

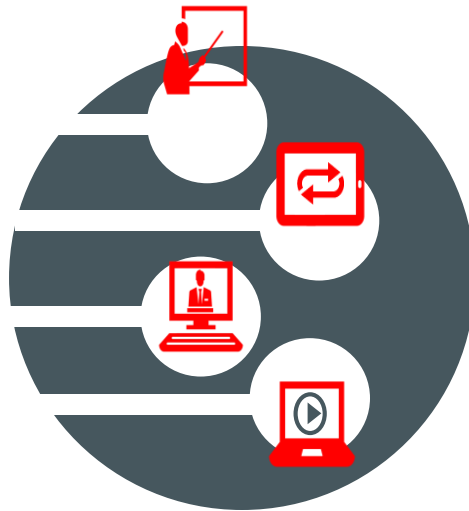


Keep Learning with Oracle University

ORACLE®

UNIVERSITY

Classroom Training
Learning
Subscription
Live Virtual Class
Training On
Demand



Cloud
Technology
Applications
Industries



education.oracle.com

Session Surveys

Help us help you!!

- Oracle would like to invite you to take a moment to give us your session feedback. Your feedback will help us to improve your conference.
- Please be sure to add your feedback for your attended sessions by using the Mobile Survey or in Schedule Builder.

Java Connection Pool Performance and Scalability with Wait-Free Programming

Yuri Dolgov

Principal Member of Technical Staff

Oracle Corporation

Java Database Connectivity development group

October 25, 2015



Safe Harbor Statement

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.

