ORACLE

Fork/Join: реализация, использование, производительность

Алексей Шипилёв
Java SE Performance
aleksey.shipilev@oracle.com
@shipilev

India

3–4 May 2012

San Francisco

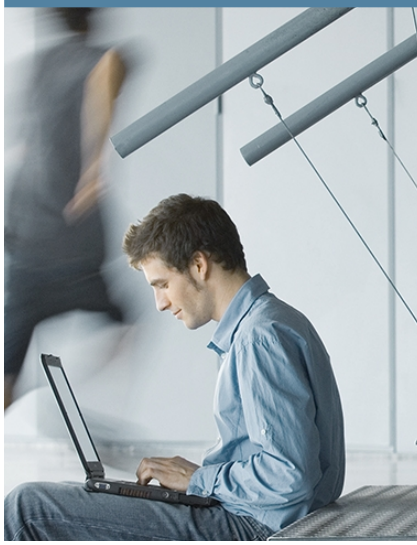
September 30–October 4, 2012

ORACLE

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle (and Doug Lea in this particular case).

Программа



- Введение
  (зачем это всё нужно)
- Fork/Join в JDK7
  (как это всё реализовано)
- Продвинутые вопросы
  (на которые докладчик знает ответ)
- Заключение и вопросы из зала
  (на которые докладчик ответа может не знать)

I.1 Философские предпосылки

- Параллельный софт – штука сложная
  - Несколько параллельных процессов
    - Происходят одновременно!
    - Как их синхронизировать?
    - Как обмениваться данными?
  - Гейзенбаги неуловимы
    - Многие ошибки проявляются в хитросплетениях времён
    - Дебаг настолько затруднён, что дешевле написать медленную, но гарантированно корректную программу
  - Библиотеки корректны и быстры
    - Предоставляют высокоуровневые абстракции с явной семантикой
    - ...и вкупе с правилами композиции позволяют собирать большие куски

I.1 Философские предпосылки

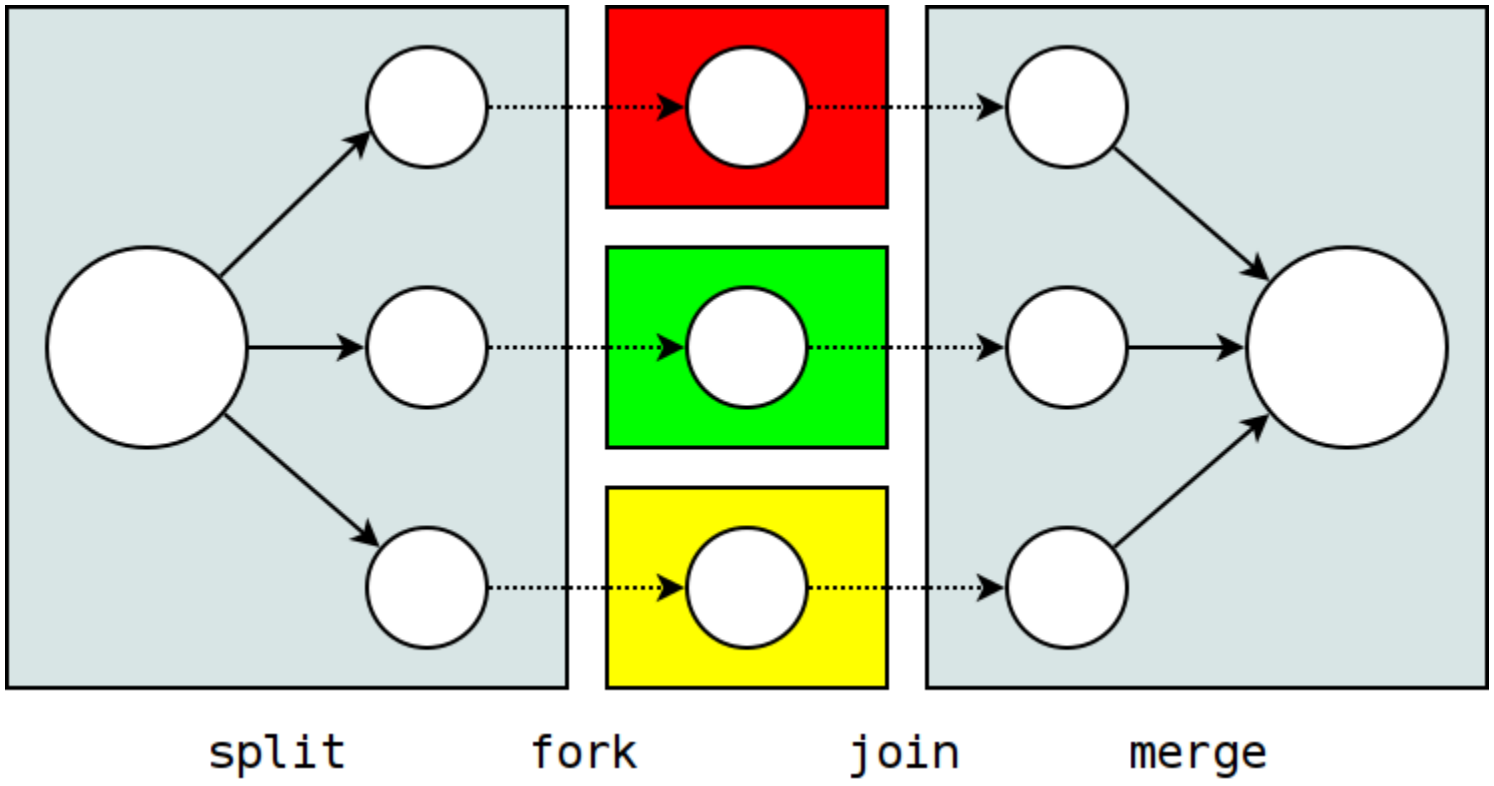
- Последовательные вычисления
  - Давным-давно придумал дядя фон-Нейман
    - Вся задача выполняется в одном потоке целиком
  - Очень успешная ментальная модель
    - Люди – разумно-однозадачные существа
  - Упёрлось в физические ограничения для одного потока
    - Пределы квантования времени (= частота проца)
    - Пределы ILP (= внутренний параллелизм задачи)
  - Натуральным расширением стало *несколько* потоков

I.1 Философские предпосылки

- Разделяй и властвуй (с)
  - “Divide and conquer”
  - Идея: разбить большую задачу на подзадачи
    - Тогда подзадачи можно выполнить одновременно
    - Многие задачи разбиваются естественным образом
  - Интуиция: хорошо, если подзадачи независимы
    - Иначе требуется коммуникация между подзадачами
    - ...а это требует нетривиального планирования, иначе deadlock'и

I.2 Да вы это уже и так знаете

- Многие задачи уже суть тривиальный F/J:



I.2 Да вы это уже и так знаете (2)

- Проблемы с тривиальным F/J:
  - Порезка задачи на куски последовательна
    - Доминирует исполнение и сводит на нет выигрыш от параллельности
    - Идея: амортизировать нарезку между задачами
  - Склейка результатов последовательна
    - Та же идея
  - Иерархические декомпозиции?
    - Не очень понятно, что делать, когда подзадача может разбиться ещё
    - Кого просить исполнить новые задачи?
    - Сколько потоков нужно для гарантий прогресса?
    - Идея: чуть позже

