ORACLE

High-Performance Fork/Join

под капотом параллелизма в платформе

Алексей Шипилёв
aleksey.shipilev@oracle.com, @shipilev

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.

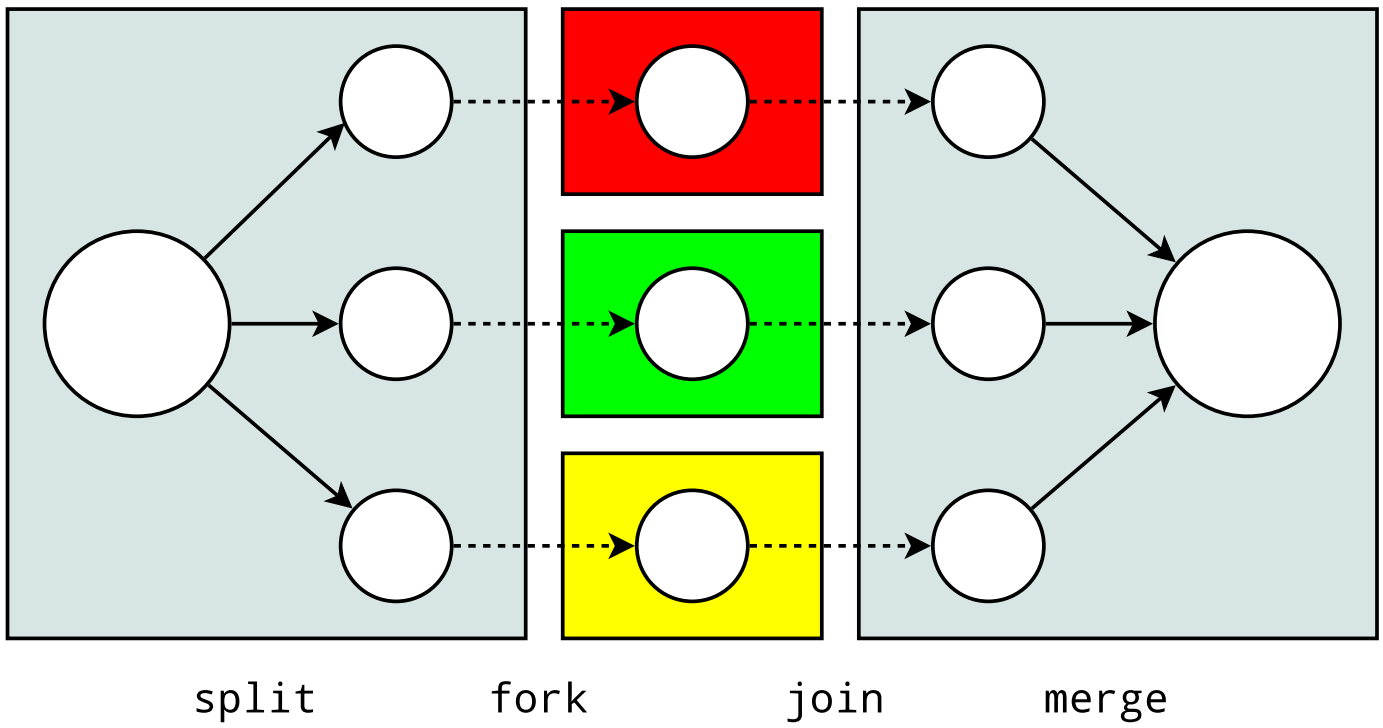
Введение

Введение: предпосылки

- Мы, человеки, хорошо умеем писать последовательный софт
  - Писать паралелльный софт сложно
  - Сохранить иллюзию последовательности?
  - Хардвар всё равно последовательный?
- «Разделяй и властвуй»
  - Разбить большую задачу на подзадачи
  - Выполнить подзадачи параллельно
  - Поклеить результаты

Введение: уже и так всё ясно

Каждый девелопер **уже** писал Fork/Join:



Введение: проблемы

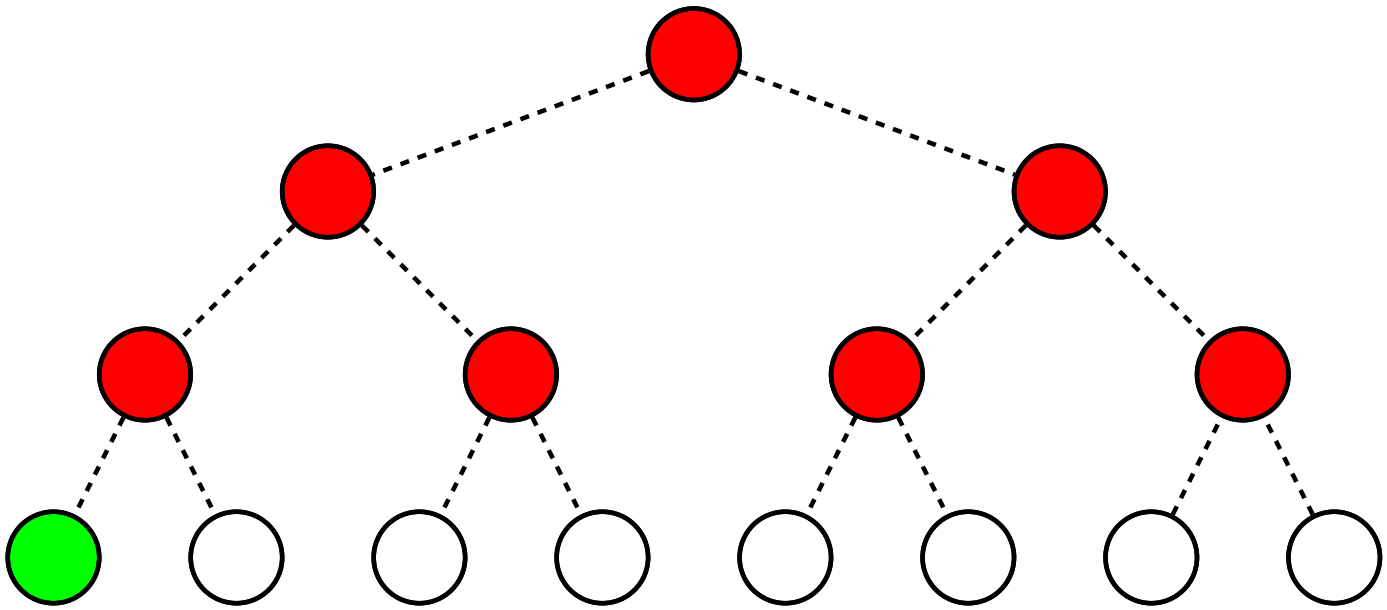
- Привет, закон Амдала:
  - Порезка задачи последовательна
  - Склейка результатов последовательна
  - (доминирует время исполнения)
- Балансировка нагрузки
  - Поднятие потоков
  - Раздача работы
- Иерархические декомпозиции
  - Гарантии прогресса

Введение: общий вид

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

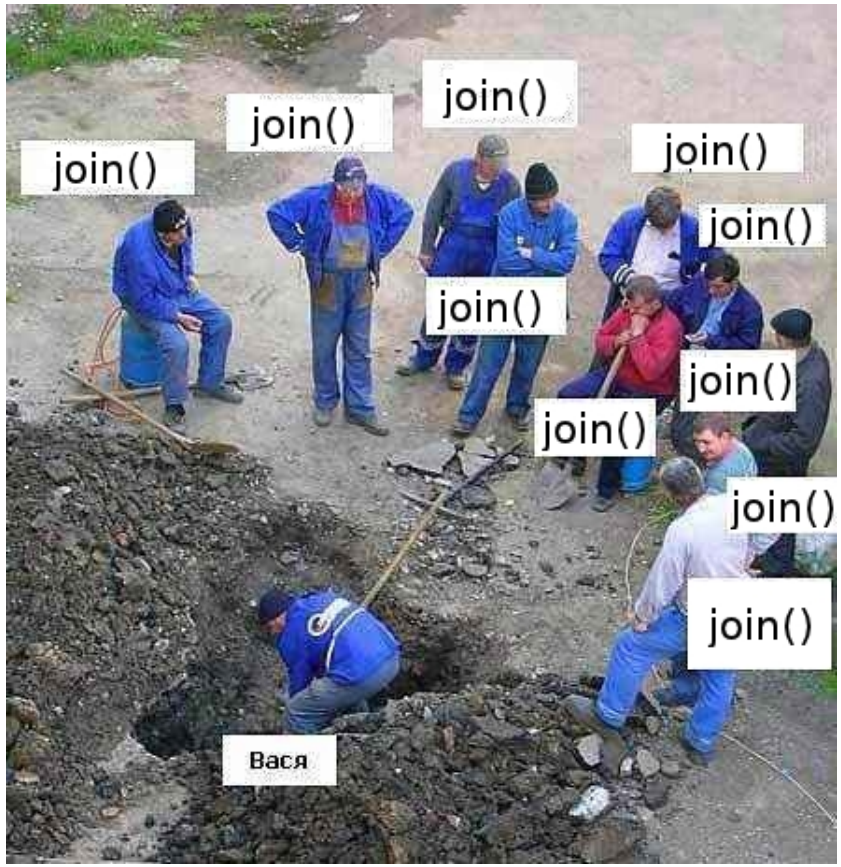

Введение: банальные проблемы

Для N-арного дерева глубины K:
в худшем случае $\frac{N^K-1}{N-1}+1$ потоков



Введение: суть

- Блокирующее ожидание теряет активный поток
- Идея: `join()` не должен занимать поток!
- Заставим «застрявшие» потоки работать: требуется нехилая координация



Для начинающих

Для начинающих: API

- `ForkJoinPool`
  - `...extends ExecutorService`
  - принимает в себя `Runnable`'ы, `Callable`'ы
  - и вообще работает как обычный пул

- `ForkJoinTask<V>`
  - общий предок всех задач в FJP
  - удобнее стандартные подклассы:
  - `RecursiveTask<V>` – с газом
  - `RecursiveAction` – без газа

