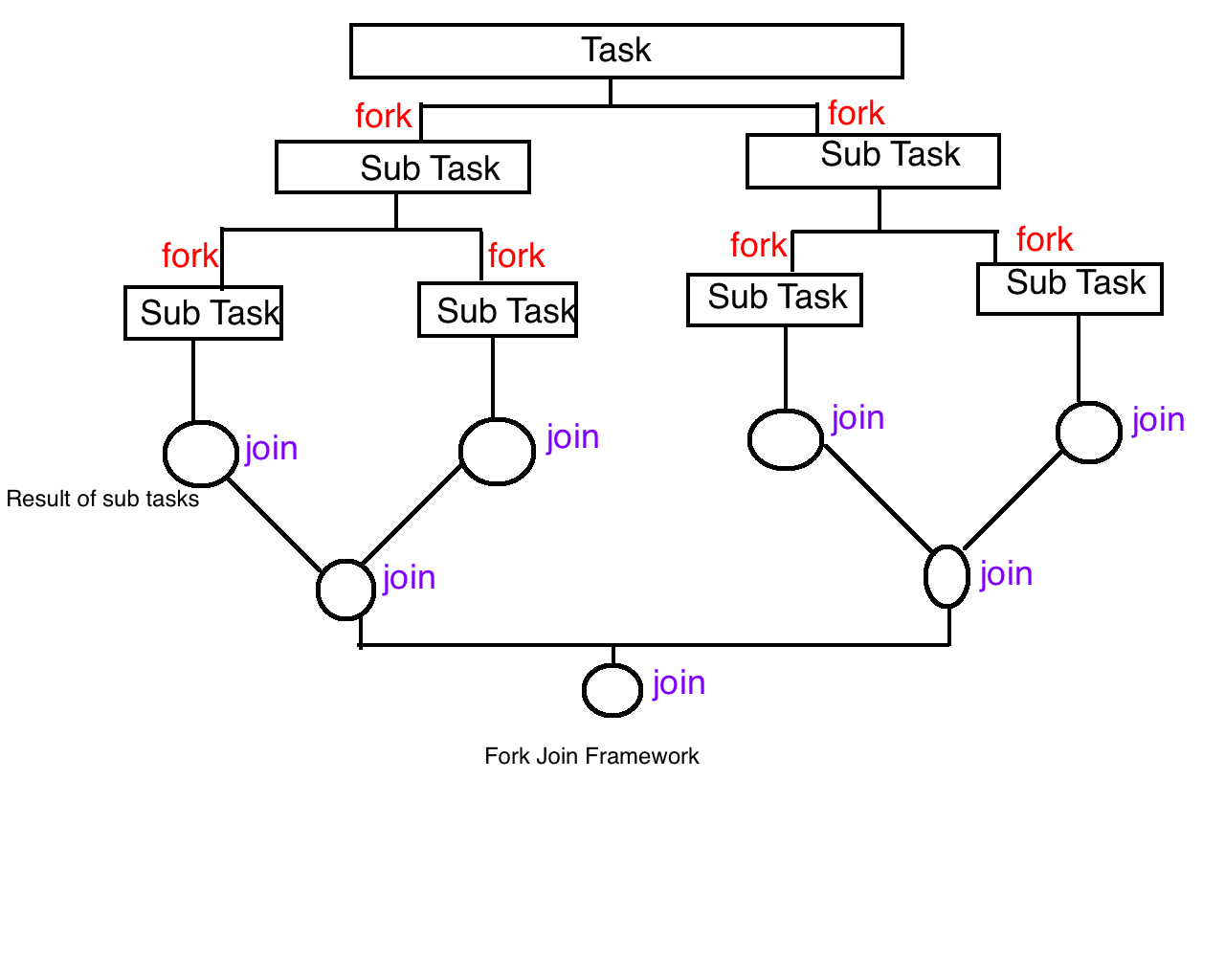
, ForkJoinPool.commonPool-worker-1

**Слайд 3. Чё за ForkJoinPool…**.

Это выглядит вот так:

**Fork join** — метод, применяемый для увеличения производительности выполнения большого количества задач. Метод заключается в том, что каждая задача разбивается на множество более мелких синхронизированных задач, которые обрабатываются параллельно на разных ядрах.