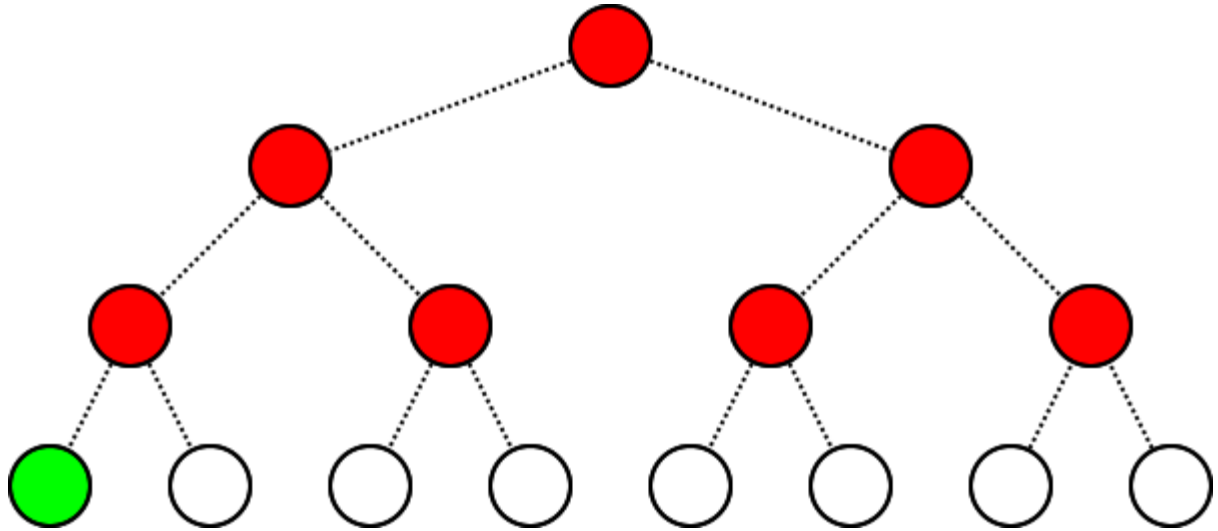
I.2 Общий вид D-n-C

- **task1** и **task2** – независимые подзадачи
  - **solveDirectly()** выполняет задачу последовательно
  - **invokeAll()** запускает подзадачи и ждёт их завершения
  - **split()** выполняется по месту, а не предварительно
  - **merge()** выполняется по месту

```
Result solve(Problem problem) {
    if (problem.smallEnough())
        return solveDirectly(solve);
    else {
        (task1, task2) = split(problem);
        (result1, result2) = invokeAll(task1, task2);
        return merge(result1, result2);
    }
}
```

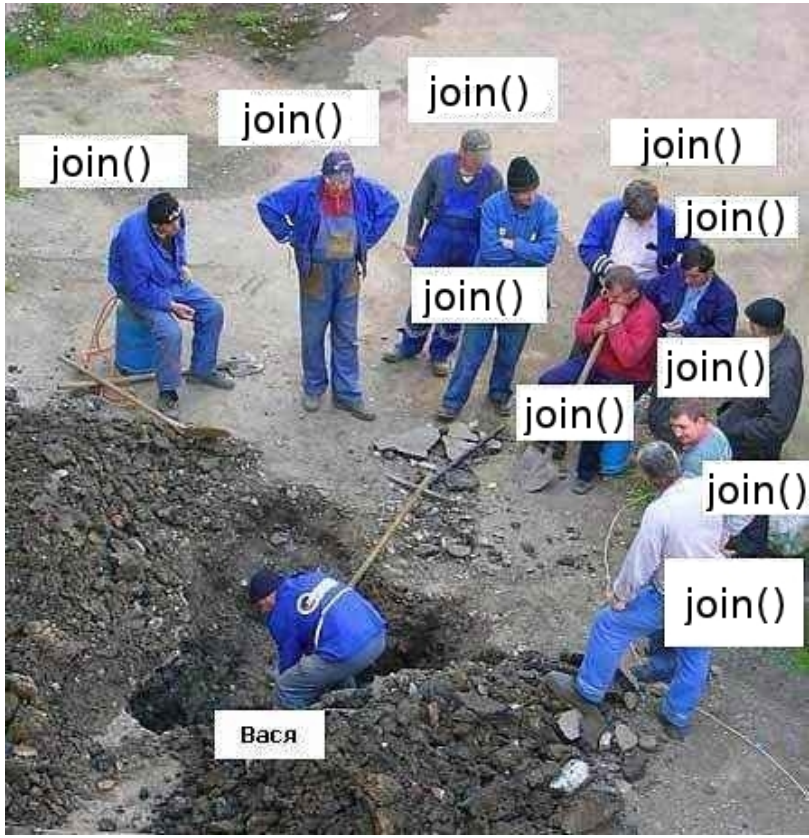

I.2 Проблемы с наивным подходом

- Задачи ждут друг друга?
  - для простоты, N-арное дерево глубины K
  - в худшем случае требуется $\frac{N^K-1}{N-1}+1$ потоков
  - в лучшем случае требуется $M(K-1)+1$ потоков

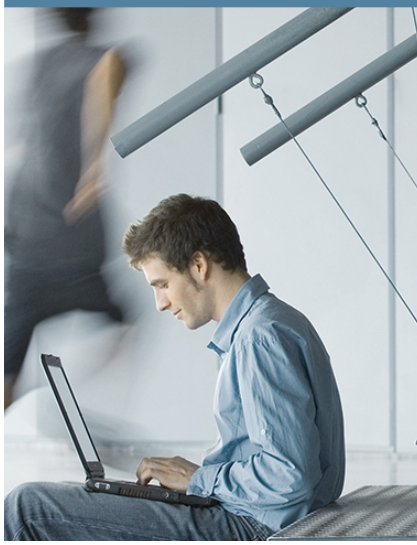


I.2 Без continuation'ов и жизнь грустна

- Блокирующее ожидание занимает поток
  - Не знаете пиковый параллелизм задачи? Ограничили количество потоков? Deadlock.
- Идея: join() не должен занимать поток!
  - Можно сделать полезную работу
  - Выполнять внешнюю задачу? Starvation.
  - Помогать другим? Требует координации потоков.



Программа



- Введение
  (зачем это всё нужно)
- Fork/Join в JDK7
  (как это всё реализовано)
- Продвинутые вопросы
  (на которые докладчик знает ответ)
- Заключение и вопросы из зала
  (на которые докладчик ответа может не знать)

II.1: JDK7 Fork/Join API

- **ForkJoinPool**
  - … implements ExecutorService
    - т.е. туда можно засылать и обычные Runnable, Callable
  - $parallelism
    - Сколько одновременно выполняющихся потоков поддерживать
    - Не жёсткий предел, FJP может создавать дополнительные потоки
  - $threadFactory
  - $uncaughtExceptionHandler
  - $asyncMode
    - Никто ещё на моей памяти не отгадал правильно, что это
    - (подробности попозже)

II.1: JDK7 Fork/Join API

- **ForkJoinTask<V>**
  - Абстракция для типичной задачи, напрямую не используют, есть два публичных подкласса:
    - **RecursiveAction**: Рекурсивное действие
    - **RecursiveTask<V>**: Рекурсивное действие, возвращающее результат
  - Пользовательские задачи должны быть подклассами **RecursiveAction** или **RecursiveTask<V>**
  - Продвинутые парни могут подкласситься от **ForkJoinTask<V>**
    - Приснится Даг Ли – не говорите, что вас не предупреждали