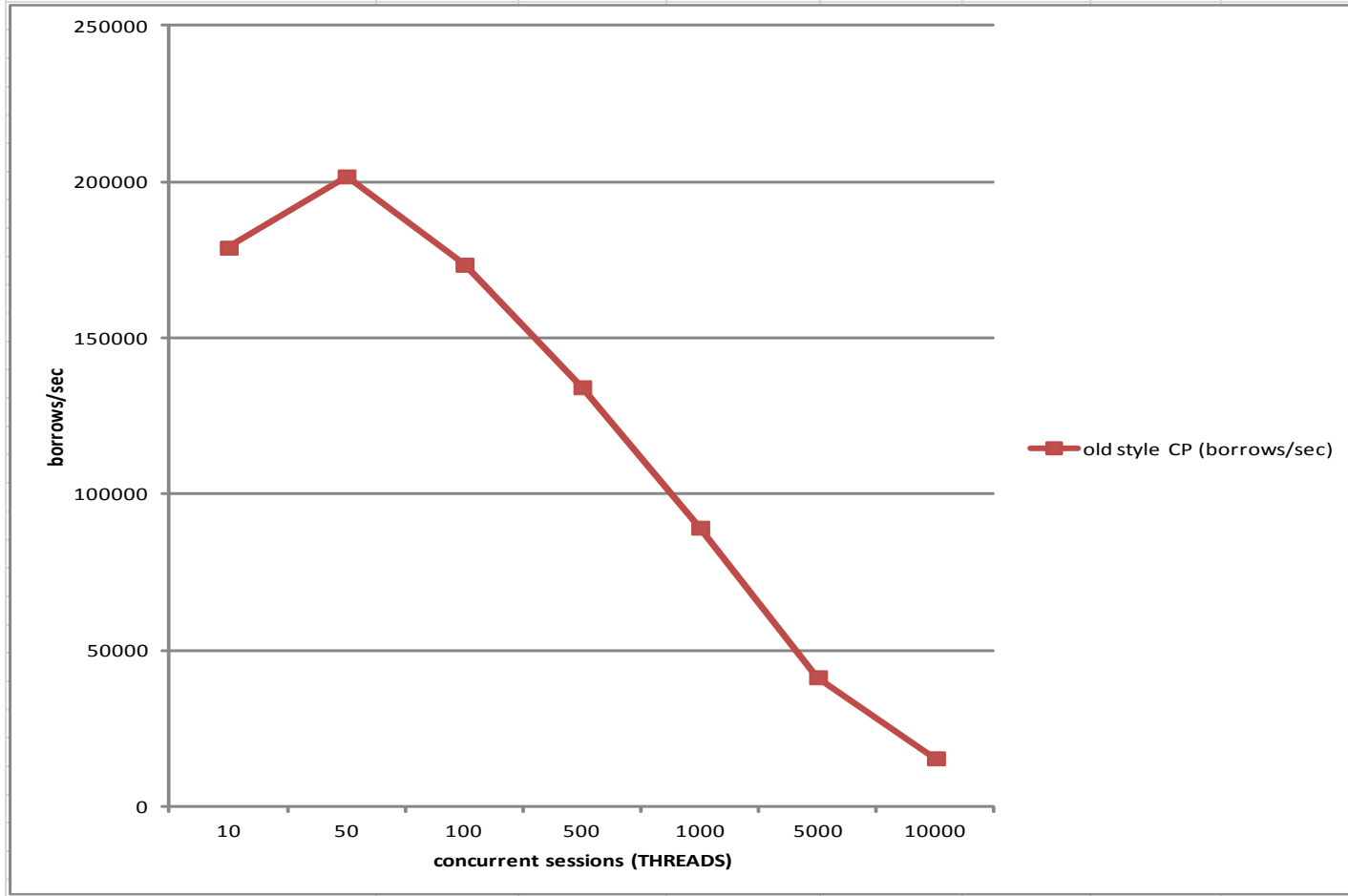
Program Agenda

- 1 Problem
- 2 Requirements
- 3 How it works
- 4 Benchmark
- 5 Conclusion

- 1 Problem
- 2 Requirements
- 3 How it works
- 4 Benchmark
- 5 Conclusion

Non-wait-free connection pool's performance

| concurrent sessions (THREADS) | 10 | 50 | 100 | 500 | 1000 | 5000 | 10000 |
|-------------------------------|--------|--------|--------|--------|-------|-------|-------|
| old style CP (borrows/sec) | 178890 | 201743 | 173464 | 134211 | 89263 | 41362 | 15404 |



The problem

We need a concurrent data structure that does not degrade a performance in case of a large number of accessing threads.

A ***wait-free*** solution is desirable (***lock-free*** if there is guaranteed system-wide progress, and ***wait-free*** if there is also guaranteed per-thread progress).

- 1 Problem
- 2 Requirements**
- 3 How it works
- 4 Benchmark
- 5 Conclusion

What is needed

A concurrent data structure for:

1. Fast element insertion
2. Fast element removal
3. Fast sequential search
4. Fast navigation

Generic example – all 6 are concurrently running threads

- Thread 1 puts an object 'Potato'
- Thread 2 puts an object 'Tomato'
- Thread 3 puts an object 'Cucumber'
- Thread 4 looks for an object 'Potato' with specific color, size, specific shape, etc.
- Thread 5 navigates all vegetables and makes a list of them
- Thread 6 removes all objects 'Tomato' with specific size

Issue!

There is a locking issue here!

Connection attributes

| Attribute | Description |
|-----------------------------------|--|
| Available | Whether it is available or not |
| Valid | Whether valid or not |
| URL | Database |
| Username | Database account username |
| Password | Database account password |
| Origin, 1st dimension | Origin 1 name |
| Origin, 2 nd dimension | Origin 2 name |
| Origin, 3 rd dimension | Origin 3 name |
| Origin, 4 th dimension | Origin 4 name |
| Labels | List of connection's labels, for cost-based search |

Connection attributes (continued)

| Attribute | Description |
|--------------------------|---|
| Creation timestamp | When a connection was created |
| Last borrow timestamp | When a connection was borrowed for the last time |
| Last activity timestamp | When there was a latest operation on a connection |
| Borrow count | Cumulative borrow count |
| Borrow time | Cumulative borrow time |
| Lots of other attributes | |

Actors

| Actor | Description | Typical amount |
|----------------------|---|----------------|
| User session threads | Usually a user thread that borrows a connection | 0-unlimited |
| Inactive timer | Closes inactive connections | 1 |
| Abandoned timer | Makes abandoned connection available again | 1 |
| TTL timer | Makes TTL-ed connections available again | 1 |
| Load balancer thread | Listens to the LB advisory and re-arranges connections among origins | 1 |
| Failover thread | Listens to the FF events and re-arranges connections among instances | 1 |
| Worker threads | Lots of various activities in the background: keeping the pool's min/max bounds, validation, rebalancing, failover handling, etc. | N |

Operations

| Operation | Actor | Description |
|--|---------------------|---|
| Borrow | User Session Thread | Borrow any available valid connection |
| Borrow cheapest labeled | User Session Thread | Borrow a “cheapest” available valid connection for a given set of labels |
| Borrow on a proper instance | User Session Thread | Borrow an available valid connection while a) taking care of the proper load balancing; b) affinity-based borrow |
| Borrow cheapest labeled on a proper instance | User Session Thread | Combination of previous two types of borrow, appropriate instance depends on either load balancing or some type of affinity |

