

# Phaser and StampedLock Concurrency Synchronizers

**Dr Heinz M. Kabutz**  
**[www.javaspecialists.eu](http://www.javaspecialists.eu)**

Last updated 2014-12-17



Javaspecialists.eu  
java training

© 2013-2014 Heinz Kabutz – All Rights Reserved

# Heinz Kabutz

## ● Brief Biography

- German from Cape Town, now lives in Chania on island of Crete
- The Java Specialists' Newsletter
  - 135 countries
- Java Champion since 2005
- JavaOne Rock Star Speaker



Join us here for an advanced Java course!



# Why Synchronizers?



## Why Synchronizers?

- **Synchronizers keep shared mutable state consistent**
  - Don't need if we can make state immutable or unshared
- **But many applications need large amounts of state**
  - Immutable would stress the garbage collector
  - Unshared would stress the memory volume
- **Some applications have hash maps of hundreds of GB!**

## Coarse Grained Locking

- **Overly coarse-grained locking means the CPUs are starved for work**
  - Only one core is busy at a time
- **Took 51 seconds to complete**



## Fine Grained Locking

- **"Synchronized" causes *voluntary context switches***
  - Thread cannot get the lock, so it is parked
    - Gives up its allocated time quantum
- **Took 745 seconds to complete**



- **It appears that system time is 50% of the total time**
  - So should this not have taken the same elapsed time as before?

## Independent Tasks With No Locking

- **Instead of shared mutable state, every thread uses only local data and in the end we merge the results**
- **Took 28 seconds to complete with 100% utilization**





## Nonblocking Algorithms

- **Lock-based algorithms can cause scalability issues**
  - If a thread is holding a lock and is swapped out, no one can progress
- **Definitions of types of algorithms**
  - *Nonblocking*: failure or suspension of one thread, cannot cause another thread to fail or be suspended
  - *Lock-free*: at each step, *some* thread can make progress

# Phaser

**New synchronizer compatible with Fork/Join**



## Synchronizers - Structural Properties

- **Encapsulate state that determines whether arriving threads should be allowed to pass or forced to wait**
- **Provide methods to manipulate that state**
- **Provide methods to wait (efficiently) for the synchronizer to enter a desired state**

## CountDownLatch

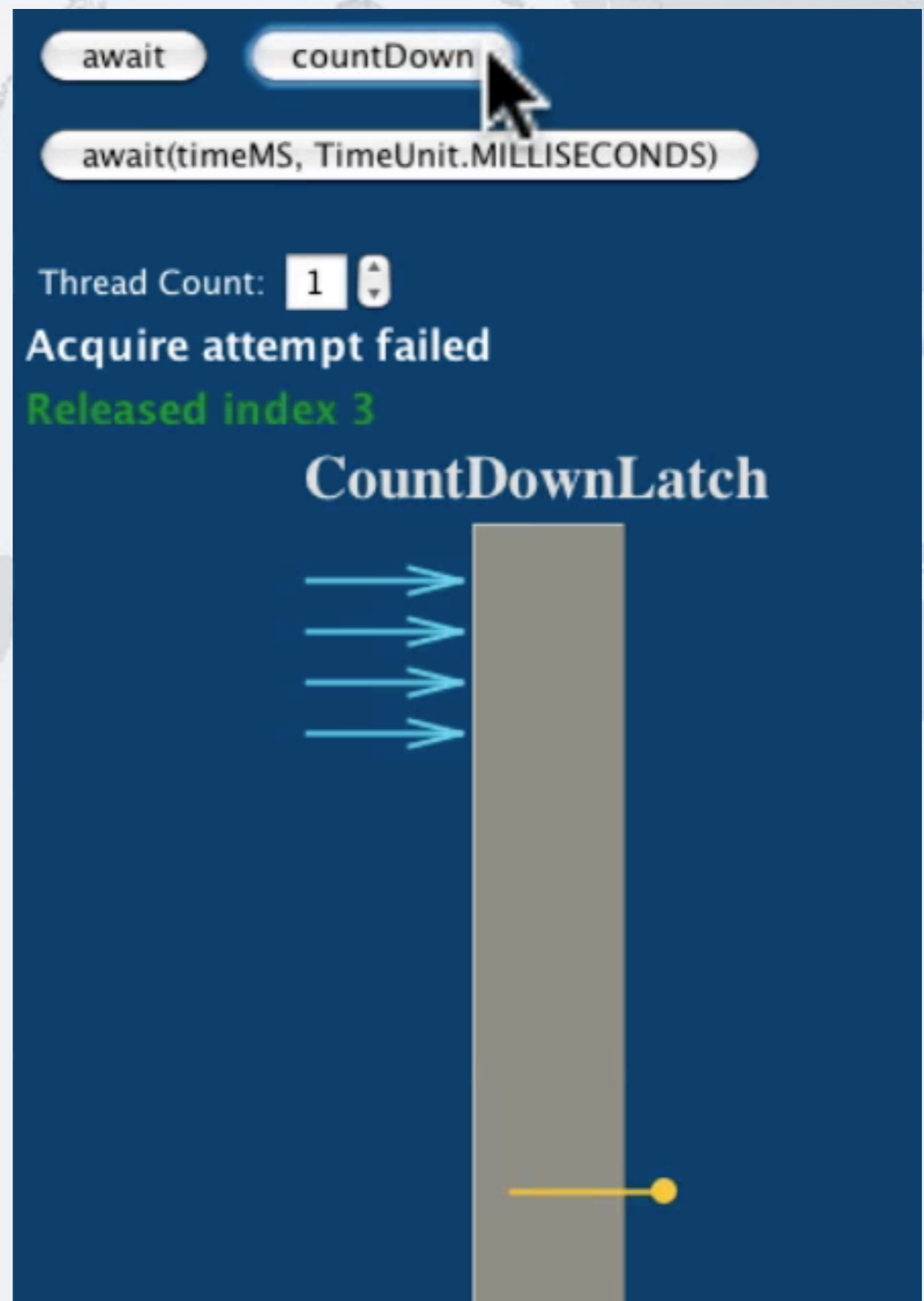
- **A latch is a synchronizer that blocks until it reaches its terminal state, at which point it allows all threads to pass**
- **Once it reaches the terminal state it remains open forever**
- **Ensures that activities do not start until all of the dependent activities have completed. For example:**
  - All resources have been initialized
  - All services have been started
  - All horses are at the gate

## Interface: CountdownLatch

```
public class CountdownLatch {  
    CountdownLatch(int count) Fixed number of  
initial permits  
  
    void await() throws InterruptedException  
    boolean await(long timeout, TimeUnit unit)  
        throws InterruptedException  
        A thread can wait for  
count to reach zero  
  
    void countDown() We can count down, but  
never up. No reset possible.  
}
```

## CountDownLatch

- **Concurrent Animation by Victor Grazi**
  - [www.jconcurrency.com](http://www.jconcurrency.com)
- **Threads are waiting until the count down latch is zero**
  - Then they immediately continue running



## Code sample: CountdownLatch

```
Service getService()  
    throws InterruptedException {  
    serviceCountDown.await();  
    return service;  
}  
  
void startDb() {  
    db = new Database();  
    db.start();  
    serviceCountDown.countDown();  
}  
  
void startMailServer() {  
    mail = new MailServer();  
    mail.start();  
    serviceCountDown.countDown();  
}
```

## CyclicBarrier

- **CyclicBarrier is similar to CountdownLatch**
  - Group of threads blocks until all have reached the same point
  - But then it is reset to the initial value
- **CyclicBarrier allows a fixed number of parties to rendezvous repeatedly at a barrier point**
- **CyclicBarrier also lets you pass a "barrier action" in the constructor**
  - The Runnable is executed when the barrier is successfully passed but before the blocked threads are released.



## Interface: CyclicBarrier

```
public class CyclicBarrier {  
    CyclicBarrier(int parties)  
    CyclicBarrier(int parties, Runnable barrierAction)
```

Fixed number of parties meet regularly

await() waits for all of the threads to arrive

```
int await() throws InterruptedException,  
           BrokenBarrierException  
int await(long timeout, TimeUnit unit)  
           throws InterruptedException,  
           BrokenBarrierException,  
           TimeoutException
```

```
void reset()  
}
```

If one of the parties times out, the barrier is broken and must be reset

## CyclicBarrier

- **Concurrent Animation**  
by Victor Grazi

```
await()
```

```
await(timeMS, TimeUnit.MILLISECONDS)
```

```
barrier.reset()
```

Thread Count:

barrier complete 1

Runnable hit

### CyclicBarrier



## Phasers

- **Mix of CyclicBarrier and CountdownLatch functionality**
  - But with more flexibility
- **Registration**
  - Number of parties *registered* may vary over time
    - Same as *count* in count down latch and *parties* in cyclic barrier
  - A party can register/deregister itself at any time
  - In contrast, both the other mechanisms have fixed number of parties
- **Compatible with Fork/Join framework**

## Interface: Phaser Registration Methods

```
public class Phaser {  
    Phaser(Phaser parent, int parties)
```

Initial parties - both parameters are optional

Phasers can be arranged in tree to reduce contention

```
int register()
```

```
int bulkRegister(int parties)
```

We can change the parties dynamically by calling register()

## Interface: Phaser Signal And Wait Methods

```
public class Phaser { (continued...)
```

```
int arrive()
```

```
int arriveAndDeregister()
```

Signal only

```
int awaitAdvance(int phase)
```

Wait only - default  
is to save interrupt

```
int awaitAdvanceInterruptibly(int phase[, timeout])  
throws InterruptedException
```

```
int arriveAndAwaitAdvance()
```

Signal and wait -  
also saves interrupt

## Interface: Phaser Action Method

```
public class Phaser { (continued...)  
    protected boolean onAdvance(  
        int phase, int registeredParties)  
    }  
}
```