Для начинающих: API

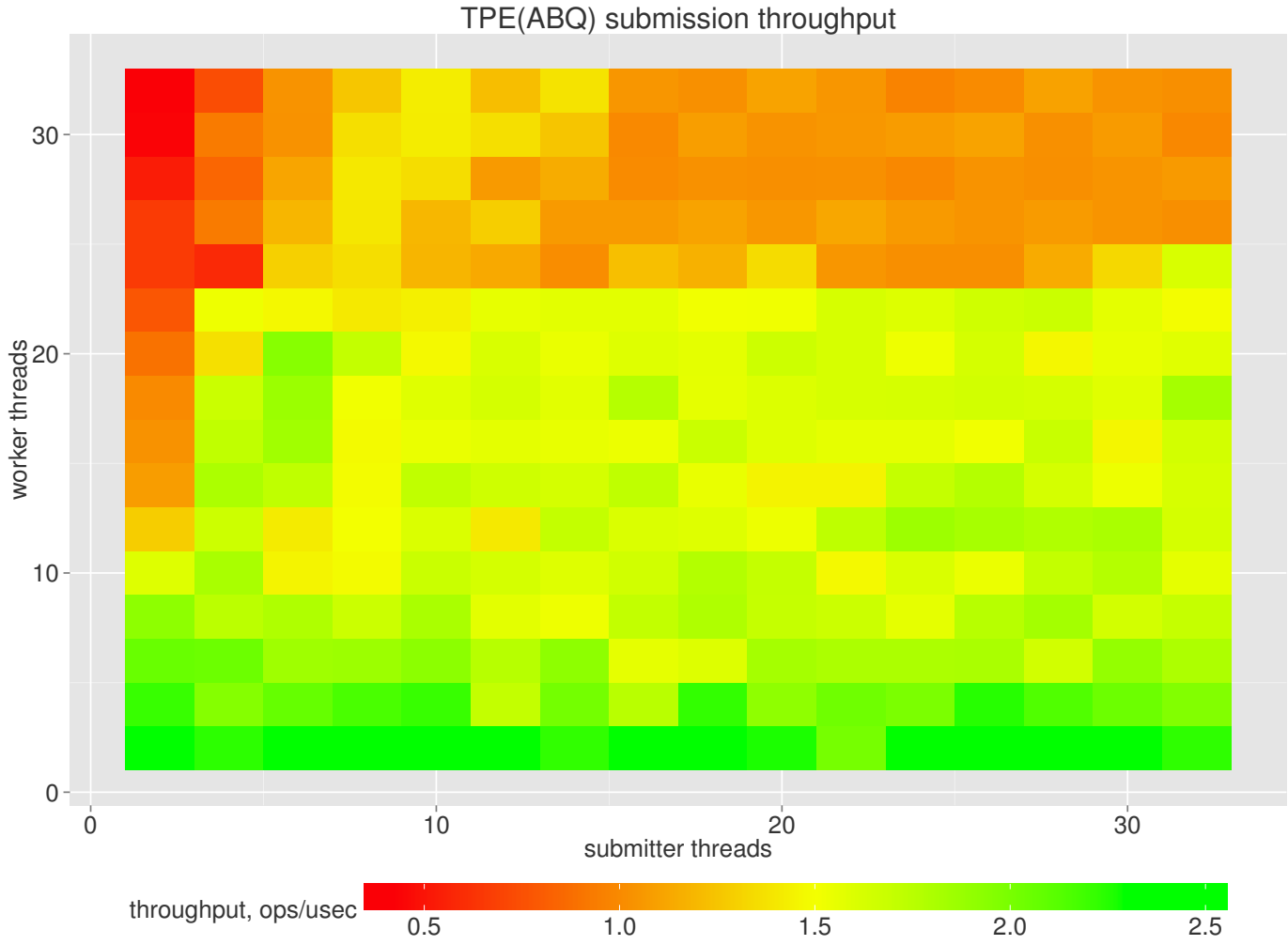
```
ForkJoinPool fjp = new ForkJoinPool();

fjp.submit(new MyRunnable());
fjp.invoke(new MyCallable());
fjp.invoke(new MyRecursiveTask());

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

ORACLE

Балансировка задач: подходы

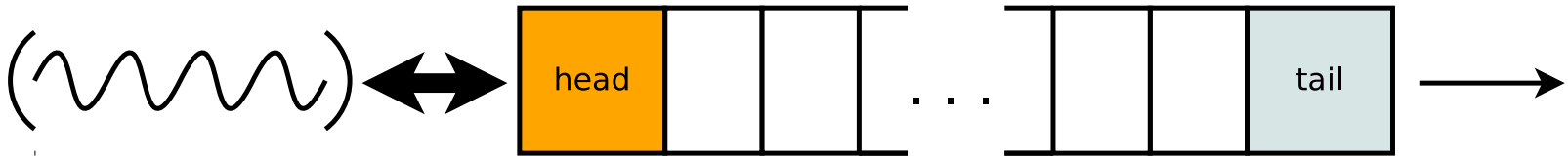
- балансировка хорошо решается в динамике
- три базовых подхода:
  1. **Work arbitrage**: общий арбитр, раздающий задачи; часто обычная blocking queue
  2. **Work dealing**: у каждого свой набор задач, перегруженные потоки отдают свои задачи на сторону
  3. **Work stealing**: у каждого свой набор задач, свободные потоки «крадут» задачи у перегруженных

Балансировка задач: очередь?



2x8x2 SandyBridge, RHEL 5.5, JDK 8b89 + jsr166 2013-05-15

Балансировка задач: Work stealing



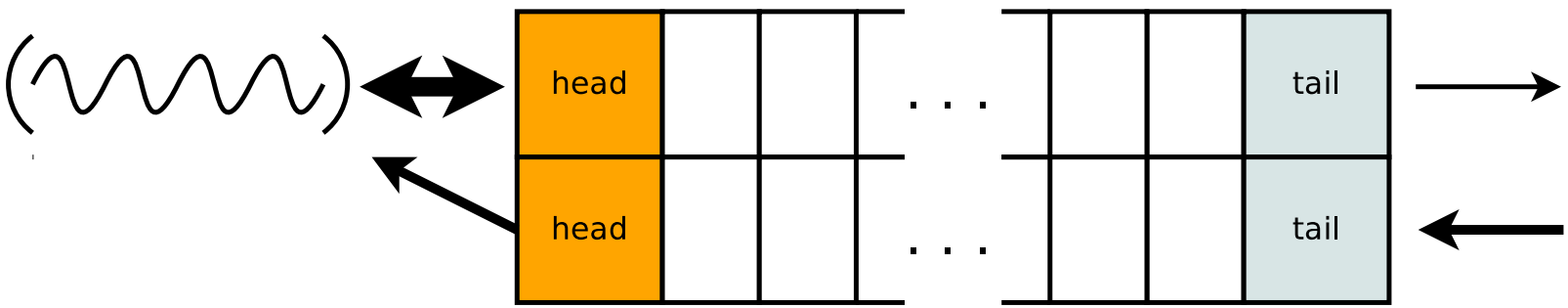
- Локальная очередь для каждого потока
  - lock-free `WorkQueue` $\sim$ `ForkJoinTask<?>[]`
  - тщательно изолирована от остальных данных
- Владелец работает с головой очереди
  - без синхронизации!
- Другие потоки могут «тырить» из хвоста
  - «stealing» с минимальной синхронизацией

Балансировка задач: submit

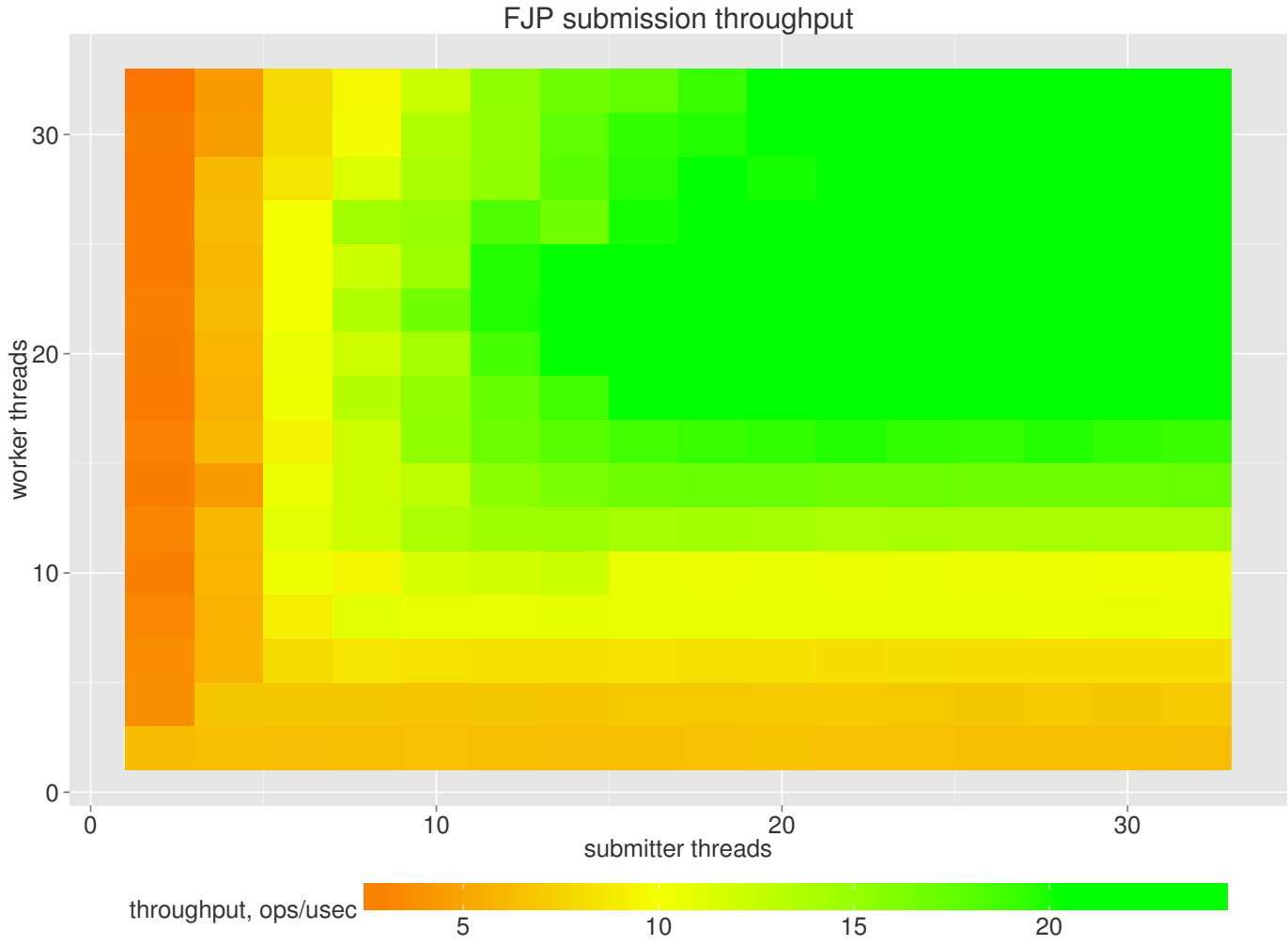
- Куда происходит `submit()` внешних задач?

Балансировка задач: submit

- Куда происходит `submit()` внешних задач?
  - в голову случайной очереди? синхронизация!
  - в хвост случайной очереди? FIFO!
- Отдельная очередь для внешних задач?
- Расклеить очереди входных задач!

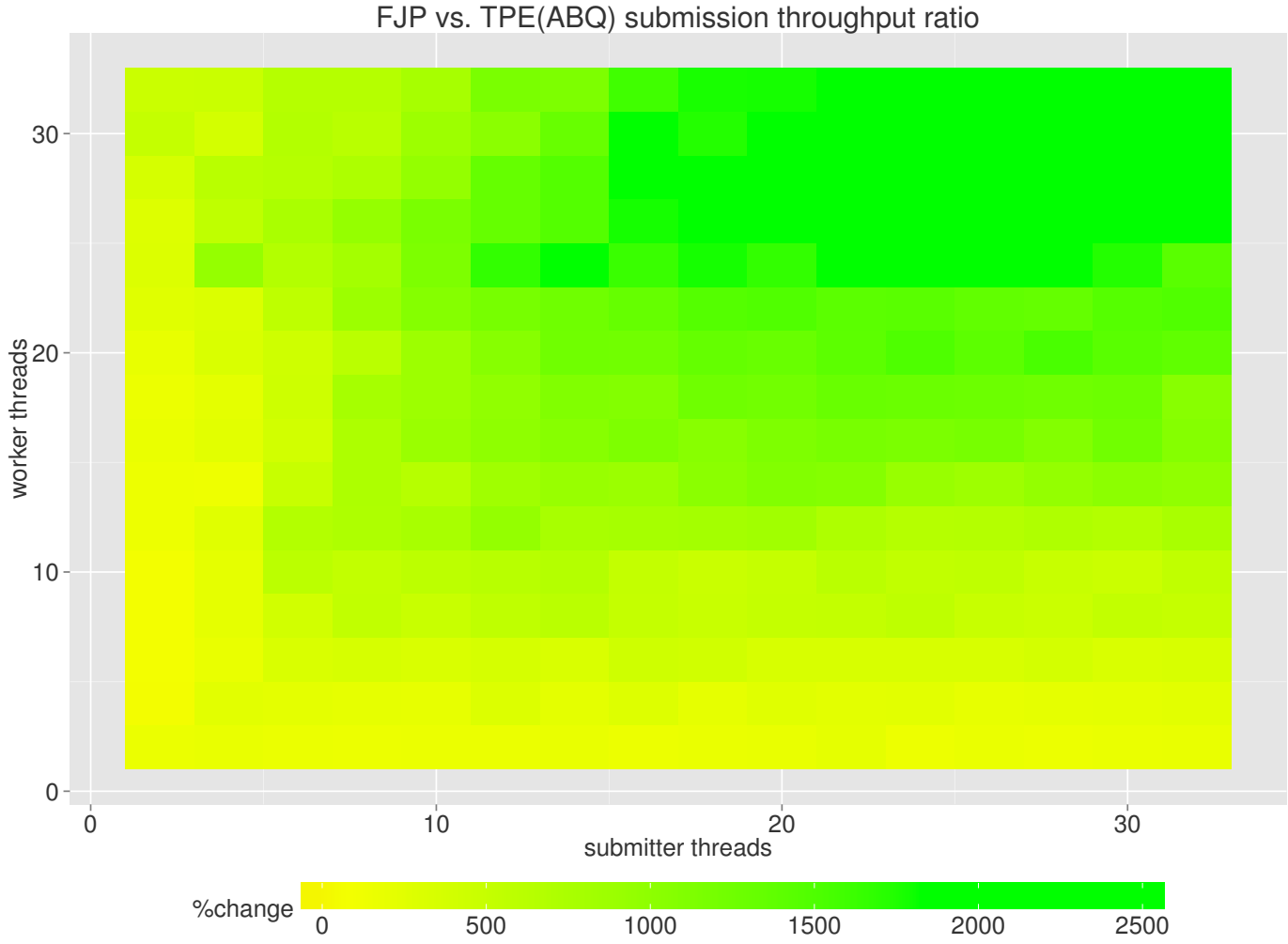


Балансировка задач: результат



2x8x2 SandyBridge, RHEL 5.5, JDK 8b89 + jsr166 2013-05-15

Балансировка задач: результат



2x8x2 SandyBridge, RHEL 5.5, JDK 8b89 + jsr166 2013-05-15

Типичная задача: в коде

```
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(); // fork 1
        t2.fork(); // fork 2

        long res = 0;
        res += t2.join(); // join 2
        res += t1.join(); // join 1
        return res;
    }
}
```

ORACLE

Типичная задача: в коде, поближе

```
ForkJoinTask<Long> t1, t2;

int mid = (l + r) >>> 1;
t1 = new StandardTask(problem, l, mid);
t2 = new StandardTask(problem, mid, r);
t1.fork(); // fork 1
t2.fork(); // fork 2

long res = 0;
res += t2.join(); // join 2
res += t1.join(); // join 1
return res;
```


Типичная задача: ещё ближе

```
ForkJoinTask<Long> t1, t2;