Operations (continued)

| Operation | Actor | Description |
|-------------------------------------|------------------------|---|
| Adjust pool size | Worker Threads | Add new available valid connections if total < minPoolSize, or get rid of some connections if total > maxPoolSize |
| Rearrange among accounts or origins | Worker Threads | Close connections to some database accounts, create connections to some other database accounts |
| Rebalance | Load Balancer Thread | Close some portion of connections on some instances and create some portion on another ones to match the load balancing advisory |
| Failover DOWN Event Processing | Failover Driver Thread | Drain all connections on an instance/host/database that is about to go DOWN, create some connections on other instances/databases/hosts to be within min-max bounds |

Operations (continued)

| Operation | Actor | Description |
|------------------------------|-----------------------------|--|
| Failover UP Event Processing | Failover Driver Thread | Create connection on an instance/host/database that went UP and close connections on other instances/hosts/databases to match min-max bounds |
| Free Abandoned Connections | Abandoned Connections Timer | Navigate through a pool and free (make available) all abandoned connection |
| Free TTL-ed Connections | TTLED Connections Timer | Navigate through a pool and free (make available) all TTL-ed connections |
| Close Inactive Connections | Inactive Connections Timer | Navigate through a pool and close inactive connections |

Possibility of Indexing:

- Some attributes can be indexed
- But hard to index everything (too many keys, partial match, etc.)
- KD-Tree (aka “Multidimensional tree”) – way to go, especially for huge pools; and we plan to implement it!
- We come up to a sequential search, which is not bad once in most cases a proper item is at the beginning of a sequence

```
final List<Item> pool = new ArrayList<Item>();
```

PROS:

- Easy, available and well-known implementation working in tons of applications
- Earlier UCP implementation worked using ArrayList

CONS:

- Every insert, remove and search operation needs to be synchronized

RESULT: not effective for the connection pool.

```
final List<Item> pool = new  
CopyOnWriteArrayList<Item>();
```

PROS:

- Very effective non-blocking reads
- Separate writes (into a copy, then a copy becomes original by just flipping a reference)

CONS:

- Good only for applications that do a lot of reads but rarely write.

RESULT: not effective for the connection pool.

What is 'synchronized'?

Synchronization sample:

```
void onlyMe(Foo f) {  
    synchronized(f) {  
        doSomething();  
    }  
}
```

is compiled into the following Java bytecode:

Method void onlyMe(Foo)

```
0  aload_1           // Push f  
1  dup              // Duplicate it on the stack  
2  astore_2         // Store duplicate in local variable 2  
3  monitorenter     // Enter the monitor associated with f  
4  aload_0          // Holding the monitor, pass this and...  
5  invokevirtual #5 // ...call Example.doSomething()V  
8  aload_2          // Push local variable 2 (f)  
9  monitorexit     // Exit the monitor associated with f  
10 goto 18         // Complete the method normally  
13 astore_3        // In case of any throw, end up here  
14 aload_2         // Push local variable 2 (f)  
15 monitorexit     // Be sure to exit the monitor!  
16 aload_3        // Push thrown exception...  
17 athrow         // ...then rethrow the value to the invoker  
18 return         // Return in the normal case
```

Exception table:

| From | To | Target | Type |
|------|----|--------|------|
| 4 | 10 | 13 | any |
| 13 | 16 | 13 | any |

monitorEnter

`monitorenter` (from the JVM spec):

“Each object has a monitor associated with it. The thread that executes ***monitorenter*** gains ownership of the monitor associated with *objectref*. If another thread already owns the monitor

associated with *objectref*, **the current thread**

waits until the object is unlocked,

then tries again to gain ownership. If the current thread already owns the monitor associated with *objectref*, it increments a counter in the monitor indicating the number of times this thread has entered the monitor. If the monitor associated with *objectref* is not owned by any thread, the current thread becomes the owner of the monitor, setting the entry count of this monitor to 1.”

Atomics

Atomics (like `java.util.concurrent.atomic.AtomicLong`) are implemented as native calls, they are hardware instructions.