Override onAdvance()  
to let phaser finish early

Bunch of lifecycle  
methods left out

## E.G. Coordinated Start Of Threads

- **We want a number of threads to start their work together**
  - Or as close together as possible, subject to OS scheduling
- **All threads wait for all others to be ready**
  - Once-off use
  - CountdownLatch or Phaser

## CountDownLatch: Waiting for threads to start

```
static void runTasks(List<Runnable> tasks)
    throws InterruptedException {
    int size = tasks.size() + 1;
    final CountDownLatch latch = new CountDownLatch(size);
    for (final Runnable task : tasks) {
        new Thread() {
            public void run() {
                try {
                    latch.countDown();
                    latch.await();
                    System.out.println("Running " + task);
                    task.run();
                } catch (InterruptedException e) { /* returning */ }
            }
        }.start();
        Thread.sleep(1000);
    }
    latch.countDown();
}
```



## CountDownLatch: Dealing with Interruptions

- **"Saving" interruptions until we can deal with them is a lot of work with CountDownLatch**

```
public void run() {
    latch.countDown();
    boolean wasInterrupted = false;
    while (true) {
        try {
            latch.await();
            break;
        } catch (InterruptedException e) {
            wasInterrupted = true;
        }
    }
    if (wasInterrupted) Thread.currentThread().interrupt();
    System.out.println("Running: " + task);
    task.run();
}
```

## Phaser: Waiting For Threads To Start

- The code for Phaser is simpler and more intuitive

```
static void runTasks(List<Runnable> tasks)
    throws InterruptedException {
    final Phaser phaser = new Phaser(1); // we register ourselves
    for (final Runnable task : tasks) {
        phaser.register(); // and we register all our new threads
        new Thread() {
            public void run() {
                phaser.arriveAndAwaitAdvance();
                System.out.println("Running: " + task);
                task.run();
            }
        }.start();
        Thread.sleep(1000);
    }
    phaser.arriveAndDeregister(); // we let the main thread arrive
}
```

**phaser.arrive()** and **phaser.arriveAndAwaitAdvance()** also work

## Waiting For A Set Number Of Phases

- The `CyclicBarrier` does not know how many times we have passed through
- The `Phaser` remembers the "phase" we are in
  - If we go past `Integer.MAX_VALUE`, it resets to zero
- We do this by subclassing `Phaser` and overriding `onAdvance()`

```
private void addButtons(int buttons, final int blinks) {
    final Phaser phaser = new Phaser(buttons) {
        protected boolean onAdvance(
            int phase, int registeredParties) {
            return phase >= blinks - 1 ||
                registeredParties == 0;
        }
    };
    // ...
}
```

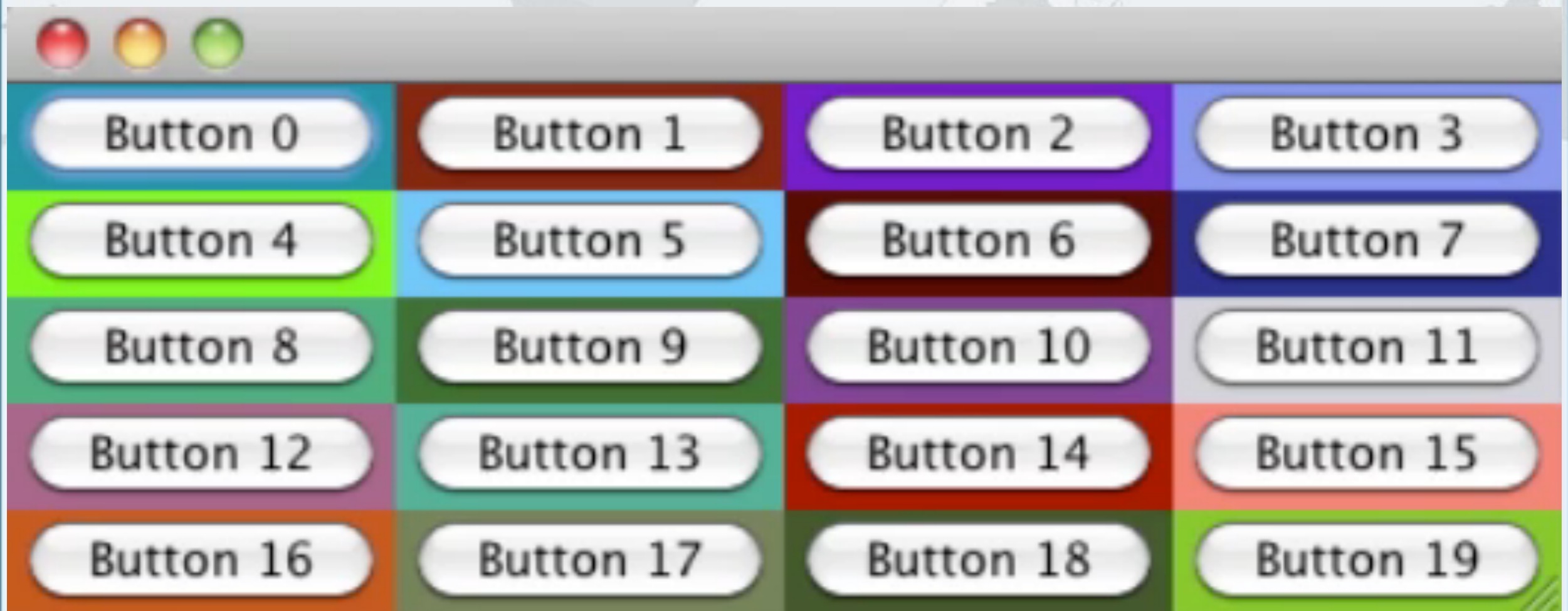
## Setting The Buttons A Random Color

- We carry on changing color until the phaser is terminated

```
new Thread() {
    public void run() {
        Random rand = ThreadLocalRandom.current();
        try {
            do {
                Color newColor = new Color(rand.nextInt());
                changeColor(comp, newColor); // sets it with the EDT
                Thread.sleep(rand.nextInt(500, 3000));
                changeColor(comp, defaultColor);
                Toolkit.getDefaultToolkit().beep();
                Thread.sleep(2000);
                phaser.arriveAndAwaitAdvance();
            } while (!phaser.isTerminated());
        } catch (InterruptedException e) { return; }
    }
}.start();
```

## Sample Run With Phaser

- **Running with 20 buttons and 3 phases**
  - Note, all the phases start at the same time for the 20 threads, but each phase ends when the color is reset to the original
  - With `CyclicBarrier`, we would have had to count the phases ourselves



## Tiered Phasers

- **Phasers can be arranged in a tree structure to reduce contention**
- **It is a bit complicated to understand (at least for me)**
  - Parent does not know what children it has
  - When a child is added, parent # parties increases by 1
    - If child's registered parties  $> 0$
  - Once child arrived parties  $== 0$ , one party automatically arrives at parent
  - If we use `arriveAndAwaitAdvance()`, we have to wait until all the parties in the whole tree have arrived
    - Thus the parties in the current phaser have to all arrive and in the parent

## Tiered Phasers

- **When a child phaser has non-zero parties, then the parent parties are incremented**

```
Phaser root = new Phaser(3);
Phaser c1 = new Phaser(root, 4);
Phaser c2 = new Phaser(root, 5);
Phaser c3 = new Phaser(c2, 0);
System.out.println(root);
System.out.println(c1);
System.out.println(c2);
System.out.println(c3);
```

- **outputs**

```
j.u.c.Phaser [phase = 0 parties = 5 arrived = 0] (root)
j.u.c.Phaser [phase = 0 parties = 4 arrived = 0] (c1)
j.u.c.Phaser [phase = 0 parties = 5 arrived = 0] (c2)
j.u.c.Phaser [phase = 0 parties = 0 arrived = 0] (c3)
```