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

ForkJoinTask.invokeAll(t1, t2);

long res = 0;
res += t2.join(); // get result 1
res += t1.join(); // get result 2
return res;
```


Типичная задача: fork().join()

- `fork()`
  - кладёт в очередь и возвращается
  - даёт возможность «украсть» задачу
- `join()`
  - «ждать, пока закончится»
  - даёт возможность занять поток ещё чем-нибудь

Типичная задача: join()

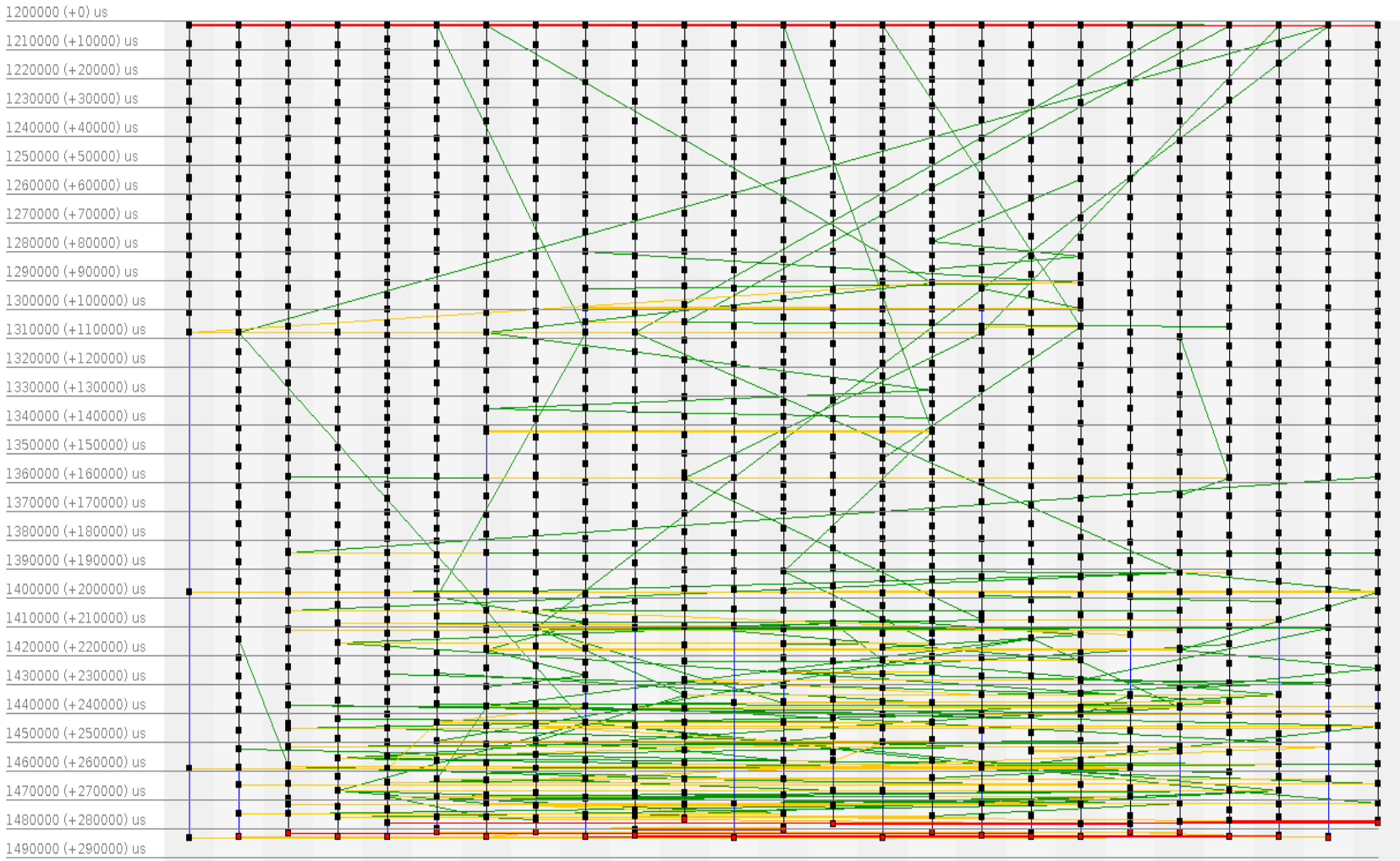
- магия начинается здесь!

- можно сделать одно из действий:
  1. выполнить эту join-ящуюся задачу
  2. найти в очереди задачу и выполнить
  3. пойти в чужую очередь и «украсть» задачу
  4. ...
  5. заблокироваться

Главный козырь: идеи

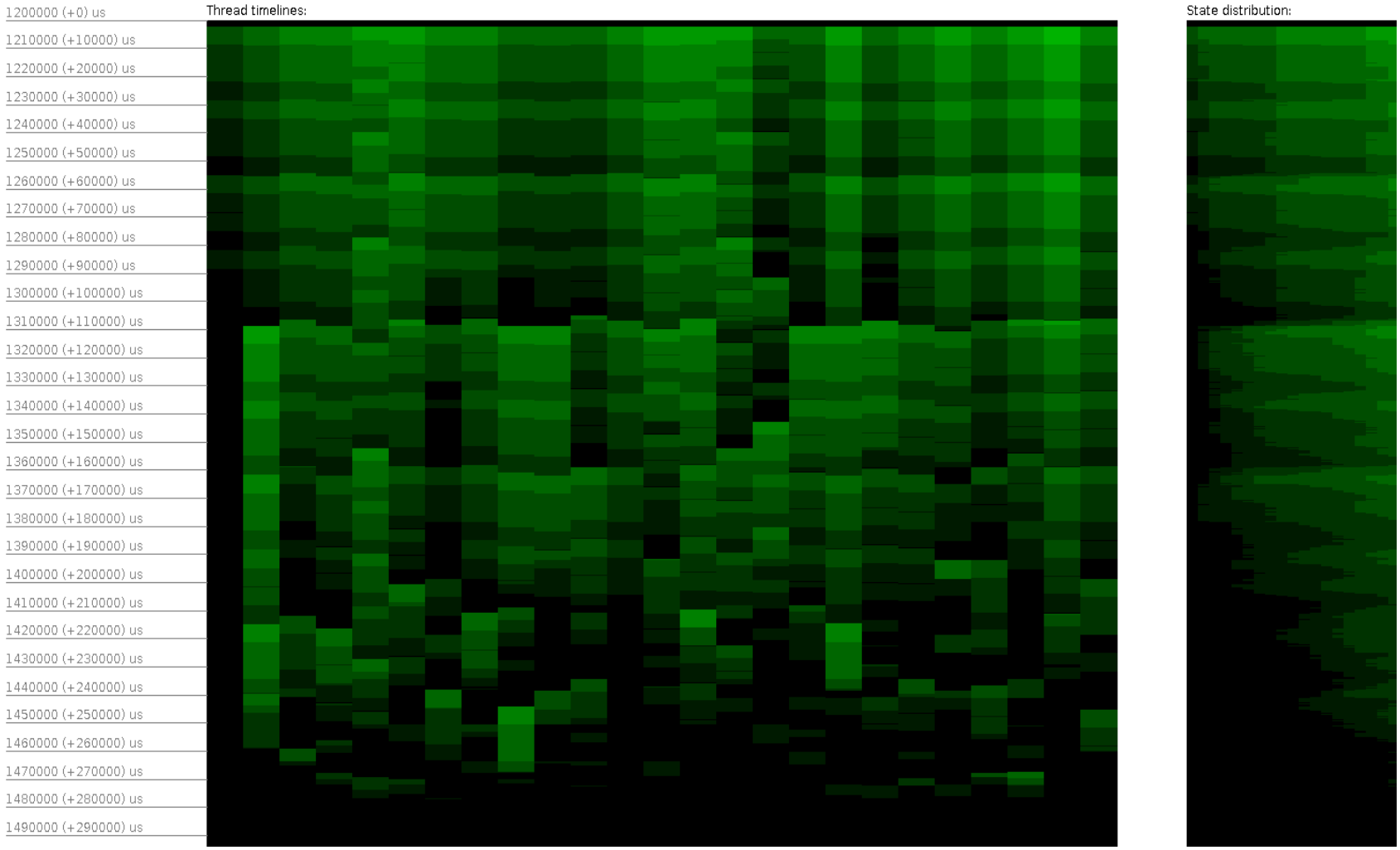
- обычно в локальной очереди куча задач
- почти все потоки почти всегда заняты
- балансировка требуется редко
- «украденная» задача засевает очередь

Главный козырь: Task Trees



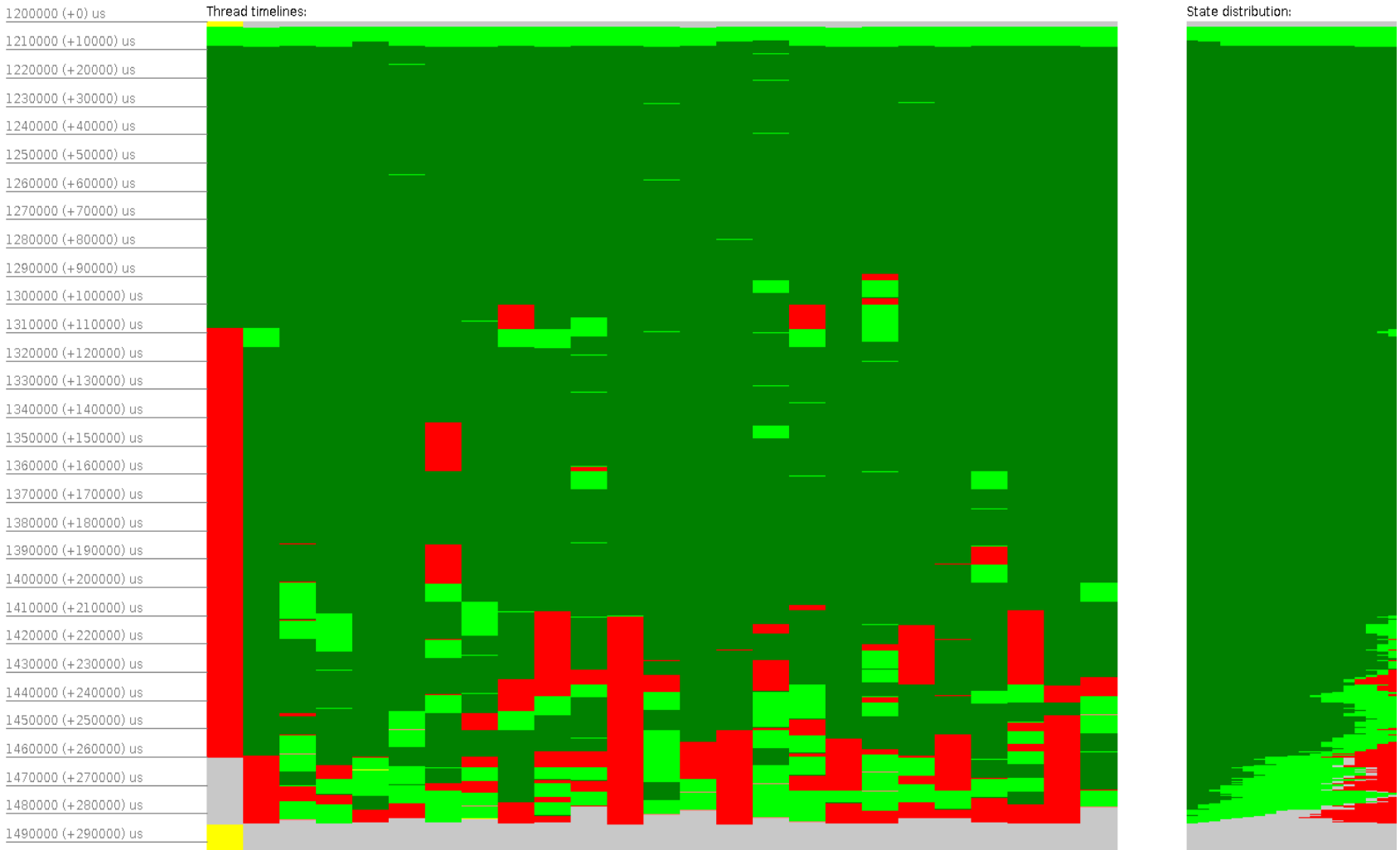
2x8x2 SandyBridge, RHEL 5.5, JDK 8b89 + jsr166 2013-05-15

Главный козырь: Queue Occupancy



2x8x2 SandyBridge, RHEL 5.5, JDK 8b89 + jsr166 2013-05-15

Главный козырь: Workers State



2x8x2 SandyBridge, RHEL 5.5, JDK 8b89 + jsr166 2013-05-15

Выбор ширины: проблема