II.1: Типичный пример использования

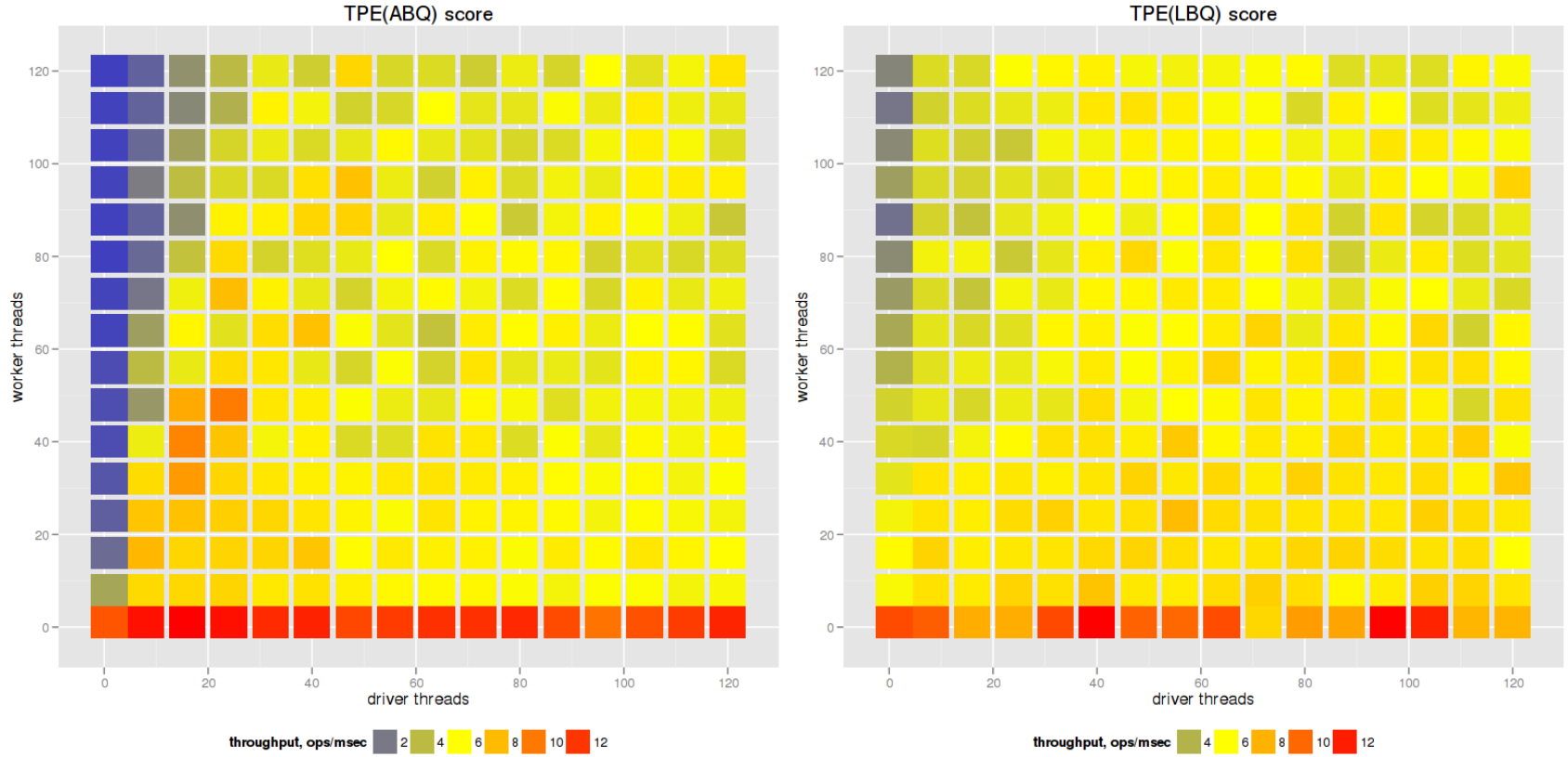
```
// Java 7
ForkJoinPool fjp = new ForkJoinPool();

// Java 8 (скорее всего, будет так)
ForkJoinPool fjp = ForkJoinUtils.getPool();

// как обычный ExecutorService
fjp.invoke(new MyRunnable());
fjp.submit(new MyRunnable());

// как обычный ExecutorService
fjp.invoke(new MyCallable());
fjp.submit(new MyCallable());

// специфичные для FJP
fjp.invoke(new MyRecursiveAction());
fjp.invoke(new MyRecursiveTask());

class MyRunnable implements Runnable { ... };
class MyCallable<T> implements Callable<T> { ... };
class MyRecursiveAction extends RecursiveAction { ... };
class MyRecursiveTask<T> extends RecursiveTask<T> { ... };
```


II.2: Балансировка задач

- Хорошо решается только в динамике
  - Задачи могут сильно отличаться по размеру
  - Процессоры могут быть неравномерно заняты, etc.
- Три базовых подхода:
  - Арбитраж задач
    - Общая очередь задач
    - Общая очередь задач с маленькими thread-локальными буферами
  - Work dealing
    - У каждого потока своя очередь
    - Перегруженные потоки отдают свои задачи на сторону
  - Work stealing
    - У каждого потока своя очередь
    - Свободные потоки крадут задачи у перегруженных

II.2: Балансировка задач

- Практика и интуиция:
  - Общая очередь – это плохо



4x10x2 Intel Xeon (Nehalem), RHEL 5.5
jsr166.jar @ 2012-04-02

II.2: Балансировка задач

- Подход FJP – work stealing:
  - Локальные очереди для каждого потока
  - С головой очереди может работать только владелец
    - Это автоматически превращает очередь в стек (LIFO)
    - Даже без синхронизации
  - Из хвоста могут тырить* задачи другие потоки
    - Если известно, что очередь не пуста, то можно не синхронизироваться
  - Из хвоста же может брать и владелец
    - $asyncMode = true
    - Тогда очередь становится действительно очередью (FIFO)

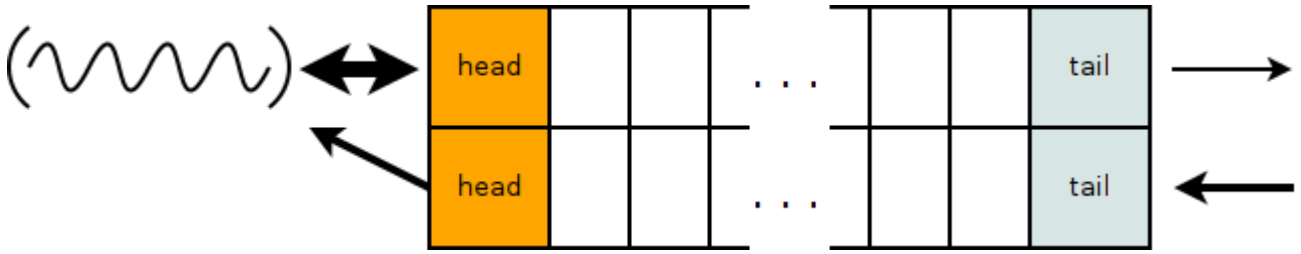
* В недрах тундры выдры в гетрах тырят в вёдра ядра кедров

II.2: Балансировка задач

- Куда происходит submit() внешних задач?
  - Загвоздка:
    - В голову очереди потока нельзя: требуется синхронизация
    - В хвост тоже особенно нельзя: порушим FIFO/LIFO
  - Решение: отдельная очередь для внешних задач