# Phaser "Root" Is Created With 3 Parties



root  
parties = 3

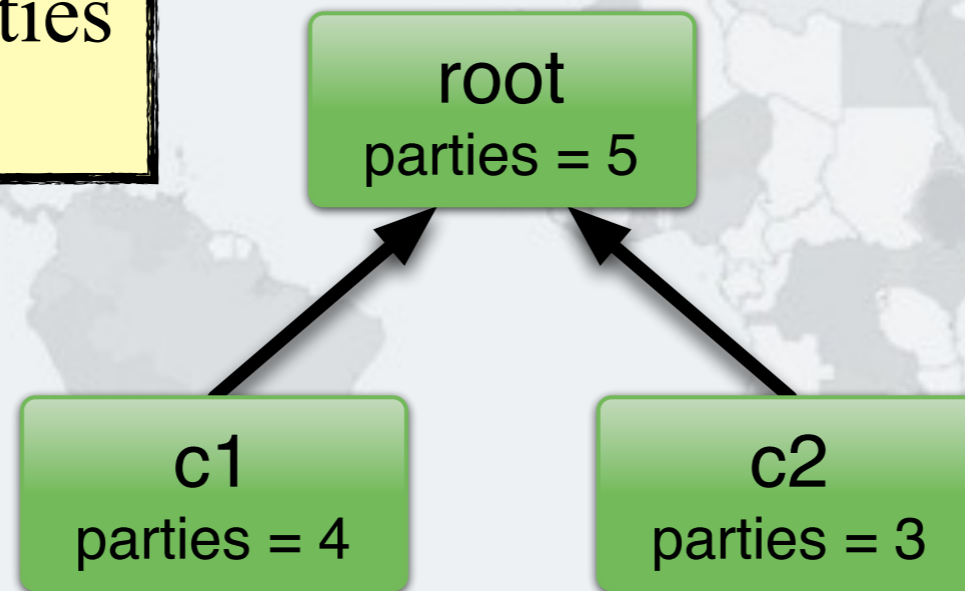


# Phaser "C1" Is Created With 4 Parties

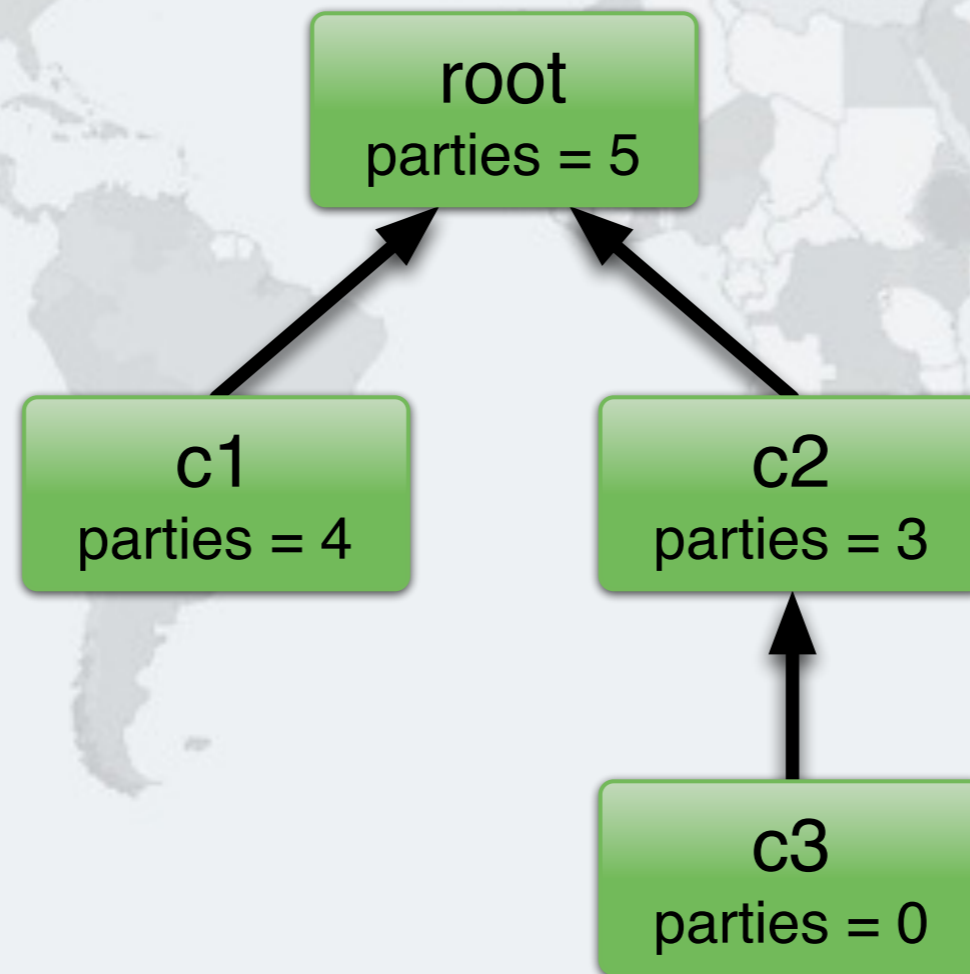


## Phaser "C2" Is Created With 3 Parties

Again increases parties  
in "root" phaser



# Phaser "C3" Is Created With 0 Parties



Does **not** increase parties in "c2" phaser, because c3's parties == 0

## Only Synchronizer Compatible With Fork/Join

- [JavaDoc] **Phasers** may also be used by tasks executing in a **ForkJoinPool** which will ensure sufficient parallelism to execute tasks when others are blocked waiting for a phase to advance.
- **Fork/Join Pools** do not have an upper limit on threads
  - They have a *parallelism level* and the FJ Pool will try to have at least that many *active* threads to prevent starvation
  - If one of the active threads is paused waiting for a phaser, another is simply started to maintain required parallelism
    - No other wait would do that
      - `Condition.await()`, `wait()`, `Semaphore.acquire()`, `CountDownLatch.await()`, etc.

```
public class ForkJoinPhaser {
    public static void main(String[] args) {
        ForkJoinPool fjp = new ForkJoinPool();
        fjp.invoke(new PhasedAction(100, new Phaser(100)));
        System.out.println(fjp);
    }
    private static class PhasedAction extends RecursiveAction {
        private final int phases;
        private final Phaser ph;
        private PhasedAction(int phases, Phaser ph) {
            this.phases = phases; this.ph = ph;
        }
        protected void compute() {
            if (phases == 1) {
                System.out.printf("wait: %s%n", Thread.currentThread());
                ph.arriveAndAwaitAdvance();
                System.out.printf("done: %s%n", Thread.currentThread());
            } else {
                int left = phases / 2;
                int right = phases - left;
                invokeAll(new PhasedAction(left, ph),
                    new PhasedAction(right, ph));
            }
        }
    }
}
```

## Threads Are Created To Maintain Parallelism

```
done: Thread[ForkJoinPool-1-worker-227,5,main]
done: Thread[ForkJoinPool-1-worker-239,5,main]
done: Thread[ForkJoinPool-1-worker-197,5,main]
done: Thread[ForkJoinPool-1-worker-209,5,main]
done: Thread[ForkJoinPool-1-worker-253,5,main]
done: Thread[ForkJoinPool-1-worker-139,5,main]
done: Thread[ForkJoinPool-1-worker-167,5,main]
done: Thread[ForkJoinPool-1-worker-179,5,main]
done: Thread[ForkJoinPool-1-worker-207,5,main]
ForkJoinPool[
  Running,
  parallelism = 2,
  size = 100,
  active = 0, running = 0, steals = 100,
  tasks = 0, submissions = 0]
```

## Synchronizers Summary

- **CountDownLatch**
  - Makes threads wait until the latch has been counted down to zero
- **CyclicBarrier**
  - A barrier that is reset once it reaches zero
- **Phaser**
  - A flexible synchronizer in Java 7 to do latch and barrier semantics
    - With less code and better interrupt management

# StampedLock





## Motivation for StampedLock

- **Some constructs need a form of read/write lock**
- **ReentrantReadWriteLock can cause starvation (next slide)**
  - Plus it always uses pessimistic locking
- **StampedLock provides optimistic locking on reads**
  - Which can be converted easily to a pessimistic lock
- **Write locks are always pessimistic**
  - Also called *exclusive* locks

## Read-Write Locks Refresher

- **ReadWriteLock interface**
  - The `writeLock()` is *exclusive* - only one thread at a time
  - The `readLock()` is given to lots of threads at the same time
    - Much better when mostly reads are happening
  - Both locks are pessimistic

## Bank Account with ReentrantReadWriteLock

```
public class BankAccountWithReadWriteLock {
    private final ReadWriteLock lock =
        new ReentrantReadWriteLock();
    private double balance;
    public void deposit(double amount) {
        lock.writeLock().lock();
        try {
            balance = balance + amount;
        } finally {
            lock.writeLock().unlock();
        }
    }
    public double getBalance() {
        lock.readLock().lock();
        try {
            return balance;
        } finally {
            lock.readLock().unlock();
        }
    }
}
```

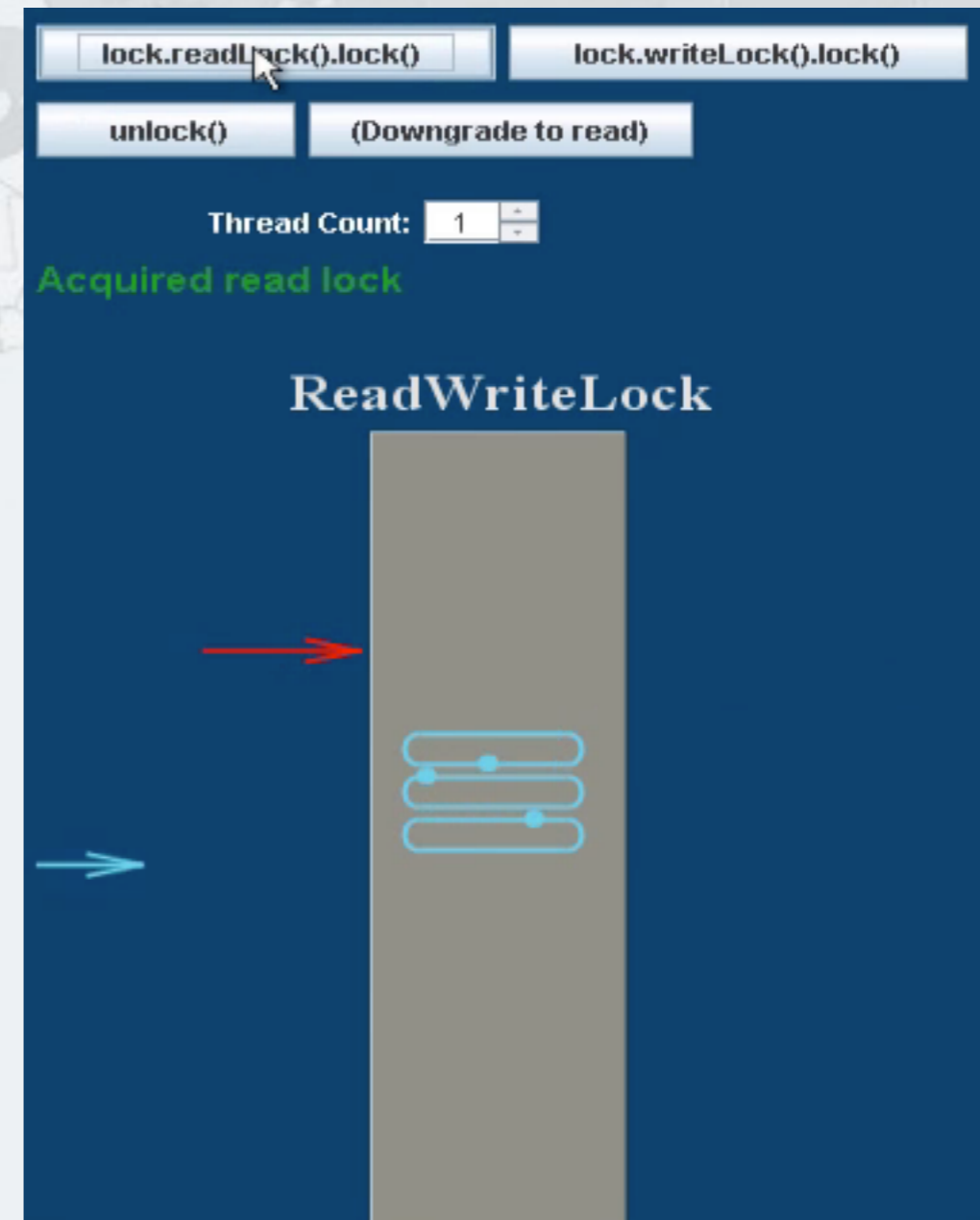
The cost overhead of the RWLock means we need at least 2000 instructions to benefit from the readLock() added throughput

