**Когда использовать параллельные потоки**

Инфраструктура java.util.streams поддерживает управляемые данными операции с коллекциями и другими источниками. Большинство потоковых методов применяют одну и ту же операцию к каждому элементу данных. Когда доступно несколько ядер, «управляемый данными» может стать «параллельным данным», используя parallelStream() метод сбора данных. Но когда ты должен это сделать?

Попробуйте использовать  S.parallelStream().operation(F) вместо того, S.stream().operation(F) когда операции являются независимыми, и либо вычислительно дорогими, либо применяются ко многим элементам эффективно разделяемых структур данных, либо к обоим. Более подробно:

* **F** , функция для каждого элемента (обычно лямбда-функция) независима: вычисления для каждого элемента не полагаются и не влияют на вычисления любого другого элемента. (См. [Сводку пакета потока](http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html) для получения дополнительных указаний по использованию невмешивающих функций без сохранения состояния.)
* **S** , исходная коллекция эффективно разделяется. Помимо коллекций есть несколько других легко распараллеливаемых потоковых источников, например, java.util.SplittableRandom (для которого вы можете использовать stream.parallel() метод для распараллеливания). Но большинство источников, основанных на IO, предназначены в основном для последовательного использования.
* Общее время выполнения *последовательной* версии превышает минимальный порог. В настоящее время порог составляет примерно (в десять раз) 100 микросекунд на большинстве платформ. Вам не нужно измерять это точно, хотя. Вы можете достаточно хорошо оценить это на практике, умножив **N** (количество элементов) на **Q** (стоимость на элемент **F** ), в свою очередь, оценив **Q** как число операций или строк кода, а затем проверив, что **N \* Q** имеет минимум 10000. (Если вы чувствуете себя трусливым, добавьте еще ноль или два.) Так что, когда **F** - крошечная функция x -> x + 1, тогда потребуется **N> = 10000.**элементы для параллельного выполнения должны быть полезны. И наоборот, когда **F** - это массивное вычисление, подобное поиску лучшего следующего хода в шахматной игре, **добротность** настолько высока, что **N** не имеет значения, пока коллекция полностью разделима.

Платформа потоков не обеспечивает (и не может) применять ни один из них. Если вычисление не является независимым, то его параллельное выполнение не будет иметь никакого смысла и даже может быть опасным. Другие критерии проистекают из трех технических проблем и компромиссов:

Запускать

Поскольку процессоры добавляли ядра на протяжении многих лет, большинство из них также добавили механизмы управления питанием, которые могут замедлять запуск этих ядер, иногда с дополнительными издержками, накладываемыми JVM, ОС и гипервизорами. Пороговое значение приблизительно соответствует времени, которое может потребоваться достаточному количеству ядер для начала обработки параллельных подзадач. Как только они начнут работать, параллельные вычисления могут быть более энергоэффективными, чем последовательные (в зависимости от различных деталей процессора и системы; см., Например, [эту статью Federova et al](http://queue.acm.org/detail.cfm?id=1658422) ).

Зернистость

Подразделение уже небольших вычислений редко стоит. Среда обычно разделяет проблемы, так что части могут обрабатываться всеми доступными ядрами в системе. Если после запуска практически ничего не нужно делать каждому ядру, то (в основном последовательное) усилие по настройке параллельных вычислений теряется. Учитывая, что практический диапазон ядер в наши дни составляет от 2 до 256, пороговое значение также остается в стороне от эффектов чрезмерного разделения.

расщепляемости

Наиболее эффективно разделяемые коллекции включают ArrayLists и {Concurrent} HashMaps, а также простые массивы (т. Е. Массивы формы T[], разделенные с использованием статических методов java.util.Arrays). Наименее эффективными являются LinkedLists, BlockingQueues и большинство источников на основе ввода-вывода. Другие где-то посередине. (Структуры данных имеют тенденцию эффективно разделяться, если они внутренне поддерживают произвольный доступ, эффективный поиск или и то и другое.) Если для разделения данных требуется больше времени, чем для их обработки, усилия будут потрачены впустую. Таким образом, если **Q-** фактор вычислений достаточно высок, 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 в том, насколько тонко разделены задачи.

Сбор подробных измерений этих эффектов может быть затруднен (хотя это возможно при некоторой осторожности с использованием таких инструментов, как [JMH](http://openjdk.java.net/projects/code-tools/jmh/) ). Но общие эффекты легко увидеть. Вы можете экспериментировать сами, чтобы почувствовать их. Например, на 32-ядерном тестовом компьютере, выполняющем крошечные функции, такие как max () или sum () в ArrayListБезубыточность очень близка к размеру 10K. Большие размеры увеличивают до 20 раз. Время выполнения для размеров менее 10K не намного меньше, чем для 10K, поэтому часто медленнее, чем последовательное. Наихудшие замедления происходят, когда имеется менее чем около 100 элементов - это активирует группу потоков, которые заканчивают тем, что им нечего делать, потому что вычисления завершаются еще до их запуска. С другой стороны, когда вычисления для каждого элемента занимают много времени, преимущества становятся незамедлительными при использовании эффективно и полностью разделяемой коллекции, такой как ArrayList.