```
protected Long compute() {
    if (r - l <= THRESHOLD) {
        return seq();
    }
    return par();
}
```

- как выбрать THRESHOLD?
- слишком маленький = куча объектов-задач
- слишком большой = мало параллелизма

Выбор ширины: наивный подход

$$T = \frac{N}{C},$$

где

N – размер задачи,

C – количество CPU

Выбор ширины: наивный подход

$$T = \frac{N}{C},$$

где
N – размер задачи,
C – количество CPU

Плохо работает при неравномерных задачах, несимметричных потоках, непредсказуемых задержках.

Выбор ширины: правильный метод

$$T = \frac{N}{CL},$$

где

N – размер задачи,

C – количество CPU,

L – load factor

Выбор ширины: правильный метод

$$T = \frac{N}{CL},$$

где

N – размер задачи,

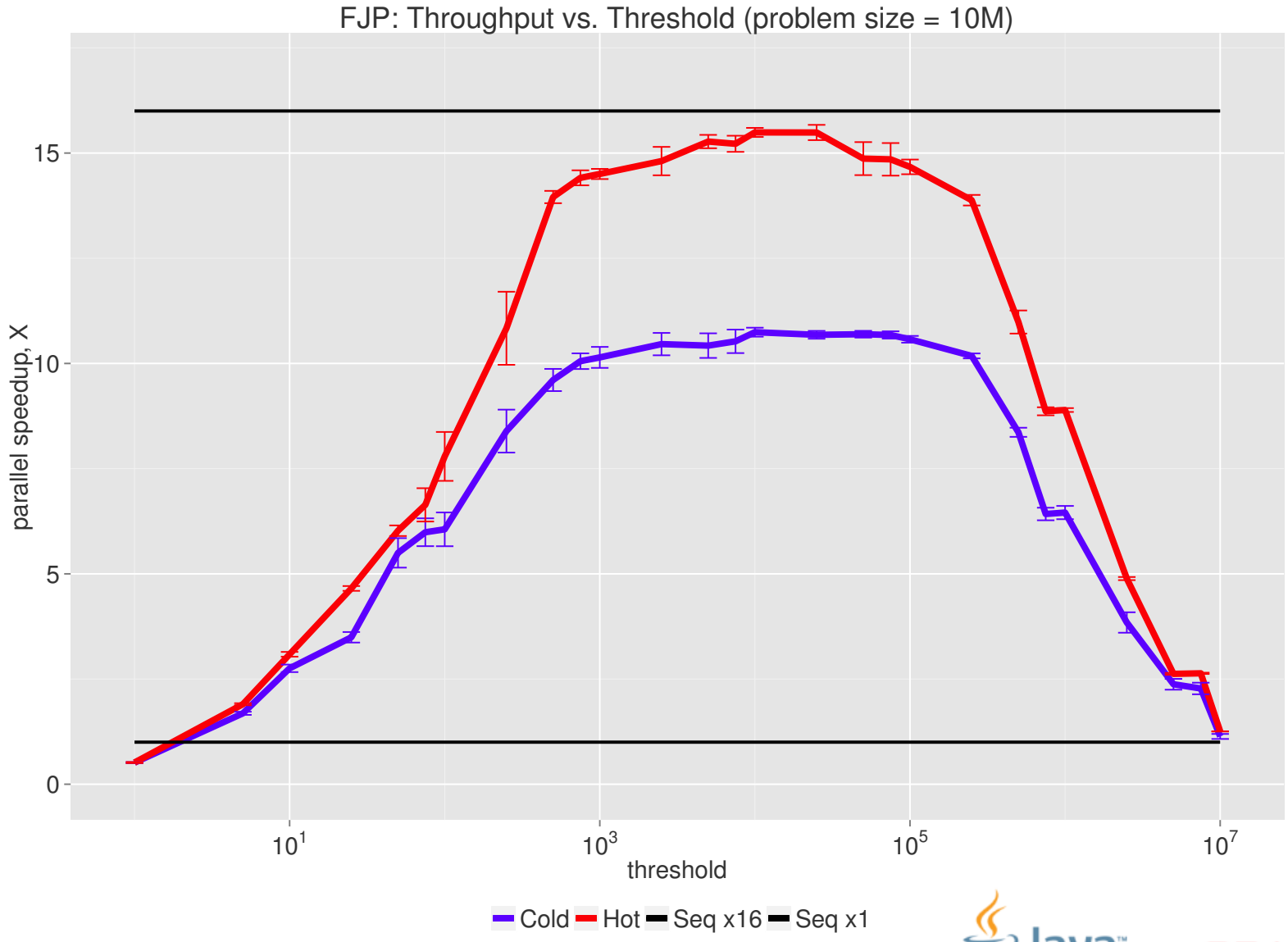
C – количество CPU,

L – load factor

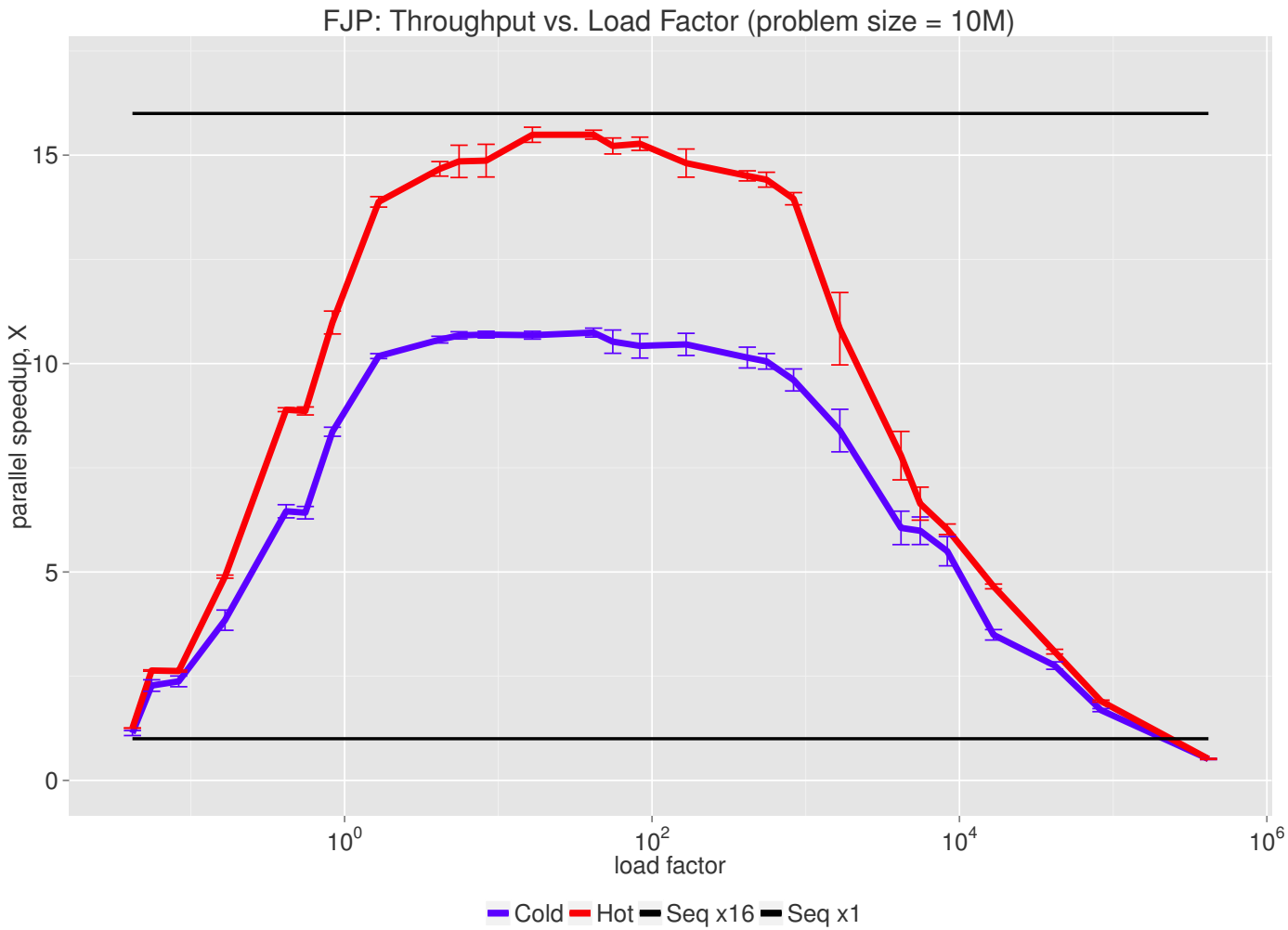
Load factor даёт «запас» задач, которые амортизируют задержки и неравномерности.

Обычное значение $L \in [10..100]$

Выбор ширины: threshold

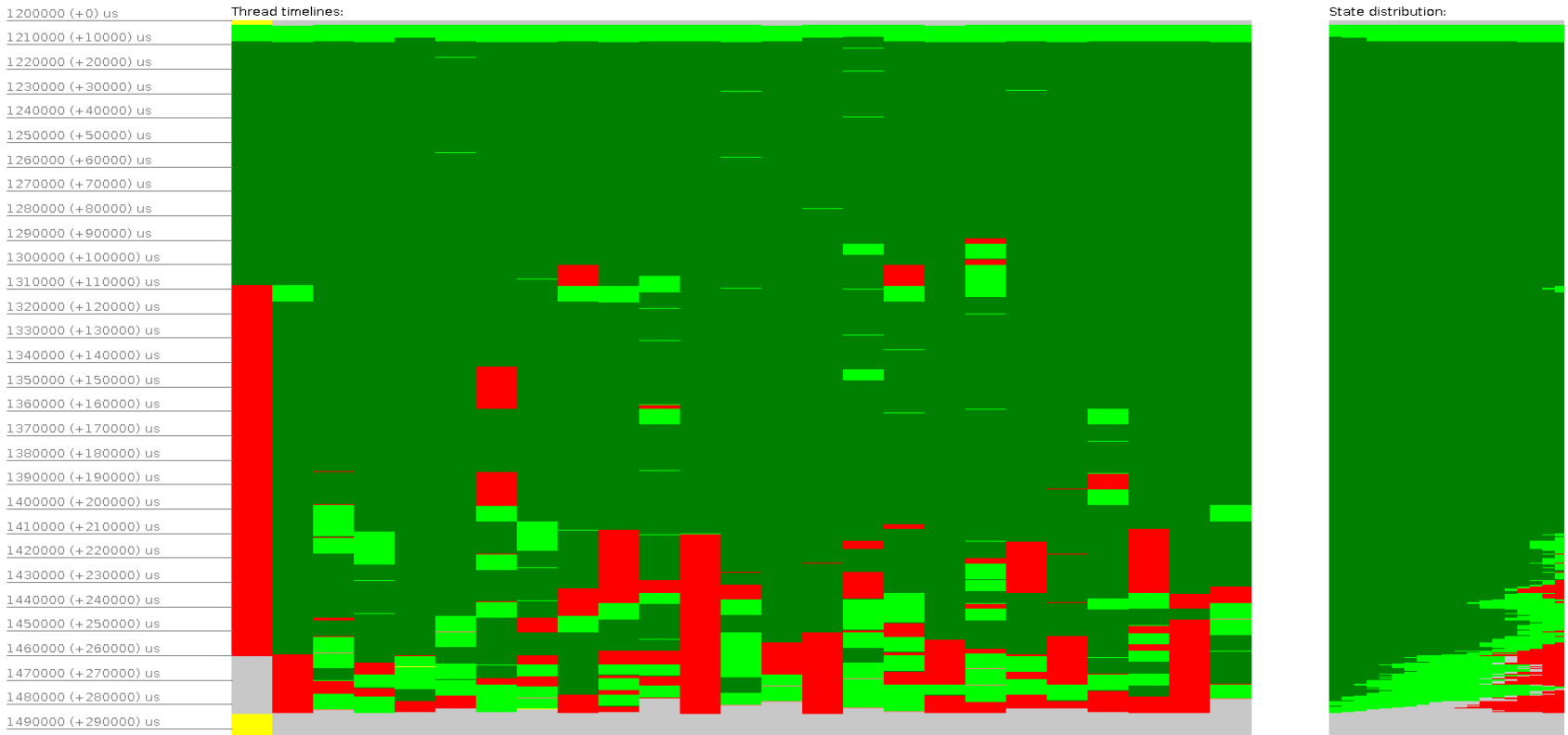


Выбор ширины: load factor

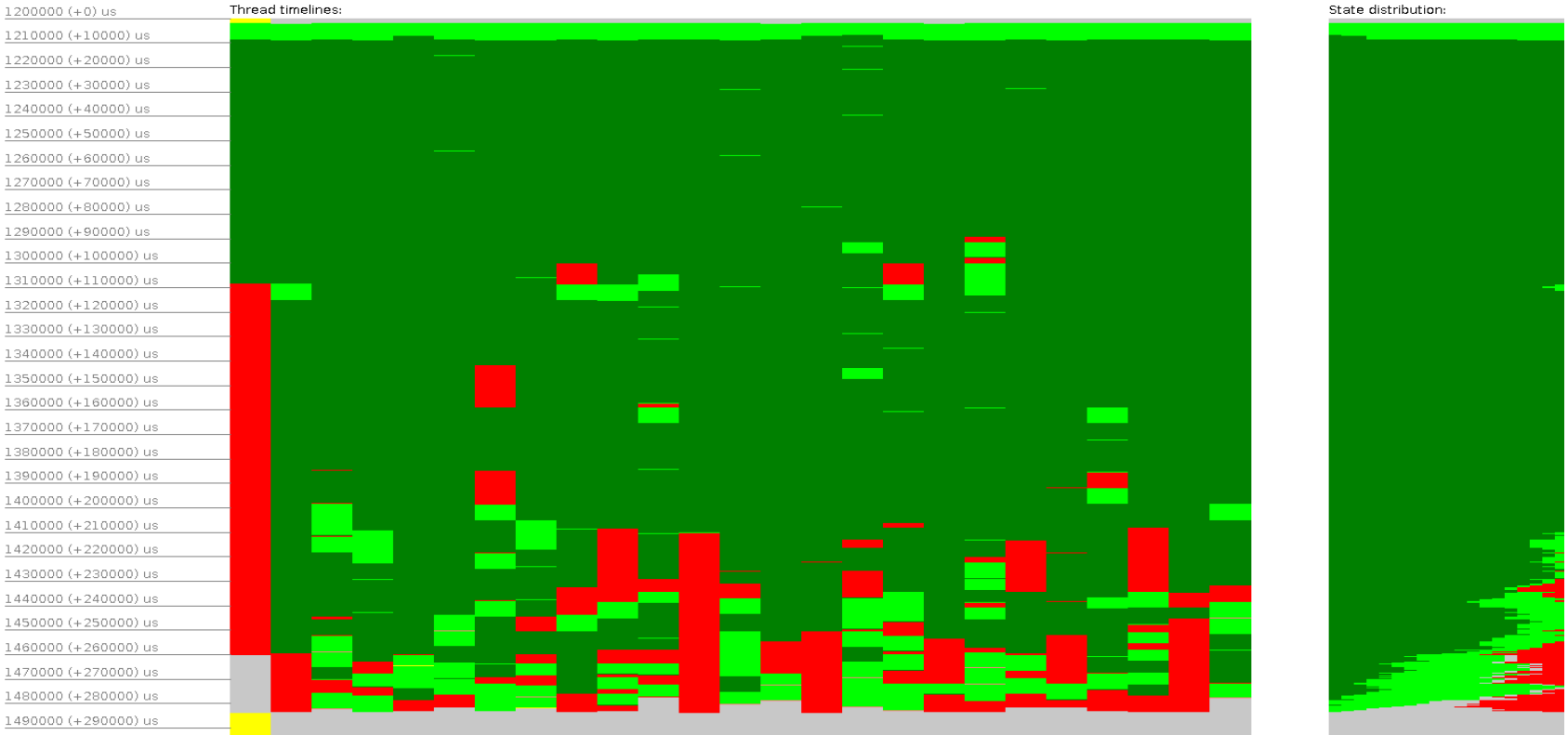


Для продолжающих

CountedCompleter: зачем

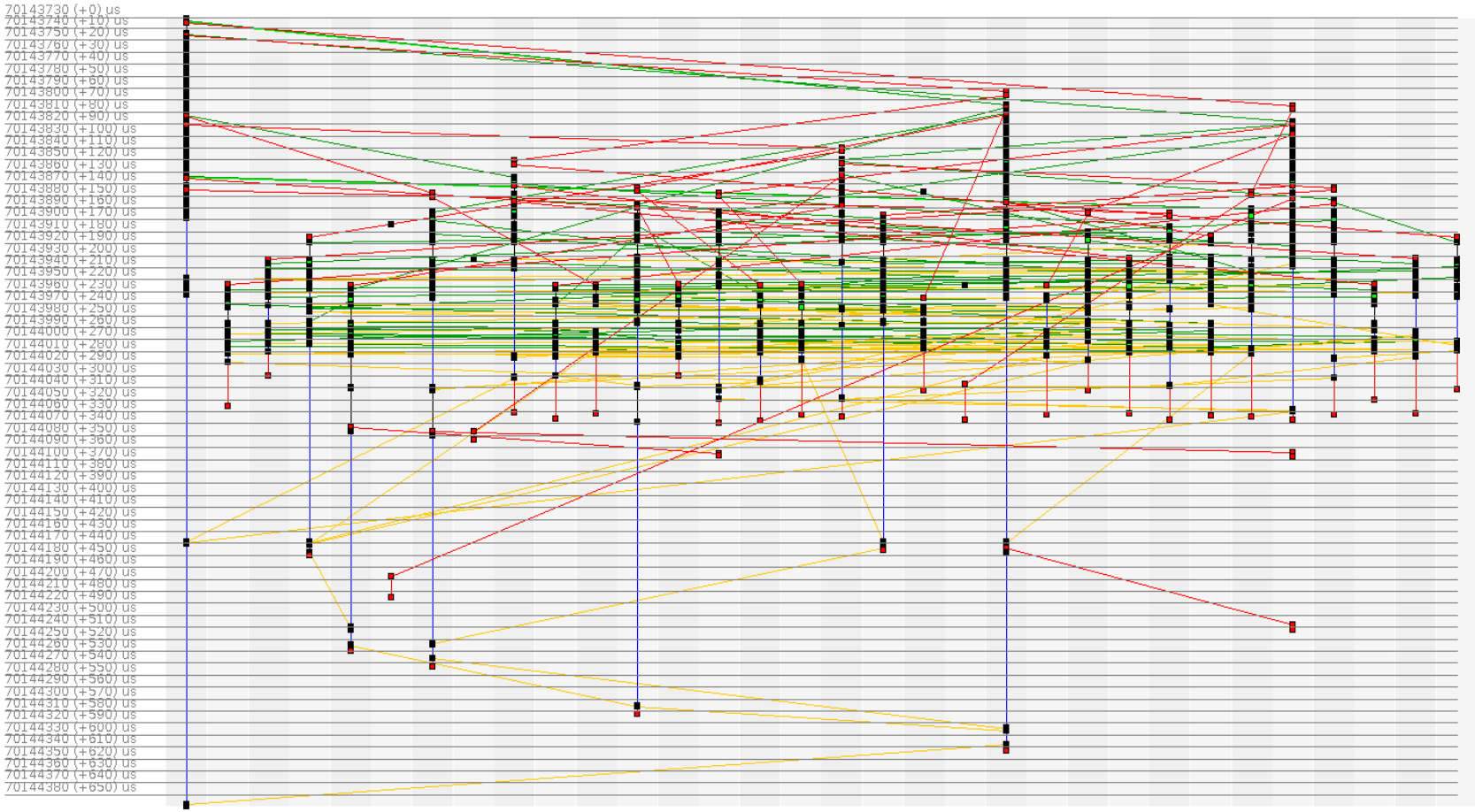


CountedCompleter: зачем



WTF: заблокированные потоки в хвосте?

CountedCompleter: помельче



Рекурсивные `join()`-ы в хвосте!

CountedCompleter: идея

- в отсутствие работы потоки блокируются
- разбудить поток – целое дело
- чем глубже разбиение, тем больше будить
- на мелких задачах пожирает время

CountedCompleter: идея

- в отсутствие работы потоки блокируются
- разбудить поток – целое дело
- чем глубже разбиение, тем больше будить
- на мелких задачах пожирает время

Идея: вот бы у нас были continuation-ы...

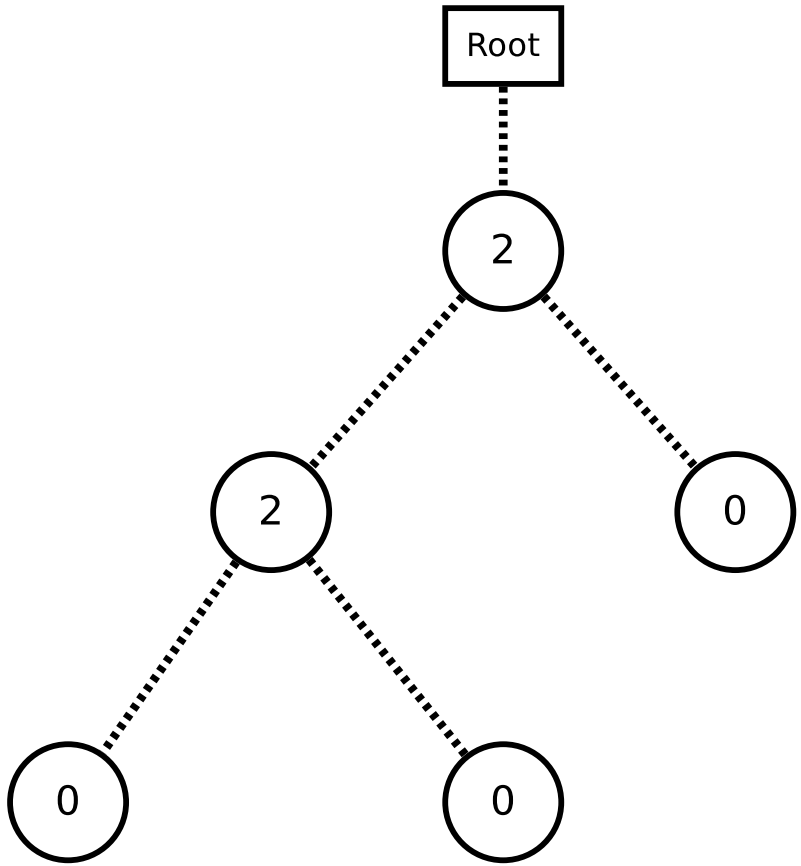
CountedCompleter: их есть у нас!