## ReentrantReadWriteLock Starvation

- **When readers are given priority, then writers might never be able to complete (Java 5)**
- **But when writers are given priority, readers might be starved (Java 6)**
- **See <http://www.javaspecialists.eu/archive/Issue165.html>**

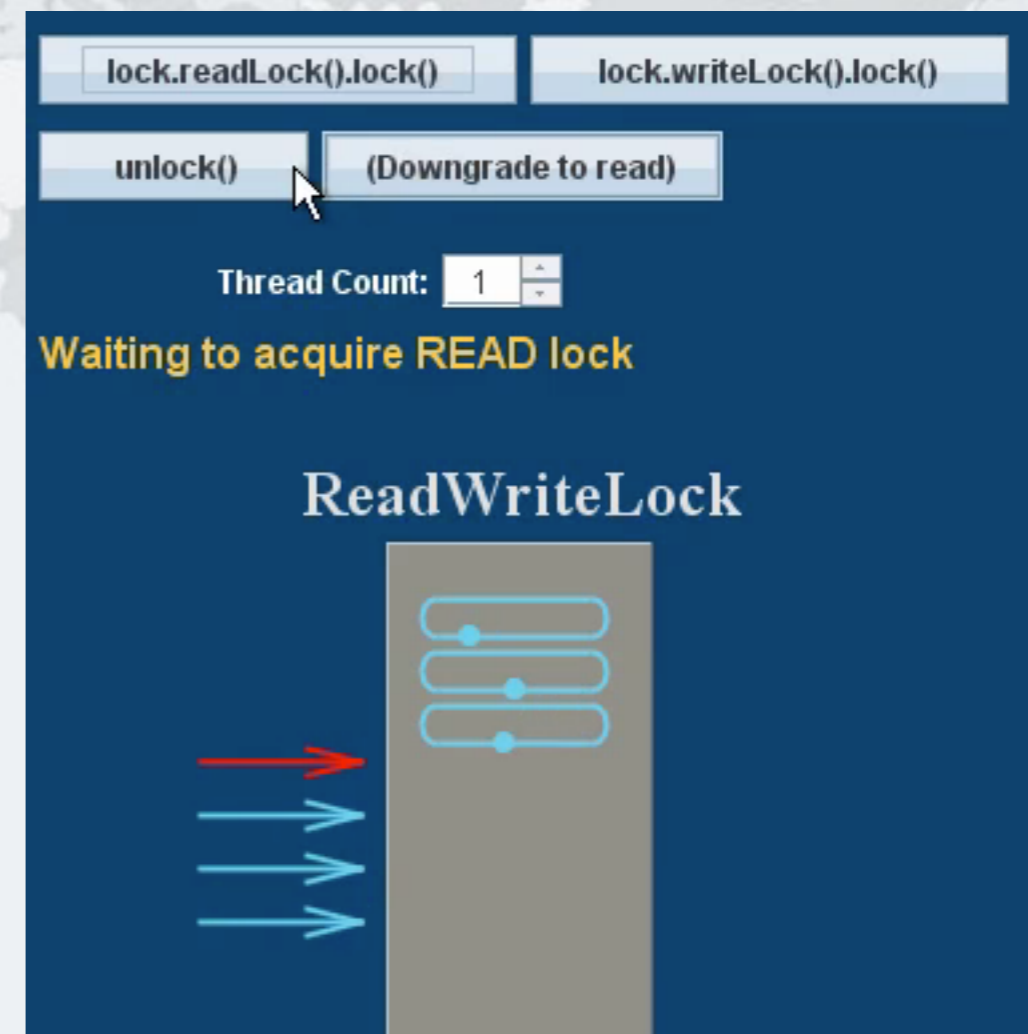
## Java 5 ReadWriteLock Starvation

- We first acquire some read locks
- We then acquire one write lock
- Despite write lock waiting, read locks are still issued
- If enough read locks are issued, write lock will never get a chance and the thread will be starved!



## ReadWriteLock in Java 6

- Java 6 changed the policy and now read locks have to wait until the write lock has been issued
- However, now the readers can be starved if we have a lot of writers



## Synchronized vs ReentrantLock

- **ReentrantReadWriteLock, ReentrantLock and synchronized locks have the same memory semantics**
- **However, synchronized is easier to write correctly**

```
synchronized(this)  
    // do operation  
}
```

```
rwlock.writeLock().lock();  
try {  
    // do operation  
} finally {  
    rwlock.writeLock().unlock();  
}
```

## Bad Try-Finally Blocks

- **Either no try-finally at all:**

```
rwlock.writeLock().lock();  
// do operation  
rwlock.writeLock().unlock();
```

- **Or the lock is locked inside the try block**

```
try {  
    rwlock.writeLock().lock();  
    // do operation  
} finally {  
    rwlock.writeLock().unlock();  
}
```

- **Or the unlock() call is forgotten in some places altogether!**

```
rwlock.writeLock().lock();  
// do operation  
// no unlock()
```



## Introducing StampedLock

### ● Pros

- Has *much* better performance than `ReentrantReadWriteLock`
- Latest versions do not suffer from starvation of writers

### ● Cons

- Idioms are more difficult to get right than with `ReadWriteLock`
- A small difference can make a big difference in performance

## Interface: StampedLock

```
public class StampedLock {  
    long writeLock()  
  
    long writeLockInterruptibly()  
        throws InterruptedException  
  
    long tryWriteLock()  
  
    long tryWriteLock(long time, TimeUnit unit)  
        throws InterruptedException  
  
    void unlockWrite(long stamp);  
    boolean tryUnlockWrite();  
  
    Lock asWriteLock();  
    long tryConvertToWriteLock(long stamp);  
}
```

Methods for managing exclusive write locks (pessimistic)

Methods return a number as a stamp. A value of zero means no write lock was granted

Stamp returned by writeLock()

Upgrade to a write lock

## Interface: StampedLock

```
public class StampedLock { (continued ...)  
    long readLock();  
  
    long readLockInterruptibly()  
        throws InterruptedException;  
  
    long tryReadLock();  
  
    long tryReadLock(long time, TimeUnit unit)  
        throws InterruptedException;  
  
    void unlockRead(long stamp);  
    boolean tryUnlockRead();  
  
    Lock asReadLock();  
    long tryConvertToReadLock(long stamp);
```

Pessimistic read is basically the same as the write lock

Optimistic reads to come ...

## Bank Account with StampedLock

```
public class BankAccountWithStampedLock {  
    private final StampedLock lock = new StampedLock();  
    private double balance;  
    public void deposit(double amount) {  
        long stamp = lock.writeLock();  
        try {  
            balance = balance + amount;  
        } finally {  
            lock.unlockWrite(stamp);  
        }  
    }  
    public double getBalance() {  
        long stamp = lock.readLock();  
        try {  
            return balance;  
        } finally {  
            lock.unlockRead(stamp);  
        }  
    }  
}
```

The StampedLock is a lot cheaper than ReentrantReadWriteLock

## Bank Account With Synchronized/Volatile

```
public class BankAccountWithVolatile {  
    private volatile double balance;  
  
    public synchronized void deposit(double amount) {  
        balance = balance + amount;  
    }  
  
    public double getBalance() {  
        return balance;  
    }  
}
```

"balance" needs to be volatile for two reasons:  
1.visibility and 2.it is a 64-bit value, so access is not necessarily atomic

Much easier!  
Works because there are no invariants across the fields.