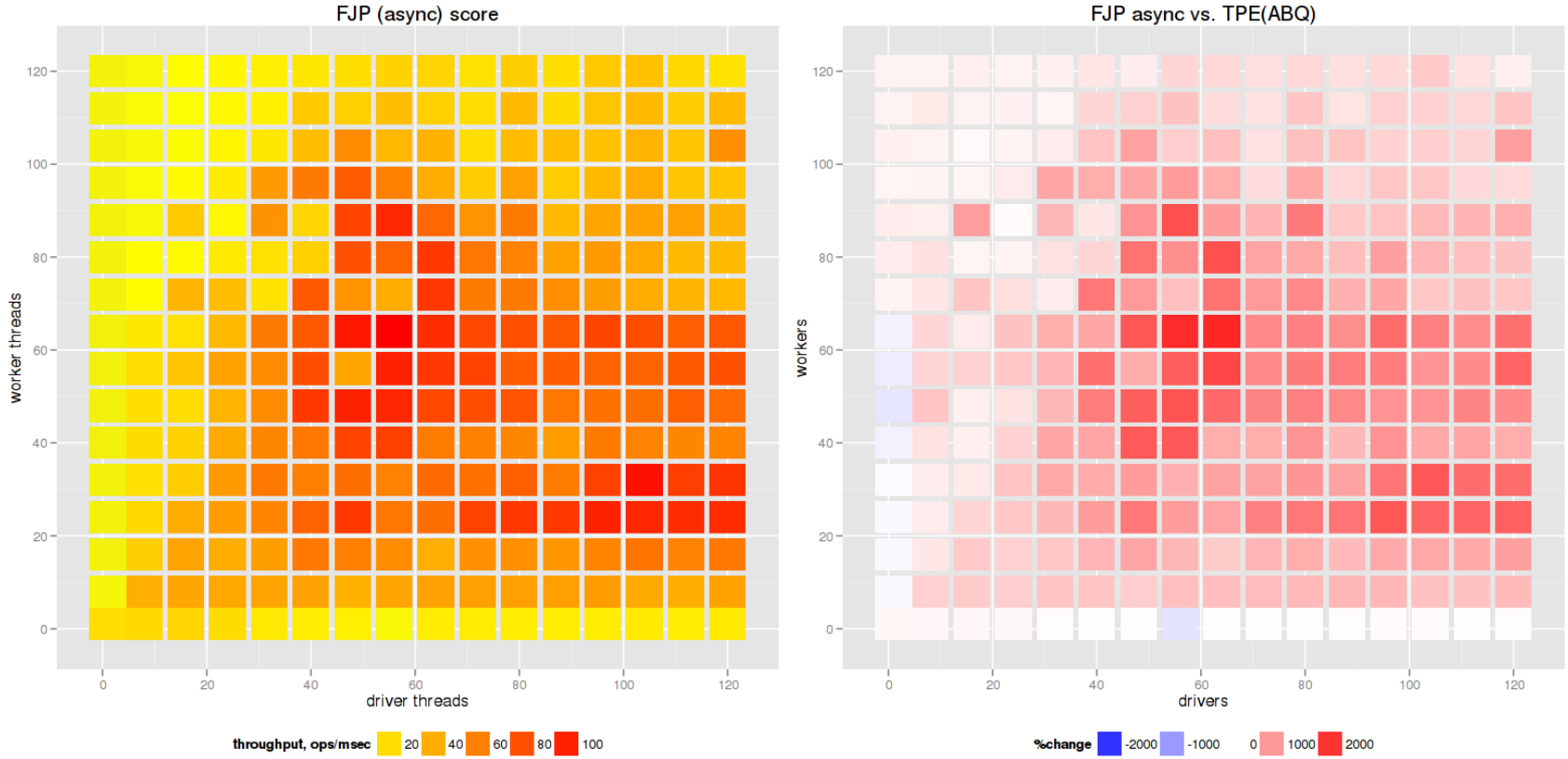
II.2: Балансировка задач

- Куда происходит submit() внешних задач?
  - Загвоздка:
    - В голову очереди потока нельзя: требуется синхронизация
    - В хвост тоже особенно нельзя: порушим FIFO/LIFO
  - Решение: отдельная очередь для внешних задач
    - Wait. Шерлок, мне кажется, или это ещё одна глобальная очередь?
  - Решение №2: давайте расклеим очередь!
    - Каждому потоку по submission queue!
    - Клиенты случайным образом мультиплексируются на эти очереди



II.2: Балансировка задач

- Конец немного предсказуем:
  - На пустых задачах FJP рвёт обычные TPE в клочья



4x10x2 Intel Xeon (Nehalem), RHEL 5.5
jsr166.jar @ 2012-04-02

II.3: Типичная задача

```
private static class StandardTask extends RecursiveTask<Long> {
    private final Problem problem;
    private final int l;
    private final int r;

    public StandardTask(Problem p, int l, int r) {
        this.problem = p;
        this.l = l;
        this.r = r;
    }

    @Override
    protected Long compute() {
        if (r - l <= THRESHOLD) {
            return problem.solve(l, r);
        }

        int mid = (l + r) >>> 1;
        ForkJoinTask<Long> t1 = new StandardTask(problem, l, mid);
        ForkJoinTask<Long> t2 = new StandardTask(problem, mid, r);

        t1.fork(); // форкнул раз
        t2.fork(); // форкнул два

        long res = 0;
        res += t2.join(); // джойнул раз
        res += t1.join(); // джойнул два
        return res; // закончили упражнение
    }
}
```


II.3: Типичная задача

```
int mid = (l + r) >>> 1;
ForkJoinTask<Long> t1 = new Task(problem, l, mid);
ForkJoinTask<Long> t2 = new Task(problem, mid, r);

t1.fork(); // форкнул раз
t2.fork(); // форкнул два

long res = 0;
res += t2.join(); // джойнул раз
res += t1.join(); // джойнул два
return res; // закончили упражнение
```


II.3: Базовая семантика методов

- **fork():**
  - Кладёт задачу в очередь, и возвращается
  - Кто-нибудь другой может эту задачу подхватить
- **join():**
  - Блокируется, пока задача не закончится
  - Но поток терять на этом нельзя!
  - FJP может дать ему что-нибудь повыполнять

II.3: Базовая семантика методов

- **fork():**
  - Кладёт задачу в очередь, и возвращается
  - Кто-нибудь другой может эту задачу подхватить
- **join():**
  - Блокируется, пока задача не закончится
  - Но поток терять на этом нельзя!
  - FJP может дать ему что-нибудь повыполнять
- fork().join()?

```
ForkJoinTask<Long> t1, t2;
t1.fork().join(); // я увидел на J1, что fork+join -- это хорошо.
t2.fork().join(); // но почему-то оно работает очень медленно :(
```


II.4: Что можно сделать на join()-е?

- Если мы что-то join()-им, значит, мы это недавно fork()-нули
  - Значит, это что-то где-то в пуле
- Если задача на верхушке моей очереди, **выполним**
  - А чего ждать у моря погоды?
- Если задача где-то в моей очереди, **выполним**
  - Сначала нужно пробежимся по очереди, найти её, и удалить
  - В async mode всегда будем бежать до конца
- Если задача не в моей очереди, **выполним** или **поможем**
  - Значит, какой-то гасконская каналья у нас её стырила
  - Если получается отобрать у стырившего задачу назад, то выполним
  - Если нет, то помогаем стырившему
- Если выполнять нечего, **блокируемся**
  - Даже если есть внешние задачи (sic!)
  - ...но поскольку тред потерялся, нужно скомпенсировать его отсутствие

II.4: Helping

- Один за всех и все за одного
  - Мы знаем, кто одолжил у нас задачу
  - Если из той задачи родятся новые, они нам нужны
    - Мы бы их и сами выполнили, если бы не появился этот д'Артаньян
    - Имеет смысл помочь выполнить новые задачи из очереди д'Артаньяна
  - Если у него тоже стырили задачу, помогаем транзитивно
    - Потому что зависимые задачи могут стырить другие... кхм... мушкетёры
  - Охотимся только за зависимыми задачами
    - Иначе рискуем starvation'ом внутренних задач под напором внешних
    - Вся группировка потоков сфокусирована на исполнении текущих

II.5: Выбор способа join()-а

- Из анализа внутренностей join()'а теоретически следует оптимальный способ
- Кроме того, есть другие способы:

II.5: Выбор способа join()-а

- Из анализа внутренностей join()'а теоретически следует оптимальный способ
- Кроме того, есть другие способы:

```
// naïve:
ForkJoinTask<Long> t1, t2;
t1.fork();
t2.fork();
long res = 0;
res += t1.join();
res += t2.join();
return res;
```


II.5: Выбор способа join()-а

- Из анализа внутренностей join()'а теоретически следует оптимальный способ
- Кроме того, есть другие способы:

```
// naïve:
ForkJoinTask<Long> t1, t2;
t1.fork();
t2.fork();
long res = 0;
res += t1.join();
res += t2.join();
return res;
```

```
// smarter:
ForkJoinTask<Long> t1, t2;
t1.fork();
t2.fork();
long res = 0;
res += t2.join();
res += t1.join();
return res;
```