Суть метода проста: большая задача разбивается на задачи поменьше, те, в свою очередь, на ещё более мелкие задачи, и так до тех пор, пока это имеет смысл. В самом конце получившаяся тривиальная задача выполняется последовательно. Данный этап называется Fork. Результат выполнения последовательных задач объединяется вверх по цепочке, пока не получится решение самой верхней задачи. Данный этап называется Join. Выполнение всех задач происходит параллельно.



–

java.util.concurrent

## Class ForkJoinPool

* [java.lang.Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)
  + [java.util.concurrent.AbstractExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/AbstractExecutorService.html)
    - java.util.concurrent.ForkJoinPool

ForkJoinPool отличается от других видов [ExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html) главным образом тем, что использует кражу работы: все потоки в пуле пытаются найти и выполнить задачи, отправленные в пул и / или созданные другими активными задачами (в конечном итоге блокируя ожидание работы, если таковой не существует). Это обеспечивает эффективную обработку, когда большинство задач порождают другие подзадачи (как и большинство задач FORKJOINT), а также когда множество небольших задач отправляются в пул от внешних клиентов. Особенно при установке asyncMode в значение true в конструкторах, ForkJoinPools также могут подходить для использования с задачами в стиле событий, которые никогда не объединяются.

--

**Слайд 4. Когда использовать параллельные потоки.**

Платформа java.util.streams поддерживает операции с коллекциями и другими источниками. Большинство потоковых методов применяют одну и ту же операцию к каждому элементу данных. Когда доступно несколько ядер, можно применить управление «параллельным потоками данными» с помощью метода parallelStream () коллекции.

**Но когда это делать?**

Итак КОГДА?.

- когда операции **независимы** и **либо требуют больших вычислительных затрат**, либо применяются ко многим элементам эффективно разделяемых структур данных, либо к тем и другим одновременно.

Более подробно: функция (F) для каждого элемента (обычно лямбда) является независимой: вычисление для каждого элемента не зависит и не влияет на вычисление любого другого элемента. (S), исходная коллекция эффективно разделяется.

S.parallelStream ().Operation (F)

вместо

S.stream (). Operation (F),

Помимо коллекций, существует несколько других легко распараллеливаемых источников потоков, например java.util.SplittableRandom (для которого вы можете использовать метод stream.parallel () для распараллеливания).

**Слайд 4. Когда не следует применять, как оценить затраты .**

Большинство источников, основанных на вводе-выводе, предназначены в первую очередь для последовательного использования. Общее время выполнения последовательной версии превышает минимальный порог.

Оценить затраты на практике можно, умножив N (количество элементов) на Q (стоимость элемента F), в свою очередь оценив Q как количество операций или строк кода, а затем проверив, что N \* Q находится на уровне минимум 10000. (Если вы чувствуете себя трусливым, добавьте еще ноль или два.)

Итак, когда F - крошечная функция, такая как x -> x + 1, тогда для параллельного выполнения потребуется N> = 10000 элементов. И наоборот, когда F представляет собой масштабное вычисление, такое как поиск лучшего следующего хода в шахматной игре, коэффициент Q настолько высок, что N не имеет значения, если набор полностью разделяется.

**АХТУНГ!** Если вычисление не является независимым, то параллельное выполнение не имеет смысла и даже может быть опасным.

**ВЫПОЛНЕНИЕ.**

Поскольку с годами процессоры добавляли ядра, большинство из них также добавляли механизмы управления питанием, которые могут замедлить запуск этих ядер, иногда с дополнительными накладными расходами, налагаемыми JVM, ОС и т.д. Пороговое значение приблизительно соответствует времени, которое может потребоваться для того, чтобы достаточное количество ядер начало обрабатывать параллельные подзадачи, чтобы стать целесообразным. Как только они начнутся, параллельные вычисления могут быть более энергоэффективными, чем последовательные (в зависимости от различных деталей процессора и системы;

**ГРАНУЛЯРНОСТЬ**

Разделение и без того небольших вычислений редко имеет смысл. Фреймворк обычно разбивает проблемы, так что части могут обрабатываться всеми доступными ядрами в системе. Если каждому ядру практически нечего делать после запуска, то усилия (в основном последовательные) по настройке параллельных вычислений тратятся впустую. Учитывая, что в наши дни практический диапазон ядер составляет от 2 до 256, порог также не учитывает эффекты чрезмерного разделения.