## Example With Invariants Across Fields

- **Our Point class has x and y coordinates**
  - **We want to make sure that they always "belong together"**

```
public class MyPoint {
    private double x, y;
    private final StampedLock sl = new StampedLock();

    // method is modifying x and y, needs exclusive lock
    public void move(double deltaX, double deltaY) {
        long stamp = sl.writeLock();
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            sl.unlockWrite(stamp);
        }
    }
}
```

## Code Idiom For A Conditional State Change

```
public void changeStateIfEquals(oldState1, oldState2, ...
                               newState1, newState2, ...) {
    long stamp = sl.readLock();
    try {
        while (state1 == oldState1 && state2 == oldState2 ...) {
            long writeStamp = sl.tryConvertToWriteLock(stamp);
            if (writeStamp != 0L) {
                stamp = writeStamp;
                state1 = newState1; state2 = newState2; ...
                break;
            } else {
                sl.unlockRead(stamp);
                stamp = sl.writeLock();
            }
        }
    } finally {
        sl.unlock(stamp);
    }
}
```

## Code Idiom For A Conditional State Change

```
public void changeStateIfEquals(oldState1, oldState2, ...
                               newState1, newState2, ...) {
    long stamp = sl.readLock();
    try {
        while (state1 == oldState1 && state2 == oldState2 ...) {
            long writeStamp = sl.tryConvertToWriteLock(stamp);
            if (writeStamp != 0L) {
                stamp = writeStamp;
                state1 = newState1; state2 = newState2; ...
                break;
            } else {
                sl.unlockRead(stamp);
                stamp = sl.writeLock();
            }
        }
    } finally {
        sl.unlock(stamp);
    }
}
```

We get a pessimistic  
read lock



## Code Idiom For A Conditional State Change

```
public void changeStateIfEquals(oldState1, oldState2, ...
                               newState1, newState2, ...) {
    long stamp = sl.readLock();
    try {
        while (state1 == oldState1 && state2 == oldState2 ...) {
            long writeStamp = sl.tryConvertReadLockToWriteLock(stamp);
            if (writeStamp != 0L) {
                stamp = writeStamp;
                state1 = newState1; state2 = ...;
                break;
            } else {
                sl.unlockRead(stamp);
                stamp = sl.writeLock();
            }
        }
    } finally {
        sl.unlock(stamp);
    }
}
```

If the state is not the expected state, we unlock and exit method

Note: the general unlock() method can unlock both read and write locks

## Code Idiom For A Conditional State Change

```
public void changeStateIfEquals(oldState1, oldState2, ...
                               newState1, newState2, ...) {
    long stamp = sl.readLock();
    try {
        while (state1 == oldState1 && state2 == oldState2) {
            long writeStamp = sl.tryConvertToWriteLock(stamp);
            if (writeStamp != 0L) {
                stamp = writeStamp;
                state1 = newState1; state2 = newState2; ...
                break;
            } else {
                sl.unlockRead(stamp);
                stamp = sl.writeLock();
            }
        }
    } finally {
        sl.unlock(stamp);
    }
}
```

We try convert our read lock to a write lock

## Code Idiom For A Conditional State Change

```
public void changeStateIfEquals(oldState1, oldState2, ...
                               newState1, newState2, ...) {
    long stamp = sl.readLock();
    try {
        while (state1 == oldState1 && state2 == oldState2 ...) {
            long writeStamp = sl.tryConvertToWriteLock(stamp);
            if (writeStamp != 0L) {
                stamp = writeStamp;
                state1 = newState1; state2 = newState2; ...
                break;
            } else {
                sl.unlockRead(stamp);
                stamp = sl.writeLock();
            }
        }
    } finally {
        sl.unlock(stamp);
    }
}
```

If we are able to upgrade to a write lock (`ws != 0L`), we change the state and exit

## Code Idiom For A Conditional State Change

```
public void changeStateIfEquals(oldState1, oldState2, ...
                               newState1, newState2, ...) {
    long stamp = sl.readLock();
    try {
        while (state1 == oldState1 && state2 == oldState2 ...) {
            long writeStamp = sl.tryConvertToWriteLock(stamp);
            if (writeStamp != 0L) {
                stamp = writeStamp;
                state1 = newState1; state2 = newState2; ...
                break;
            } else {
                sl.unlockRead(stamp);
                stamp = sl.writeLock();
            }
        }
    } finally {
        sl.unlock(stamp);
    }
}
```

Else, we explicitly unlock the read lock and lock the write lock

And we try again

## Code Idiom For A Conditional State Change

```
public void changeStateIfEquals(oldState1, oldState2, ...
                               newState1, newState2, ...) {
    long stamp = sl.readLock();
    try {
        while (state1 == oldState1 && state2 == oldState2 ...) {
            long writeStamp = sl.tryConvertReadLockToWriteLock(stamp);
            if (writeStamp != 0L) {
                stamp = writeStamp;
                state1 = newState1; state2 =
                break;
            } else {
                sl.unlockRead(stamp);
                stamp = sl.writeLock();
            }
        }
    } finally {
        sl.unlock(stamp);
    }
}
```

If the state is not the expected state, we unlock and exit method

This could happen if between the unlockRead() and the writeLock() another thread changed the values

## Code Idiom For A Conditional State Change

```
public void changeStateIfEqual(
    long stamp = sl.readLock();
    try {
        while (state1 == oldState1 && state2 == oldState2 ...) {
            long writeStamp = sl.tryConvertToWriteLock(stamp);
            if (writeStamp != 0L) {
                stamp = writeStamp;
                state1 = newState1; state2 = newState2; ...
                break;
            } else {
                sl.unlockRead(stamp);
                stamp = sl.writeLock();
            }
        }
    } finally {
        sl.unlock(stamp);
    }
}
```

Because we hold the write lock,  
the `tryConvertToWriteLock()`  
method **will** succeed

We update the state and exit

## Applying The Code Idiom To Our Point Class

```
public void moveIfAt(double oldX, double oldY,  
                    double newX, double newY) {  
    long stamp = sl.readLock();  
    try {  
        while (x == oldX && y == oldY) {  
            long writeStamp = sl.tryConvertToWriteLock(stamp);  
            if (writeStamp != 0L) {  
                stamp = writeStamp;  
                x = newX; y = newY;  
                break;  
            } else {  
                sl.unlockRead(stamp);  
                stamp = sl.writeLock();  
            }  
        }  
    } finally {  
        sl.unlock(stamp);  
    }  
}
```

## Interface: StampedLock

```
public class StampedLock { (continued ...)  
    long tryOptimisticRead();
```

Try to get an optimistic read lock  
- might return zero

```
    boolean validate(long stamp);
```

checks whether a write lock was issued  
after the tryOptimisticRead() was called

Note: sequence validation requires stricter  
ordering than apply to normal volatile reads -  
a new explicit loadFence() was added

```
    long tryConvertToOptimisticRead(long stamp);
```



## Code Idiom For Optimistic Read

```
public double optimisticRead() {  
    long stamp = sl.tryOptimisticRead();  
    double currentState1 = state1,  
           currentState2 = state2, ... etc.;  
    if (!sl.validate(stamp)) {  
        stamp = sl.readLock();  
        try {  
            currentState1 = state1;  
            currentState2 = state2, ... etc.;  
        } finally {  
            sl.unlockRead(stamp);  
        }  
    }  
    return calculateSomething(  
        currentState1, currentState2);  
}
```

## Code Idiom For Optimistic Read

```
public double optimisticRead() {  
    long stamp = sl.tryOptimisticRead();  
    double currentState1 = state1,  
           currentState2 = state2, ... etc.;  
    if (!sl.validate(stamp)) {  
        stamp = sl.readLock();  
        try {  
            currentState1 = state1;  
            currentState2 = state2, ... etc.;  
        } finally {  
            sl.unlockRead(stamp);  
        }  
    }  
    return calculateSomething(  
        currentState1, currentState2);  
}
```

We get a stamp to use for the optimistic read

## Code Idiom For Optimistic Read

```
public double optimisticRead() {  
    long stamp = sl.tryOptimisticRead();  
    double currentState1 = state1,  
           currentState2 = state2, ... etc.;  
    if (!sl.validate(stamp)) {  
        stamp = sl.readLock();  
        try {  
            currentState1 = state1;  
            currentState2 = state2, ... etc.;  
        } finally {  
            sl.unlockRead(stamp);  
        }  
    }  
    return calculateSomething(  
        currentState1, currentState2);  
}
```

We read field values  
into local fields

## Code Idiom For Optimistic Read

```
public double optimisticRead() {
    long stamp = sl.tryOptimisticRead();
    double currentState1 = state1,
           currentState2 = state2, ... etc.;
    if (!sl.validate(stamp)) {
        stamp = sl.readLock();
        try {
            currentState1 = state1;
            currentState2 = state2, ...
        } finally {
            sl.unlockRead(stamp);
        }
    }
    return calculateSomething(
        currentState1, currentState2);
}
```

Next we validate that no write locks have been issued in the meanwhile

## Code Idiom For Optimistic Read

```
public double optimisticRead() {
    long stamp = sl.tryOptimisticRead();
    double currentState1 = state1,
           currentState2 = state2,
    if (!sl.validate(stamp)) {
        stamp = sl.readLock();
        try {
            currentState1 = state1;
            currentState2 = state2, ... etc.;
        } finally {
            sl.unlockRead(stamp);
        }
    }
    return calculateSomething(
        currentState1, currentState2);
}
```

If they have, then we don't know if our state is clean

Thus we acquire a pessimistic read lock and read the state into local fields

## Optimistic Read In Point Class

```
public double distanceFromOrigin() {  
    long stamp = sl.tryOptimisticRead();  
    double currentX = x, currentY = y;  
    if (!sl.validate(stamp)) {  
        stamp = sl.readLock();  
        try {  
            currentX = x;  
            currentY = y;  
        } finally {  
            sl.unlockRead(stamp);  
        }  
    }  
    return Math.hypot(currentX, currentY);  
}
```

Shorter code path in optimistic read leads to better read performance than with examples in JavaDoc

## Retrying Optimistic Read Can Improve Performance - Have Fun Tuning

```
private static final int RETRIES = 5;
public double distanceFromOrigin() {
    int localX, localY;
    out: {
        for (int i = 0; i < RETRIES; i++) {
            long stamp = sl.tryOptimisticRead();
            localX = x; localY = y;
            if (sl.validate(stamp)) break out;
        }
        long stamp = sl.readLock(); // pessimistic read
        try {
            localX = x; localY = y;
        } finally {
            sl.unlockRead(stamp);
        }
    }
    return Math.hypot(localX, localY);
}
```