Еще один способ сказать все это состоит в том, что если вы используете параллел (), когда недостаточно вычислений, чтобы оправдать это, то это может стоить вам около 100 микросекунд, а его использование, когда это оправдано, должно сэкономить вам как минимум это много времени (возможно, часы для очень больших проблем). Точные затраты и выгоды зависят от времени и платформы, а также от контекста. Например, параллельное выполнение крошечных вычислений внутри последовательного цикла усиливает эффекты линейного увеличения и уменьшения. (Микробенчмарки, которые делают это, не могут быть предиктивными относительно фактического использования.)

**Некоторые вопросы и ответы**

1. Почему JVM не может понять, использовать ли параллельный режим?

Можно было бы попробовать, но это дало бы плохие ответы слишком часто. Стремление к полностью неуправляемому автоматизированному многоядерному параллелизму не было одинаково успешным в течение последних трех десятилетий, поэтому потоковая структура использует более безопасный подход - просто требовать от пользователей решений «да» или «нет». Эти решения основаны на технических компромиссах, которые вряд ли когда-либо полностью исчезнут, и аналогичны тем, которые все время принимаются в последовательном программировании. Например, вы можете столкнуться с коэффициентом сотен накладных расходов (замедлением) при поиске максимального элемента в коллекции, содержащего 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 применяется нечасто в последовательных случаях и никогда в параллельных случаях. С другой стороны,

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

Это также похоже на общие проблемы последовательного программирования. Например, вызов метода Collection S.contains(x)обычно быстрый, если S является HashSet, медленный, если LinkedList, иначе 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. Пользователи тогда изолированы от этих решений. То же самое относится и к параллельным операциям. Например, компонент с внутренними «ценами» Коллекции может определять метод, использующий порог размера, если вычисления для каждого элемента не являются дорогостоящими. Например:

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

частный поток priceStream () {

возврат (values.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.

1. Что если моя функция может выполнять IO или синхронизацию?

С одной стороны, это функции, которые не соответствуют критерию независимости, в том числе внутренне последовательный ввод-вывод, доступ к заблокированным синхронизированным ресурсам и случаи, когда сбой в одной параллельной подзадаче, выполняющей ввод-вывод, имеет побочные эффекты для других. Распараллеливание их не имело бы большого смысла. Другой крайностью являются вычисления, выполняющие случайные переходные операции ввода-вывода или синхронизации, которые редко блокируют (например, большинство форм ведения журнала и большинство случаев использования параллельных коллекций, таких как ConcurrentHashMap). Это безвредно. Промежуточные дела требуют наибольшего суждения. Если каждая подзадача может быть заблокирована в течение значительного времени в ожидании ввода-вывода или доступа, ресурсы ЦП могут бездействовать без какого-либо способа для программы или JVM использовать их. Все несчастны. В этих случаях параллельные потоки обычно не являются хорошим выбором,

1. Что если мой источник основан на IO?

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

1. Что делать, если моя программа работает на занятом 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, если критерии для выбора параллельного выполнения четко соблюдены, обычно нет никаких причин для беспокойства. Ваши параллельные задачи будут конкурировать за процессорное время с другими, так что вы увидите меньшее ускорение. В большинстве случаев это все еще более эффективно, чем альтернативы. Механизм, лежащий в основе, спроектирован таким образом, что если нет других доступных ядер, вы увидите только небольшое замедление по сравнению с последовательной производительностью, если система уже не перегружена настолько, что она тратит все свое время на переключение контекста, а не выполняет какую-либо реальную работу настроенный, предполагая, что вся обработка является последовательной. 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 можете подумать об этом.

1. Все операции распараллелены в параллельном режиме?

Да, по крайней мере, в некоторой степени, хотя платформа Stream 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 структура извлечет и применит все возможные возможности для параллелизма. В некоторых случаях, если у вас есть время и опыт, вы можете создать значительно более качественное параллельное решение.

1. Сколько параллельного ускорения я получу?

Если вы следуете этим рекомендациям, как правило, достаточно, чтобы стоить. Предсказуемость не является сильной стороной современного оборудования и систем, поэтому общие ответы невозможны. Локальность кэша, скорость сбора мусора, JIT-компиляция, нехватка памяти, компоновка данных, политики планирования ОС и наличие гипервизоров являются одними из факторов, которые могут оказать существенное влияние. Они также играют роль в последовательной производительности, но часто увеличиваются при параллельных настройках: проблема, вызывающая десятипроцентную разницу в последовательном выполнении, может привести к десятикратной разнице параллельно.

Структура потока включает в себя некоторые средства, которые могут помочь вам повысить шансы на ускорение. Например, использование специализаций для примитивов, таких как IntStream, часто имеет больший эффект в параллельном, чем последовательном, потому что это не только уменьшает накладные расходы (и занимаемую площадь), но также увеличивает локальность кэша. А использование ConcurrentHashMap вместо HashMap в качестве цели параллельной операции «сбора» снижает внутренние издержки. Дальнейшие советы и рекомендации будут появляться по мере того, как люди приобретут опыт работы с этой структурой

1. Это все слишком страшно! Разве мы не должны просто разработать политику использования свойств JVM для отключения параллелизма?

Мы не хотим говорить вам, что делать. Представление программистам новых способов ошибиться может быть страшным. Ошибки в кодировании, дизайне и оценке наверняка произойдут. Но некоторые люди на протяжении десятилетий предсказывали, что включение параллелизма на уровне приложений приведет к крупным бедствиям, которые так и не произошли.