```
public abstract class CountedCompleter<T>
 extends ForkJoinTask<T> {

    // bind to parent
    public CountedCompleter(CountedCompleter<?> parent) { ... }

    // compute as usual
    public abstract void compute() { ... }

    // pending count = how many subtasks are still alive
    public void setPendingCount(int count) { ... }

    // try to complete
    public void tryComplete() { ... }

    // to be called on completion
    public void onCompletion() { ... }
}
```

ORACLE

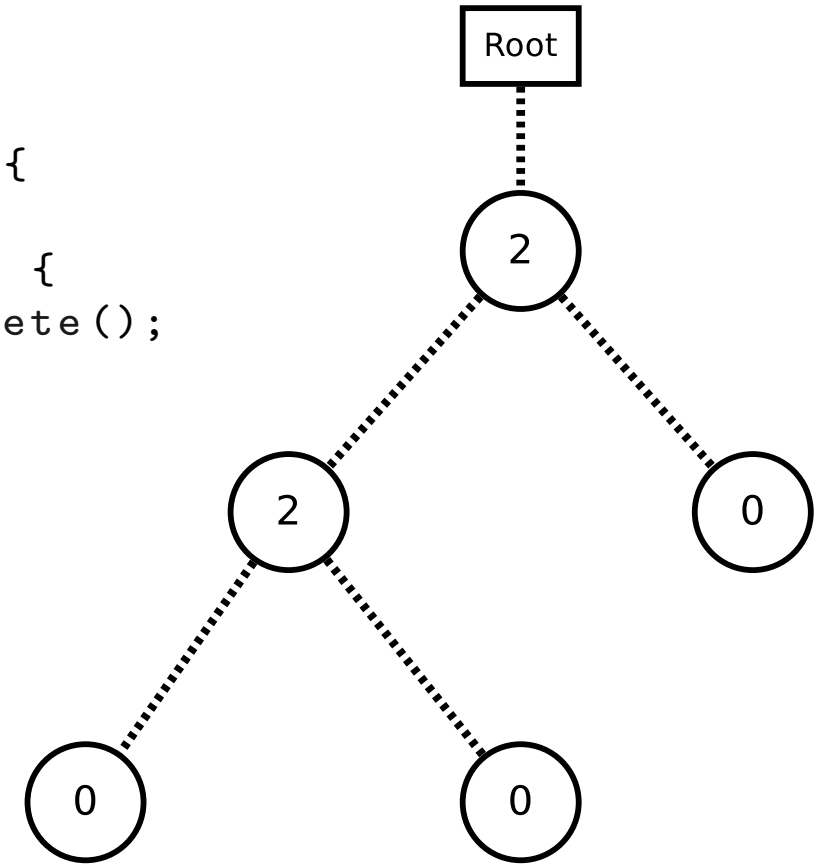
CountedCompleter: логика

- образуют дерево на задачах
- У каждой задачи есть `pendingCount` = количество невыполненных подзадач



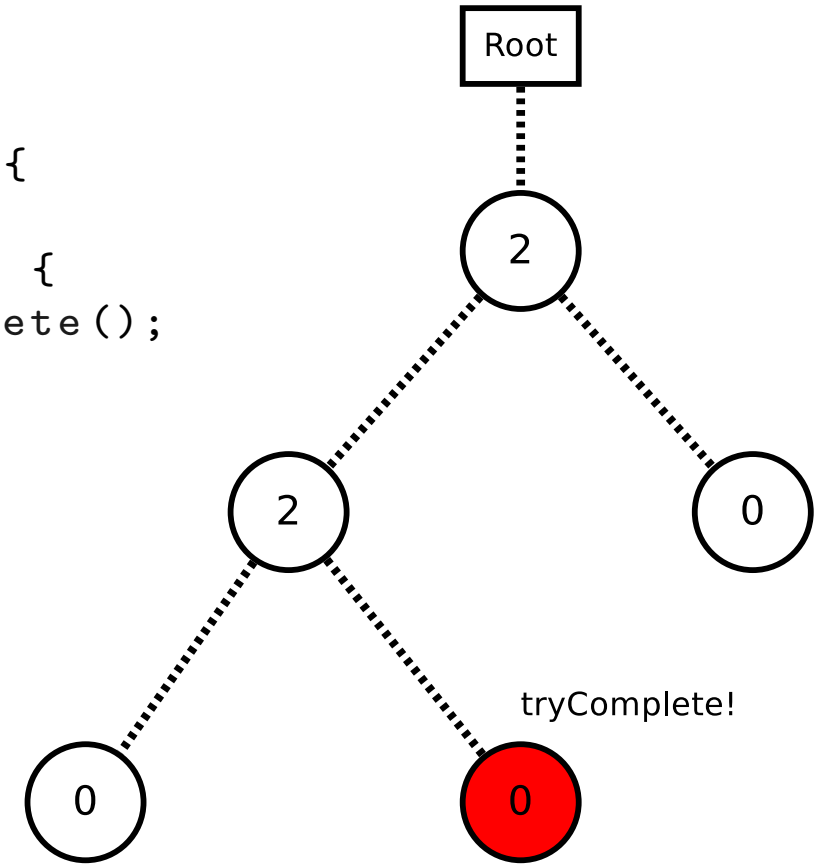
CountedCompleter: логика

```
void tryComplete() {
    pendingCount--;
    if (pendingCount <= 0) {
        onCompletion();
        if (parent != null) {
            parent.tryComplete();
        }
    }
}
```



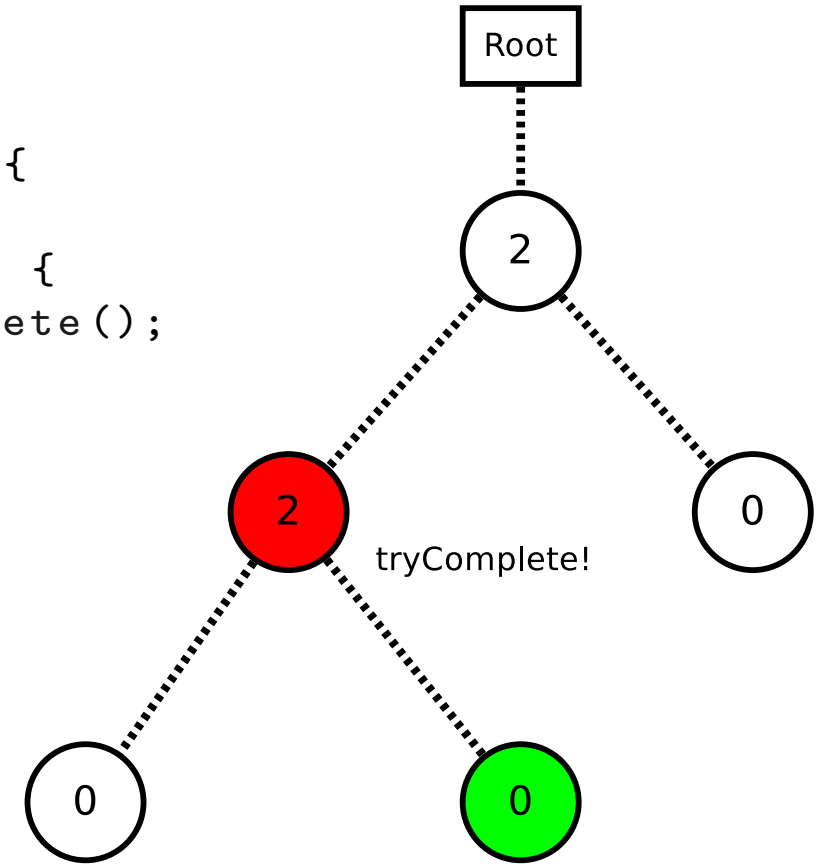
CountedCompleter: логика