**ДЕЛЕНИЕ**

Наиболее эффективно разделять коллекции включающие ArrayLists, HashMaps, а также простые массивы (т.е. массивы формы T [], разделенные с помощью статических методов java.util.Arrays). Наименее эффективными являются LinkedLists, BlockingQueues и большинство источников на основе ввода-вывода. Остальные где-то посередине. (Структуры данных обычно эффективно разделяются, если они внутренне поддерживают произвольный доступ, эффективный поиск или и то, и другое.) Если разделение данных занимает больше времени, чем их обработка, усилия тратятся зря. Итак, если добротность вычислений достаточно высока, вы можете получить параллельное ускорение даже для LinkedList, но это не очень распространено. Кроме того, некоторые источники не могут быть полностью разделены на отдельные элементы, поэтому могут быть ограничения на то, насколько точно разделяются задачи. Общие эффекты легко увидеть. Итак нафиг теорию давайте посмотрим на КОД. Наихудшее замедление происходит, когда элементов меньше примерно 100 - это активирует кучу потоков, которым в конечном итоге нечего делать, потому что вычисления завершаются еще до их запуска. С другой стороны, когда вычисления для каждого элемента отнимают много времени, выгоды неизменны.

Другими словами, использование parallel (), когда недостаточно вычислений для его оправдания, может стоить вам около 100 микросекунд, а использование его, когда это оправдано, должно сэкономить вам, по крайней мере, столько времени (возможно, часы для очень больших проблем). Точные затраты и выгоды меняются в зависимости от времени и платформы, а также в зависимости от контекста. Например, параллельное выполнение крошечных вычислений внутри последовательного цикла усиливает эффекты нарастания и спада. (Микробенчмарки, которые делают это, могут не давать прогнозов относительно фактического использования.)

**Некоторые вопросы и ответы**

Почему JVM не может самостоятельно решить, использовать ли параллельный режим?

Он мог бы попробовать, но слишком часто давал бы плохие ответы. Поиски полностью неуправляемого автоматизированного многоядерного параллелизма не были одинаково успешными в течение последних трех десятилетий, поэтому платформа stream использует более безопасный подход, просто требующий от пользователей принятия решений "да / нет". Эти решения основаны на инженерных компромиссах, которые вряд ли когда-либо полностью исчезнут, и похожи на те, которые постоянно делаются при последовательном программировании. Например, вы можете столкнуться с коэффициентом в сто накладных расходов (замедление) при нахождении максимального элемента в коллекции, содержащей только один элемент, по сравнению с простым использованием этого элемента напрямую (не внутри коллекции). Иногда JVM могут оптимизировать такие накладные расходы для вас. Но это нечасто применяется в последовательных случаях и никогда в параллельных случаях. С другой стороны, мы ожидаем, что инструменты будут развиваться, чтобы помочь пользователям принимать более эффективные решения.

Что делать, если у меня слишком мало знаний о параметрах (F, N, Q, S), чтобы принять правильное решение?

Это также похоже на распространенные проблемы последовательного программирования. Например, вызов метода сбора S.contains(x) обычно выполняется быстро, если S является хэш-набором, медленно, если список ссылок, или между ними. Обычно лучший способ справиться с этим для автора компонента, который использует коллекцию, чтобы не экспортировать ее напрямую, а вместо этого экспортировать операции на ее основе. Затем пользователи изолируются от этих решений. То же самое относится и к параллельным операциям. Например, компонент с внутренней коллекцией "цены" может определять метод с использованием порогового значения размера, если только вычисления для каждого элемента не являются дорогостоящими. Например:

public long getMaxPrice() { return priceStream().max(); }

private Stream priceStream() {

return (prices.size() < MIN\_PAR) ?

prices.stream() : prices.parallelStream();

}

Вы можете расширить эту идею всевозможными способами, чтобы учесть всевозможные соображения о том, когда и как использовать параллелизм.