II.5: Выбор способа join()-а

- Из анализа внутренностей join()'а теоретически следует оптимальный способ
- Кроме того, есть другие способы:

```
// naïve:
ForkJoinTask<Long> t1, t2;
t1.fork();
t2.fork();
long res = 0;
res += t1.join();
res += t2.join();
return res;
```

```
// smarter:
ForkJoinTask<Long> t1, t2;
t1.fork();
t2.fork();
long res = 0;
res += t2.join();
res += t1.join();
return res;
```

```
// standard:
ForkJoinTask<Long> t1, t2;
ForkJoinTask.invokeAll(t1, t2);
long res = 0;
res += t1.join();
res += t2.join();
return res;
```


II.5: Выбор способа join()-а

- Из анализа внутренностей join()'а теоретически следует оптимальный способ
- Кроме того, есть другие способы:

```
// naïve:
ForkJoinTask<Long> t1, t2;
t1.fork();
t2.fork();
long res = 0;
res += t1.join();
res += t2.join();
return res;
```

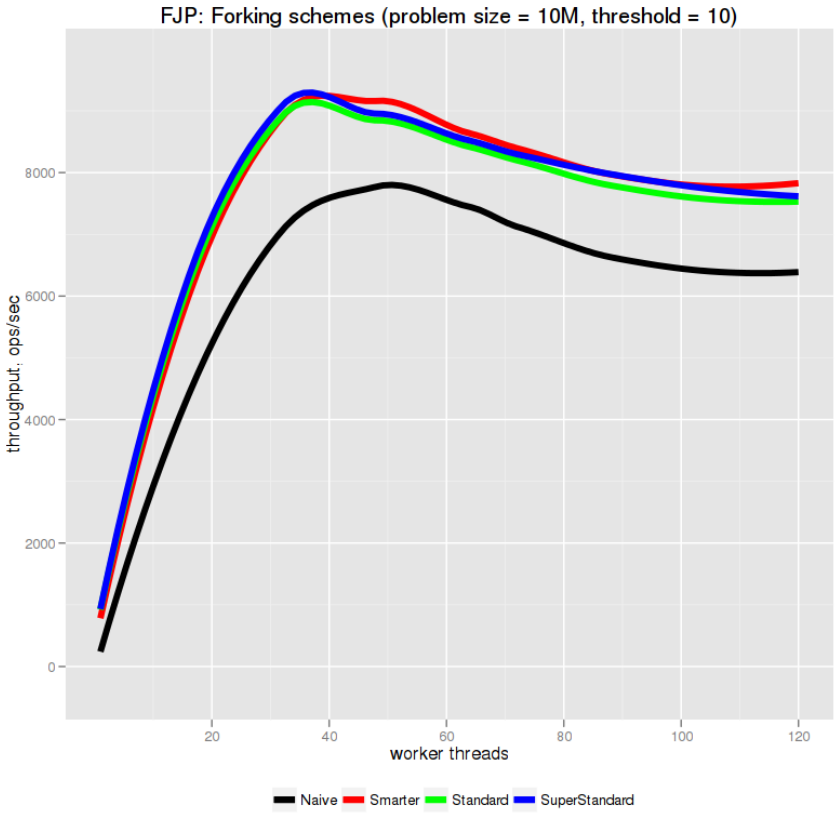
```
// smarter:
ForkJoinTask<Long> t1, t2;
t1.fork();
t2.fork();
long res = 0;
res += t2.join();
res += t1.join();
return res;
```

```
// standard:
ForkJoinTask<Long> t1, t2;
ForkJoinTask.invokeAll(t1, t2);
long res = 0;
res += t1.join();
res += t2.join();
return res;
```

```
// super-standard (inlined):
ForkJoinTask<Long> t1, t2;
t2.fork();
long res = 0;
res += t1.invoke();
res += t2.join();
return res;
```


II.5: Выбор способа join()'a

- Эксперимент:
  - **Naïve**: nuff said
  - **Smarter**: сказывается переход join() → invoke()
  - **Standard**: ок
  - **SuperStandard**: сэкономили десяток инструкций
- Вывод:
  - Пользуйтесь, чем хотите, только джойнитесь в обратном порядке



4x10x2 Intel Xeon (Nehalem), RHEL 5.5
jsr166.jar @ 2012-04-02

ORACLE

II.6: Выбор threshold'a

- Слишком маленький?
  - оверхеды на инфраструктуру
- Слишком большой?
  - мало параллелизма
- Наивная интуиция:

$$T = N/CPU\#$$

  - **N** – размер задачи
  - **CPU#** – доступный параллелизм
  - В теории работает хорошо, на практике плохо
    - Задачи занимают разное время, балансировка страдает

II.6: Выбор threshold'a

- Продвинутая интуиция:

$$T = N/(L*CPU\#)$$

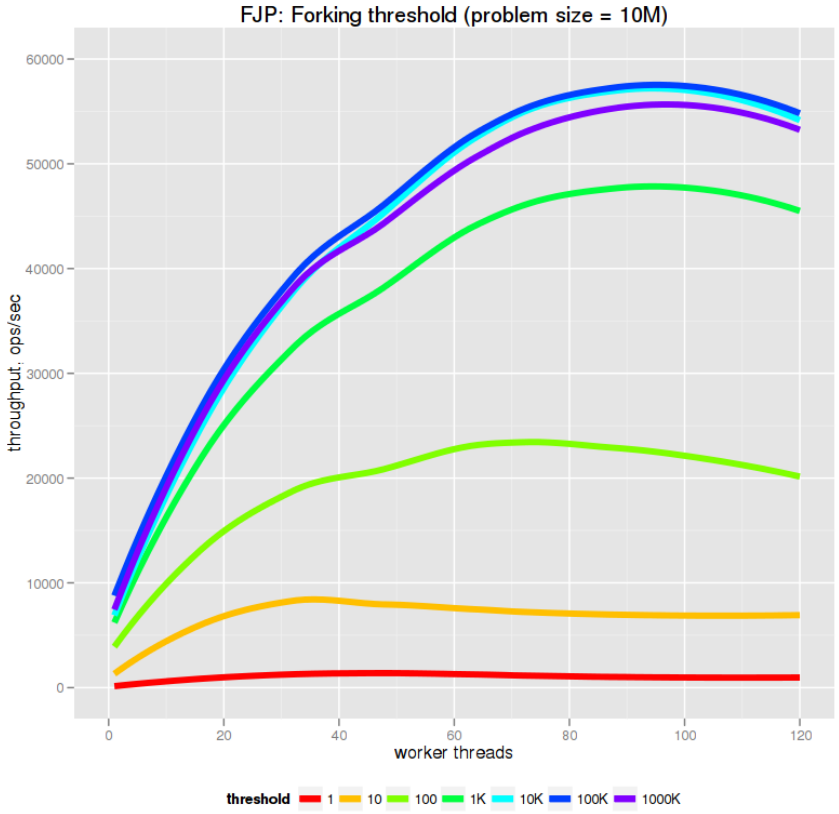
  - **N** – размер задачи
  - **CPU#** – доступный параллелизм
  - **L** – load factor (~= количество задач на поток)
- Load Factor менее влияет на скорость
  - Отвязан от размеров задачи и количества процессоров
  - Труднее ошибиться с выбором
  - Обычно принимают L = 8..16