## Performance of StampedLock vs RWLock

- **We researched ReentrantReadWriteLock in 2008**
  - Discovered serious starvation of *writers* (exclusive locks) in Java 5
  - And also some starvation of *readers* in Java 6
  - <http://www.javaspecialists.eu/archive/Issue165.html>
- **StampedLock released to concurrency-interest list Oct 12**
  - Worse *writer* starvation than in the ReentrantReadWriteLock
  - Missed signals could cause StampedLock to deadlock
- **Revision 1.35 released 28<sup>th</sup> Jan 2013**
  - Changed to use an explicit call to `loadFence()`
  - Writers do not get starved anymore
  - Works correctly



# Performance of StampedLock vs RWLock

- **In our test, we used**

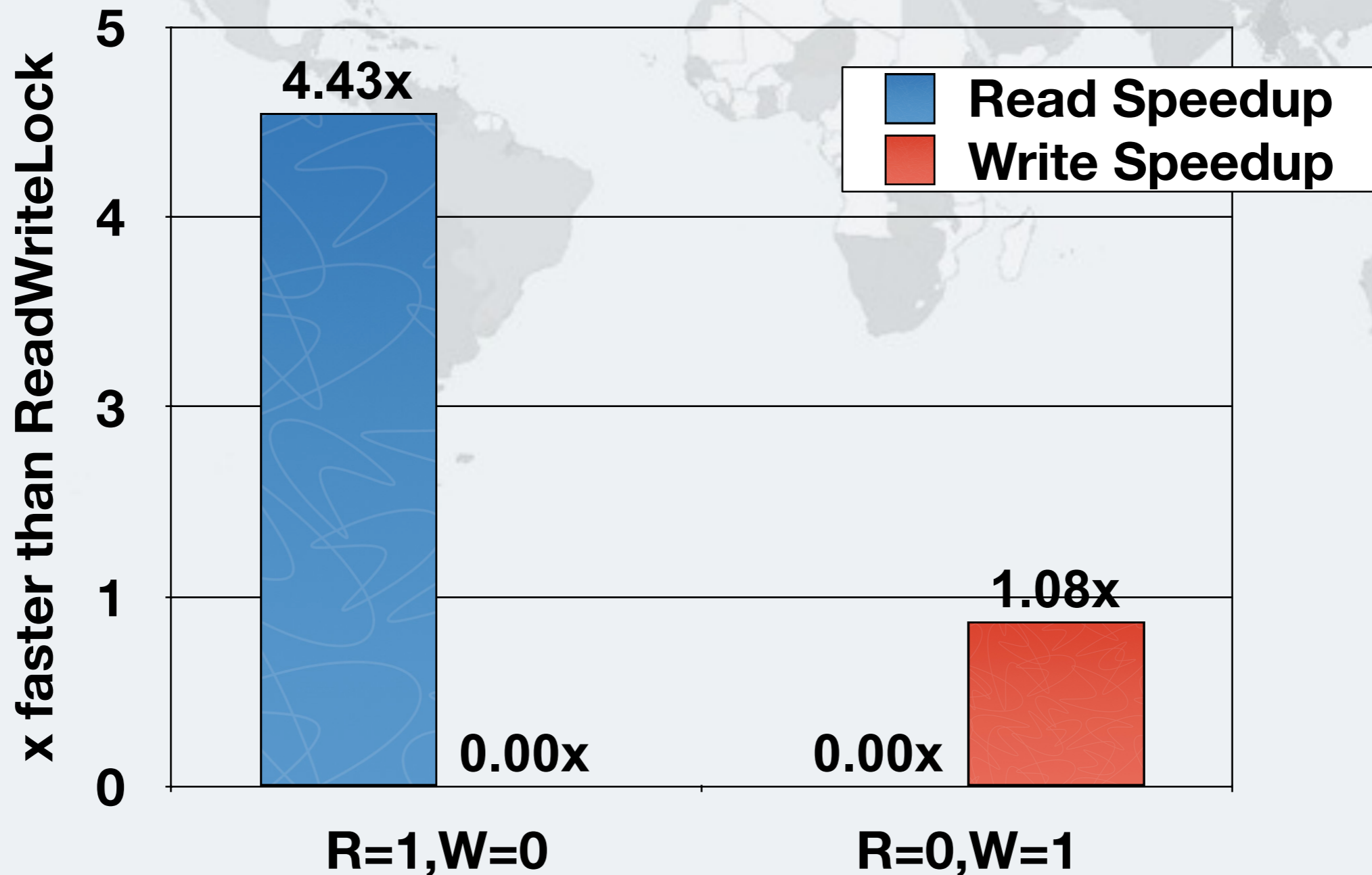
- **lambda-8-b75-linux-x64-28\_jan\_2013.tar.gz**
- **Two CPUs, 4 Cores each, no hyperthreading**
  - **2x4x1**
- **Ubuntu 9.10**
- **64-bit**
- **Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz**
  - **L1-Cache: 256KiB, internal write-through instruction**
  - **L2-Cache: 1MiB, internal write-through unified**
  - **L3-Cache: 8MiB, internal write-back unified**
- **JavaSpecialists.eu server**
  - **Never breaks a sweat delivering newsletters**

## Conversions To Pessimistic Reads

- **In our experiment, reads had to be converted to pessimistic reads less than 10% of the time**
  - And in most cases, less than 1%
- **This means the optimistic read worked most of the time**