```
void tryComplete() {
    pendingCount--;
    if (pendingCount <= 0) {
        onCompletion();
        if (parent != null) {
            parent.tryComplete();
        }
    }
}
```



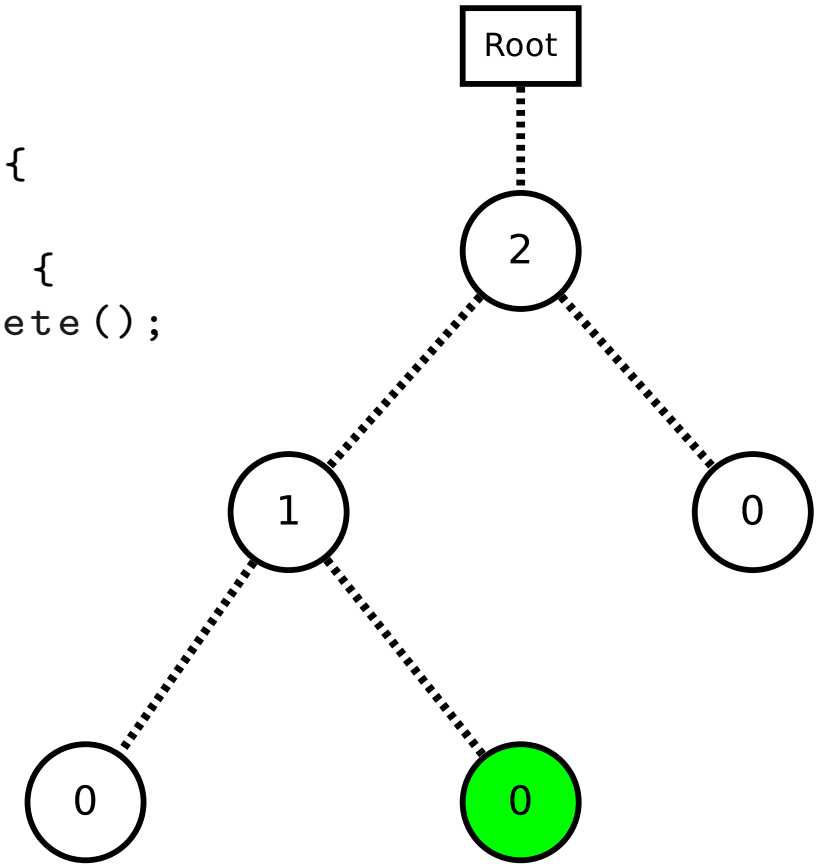
CountedCompleter: логика

```
void tryComplete() {
    pendingCount--;
    if (pendingCount <= 0) {
        onCompletion();
        if (parent != null) {
            parent.tryComplete();
        }
    }
}
```



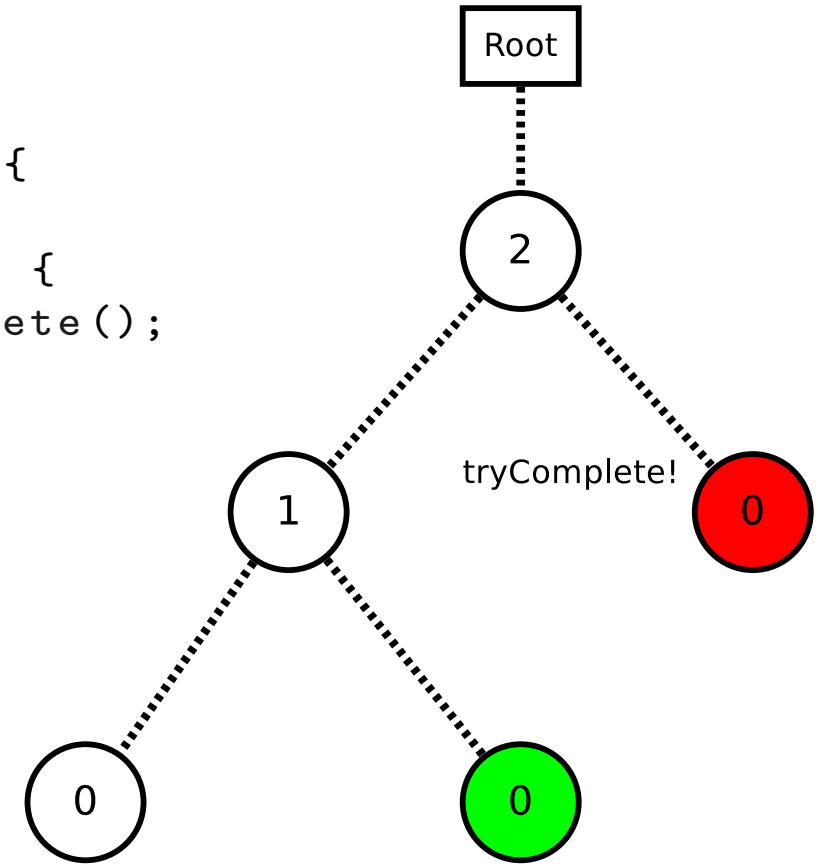
CountedCompleter: логика

```
void tryComplete() {
    pendingCount--;
    if (pendingCount <= 0) {
        onCompletion();
        if (parent != null) {
            parent.tryComplete();
        }
    }
}
```



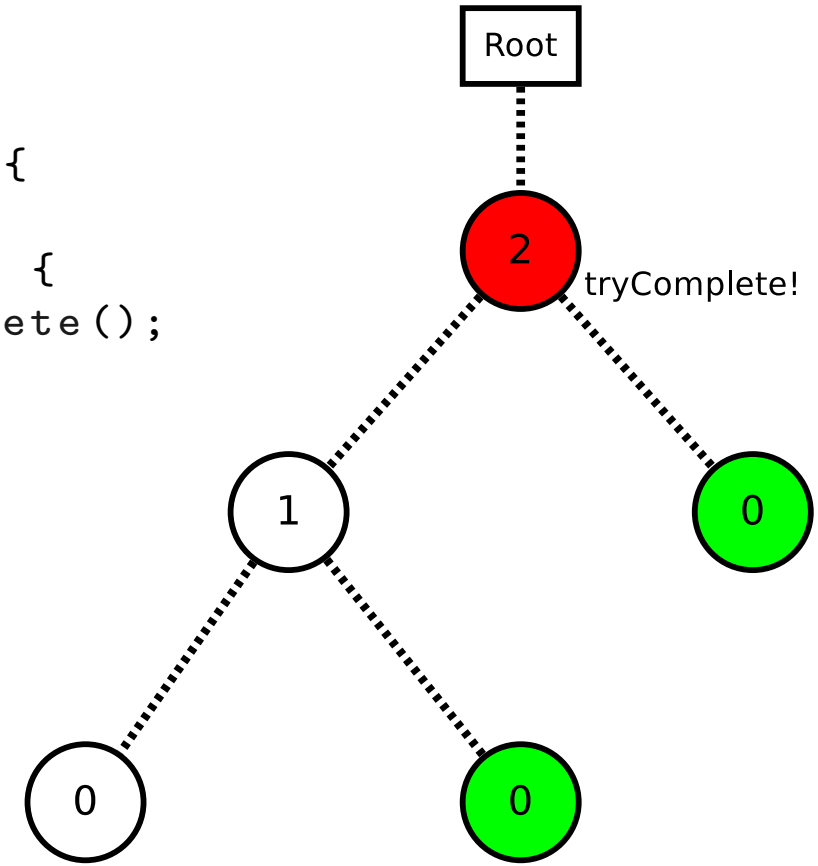
CountedCompleter: логика

```
void tryComplete() {
    pendingCount--;
    if (pendingCount <= 0) {
        onCompletion();
        if (parent != null) {
            parent.tryComplete();
        }
    }
}
```



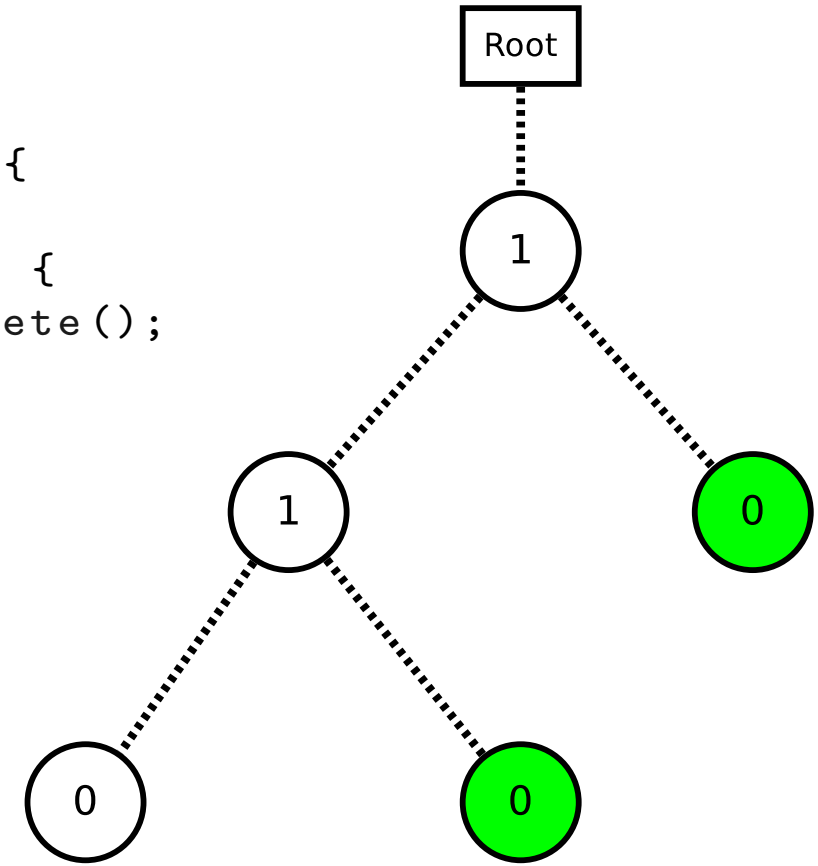
CountedCompleter: логика

```
void tryComplete() {
    pendingCount--;
    if (pendingCount <= 0) {
        onCompletion();
        if (parent != null) {
            parent.tryComplete();
        }
    }
}
```



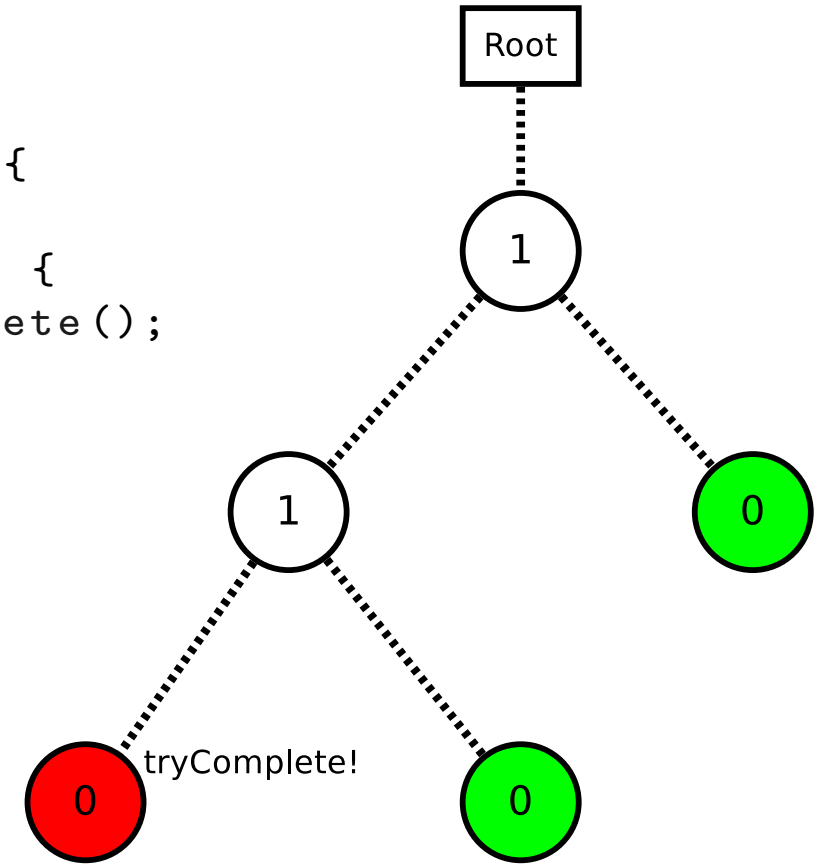
CountedCompleter: логика