II.6*: Лирическое отступление про бенчмарк

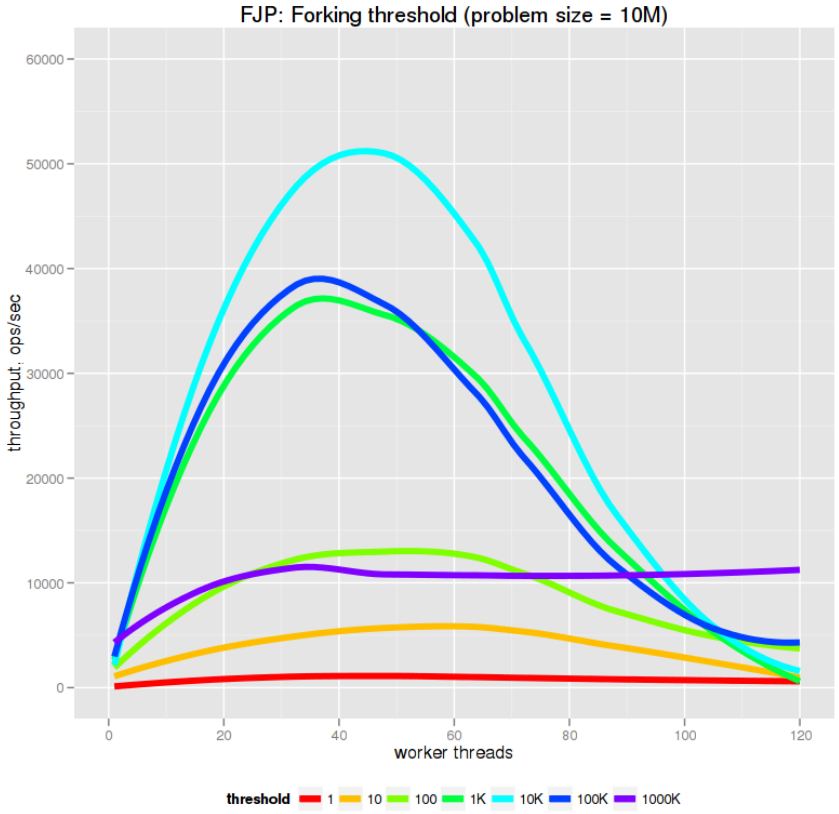
- “Концевые эффекты”
  - Бич всех бенчмарок, но особенно thread pool'ов
  - Сложно контролировать накладные расходы
    - Активация потоков (когда пул разгоняется)
    - Ожидание джойнов последних задач
    - Деактивация потоков (когда пул останавливается)
  - В ненагруженных пулах КЭ по полной программе
    - Но зато имеется возможность измерить one-shot performance
  - В нагруженных пулах КЭ нивелируются
    - Но зато появляется лишний параллелизм от перекрывающихся задач
    - “steady state” performance
  - Для того, чтобы хоть что-то понять, нужно измерять оба режима

II.6: Выбор threshold'a

Saturated:



Unsaturated:

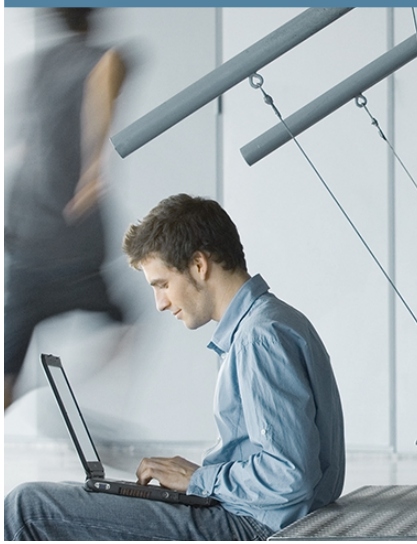


4x10x2 Intel Xeon (Nehalem), RHEL 5.5
jsr166.jar @ 2012-04-02

T = 10M / 16*80 = ~78K

ORACLE

Программа



- Введение
  (зачем это всё нужно)
- Fork/Join в JDK7
  (как это всё реализовано)
- Продвинутые вопросы
  (на которые докладчик знает ответ)
- Заключение и вопросы из зала
  (на которые докладчик ответа может не знать)

III.1: Автоматический выбор threshold'a

- Инспектирование своих очередей
  - **ForkJoinTask.getQueuedTaskCount()**
    - Количество задач в моей локальной очереди
  - **ForkJoinTask.getSurplusQueuedTaskCount()**
    - Количество задач, которые останутся в моей очереди после балансировки (= когда проснутся все спящие потоки)
- Есть задачи? Не надо пилить дальше!
  - Хорошо работает в нагруженных пулах
  - Предполагаем, что балансировка уже состоялась
  - Если у меня в очереди **X** задач, то у всех тоже по **X**

III.1: Автоматический выбор threshold'a

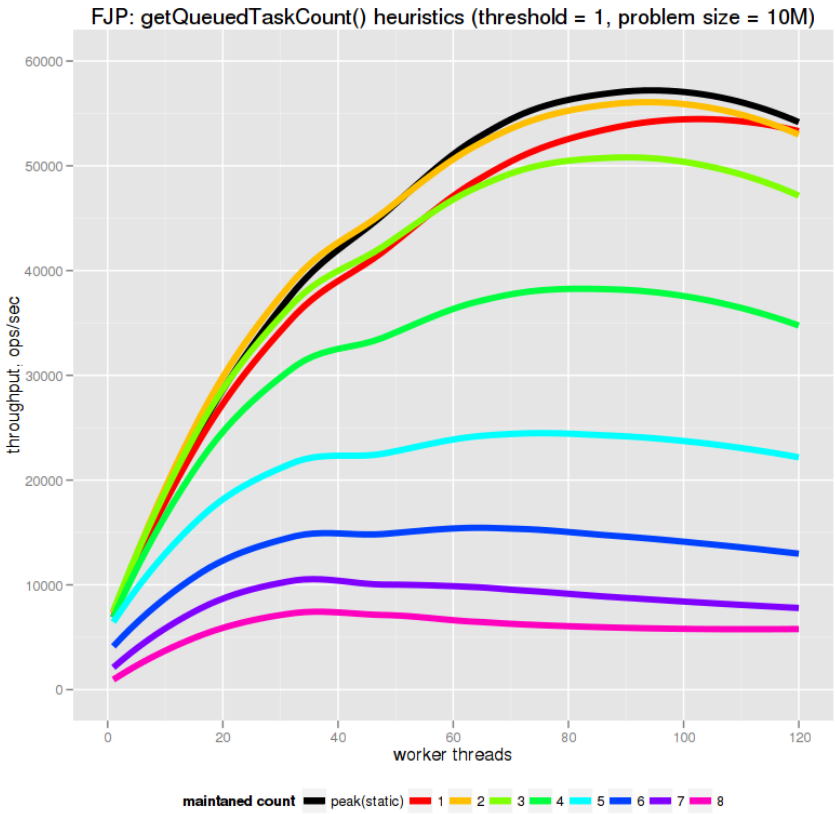
- Страхуемся от неправильного threshold'a

```
protected T compute() {
  if (r - l <= T1 || getQTC() >= T2)
    return problem.solve(l, r);

  // decompose
}
```

- Обычные значения:

  T2 = 1..3

- В нагруженных пулах близко к пиковой скорости



4x10x2 Intel Xeon (Nehalem), RHEL 5.5
jsr166.jar @ 2012-04-02

ORACLE

III.1: Автоматический выбор threshold'a

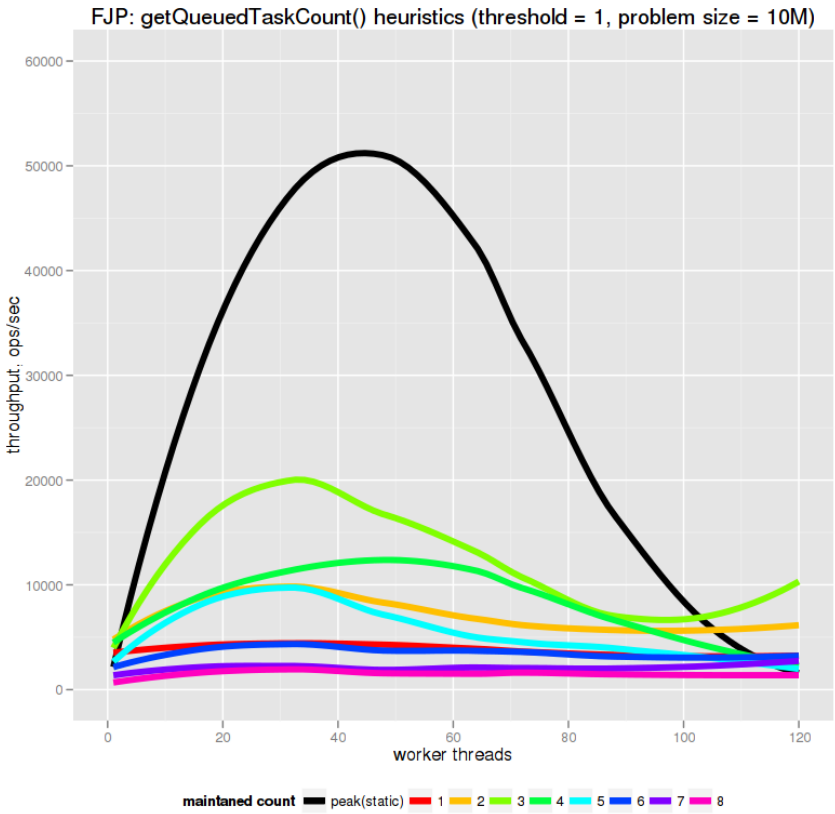
- Страхуемся от неправильного threshold'a

```
protected T compute() {
  if (r - l <= T1 || getQTC() >= T2)
    return problem.solve(l, r);

  // decompose
}
```