**Что делать, если моя функция может выполнять ввод-вывод или синхронизацию?**

С одной стороны, это функции, которые не соответствуют критерию независимости, включая последовательный ввод-вывод, доступ к заблокированным синхронизированным ресурсам и случаи, когда сбой в одной параллельной подзадаче, выполняющей ввод-вывод, оказывает побочное воздействие на другие. Распараллеливать их не имело бы особого смысла. Другой крайностью являются вычисления, выполняющие случайный временный ввод-вывод или синхронизацию, которые редко блокируются (например, большинство форм ведения журнала и большинство видов использования параллельных коллекций, таких как ConcurrentHashMap). Они безвредны. Промежуточные дела требуют наибольшего суждения. Если каждая подзадача может быть заблокирована в течение значительного времени в ожидании ввода-вывода или доступа, ресурсы ЦП могут простаивать без возможности их использования программой или JVM. Все несчастны. В этих случаях параллельные потоки обычно не являются хорошим выбором, но доступны хорошие альтернативы, например, асинхронный ввод-вывод и завершаемые проекты.

**Что делать, если мой источник основан на IO?**

В настоящее время источники потоков на основе ввода-вывода JDK (например, BufferedReader.lines()) в основном предназначены для последовательного использования, обрабатывая элементы один за другим по мере их поступления. Существуют возможности для поддержки высокоэффективной массовой обработки буферизованного ввода-вывода, но в настоящее время они требуют специальной разработки источников потоков, разделителей и/или коллекторов. Некоторые распространенные формы могут поддерживаться в будущих выпусках JDK.

**Что делать, если моя программа запущена на загруженном компьютере и используются все ядра?**

Машины обычно имеют только фиксированный набор ядер и не могут волшебным образом создавать больше, когда вы выполняете параллельную операцию. Однако до тех пор, пока критерии выбора параллельного выполнения четко соблюдены, обычно нет никаких причин для беспокойства. Ваши параллельные задачи будут конкурировать за процессорное время с другими, поэтому вы увидите меньшее ускорение. В большинстве случаев это все еще более эффективно, чем альтернативные варианты. Базовая механика разработана таким образом, что если другие ядра недоступны, вы увидите лишь небольшое замедление по сравнению с последовательной производительностью, если только система уже не перегружена настолько, что тратит все свое время на переключение контекста, а не на выполнение какой-либо реальной работы, или настроена при условии, что вся обработка выполняется последовательно. Если вы находитесь в такой системе, администратор, возможно, уже отключил использование нескольких потоков /ядер как часть конфигурации JVM. И если вы являетесь администратором такой системы, вы могли бы подумать об этом.

**Все ли операции распараллеливаются в параллельном режиме?**

Да, по крайней мере, в какой-то степени, хотя структура потока подчиняется ограничениям источников и методов при выборе того, как это сделать. В целом, меньшее количество ограничений обеспечивает больший потенциальный параллелизм. С другой стороны, нет никакой гарантии, что фреймворк извлечет и применит все возможные возможности для параллелизма. В некоторых случаях, если у вас есть время и опыт, вы можете самостоятельно разработать значительно лучшее параллельное решение.

**Сколько параллельного ускорения я получу?**

Если вы будете следовать этим рекомендациям, обычно этого достаточно, чтобы быть стоящим. Предсказуемость не является сильной стороной современного оборудования и систем, поэтому общие ответы невозможны. Локальность кэша, скорость сбора мусора, JIT-компиляция, конкуренция в памяти, компоновка данных, политики планирования ОС и наличие гипервизоров относятся к числу факторов, которые могут оказать существенное влияние. Они также играют роль в последовательной производительности, но часто увеличиваются в параллельных настройках: проблема, вызывающая десятипроцентную разницу в последовательном выполнении, может привести к десятикратной разнице в параллельном выполнении.

Структура потока включает в себя некоторые средства, которые могут помочь вам повысить шансы на ускорение. Например, использование специализаций для примитивов, таких как IntStream, часто дает больший эффект параллельно, чем последовательно, поскольку это не только снижает накладные расходы (и занимаемую площадь), но и улучшает локальность кэша. И использование ConcurrentHashMap вместо хэш-карты в качестве цели параллельной операции "сбора" снижает внутренние накладные расходы. Дополнительные советы и рекомендации появятся по мере того, как люди приобретут опыт работы с этой структурой.