```
void tryComplete() {
    pendingCount--;
    if (pendingCount <= 0) {
        onCompletion();
        if (parent != null) {
            parent.tryComplete();
        }
    }
}
```



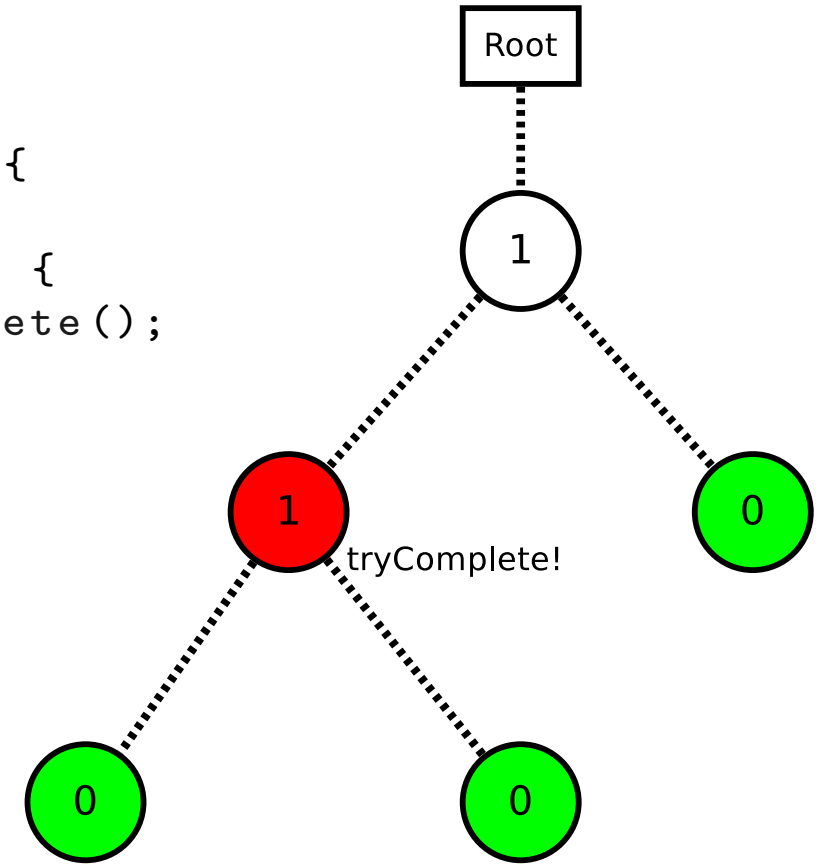
CountedCompleter: логика

```
void tryComplete() {
    pendingCount--;
    if (pendingCount <= 0) {
        onCompletion();
        if (parent != null) {
            parent.tryComplete();
        }
    }
}
```



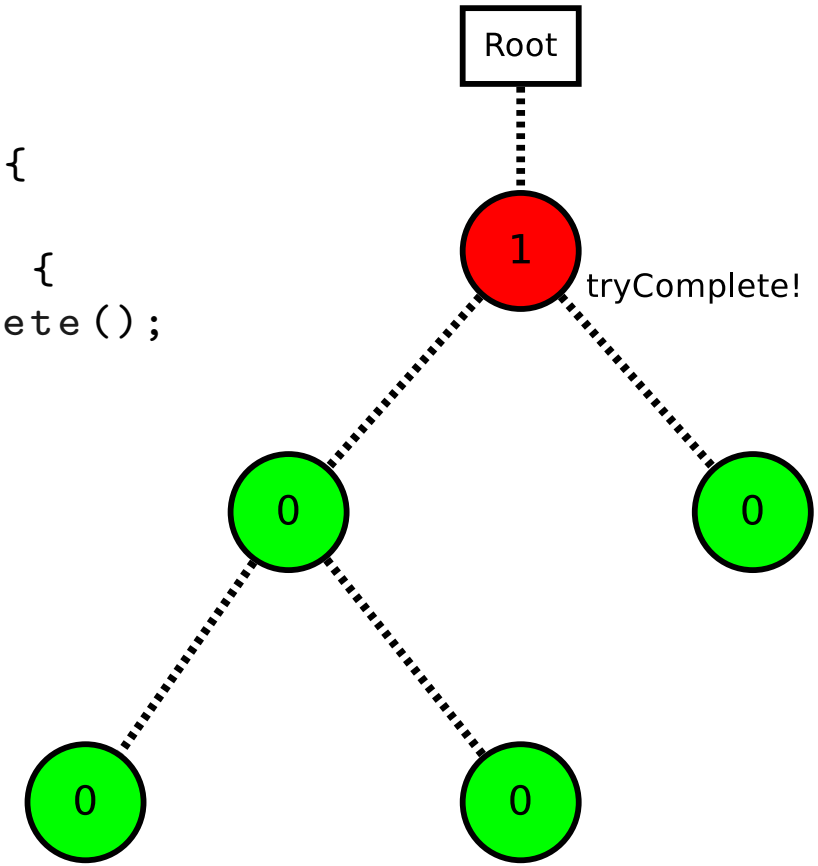
CountedCompleter: логика

```
void tryComplete() {
    pendingCount--;
    if (pendingCount <= 0) {
        onCompletion();
        if (parent != null) {
            parent.tryComplete();
        }
    }
}
```



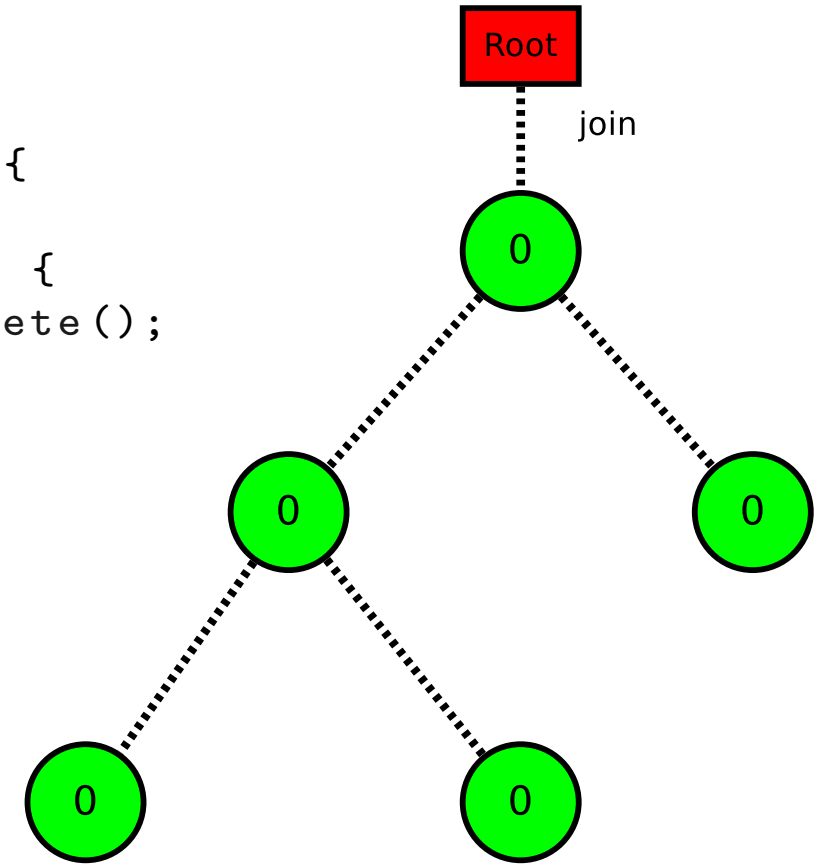
CountedCompleter: логика

```
void tryComplete() {
    pendingCount--;
    if (pendingCount <= 0) {
        onCompletion();
        if (parent != null) {
            parent.tryComplete();
        }
    }
}
```



CountedCompleter: логика

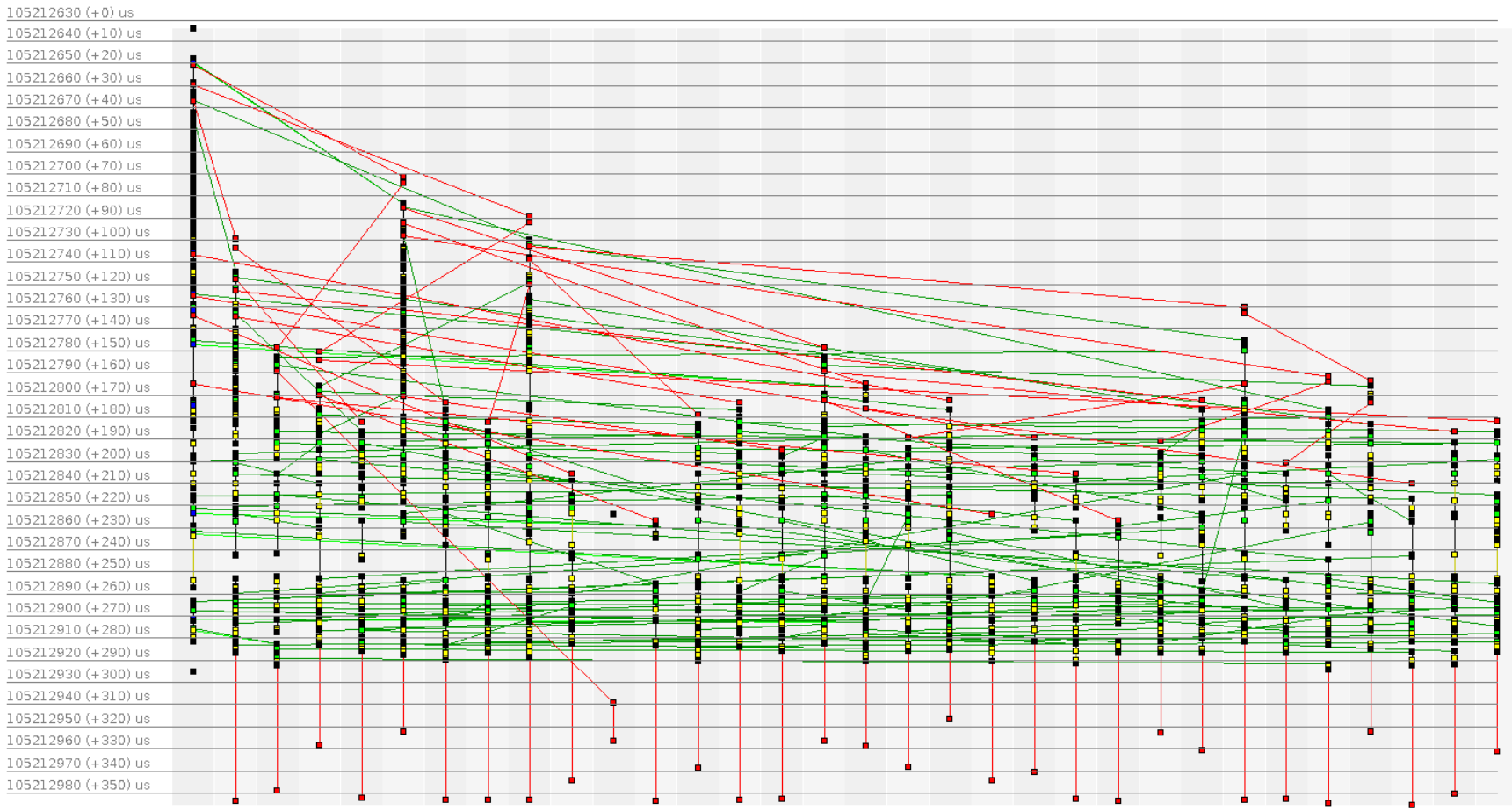
```
void tryComplete() {
    pendingCount--;
    if (pendingCount <= 0) {
        onCompletion();
        if (parent != null) {
            parent.tryComplete();
        }
    }
}
```



CountedCompleter: в коде

```
public static class MyTask extends CountedCompleter<Void> {

    public MyTask(CountedCompleter<?> parent, ...) {
        super(parent);
        ...
    }