# How much faster is StampedLock than ReentrantReadWriteLock?

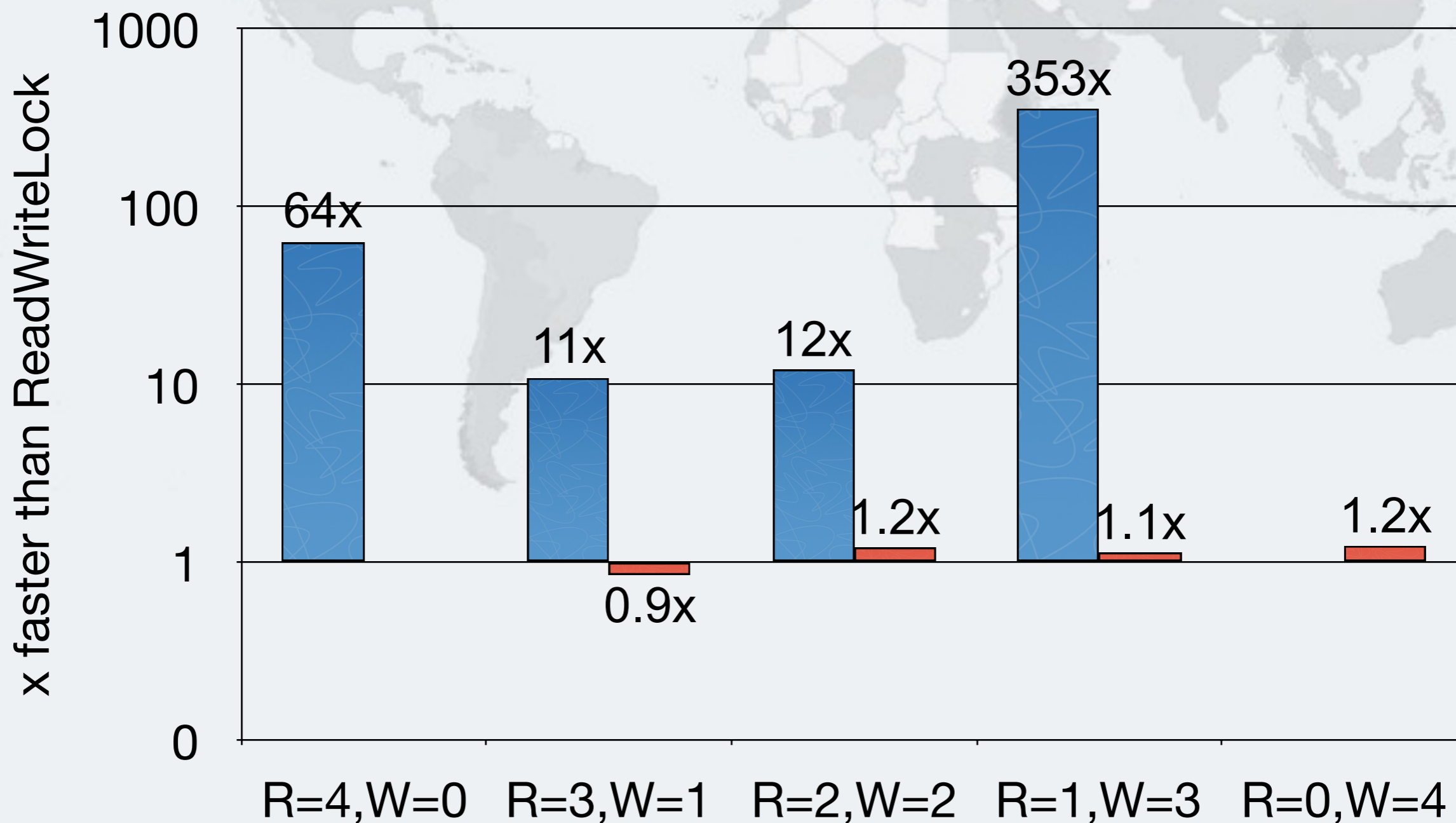
- With a single thread



# How much faster is StampedLock than ReentrantReadWriteLock?

- **With four threads**

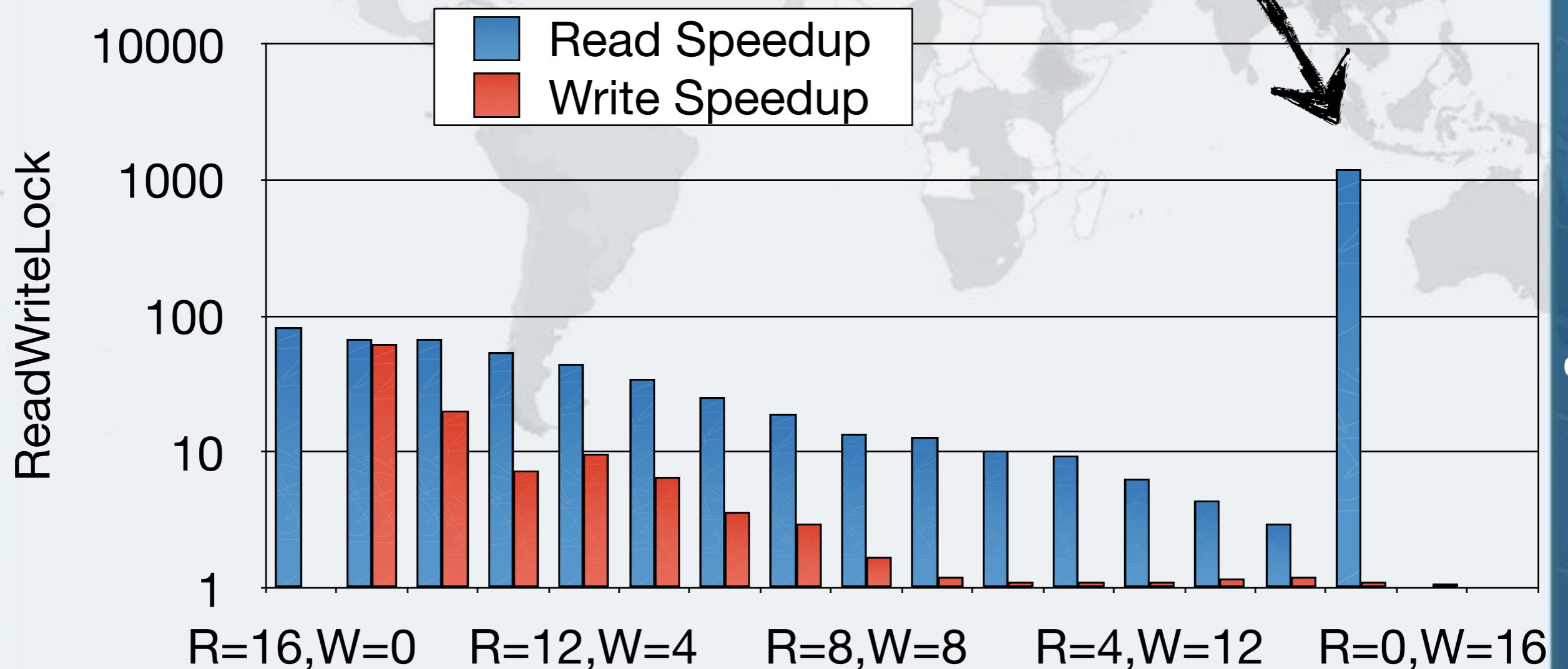
Read Speedup  
Write Speedup



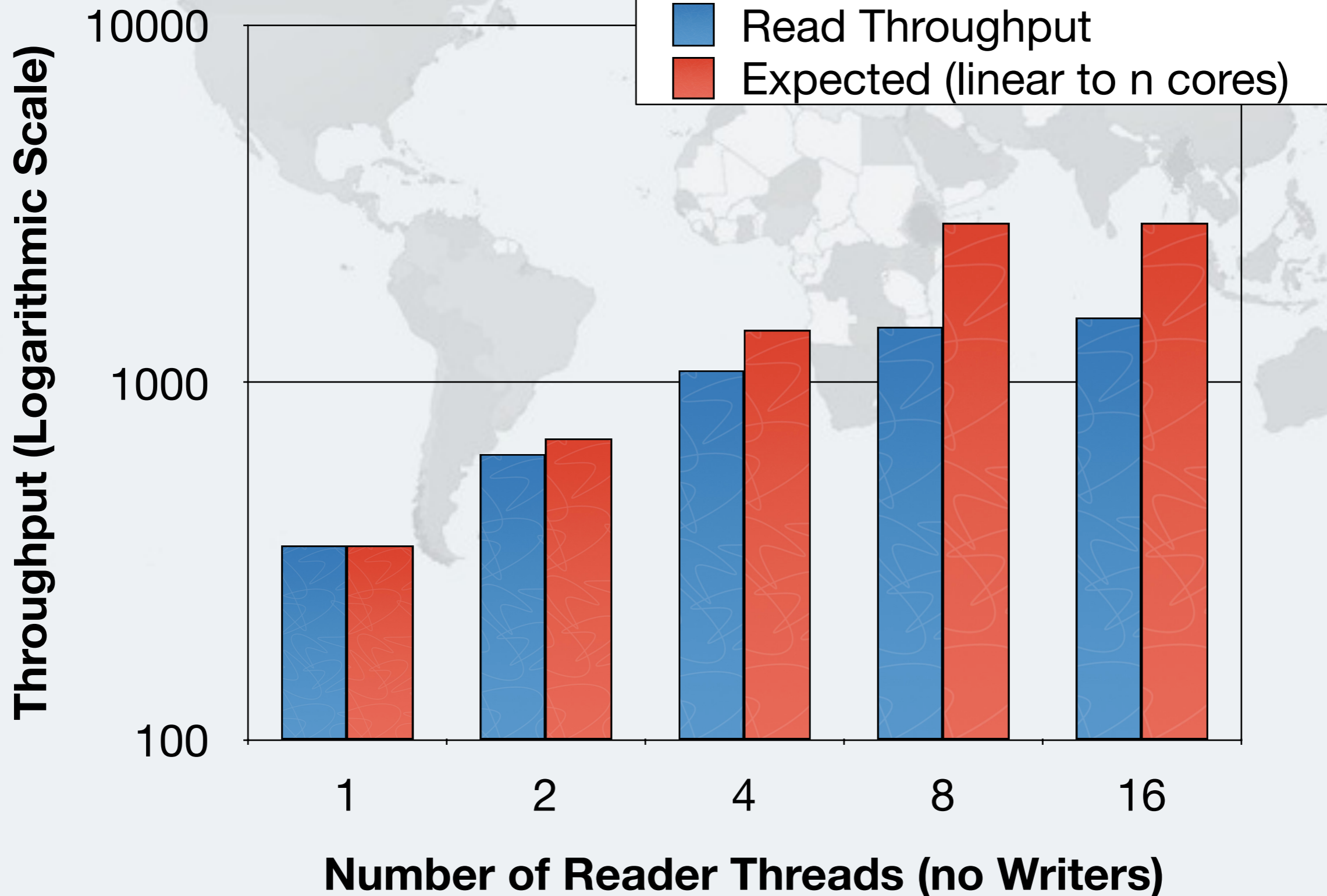
# How much faster is StampedLock than ReentrantReadWriteLock?