- Обычные значения:

  T2 = 1..3

- В ненагруженных пулах работает приемлемо
  - Но статический threshold всё равно быстрее



4x10x2 Intel Xeon (Nehalem), RHEL 5.5
jsr166.jar @ 2012-04-02

ORACLE

III.1: Автоматический выбор threshold'a

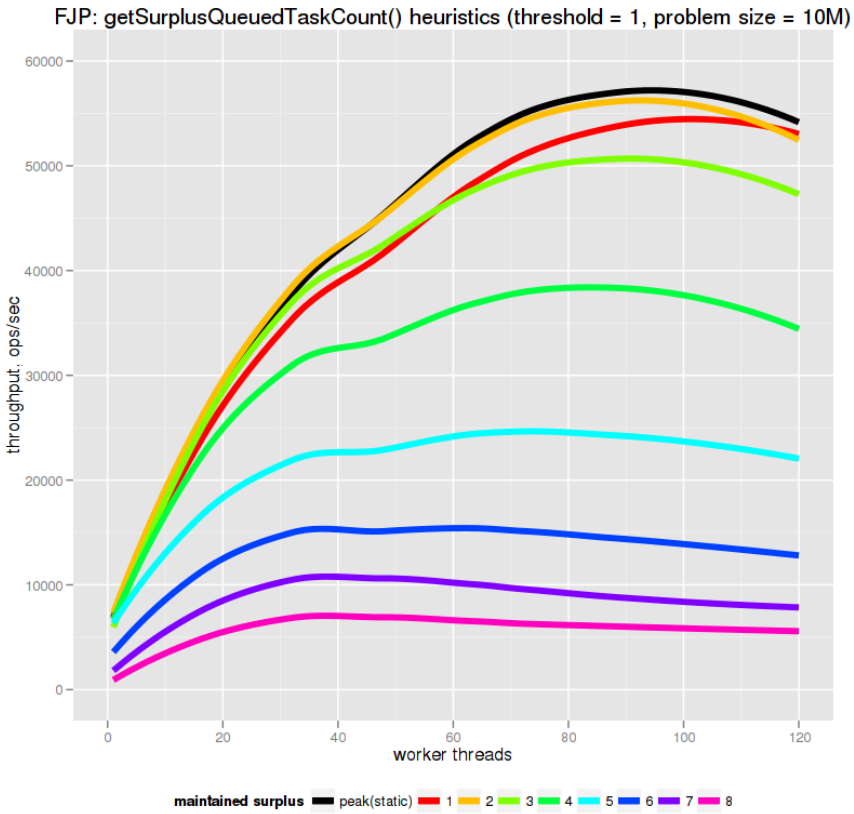
- Страхуемся от неправильного threshold'a

```
protected T compute() {
  if (r - l <= T1 || getSQTC() >= T2)
    return problem.solve(l, r);

  // decompose
}
```

- Обычные значения:

  T2 = 1..3

- В нагруженных пулах работает близко к пиковой скорости



4x10x2 Intel Xeon (Nehalem), RHEL 5.5
jsr166.jar @ 2012-04-02

ORACLE

III.1: Автоматический выбор threshold'a

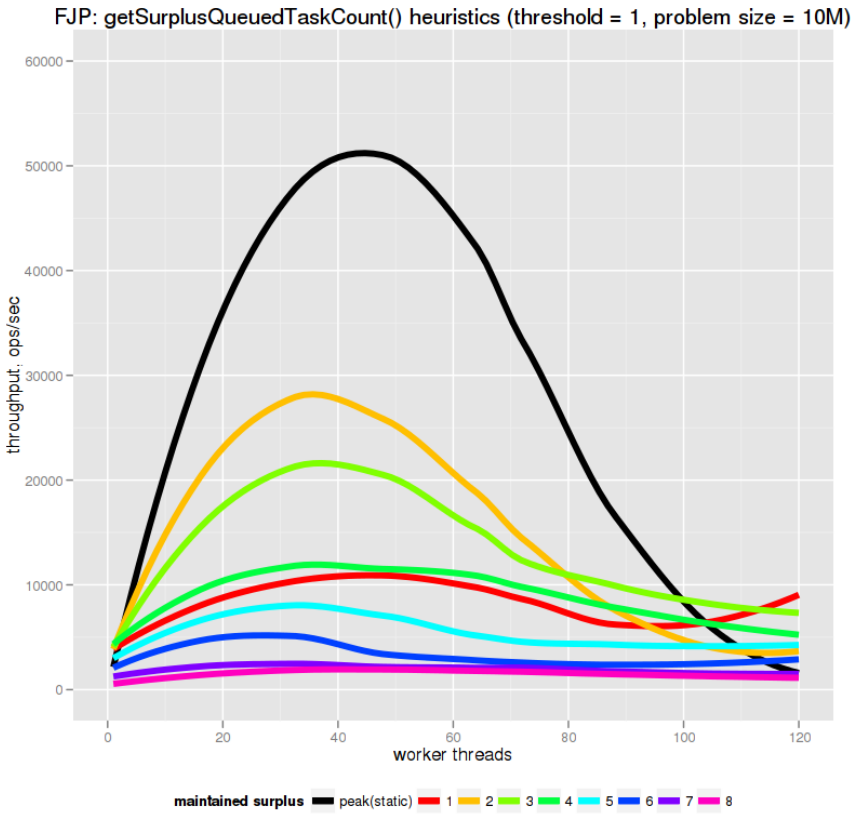
- Страхуемся от неправильного threshold'a

```
protected T compute() {
  if (r - l <= T1 || getSQTC() >= T2)
    return problem.solve(l, r);

  // decompose
}
```

- Обычные значения:

  T2 = 1..3

- В ненагруженных пулах работает лучше, чем getQTC()
  - Учитывает простаивающие потоки



4x10x2 Intel Xeon (Nehalem), RHEL 5.5
jsr166.jar @ 2012-04-02

ORACLE

III.2: Managed Blockers

- Теряем потоки не только на join()
  - Можно потерять поток на более произаических вещах
    - Sleep
    - I/O
    - Locks
  - Нужно оповестить FJP о блокировке потока
    - Тогда FJP будет иметь возможность скомпенсировать потерю
    - В этом месте появляются Managed Blockers

III.2: Managed Blockers

- **interface ForkJoinPool.ManagedBlocker**
  - **boolean isReleasable()**
    - Оптимистичная попытка заблокироваться
    - “true”, если получилось; “false”, если не получилось
  - **boolean block()**
    - Пессимистичная блокировка, задерживает поток
    - “true”, если хватит; “false”, если нужно будет ещё раз заблокироваться
- **ForkJoinPool.managedBlock(MB blocker)**
  - Инструктирует FJP попытать счастья
    - Сразу выйдет, если получилось без блокировок (**isReleasable()** == true)
    - Заблокируется и компенсирует поток, если блокировка-таки случится