Atomics Implementation

```
public native void monitorExit(Object var1);  
public native boolean tryMonitorEnter(Object var1);  
public native void throwException(Throwable var1);  
public final native boolean compareAndSwapObject(Object var1, long var2, Object var4,  
Object var5);  
public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int  
var5);  
public final native boolean compareAndSwapLong(Object var1, long  
var2, long var4, long var6);  
public native Object getObjectVolatile(Object var1, long var2);  
public native void putObjectVolatile(Object var1, long var2, Object var4);
```

Atomics Hardware

- CAS (cmp&swap): when 2 processors access the same memory location, the machine “hangs” a memory for a moment for one processor to operate a memory exclusively (interrupts are waiting)
- Spinlock for SPARC-like architectures

Also....

Let's **skip**, not **wait**, and:

- atomics are all about skipping,
- 'synchronized' are all about waiting

Self-Restriction #1

The wait-free pool never uses any synchronization primitive (including 'synchronized') but just atomic variables

Self-Restriction #2

Bring the number of declared atomics variables to an absolute minimum

Self-Restriction #3

Use only the wait-free pool locking mechanism for the rest of the pool:

The core of Java Connection Pool functionality, highly concurrent logic, including complex mechanisms like fast failover and load balancing, over 17,000 lines of Java code, – not a single ‘synchronized’ keyword