This demonstrates the starvation problem on readers in RWLock

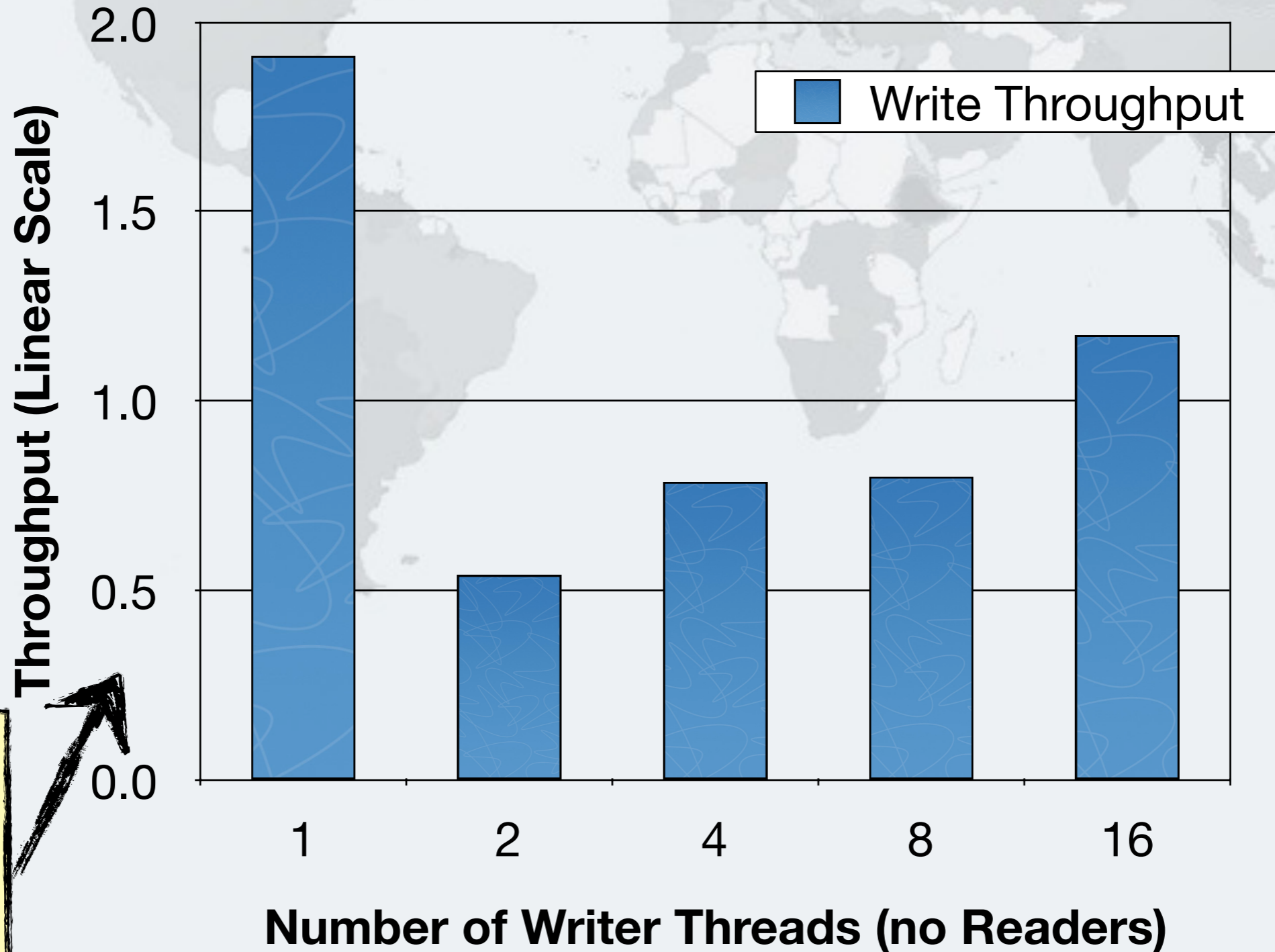
- With sixteen threads



# Reader Throughput with StampedLock



# Writer Throughput with StampedLock

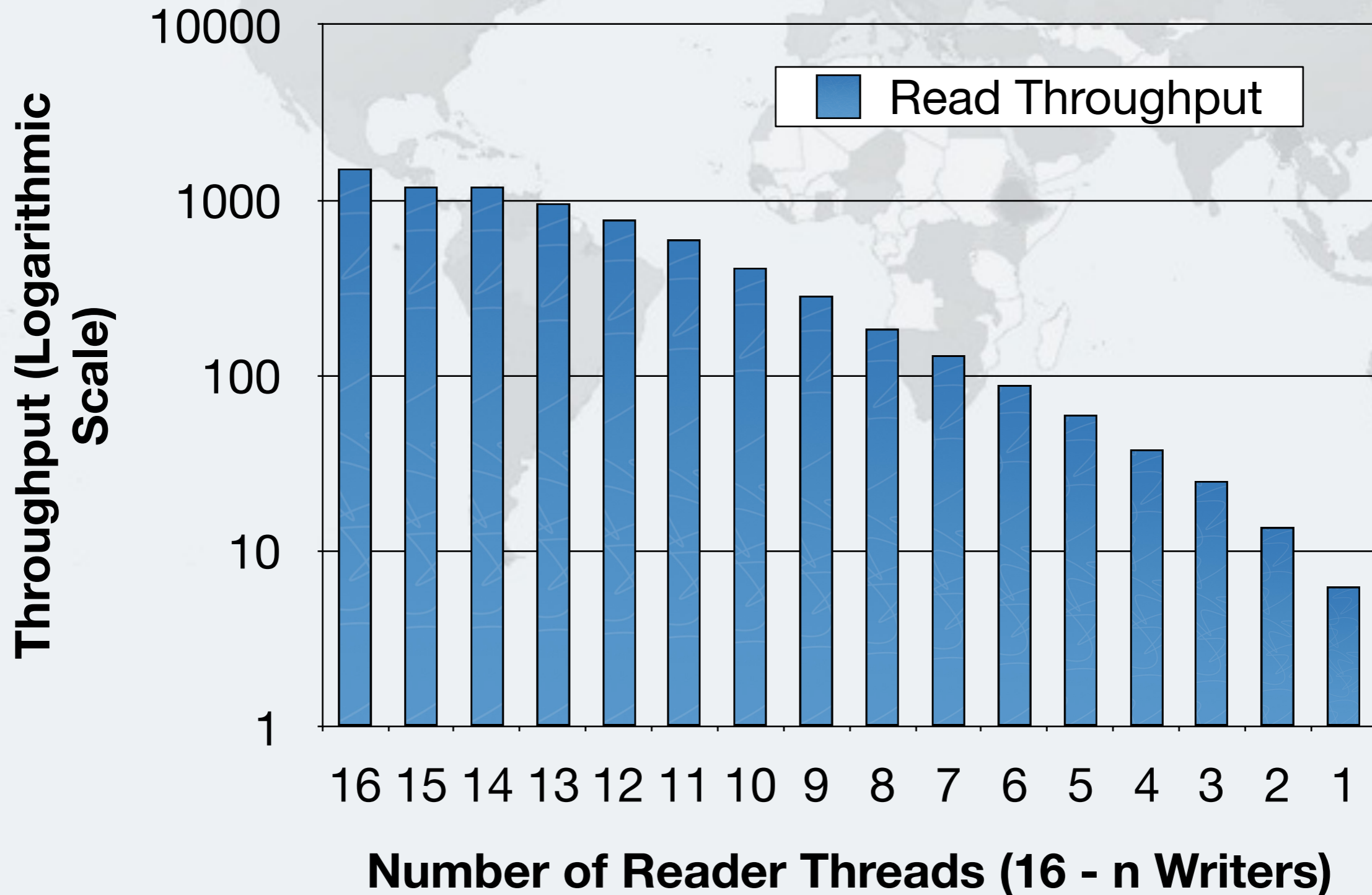


Throughput (Linear Scale)



Note:  
Linear  
Scale  
throughput

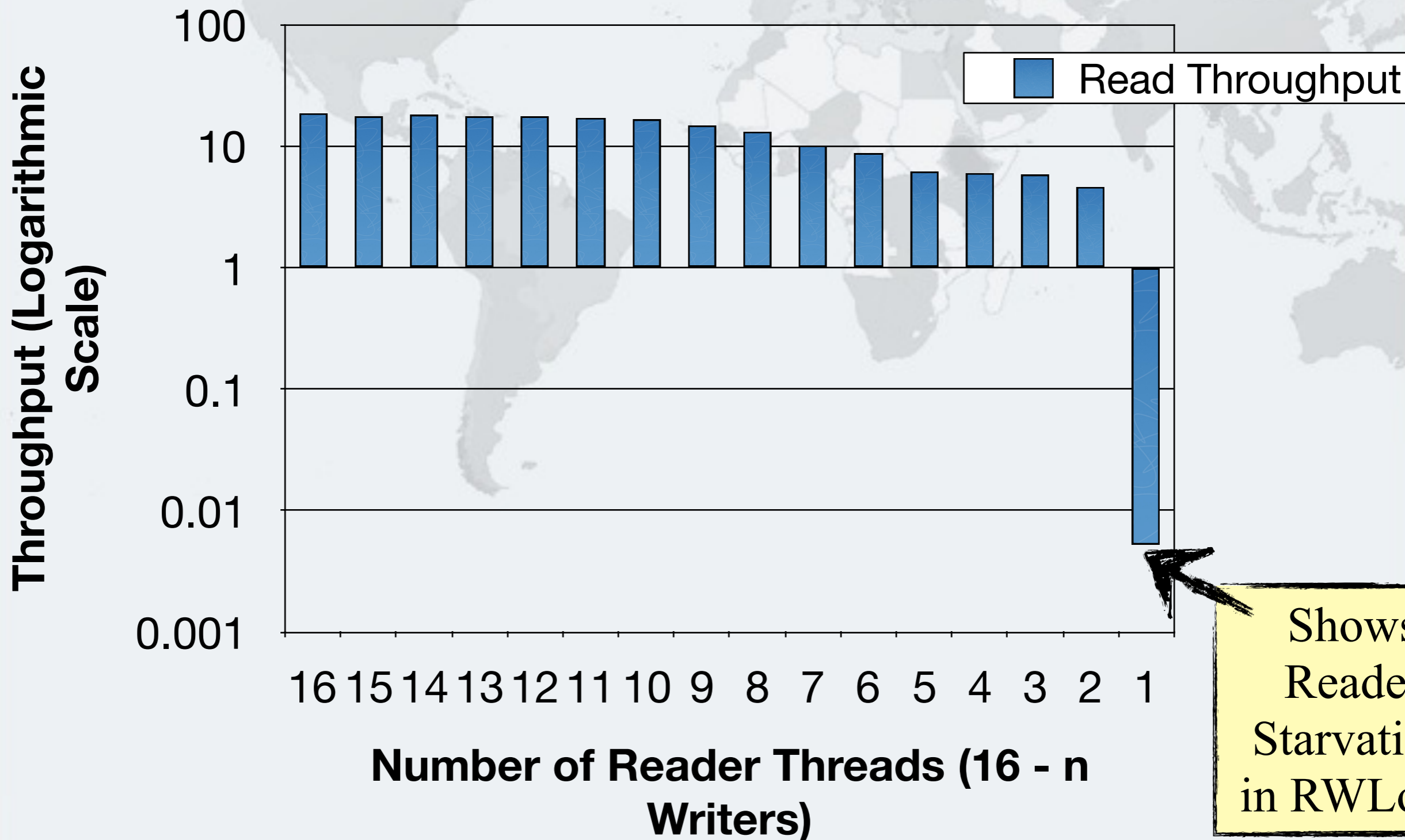
# Mixed Reader Throughput with StampedLock





# Mixed Reader Throughput with RWLock

## ReentrantReadWriteLock



Shows Reader Starvation in RWLock

## Conclusion Of Performance Analysis

- **StampedLock performed very well in all our tests**
  - Much faster than `ReentrantReadWriteLock`
- **Offers a way to do optimistic locking in Java**
- **Good idioms have a big impact on the performance**

# Conclusion

Where to next?



## JSR 166

- <http://gee.cs.oswego.edu/>
- **Concurrency-Interest mailing list**
  - Usage patterns and bug reports on Phaser and StampedLock are always welcome on the list

# Phaser and StampedLock Concurrency Synchronizers

[heinz@kabutz.net](mailto:heinz@kabutz.net)

[javaspecialists.eu](http://javaspecialists.eu)



Javaspecialists.eu  
java training