    @Override
    public void compute() {
        if (size < thresh) {
            seq(...);
        } else {
            setPendingCount(2); // expect two subtasks
            new MyTask(this, size / 2, thresh).fork();
            new MyTask(this, size / 2, thresh).fork();
        }
        tryComplete(); // increment and try to call completion
    }
}
```


CountedCompleter: получилось



«никаких разрывов»

CountedCompleter: рецепты

- сильно лучше на мелких задачах

$$\texttt{recursive}: 450 \pm 15\ \mu s$$
$$\texttt{completers}: 290 \pm 10\ \mu s$$

- так же быстры на крупных задачах

CountedCompleter: рецепты

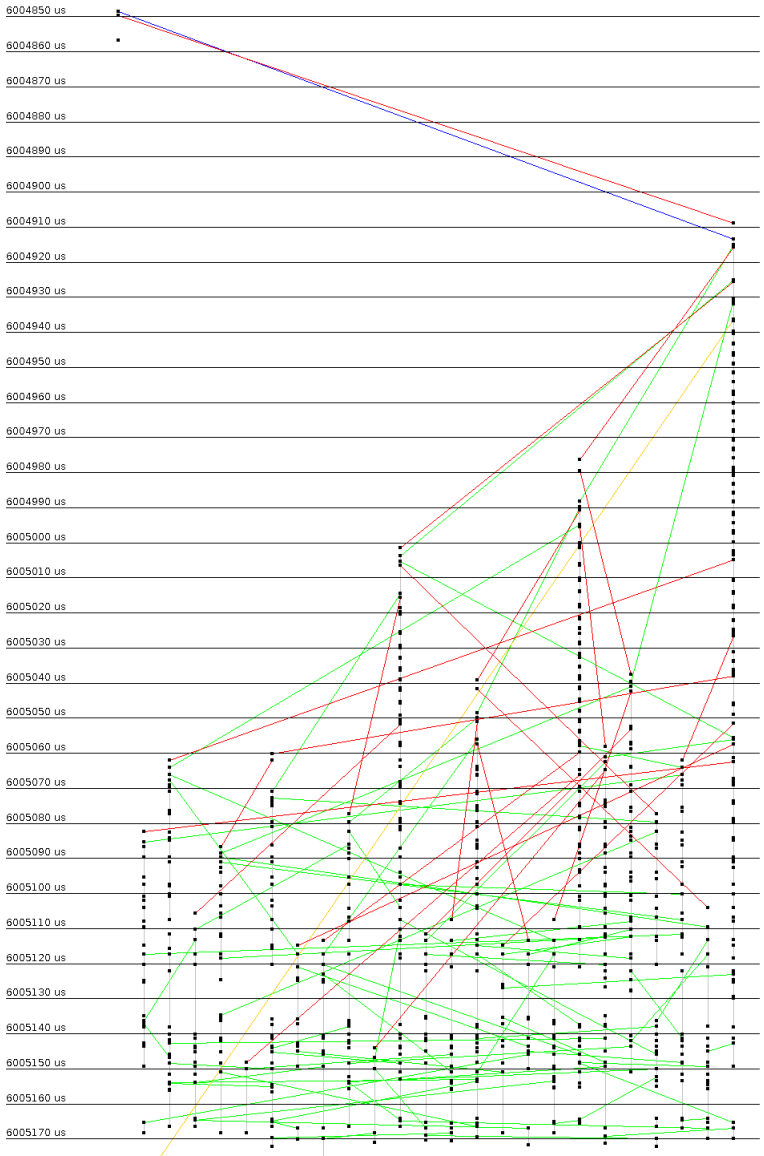
- сильно лучше на мелких задачах

$$\texttt{recursive}: 450 \pm 15\ \mu s$$
$$\texttt{completers}: 290 \pm 10\ \mu s$$

- так же быстры на крупных задачах

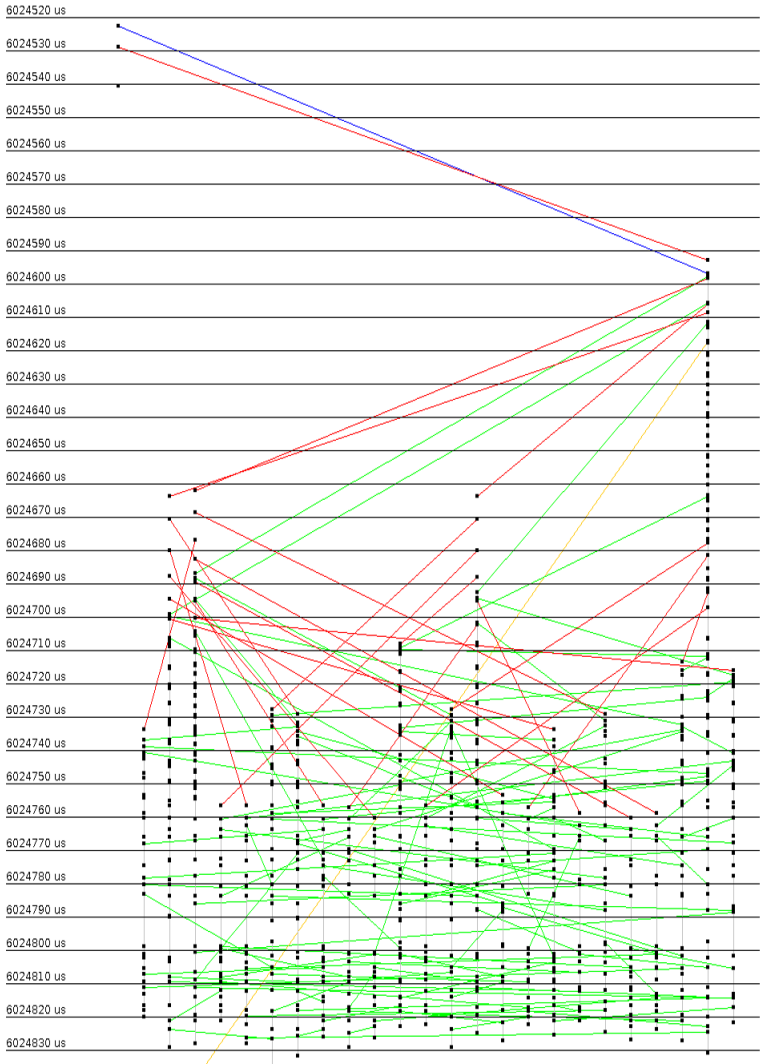
- сильно усложняют логику сборки результатов (упражнение читателю)

Over-signalling: проблема



- разбудить весь пул – большая проблема на мелких задачах
- характерное время для одного потока $\sim 50\mu s$
- для N потоков может съесть кучу времени $\sim 50\ log(N)\ \mu s$

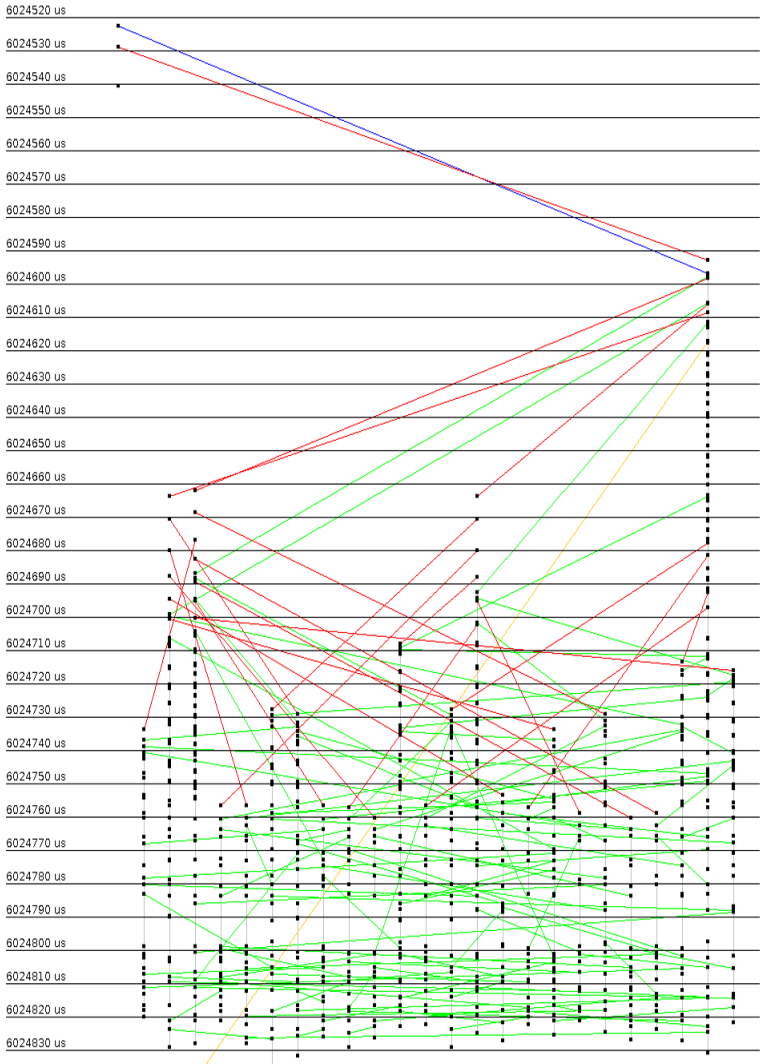
Over-signalling: решение



$\sim 50\ log(N)\ \mu s$

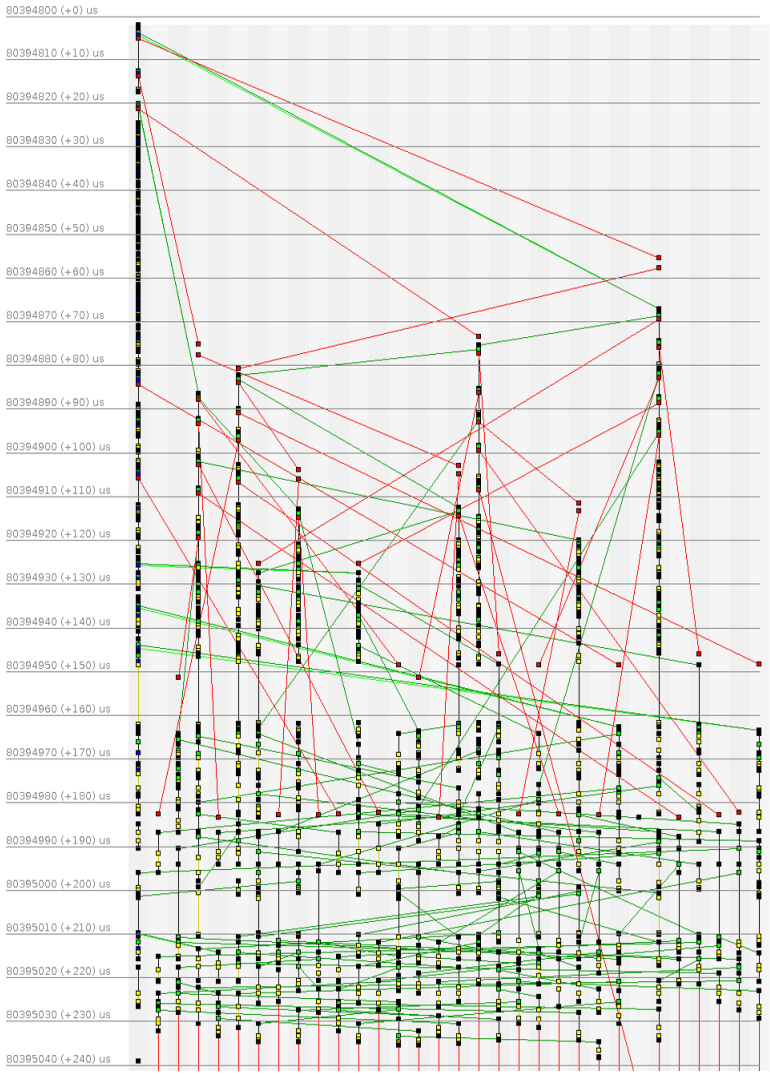
- константа от нас не зависит, только OS
- можно только базу логарифма уменьшить
- «полундра!»-mode: будить всех и вся при всяком удобном случае

Common Pool: проблема



- в конце концов: первый hand-off в пул тормозит всё
- характерное время от первого wakeup'а до полного параллелизма $\sim 200\mu s$

Common Pool: решение



Заставить submitter-ы работать на нас!

- ...они будят пул
- ...и вообще на нас работают

Выигрываем $100\mu s$ и больше!

Common Pool: реализация

- submitter-у нужно прикинуться `FJPThread`-ом
- где-то держать `WorkQueue`?
- занять `WorkQueue` у `ForkJoinPool`?
- у какого `ForkJoinPool`?

Common Pool: реализация

- submitter-у нужно прикинуться `FJPThread`-ом
- где-то держать `WorkQueue`?
- занять `WorkQueue` у `ForkJoinPool`?
- у какого `ForkJoinPool`?

Сделаем общесистемный пул по умолчанию
= FJP common pool.

Common Pool: API

```
public class MyTask
    extends RecursiveAction { ... }

// submit to the pool.
pool.submit(new MyTask());

// implicitly runs in caller.
// may submit to common pool.
new MyTask().invoke();

// fire and forget
new MyTask().fork();
```


Common Pool: API

```
public class MyTask
    extends RecursiveAction { ... }

// submit to the pool.
pool.submit(new MyTask());

// implicitly runs in caller.
// may submit to common pool.
new MyTask().invoke();

// fire and forget
new MyTask().fork();
```

Fork/Join – неиллюзорная часть платформы.

ORACLE

Для проклятых

Для проклятых: False Sharing

- `WorkQueue` чувствительны к memory layout
- false sharing $\Rightarrow$ 1-10x slower
- особенно из-за `WorkQueue[]` $\leftarrow$ лягут рядом

Для проклятых: False Sharing

- `WorkQueue` чувствительны к memory layout
- false sharing $\Rightarrow$ 1-10x slower
- особенно из-за `WorkQueue[]` $\leftarrow$ лягут рядом

Нас это задолбало, и мы форсировали работы по `@Contended`:

```
@sun.misc.Contended
class MyClass {
    private int myIsolatedField;
}
```

Для проклятых: TLR

- FJP нуждается в быстром ThreadLocalRandom
- `TLR.current()` ~ `ThreadLocal.get()`
- Для критичного кода это очень плохо

Для проклятых: TLR

- FJP нуждается в быстром ThreadLocalRandom
- `TLR.current()` ~ `ThreadLocal.get()`
- Для критичного кода это очень плохо

Поэтому мы утащили TLR в Thread:

```
class Thread {
    long threadLocalRandomSeed;
    int threadLocalRandomProbe;
    int threadLocalRandomSecondarySeed;
}
```


Для проклятых: new intrinsics

- из соображений атомарности:

```
class ForkJoinPool {
    // packed state
    volatile long state;
    ...
}
```

- часто нужно делать апдейт конкретных битов
- приходится делать CAS loop

Для проклятых: new intrinsics

- из соображений атомарности:

```
class ForkJoinPool {
    // packed state
    volatile long state;
    ...
}
```

- часто нужно делать апдейт конкретных битов
- приходится делать CAS loop

Поэтому у нас теперь есть
`Unsafe.getAndAddLong()` сотоварищи.

ORACLE

: Параллелизм в моей Джаве

Fork/Join:

- Путь к параллелизму на платформе
- Всегда есть, всегда доступен, всегда готов
- Жёстко затюнен и тюнится
- Поддерживается платформой на всех уровнях
- Множество подсистем уже использует

Миллионы леммингов не могут ошибаться:
JDK 8, Scala/Akka, GPars, Clojure, X10, Fortress

: Ссылки

- JSR 166 Interest:
  `http://g.oswego.edu/dl/concurrency-interest/`

- java-forkjoin-trace:
  `https://github.com/shipilev/java-forkjoin-trace/`