- 1 Problem
- 2 Requirements
- 3 How it works**
- 4 Benchmark
- 5 Conclusion

Empty Pool



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads

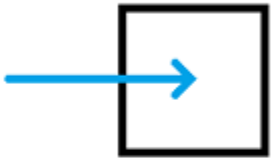


Data In



Data Out

Create An Empty Slot



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads

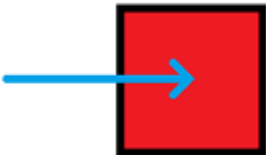


Data In



Data Out

Reserve A Slot



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads

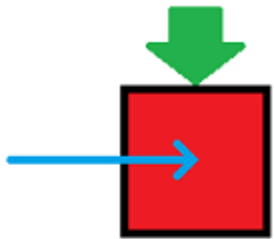


Data In



Data Out

Write A Data



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads

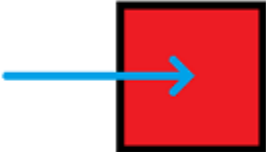


Data In



Data Out

Data Is Written



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads



Data In



Data Out

Release A Slot



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads

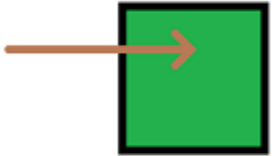


Data In



Data Out

Two Threads Write Data



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads

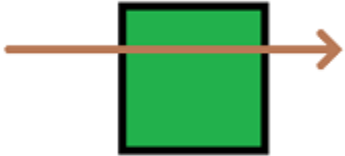


Data In



Data Out

Two Threads Write Data



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads

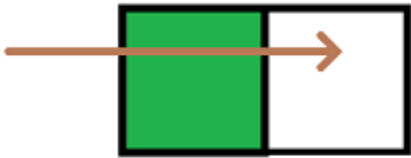


Data In



Data Out

Two Threads Write Data



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads



Data In



Data Out

Two Threads Write Data



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads



Data In



Data Out

Two Threads Write Data



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads



Data In



Data Out



Two Threads Write Data



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads



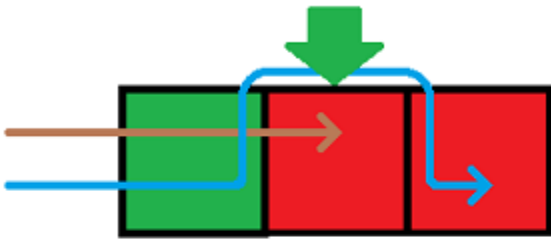
Data In



Data Out



Two Threads Write Data



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads

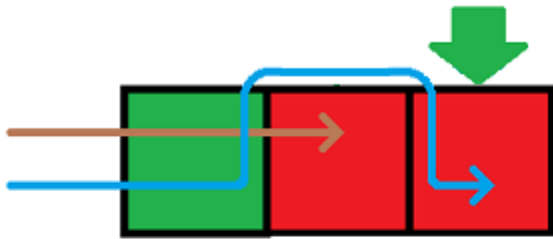


Data In



Data Out

Two Threads Write Data



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads

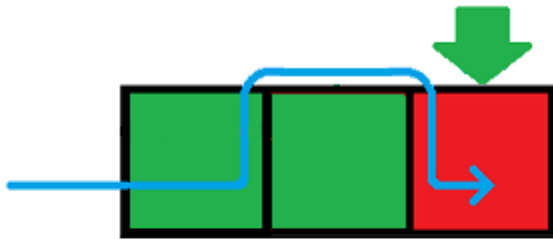


Data In



Data Out

Two Threads Write Data



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads



Data In



Data Out

Two Threads Write Data



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads



Data In



Data Out



Two Threads Write Data



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads

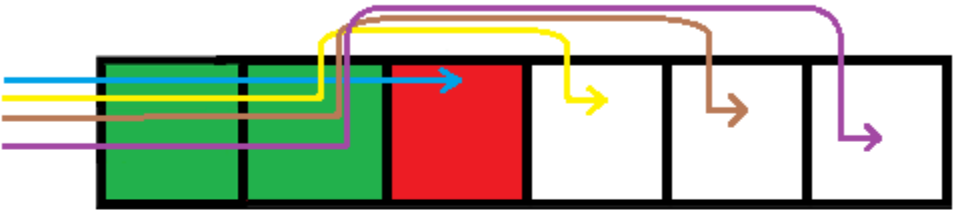


Data In











Data Out

Four Threads: Delete And Write Data











Legend:

-  Empty Slot
-  Reserved Slot
-  Data Slot
-  Threads
-  Threads
-  Threads
-  Data In
-  Data Out

Four Threads: Delete And Write Data











Legend:

-  Empty Slot
-  Reserved Slot
-  Data Slot
-  Threads
-  Threads
-  Threads
-  Data In
-  Data Out

Four Threads: Delete And Write Data









Legend:

-  Empty Slot
-  Reserved Slot
-  Data Slot
-    Threads
-  Data In
-  Data Out

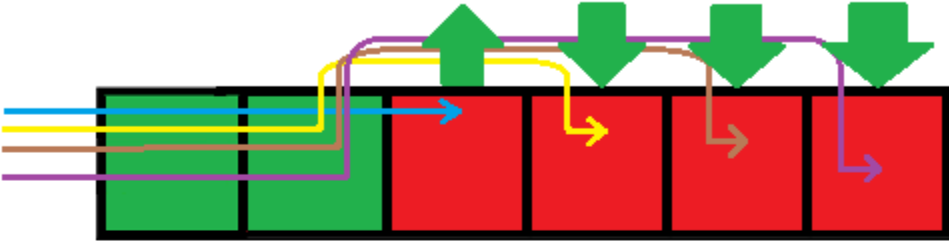
Four Threads: Delete And Write Data



Legend:

-  Empty Slot
-  Reserved Slot
-  Data Slot
-  Threads
-  Data In
-  Data Out

Four Threads: Delete And Write Data



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads



Data In









Data Out

Four Threads: Delete And Write Data



Legend:

-  Empty Slot
-  Reserved Slot
-  Data Slot
-  Threads
-  Data In
-  Data Out

Four Threads: Delete And Write Data



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads

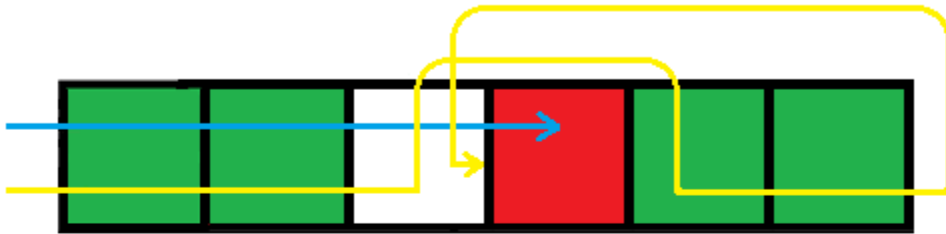


Data In



Data Out

Two Threads: Revisiting



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads



Data In



Data Out



Five Slots Occupied, One Empty



Legend:



Empty Slot



Reserved Slot



Data Slot



Threads



Data In



Data Out

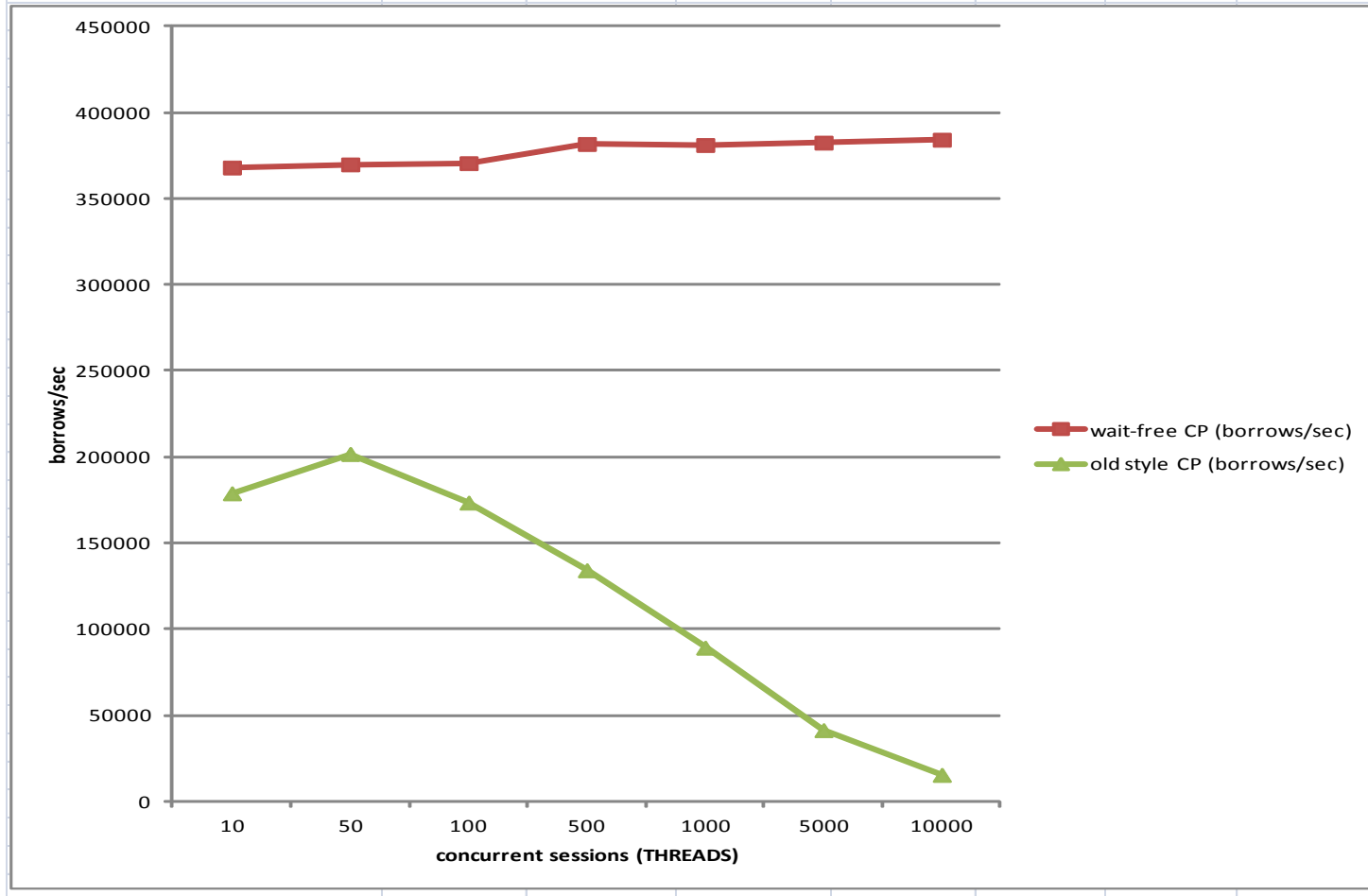
- 1 Problem
- 2 Requirements
- 3 How it works
- 4 Benchmark**
- 5 Conclusion

Benchmark

1. Many threads are running an iterative task concurrently (same task is running for many times within one thread)
2. Only 6 physical connections
3. A task consists of a pool's borrow and a return operations
4. Oracle Enterprise Linux 5 Box with 4 CPUs (I tried 64), 100% CPU utilization
5. UCP 11.2.0.4 or ICC (later will show UCP 12.2)

Comparison Graph

| concurrent sessions (THREADS) | 10 | 50 | 100 | 500 | 1000 | 5000 | 10000 |
|-------------------------------|--------|--------|--------|--------|--------|--------|--------|
| wait-free CP (borrows/sec) | 368066 | 369876 | 370554 | 381744 | 381129 | 382482 | 384204 |
| old style CP (borrows/sec) | 178890 | 201743 | 173464 | 134211 | 89263 | 41362 | 15404 |



- 1 Problem
- 2 Requirements
- 3 How it works
- 4 Benchmark
- 5 Conclusion**

Conclusions

1. It is important not to use critical sections
2. It is important to use as less atomics as possible
3. It is important to use per-slot per-thread individual locking
4. We never wait to visit locked slot, but re-visit instead
5. Wait-free programming is challenging, but pays back