III.2: Managed Blockers

- Пример:

```
class ForkJoinSleeper implements ForkJoinPool.ManagedBlocker {
    private final long timeout;
    private final TimeUnit timeUnit;
    private boolean slept;

    public ForkJoinSleeper(long timeout, TimeUnit timeUnit) {
        this.timeout = timeout;
        this.timeUnit = timeUnit;
    }

    public boolean isReleasable() {
        return slept;
    }

    public boolean block() throws InterruptedException {
        if (!slept) {
            timeUnit.sleep(timeout);
            slept = true;
        }
        return true;
    }
}
```


III.2: Managed Blockers

- Мелкая засада!
  - **FJP.managedBlock(...)** эквивалентен:

```
while (!blocker.isReleasable()) { ... }
```

  - А вот такой **ManagedBlocker** можно написать:

```
public boolean isReleasable() {
   return (result != null);
}

public boolean block() throws InterruptedException {
   try {
       result = future.get();
   } catch (ExecutionException e) { … }
   return true;
}
```


III.2: Managed Blockers

- Мелкая засада**!**
  - **FJP.managedBlock(...)** эквивалентен:

```
while (!blocker.isReleasable()) { ... }
```

  - А вот такой **ManagedBlocker** можно написать:

```
public boolean isReleasable() {
   return (result != null);
}

public boolean block() throws InterruptedException {
   try {
       result = future.get();
   } catch (ExecutionException e) { … }
   return true;
}
```

  - future.cancel()? Surprise!
    - Бесконечный цикл к вашим услугам
    - Даже если FJP.shutdown() пройдёт “успешно”

III.2: Managed Blockers

- Pro:
  - Позволяют выжать больше производительности
    - Поддерживать $parallelism, несмотря на заблокированные потоки
  - Позволяют гарантировать прогресс
    - Особенно, когда группы потоков ждут чего-то одновременно

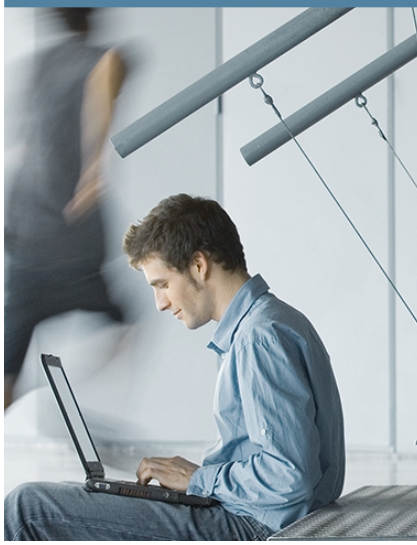
III.2: Managed Blockers

- Pro:
  - Позволяют выжать больше производительности
    - Поддерживать $parallelism, несмотря на заблокированные потоки
  - Позволяют гарантировать прогресс
    - Особенно, когда группы потоков ждут чего-то одновременно
- Contra:
  - В наивных схемах приводят к неограниченному росту количества потоков
    - Что, например, может повалить всю JVM целиком
    - (Более мягкие схемы компенсации разрабатываются)

III.3: Несколько FJP

- FJP активно разгоняется до $parallelism потоков
  - И не очень-то охотно сдувается обратно
- а) Используем один FJP на подсистему
  - Внутренние балансировки всех задач
  - Внутренняя компенсация
- б) Несколько FJP с ограниченным $parallelism
  - Сужает область для балансирования задач между пулами
  - FJP не компенсируют ожидания друг на друге
    - Сабмиттят друг в друга? Привет, deadlock'и
    - Обернуть ManagedBlockers? Привет, толпа потоков

Программа



- Введение
  (зачем это всё нужно)
- Fork/Join в JDK7
  (как это всё реализовано)
- Продвинутые вопросы
  (на которые докладчик знает ответ)
- Заключение и вопросы из зала
  (на которые докладчик ответа может не знать)

IV.1: Заключение

- ForkJoinPool – клёвая штука
  - Отличная производительность
  - Понятный расширяемый API и разумные умолчания
- Кое-где требуется ручной тюнинг
  - Настройки fork threshold'a
  - Блокировки на долгоиграющих задачах
- Активно разрабатывается и допиливается
  - Нашли баг или перформансную проблему?
  - Пишите на concurrency-interest@cs.oswego.edu

IV.2: Q/A

- Puzzler для скучающих (“Что напечатается?”):

```
private final AtomicInteger counter = new AtomicInteger();

private void run() throws ExecutionException, InterruptedException {
    ExecutorService tpe = new ThreadPoolExecutor(...);
    ForkJoinPool fjp = new ForkJoinPool(...);
    Callable callable = new Callable() ... { return counter.getAndIncrement(); }
    RecursiveTask task = new RecursiveTask() ... { return counter.getAndIncrement(); }

    counter.set(0);
    for (int c = 0; c < 10; c++) { print(tpe.submit(callable).get()); }
    println(", counter = " + counter.get());

    counter.set(0);
    for (int c = 0; c < 10; c++) { print(fjp.submit(callable).get()); }
    println(", counter = " + counter.get());

    counter.set(0);
    for (int c = 0; c < 10; c++) { print(fjp.submit(task).get()); }
    println(", counter = " + counter.get());
}
```

Вариант A:
0123456789, counter = 10
9876543210, counter = 10
9876543210, counter = 10

Вариант B:
0123456789, counter = 10
0123456789, counter = 10
9876543210, counter = 10

Вариант C:
0123456789, counter = 10
0123456789, counter = 10
0123456789, counter = 10

Вариант D:
Что-то другое.

IV.2: Q/A

- Puzzler для скучающих (“Что напечатается?”):

```
private final AtomicInteger counter = new AtomicInteger();

private void run() throws ExecutionException, InterruptedException {
    ExecutorService tpe = new ThreadPoolExecutor(...);
    ForkJoinPool fjp = new ForkJoinPool(...);
    Callable callable = new Callable() ... { return counter.getAndIncrement(); }
    RecursiveTask task = new RecursiveTask() ... { return counter.getAndIncrement(); }

    counter.set(0);
    for (int c = 0; c < 10; c++) { print(tpe.submit(callable).get()); }
    println(", counter = " + counter.get());

    counter.set(0);
    for (int c = 0; c < 10; c++) { print(fjp.submit(callable).get()); }
    println(", counter = " + counter.get());

    counter.set(0);
    for (int c = 0; c < 10; c++) { print(fjp.submit(task).get()); }
    println(", counter = " + counter.get());
}
```

Вариант D:
Что-то другое.

IV.2: Q/A

- Puzzler для скучающих (“Что напечатается?”):

```
private final AtomicInteger counter = new AtomicInteger();

private void run() throws ExecutionException, InterruptedException {
    ExecutorService tpe = new ThreadPoolExecutor(...);
    ForkJoinPool fjp = new ForkJoinPool(...);
    Callable callable = new Callable() ... { return counter.getAndIncrement(); }
    RecursiveTask task = new RecursiveTask() ... { return counter.getAndIncrement(); }

    counter.set(0);
    for (int c = 0; c < 10; c++) { print(tpe.submit(callable).get()); }
    println(", counter = " + counter.get());

    counter.set(0);
    for (int c = 0; c < 10; c++) { print(fjp.submit(callable).get()); }
    println(", counter = " + counter.get());

    counter.set(0);
    for (int c = 0; c < 10; c++) { print(fjp.submit(task).get()); }
    println(", counter = " + counter.get());
}
```

0123456789, counter = 10
0123456789, counter = 10
0000000000, counter = 1

Вариант D:
Что-то другое.

III.5: Реиспользование ForkJoinTask

- **RecursiveAction extends ForkJoinTask<V>**
- **ForkJoinTask<V> implements Future<V>**
  - Хранит результат
  - Хранит состояние (done, cancelled, exceptional)
  - Использовать один и тот же инстанс опасно ;)
- **ForkJoinTask.reinitialize()**
  - Сбрасывает внутренние состояния
  - Позволяет использовать инстанс опять


JavaOne™